CRM 97-61.10/August 1997

Irreducible Semi-Autonomous Adaptive Combat (ISAAC): An *Artificial-Life* Approach to Land Warfare (U)

Andrew Ilachinski

19990429 070

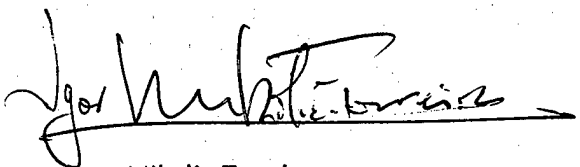
DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

Center for Naval Analyses
4401 Ford Avenue • Alexandria, Virginia 22302-1498

DTIC QUALITY INSPECTED 2

Approved for distribution:

Au



Dr. Igor Mikolic-Torreira
Director, Systems and Tactics Team
Operating Forces Division

This document represents the best opinion of CNA at the time of issue.
It does not necessarily represent the opinion of the Department of the Navy.

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

For copies of this document, call the CNA Document Control and Distribution Section (703)

Copyright © 1997 The CNA Corporation

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1997	Final

4. TITLE AND SUBTITLE
Irreducible Semi-Autonomous Adaptive Combat (ISAAC): An Artificial-Life Approach to Land Warfare

5. FUNDING NUMBERS
C - N00014-96-D-0001

6. AUTHOR(S)
A Ilachinski

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Center for Naval Analyses
4401 Ford Avenue
Alexandria, Virginia 22302-1498

8. PERFORMING ORGANIZATION REPORT NUMBER
CRM 97-61.10

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

This study is a follow-on effort to a recently completed project, sponsored by the Commanding General, Marine Corps Combat Development Command, that assessed the general applicability of the new sciences to land warfare. "New Sciences" is a catch-all phrase that refers to the tools and methodologies used in nonlinear dynamics and complex systems theory to study physical systems that exhibit a "complicated dynamics." CNA is currently developing a multiagent-based simulation of notional combat called ISAAC (Irreducible Semi-Autonomous Adaptive Combat), a preliminary version of which is described in this report. ISAAC takes a bottom-up, synthesist approach to the modeling of combat, vice the more traditional top-down, or reductionist approach.

14. SUBJECT TERMS
artificial intelligence, computerized simulation, ground combat, ISAAC (irreducible semi-autonomous adaptive combat), Lanchester method, land warfare, modeling and simulation (M&S) statistical data, user manuals, war games

15. NUMBER OF PAGES
399

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

"Only when we are able to view *life-as-we-know-it* in the larger context of *life-as-it-could-be* will we really understand the nature of the beast. Artificial Life (AL) is a relatively new field employing a *synthetic* approach to the study of *life-as-it-could-be.* It views life as a property of the *organization* of matter, rather than a property of the matter which is so organized."

– Chris Langton, *Artificial Life* (1989)

Now, substitute the word ***combat*** for ***life*** .....

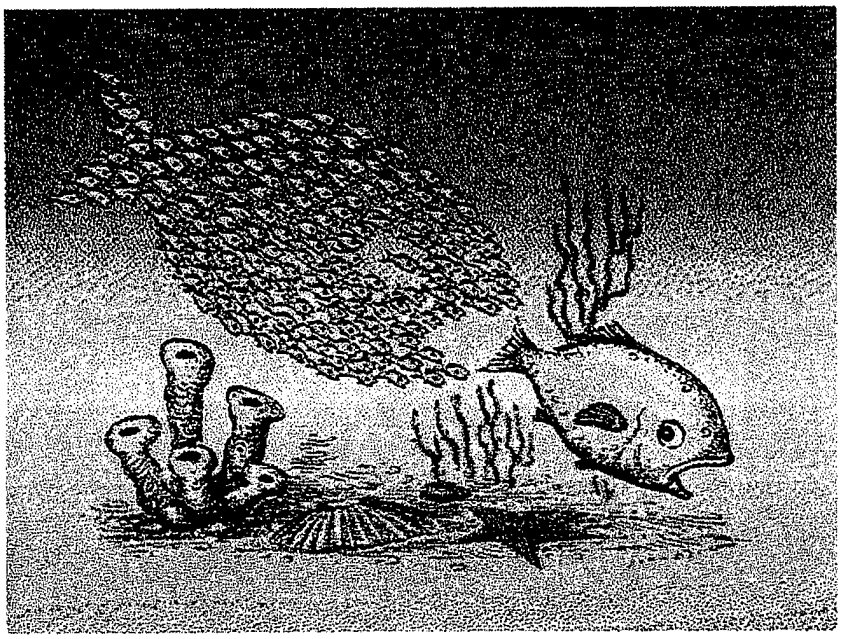


Image from **ftp://parcftp.xerox.com/pub/dynamics/dynamics.html**

"War is ... not the action of a living force upon lifeless mass ... but always the collision of two *living forces.*"

– Carl von Clausewitz, *On War*

Executive Summary

This study is a follow-on effort to a recently completed project, sponsored by the Commanding General, Marine Corps Combat Development Command (MCCDC), that assessed the general applicability of the new sciences to land warfare. "New sciences" is a catch-all phrase that refers to the tools and methodologies used in nonlinear dynamics and complex systems theory to study physical systems that exhibit a "complicated dynamics."

Perhaps the single most important lesson of the new sciences is the observation that the collective decentralized interaction among individual agents obeying local rules often appears locally disordered but induces – on a higher level – a globally ordered pattern of behavior. The central thesis of this report (and developed in earlier reports [1] and [2]) is that the general mechanisms responsible for emerging patterns in complex adaptive systems can be used to further our insight into the patterns of behavior that arise on the real combat battlefield. That is, that *land combat can be modeled as a complex adaptive system.*

As a background to why such an approach might be an important one to take at this time – and how it differs from most current state-of-the-art models of land combat – consider what has been, for the last century, the "conventional wisdom" regarding our fundamental understanding of the basic processes of war.

In 1914, F. W. Lanchester introduced a set of coupled ordinary differential equations – now commonly called the Lanchester Equations (LEs) – as models of attrition in modern warfare [3]. Similar ideas were proposed around that time by Chase [4] and Osipov [5]. The virtue of the LEs, and their intuitive appeal, lies in their shear simplicity. For example, their most basic form consists simply of the statement that one side's attrition rate is proportional to the opposing side's size. The Lanchesterian approach, in general, represents a view of combat in which the driving phenomenon is always force-on-force attrition. This view has served venerably as the conceptual foundation upon which most modern theories of combat attrition are based.

From a fundamental standpoint, however, there are many limitations to using LEs to represent modern combat. Two of the biggest limitations are (1) they do not account for any spatial variation of forces (i.e., no link is established, for example, between movement and attrition) and (2) they completely disregard the human factor in combat (i.e., the psychological and/or decision-making capability of the human combatant).

Therefore, LE-derived models of land warfare are inadequate for assessing advanced warfighting concepts, such as those being explored

by the Marine Corps. In particular, the Lanchesterian view of combat does not adequately represent the Marine Corps' vision of combat: small, highly trained, well-armed autonomous teams working in concert, continually adapting to changing conditions and environments. As an alternative, we suggest that recent developments in complex systems theory – particularly the set of multiagent-based simulation tools developed in the *artificial life* community – provide a new set of tools for addressing land warfare in a fundamentally different way.

To this end, CNA is currently developing a multiagent-based simulation of notional combat called ISAAC (Irreducible Semi-Autonomous Adaptive Combat), a preliminary version of which is described in this report. ISAAC takes a *bottom-up, synthesist* approach to the modeling of combat, vice the more traditional *top-down*, or *reductionist* approach.

Models based on differential equations homogenize the properties of entire populations and ignore the spatial component altogether. Partial differential equations – by introducing a physical space to account for troop movement – fare somewhat better, but still treat the agent population as a continuum. In contrast, ISAAC consists of a discrete heterogeneous set of spatially distributed individual agents (i.e., combatants), each of which has its own characteristic properties and rules of behavior. These properties can also change (i.e., adapt) as an individual agent evolves in time.

The basic element of ISAAC is an ISAAC Agent (or ISAACA), which loosely represents a primitive combat unit (infantryman, tank, transport vehicle, etc.) that is equipped with the following characteristics:

- A default local-rule set specifying how to act in a generic environment (i.e., an embedded "doctrine")
- Goals directing behavior ("mission")
- Sensors generating an internal map of environment ("situational awareness")
- An internal adaptive mechanism to alter behavior and/or rules.

A global rule set determines combat attrition, reconstitution and (in future versions) reinforcement. ISAAC also contains both local and global commanders, each with their own command radii, and obeying an evolving C^2 hierarchy of rules.

Most traditional models focus on looking for equilibrium "solutions" among some set of (pre-defined) aggregate variables. The LEs are effectively *mean-field* equations (in the parlance of physics), in which

certain variables such as attrition rate are assumed to represent an entire force and the outcome of a battle is said to be "understood" when the equilibrium state has been explicitly solved for. In contrast, ISAAC focuses on understanding the kinds of emergent patterns that might arise while the overall system is *out of equilibrium.*

In ISAAC, the "final outcome" of a battle – as defined, say, by measuring the surviving force strengths – takes second stage to exploring how two forces might "co-evolve" during combat. A few examples of the profoundly *non-equilibrium* dynamics that characterizes much of real combat include: the sudden "flash of insight" of a clever commander that changes the course of a battle; the swift flanking maneuver that surprises the enemy; and the serendipitous confluence of several far-separated (and unorchestrated) events that lead to victory. These are the kinds of behavior that Lanchesterian-based models are in principle incapable of addressing. ISAAC represents a first step toward being able to explore such questions.

ISAAC is designed to allow the user to explore the evolving patterns of macroscopic behavior that result from the collective interactions of individual agents, as well as the feedback that these patterns might have on the rules governing the individual agents' behavior. ISAAC can currently be run in three different "modes":

- **Interactive Mode**, in which the user can make 'on-the-fly' changes to the values of any (or all) parameters defining a given run (including the "decision-making personality" of individual ISAACAs). This mode is well suited for playing simple *"What if?"* scenarios and for interactively "searching" for interesting emergent behavior.

- **Data-Collection Mode**, in which the user can (1) generate time series of various changing quantities describing the step-by-step evolution of a battle, and (2) keep track of certain measures of "how well" mission objectives are met at a battle's conclusion. Additionally, the user can generate complete behavioral profiles on two dimensional slices of ISAAC's N-dimensional parameter space.

- **Genetic Algorithm "Evolver" Mode**, in which the user can evolve a local rule set (i.e., "personality") for one side that is "best" suited for performing some well-defined mission against a fixed rule set for the other. This mode illustrates how programs such as this can eventually be used to evolve real-world "tactics" and "strategies."

While this preliminary version of ISAAC can do no more than suggest new ways of thinking about some old issues, it is encouraging to note

that, even at this early juncture, ISAAC already has an impressive repertoire of emergent behaviors:

- Forward advance
- Frontal attack
- Local clustering
- Penetration
- Retreat
- Attack posturing
- Containment
- Flanking Maneuvers
- Defensive posturing
- "Guerilla-like" assaults
- Encirclement of enemy forces
- *many more ...*

Moreover, ISAAC frequently displays behaviors that appear to involve some form of "intelligent" division of red and blue forces to deal with local "firestorms" and skirmishes, particularly those forces whose personalities have been "evolved" (via a *Genetic Algorithm*) to perform a specific mission. It must be remembered that such behaviors are not hard-wired but are effectively an emergent property of a decentralized and nonlinear local dynamics.

ISAAC has been developed primarily to address the basic question: *"To what extent is land combat a self-organized emergent phenomenon?"* As such, its intended use is not as a full system-level model of combat but as an interactive toolbox (or "conceptual playground") in which to explore high-level emergent behaviors arising from various low-level (i.e., individual combatant and squad-level) "interaction rules." The idea behind ISAAC is not to model in detail a specific piece of hardware (M16 rifle, M101 105mm howitzer, *etc.*), but to provide an understanding of the fundamental behavioral tradeoffs involved among a large number of notional variables.

The payoff of using ISAAC, or some other multiagent-based model of land combat, is a radically new – and decidedly non-Lanchesterian – way of looking at some fundamental issues of land combat. Specifically, ISAAC is being designed to help analysts ...

- Understand how all of the different elements of combat fit together in an overall "combat phase space:" *Are there regions that are "sensitive" to small perturbations, and might there be a way to exploit*

this in combat (as in selectively driving an opponent into more sensitive regions of phase space)?

- Understand the out-of-equilibrium patterns of behavior vice the approach to equilibrium states stressed by most conventional models
- Identify and explore emergent collective patterns of behavior on the battlefield
- Assess the value of information: *How can I exploit what I know the enemy does not know about me?*
- Explore tradeoffs between centralized and decentralized command-and-control (C2) structures: *Are some C2 topologies more conducive to information flow and attainment of mission objectives than others? What do the emergent forms of a self-organized C2 topology look like?*
- Provide a natural arena in which to explore consequences of various qualitative characteristics of combat (unit cohesion, morale, leadership, etc.)
- Study the general efficacy of combat doctrine and tactics
- Explore emergent properties and/or other "novel" behaviors arising from low-level rules (even doctrine if it is well encoded)
- Capture universal patterns of combat behavior by focusing on a reduced set of critical drivers
- Suggest likelihood of possible outcomes as a function of initial conditions
- Provide near-real-time tactical decision aids by providing a "natural selection" (via a genetic algorithm) of tactics and/or strategies for a given combat scenario.

ISAAC provides a natural arena in which to explore the Clausewitzian "fog-of-war," or the effects of uncertainties and/or inaccuracies of intelligence data and of time-delays in reporting information. More important, from an *Information Warfare* perspective, ISAAC provides a framework for quantifying the "value" of information on a battlefield. ISAAC can, in principle, be used to explore the consequences of given (personality-defined) force and/or weapon mixes. It can also be used to re-examine traditional measures of combat effectiveness and define requirements for what might loosely be called *nonlinear data collection*, which refers to data that capture the continuously evolving

relationships among all of the interdependent components of combat (as compared with more static measures — such as force attrition — commonly used by conventional models).

The ultimate goal is for ISAAC to become a fully developed complex systems theoretic analyst's toolbox for *identifying, exploring and possibly exploiting emergent collective patterns of behavior on the battlefield.*

Organization of Paper

This paper is organized into seven main sections, each of which is relatively self-contained:

- ***Introduction.*** This section provides a thorough discussion of the background behind and motivation for the artificial-life approach to modeling land warfare, including several examples of decentralized self-organization. It also provides a short general introduction to multiagent-based modeling.

- ***Overview of ISAAC.*** This section provides a detailed overview of the design philosophy and dynamical features of ISAAC. It discusses the overall program flow, introduces ISAACAs and what is meant by an ISAACA "personality," and provides an overview of ISAAC's built-in command and control hierarchy.

- ***User's Guide to ISAAC.*** This section provides a self-contained user's guide to ISAAC. This guide includes step-by-step instructions for loading the program, providing data input, running and interpreting all graphics output, and interacting with the program as it is running. This section also includes a detailed listing of all parameters appearing in ISAAC's input data file.

- ***Sample Runs.*** This section provides a small sampling of ISAAC's "repertoire" of dynamical patterns of behavior by focusing on thirteen sample runs. These runs illustrate such "emergent" behaviors as forward advance, penetration, encirclement, containment, and flanking maneuvers, among many others. Some samples also show the effects of communications and both local and global command structures.

- ***Data Collection.*** This section provides an overview of ISAAC's rudimentary data collection capability (including a time series of remaining force size, distance and cluster-size distributions, and spatial entropy). A discussion is provided on how to use a separate stand-alone program to effectively "map out" ISAAC's

dynamical behavior over two-dimensional slices of ISAAC's (much larger) overall N-dimensional parameter space.

- ***Genetic Algorithm Evolution.*** This section essentially "mirrors" the content of sections 2 through 5 by providing a self-contained discussion of how to use a genetic algorithm to "evolve" ISAACA personalities. It contains an overview of the basic genetic algorithm recipe as it is used by ISAAC, defines "mission fitness," provides a user's guide to the stand-alone program that incorporates this recipe (including a complete listing of the contents of the appropriate input and output data files), and uses several sample runs to illustrate how this stand-alone program can be used to "evolve" personalities to perform a specific mission.

- ***Future Enhancements.*** The final section discusses future design plans and provides a conceptual roadmap for how ISAAC can be used to explore some fundamental issues in land warfare, both in the short and long term. A more speculative discussion is provided centering on ways to use ISAAC to explore self-organized command and control structures and novel "self-organized" filtering of battlefield information.

Additional information is provided in the appendices:

- ***Appendix A*** consists of a short primer on cellular automata, and is useful background reading for those not familiar with this common tool in complex system theory modeling.

- ***Appendix B*** provides a primer on genetic algorithms, and includes a short description of a novel way (first proposed by Hillis [6]) in which a genetic algorithm's ability to "solve" certain problems might be enhanced by pitting one genetic algorithm against another.

- ***Appendix C*** contains a fragment of the ANSI-C source code for ISAAC, including header files, structures, the main function module (in its entirety) and a short description of all other functions appearing in the main module.

- ***Appendix D*** contains a fragment of the ANSI-C source code for ISAAC_GA (i.e., the stand-alone genetic algorithm "evolver"), including header files, structures, and the main function module (in its entirety).

- ***Appendix E*** defines each of the individual data fields appearing in ISAAC's output statistics files. This information can be used to generate desired plots using a stand-alone plotting program.

- ***Appendix F*** provides a brief heuristic description of (and C source code for) the cluster counting algorithm used by ISAAC's data collection module.

- ***Appendix G*** contains sample data input files for ISAAC and the stand-alone genetic algorithm "personality-evolver" program.

ISAAC is very much a "work in progress." This paper, and all accompanying programs and data files, must therefore be viewed as preliminary work only. However, even at this early stage of development, ISAAC shows where the serious user can gain significant insight into the fundamental processes of land combat.

Table of Contents

Introduction **1**
Background 2
Motivation 3
Lanchester equations 5
Artificial Life 7
Decentralized Self-Organization 9
Cellular Automata 9
"Boids" 11
Collective Sorting 12
Agent-Based Models 13
Recent Examples of Agent-Based Simulations 16
Agent-Based Simulations vs. Traditional Modeling Approaches 16
Agent-Based Simulations vs. Traditional Artificial Intelligence 17

Overview of ISAAC **19**
What is ISAAC? 19
What is an ISAACA? 21
Design Philosophy 22
Information Levels 23
Guiding Principles 24
Combat Battlefield 25
Program Flow 26
ISAACA Ranges 27
Sensor Range 27
Fire Range 27
Threshold Range 28
Movement Range 28
Communications Range 28
ISAACA Personalities 29
Personality Weight Vector 29
Squads 31
White Forces 31
ISAACA Move Selection 31
Example 33
Move Sampling Order 34
ISAACA Adaptability 34
Advance Constraint 35
Cluster Constraint 36
Combat Constraint 36
Minimum Local-Distance Constraints 36
ISAACA Combat 37

Defense 38
Reconstitution 39
Fratricide 39
Communication 40
Command and Control 40
Local Command 42
Local Command Area 43
Subordinate ISAACAs 44
Example 45
Global Command 45
GC Command of LC-LC interaction 45
GC Command of Autonomous LC Movement 47
LC Response to GC Commands 49

A Concise User's Guide to ISAAC 51
Hardware Requirements 51
Computer Memory 51
Graphics 51
Installing ISAAC 52
Starting ISAAC 52
Contents of ISAAC's Input Data File 54
General Battle Parameters 56
battle_size 56
init_dist_flag 56
R_box_(l,w) 57
RED_cen_(x,y) 57
B_box_(l,w) 57
BLUE_cen_(x,y) 57
B_flag_(x,y) 57
R_flag_(x,y) 57
termination? 57
move_order? 58
combat_flag? 58
terrain_flag? 58
red_frat_flag? 58
blue_frat_flag? 58
red_frat_rad 58
blue_frat_rad 59
red_frat_prob 59
blue_frat_prob 59
reconst_flag? 59
RED_recon_time 59
BLUE_recon_time 59
Statistics Parameters 59
stat_flag? 60
goal_stat_flag? 60
center_mass_flag? 60

interpoint_flag? 60
entropy_flag? 60
cluster_1_flag? 61
cluster_2_flag? 61
neighbors_flag? 61
Blue Global Command Parameters 61
BLUE_global_flag? 61
GC_fear_index 61
GC_w_alpha 62
GC_w_beta 62
GC_frac_R[1] 62
GC_frac_R[2] 62
GC_w_swath[1] 63
GC_w_swath[2] 63
GC_w_swath[3] 63
GC_max_red_f 63
GC_help_radius 63
GC_h_thresh 63
GC_rel_h_thresh 63
Red Global Command Parameters 64
Blue Local Command Parameters 64
BLUE_local_flag 65
num_BLUE_cmdrs 65
B_patch_type 65
B_patch_flag 65
(n)_B_undr_cmd 65
(n)_B_cmnd_rad 65
(n)_B_SENSOR_rng 66
(n)_w1:alive_B 66
(n)_w2:alive_R 66
(n)_w3:injrd_B 66
(n)_w4:injrd_R 66
(n)_w5:B_goal 66
(n)_w6:R_goal 66
(n)_B_THRS_range 67
(n)_ADVANCE_num 67
(n)_CLUSTER_num 67
(n)_COMBAT_num 67
(n)_B_w_alpha 67
(n)_B_w_beta 67
(n)_B_w_delta 68
(n)_B_w_gamma 68
(n)_w_obey_GC_def 68
(n)_w_help_LC_def 68
Red Local Command Parameters 68
Blue ISAACA Parameters 69

num_blues 69
squads 69
num_per_squad 69
M_range 69
personality 69
w1_a:B_alive_B 71
w2_a:B_alive_R 71
w3_a:B_injrd_B 71
w4_a:B_injrd_R 71
w5_a:B_B_goal 72
w6_a:B_R_goal 72
w1_i:B_alive_B 72
w2_i:B_alive_R 72
w3_i:B_injrd_B 73
w4_i:B_injrd_R 73
w5_i:B_B_goal 73
w6_i:B_R_goal 73
w7:B_loc_comdr 74
w8:B_loc_goal 74
defense_flag 74
alive_strength 74
injured strength 74
S_range 75
F_range 75
COMM_flag 75
COMM_range 75
COMM_weight 75
movement_flag 76
T_range 76
A:ADVANCE_num 76
A:CLUSTER_num 76
A:COMBAT_num 76
I:ADVANCE_num 77
I:CLUSTER_num 77
I:COMBAT_num 77
T_RANGE_(m,M) 77
A:ADV_(m,M) 78
A:CLUS_(m,M) 78
A:COMB_(m,M) 78
I:ADV_(m,M) 79
I:CLUS_(m,M) 79
I:COMB_(m,M) 79
A:B_B_min_dist 80
A:B_R_min_dist 80
A:B_B_goal_min 80
I:B_B_min_dist 80

I:B_R_min_dist 81
I:B_B_goal_min 81
shot_prob 81
B_max_eng_num 81
Red ISAACA Parameters 81
Terrain Parameters 82
(n)_size 82
(n)_center_x 82
(n)_center_y 82
Sample Graphics Display 83
ISAACA Data Regions 85
Squad Identifier 85
Range Parameters 85
Offensive/Defensive Parameters 86
Personality Weight Vector 86
Constraint Parameters 86
Reconstitution 87
Fratricide 87
Attrition 87
"Hot-Key" Menu 88
On-the-Fly Parameter Changes 90
Combat Parameters 92
Red ISAACA Parameters 93
Blue ISAACA Parameters 95
Red Local Command Parameters 95
Blue Local Command Parameters 97
Red Global Command Parameters 97
Blue Global Command Parameters 97
Statistics Calculations 97

Sample Runs 99
Sample Run #1: MISMATCH.out 101
Sample Run #2: FLUID_1.out 105
Sample Run #3: FLUID_2.out 108
Sample Run #4: PRECESS.out 110
Sample Run #5: GOALDEF1.out 113
Sample Run #6: GOALDEF2.out 115
Sample Run #7: CIRCLE.out 117
Sample Run #8: FIRESTM1.out 119
Sample Run #9: FIRESTM2.out 122
Sample Run #10: SENSOR.out 123
Sample Run #11: LOCALCMD.out 126
Sample Run #12: GLBALCMD.out 129
Sample Run #13: BATTLE1.out 132

Data Collection 135
Built-in Statistics 135

Classes of Data 136
Class 1: Force sizes 137
Class 2: ISAACA interpoint distance distributions 137
Class 3: ISAACA neighbor-number distributions 137
Class 4: ISAACA:enemy-flag interpoint distance distributions 137
Class 5: ISAACA cluster-size distributions 137
Class 6: Center-of-mass positions 138
Class 7: Spatial entropy 138
Sample Output 140
Taking 2D "Slices" of ISAAC's Parameter Space 146
The Basic Idea 148
Mission 148
Mission Fitness 148
Pseudo-code 149
Concise User's Guide to ISAAC_PM 149
Starting ISAAC_PM 149
Contents of ISAAC_PM's Data Input File: PHASE.dat 152
Contents of ISAAC_PM's Data Output File: PHASEOUT.dat 153
Sample Graphics Display 154
Sample Output 157
Sample #1: Forward Advance 157
Sample #2: Red Offense 159
Sample #3: Red Defense 160

Genetic Algorithm Evolutions of ISAACA Personalities 163
Genetic Algorithms : Brief Overview 164
The Basic GA Recipe 165
Genetic Algorithms : Adapted to ISAAC 166
Personality Chromosome 167
Mission Objective 169
Mission primitive m_1 171
Mission primitive m_2 172
Mission primitive m_3 172
Mission primitive m_4 172
Mission primitive m_5 173
Mission primitive m_6 174
Mission primitive m_7 175
Mission primitive m_8 175
Mission primitive m_9 176
Mission primitive m_{10} 176
ISAAC_GA's GA Recipe 177
Concise User's Guide to ISAAC_GA 178
Starting ISAAC_GA 178

Contents of ISAAC_GA's Data Input File: GA_DATA ... 179
num_generations ... 180
num_initial_conds ... 180
max_time_to_goal ... 180
penalty_power ... 180
best_personalities_to_file? ... 181
min_dist_genes_flag ... 181
initial_condition_genes_flag ... 181
w1_time_to_goal ... 181
w2_friendly_loss ... 181
w3_enemy_loss ... 181
w4_red_to_blue_survival_ratio ... 182
w5_friendly_CM_to_enemy_flag ... 182
w6_enemy_CM_to_friendly_flag ... 182
w7_friendly_near_enemy_flag ... 182
w8_enemy_near_friendly_flag ... 182
w9_red_fratricide_hits ... 182
w10_blue_fratricide_hits ... 182
termination_code? ... 182
flag_containment_range ... 183
containment_number ... 183
red_CM_to_BF_frac ... 183
ISAACA Chromosome Entries: gene[i] ... 183
Contents of ISAAC_GA's Data Output Files ... 186
GA_STAT.dat ... 186
Sample Graphics Display ... 188
Fitness Summary ... 188
"Hot-Key" Menu ... 189
Sample Runs ... 192
Typical Run-Times ... 192
Typical Learning Curves ... 192
Sample GA Run #1 ... 194
Sample GA Run #2 ... 196

Future Enhancements to ISAAC ... 203
Basic Enhancements to the "Core Engine" ... 204
More Realistic ISAACA State-Space ... 204
Enhanced Offensive and Defensive Capabilities ... 205
An Enhanced Command and Control Structure ... 206
Enhanced Personality "Value-Systems" ... 208
Generalized Personality Matrix ... 208
Hostility Rings ... 209
Greater "Depth" to, and Variety of, Local Moves ... 210
Added Environmental Realism ... 210
Enhanced Combat Adjudication ... 211
Targeting Strategies ... 212
Memory and Learning ... 212

Memory 213
Neural-Network-Derived Move Selection 214
Reinforcement Learning 214
Nested ISAAC Dynamics 216
Data Collection Enhancements 217
Trajectory-Difference Measures 218
Combat Entropy 218
Activity Maps 220
Enhancements to GA Evolution 221

What Is ISAAC Useful For? 225
How is Work in the "New Sciences" Actually Done? 228
Sample Issues 230
Centralized Versus Decentralized Command and Control Structure 230
A Self-Organized C^2 Structure? 231
The "Human Element" of Combat Modeling 232
Combat "State- Space" 233
Miscellaneous Issues and Questions 235
Selfishness vs Altruism 235
Kauffman's Patch-Optimization Procedure 236
Self-Organized Criticality in Combat 236
Scaling Problem 239
Self-Organized Information 239

Epilogue: On the Use of Simulations 243

Appendix A: A Brief Primer on Cellular Automata 245
Example #1: One-dimensional CA 246
Example #2: Conway's Life 248
Example #3: Lattice Gases 250
Example #4: Collective Behavior in Higher Dimensions 251

Appendix B: A Brief Primer on Genetic Algorithms 253
Genetic Operators 254
The Basic GA Recipe 255
Example: Function Maximization 255
The Fitness Landscape 256
How Do GAs Work? 258
The Building-Block Hypothesis 259
Dueling Parasites 260

Appendix C: Source Code for ISAAC 263
Header File 263
Structures 264
Main Module 272
Function Modules 299

Appendix D: Source Code for ISAAC_GA 303

Header File 303
Structures 303
Main Module 308

Appendix E: STATS_X.dat Data Fields 335
STATS_1.dat 335
STATS_2.dat 335
STATS_3.dat 336
STATS_4.dat 336
STATS_5.dat 336
STATS_6.dat 337
STATS_7.dat 337
STATS_8.dat 338
STATS_9.dat 338
STATS_10.dat 338
STATS_11.dat 339
STATS_12.dat 340
STATS_13.dat 340
STATS_14.dat 341
STATS_15.dat 342
STATS_16.dat 342
STATS_17.dat 343
STATS_18.dat 344
STATS_19.dat 345
STATS_20.dat 346
STATS_21.dat 347

Appendix F: Cluster Counting Algorithm 349
Heuristic Recipe 349
Source Code 351

Appendix G: Sample Input Data Files 355
Sample Data Input File for ISAAC_CE: ISAAC.dat 355
Sample Data Input Files for ISAAC_GA 359
GA_ISAAC.dat 359
GA_DATA.dat 362
Sample Data Input File for ISAAC_PM: PHASE.dat 364

References 365

List of Figures 371

List of Tables 375

Introduction

This study is a follow-on effort to a recently completed project, sponsored by the Commanding General, Marine Corps Combat Development Command (MCCDC), that assessed the general applicability of the "new sciences" to land warfare. "New sciences" is a catch-all phrase that refers to the tools and methodologies used in nonlinear dynamics and complex systems theory to study physical dynamical systems that exhibit a "complicated dynamics."

The final reports ([1] and [2]) for that assessment provide a broad-brush overview of the applicability of nonlinear dynamics and complex systems theory to land warfare. Reference [1] is a general technical source book of information on the key ideas, concepts and methodologies of nonlinear dynamics and complex systems theory, and contains an extensive glossary of terms. Reference [2] discusses specific "new sciences" ideas that can potentially add insight into our conventional understanding of land warfare.

The central thesis of both these reports is that land combat can be thought of as *a complex adaptive system.* That is to say, land combat is a nonlinear dynamical system composed of many interacting semi-autonomous and hierarchically organized agents continuously adapting to a changing environment.

Military conflicts, particularly land combat, have all of the key features of complex adaptive systems (see table 1): combat forces are composed of large numbers of nonlinearly interacting parts and are organized in a command and control hierarchy; local action, which often appears disordered, induces long-range order (i.e., combat is self-organized); military conflicts, by their nature, proceed far from equilibrium; military forces, in order to survive, must continually adapt to a changing combat environment; there is no master "voice" that dictates the actions of each and every combatant (i.e., battlefield action effectively proceeds according to a decentralized control); and so on. This means that, in principle, land combat ought to be amenable to precisely the same methodological course of study as any other complex adaptive system, such as the stock market, a natural ecology, or the human brain.

Implicit in this central thesis is the idea that these largely conceptual links between properties of land warfare and properties of complex systems in general can be extended to forge a set of practical connections as well. Land warfare does not just look like a complex system on paper, but can well be characterized in practice using the same basic principles that are used for discovering and identifying behaviors in complex systems.

The program described in this report, called ISAAC, is a multiagent-based simulation of notional combat (an early version of which is described in [7]). ISAAC represents a first step toward developing a complex systems theoretic analyst's toolbox for identifying, exploring, and possibly exploiting emergent collective patterns of behavior on the real battlefield.

Table 1. Land combat as a complex adaptive system

General Property of Complex Systems	Description of Relevance to Land Combat
Nonlinear interaction	Combat forces composed of a large number of nonlinearly interacting parts; sources include feedback loops in C2 hierarchy, interpretation of (and adaptation to) enemy actions, decision-making process, and elements of chance
Nonreductionist	The fighting ability" of a combat force cannot be understood as a simple aggregate function of the fighting ability of individual combatants
Emergent Behavior	The global patterns of behavior on the combat battlefield unfold, or emerge, out of nested sequences of local interaction rules and doctrine
Hierarchical structure	Combat forces are typically organized in a command and control (fractal-like) hierarchy
Decentralized control	There is no master "oracle" dictating the actions of each and every combatant; the course of a battle is ultimately dictated by local decisions made by each combatant
Self-organization	Local action, which often appears "chaotic," induces long-range order
Nonequilibrium order	Military conflicts, by their nature, proceed far from equilibrium; understanding how combat unfolds is more important that knowing the "end state"
Adaptation	In order to survive, combat forces must continually adapt to a changing environment, and continually look for better ways of adapting to the adaptation pattern of their enemy
Collectivist dynamics	There is a continual feedback between the behavior of (low-level) combatants and the (high-level) command structure

Background

Reference [2] introduces a convenient eight-tier scaffolding on which to organize potential applications of nonlinear dynamics and complex systems theory to land warfare. This scaffolding includes applications that range roughly from those (on Tier I) that involve the least risk but are likely to incur the least payoff, to those (on Tier VIII) that involve the greatest risk but are also likely to incur the greatest potential payoff (see table 2):[1]

[1] See http://www.marine-ns.cots-q.com/second~1/nsappl~1/eightt~1/eightt.htm.

- **Tier I:** *General Metaphors for Complexity in War*
- **Tier II:** *Policy and General Guidelines for Strategy*
- **Tier III:** *Conventional Warfare Models and Approaches*
- **Tier IV:** *Description of the Complexity of Combat*
- **Tier V:** *Combat Technology Enhancement*
- **Tier VI:** *Combat Aids for the Battlefield*
- **Tier VII:** *Synthetic Combat Environments*
- **Tier VIII:** *Original Conceptualizations of Combat*.

For many obvious reasons, the most natural application of complexity theory to land warfare is to provide an *agent-based simulation of combat.* That is to say, a Tier-VII application that attempts to model land combat as a co-evolving ecology of local-rule-based autonomous adaptive agents (see shaded box in table 2).

Agent-based simulations of complex adaptive systems are predicated on the idea that the global behavior of a complex system derives entirely from the low-level interactions among its constituent agents. By relating an individual constituent of a complex adaptive system to an agent, one can simulate a real system by an artificial world populated by interacting processes. Agent-based simulations are particularly adept at representing real-world systems composed of individuals that have a large space of complex decisions and/or behaviors to choose from.

Motivation

It is a bit ironic that in this modern age of distributed interactive simulations and gigabyte-sized code driving networked 3D virtual-reality systems with embedded artificial intelligence, the underlying principles of combat attrition calculations in land warfare models remain largely unchanged since the turn of the century. Most attrition models still depend on some form of Lanchester's equations (see below), even in contexts that are wholly inappropriate for their use.

This study was motivated by two fundamental insights:

1. The fundamental principles underlying modern land warfare – with its general emphasis on maneuver and adaptation – ***cannot be elucidated from the (reductionist-style) analysis of force-on-force attrition alone.***

2. The main lesson from complex systems theory – namely, that local nonlinear interaction of many "simple parts" often results in "apparently complex" emergent behavior – can be used to develop a ***radically new synthesist approach to understanding the fundamental processes of war.***

ISAAC was designed to take a new look at a very old problem by exploiting what the "new sciences" have taught us about how global patterns often emerge from the collective behaviors of individuals. *What higher-level phenomena might emerge on the real battlefield out of the collective interactions among individual combatants?*

Table 2. Eight tiers of applicability

Tier of Applicability	Description	Examples
I. General Metaphors for Complexity in War	Build and continue to expand base of images to enhance conceptual links between complexity and warfare	nonlinear vice linear synthesist vice analytical edge-of-chaos vice equilibrium process vice structure holistic vice reductionist
II. Policy and General Guidelines for Strategy	Guide formulation of policy and apply basic principles and metaphors of CST[1] to enhance and/or alter organizational structure	Use general metaphors and lessons learned from CST to guide and shape policy making; Use genetic algorithms to evolve new organizational forms
III. "Conventional" Warfare Models and Approaches	Apply tools and methodologies of CST to better understand and/or extend existing models	chaos in Lanchester equations chaos in arms-race models analogy with ecological models
IV. Description of the Complexity of Combat	Describe real-world combat from a CST perspective	power-law scaling Lyapunov exponents entropic parameters
V. Combat Technology Enhancement	Apply tools and methodologies of CST to certain limited aspects of combat, such as intelligent manufacturing, cryptography and data dissemination	intelligent manufacturing data compression cryptography IFF computer viruses fire ants
VI. Combat Aids	Use CST tools to enhance real-world combat operations	autonomous robotic devices tactical picture agents tactics/strategy evolution via GA
VII. Synthetic Combat Environments	Full system models for training and/or to use as research "laboratories"	agent-based models (*SimCity*) Soar/IFOR SWARM
VIII. Original Conceptualizations of Combat	Use CST-inspired basic research to develop fundamentally new conceptualizations of combat	pattern recognition controlling/exploiting Chaos Universality?

[1] CST = Complex Systems Theory

Lanchester equations

In 1914, F. W. Lanchester [3] introduced a set of coupled ordinary differential equations – now commonly called the Lanchester Equations (LEs) – as models of attrition in modern warfare.[2] These equations have since venerably served as the fundamental models upon which most modern theories of combat attrition are based. For the simplest case of directed fire, for example, Lanchester hypothesized that one side's casualty rate is proportional to the number of the opposing side's unit strength.

In mathematical terms, let R(t) and B(t) represent the numerical strengths of the *red* and *blue* forces at time t, respectively, and α_R and α_B represent the constant effective firing rates at which one unit of strength on one side causes attrition of the other side's forces. Then Lanchester's well known *directed fire* (or *square law*) model of attrition is given by

$$\begin{cases} \frac{dR}{dt} = -\alpha_B B(t), & R(0) = R_0 \\ \frac{dB}{dt} = -\alpha_R R(t), & B(0) = B_0 \end{cases},$$

where R_0 and B_0 are the initial red and blue force levels, respectively. The closed form solution of these equations is given in terms of hyperbolic functions as

$$\begin{cases} R(t) = R_0 \cosh\left(\sqrt{\alpha_B \alpha_R}t\right) - B_0\sqrt{\alpha_B/\alpha_R}\,\sinh\left(\sqrt{\alpha_B \alpha_R}t\right) \\ B(t) = B_0 \cosh\left(\sqrt{\alpha_B \alpha_R}t\right) - R_0\sqrt{\alpha_B/\alpha_R}\,\sinh\left(\sqrt{\alpha_B \alpha_R}t\right) \end{cases},$$

and satisfies the simple "square-law" state equation

$$\alpha_R\left[R_o^2 - R(t)^2\right] = \alpha_B\left[B_o^2 - B(t)^2\right].$$

Despite the simplicity of this equation (which can be found embedded in many large-scale combat models), almost all attempts to correlate LE-based models with historical combat data have proven inconclusive, a result that is in no small part due to the paucity of data. Most data consist only of initial force levels and casualties, and typically for one side only. Moreover, the actual number of casualties is usually uncertain because the definition of "casualty" varies (killed, killed + wounded, killed + missing, etc).

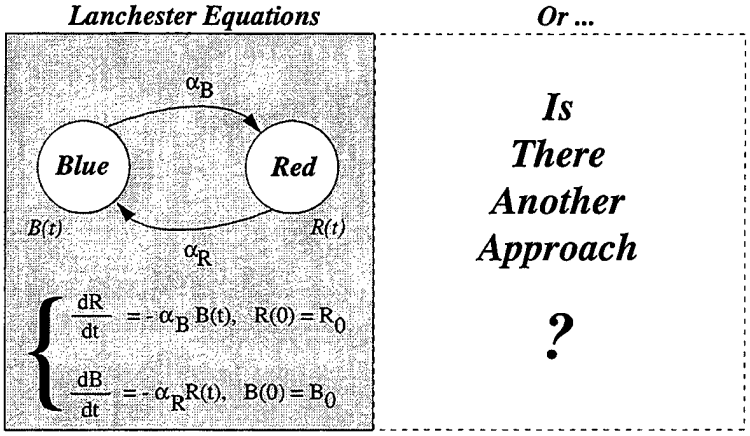
Two noteworthy battles for which detailed daily attrition data and daily force levels do exist are the battle of Iwo Jima in World War II and Inchon-Seoul campaign in the Korean War. While the battle of Iwo Jima

[2] Similar ideas were proposed around that time by Chase [4] and Osipov [5].

is frequently cited as evidence for the efficacy of the classic LEs, it must be remembered that the conditions under which it was fought were very close to the ideal list of assumptions under which the LEs themselves are derived. A detailed analysis of the Inchon-Seoul campaign [8] has also proved inconclusive. Weiss [9], Fain [10] and others describe analyses of battles fought from 200 B.C. to World War II.

While LEs capture some important basic elements of combat, they apply only under a very strict set of assumptions. These assumptions include having homogeneous forces that are continually engaged in combat, firing rates that are independent of opposing force levels and are constant in time, and units that are always aware of the position and condition of all opposing units, among many others. Because Lanchester's direct-fire equations assume that each side has perfect information about where the opposing side's forces are located and what opposing force units have been hit, they are models of highly organized combat with complete and instantaneous information.

Figure 1. The force-on-force attrition "challenge"



LEs suffer from other fundamental shortcomings, including modeling combat as a deterministic process, requiring knowledge of "attrition-rate coefficients" (the values of which are, in practice, very difficult if not impossible to obtain), an inability to account for any suppressive effects of weapons, an inability to account for terrain effects, and the inability to account for any spatial variation of forces. Generally speaking, Lanchester's equations simply lack the spatial degrees-of-freedom needed to model real-world combat. More important, they also leave out the all-important human factor; i.e., the psychological and/or decision-making capability of the human operator.

When these shortcomings are coupled with the Marine Corps' *Maneuver-Warfare* land combat doctrine – which is fundamentally based

on the art of maneuver and adaptation vice pure force-on-force attrition [11] – the use of a purely force-on-force-driven analytical methodology to describe modern combat begins not just to strain credibility, but to literally smack of an oxymoron. The question is, *"Is there anything better?"* Is there a way, perhaps that bucks the conventional way of representing land combat? See figure 1.

While there have been many extensions to and generalizations of Lanchester's equations over the years – including their reformulations as stochastic differential equations and partial differential equations designed to minimize the inherent deficiencies – very little has really changed in the way we fundamentally view and model combat attrition. However, recent developments in nonlinear dynamics and complex systems theory – particularly those in an emerging new field called *artificial life* (see below) – provide a potentially powerful new set of theoretical and practical tools to address many of the deficiencies mentioned above. These developments provide a fundamentally new way of looking at land combat.

Artificial Life

Artificial Life (AL), introduced in the quote that appears on the first page of this report by Chris Langton,[3] is an attempt to understand *life as it is* by examining a larger context of *life as it could be.* The underlying supposition is that life owes at least as much to its existence to the way in which information is *organized* as it does to the physical substance (i.e., matter) that embodies that information. Similarly, ISAAC is designed to be a tool that can help us understand *combat as it is* by allowing analysts to explore a larger context of *combat as it could be.*

The fundamental concept of AL is *emergence,* or the appearance of higher-level properties and behaviors of a system that – while obviously originating from the collective dynamics of that system's components – are neither to be found in nor are directly deducible from the lower-level properties of that system. Emergent properties are properties of the "whole" that are not possessed by any of the individual parts making up that whole: an air molecule is not a tornado and a neuron is not conscious.

AL thus studies life by using artificial components (such as computer programs) to capture the behavioral essence of living systems. The supposition is that if the artificial parts are organized correctly, in a way that respects the organization of the living system, then the artificial system will exhibit the same characteristic dynamical behavior as the natural system on higher levels as well. Notice that this *bottom-up, synthesist* approach stands in marked contrast to more conventional *top-down, analytical* approaches.

[3] Chris Langton organized the first international conference on AL in 1987 [12] and is currently among AL's conceptual leaders.

AL-based computer simulations are characterized by these five general properties [12]:

1. They are defined by populations of simple programs or instructions about how individual parts all interact.
2. There is no single "master oracle" program that directs the action of all other programs.
3. Each program defines how simple entities respond to their environment locally.
4. There are no rules that direct the global behavior.
5. Behaviors on levels higher than individual programs are emergent.

Figure 2. *Artificial Life:* a new approach to the classic force-on-force attrition problem?

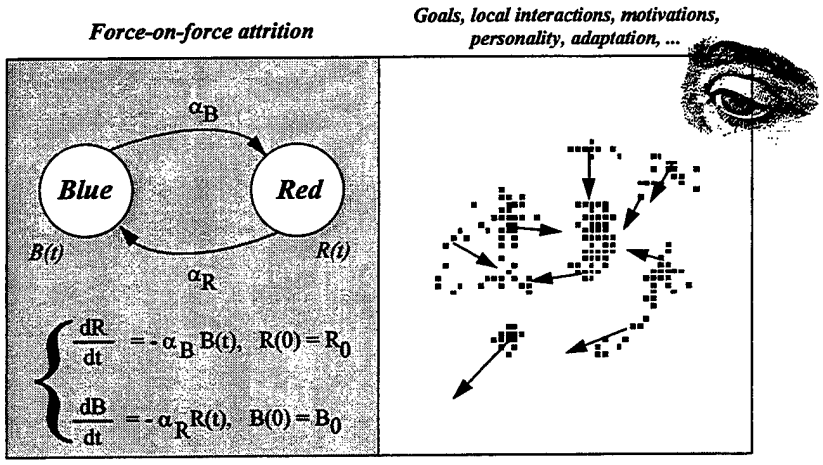


Figure 2 shows, schematically, the form of an artificial-life-based "solution" to the force-on-force attrition challenge posed in figure 1. The eye in figure 2 symbolizes the importance of *emergence* in AL models, and therefore the need for pattern recognition.

Fundamentally, the AL-based approach represents a shift in focus ***from*** "hard-wiring" into a model a sufficient number of (both low- and high-level) details of a system to yield a desired set of "realistic" behaviors – the rallying cry of such models being *"more detail, more detail, we need more detail!"*

to....

Looking for universal patterns of high-level behavior that naturally and spontaneously emerge from an underlying set of low-level interactions and constraints – the rallying cry in this case being *"allow evolving global patterns to emerge on their own from the local rules!"*

Decentralized Self-Organization

A basic property of many complex systems is *decentralized self-organization.* Not every pattern must be centrally orchestrated. In fact, decentralized systems, whose components ***do not*** simply follow rules that are passed down echelon from some higher "authority" possessed of a more global perspective (and instead assimilate and react only to local information), often display a self-organized order on the macro-scale.

To motivate the discussion of ISAAC, we present three simple examples of decentralized self-organization: (1) the space-time patterns of cellular automata, (2) the flocking of birds, and (3) a collective sorting program based on the foraging patterns of ants. Other examples include the colorful dynamical spirals of the Belousov-Zhabotinski chemical reaction, the functioning of the human immune system, and patterns of traffic flow that arise from purely local interactions among individual cars (see [1]). The program that is the subject of this report, ISAAC, was essentially developed to address this basic question: *"To what extent is combat a self-organized emergent phenomenon?"*

Cellular Automata

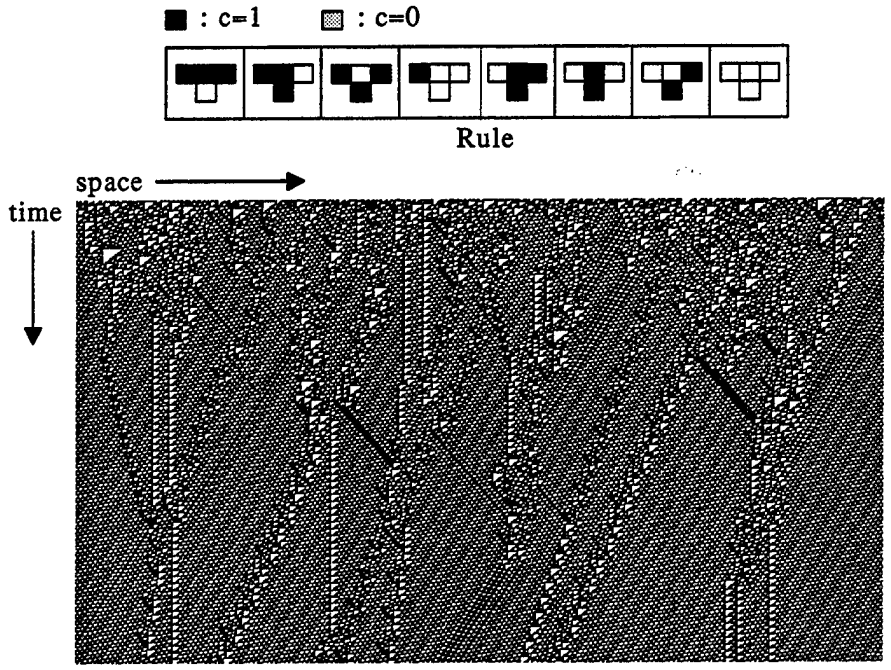
One-dimensional cellular automata are simple discrete dynamical systems consisting of a single "line" of cells.[4] Each cell has a "value" (either *one* or *zero*) that changes in discrete time steps depending on what that cell's value was on the previous time step and what values that given cell's left- and right-neighbors had on the previous time step. A cellular automata *rule* is an explicit prescription of this functional dependency. A *space-time diagram* for this rule consists of stacking the entire row of cells at successive time steps on top of one another (with the color black representing cell value 1, and the color white representing cell value 0).

To illustrate how even "simple rules" acting on the micro-scale can give rise to "complicated dynamics" on the macro-scale, consider the one-dimensional cellular automaton rule defined at the top of figure 3. Its space-time evolution, starting from a random initial state, is shown at the bottom of the figure. Note that this space-time pattern can be described on two different levels: either on the cell-level, by explicitly reading off the values of the individual cells, or on a higher-level by

[4] A short primer on cellular automata is given in Appendix A.

describing it as a sea of particle-like structures superimposed on a periodic background. In fact, following a small initial transient period, temporal sections of this space-time pattern are always of the form "...BBBBPBB...BB... BBBP'BB...BBBP''BBB...", where "B" is a state of the periodic background consisting of repetitions of the sequence "10011011111000" (with spatial period 14 and temporal period 7), and the **P**'s represent "particles." The particle pattern P = "11111000", for example, repeats every four steps while being displaced two cells to the left; the particle P = "11101011000" repeats every ten steps while being displaced two cells to the right.

Figure 3. Evolution of a one-dimensional CA starting from a random initial state



Although the underlying dynamics describing this system is very simple, and entirely deterministic, there is an enormous variety, and complexity, of emergent particle-particle interactions. Such simple systems are powerful reminders that complex higher-level dynamics need not have a complex underlying origin. Indeed, suppose that we had been shown such a space-time pattern but were told *nothing whatsoever about its origin.* How would we make sense of its dynamics? Perhaps the only reasonable course of action would be to follow the lead of any good experimental particle-physicist and begin cataloging the various possible particle states and interactions: *there are N particles of size s moving to the left with speed v; when a particle p of type P collides with q of type Q, the result is the set of particles* $\{p_1, ..., p_n\}$; and so on. It would take a tremendous leap of intuition to fathom the utter simplicity of the real dynamics.

"Boids"

One of the most breathtakingly beautiful displays of nature is the synchronous, fluid-like flocking of birds. It is also an excellent example of emergence in complex systems. Large or small, the magic of flocks is the very strong impression they convey of some intentional centralized control directing the overall traffic. Though ornithologists still do not have a complete explanation for this phenomenon, evidence strongly suggests that flocking is a decentralized activity, where each bird acts according to its local perceptions of what nearby birds are doing. Flocking is therefore a group behavior that emerges from collective action.

Craig Reynolds [13] programmed a set of artificial birds — which he called *boids* — to follow three simple local rules:

- ***Rule 1:*** *maintain a minimum distance from other objects (including other boids)*
- ***Rule 2:*** *match the velocity of nearby boids*
- ***Rule 3:*** *move toward the perceived center of nearby boids.*

Each boid thus "sees" only what its neighbors are doing and acts accordingly. Reynolds found that the collective motion of all the boids was remarkably close to real flocking, despite the fact that there is nothing explicitly describing the flock as a whole. The boids initially move rapidly together to form a flock. The boids at the edges either slow down or speed up to maintain the flock's integrity. If the path bends or zigzags in any way, the boids all make whatever minute adjustments need to be made to maintain the group structure. If the path is strewn with obstacles, the boids flock around whatever is in their way naturally, sometimes temporarily splitting up to pass an obstacle before reassembling beyond it. There is no central command that dictates this action.

Reynolds' *Boids* is a good example of decentralized order not because the boids' behavior is a perfect replica of the flocking of birds that occurs in nature — although it is a close enough match that Reynold's model has attracted the attention of professional ornithologists — but that much of the boids' collective behavior is entirely unanticipated, and cannot be easily derived from the rules defining what each individual boid does.

In the same way, the boid-like program described in this paper can be used to show that certain aspects of land combat can be viewed as *emergent* phenomena resulting from the collective, nonlinear, decentralized interactions among elementary "combatants."

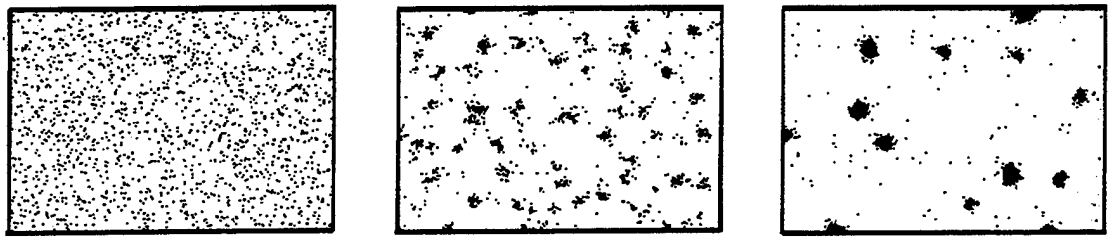
Collective Sorting

Deneubourg, et al. [14], have introduced a simple distributed sorting algorithm that is inspired by the self-organized way in which ant colonies sort their brood.

Implemented by robot teams, their algorithm has the robots move about a fenced-in environment that is randomly littered with objects that can be scooped up. These robots (1) move randomly, (2) do not communicate with each other, (3) can perceive only those objects directly in front of them (but can distinguish between two or more types of objects with some degree of error), and (4) do not obey any centralized control. The probability that a robot picks up or puts down an object is a function of the number of the same objects that it has encountered in the past.

Coordinated by the positive feedback these simple rules induce between robots and their environment, the result, over time, is a seemingly intelligent, coordinated sorting activity. Clusters of randomly distributed objects spontaneously and quite naturally emerge out of a simple set of autonomous local actions having nothing at all to do with clustering per se; see figure 4.

Figure 4. Collective sorting by ant-like robots



The authors suggest that this system's simplicity, flexibility, error tolerance, and reliability compensate for their lower efficiency. While one can argue that this collective sorting algorithm is much less efficient than a hierarchical one, the cost of having a hierarchy is that the sorting would no longer be ant-like but would require a god-like oracle analyzing how many objects of what type are where, deciding how best to communicate strategy to the ants. Furthermore, the ants would require some sort of internal map, a rudimentary intelligence to deal with fluctuations and surprises in the environment (what if an object was not where the oracle said it would be?), and so on. In short, a hierarchy, while potentially more efficient, would of necessity have to be considerably more complex as well. The point Deneubourg, et al. are making is that a much simpler collective decentralized system can lead to seemingly intelligent behavior while being more flexible, more tolerant of errors, and more reliable than a hierarchical system.

Agent-Based Models

Agent-based simulations of complex adaptive systems are predicated on the idea that the global behavior of a complex system derives entirely from the low-level interactions among its constituent agents. Lessons about the real-world system that an agent-based simulation is designed to model can be learned by looking at the emergent structures induced by the interaction processes taking place within the simulation.

The purpose behind building agent-based simulations is twofold: it is to learn both the *quantitative* and *qualitative* properties of the real system. Agent-based simulations are well suited for testing hypotheses about the origin of observed emergent properties in a system. This can be done simply by experimenting with sets of initial conditions at the micro-level necessary to yield a set of desired behaviors at the macro-level. On the other hand, they also provide a powerful framework within which to integrate ostensibly "disjointed" theories from various related disciplines. For example, while basic agent-agent interactions may be described by simple physics and sociology, the internal decision-making capability of a single agent may be derived, in part, from an understanding of cognitive psychology.

The fundamental building block of most models of complex adaptive systems is the so-called *adaptive autonomous* agent. Adaptive autonomous agents try to satisfy a set of goals (which may be either fixed or time-dependent) in an unpredictable and changing environment. These agents are "adaptive" in the sense that they can use their experience to continually improve their ability to deal with shifting goals and motivations. They are "autonomous" in that they operate completely autonomously, and do not need to obey instructions issued by a God-like oracle.

Depending on the system being modeled and the environment that an agent populates, an adaptive autonomous agent can take on many different forms. In Deneubourg et al.'s [14] study of decentralized collective sorting, for example, the agents of the system are simple (*non*adaptive) robots that move about their physical environment and make elementary decisions about whether to pick up or drop an object. Examples of adaptive agents populating "cyberspace" are the so-called "software agents" (or "knobots") that are entities that navigate computer networks or cruise the World-Wide-Web searching for relevant bits of data.

In general, an adaptive autonomous agent is characterized by the following properties:

- It is an entity that, by sensing and acting upon its environment, tries to fulfill a set of goals in a complex, dynamic environment.

- It can *sense* the environment through its sensors and *act* on the environment through its actuators.
- It has an internal information processing and decision-making capability.
- Its anticipation of future states and possibilities, based on internal models (which are often incomplete and/or incorrect), often significantly alters the aggregate behavior of the system of which an agent is part.
- An agent's goals can take on diverse forms:
  - Desired local states
  - Desired end goals
  - Selective rewards to be maximized
  - Internal needs (or motivations) that need to be kept within desired bounds.

Since a major component of an agent's environment consists of other agents, agents generally spend a great deal of their time adapting to the adaptation patterns of other agents. The adaptive mechanism of an adaptive autonomous agent is typically based on a *genetic algorithm.*[5]

Insofar as complex adaptive systems can be regarded as being essentially open-ended problem-solvers, their lifeblood consists mostly of novelty. The ability of a complex adaptive system to survive and evolve in a constantly changing environment is determined by its ability to continually find — either by chance, or experience, or more typically both — insightful new strategies to increase its overall "fitness" (which is, of course, a constantly changing function in time).

Military campaigns likewise depend on the creative leadership of their commanders, success or failure often hinging either on the brilliant tactic conceived in the heat of combat or the mediocre one issued in its place.

To be realistic, such novelty must not consist solely of a randomly selected option from a main-options list (a common approach taken by conventional warfare models), but must at least have the possibility of being as genuinely unanticipated in the model as it often is on a real battlefield. To this end, each command-agent (and to a somewhat more limited extent, each primitive agent) must possess both a memory and an internal anticipatory mechanism that it uses to select the optimal tactic and/or strategy from among a set of predicted outcomes. This is an important point: except for doctrine and the historical lessons of warfare, the super-set of tactics *must not be hard-wired in.*

[5] A primer on genetic algorithms is given in Appendix B.

Such local-rule-based multiagent simulations are well suited for:

- Providing a theoretical framework for understanding aggregate behavior as fundamentally nonlinear and synergistic
- Studying the general efficacy of combat doctrine and tactics
- Exploring emergent properties and/or other "novel" behaviors arising from low-level rules (even doctrine if it is well encoded[6])
- Capturing universal patterns of combat behavior by focusing on a reduced set of critical drivers
- Suggesting likelihood of possible outcomes as a function of initial conditions
- Use as training tools along the lines of some commercially available agent-based "games," such as *SimCity*, *SimFarm*, and *SimLife*[7]
- Providing near-real-time tactical decision aids by providing a "natural selection" (via *genetic algorithm*) of superior tactics and/or strategies for a given combat situation
- Providing a natural arena in which to explore consequences of various qualitative characteristics of combat (unit cohesion, morale, leadership, etc.)
- Giving an intuitive "feel" for how and/or why unanticipated events occur on the battlefield, and to what extent the overall process is shaped by such events.

Ideally, one would hope to find universal patterns of behavior and/or tactics and strategies that are independent of the details of the make up of individual combat agents.

Agent-based simulations ought *not* be used either to predict real battlefield outcomes or to provide a realistic simulation of combat. While commercial networkable 3D virtual-reality games such as *Quake*[8] are much better suited to providing a virtual combat environment for training purposes, agent-based simulations are designed to help understand the basic processes that take place on the battlefield. It is not realism, for its own sake, that agent-based simulations are after, but

[6] It is an intriguing speculation that doctrine as a whole may contain both desirable and undesirable latent patterns that emerge only when allowed to "flow" through a system of elementary agents. An agent-based model of combat may provide an ideal simulation environment in which to explore such possibilities.

[7] W. Wright, *SimCity* (computer game), Orinda, California: Maxis Corporation, 1989.

[8] *Id Software*, World-Wide Web URL link = http://www.idsoftware.com.

rather a realistic understanding of the drivers (read: interactivity, decision-making capability, adaptability, and so on) behind what is really happening.

Recent Examples of Agent-Based Simulations

As fundamental research on complex systems grows, and its set of associated theoretical and/or exploratory tools is refined, the use of agent-based simulations can only become more widespread. Some of the more recent examples of agent-based simulations include Chris Barrett's TRANSIM model of traffic flow [15] (in which Albuquerque's road-traffic network is meticulously reproduced, boulevard by boulevard, and the simultaneous actions of many agent-drivers are used to explore countless *"What if?"* scenarios) and Epstein and Axtell's *Sugarscape* model of the evolution of social systems [16] (in which cultural evolution is studied by observing the collective behavior of many interactive agents, each endowed with a notional set of social interaction rules). Much attention has also been recently focused on developing intelligent software agents to help assimilate the exponentially growing information on the *World-Wide-Web* (see, for example, Maes [17]).

SWARM, Santa Fe Institute's SWARM project, headed by Chris Langton,[9] is a multiagent *meta*-simulation platform for the study of complex adaptive systems. The goal of the project is to provide the research community with a general-purpose artificial-life simulator. The system comes with a variety of generic artificial worlds populated with generic agents, a large library of design and analysis tools, and a "kernel" to drive the actual simulation. These artificial worlds can vary widely, from simple 2D worlds in which elementary agents move back and forth, to complex multidimensional "graphs" representing multidimensional telecommunication networks in which agents can trade messages and commodities, to models of real-world ecologies.

As SWARM was only in the initial stages of beta-testing at the conception of this project, ISAAC was coded in the C programming language to reduce development time. As the list of SWARM features continues to increase, however, and as SWARM itself matures as a bona-fide research vehicle, future versions of ISAAC may be written for SWARM. Ultimately, ISAAC may become a full fledged "combat programming language" (i.e., a *Mathematica*[10] for combat).

Agent-Based Simulations vs. Traditional Modeling Approaches

Fundamentally, an agent-based approach to modeling complex systems differs from more traditional differential-equation based approaches in

---

[9] World-Wide-Web URL link = http://www.santafe.edu/projects/swarm/.

[10] World-Wide-Web URL link = http://www.wri.com.

that it represents a shift from force-on-force attrition calculations to considering how high-level properties and behaviors of a systems emerge out of low-level rules. The conceptual focus of agent-based models shifts from finding a mathematical description of an entire system to a low-level rule-based specification of the behavior of the individual agents making up that system.

Table 3 compares the traditional reductionist approach to modeling complex systems with complex adaptive system/agent-based simulations.

Table 3. Comparison between traditional and agent-based approaches to complex systems modeling

	Traditional (Reductionist) Approach	Agent-Based Simulation
degrees-of-freedom	relatively few	typically many
interactions	typically weak and linear; need to be hard-wired into model	usually strong and nonlinear; low-level agents continually adapt to a changing environment
characteristic length and time scales	=1	>>1
specification of complex boundary conditions	can be difficult to specify analytically (say, as part of a partial differential equation model)	very easy to implement
model of individual combatant?	necessarily crude; assumes that all combatants are the same	more realistic; each combatant has its own unique history and therefore its own unique way of responding to the world
aggregation of variables	simpleminded aggregation of low-level variables	sets of high-level variables are self-organized and emergent; aggregate behavior is fundamentally nonlinear and synergistic
long term behavior	solve for steady-state equilibrium solution	nonequilibrium behavior is more descriptive of long-term dynamics
sought-for behavior	is either accounted for explicitly or is typically absent; focuses on force-on-force attrition ratios	high-level behavior (not accounted for directly) emerges naturally from low-level rules; focuses more on the overall attrition *process*

Agent-Based Simulations vs. Traditional Artificial Intelligence

"It is hard to point at a single component [of an AI program] as the seat of intelligence, there is no homunculus. Rather, intelligence emerges from the interactions of the components of the system. The way in which it emerges, however, is quite different for traditional and behavior-based AI systems." [18]

At first sight, the kinds of problems best suited for agent-based simulations appear to be similar to the kinds of problems for which

traditional artificial intelligence (AI) techniques have been developed. *How is an agent-based simulation different from a traditional artificial intelligence approach?* Maes [19] lists four key points that distinguish traditional AI from the study of adaptive autonomous agents:

1. Traditional AI focuses on systems exhibiting isolated "high-level" competencies, such as medical diagnoses, chess playing, and so on; in contrast, agent-based systems target lower-level competencies, with high-level competencies emerging naturally, and collectively, of their own accord.

2. Traditional AI has focused on "closed systems" in which the interaction between the problem domain and the external environment is kept to a minimum; in contrast, agent-based systems are "open systems," and agents are directly coupled with their environment.

3. Most traditional AI systems deal with problems in a piecemeal fashion, one at a time; in contrast, the individual agents in an agent-based system must deal with many conflicting goals simultaneously.

4. Traditional AI focuses on "knowledge structures" that model aspects of their domain of expertise; in contrast, an agent-based system is more concerned with dynamic "behavior producing" modules. It is less important for an agent to be able to address a specific question within its problem domain (as it is for traditional AI systems) than it is to be flexible enough to adapt to shifting domains.

Overview of ISAAC

What is ISAAC?

ISAAC is an acronym for Irreducible Semi-Autonomous Adaptive Combat, and represents (in its current form) a skeletal agent-based model of combat.[11] ISAAC's discrete, local-rule-based nonlinear dynamics is patterned loosely after two-dimensional cellular automata rules, with one major difference. In contrast to cellular automata models,[12] in which it is typically *information* that moves throughout the lattice with the sites themselves remaining fixed, ISAAC allows certain privileged kinds of sites (to be described below) to move throughout the lattice and to carry information with them. For that reason, ISAAC can be thought of as a *mobile cellular automata* model.

Mobile cellular automata have been used before to model predator-prey interactions in natural ecologies [20 through 22]. They have also been applied to combat modeling [23], but in a much more limited fashion than the one ultimately envisioned for ISAAC. The goal is for ISAAC to become a fully developed complex systems theoretic analyst's toolbox for identifying, exploring, and possibly exploiting emergent collective patterns of behavior on the battlefield.

ISAAC currently consists of four separate programs (see table 4): an interactive *Core Engine* that incorporates all of the behavioral and dynamical features that are described below (**ISAAC_CE**), a standalone *Genetic Algorithm Evolver* [13] that uses a slightly older version of ISAAC to *evolve* force characteristics that are "best-fit" for performing user-defined missions (**ISAAC_GA**), a single-squad version of the core engine that can be used to run and display data files generated by **ISAAC_GA** (**ISAAC_SQ**), and a *Parameter-Space Mapper* that also uses a slightly older version of ISAAC to map out the behavior of a system over a two-dimensional "slice" of ISAAC's total N-dimensional phase space (**ISAAC_PM**). Each of these programs will be described in detail in the following sections. The accompanying diskette also contains the program **ISAAC_PB** that can be used to "play-back" the recorded runs described in the ***Sample Runs*** section. A user's guide for **ISAAC_GA** appears in the section ***Genetic Algorithm Evolutions of ISAACA Personalities.*** **ISAAC_PM** is discussed in ***Taking 2D "Slices" of ISAAC's Parameter Space*** in the ***Data Collection*** section.

---

[11] Apart from its descriptive value, the acronym ISAAC was chosen to pay tongue-in-cheek homage to Isaac Newton. It seemed an appropriate choice to make given that the "new" (complex systems theoretic) sciences represent a fundamental shift *away* from linear (or so-called "Newtonian") thinking. Newton was, in fact, well aware of nonlinearities and their implications.

[12] For a brief primer on cellular automata, see appendix A of this report.

[13] For a brief primer on genetic algorithms, see appendix B of this report.

Table 4. A listing of program "modules" making up ISAAC

Program	Version	Decription
ISAAC_CE	1.8.4	The fully interactive (multi-squad version of the) ***Core Engine*** that includes all of the features described in this CRM
ISAAC_SQ	1.7.4	The *single-squad* version of the core engine that can be used to display files gnerated by **ISAAC_GA**
ISAAC_GA	1.5.1	A stand-alone ***Genetic Algorithm Evolver*** that uses the core engine to evolve personalities "best-fit" to perform a specified mission
ISAAC_PM	1.2.1	A stand-alone ***Parameter-Space Mapper*** that uses the core engine to "map-out" the behavior over a user-defined two-dimensional slice of ISAAC's total N-dimensional phase-space
ISAAC_PB	1.0.1	A stand-alone program that can be used to Play-Back peviously recorded runs; in particular, all of the *.out files on the distribution diskette

ISAAC can therefore be used in three different run "modes":

- **Interactive Mode**, in which **ISAAC_CE** (or **ISAAC_SQ**) is run interactively using a fixed set of rules. This mode, which allows the user to make 'on-the-fly' changes to the values of any (or all) parameters defining a given run (including the "decision-making personality" of individual ISAACAs), is particularly well suited for quickly and easily playing multiple *"What if?"* scenarios. This purely graphical run mode is also useful for interactively "searching" for interesting emergent behavior.

- **Data-Collection Mode**, in which **ISAAC_CE**, **ISAAC_SQ**, or **ISAAC_PM** are used to summarize entire runs by sets of various statistical measures. In particular, **ISAAC_PM** can be used to gain insight into what ISAAC's ostensibly N-dimensional parameter space looks like (modulo some well-defined mission "objective") by generating complete behavioral profiles on two dimensional slices of that parameter space.

- **GA-"Evolver" Mode**, in which **ISAAC_GA** is used to evolve a personality for one side that is "best" suited for performing some well-defined mission against a fixed personality (and force disposition) for the other. While **ISAAC_GA**'s current design is itself evolving (its embedded genetic algorithm, for example, is very simple at this stage and can be improved considerably), it is

powerful enough to illustrate how programs such as this can eventually be used to evolve real-world "tactics."

Keep in mind that while ISAAC is already arguably rich in structure and function, ISAAC is still very much a "work in progress." This report (and the accompanying programs) should therefore be viewed as preliminary work only. In particular, many of ISAAC's basic algorithms, behavioral rules, command and control structures, and data-collection routines are all subject to change.

What is an ISAACA?

The basic element of ISAAC is an ISAAC Agent (or ISAACA), which loosely represents a primitive combat unit (infantryman, tank, transport vehicle, etc.) that is equipped with the following characteristics:

- A default local-rule set specifying how to act in a generic environment (i.e., an embedded "doctrine")
- Goals directing behavior ("mission")
- Sensors generating an internal map of environment ("situational awareness")
- An internal adaptive mechanism to alter behavior and/or rules.

A global rule set determines combat attrition, reconstitution and (in future versions) reinforcement. ISAAC also contains both local and global commanders, each with their own command radii, and obeying an evolving C^2 hierarchy of rules.

ISAAC possesses the following general characteristics:

1. All low-level combatants respond to strictly *local* forms of information.
2. All local decision-dynamics is *decentralized* and *personality driven*; that is, driven by individual goals and motivations.
3. Local dynamics is *adaptive* and *nonlinear.*
4. The generic "template" of decision-making is consistent among the various levels of decision-making (that range from low-level ISAACAs to local commanders to global commanders to the super commander).

ISAAC is *local* because each ISAACA senses, reacts, and adapts only to information existing within a prescribed finite sensor range. It is *decentralized* because there is no master "oracle" dictating the actions of each and every ISAACA. Instead, each ISAACA senses, assimilates, and reacts to all information individually and without guidance. It is *nonlinear* because of the nonlinear nature of the local decision-making process that each ISAACA uses to choose a "move." It is *adaptive* because each ISAACA adaptively changes its default ("doctrinal") rules according to its local environment at each time step. There is thus a continual dynamical feedback between the local and global levels. The manner in which its rules are changed proceeds according to each ISAACA's *personality*, or its intrinsic value system. Each of these points is discussed in more detail below.

ISAAC's basic approach is similar in spirit to a *cellular automaton* (CA) model but augments the conventional CA framework in two ways: (1) individual units can move through the lattice (recall that in CA, what moves is the information, not the site), and (2) evolution proceeds not according to a fixed set of rules, but to a set of rules that adaptively evolves over time. When the appropriate internal flags are set to make use of a hierarchical command and control structure, ISAAC differs from conventional CA models in one additional way: individual states of cells (or combatants) do not just respond to local information, but are capable of assimilating nonlocal information (via an embedded C^2 topology) and command hierarchy. In future versions, global rule (i.e., command) strategies will evolve in time (say, via a genetic algorithm). In this case, orders pumped down echelon will be based on evolved strategies played out on possibly imprecise mental maps of local and/or global commanders. Thus, ISAAC will be an ideal test-bed in which to explore such questions as *"What is the tactical and/or strategic impact of information?"*

Design Philosophy

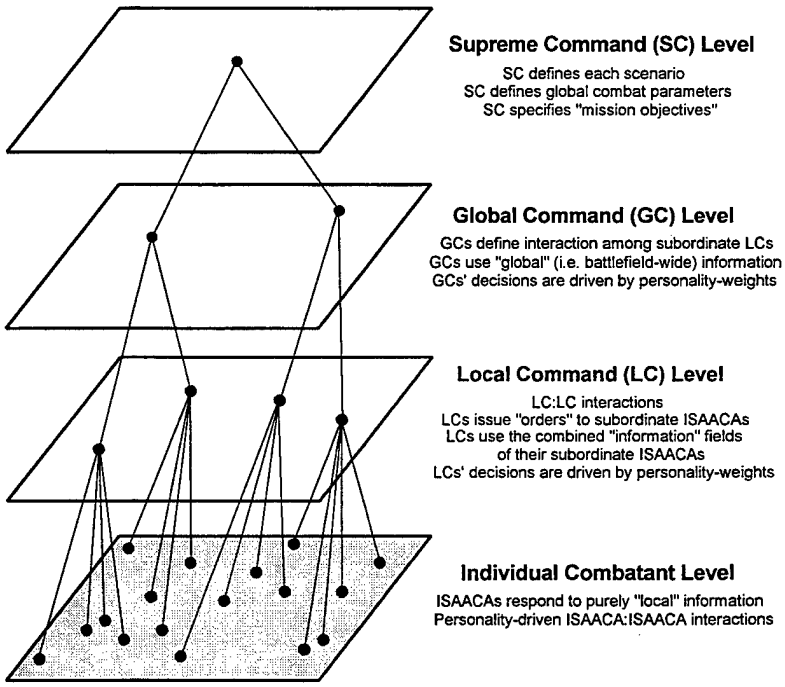
ISAAC has been developed primarily to address the basic question: *"To what extent is land combat a self-organized emergent phenomenon?"* As such, its intended use is not as a full system-level model of combat but as an interactive toolbox (or "conceptual playground") in which to explore high-level emergent behaviors arising from various low-level (i.e., individual combatant and squad-level) "interaction rules." The idea behind ISAAC is not to model in detail a specific piece of hardware (M16 rifle, M101 105mm howitzer, *etc.*), but to provide an understanding of the fundamental behavioral tradeoffs involved among a large number of notional variables. ISAAC is designed to be an interactive dynamical arena in which the user can explore various *"What if?"* scenarios of the form: *"What if blue's sensor range is halved but its fire power is doubled?"*, *"What if the enemy is a much more 'aggressive' force than*

anticipated?", "What if reinforcements are added to a force whose combat personalities are 'at odds' with the personality of the existing force?", etc.

Information Levels

Figure 5 shows a schematic of ISAAC's hierarchy of information levels. The details of these levels will be given in appropriate sections that follow; here, we describe only those basic aspects of these levels that are relevant for illustrating ISAAC's overall design philosophy.

Figure 5. Schematic of ISAAC's hierarchy of information levels



The lowest level of the hierarchy is the level of the individual combatant, or ISAACA, and consists of all information contained within the notional battlefield that an individual ISAACA can sense and react to; namely, friendly and enemy ISAACAs, and proximity to "goals" (see below) and/or terrain. This lowest level is the one on which the dynamical interactions between ISAACAs occur.

The next two levels are command levels that consist of information pertinent to making "decisions" regarding the behavior on lower levels. Local commanders, for example, assimilate and respond to a "pool" of mid-level information consisting partly of the information contained within their own field-of-view (which typically extends beyond that of a single ISAACA) and partly of the information communicated to them

by their subordinate ISAACAs. Local commanders use this mid-level information to adjust the movement vectors of the individual ISAACAs under their command. Global commanders use global (i.e., battlefield-wide) information to issue movement vectors to local commanders (and, therefore, their subordinate ISAACAs) as well as to define how the subordinate ISAACAs under the command of one local commander are to interact with the subordinate ISAACAs under the command of another local commander.

Finally, the top-level supreme commander represents the interactive user of the software. The user is responsible for completely defining a given scenario, fixing the size and features of the notional battlefield, setting the initial force dispositions, and specifying any auxiliary combat conditions (such as fratricide, reconstitution, combat termination conditions, and so on). The supreme commander also defines the "mission objective" required by the standalone *genetic algorithm* (**ISAAC_GA**) and *parameter-space mapper* (**ISAACA_PM**) programs.

Guiding Principles

ISAAC's design philosophy is grounded upon two guiding principles: (1) keep all dynamical components and rules as simple as possible, and (2) treat all forms of information (and the way in which all forms of information are assimilated) equally, but in a contextually consistent manner.

The first principle refers to a concerted effort to adhere to a relatively small set of basic combat and movement rules, and to define those rules as *intuitively* as possible. Thus, the power projection rule is essentially *"target and fire upon any enemy ISAACA within a threshold fire range"* vice some other, more complicated (albeit, a possibly more realistic) prescription. Remember that the idea is to qualitatively probe the behavioral consequences of the interaction among a large number of notional variables, not to provide an explicit detailed model of combat.

The second principle refers to the fact that almost all dynamical "decisions" in ISAAC – whether they are made by individual ISAACAs, by local or global commanders, or by the user himself (in defining a scenario's "mission objectives") – are *personality driven.* That is to say, all decisions are based on what might loosely be called a "personality" that attaches greater or lesser degrees of importance to each factor relevant to making a particular decision. It is in this sense that all forms of information, on all levels, are treated equally. Because decisions on different levels necessarily involve different kinds of information – for example, an individual ISAACA's decision to "stay put" in order to survive is quite different, and uses a different kind of information, from a global commander's drive to "get to the enemy flag as quickly as possible" – one must be careful to use whatever information must be

used for a decision on a given level in a manner that is appropriate for that level.

The decision-dynamics taking place on different levels are all mutually consistent in that each decision-maker (whether it is an individual ISAACA, a local or global commander) follows the same general template of probing and responding to his environment. Each decision really consists of answering the following three basic questions:

- ***Question 1:*** *What are my immediate and/or long-term goals?*
- ***Question 2:*** *What do I currently sense in (and/or know about) my environment?*
- ***Question 3:*** *How can I use what I currently know of my environment to attain my goals?*

As we shall see in detail below, an individual ISAACA cares only about "moving toward" or "moving away from" all other ISAACAs and/or his own and the enemy's flag. An ISAACA's personality prescribes how much relative weight is assigned to each of these immediate "goals." On the other hand, a global commander must weigh such factors as overall force strength, casualty rate, rate of advance, and so on in order to attain certain long-term goals. Local and supreme commanders have their own unique concerns. While the actual decisions are different in each case and on each level, the general template of how those decisions are made has been designed to be essentially the same.

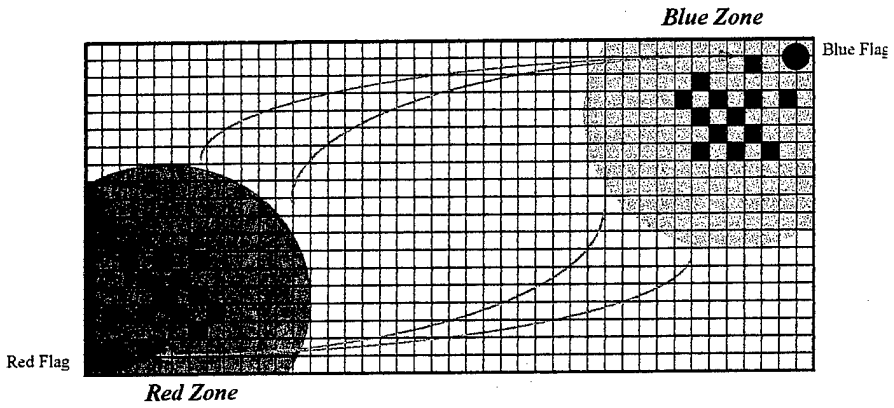
Combat Battlefield

The putative "combat battlefield" is represented in ISAAC by a two-dimensional lattice of discrete sites. Each site of the lattice may be occupied by one of two kinds of ISAACAs: *red* or *blue.* The initial state consists of either user-specified formations of red and blue ISAACAs positioned at diagonally opposite corners of the battlefield or of a random distribution of red and blue ISAACAs occupying the central square region (of user-specified dimension). Red and blue "flags" are also typically (but not always) positioned in diagonally opposite corners: a red flag in the red ISAACAs corner and a blue flag in the blue ISAACAs corner. A typical "goal," for both red and blue ISAACAs, is to successfully reach the "flag" positioned in the diagonally opposite corner (see figure 6). ISAAC also has the provision of defining a notional terrain. Future versions will include a menu of environmental obstacles as well.

Each ISAACA exists in one of three states: *alive, injured,* or *dead.* Injured ISAACAs can (but are not required to) have different personalities from when they were alive. By default, an injured ISAACA's ability to shoot an

enemy is equal to 1/2 of its ability when alive. Also, if the alive ISAACA chooses its moves from among lattice sites within a distance of two or more from its current position, an injured ISAACA's move range is reduced to the minimum possible range of one unit.

Figure 6. Putative two-dimensional "Combat Battlefield" in ISAAC



Each ISAACA has associated with it a set of *ranges*, within which it senses and assimilates simple forms of local information, and a *personality*, which determines the general manner in which it responds to its environment. Ranges and personality are described in more detail below.

Note that while ISAAC is currently designed to accommodate only two kinds of forces (*red* and *blue*), a notional *white* force can also be defined by exploiting the features of ISAAC's multi-squad option (see ***ISAACA Squads***).

Program Flow

The program loops through the following basic steps:

- *Step 1:* Initialize "battlefield" and ISAACA distribution parameters
- *Step 2:* Display summary descriptions of red and blue ISAACAs
- *Step 3:* Set time counter to 1
- *Step 4:* Adjudicate combat
- *Step 5:* Refresh Battlefield Graphics Display

- *Step 6:* Adapt personality weight vector for each red and blue ISAACA
- *Step 7:* Compute local penalty for each of the possible "moves" that each red and blue ISAACA may choose to take from its current position
- *Step 8:* Move ISAACAs to their newly selected position (some may choose to "do nothing")
- *Step 9:* Go to *Step 4* and repeat.

The most important parts of this skeletal structure are contained in steps 4, 6, and 7, or the parts dealing with the adjudication of combat, the adaptation of personality weights, and the decision-making process that each ISAACA goes through in order to "choose" its next move. Before describing the details of what each of these steps involves, we must first discuss how each ISAACA partitions its local information.

ISAACA Ranges

As noted earlier, each ISAACA can know and respond only to information that is local to its immediate position. Figure 7 shows a schematic of the five kinds of user-specified ranges that surround each ISAACA:

- Sensor range
- Fire range
- Threshold range
- Movement range
- Communications Range

Sensor Range

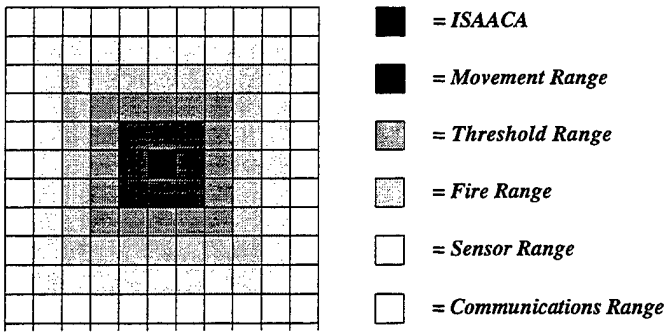
The *sensor range* (= r_S), shown in figure 7 as the outer-most shaded box surrounding the ISAACA positioned at the black-colored center site, defines the maximum range at which the ISAACA can sense other ISAACAs in its neighborhood. Note that this range effectively defines a boxed area around the ISAACA and not a circle of radius r_S.

Fire Range

The *fire range* (= r_F) defines the boxed area surrounding an ISAACA within which the ISAACA can engage enemy ISAACAs in combat (see discussion below). Any enemy ISAACA that is closer to a given ISAACA

than the given ISAACA's fire range may be fired upon by the given ISAACA.

Figure 7. Various kinds of ranges that surround each ISAACA



Threshold Range

The *threshold range* (= r_T) defines a boxed area surrounding an ISAACA with respect to which that ISAACA computes the numbers of friendly and enemy ISAACAs that play a role in determining what move to make on a given time step. This local decision-making process is described below.

Movement Range

The *movement range* (= r_M) defines a boxed area surrounding an ISAACA that defines the region of the lattice from which the ISAACA can select a move on a given time step. In the current version of ISAAC, $r_M = 1$ or 2, and each ISAACA can select to either "do nothing" (that is, remain where it is) or move to an adjacent lattice site.

Communications Range

The *communications range* (= r_C) defines a boxed area surrounding an ISAACA such that any friendly ISAACA within a range r_C of that centrally located ISAACA communicates the information content of its local sensor field. How the centrally located ISAACA makes use of that information is discussed in a later section.

Note that while figure 7 shows these four ranges in the particular order $r_C > r_S > r_F > r_T > r_M$, the user can specify any ordering.

ISAACA Personalities

The *personality* of an ISAACA **X** represents **X**'s internal value-system as applied to the set of all possible relevant information that **X** must use to select a move or strategy. It is defined by a personality weight vector.

Personality Weight Vector

Each ISAACA is equipped with a user-specified "personality," defined by a six-component *personality weight vector,*

$$\vec{w} = (w_1, w_2, ..., w_6)$$

where $0 \leq |w_i| \leq$ and $\sum_i |w_i| =$. The components of the personality weight vector specify how an individual ISAACA responds to distinct kinds of local information within its sensor and threshold ranges.

The personality weight vector may be state-dependent. That is to say, $\vec{w}(alive)$ need not. in general, be equal to $\vec{w}(injured)$. The components of $\vec{w}$ can be also *negative,* in which case they signify a propensity for moving *away from,* rather than toward, a given entity.

The default personality rule structure is defined as follows. Since there are two kinds of ISAACAs (red and blue), and each functioning (i.e., non-dead) ISAACA can exist in one of two states (alive and injured), each ISAACA can respond to effectively four different kinds of information appearing within its sensor range r_s:

- The number of *alive friendly* (i.e., like-colored) ISAACAs,
- The number of *alive enemy* (i.e., different colored) ISAACAs,
- The number of *injured friendly* ISAACAs, and
- The number of *injured enemy* ISAACAs.

Additionally, each ISAACA can respond to how far it is from both its own (like-colored) "flag" and its enemy's "flag." Table 5 summarizes the association among the components of the personality weight vector and these six kinds of information, and gives examples of *defensive* and *offensive* personalities.

The meaning of these components is to be interpreted as follows: w_i represents the *relative weight afforded to moving closer to the i^{th} type of information.*

Table 5. Components of the personality weight vector $\vec{w}$

		Examples	
Personality weight	**Type of Information**	*Defensive*	*Offensive*
w_1	alive friendly ISAACAs	30/100	5/100
w_2	alive enemy ISAACAs	-10/100	25/100
w_3	injured friendly ISAACAs	40/100	0
w_4	injured enemy ISAACAs	10/100	50/100
w_5	own flag	10/100	0
w_6	enemy flag	0	20/100

A personality is defined by assigning a weight to each of the six kinds of information. For example, one ISAACA might give all its attention to like-colored ISAACAs, and effectively ignore the enemy. The personality weight vector for such an ISAACA might be given by $\vec{w} = (1/3, 0, 1/3, 0, 1/3)$, signifying that this ISAACA gives equal weight to moving closer to friendly ISAACAs and the enemy flag. Another ISAAC might care only about defending its own goal, and might thus have a personality prescribed by $\vec{w} = (0, 1/3, 0, 1/3, 1/3, 0)$. More "well-rounded" ISAACAs might better distribute their attention to both friendly and enemy ISAACAs with, say, a weight vector given by $\vec{w} = (1/5, 1/5, 1/5, 1/5, 0, 1/5)$.

An example of a fairly aggressive personality is one whose weight vector is given by $\vec{w} = (1/20, 5/20, 0, 9/20, 0, 5/20)$. Such a personality is five times more "interested" in moving toward alive enemies than it is in moving toward alive friendlies (effectively ignoring injured friendlies altogether), and is more interested in moving toward injured enemies than it is even in advancing toward the enemy flag. An ISAAC that has a personality defined by entirely *negative* weights – say, $\vec{w} = (-10, -10, -10, -10, -10, -10)$ – wants to move away from, rather than toward, every other ISAACA and both flags.

Among the many interesting questions that such a weight specified internal value-system immediately suggests is, "What is the "best" personality mix to use for a given measure-of-effectiveness?" A technique for directly addressing such questions using *Genetic Algorithms* is described in a later section.

The personality weight vector as defined above fixes the default personality of a given ISAACA. That is, it fixes the personality with which an ISAACA responds to local information *in the absence of other constraints.* Constraints might include a condition such as clamping a given ISAACA's innate desire to "get closer to" friendly ISAACAs once the number of surrounding friendly ISAACAs (within the constraint range, r_C; see above) exceeds a given threshold. Another constraint might be for a given ISAACA not to advance toward the enemy flag

unless it is surrounded by a threshold number of friendly ISAACAs. These and other constraints are discussed in more detail below.

A given ISAACA's personality weight vector is used to *rank* the set of possible moves that it can choose to take during the current time step.

Squads

ISAAC allows the user to define up to 10 different personality weight vectors (both alive and injured) for 10 separate *squads* of ISAACAs. Like almost everything else in ISAAC, squads are entirely notional entities and refer simply to collections of ISAACAs sharing the same personality. Aside from enhancing the innate dynamical richness of ISAAC's general conceptual phase-space in an intuitive way, squad-specific parameters can be used to explore such basic *"What If?"* questions of the form "What if I had a just a few more good soldiers?"

Squad-specific parameters in the current version of ISAAC include initial spatial disposition, sensor, fire and movement range, alive and injured personality weight vectors, notional defensive strength, movement constraint thresholds, single-shot probability and maximum target number. (See ***Contents of Input Data File***)

White Forces

Having squad-specific parameters available makes it possible to effectively populate the notional battlefield with a third *white* force; i.e., one whose personality does not include a motivation factor to move toward either red or blue flags.

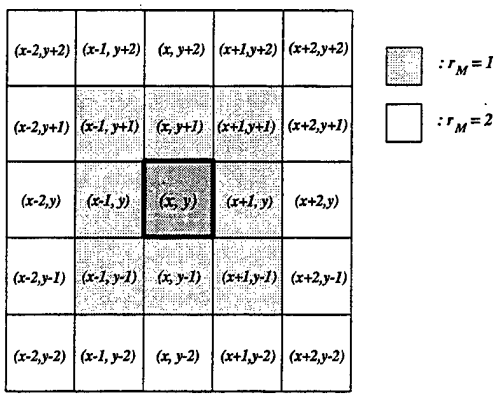
ISAACA Move Selection

In the current version of ISAAC, each ISAACA can choose to move from its current position at time t – say, (x_t,y_t) – to any of the sites that are either a distance 1 (if the movement range is equal to r_M=1) or 2 (if the movement range is equal to r_M=2) from (x_t,y_t); see figure 8. It can also select to "do nothing" and remain at its current position. Each site of the battlefield lattice may be occupied by at most one ISAACA at a given time.

An ISAACA's personality weight vector is used to rank each possible move according to a *penalty function.* The penalty function effectively measures the total distance that the ISAACA will be from other ISAACAs (which includes both friendly and enemy ISAACAs) and from its own and enemy flags, each weighted according to the appropriate component of the personality weight vector, $\vec{w}$. An ISAACA moves to the position that incurs the least penalty, or the move that best satisfies

the ISAACA's personality-driven desire to "move closer to" other ISAACA's in given states and either of the two flags.

Figure 8. Set of possible ISAACA moves from its current (x,y) position



The general form of the penalty function is given by:

$$
\begin{aligned}
Z(x,y) = \; & w_1 \, s_{\text{red}}^{-1} \, N_{\text{alive red}}^{-1} \sum_{\text{alive red; i}} d[i;(x,y)] \; + \\
& w_2 \, s_{\text{blue}}^{-1} \, N_{\text{alive blue}}^{-1} \sum_{\text{alive blue; i}} d[i;(x,y)] \; + \\
& w_3 \, s_{\text{red}}^{-1} \, N_{\text{injured red}}^{-1} \sum_{\text{injured red; i}} d[i;(x,y)] \; + \\
& w_4 \, s_{\text{blue}}^{-1} \, N_{\text{injured blue}}^{-1} \sum_{\text{injured blue; i}} d[i;(x,y)] \; + \\
& w_5 \, d_{\text{new}}[\text{red flag};(x,y)] \,/\, d_{\text{old}}[\text{red flag};(x,y)] \; + \\
& w_6 \, d_{\text{new}}[\text{blue flag};(x,y)] \,/\, d_{\text{old}}[\text{blue flag};(x,y)] \; ,
\end{aligned}
$$

where w_i are the components of the personality weight vector (see ***ISAACA Personalities***), $s_{\text{red}} = \sqrt{2}\, r_{\text{red}}$, and $s_{\text{blue}} = \sqrt{2}\, r_{\text{blue}}$ are red and blue scale factors, respectively, d[i; (x,y)] is the distance between the i^{th} element of a given sum and the ISAACA positioned at (x,y), N_i is the total number of elements within the given ISAACA's sensor range, and d_{new} and d_{old} represent distances computed using the given ISAACA's new (i.e., candidate move) position and old (i.e., current) position, respectively. For example, the summation $\sum_{\text{alive red; i}} d[i;(x,y)]$ appearing at the top of the above expression represents the sum of distances from the position (x,y) to all red alive ISAACAs located within the sensor range box of position (x,y). In the case of a red ISAACA, this sensor range box is defined by sensor range $r_{\text{red,S}}$; in the case of a blue ISAACA, it is defined by sensor range $r_{\text{blue,S}}$.

Figure 9. General movement rule

Move to the square that best satisfies personality-driven "desire" to get closer to (or farther away from) friendly and enemy ISAACAs and enemy (or own) goal

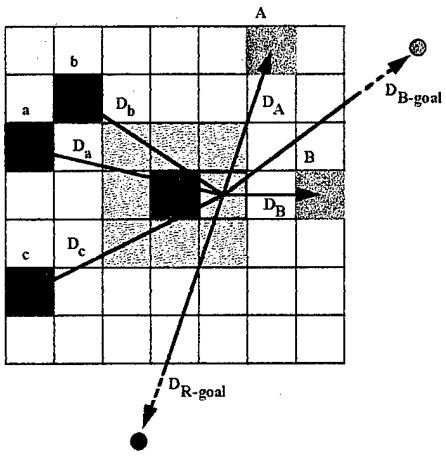
A "penalty" is computed for each possible move: Z_1, Z_2,..., Z_N. If the movement range r_M=1, N=9; if r_M=2, N=25. The actual move is the one that incurs the least penalty. If there is a *set* of moves (consisting of more than one possible move) that incur exactly the same minimum penalty, an ISAAC randomly selects the actual move from among the candidate moves making up that set.

The most general movement rule is summarized in figure 9.

Example

Figure 10 shows a portion of the notional battlefield surrounding a red ISAACA **X** positioned at (x,y). There are three red ISAACAs (a,b and c at distances D_a, D_b and D_c from **X**, respectively) and two blue ISAACAs (A and B, at distances D_A and D_B from **X**, respectively) within **X**'s sensor range.

Figure 10. Sample penalty calculation



Assuming that **X**'s movement range r_M=1, **X**'s next move is determined by minimizing the penalty Z(x',y') that will be incurred by selecting each of the nine nearest neighboring sites, $(x' = x, y' = y)$ and $(x' = x \pm 1, y' = y \pm 1)$ (shown in gray in figure 10). The penalty is given explicitly by

$$Z(x',y') = w_1 s_{red}^{-1}\left(\frac{1}{3}\right)[D_a + D_b + D_c] + w_2 s_{red}^{-1}\left(\frac{1}{2}\right)[D_A + D_B] + w_5\left(\frac{D_{R-goal}}{D^0_{R-goal}}\right) + w_6\left(\frac{D_{B-goal}}{D^0_{B-goal}}\right),$$

where $D_{R\text{-goal}}$ and $D^0_{R\text{-goal}}$ are the distances from (x,y) and (x',y') to the red goal, respectively, and $D_{B\text{-goal}}$ and $D^0_{B\text{-goal}}$ are the distances from (x,y) and (x',y') to the blue goal, respectively.

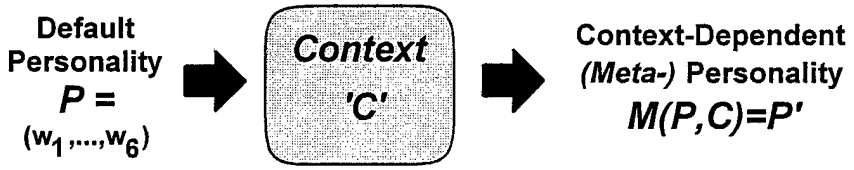
Move Sampling Order

There are two ways in which moves can be sampled during an ISAAC run. At the start of each run, a randomly ordered list of red and blue ISAACAs is first set up prior to the start of the actual dynamics loop. During all subsequent passes, ISAACA moves are then determined either by sequencing through the ISAACAs on this list in *fixed order*, or, for better realism, in *random order*. The user can choose a fixed or random sampling order by setting a "move-sampling flag" at run-time (See ***General Battle Parameters*** in ***Contents of Input Data File***).

ISAACA Adaptability

As discussed above, ISAAC has been specifically designed to accommodate a personality-based local dynamics. This means that, despite changing local environments and conditions, each ISAACA responds to what it "senses" round itself according to its own individual personality. The term "personality" here is of course used somewhat figuratively, as it only loosely refers to what we conventionally mean by a human personality. However, it *does* accurately reflect the individually consistent manner in which ISAACA's assimilate and act upon information in their sensor fields. One way in which such a consistent, but thus far *fixed*, manner of responding to information can be augmented is to allow each ISAACA's personality to adapt to changing contexts and/or evolve over time.

Figure 11. Schematic of ISAACA *Meta*-Personality



Conceptually, the idea is fairly simple. Each ISAACA is equipped not only with a fixed default personality (as defined by its personality weight vector), but with a set of rules that tell it how to alter its default personality according to various environmental conditions. In other words, each ISAACA is endowed with a *meta*-personality that tells it how to adapt its default personality. Figure 11 shows a schematic representation of this approach.

In the current version of ISAAC, each ISAACA "adapts" to its local environment in a relatively simple fashion. A typical meta-personality consists essentially of altering a few of the components of the default personality weight set according to several local threshold constraints, measured with respect to a user-specified threshold range (= r_T; see ***ISAACA Ranges***).

There are six possible constraints:

- Advance Constraint
- Cluster Constraint
- Combat Constraint
- Minimum distance to friendly ISAACAs
- Minimum distance to enemy ISAACAs
- Minimum distance to own flag.

Advance Constraint

Consider the *advance constraint.* The constraint consists of specifying a threshold number of friendly ISAACAs that must be within a given ISAACA's constraint range r_C in order for that ISAACA to continue advancing toward the enemy flag.

Recall that w_6 represents the relative weight that is assigned toward "moving closer to" the enemy flag (see table 5). If the actual number of neighboring friendly ISAACAs is greater than or equal to the threshold value, the given ISAACA uses the default weight $+w_6$ to decide upon its next move. However, if the actual number of neighboring friendly ISAACAs is less than the threshold value, the given ISAACA decides its next move by using $- w_6$ to decide upon its next move. That is, it effectively attempts to "move away from" rather than "move closer to" the enemy goal. Intuitively, the advance constraint embodies the idea that unless a combatant is surrounded by a sufficient number of friendly forces, he will not advance toward the goal.

Cluster Constraint

Similarly, the *cluster constraint* embodies the idea that once an ISAACA is surrounded by a sufficient number of friendly forces, that ISAACA will no longer attempt to "move closer to" friendly forces.

The cluster constraint consists of specifying a threshold number of friendly ISAACAs that must be within a given ISAACA's constraint range r_C in order for the given ISAACA to no longer move closer to friendly ISAACAs. If the actual number exceeds that threshold, the given ISAACA will decide upon its next move using $w_1 = w_3 = 0$ (see table 5), thereby effectively ignoring nearby friendly forces.

Combat Constraint

The *combat constraint* determines the local conditions for which a given ISAACA will choose to move toward or away from possibly engaging an enemy ISAACA (see ***ISAACA Combat*** below).

Intuitively, the idea is that if a given ISAACA senses that it has less than a threshold advantage of surrounding forces over enemy forces, it will choose to move away from engaging enemy ISAACAs rather than moving toward (and, thereby, possibly engaging) them. More specifically, the combat constraint consists of choosing a threshold value of the difference ($= \Delta_c$) between the number of friendly forces contained within the given ISAACAs constraint-range box (= $N_{friendly}(r_C)$) and the number of enemy forces contained within the given ISAACAs sensor range ($= N_{enemy}(r_S)$): $\Delta_c = N_{friendly}(r_C) - N_{enemy}(r_S)$. If the actual difference, Δ_{actual}, is greater than this threshold advantage, the default weight set remains unaffected and the given ISAACA proceeds to move toward the enemy. If Δ_{actual} is less than Δ_c, then the given ISAACA will decide upon its next move using the weights $w_2 = - w_{2,\,default}$ and $w_4 = - w_{4,default}$, where $w_{2,default}$ and $w_{4,default}$ are the default weights for moving toward alive and injured enemy ISAACAs (see table 5). A large*positive* combat threshold represents a *defensive* mannered ISAACA force, since such ISAACA's will choose to move away from rather than engage an enemy unless they have a strong advantage. A large *negative* combat threshold represents an *offensive* mannered ISAACA force, since such a force will choose to move toward and possible engage an enemy even if the relative force strengths overwhelmingly favor the enemy.

Minimum Local-Distance Constraints

The last three constraints – minimum distance to friendly ISAACAs, minimum distance to enemy ISAACAs and minimum distance to own flag – specify distances such that if a given ISAACA ever finds itself at a distance *less than* the threshold distance to the given entity it will choose to move away from rather than toward that entity. For example, in the case of the minimum distance to friendly ISAACAs, say that threshold distance is set to 3. If the given ISAACA finds itself at a distance 5 from a

particular friendly ISAACA, this constraint has no effect, and the given ISAACA chooses its next move using the appropriate default personality weight $w_1 = w_{1,default}$. However, if the given ISAACA finds itself at a distance 2 from another particular friendly ISAACA, the constraint induces the given iSAACA to use $w_1 = -w_{1,default}$ instead of $+w_{1,default}$. The last constraint, specifying the minimum threshold distance to an ISAACA's own flag, can be meaningfully applied only if $w_{5,default} > 0$, otherwise it has no effect on ISAACA's decision on where to move.

Note that unlike the first three constraints (advance, cluster and combat) – which determine what weights will be applied to neighboring ISAACAs *collectively* – these last three constraints are applied to neighboring ISAACAs *individually* and locally. That is to say, that the decision to use a default personality weight or take its negative is made on an individual basis, and is made according to whether each neighboring ISAACA is closer to or farther away from the given ISAACA than the prescribed threshold distance. This decision is made during the calculation of the penalty function and is therefore implicit in each of the sums appearing in the previous expression for Z.

Constraint rules are summarized in figure 12.

Figure 12. Sample constraint rules

ADVANCE:	**Advance toward enemy goal if and only if the number of surrounding friendlies exceeds a given threshold**
CLUSTER:	**Stop moving toward surrounding friendlies if the number of friendlies exceeds a given threshold**
COMBAT:	**Move toward and engage the enemy if and only if the difference between friendly and enemy force strengths exceeds a given threshold**

ISAACA Combat

In its current version, ISAAC adjudicates combat in the simplest possible manner (see figure 13). During the combat phase of an iteration step for the whole system, each ISAACA **X** (on either side) is given an opportunity to "fire" at all enemy ISAACAs **Y** that are within a fire range r_F of **X**'s position. If an ISAACA is shot by an enemy ISAACA, its current state is degraded either from alive to injured or from injured to dead. Once "dead," that ISAACA is permanently removed from

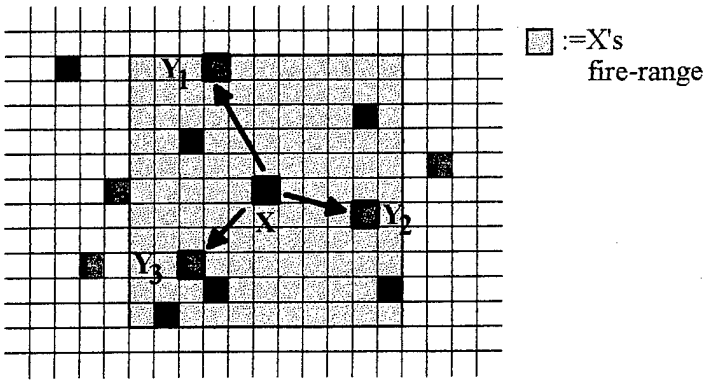
further play. The probability that a given enemy ISAACA is "shot" is fixed by user specified single-shot probabilities for red-by-blue ($p_{ss} = p_{rb}$) and blue-by-red ($p_{ss} = p_{br}$). The single-shot probability for an injured ISAACA, $p_{ss}^{injured}$, is, by default, equal to one half of its single-shot probability when it is alive ($p_{ss}^{injured} = 1/2\ p_{ss}^{alive}$).

By default, *all* enemy ISAACAs within a given ISAACA's fire range are targeted for a possible hit. However, the user has the option of limiting the number of simultaneously engageable enemy targets. If this option is selected, and the number of enemy ISAACAs within an ISAACA's fire-range exceeds a user-defined threshold number (say N), then N ISAACAs are randomly chosen from among the ISAACAs in this set.

This basic combat "logic" may be enhanced (by setting the appropriate software "flags" prior to run-time; see ***User's Guide***) by three additional functions:

- ***Defense,*** which adds a notional ability to ISAACAs to be able to withstand a greater number of "hits" before having their state degraded,
- ***Reconstitution,*** which adds a provision for previously injured ISAACAs to be reconstituted to their alive state, and
- ***Fratricide,*** which adds an element of realism to ISAAC combat by making it possible to inadvertently "hit" friendly forces

Figure 13. Blue **X** targets 3 Red ISAACAs, Y_1, Y_2 and Y_3



Defense

Each ISAACA is endowed with a notional defensive capability (i.e., "armor") that defines the number of successful "hits" that are required to degrade an alive ISAACA to an injured state or remove an injured ISAACA from further play. By default the notional defensive strength

of all (alive and injured) ISAACAs is equal to 1, meaning that a single hit is sufficient to degrade an ISAACAs state. Setting the notional defensive strength of one side equal to the total number of iteration steps desired for the entire run effectively renders that side impervious to enemy fire.

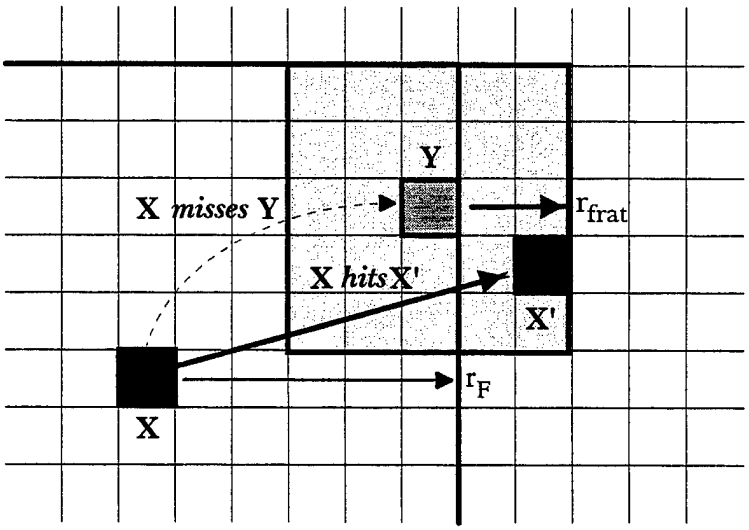
Reconstitution

If the reconstitution flag is set at run time (see ***User's Guide***), each ISAACA is endowed with a fixed reconstitution time t_{recon}. This adds the logic that if an ISAACA **X**, after being "hit" by an enemy ISAACA **Y** (and thereby being degraded from an alive to an injured state), is not hit during the next t_{recon} iteration steps, **X** is reconstituted to its alive state. Note that setting reconstitution time to $t_{recon} = 0$ is effectively equal to having an *infinite* notional defense, since an ISAACA that is "hit" by the enemy is immediately reconstituted.

Fratricide

If the fratricide flag is set at run time (see ***User's Guide***), every potential engagement of an enemy ISAACA entails the possibility of fratricide. Specifically, if an ISAACA **X** targets an enemy ISAACA **Y** (that is within the fire range r_F of **X**) but *does not* "hit" **Y** – a hit/miss being decided by **X**'s single-shot probability p_{ss} – then, with probability p_{frat}, a friendly ISAACA **X'** within a fratricide range r_{frat} of **Y** may be hit instead (see figure 14).

Figure 14. Schematic of a fratricide "hit" of **X'** by **X**



Communication

If the communication flag is set (see ***User's Guide***), each ISAACA **X** can communicate with all friendly ISAACAs **Y** that are located within a *communications range* r_C of **X**; see figure 15.

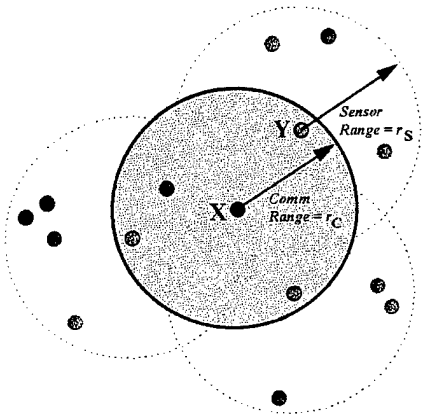
All friendly ISAACAs **Y** within range r_C of **X** communicate to **X** the information contained within their own sensor range r_S. ISAACA **X** then incorporates this additional information into its penalty calculation by weighing all communicated information with a communication weight $w_{comm} \geq 0$. The full penalty is defined as

$$Z(x,y) = Z_0(x,y) + w_{comm} Z_{comm}(x,y) ,$$

where $Z_0(x,y)$ is the communications-*free* penalty function defined earlier, and $Z_{comm}(x,y)$ is the same penalty function applied to communicated information.

If $w_{comm} = 0$, **X** effectively ignores all communicated information; if $w_{comm} = 1/2$, **X** considers all communicated information "half as important" as the information contained within its own sensor field; if $w_{comm} = 1$, **X** considers all communicated information on equal terms with information within its sensor range.

Figure 15. Schematic of ISAACA communications



Command and Control

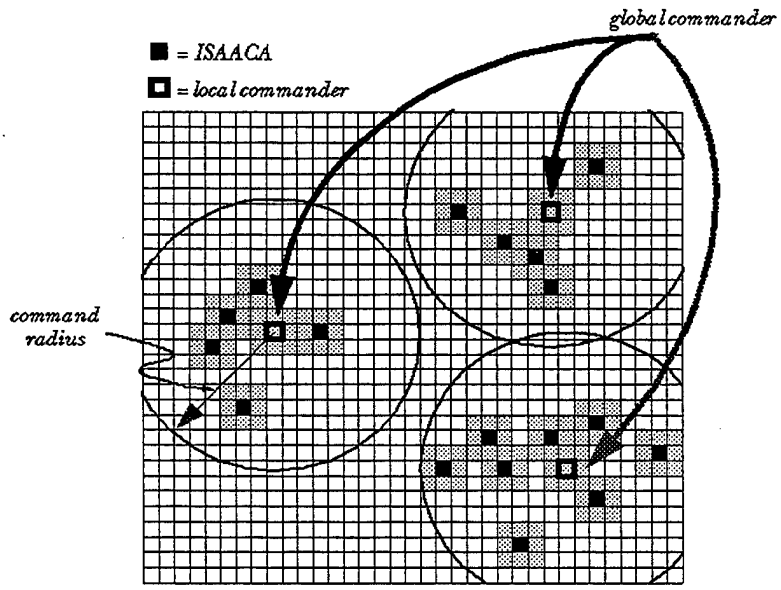
In its simplest run mode, ISAAC has only one kind of ISAACA, and uses a strictly decentralized command and control structure: ISAACAs do not communicate with any other ISAACAs and all ISAACAs base their decisions on information that is strictly local to their sensor's

field-of-view. Although such a design is entirely adequate for exploring the consequences of having a rigorously decentralized C^2 structure, any serious analysis tool of real combat must, of course, include some form of a functioning C^2 hierarchy.

To this end, the user has the option of defining a notional command and control (C^2) hierarchy within ISAAC. This hierarchy consists of ISAACA *collectives*, wherein red and blue sides both consist of three different kinds of ISAACAs (see figure 16):

- *Elementary combatants,* such as have been already described above.
- *Local commanders,* which are ISAACAs that command, and coordinate information flow among, local clusters of elementary combatants.
- *Global commanders,* which are ISAACAs that have a more global perspective of the battlefield, and coordinate the actions of the local commanders under their command.

Figure 16. Schematic representation of a ISAACA C^2 hierarchy



In its current design (which will undoubtedly evolve in sophistication in future versions of ISAAC), local and global commands involve a goal-specification on successive nestings of blocked sites. That is to say, one way in which, say, local commanders can issue orders to the elementary combatants under their command — in a way that is also consistent with the general individual-personality-driven decision-making process introduced in this paper — is to issue

intermediate "goals" that the elementary combatants must attain within certain time frames within given blocks of sites. The elementary combatants use exactly the same personality-driven criteria to select their local moves, but their goals no longer consist solely of getting to the enemy's flag; in addition, their goals now include a variety of lesser, local goals as specified by their local commanders. The selection of these local goals, in turn, are driven by more global strategies that would be the result of the decision-making processes taking place in the "minds" of global commanders (see below).

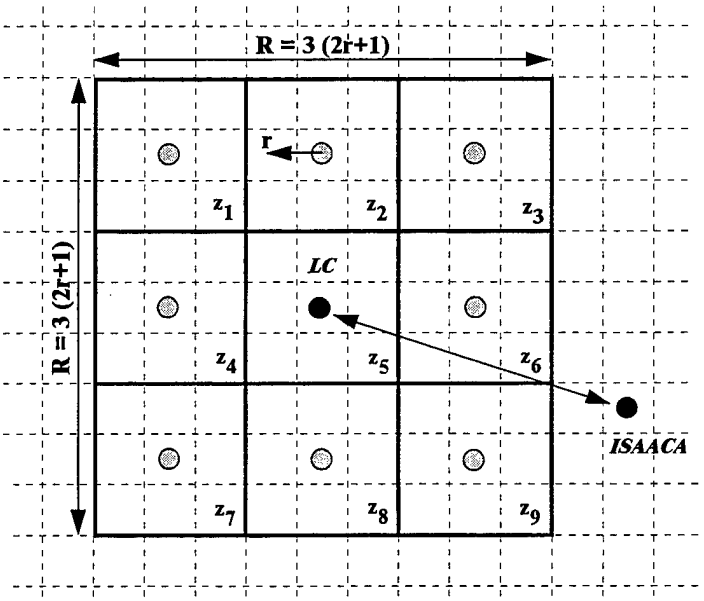
Local Command

If the user chooses to use the local command option (which is done by setting an appropriate "flag" in ISAAC's data-input file; see discussion in ***A Concise User's Guide to ISAAC***), the internal logic of the program is enhanced in two ways:

1. *Local commanders* (LCs) are introduced, and are given a certain number of subordinate ISAACAs to command.

2. Elementary ISAACAs that are under the command of a LC are endowed with two additional weights, one of which defines their propensity for "staying close to" their LC ($=w_{LC}$) and the other the propensity for "obeying" the orders issued by their LC ($=w_{obey}$).

Figure 17. Local command (see text)



Local Command Area

Local commanders are endowed with a surrounding "command area" (defined by a command radius $R_{command}$) that moves with them as they move throughout the lattice. This command area is also partitioned either into 3-by-3 or 5-by-5 blocks of smaller blocks. Figure 17 shows a schematic of a typical local command structure partitioned into nine sub-blocks.

The center positions of these smaller blocks represent transient "local goals" that a local commander can order his subordinates to "move toward" during a given move (how these orders are actually issued is discussed immediately below). The size of these smaller blocks is equal to $(2r_{block}+1)$-by-$(2r_{block}+1)$ where r_{block} is the effective (user-defined) "radius" of the block. The overall command area is therefore either a $3(2r_{block}+1)$-by-$3(2r_{block}+1)$ or a $5(2r_{block}+1)$-by-$5(2r_{block}+1)$ square.

The user can define up to 25 different LCs, each of which can have up to 100 subordinate ISAACAs under their command. Each LC is also endowed with a unique *movement*– and *command*–personality.

A LC's movement personality is defined by the same personality weight vector described earlier, except that it does not have to equal the personality weight vector assigned to its subordinate ISAACAs. For example, the "personality" of a LC's subordinates may be defined by the personality weight vector $\vec{w} = (10, 40, 10, 40, 0, 50)$ and continue moving toward neighboring friendly forces only until surrounded by five friendlies, while their commander "wants" only to progress toward goal (i.e., $\vec{w} = (0, 0, 0, 0, 0, 10)$) while *always* seeking to cluster with whatever friendly forces may be nearby.

If no enemy ISAACAs are sensed in the local command area, all components of the LC's personality weight vector are temporarily set to zero (w_1=...=w_5=0) except w_6 (i.e., enemy goal).

The local command personality is defined by four weights (α_{local}, β_{local}, δ_{local} and γ_{local}) that prescribe the relative degree of importance the LC places on various measures of *relative information* contained in each block of sites within his command area. Specifically, the LC weighs each block of sites by a penalty weight z_i given by

$$z_i = \alpha_{local}\left(F_i^{alive} - E_i^{alive}\right)F_i^{-1} + \beta_{local}\left(F_i^{alive} - E_i^{injured}\right)F_i^{-1} + \delta_{local}\left(F_i^{injured} - E_i^{alive}\right)F_i^{-1} + \gamma_{local}\left(F_i^{injured} - E_i^{injured}\right)F_i^{-1},$$

where F_i^{alive} and $F_i^{injured}$ are the number of alive and injured *friendly* ISAACAs in the i^{th} block, E_i^{alive} and $E_i^{injured}$ are the number of alive and

injured *enemy* ISAACAs in the i^{th} block, $0 \le \alpha_{local}, \beta_{local}, \delta_{local}, \gamma_{local}$ and $\alpha_{local} + \beta_{local} + \delta_{local} + \gamma_{local}$.

In words, an LC identifies the block of sites within his command area that contains the smallest fractional difference between friendly and enemy forces; i.e., the block B_i for which z_i is a minimum. All subordinate ISAACAs are then "ordered" to move toward the center of that block. In the event that more than one block yields the same minimum value, the LC chooses the one that is closest to the block chosen on the previous iteration step.

If a local commander is killed, a random ISAACA under its command is "promoted" to LC status and resumes the previous LC's function.

Subordinate ISAACAs

As mentioned above, once the local command option is enabled, the personality weight vector defining the elementary ISAACAs under the command of a LC is automatically enhanced to include two new weights:

- $0 \le w_{LC} \le 1$, that defines the relative weight afforded to "staying close to" their LC, and
- $0 \le w_{obey} \le 1$, that defines the relative weight afforded to "obeying" the orders issued by their LC.

In the case of $w_{LC} > 0$, a subordinate ISAACA will seek to "move closer to" his LC (or, more specifically, his LC's x,y coordinates) whenever that subordinate is *outside* his LC's command area. If the subordinate ISAACA is *inside* his LC's command area, w_{LC} is temporarily (i.e., during that iteration step) set to zero. Note that w_{LC} is actually determined by a user-specified pre-factor α that multiplies the maximum value of an ISAACA's personality weight vector (see ***User's Guide***); i.e.,

$$w_{LC} = \alpha \times \max(|w_1|, ..., |w_6|) .$$

Once the LC issues a "move to block B" order, his subordinate ISAACAs respond by incorporating the (x,y) coordinates of the center of block B $(=(x_B, y_B))$ by using the following penalty function for their individual move selection:

$$Z = Z_0 + w_{LC}(\text{move to } x_{LC}, y_{LC}) + w_{obey}(\text{move to } x_B, y_B) ,$$

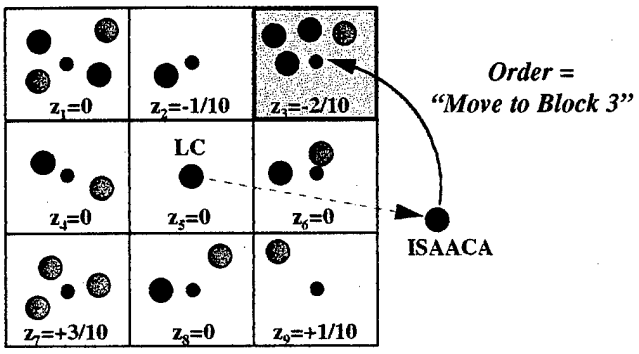
where Z_0 is the penalty function used by individual ISAACAs (without local command).

The values of w_{LC} and w_{obey} relative to 1 (the effective constant in front of Z_0) define the relative weights that an individual ISAACA gives to either "staying close to" or "obeying" his LC. For example, a maximally insubordinate ISAACA that totally disregards his LC's orders has $w_{obey}=0$.

Example

Figure 18 shows an example of a blue LC defined by local command weights $\alpha_{local} = \beta_{local} = \delta_{local} = \gamma_{local} = 1/4$. In this case, the penalty weight for the i^{th} block, B_i, of his command area is simply equal to the difference between the number of blue and red ISAACAs in B_i divided by the total number of red ISAACAs in the entire command area. With the local distribution of red and blue ISAACAs as shown in figure 18, the LC finds that – *consistent with his command personality as defined by the weights* α_{local}, β_{local}, δ_{local}, and γ_{local} – block B_3 (in which a single blue ISAAC in block B_3 is outnumbered by three red ISAACAs) is the block that is in the greatest need of local blue assistance. The LC therefore issues a "move to block B_3" order to all subordinate ISAACAs.

Figure 18. Local command (see text)



Global Command

Global commanders (GCs) issue orders to local commanders using global (i.e., battlefield-wide) information. GCs effectively know *everything* about the overall state of the battle, at least during the preceding iteration step. Their orders to subordinate LCs consist of two parts: (1) the manner in which to respond to other LCs, and (2) a direction into which to move.

GC Command of LC-LC interaction

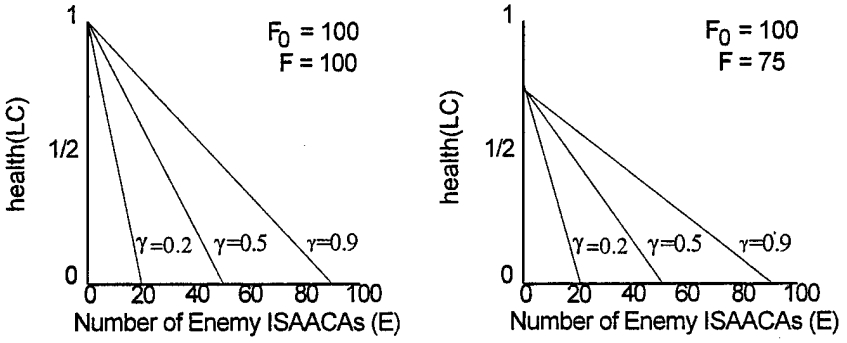
Interaction among LCs is mediated by the GC, who effectively "decides" when to send a local commander LC_i (and his subordinates) to "help" a nearby LC_j according to the relative *health states* of LC_i and LC_j. The health state of the i^{th} local commander, $0 \leq \text{health}(LC_i) \leq 1$, is a simple

measure of how close the overall state of that LC's command area is to its initial state. If all subordinates are present[14] and there are no enemy ISAACAs within the command area, the health state is maximum and health(LC_i) = 1; if all subordinates have been killed or the command area contains the maximum number of allowable enemy ISAACAs, the health state is minimum and health(LC_i) = 0. Intuitively, as an LC's health value decreases, "need for assistance" increases. More specifically, health(LC_i) is defined as follows:

$$\text{health}(LC_i) = \sigma\left(\frac{F(1+\gamma F_0 - E)}{F_0(1+\gamma F_0)}\right),$$

where $\sigma(x) = x$ if $0 \le x \le 1$ else $\sigma(x) = 0$, F_0 is the total number of alive friendly subordinates under LC_i's command, F is the current number of subordinates within the local command area, E is the current number of enemy ISAACAs within the local command area, and $0 < \gamma \le 1$ is a factor that specifies the maximum number of "allowable" enemy ISAACAs (as a fraction of the initial number of friendly subordinates). For example, if γ=1, then health(LC_i) ~ 0 when $E = F_0$ and health(LC_i) ~ 1/2 when $E = 1/2\ F_0$. Figure 19 shows plots of health(LC_i) as a function of E for γ=0.2, γ=0.5, and γ=0.9 for the cases (F_0 = 100, F=100) and (F_0 = 100, F=75).

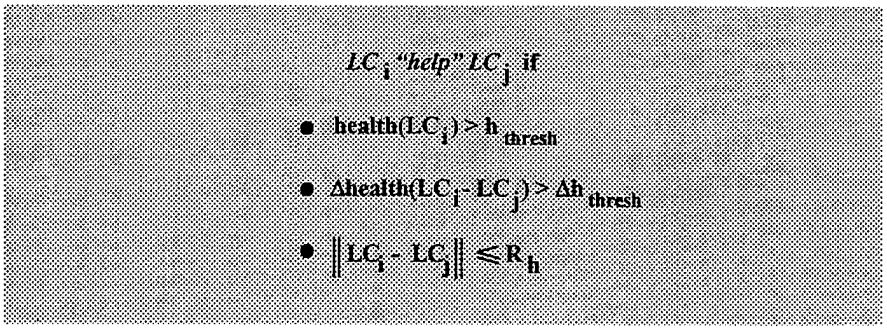
Figure 19. Plot of health(LC) versus number of enemy ISAACAs



If a given LC_i is "healthy" enough – that is, if his health state exceeds a given threshold, h_{thresh} – he looks for other LCs, LC_j, within his help range R_h that he can move toward to help. Candidate LCs to help are those for which the relative fractional health (= Δ_{ij}) is greater than a health threshold (=Δh_{thresh}). LC_i moves toward the closest LC_j that needs help. This GC LC-LC interaction rule is summarized in figure 20.

[14] The health measure as defined in this version of ISAAC does not yet discriminate between individual alive and injured ISAACA states.

Figure 20. Rule for GC mediated LC-LC interaction



GC Command of Autonomous LC Movement

In addition to mediating the interaction among subordinate local commanders, the GC also determines the direction into which each of his LCs shall move. In order to explain how a GC "decides" upon a direction, we must first introduce three ideas: (1) *battlefield sectors,* (2) *way-points,* and (3) the *GC-fear index.* Figure 21 illustrates the pertinent parameters.

Consider a local commander LC_i under the command of a GC. Using the (x_i,y_i) coordinates of LC_i's position at time t, the GC partitions the entire battlefield into 16 sectors $(S_1,S_2,...S_{16})$. The boundary of each of these sectors is set by (x_i,y_i) and the (x,y) coordinates of 16 next-nearest neighboring *way-points* $(\omega_1, \omega_2, ..., \omega_{16})$, equally spaced along the edge of the battlefield. These way-points represent the possible "directions" into which a subordinate LC might be ordered to move.[15] The definition of sectors S_1 and S_7 are shown in figure 21.

Each way-point ω_i is assigned a weight that represents *the penalty that will be incurred if LC_i moves in that point's direction.* Assume that the red and blue flags are positioned near the lower-left and upper right of the battlefield, respectively. Since the red GC wants to get to the blue flag (near ω_9), the red GC way-point weight distribution is fixed by setting ω_1 to the maximal possible value, 1, and ω_9 to the minimal value, 0; the remaining weights for ω_2 - ω_8 and ω_{10} - ω_{16} are then assigned values between 0 and 1, with higher penalties appearing for points closer to ω_1: $\omega_9 = 0 \leq \omega_8, \omega_{10} \leq \cdots \leq \omega_2, \omega_{16} \leq \omega_1 =$. For blue GCs, that want to get to the red flag near ω_1), the way-point weight distribution is just the opposite: ω_9 is assigned the maximal possible value, 1, and ω_1 the minimal value, 0, and $\omega_1 = 0 \leq \omega_2, \omega_{16} \leq \cdots \leq \omega_8, \omega_{10} \leq \omega_9 = 1$.

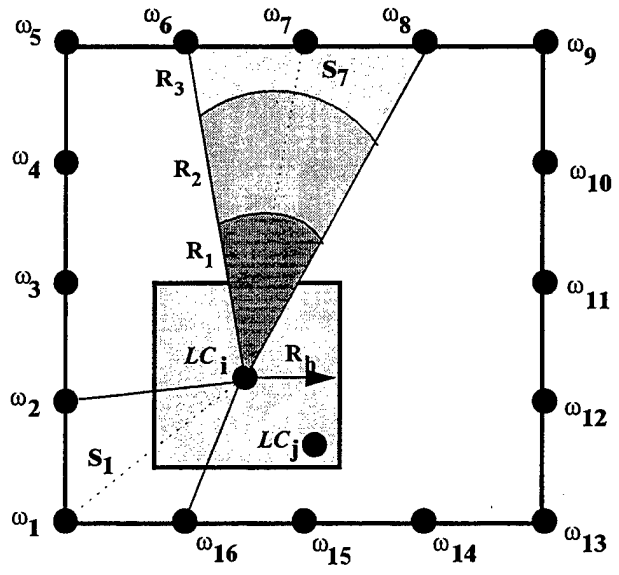
[15] Note that this program logic is currently defined only for the case where the red and blue flags are located in the lower-left and upper-right of the notional battlefield. Future versions of ISAAC will eliminate this arbitrary constraint.

Figure 21 shows that each sector S_i is subdivided into three rings: an inner ring (that is closest to LC_i's position) R_1, a middle ring R_2, and an outer ring R_3. These rings, from inner to outer, define successively "less important" regions to the GC, as far as the information that he will use to guide LC_i's motion is concerned. Specifically, this information is the density of enemy ISAACAs within the different sub-regions of a given sector. Thus the inner-most region, out to a radius R_1 from LC_i's position, represents that area around LC_i that the GC cares most about. The middle area, at distances $R_1 < R < R_2$ from LC_i's position, represents an intermediate level of importance. The outer ring, at distances greater than R_2, represents an area of the battlefield that a GC is currently least concerned about in deciding what direction to order LC_i to move into.

In defining these sectors and their sub-regions, the user supplies values for R_1, and R_2 along with the relative weights $0 \le w_{R_1}, w_{R_2}, w_{R_3} \le 1, w_{R_1} + w_{R_2} + w_{R_3} = 1$ that specify the relative degree of importance that the GC will assign to the corresponding sub-regions of each sector.

Having defined *battlefield sectors* and *way-points*, we are now finally in a position to describe how a GC "decides" upon a direction into which to send each of his subordinate LCs, as well as what those LCs do with such orders. In words, the GC computes a penalty value, P_i, for ordering a LC into sector S_i (spanning way-points ω_{i-1} and ω_{i+1}), and orders the LC toward the sector S_i for which P_i is minimal.

Figure 21. Global command (see text)



The penalty value consists of two parts, the intrinsic penalty incurred by moving toward way-point ω_i, and the penalty incurred by moving into a sector that has a given density of enemy ISAACAs. Specifically,

$$P_i = (1-f)\,\omega_i + f\rho_i^E,$$

where ρ_i^E is the number of enemy ISAACA per unit area in sector S_i (weighed using weights w_{R_1}, w_{R_2}, and w_{R_3} for sub-regions R_1, R_2 and R_3 of each sector; see above), and $0 \le f \le 1$ is the GC *fear index*. If $f = 0$, the GC is effectively fearless of the enemy and the criteria by which he decides what direction to send each of his LCs into consists entirely of the intrinsic penalty value associated with each way-point; i.e., $P_i = \omega_i$. On the other hand, if $f = 1$, the GC is concerned only with keeping his LCs (and their subordinates) away from harm's way and his choice of movement vector is made entirely on the basis of the enemy force strength in each sector. Any value for f between these two limits represents a GC command-personality-defined tradeoff between wanting to *simultaneously* satisfy two desires: moving LCs closer to the enemy flag and preventing them from encountering too many enemy forces while doing so.

The general rule for GC command of LC movement is summarized in figure 22.

Figure 22. Rule for GC command of LC movement

Order LC to move to the sector S_i for which the penalty P_i is minimum:

- $P_i = (1 - f)\,\omega_i + f\,\rho_i^E$
- $\rho_i^E = \alpha\,\rho_i^E(AE) + \beta\,\rho_i^E(IE)$
- $0 \leqslant f \leqslant 1$

LC Response to GC Commands

Having received an "order" from his GC, a local commander must now weigh several different factors to decide on his own course of action: his sensor view of battlefield, the disposition of his subordinate ISAACAs within his command area, and the tradeoff between helping other local commanders (who, according to the GC, may be in need of assistance), and moving toward the enemy goal. The LC's decision is shaped by using three additional command weights:

- $0 \leq \Omega_{help} \leq 1$, that defines the relative weight afforded to moving toward and "assisting" another LC,
- $0 \leq \Omega_{sector} \leq 1$, that defines the relative weight afforded to moving into a GC-ordered battlefield sector , and
- $0 \leq \Omega_{obey\text{-}GC} \leq 1$, that defines the relative weight afforded to obeying GC orders.

Once the GC issues orders to "move toward another LC" and/or "move toward way-point ω_i", his subordinate LCs decide upon their own moves according to the following penalty function:

$$Z = Z_0 + \Omega_{obey\text{-}GC} \{\Omega_{help}(\text{move toward } LC_i) + \Omega_{sector}(\text{move toward } LC_i)\}$$

The value of $\Omega_{obey\text{-}GC}$ relative to 1 (the effective constant in front of Z_0) defines the relative weight that a LC assigns to the movement vectors ordered by his GC. For example, $\Omega_{obey\text{-}GC}= 1$ means that the LC treats his own information and the information supplied by his GC on an equal footing; $\Omega_{obey\text{-}GC}= 0$ means that the LC effectively ignores his GC's orders.

A Concise User's Guide to ISAAC

ISAAC is a DOS program. It can be run from either the DOS command line or within a DOS-box in windows. Since it is written in ANSI-C, ISAAC is highly portable, though its current version uses graphics primitives defined in *Microsoft's* Visual C/C++ compiler for DOS (v1.52). A fragment of ISAAC's source code and header files is provided in appendix C.

Hardware Requirements

Computer Memory

The executable supplied on the accompanying disk has been compiled using *Phar Lap's* 286|DOS-Extender[16] to allow it to use up to 16 MB of extended memory. While the source code can of course by compiled without a memory extender, and ISAAC can be run with computers equipped with even 1 MB of RAM, DOS's 640K memory ceiling places a significant constraint on the battlefield size and/or maximal number of ISAACAs that can be defined per run (see table 6).

Table 6. Tradeoff between available memory and some basic run-time parameters in ISAAC

Parameter	Using DOS-Extender (16 MB)	No DOS-Extender (640K Max)
Number of ISAACAs/side	< 500	< 80
Battlefield size	< 150-by-150	< 80-by-80

Graphics

ISAAC can be run in either VGA (640-by-480) or SVGA (800-by-600) graphics modes, the default (and preferred) mode being SVGA. On some older computers, a utility program such as `vvesa.com` must be run prior to running ISAAC to enable SVGA graphics.

[16] *Phar Lap Software*, Inc., 60 Aberdeen Avenue, Cambridge, MA 02138; WWW address: http://www.pharlap.com.

Installing ISAAC

To install ISAAC, copy all the files from the accompanying disk into a subdirectory on a hard drive. For example, assuming the disk is in drive **A** and you wish to install ISAAC in a subdirectory called ISAAC on hard drive **C**, enter the command

xcopy a:*.*/s/v c:/ISAAC (...followed by **ENTER**)

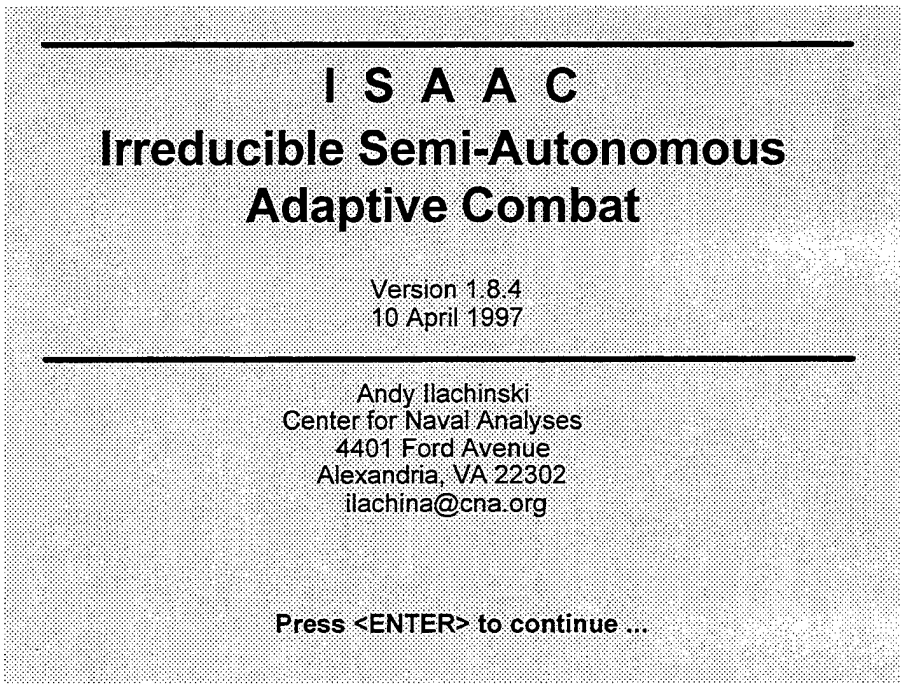
This command will copy all executables and necessary font files, and create two additional subdirectories in **c:/ISAAC>**:

- **c:/ISAAC/DAT>** – which contains sample input data files
- **c:/ISAAC/OUT>** – which contains sample *.out files of previously recorded runs

Starting ISAAC

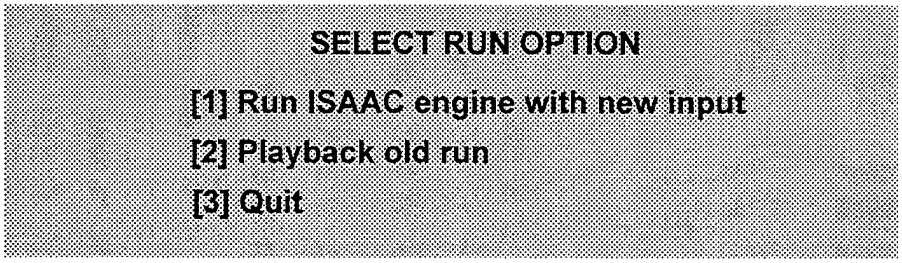
To start the *Core Engine* of ISAAC (see table 4), go to the appropriate subdirectory on the hard drive (say, **C:/ISAAC>**) and type the command **ISAAC** followed by **<ENTER>** on the DOS command line. You will see the opening screen (figure 23), specifying the current version and build date of the program and a prompt to press **<ENTER>** to continue.

Figure 23. ISAAC's opening screen



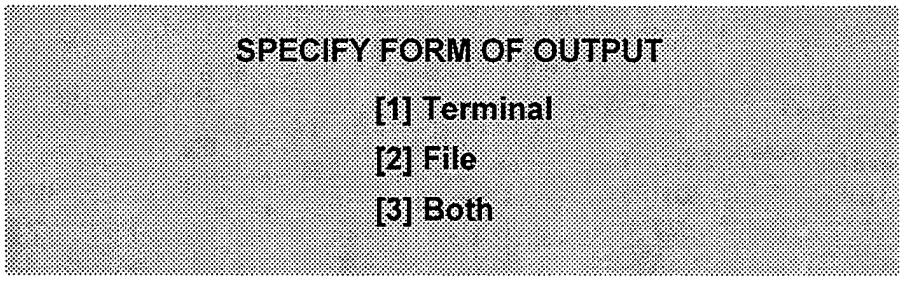
The next screen prompts for the type of run desired (see figure 24). You may select to either run the core engine with some new input, to replay an old run at high speed using a previously stored data file, or quit the program. Because ISAAC is computation intensive, the speed of real-time screen updates of an actual run (i.e., using the core engine) will depend heavily on the speed of the computer. As a baseline measure, each iteration of a scenario in which there are 50 red and 50 blue ISAACAs, each ISAACA evolves according to all six movement constraints outlined in the previous section and each ISAACA possesses a moderate sensor range ($r_S = 4$), takes about 1-1.5 sec to update on the screen of a 33 MHz 486-class computer. Play-back of the same scenario on the same computer from a data file proceeds roughly ten times faster.

Figure 24. ISAAC's main option screen



If option 1 is selected, the user is prompted to select the form of data input. Data may be input by either having ISAAC prompt the user for all information on screen, or by reading directly from an input data file. Several sample input files (**ISAAC.dat**, **ISAAC_LC.dat**, **ISAAC_GC.dat**, and **TERRAIN.dat**) are provided on the supplied disk. Their contents are described below.

Figure 25. ISAAC's second option screen



If option 2 is chosen, the program prompts for a file name, then loads and executes the playback of that file. The accompanying disk contains several self-extracting compressed "OUT" files (as previously recorded files will be called) that, when executed, automatically decompress (and

store on the hard drive) several sample `*.out` data files that can then be run with option 2. (See **Sample Runs**)

The user is next asked to select the method of data output during the current run; see figure 25.

The choices are either to display output only to the computer's screen (option 1), to by-pass the screen and store directly to a file (option 2), or to do both (option 3). If the selected option involves writing to a file, the user is prompted for an output data file name and the total number of iteration steps to record. (As mentioned above, ISAAC's sample output files are named ***.out**, but the user may, of course, choose any other name.) The default, and preferred graphics mode is SVGA (800-by-600 resolution), but ISAAC can "fall back" to VGA (640-by-480) automatically if the hardware and/or software combination does not permit the higher resolution mode.

Contents of ISAAC's Input Data File

As mentioned above, the accompanying disk contains several sample data files: (1) **ISAAC.dat**, which contains parameters defining a simple scenario in which 50 red and 50 blue ISAACAs advance toward one another without either side having any command and control structure; (2) **ISAAC_LC.dat**, which defines a scenario in which red is endowed with a single local commander; (3) **ISAAC_GC.dat**, which gives red a global commander with three subordinate local commanders, and (4) **TERRAIN.dat**, which is a sample terrain data file populating the battlefield with several impenetrable terrain features.

Because of the number of user-specified options and parameter settings available, the preferred mode of data entry is always direct file input, though the user always has the option of using on-screen prompts to input all values. In this section we will describe in some detail the contents of a generic ISAAC *.dat file.

A typical ISAAC *.dat file is nothing more than a lengthy listing of labeled parameter values partitioned into several self-contained sections:

- **General battle parameters**
  - Initial distribution
  - Fratricide parameters
  - Reconstitution
- **Statistics parameters**
- **Red global command parameters**

  - Direction parameters
  - Help parameters
- **Blue global command parameters**
  - Direction parameters
  - Help parameters
- **Red local command parameters**
  - Local commander parameters
  - Local commander personality
  - Local commander constraints
  - Local command weights
  - Global command weights
- **Blue local command parameters**
  - Local commander parameters
  - Local commander personality
  - Local commander constraints
  - Local command weights
  - Global command weights
- **Red ISAACA parameters**
  - ALIVE personality weights
  - INJURED personality weights
  - ISAACA-LC weights
  - Sensor/fire ranges
  - Communications
  - Movement constraints
  - Combat/engagement
- **Blue ISAACA parameters**
  - ALIVE personality weights
  - INJURED personality weights
  - ISAACA-LC weights
  - Sensor/fire ranges
  - Communications
  - Movement constraints
  - Combat/engagement
- **Terrain parameters**

Note that most sections are further subdivided into one or more subsections containing clusters of related variables. Not all sections contain values for all scenarios, however. Also, some sections, such as

those for defining local command parameters and terrain are variable in length (see below).

General Battle Parameters

The first section of user-specified parameters is the ***General Battle Parameters*** section. A sample fragment appears in figure 26. A short description of each variable appearing in this section is given below.

Figure 26. General battle parameters

```
*******************************
* GENERAL BATTLE PARAMETERS
*******************************
battle_size         80
*
* initial distribution
*
init_dist_flag?     1
R_box_(l,w)         20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20
RED_cen_(x,y)       20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20
B_box_(l,w)         20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20
BLUE_cen_(x,y)      70,70 70,70 70,70 70,70 70,70 70,70 70,70 70,70 70,70 70,70 70,70 70,70
B_flag_(x,y)        79,79
R_flag_(x,y)        1,1
termination?        2
move_order?         2
combat_flag?        2
terrain_flag?       0
*
* fratricide parameters
*
red_frat_flag?      0
blue_frat_flag?     0
red_frat_rad        1
blue_frat_rad       1
red_frat_prob       0.001000
blue_frat_prob      0.001000
*
* reconstitution
*
reconst_flag?       0
RED_recon_time      10
BLUE_recon_time     10
```

battle_size

The first entry is **battle_size**, which defines the length of one of the sides of the two-dimensional square lattice on which the run is to take place. The user can specify any integer number between 10 and 150.

init_dist_flag

init_dist_flag can take on one of three integer values: 1, 2 or 3. If **init_dist_flag** = 1, the user defines the actual spatial distribution of red and blue ISAACAs (see next few parameter entries); if **init_dist_flag** = 2,

red and blue ISAACAs initially consist of random formations near the lower-left and upper-right corners of the notional battlefield; if **init_dist_flag** = 3, red and blue ISAACAs are initially randomly placed within a square box at the center of the battlefield.

R_box_(l,w)

This defines the (length, width) of the "box" containing the initial distribution of red ISAACAs for each of ten squads. Note that ISAAC assumes that all ten fields will be filled in even if there are fewer than ten red squads.

RED_cen_(x,y)

The (x,y) coordinates of the center of the "box" containing the initial distribution of red ISAACAs for each of ten squads: $0 < x, y <$ **battle_size**. Note that ISAAC assumes that all ten fields will be filled in even if there are fewer than ten red squads. x and y are constrained to lie between 1 and **battle_size.**

B_box_(l,w)

This defines the (length, width) of the "box" containing the initial distribution of blue ISAACAs for each of ten squads. Note that ISAAC assumes that all ten fields will be filled in even if there are fewer than ten red squads.

BLUE_cen_(x,y)

The (x,y) coordinates of the center of the "box" containing the initial distribution of blue ISAACAs for each of ten squads: $0 < x, y <$ **battle_size**. Note that ISAAC assumes that all ten fields will be filled in even if there are fewer than ten red squads. x and y are constrained to lie between 1 and **battle_size.**

B_flag_(x,y)

The (x,y) coordinates of the blue flag: $0 < x,y <$ **battle_size**.

R_flag_(x,y)

The (x,y) coordinates of the red flag: $0 < x,y <$ **battle_size**.

termination?

This parameter "flag" specifies the termination condition that will be used during this run: if **termination** is set equal to 1 then the run is terminated whenever any ISAACA (red or blue) reaches the opposing color's flag for the first time; if it is set to 2, the run continues until the run is terminated by the user (by pressing the "Q" key).

move_order?

There are two ways in which moves can be sampled during an ISAAC run. If **move_order** = 1, then, at the start of each run, a randomly ordered list of red and blue ISAACAs is first set up prior to the start of the actual dynamics loop. During all subsequent passes, ISAACA moves are then determined either by sequencing through the ISAACAs on this list in *fixed order.* If **move_order** = 2, this sequencing occurs in *random order.* See ***ISAACA Move Selection.***

combat_flag?

If **combat_flag** = 0 then there is no limit to the maximum number of possible simultaneous engagements: all enemy ISAACAs within a given ISAACA's fire range will be automatically targeted for engagement. If **combat_flag** = 1 then each side will be able to simultaneously target a certain maximum number of enemy ISAACAs per iteration step. See **R_max_eng_num** and **B_max_eng_num**. See **ISAACA Combat.**

terrain_flag?

The software "flag" **terrain_flag** controls the use of notional terrain and takes on one of two values: 0 or 1. If **terrain_flag** = 1 then terrain will be used (see ***Terrain Parameters***); if **terrain_flag** = 0 then terrain will not be used.

red_frat_flag?

The software "flag" **red_frat_flag** controls the use of fratricide on the red side and takes on one of two values: 0 or 1. If **red_frat_flag** = 1 then red ISAACAs will be able to accidentally target friendly red ISAACAs; if **red_frat_flag** = 0 then fratricide will not be possible. See ***ISAACA Fratricide.***

blue_frat_flag?

The software "flag" **blue_frat_flag** controls the use of fratricide on the red side and takes on one of two values: 0 or 1. If **blue_frat_flag** = 1 then red ISAACAs will be able to accidentally target friendly blue ISAACAs; if **blue_frat_flag** = 0 then fratricide will not be possible. See ***ISAACA Fratricide.***

red_frat_rad

This parameter defines the radius around a targeted enemy ISAACA such that, if the **red_frat_flag**=1 (so that fratricide is possible on the red side), all red ISAACAs located within the "box" defined by this radius become potential victims of fratricide. See ***ISAACA Fratricide*** for more detailed discussion.

blue_frat_rad

This parameter defines the radius around a targeted enemy ISAACA such that, if the **blue_frat_flag**=1 (so that fratricide is possible on the blue side), all blue ISAACAs located within the "box" defined by this radius become potential victims of fratricide. See ***ISAACA Fratricide*** for more detailed discussion.

red_frat_prob

The probability that a red (i.e., friendly) ISAACA is inadvertently "hit" by a shot that was intended to hit a nearby enemy (i.e., blue) ISAACA. See ***ISAACA Fratricide.***

blue_frat_prob

The probability that a blue (i.e., friendly) ISAACA is inadvertently "hit" by a shot that was intended to hit a nearby enemy (i.e., red) ISAACA. See ***ISAACA Fratricide.***

reconst_flag?

The software "flag" **reconst_flag** toggles the reconstitution option. If **reconst_flag** = 1 then reconstitution will be used; if **reconst_flag** = 0 then reconstitution will not be used. See ***Reconstitution*** in ***ISAACA Combat.***

RED_recon_time

If the reconstitution flag **reconst_flag** is set equal to 1, then **RED_recon_time** defines the number of iteration steps following a "hit" (either by blue or, if the fratricide flag **red_frat_flag** is enabled, red ISAACAs) such that if during that time interval a given red ISAACA is not hit again, that ISAACA's state is reconstituted back to *alive.* See ***Reconstitution*** in ***ISAACA Combat.***

BLUE_recon_time

If the reconstitution flag **reconst_flag** is set equal to 1, then **BLUE_recon_time** defines the number of iteration steps following a "hit" (either by red or, if the fratricide flag **blue_frat_flag** is enabled, blue ISAACAs) such that if during that time interval a given blue ISAACA is not hit again, that ISAACA's state is reconstituted back to *alive.* See ***Reconstitution*** in ***ISAACA Combat.***

Statistics Parameters

The **Statistic Parameters** section of the input data file consists of several flags the user can set to regulate the calculation of specific sets of summary statistics for a run. A sample fragment is shown in figure 27. See ***Data Collection.***

Figure 27. Statistics Parameters

```
*****************************
*   STATISTICS PARAMETERS
*****************************
stat_flag?          0
goal_stat_flag?     0
center_mass_flag?   0
interpoint_flag?    0
entropy_flag?       0
cluster_1_flag?     0
cluster_2_flag?     0
neighbors_flag?     0
```

stat_flag?

If **stat_flag** is set to 1 then statistics will be calculated for this run, otherwise no. See ***Data Collection.***

goal_stat_flag?

Assuming that **stat_flag** is set to 1 (so that statistics calculations are enabled for this run), ISAAC will calculate various "proximity to goal" statistics if **goal_stat_flag** = 1, otherwise no. Goal statistics include the number of red and blue ISAACAs within range R=1,2,... 5 of the red and blue flags. See ***Data Collection.***

center_mass_flag?

Assuming that **stat_flag** is set to 1 (so that statistics calculations are enabled for this run), ISAAC will calculate various "center-of-mass" statistics if **center_mass_flag** = 1, otherwise no. Center-of-mass statistics include keeping track of the (x,y) coordinates of the center-of-mass of all red ISAACAs, all blue ISAACAs and all combined forces, as well as distances between the center-of-mass of red and blue ISAACAs and enemy flag. See ***Data Collection.***

interpoint_flag?

Assuming that **stat_flag** is set to 1 (so that statistics calculations are enabled for this run), ISAAC will calculate various "interpoint distance" statistics if **interpoint_flag** = 1, otherwise no. Interpoint distance statistics include keeping track of the distribution of distances between red and red ISAACAs, blue and blue ISAACAs, red and blue ISAACAs, red ISAACAs and blue flag, and blue ISAACAs and red flag. See ***Data Collection.***

entropy_flag?

Assuming that **stat_flag** is set to 1 (so that statistics calculations are enabled for this run), ISAAC will keep track of the approximate spatial entropy of the entire force disposition if **entropy_flag** = 1, otherwise no.

Red, blue and total spatial entropy is calculated using 16 blocks of 20x20 sub-blocks, 64 blocks of 10x10 sub-blocks and 256 5x5 sub-blocks. See ***Data Collection.***

cluster_1_flag?

Assuming that **stat_flag** is set to 1 (so that statistics calculations are enabled for this run), ISAAC will calculate the cluster-size distribution (including ave +/- deviation) at each iteration step assuming an inter-cluster distance criteria of D=1, otherwise no. See ***Data Collection.***

cluster_2_flag?

Assuming that **stat_flag** is set to 1 (so that statistics calculations are enabled for this run), ISAAC will calculate the cluster-size distribution (including ave +/- deviation) at each iteration step assuming an inter-cluster distance criteria of D=2, otherwise no. See ***Data Collection.***

neighbors_flag?

Assuming that **stat_flag** is set to 1 (so that statistics calculations are enabled for this run), ISAAC will calculate various "neighboring ISAACA" statistics, otherwise no. Neighboring ISAACA statistics include averages and deviations for the number of friendly, enemy and total ISAACAs <= range R=1, 2,..., r_s (including red in red, red in blue, blue in blue, blue in red, all in red and all in blue). See ***Data Collection.***

Blue Global Command Parameters

The **Blue Global Command Parameters** section of the input data file consists of flags and variables defining blue's global command personality. A sample fragment appears in figure 28. See ***Global Command*** for a detailed discussion of all variables appearing in this section.

BLUE_global_flag?

If **blue_global_flag** is set to 1 then a global commander will be used for the blue ISAACAs during this run, otherwise there will be no global commander (even if the other variables in this input section have valid entries).

GC_fear_index

The GC's fear index, which is a number between 0 and 1, represents a GC personality-defined tradeoff between wanting to *simultaneously* satisfy two desires: moving LCs closer to the enemy flag and preventing them from encountering too many enemy forces while doing so. If **GC_fear_index** = 0, the GC is effectively fearless of the enemy; if **GC_fear_index** = 1, the GC is maximally fearful of the enemy and

wishes only to keep LCs and their subordinate ISAACAs away from the enemy. See ***Global Command.***

GC_w_alpha

This is relative weight that the global commander assigns to the density of *alive enemy ISAACAs* located within each of the three annular subregions of the battlefield sectors. It is a number between 0 and 1. See ***GC Command of Autonomous LC Movement.***

Figure 28. Blue global command parameters

```
*****************************
* BLUE GLOBAL COMMAND PARAMETERS
*****************************
BLUE_global_flag  1
*
* direction parameters
*
GC_fear_index    1.
GC_w_alpha       1.
GC_w_beta        1.
GC_frac_R[1]     .3
GC_frac_R[2]     .6
GC_w_swath[1]    1.
GC_w_swath[2]    1.
GC_w_swath[3]    1.
*
* help parameters
*
GC_max_red_f     2.5
GC_help_radius   40
GC_h_thresh      .1
GC_rel_h_thresh  1.5
```

GC_w_beta

This is relative weight that the global commander assigns to the density of *injured enemy ISAACAs* located within each of the three annular subregions of the battlefield sectors. It is a number between 0 and 1. See ***GC Command of Autonomous LC Movement.***

GC_frac_R[1]

This defines the size of the *first* of the three annular subregions of the battlefield sectors as the fraction (between 0 and 1) of the distance between the (x,y) coordinates of a given local commander and the way-point corresponding to a given sector. See ***GC Command of Autonomous LC Movement.***

GC_frac_R[2]

This defines the size of the *second* of the three annular subregions of the battlefield sectors as the fraction (between 0 and 1) of the distance between the (x,y) coordinates of a given local commander and the

way-point corresponding to a given sector. Note that ISAAC automatically sets **GC_frac_R[3]** = 1- **GC_frac_R[1]** - **GC_frac_R[2]**. See ***GC Command of Autonomous LC Movement.***

GC_w_swath[1]

This is relative weight that the global commander assigns to the first of the three annular subregions of the battlefield sectors; i.e., the sector that is closest to the (x,y) coordinates of a given local commander. It is a number between 0 and 1. See ***GC Command of Autonomous LC Movement.***

GC_w_swath[2]

This is relative weight that the global commander assigns to the *second* of the three annular subregions of the battlefield sectors. It is a number between 0 and 1. See ***GC Command of Autonomous LC Movement.***

GC_w_swath[3]

This is relative weight that the global commander assigns to the *third* of the three annular subregions of the battlefield sectors. It is a number between 0 and 1. See ***GC Command of Autonomous LC Movement.***

GC_max_red_f

This defines the maximum number of "allowable" enemy ISAACAs (as a fraction of the initial number of friendly subordinates) within the local command area; i.e., the "g" factor defined in the section ***GC Command of LC-LC Interaction.***

GC_help_radius

Defines the size of the box around a given subordinate local commander within which that local commander can possibly assist other local commanders. See ***GC Command of LC-LC Interaction.***

GC_h_thresh

Defines the threshold health state for a local commander such that if that local commander's actual health is *greater than or equal to* **GC_h_thresh**, that local commander can then be ordered to "assist" (i.e., move toward) another nearby local commander. It is a number between 0 and 1. See ***GC Command of LC-LC Interaction.***

GC_rel_h_thresh

Defines the relative fractional health threshold (=Dh_{thresh}) between the health states of local commanders LC_i and LC_j such that if the actual relative fractional health $\Delta_{ij} \geq \Delta h_{thres}$, LC_i can be ordered by the GC to move toward (i.e., "assist") LC_j. See ***GC Command of LC-LC Interaction.***

Red Global Command Parameters

The **Red Global Command Parameters** section of the input data file consists of flags and variables defining the red ISAACA force's global command personality. Except for the fact that they obviously refer to red rather than blue parameters, all entries in this section of the data input file have exactly the same meaning as their blue counterparts, defined above. See ***Global Command*** for a thorough discussion of how all of these variables are used in the internal logic of the program.

Blue Local Command Parameters

The **Blue Local Command Parameters** section of the input data file consists of flags and variables defining blue's local command personality. A sample fragment appears in figure 29. See ***Local Command*** for a detailed discussion of all variables appearing in this section.

Figure 29. Blue local command parameters

```
*****************************
* BLUE LOCAL COMMAND PARAMETERS
*****************************
BLUE_command_flag  1
num_BLUE_comdrs    3
B_patch_type       1
B_patch_flag       2
*
* local commander parameters
*
(1)_B_undr_cmd     15
(1)_B_cmnd_rad     2
(1)_B_SENSOR_rng   7
*
* local commander personality
*
(1)_w1:alive_B     1.000000
(1)_w2:alive_R     5.000000
(1)_w3:injrd_B     1.000000
(1)_w4:injrd_R     5.000000
(1)_w5:B_goal      0.000000
(1)_w6:R_goal      10.000000
*
* local commander constraints
*
(1)_B_THRS_range   4
(1)_ADVANCE_num    0
(1)_CLUSTER_num    0
(1)_COMBAT_num     5
*
* local command weights
*
(1)_B_w_alpha      1.
(1)_B_w_beta       1.
(1)_B_w_delta      1.
(1)_B_w_gamma      1.
*
* global command weights
*
(1)_w_obey_GC_def 1.
(1)_w_help_LC_def .5
```

BLUE_local_flag

If **blue_local_flag** is set to 1 then the local commander option will be used for the blue ISAACAs during this run; If **blue_local_flag** = 0, there will be no local commanders. An important point to remember is that if this flag is set to 0 then all other entries in this section of data input file *must be removed.*

num_BLUE_cmdrs

This defines the number of blue local commanders (between 1 and 10). Note that all entries in this section that follow the subheading "local commander parameters" and begin with **(1)**_____ (i.e., **(1)_B_undr_cmd**, **(1)_B_cmnd_rad**, etc.) refer to parameter entries for the 1st local commander. If **num_BLUE_cmdrs** > 1, then this entire cluster of parameters beginning with **(1)**_____ must be repeated, in the same order, and with appropriate values, for each of the **num_BLUE_cmdrs** local commanders. That is, the start of the parameter cluster for the 2nd local commander (i.e., the entry **(2)_B_undr_cmd**) must immediately follow the last entry for the 1st local commander (**(1)_w_help_LC_def**; see below). The first value for the parameter cluster for the 3rd local commander follows the last value for the parameter cluster for the 2nd local commander, and so on.

B_patch_type

Recall that a local commander's "command area" may be partitioned into either 3-by-3 or 5-by-5 blocks of smaller blocks. **B_patch_type** = 1 partitions this area into 3-by-3 sub-blocks; **B_patch_type** = 2 partitions this area into 5-by-5 sub-blocks. See ***Local Command.***

B_patch_flag

A "flag" that regulates how a local commander breaks a tie between two or more sub-blocks that he calculates will incur the same "penalty" if he orders his subordinate ISAACAs to move toward them. If **B_patch_flag** = 1, the LC chooses a random sub-block out of this same-penalty set. If **B_patch_flag** = 2, the sub-block that is chosen is the one nearest the sub-block that was previously chosen.

(n)_B_undr_cmd

This parameter specifies the number of blue ISAACAs under the command of the nth blue local commander. In the current version of ISAAC, the maximum number of subordinate ISAACAs for one local commander is 100.

(n)_B_cmnd_rad

This defines the "radius" of one of the sub-blocks that the nth blue local commander's local command area is subdivided into. This area is

subdivided either into 3-by-3 subblocks (if **B_patch_type** = 1; see above) or 5-by-5 blocks (if **B_patch_type** = 2). See ***Local Command.***

(n)_B_SENSOR_rng

This defines the n^{th} blue local commander's sensor range. As such, it can be different from the sensor range of the local commander's subordinate ISAACAs.

(n)_w1:alive_B

This defines the 1^{st} component of the n^{th} blue local commander's personality weight vector. This first component represents the relative weight afforded to moving toward *alive blue* (i.e., friendly) ISAACAs. It is a number between 0 and 100. See ***Personality Weight Vector.***

(n)_w2:alive_R

This defines the 2^{nd} component of the n^{th} blue local commander's personality weight vector. This second component represents the relative weight afforded to moving toward *alive red* (i.e., enemy) ISAACAs. It is a number between 0 and 100. See ***Personality Weight Vector.***

(n)_w3:injrd_B

This defines the 3^{rd} component of the n^{th} blue local commander's personality weight vector. This third component represents the relative weight afforded to moving toward *injured blue* (i.e., friendly) ISAACAs. It is a number between 0 and 100. See ***Personality Weight Vector.***

(n)_w4:injrd_R

This defines the 4^{th} component of the n^{th} blue local commander's personality weight vector. This fourth component represents the relative weight afforded to moving toward *injured red* (i.e., enemy) ISAACAs. It is a number between 0 and 100. See ***Personality Weight Vector.***

(n)_w5:B_goal

This defines the 5^{th} component of the n^{th} blue local commander's personality weight vector. This fifth component represents the relative weight afforded to moving toward the *blue* (i.e., friendly) goal. It is a number between 0 and 100. See ***Personality Weight Vector.***

(n)_w6:R_goal

This defines the 6^{th} component of the n^{th} blue local commander's personality weight vector. This sixth component represents the relative weight afforded to moving toward the *red* (i.e., enemy) goal. It is a number between 0 and 100. See ***Personality Weight Vector.***

(n)_B_THRS_range

This defines the n^{th} blue local commander's *threshold range.* The threshold range defines a boxed area surrounding the LC with respect to which that LC computes the numbers of friendly and enemy ISAACAs that play a role in determining what move to make on a given time step. This local decision-making process is described in the section ***ISAACA Adaptability.***

(n)_ADVANCE_num

This defines the n^{th} blue local commander's *advance threshold number,* which represents the minimal number of friendly ISAACAs that must be within the threshold range (=**(n)_B_THRS_range**) for which the LC will continue moving toward the enemy flag (if it has a nonzero weight to do so). See ***Advance Constraint*** in ***ISAACA Adaptability.***

(n)_CLUSTER_num

This defines the n^{th} blue local commander's *cluster threshold number,* which represents a friendly cluster ceiling such that if the LC senses a greater number of friendly forces located within its threshold range (=**(n)_B_THRS_range**), it will temporarily set its personality weights for moving toward friendly ISAACAs (=**(n)_w1:alive_B** and **(n)_w3:injrd_B**) to zero. See ***Cluster Constraint*** in ***ISAACA Adaptability.***

(n)_COMBAT_num

This defines the n^{th} blue local commander's *combat threshold number,* which fixes the local conditions for which the LC will choose to move toward or away from possibly engaging an enemy ISAACA. Intuitively, the idea is that if the LC senses that it has less than a threshold advantage of surrounding forces over enemy forces, it will choose to move away from engaging enemy ISAACAs rather than moving toward (and, thereby, possibly engaging) them. See ***Combat Constraint*** in ***ISAACA Adaptability.***

(n)_B_w_alpha

This defines the 1^{st} of four local command weights that prescribe the relative degree of importance the LC places on various measures of *relative information* contained in each block of sites within his command area. This first component represents the relative weight afforded to the fractional difference between *alive friendly* and *alive enemy* ISAACAs relative to the total number of friendly ISAACAs in each sub-block. See ***Local Command.***

(n)_B_w_beta

This defines the 2^{nd} of four local command weights that prescribe the relative degree of importance the LC places on various measures of

relative information contained in each block of sites within his command area. This second component represents the relative weight afforded to the fractional difference between *alive friendly* and *injured enemy* ISAACAs relative to the total number of friendly ISAACAs in each sub-block. See ***Local Command.***

(n)_B_w_delta

This defines the 3rd of four local command weights that prescribe the relative degree of importance the LC places on various measures of *relative information* contained in each block of sites within his command area. This third component represents the relative weight afforded to the fractional difference between *injured friendly* and *alive enemy* ISAACAs relative to the total number of friendly ISAACAs in each sub-block. See ***Local Command.***

(n)_B_w_gamma

This defines the 4th of four local command weights that prescribe the relative degree of importance the LC places on various measures of *relative information* contained in each block of sites within his command area. This fourth component represents the relative weight afforded to the fractional difference between *injured friendly* and *injured enemy* ISAACAs relative to the total number of friendly ISAACAs in each sub-block. See ***Local Command.***

(n)_w_obey_GC_def

This defines the nth blue local commander's relative weight afforded to obeying his GC's orders. It is a number between 0 and 1. See ***LC Response to GC Commands.***

(n)_w_help_LC_def

This defines the nth blue local commander's relative weight afforded to moving toward and "assisting" another LC. It is a number between 0 and 1. See ***LC Response to GC Commands.***

Red Local Command Parameters

The **Red Local Command Parameters** section of the input data file consists of flags and variables defining the red ISAACA force's local command personality. Except for the fact that they obviously refer to red rather than blue parameters, all entries in this section of the data input file have exactly the same meaning as their blue counterparts, defined above. See ***Local Command*** for a thorough discussion of how all of these variables are used in the internal logic of the program.

Blue ISAACA Parameters

The **Blue ISAACA Parameters** section of the input data file consists of flags and variables defining the blue ISAACAs. A sample fragment appears in figure 30. See ***Overview of ISAAC*** for a detailed discussion of all variables appearing in this section.

num_blues

This defines the total number of blue ISAACAs. The current version of ISAAC limits this number to 400 or less.

squads

This defines the total number of blue squads. This is a number between 1 and a maximum of 10.

num_per_squad

This defines the number of blue ISAACAs per squad for each of the 10 possible squads. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). There is an internal check on the sum of the squad sizes that is performed by ISAAC to prevent possible overflow conditions.

M_range

This defines the *movement range*, r_M, for each of the 10 possible blue ISAACA squads. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). In the current version of ISAAC, r_M can either be set to equal 1 (meaning that ISAACAs choose there move from within a 3-by-3 box surrounding their current position) or r_M can be set to equal 2 (meaning that ISAACAs choose there move from within a 5-by-5 box surrounding their current position). See ***ISAACA Move Selection.***

personality

This software "flag" specifies how the blue ISAACA's personality weight vector $\vec{w}$will be determined. If **personality** = 1, then the components of $\vec{w}$ are defined explicitly by the appropriate parameter entries that appear below (see entries **w1_a:B_alive_B** through **w6_i:B_R_goal**). If **personality** = 2, then the components of $\vec{w}$are randomly assigned. In this case, each blue ISAACA is assigned a different random weight vector.

Figure 30. Blue ISAACA Parameters

```
*****************************
* BLUE ISAACA PARAMETERS
*****************************
num_blues          100
squads             1
num_per_squad      100  15  23  0  0  0  0  0  0  0
M_RANGE            1    1   1   2  2  2  2  2  2  2
personality        1
*
* ALIVE personality weights
*
w1_a:B_alive_B      10.0  10.0  76.0  76.0  76.0  76.0  76.0  76.0  76.0  76.0
w2_a:B_alive_R      40.0  99.0  61.0  61.0  61.0  61.0  61.0  61.0  61.0  61.0
w3_a:B_injrd_B      10.0   0.0   0.0  -4.0  -4.0  -4.0  -4.0  -4.0  -4.0  -4.0
w4_a:B_injrd_R      40.5  99.5  76.0  76.0  99.5  99.5  76.0  76.0  76.0  76.0
w5_a:B_B_goal        0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
w6_a:B_R_goal       50.0  50.0  47.0  47.0  47.0  47.0  47.0  47.0  47.0  47.0
*
* INJURED personality weights
*
w1_i:B_alive_B      10.0  76.0  76.0  76.0  76.0  76.0  76.0  76.0  76.0  76.0
w2_i:B_alive_R      40.0  61.0  61.0  61.0  61.0  61.0  61.0  61.0  61.0  61.0
w3_i:B_injrd_B      10.0  -4.0  -4.0  -4.0  -4.0  -4.0  -4.0  -4.0  -4.0  -4.0
w4_i:B_injrd_R      40.5  99.5  76.0  76.0  99.5  99.5  76.0  76.0  76.0  76.0
w5_i:B_B_goal        0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
w6_i:B_R_goal       50.0  47.0  47.0  47.0  47.0  47.0  47.0  47.0  47.0  47.0
*
* ISAACA-LC weights
*
w7:B_loc_comdr     1.000000
w8:B_loc_goal      1.000000
*
* defense parameters
*
defense_flag       1
alive_strength     1  1  1  1  1  1  1  1  1  1
injured_strength   1  1  1  1  1  1  1  1  1  1
*
* sensor/fire ranges
*
S_RANGE             5  8  7  3  3  3  3  3  3  3
F_RANGE             3  4  3  3  3  3  3  3  3  3
*
* communications
*
COMM_flag          0
COMM_range         0
COMM_weight        0.000000
*
* movement constraints
*
movement_flag      1
T_RANGE            3    4    3    7    7    7    7    7    7    7
A:ADVANCE_num      0    0    0    1    1    1    1    1    1    1
A:CLUSTER_num      5    5    7   16   16   16   16   16   16   16
A:COMBAT_num       3   -5   -1   -1   -1   -1   -1   -1   -1   -1
I:ADVANCE_num      0    9    9    9    9    9    9    9    9    9
I:CLUSTER_num      8    9    9    9    9    9    9    9    9    9
I:COMBAT_num       5   -2  -16  -16  -16  -16  -16  -16  -16  -16
T_RANGE_(m,M)      1,4
A:ADV_(m,M)        0,4
A:CLUS_(m,M)       1,8
A:COMB_(m,M)       -3,3
I:ADV_(m,M)        0,4
I:CLUS_(m,M)       1,8
I:COMB_(m,M)       -3,3
A:B_B_min_dist      0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
A:B_R_min_dist      0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
A:B_B_goal_min      0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
I:B_B_min_dist      0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
I:B_R_min_dist      0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
I:B_B_goal_min      0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
*
* combat/engagement
*
shot_prob          0.005 0.010 0.010 0.005 0.005 0.005 0.005 0.005 0.005 0.005
B_max_eng_num      5     3     2     2     2     2     2     2     2     2
```

w1_a:B_alive_B

This defines the 1st component of the alive blue ISAACA's personality weight vector. This first component represents the relative weight afforded by *alive blue* ISAACAs to moving toward *alive blue* (i.e., friendly) ISAACAs. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Personality Weight Vector.***

w2_a:B_alive_R

This defines the 2nd component of the alive blue ISAACA's personality weight vector. This second component represents the relative weight afforded by *alive blue* ISAACAs to moving toward *alive red* (i.e., enemy) ISAACAs. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Personality Weight Vector.***

w3_a:B_injrd_B

This defines the 3rd component of the alive blue ISAACA's personality weight vector. This third component represents the relative weight afforded by *alive blue* ISAACAs to moving toward *injured blue* (i.e., friendly) ISAACAs. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Personality Weight Vector.***

w4_a:B_injrd_R

This defines the 4th component of the alive blue ISAACA's personality weight vector. This fourth component represents the relative weight afforded by *alive blue* ISAACAs to moving toward *injured red* (i.e., enemy) ISAACAs. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10

squads (as defined by the **squads** parameter above). See ***Personality Weight Vector***.

w5_a:B_B_goal

This defines the 5th component of the alive blue ISAACA's personality weight vector. This fifth component represents the relative weight afforded by *alive blue* ISAACAs to moving toward the *blue* (i.e., friendly) goal. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Personality Weight Vector***.

w6_a:B_R_goal

This defines the 6th component of the alive blue ISAACA's personality weight vector. This sixth component represents the relative weight afforded by *alive blue* ISAACAs to moving toward the *red* (i.e., enemy) goal. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Personality Weight Vector***.

w1_i:B_alive_B

This defines the 1st component of the injured blue ISAACA's personality weight vector. This first component represents the relative weight afforded by *injured blue* ISAACAs to moving toward *alive blue* (i.e., friendly) ISAACAs. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Personality Weight Vector***.

w2_i:B_alive_R

This defines the 2nd component of the injured blue ISAACA's personality weight vector. This second component represents the relative weight afforded by *injured blue* ISAACAs to moving toward *alive red* (i.e., enemy) ISAACAs. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10

squads (as defined by the **squads** parameter above). See ***Personality Weight Vector.***

w3_i:B_injrd_B

This defines the 3rd component of the injured blue ISAACA's personality weight vector. This third component represents the relative weight afforded by *injured blue* ISAACAs to moving toward *injured blue* (i.e., friendly) ISAACAs. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Personality Weight Vector.***

w4_i:B_injrd_R

This defines the 4th component of the injured blue ISAACA's personality weight vector. This fourth component represents the relative weight afforded by *injured blue* ISAACAs to moving toward *injured red* (i.e., enemy) ISAACAs. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Personality Weight Vector.***

w5_i:B_B_goal

This defines the 5th component of the injured blue ISAACA's personality weight vector. This fifth component represents the relative weight afforded by *injured blue* ISAACAs to moving toward the *blue* (i.e., friendly) goal. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as {10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Personality Weight Vector.***

w6_i:B_R_goal

This defines the 6th component of the injured blue ISAACA's personality weight vector. This sixth component represents the relative weight afforded by *injured blue* ISAACAs to moving toward the *red* (i.e., enemy) goal. It is a number between 0 and 100. (Recall that only the relative values among all six components matter here: the set {1,2,3,4,5,6} represents exactly the same set of weights as

{10,20,30,40,50,60}, as far as ISAAC is concerned.) Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Personality Weight Vector.***

w7:B_loc_comdr

If the blue local command option is enabled (i.e., if the parameter **blue_local_flag** is set equal to 1), and a given blue ISAACA is under the command of blue local commander, **w7:B_loc_comdr** effectively acts as the 7th component of that blue ISAACA's personality weight vector. This seventh component defines the relative weight afforded by a subordinate blue ISAACA to *staying close to* its local commander. It is a number between 0 and 1. See ***Subordinate ISAACAs*** under ***Local Command.***

w8:B_loc_goal

If the blue local command option is enabled (i.e., if the parameter **blue_local_flag** is set equal to 1), and a given blue ISAACA is under the command of blue local commander, **w8:B_loc_goal** effectively acts as the 8th component of that blue ISAACA's personality weight vector. This seventh component defines the relative weight afforded by a subordinate blue ISAACA to *obeying the orders* issued by its local commander. It is a number between 0 and 1. See ***Subordinate ISAACAs*** under ***Local Command.***

defense_flag

A software "flag" that regulates the notional defense option. If **defense_flag** = 1, the defense option is enabled (and defined by the parameters **alive_strength** and **injured_strength** below); if **defense_flag** = 0, the defense option is disabled.

alive_strength

If the notional defense option is enabled (i.e., if **defense_flag** = 1), then **alive_strength** defines the defensive strength of alive blue ISAACAs. The value of this parameter equals the number of "hits" (either by enemy or, if the fratricide option is enabled by setting **blue_frat_flag** = 1, friendly fire) that it takes to degrade an alive blue ISAACA to an injured state. The minimal (and default) value is 1. Setting alive_strength to a large positive number effectively renders blue ISAACAs impervious to fire. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Notional Defense*** under ***ISAACA Combat.***

injured strength

If the notional defense option is enabled (i.e., if **defense_flag** = 1), then **injured_strength** defines the defensive strength of injured blue

ISAACAs. The value of this parameter equals the number of "hits" (either by enemy or, if the fratricide option is enabled by setting **blue_frat_flag** = 1, friendly fire) that it takes to kill an already injured blue ISAACA. The minimal (and default) value is 1. Setting alive_strength to a large positive number effectively renders injured blue ISAACAs impervious to fire. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Notional Defense*** under ***ISAACA Combat.***

S_range

This defines the *sensor range,* r_S, for each of the 10 possible blue ISAACA squads. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). **S_range** can take on the value zero (in which case the ISAACA "senses" nothing around itself) and any positive integer value. See ***ISAACA Ranges.***

F_range

This defines the *fire range,* r_F, for each of the 10 possible blue ISAACA squads. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). **F_range** can take on the value zero (in which case the ISAACA is unable to "shoot" at anything) and any positive integer value. See ***ISAACA Ranges.***

COMM_flag

This software "flag" regulates the communications option for blue ISAACAs. If **COMM_flag** = 1, the communications option is enabled (and defined by the parameters **COMM_range** and **COMM_weight** below); if **COMM_flag** = 0, the communications option is disabled. See ***Communication.***

COMM_range

If the communications option is enabled (i.e., if **COMM_flag** = 1), then **COMM_range** defines the range of blue ISAACA communications. See ***Communication.***

COMM_weight

If the communications option is enabled (i.e., if **COMM_flag** = 1), then **COMM_weight** defines the relative weight afforded by blue ISAACAs to using information communicated to them by other blue ISAACAs (within a communications range **COMM_range** of their position) in calculating their move selection penalty function. **COMM_weight** is typically assigned a real value between 0 and 1, though values greater than 1 can also be used to prescribe scenarios where blue ISAACAs give

greater weight to communicated information then to information existing within their own sensor field. See ***Communication.***

movement_flag

This software "flag" controls the use of constraint thresholds (see **ISAACA Adaptability**). If **movement_flag** = 1, then additional constraints (defined by the next seven parameter entries **T_RANGE** through **I:COMBAT_num**) will be used. If **movement_flag** = 0, no additional constraints will be used (and the next seven entries will therefore be ignored for this run).

T_range

This defines the blue ISAACA's *threshold range,* r_T; it can be assigned any positive integer value. The threshold range defines a boxed area surrounding the ISAACA with respect to which that ISAACA computes the numbers of friendly and enemy ISAACAs that play a role in determining what move to make on a given time step. This local decision-making process is described in the section ***ISAACA Adaptability.***

A:ADVANCE_num

This defines the alive blue ISAACA's *advance threshold number,* which represents the minimal number of friendly ISAACAs that must be within the threshold range (= **T_range**) for which the blue ISAACA will continue moving toward the enemy flag (if it has a nonzero default weight to do so). Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Advance Constraint*** in ***ISAACA Adaptability.***

A:CLUSTER_num

This defines the alive blue ISAACA's *cluster threshold number,* which represents a friendly cluster ceiling such that if the blue ISAACA senses a greater number of friendly forces located within its threshold range (= **T_range**), it will temporarily set its personality weights for moving toward friendly ISAACAs (**w1_a:B_alive_B** and **w3_a:B_injrd_B** if the blue ISAACA is alive, and **w1_i:B_alive_B** and **w3_i:B_injrd_B** if the blue ISAACA is injured) to zero. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Cluster Constraint*** in ***ISAACA Adaptability.***

A:COMBAT_num

This defines the alive blue ISAACA's *combat threshold number,* which fixes the local conditions for which the blue ISAACA will choose to move toward or away from possibly engaging an enemy ISAACA. Intuitively, the idea is that if the blue ISAACA senses that it has less than a threshold advantage of surrounding forces over enemy forces, it will choose to move away from engaging enemy ISAACAs rather than

moving toward (and, thereby, possibly engaging) them. The value of **A:COMBAT_num** must be a (positive or negative) integer. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Combat Constraint*** in ***ISAACA Adaptability***.

I:ADVANCE_num

This defines the injured blue ISAACA's *advance threshold number,* which represents the minimal number of friendly ISAACAs that must be within the threshold range (= **T_range**) for which the blue ISAACA will continue moving toward the enemy flag (if it has a nonzero default weight to do so). Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Advance Constraint*** in ***ISAACA Adaptability***.

I:CLUSTER_num

This defines the injured blue ISAACA's *cluster threshold number,* which represents a friendly cluster ceiling such that if the blue ISAACA senses a greater number of friendly forces located within its threshold range (= **T_range**), it will temporarily set its personality weights for moving toward friendly ISAACAs (**w1_a:B_alive_B** and **w3_a:B_injrd_B** if the blue ISAACA is alive, and **w1_i:B_alive_B** and **w3_i:B_injrd_B** if the blue ISAACA is injured) to zero. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Cluster Constraint*** in ***ISAACA Adaptability***.

I:COMBAT_num

This defines the injured blue ISAACA's *combat threshold number,* which fixes the local conditions for which the blue ISAACA will choose to move toward or away from possibly engaging an enemy ISAACA. Intuitively, the idea is that if the blue ISAACA senses that it has less than a threshold advantage of surrounding forces over enemy forces, it will choose to move away from engaging enemy ISAACAs rather than moving toward (and, thereby, possibly engaging) them. The value of **A:COMBAT_num** must be a (positive or negative) integer. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Combat Constraint*** in ***ISAACA Adaptability***.

T_RANGE_(m,M)

These two values (m,M) define the lower (=m) and upper (=M) limits of the interval of values within which the blue ISAACA's *threshold range,* r_T, will be assigned a random positive integer value. These parameter settings are used only if (1) the personality flag **personality** is set equal to one (so that the blue ISAACAs are assigned random personality weight vectors), and (2) the movement flag **movement_flag** is set equal

to 2 (so that blue ISAACA personalities are augmented by additional constraints). The threshold range defines a boxed area surrounding the ISAACA with respect to which that ISAACA computes the numbers of friendly and enemy ISAACAs that play a role in determining what move to make on a given time step. This local decision-making process is described in the section ***ISAACA Adaptability***.

A:ADV_(m,M)

These two values (m,M) define the lower (=m) and upper (=M) limits of the interval of values within which the alive blue ISAACA's *advance threshold number* will be assigned a random positive integer value. These parameter settings are used only if (1) the personality flag **personality** is set equal to one (so that the blue ISAACAs are assigned random personality weight vectors), and (2) the movement flag **movement_flag** is set equal to 2 (so that blue ISAACA personalities are augmented by additional constraints). The advance threshold number represents the minimal number of friendly ISAACAs that must be within the threshold range (= **T_range**) for which the blue ISAACA will continue moving toward the enemy flag (if it has a nonzero default weight to do so). See ***Advance Constraint*** in ***ISAACA Adaptability***.

A:CLUS_(m,M)

These two values (m,M) define the lower (=m) and upper (=M) limits of the interval of values within which the alive blue ISAACA's *cluster threshold number* will be assigned a random positive integer value. These parameter settings are used only if (1) the personality flag **personality** is set equal to one (so that the blue ISAACAs are assigned random personality weight vectors), and (2) the movement flag **movement_flag** is set equal to 2 (so that blue ISAACA personalities are augmented by additional constraints). The cluster threshold number represents a friendly cluster ceiling such that if the blue ISAACA senses a greater number of friendly forces located within its threshold range (= **T_range**), it will temporarily set its personality weights for moving toward friendly ISAACAs (**w1_a:B_alive_B** and **w3_a:B_injrd_B** if the blue ISAACA is alive, and **w1_i:B_alive_B** and **w3_i:B_injrd_B** if the blue ISAACA is injured) to zero. See ***Cluster Constraint*** in ***ISAACA Adaptability***.

A:COMB_(m,M)

These two values (m,M) define the lower (=m) and upper (=M) limits of the interval of values within which the alive blue ISAACA's *combat threshold number* will be assigned a random integer value. These parameter settings are used only if (1) the personality flag **personality** is set equal to one (so that the blue ISAACAs are assigned random personality weight vectors), and (2) the movement flag **movement_flag** is set equal to 2 (so that blue ISAACA personalities are augmented by additional constraints). , which fixes the local conditions for which the

blue ISAACA will choose to move toward or away from possibly engaging an enemy ISAACA. Intuitively, the idea is that if the blue ISAACA senses that it has less than a threshold advantage of surrounding forces over enemy forces, it will choose to move away from engaging enemy ISAACAs rather than moving toward (and, thereby, possibly engaging) them. The values of 'm' and 'M' must be a (positive or negative) integers. See ***Combat Constraint*** in ***ISAACA Adaptability***.

I:ADV_(m,M)

These two values (m,M) define the lower (=m) and upper (=M) limits of the interval of values within which the injured blue ISAACA's *advance threshold number* will be assigned a random positive integer value. These parameter settings are used only if (1) the personality flag **personality** is set equal to one (so that the blue ISAACAs are assigned random personality weight vectors), and (2) the movement flag **movement_flag** is set equal to 2 (so that blue ISAACA personalities are augmented by additional constraints). The advance threshold number represents the minimal number of friendly ISAACAs that must be within the threshold range (= **T_range**) for which the blue ISAACA will continue moving toward the enemy flag (if it has a nonzero default weight to do so). See ***Advance Constraint*** in ***ISAACA Adaptability***.

I:CLUS_(m,M)

These two values (m,M) define the lower (=m) and upper (=M) limits of the interval of values within which the injured blue ISAACA's *cluster threshold number* will be assigned a random positive integer value. These parameter settings are used only if (1) the personality flag **personality** is set equal to one (so that the blue ISAACAs are assigned random personality weight vectors), and (2) the movement flag **movement_flag** is set equal to 2 (so that blue ISAACA personalities are augmented by additional constraints). The cluster threshold number represents a friendly cluster ceiling such that if the blue ISAACA senses a greater number of friendly forces located within its threshold range (= **T_range**), it will temporarily set its personality weights for moving toward friendly ISAACAs (**w1_a:B_alive_B** and **w3_a:B_injrd_B** if the blue ISAACA is alive, and **w1_i:B_alive_B** and **w3_i:B_injrd_B** if the blue ISAACA is injured) to zero. See ***Cluster Constraint*** in ***ISAACA Adaptability***.

I:COMB_(m,M)

These two values (m,M) define the lower (=m) and upper (=M) limits of the interval of values within which the injured blue ISAACA's *combat threshold number* will be assigned a random integer value. These parameter settings are used only if (1) the personality flag **personality** is set equal to one (so that the blue ISAACAs are assigned random personality weight vectors), and (2) the movement flag **movement_flag** is set equal to 2 (so that blue ISAACA personalities are augmented by

additional constraints). , which fixes the local conditions for which the blue ISAACA will choose to move toward or away from possibly engaging an enemy ISAACA. Intuitively, the idea is that if the blue ISAACA senses that it has less than a threshold advantage of surrounding forces over enemy forces, it will choose to move away from engaging enemy ISAACAs rather than moving toward (and, thereby, possibly engaging) them. The values of 'm' and 'M' must be a (positive or negative) integers. See ***Combat Constraint*** in ***ISAACA Adaptability***.

A:B_B_min_dist

This defines the alive blue ISAACA's *blue-blue minimum distance constraint,* which represents the minimal distance that an alive blue ISAACA wants to maintain away from each blue (i.e., friendly) ISAACA in its sensor field. **A:B_B_min_dist** must be set equal to either zero (for no constraint) or to some positive integer value. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Minimum Local-Distance Constraints*** in ***ISAACA Adaptability***.

A:B_R_min_dist

This defines the alive blue ISAACA's *blue-red minimum distance constraint,* which represents the minimal distance that an alive blue ISAACA wants to maintain away from each red (i.e., enemy) ISAACA in its sensor field. **A:B_R_min_dist** must be set equal to either zero (for no constraint) or to some positive integer value. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Minimum Local-Distance Constraints*** in ***ISAACA Adaptability***.

A:B_B_goal_min

This defines the alive blue ISAACA's *blue/blue-goal minimum distance constraint,* which represents the minimal distance that an alive blue ISAACA wants to maintain away from the blue (i.e., friendly) goal. **A:B_B_goal_min** must be set equal to either zero (for no constraint) or to some positive integer value. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Minimum Local-Distance Constraints*** in ***ISAACA Adaptability***.

I:B_B_min_dist

This defines the injured blue ISAACA's *blue-blue minimum distance constraint,* which represents the minimal distance that an alive blue ISAACA wants to maintain away from each blue (i.e., friendly) ISAACA in its sensor field. **I:B_B_min_dist** must be set equal to either zero (for no constraint) or to some positive integer value. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as

defined by the **squads** parameter above). See ***Minimum Local-Distance Constraints*** in ***ISAACA Adaptability.***

I:B_R_min_dist

This defines the injured blue ISAACA's *blue-red minimum distance constraint,* which represents the minimal distance that an alive blue ISAACA wants to maintain away from each red (i.e., enemy) ISAACA in its sensor field. **I:B_R_min_dist** must be set equal to either zero (for no constraint) or to some positive integer value. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Minimum Local-Distance Constraints*** in ***ISAACA Adaptability.***

I:B_B_goal_min

This defines the injured blue ISAACA's *blue/blue-goal minimum distance constraint,* which represents the minimal distance that an alive blue ISAACA wants to maintain away from the blue (i.e., friendly) goal. **I:B_B_goal_min** must be set equal to either zero (for no constraint) or to some positive integer value. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***Minimum Local-Distance Constraints*** in ***ISAACA Adaptability.***

shot_prob

This defines the blue ISAACA's *single-shot probability,* p_{ss}, which represents the probability that a targeted enemy ISAACA is "hit." It is a number between 0 and 1. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***ISAACA Combat.***

B_max_eng_num

This defines the maximum number of simultaneously targetable red (i.e., enemy) ISAACAs by a blue ISAACA. If the number of targetable enemy ISAACAs within a blue ISAACA's sensor field is less than **B_max_eng_num**, the value of **B_max_eng_num** has no effect. If there are a greater number of targetable enemy ISAACAs within a blue ISAACA's sensor field than **B_max_eng_num**, then **B_max_eng_num** of them will be randomly targeted. Note that all 10 entries *must* appear in the input file, even if there are less than 10 squads (as defined by the **squads** parameter above). See ***ISAACA Combat.***

Red ISAACA Parameters

The **Red ISAACA Parameters** section of the input data file consists of flags and variables defining red ISAACAs. Except for the fact that they obviously refer to red rather than blue parameters, all entries in this

section of the data input file have exactly the same meaning as their blue counterparts, defined in **Blue ISAACA Parameters** section above.

Terrain Parameters

The last section of the data input file contains parameters defining the notional terrain to be used for a given run (see figure 31). Note that this section consists of as many triplets of the form '**(n)_size**, **(n)_center_x**, and **(n)_center_y**' (where n = 1, 2, 3, and so on) as there are individual terrain blocks. For a discussion of how notional terrain is incorporated into ISAAC, see ***Notional Terrain.***

Figure 31. Terrain parameters

```
(1)_size    2
(1)_center_x  32
(1)_center_y  50
(2)_size    2
(2)_center_x  44
(2)_center_y  44
(3)_size    2
(3)_center_x  58
(3)_center_y  50
(4)_size    2
(4)_center_x  54
(4)_center_y  58
(5)_size    4
(5)_center_x  42
(5)_center_y  54
```

(n)_size

This defines the linear side dimension of the n^{th} terrain block. It can be assigned any positive integer value.

(n)_center_x

This defines the x-coordinate of the center of the n^{th} terrain block (x=1 defines the extreme left-hand-side of the notional battlefield). **(n)_center_x** can be assigned any positive integer value.

(n)_center_y

This defines the y-coordinate of the center of the n^{th} terrain block (y=1 defines the bottom edge of the notional battlefield). **(n)_center_y** can be assigned any positive integer value.

Sample Graphics Display

Once the user has selected the form of output (to screen, file or both; see figure 24), ISAAC runs through its initialization routine and displays the main graphics page showing the disposition of forces on the notional battlefield.

A sample graphics page is shown in figure 32. Observe that there are five main parts to the display:

- A ***battlefield region***, located at the center of the display, which contains a graphic representation of activity taking place on the notional battlefield. All ISAACAs are color coded: alive red ISAACAs are colored dark red, injured red ISAACAs are light red, alive blue ISAACAs are dark blue, injured blue ISAACAs are light blue and the red and blue flags appear as a solid large dark red and dark blue disks, respectively. A "time" counter (that shows the current iteration step) appears at the center bottom of the battlefield region.
- A ***banner-display region***, located at the top of the battlefield, which identifies the program and release version, the data file that is currently open (on the left) and the size of the notional battlefield (on the right).
- A ***red ISAACA data region***, appearing to the left of the battlefield, which contains information summarizing the red ISAACA force (see below)
- A ***blue ISAACA data region***, appearing to the left of the battlefield, which contains information summarizing the blue ISAACA force (see below)
- A ***"hot-key" menu region***, appearing at the bottom of the battlefield, which contains a menu of "hot keys" that the user can use to interrupt a run at any time to perform a variety of functions (see below)

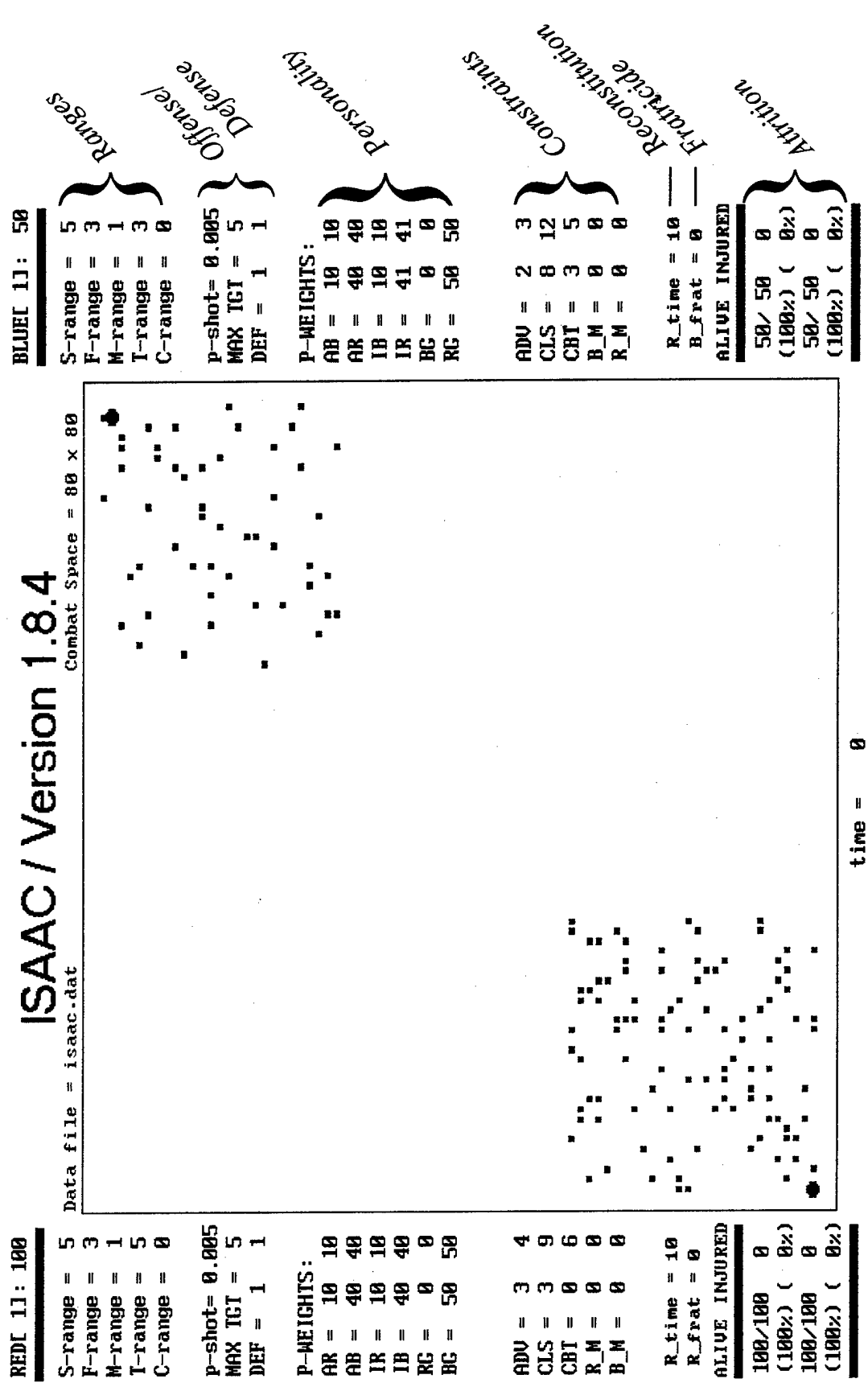


Figure 32. ISAAC's main graphics display

ISAACA Data Regions

The ISAACA data regions, located to the left and right of the battlefield shown in figure 32, summarize the parameters that define the red and blue ISAACAs. As the labels to the right of the blue ISAACA data region show, data appearing in these regions are generally clustered in groups according to the kind of information they represent. From top to bottom, there are eight such data fields:

- Squad Identifier
- Range Parameters
- Offensive/Defensive Parameters
- Personality Weight Vector
- Constraint Parameters
- Reconstitution
- Fratricide
- Attrition

Squad Identifier

The squad identifier field is the first line above the colored bar that appears at the top of the red and blue ISAACA data regions.

In figure 32, the red squad identifier is the line **RED[1]: 100**, while the blue squad identifier is the **BLUE[1]: 50**. This field conveys three pieces of information: (1) the color of the side (red ISAACA parameter values appear always on the left; blue ISAACA parameter values on the right); (2) the number of the particular squad (between 1 and 10) whose parameter values currently appear on screen; and (3) the size of that squad (i.e., the number of ISAACAs that initially made up that squad). Assuming that one or both sides consist of more than one squad, sets of parameter values corresponding to other squads may be displayed at any time during a run by pressing either the '**B**' (for **B**lue) or '**E**' (for r**E**d) "hot-keys"; see ***On-the-Fly Parameter Changes.***

Range Parameters

The second data field consists of the set of ISAACA range parameters (see the section ***ISAACA Ranges*** for a detailed discussion):

- **S-range** = sensor range, r_S
- **F-range** = fire range, r_F
- **M-range** = movement range, r_M

- **T-range** = threshold range, r_T
- **C-range** = communications range, r_C

Offensive/Defensive Parameters

The third data field consists of parameters defining an ISAACA's notional offensive and defensive capabilities (see the section ***ISAACA Combat*** for a detailed discussion):

- **p-shot** = single-shot probability
- **MAX TGT** = maximum number of simultaneously targetable enemy ISAACAs
- **DEF** = defensive strength

Note that **DEF** contains data in two columns: the first column defines the notional defensive strength for *alive* ISAACAs; the second column defines the defensive strength for *injured* ISAACAs.

Personality Weight Vector

The fourth data field (labeled **P-WEIGHTS** in figure 32) consists of parameters defining an ISAACA's personality weight vector (see the section ***ISAACA Personality*** for a detailed discussion):

- **AR** = relative weight for moving toward **A**live **R**ed ISAACAs
- **AB** = relative weight for moving toward **A**live **B**lue ISAACAs
- **IR** = relative weight for moving toward **I**njured **R**ed ISAACAs
- **IB** = relative weight for moving toward **I**njured **B**lue ISAACAs
- **RG** = relative weight for moving toward **R**ed **G**oal
- **BG** = relative weight for moving toward **B**lue **G**oal

Note that, as for the notional defense parameter **DEF** (see above), the personality data field contains data in two columns: the first column (on the left) defines a given component of the personality weight vector for *alive* ISAACAs; the second column (on the right) defines the defensive strength for *injured* ISAACAs.

Constraint Parameters

The fifth data field consists of the constraint parameters that augment an ISAACAs default personality (see the section ***ISAACA Adaptability*** for a detailed discussion):

- **ADV** = advance number threshold
- **CLS** = cluster threshold
- **CBT** = combat threshold
- **B_M** = Minimal distance from Blue ISAACAs
- **R_M** = Minimal distance from Red ISAACAs

If the user constrains either red or blue ISAACAs from maintaining a prescribed minimum distance from own goal, one additional parameter may appear in this data field: **G_M** = Minimal distance from own Goal).

Note that, as for the notional defense parameter **DEF** and parameters appearing in the personality field (see above), the constraint parameters data field contains data in two columns: the first column (on the left) defines a given component of the personality weight vector for *alive* ISAACAs; the second column (on the right) defines the defensive strength for *injured* ISAACAs.

Reconstitution

The sixth data field consists of a single entry, **R_time**, that defines the *reconstitution time* for red or blue ISAACAs.

Recall that if the reconstitution flag **reconst_flag** is set equal to 1 (see ***General Battle Parameters*** of ***Contents of Data Input File***), then the reconstitution time defines the number of iteration steps following a "hit" (either by enemy or friendly ISAACAs) such that if during that time interval a given ISAACA is not hit again, that ISAACA's state is reconstituted back to *alive*. See ***Reconstitution in ISAACA Combat***.

Fratricide

The seventh data field consists of a single entry – either **R_frat** for red ISAACAs or **B_frat** for blue ISAACAs – that displays the cumulative number of fratricide hits that have occurred up to the current iteration step.

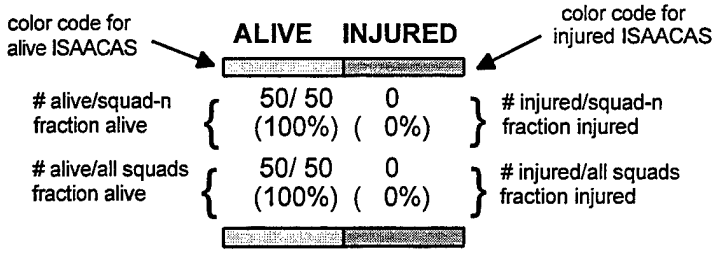
Note that the red and blue fratricide data fields appear in the display only if the appropriate software flag (**red_frat_flag** for red and **red_frat_flag** for blue; see ***General Battle Parameters*** of ***Contents of Data Input File***) has been set to "turn on" either the red or blue fratricide option.

Attrition

The eighth, and bottom-most, data field, is essentially a "tally-board" that keeps track of the remaining number (in gross and relative terms)

of either red or blue ISAACAs. A sample fragment, with explanatory text, is reproduced in figure 33 below.

Figure 33. The Attrition data field of ISAAC's main graphics display



"Hot-Key" Menu

The colored words at the bottom of the battlefield comprise a menu of (black-colored) "hot keys"[17] that the user can use to interrupt a run at any time to perform a variety of functions. There are sixteen such functions, accessed by the following keys (and defined according to the order in which they appear, left to right, on screen):

- **"A"** (for StAts): toggles the calculation of statistics (see ***Data Collection***). What specific data are accumulated depends on what statistic "flags" are set in ISAAC's data input file (see ***Statistics Parameters*** in ***Contents of Input Data File***).
- **"B"** (for Blue): increments the squad number (if blue consists of more than one squad) and displays the squad's defining parameters in the blue ISAACA data region (to the right of the notional battlefield).
- **"C"** (for Command): toggles various views of the local and/or global command structures for both red and blue ISAACAs. Assuming red and blue ISAACAs have both global and local commanders, then by default no command structure is initially shown on-screen. However, successive presses of the "C" key has the following effects:
  - 1st press: highlights each of the local commanders, using yellow for red and white for blue

[17] On a computer screen, the "hot-keys" are actually highlighted yellow instead of black. Keep in mind that because the colors white and black (the "background" color on the computer screen) have, for printing purposes, been reversed, not all colors appearing in graphics reproductions in this report and their actual computer screen counterparts match exactly.

- 2nd press: highlights each of the local commanders + draws the appropriate "boxed" local command area around each local commander (see ***Local Command***)
- 3rd press: highlights each of the local commanders + draws links between each local commander and each of its subordinate ISAACAs
- 4th press: highlights each of the local commanders + highlights each local commander's subordinates + draws the appropriate "boxed" local command area around each local commander
- 5th press: highlights each of the local commanders + draws links between each local commander and each of its subordinate ISAACAs + draws the appropriate "boxed" local command area around each local commander
- 6th press: highlights each of the local commanders + highlights each local commander's subordinates + draws links between each local commander and each of its subordinate ISAACAs
- 7th press: same as 6th strike + draws the appropriate "boxed" local command area around each local commander
- 8th press: same as 7th + draws links between each local commander (explicitly showing their "connectivity" via the global commander)

- **"D"** (for **D**ata): toggles an on-screen prompt to interactively run another ISAAC data input file.
- **"E"** (for R**E**d): increments the squad number (if red consists of more than one squad) and displays the squad's defining parameters in the red ISAACA data region (to the left of the notional battlefield).
- **"F"** (for **F**ast): enables the *fast* run mode, in which the screen is updated as rapidly as possible. For slowly single-stepping through a run, use the "S" hot-key (see below).
- **"H"** (for Snaps**H**ot): toggles an on-screen prompt to name an ***.out** file for storing a "snapshot" view of the current battle state of the system. This ***.out** file can then be "played-back" (i.e., re-displayed) by pressing the "P" hot-key (see below).
- **"L"** (for C**L**ose): closes all statistics files and stops all further data collection (see ***Data Collection***). Data collection may be restarted by pressing "A" hot-key (see above).

- **"N"** (for RaNd): reinitializes the current run (as defined by entries made during an on-screen prompt session or via an input data file) with a *random* spatial distribution of all forces.
- **"O"** (for Options): enables on-screen prompts for making on-the-fly changes to the values of any (or all) of the parameters defining the battle dynamics and/or red and blue ISAACAs. For details see ***On-the-Fly Parameters Changes***.
- **"P"** (for Play-Back): toggles an on-screen prompt to enter the name of an ***.out** file to "play-back" at high speed. See ***Play-Back of *.out Files***.
- **"Q"** (for Quit): quits back to main menu, from which the user can select to either read-in a new data input file, play-back an ***.out** file or quit the program.
- **"R"** (for Re-Run): reinitializes the current run (as defined by entries made during an on-screen prompt session or via an input data file) with exactly the same spatial distribution of all forces as the first time the current run was initialized. To randomize this initial distribution, use the "N" hot-key (see above).
- **"S"** (for Step): enables the *single-step* run mode, in which the screen is updated a single iteration step at a time, each time the "S" hot-key is pressed. For a continuous (or fast) update, use the "F" hot-key (see above).
- **"T"** (for Trace): toggles a continuous trace of red and blue movement. That is to say, old ISAACA positions are not erased as ISAACAs move throughout the battlefield. Such traces sometimes facilitate the visual detection of certain developing patterns.
- **"U"** (for SqUad): toggles a color highlighter (yellow for red, white for blue) for ISAACAs belonging to the particular squad whose parameters are currently displayed in the appropriate ISAACA data regions. See "B" and "E" hot-keys above.

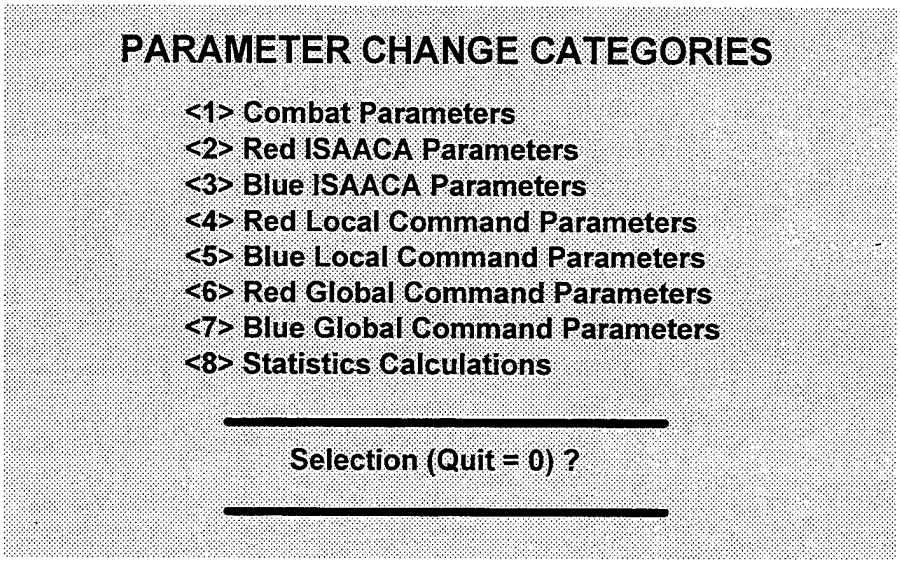
On-the-Fly Parameter Changes

During an interactive run (i.e., a run that is initialized with a full data file such as **ISAAC.dat** instead of a simple playback of a previously recorded ***.out** file), the user can press the "O" hot-key at any time to interrupt the run and make on-the-fly changes to the values of any (or all) of the parameters defining the battle dynamics and/or red and blue ISAACAs. For example, the consequences to the unfolding pattern of behavior on the battlefield of altering the blue force's aggressiveness

and/or changing red's predisposition for "helping" injured friendly forces can – by changing the values of the appropriate parameters – be immediately displayed on screen.

Enabling this *On-the-Fly Parameter Change* option by pressing the "O" key displays a menu of eight categories of changes that can be made (see figure 34).

Figure 34. Main menu of the "on-the-fly" parameter change option

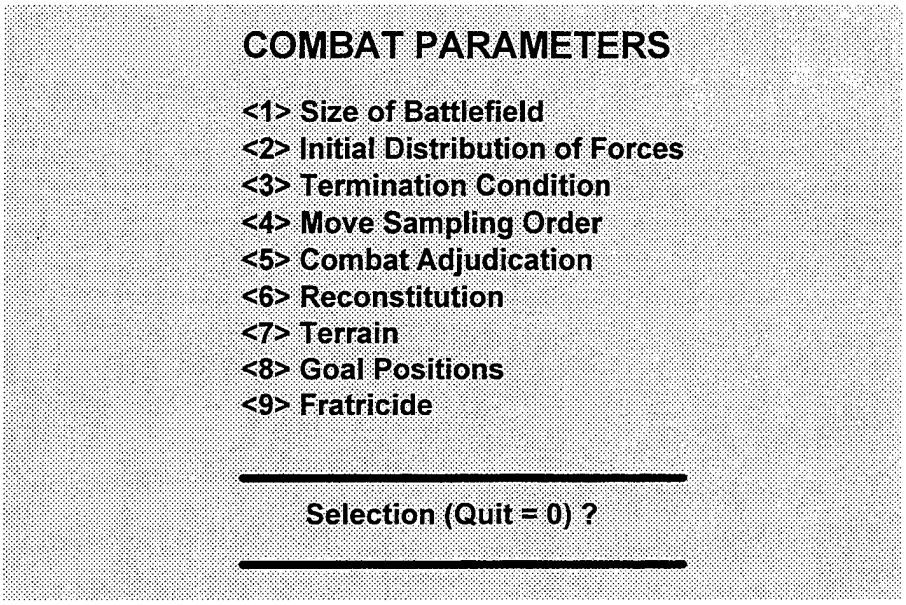


Each category is accessed by pressing the appropriate number – "1" (for **Combat Parameters**) through "8" (for **Statistics Calculations**) – followed by **<ENTER>**, and contains a more specific list of parameters whose values can be changed on-the-fly. These are described in more detail below.

Pressing *zero* (i.e., "0") from this main menu exits the menu, then queries the user if the altered parameter values should be included in a new ISAAC ***.dat** file. If the answer is "yes", the user is prompted for a new input data file name (the currently open data file can also be over-written). If the set of parameters whose values were changed during this session is such that the run must be restarted (parameters such as the size of the battlefield and the number of red and/or blue ISAACAs are in this category), then ISAAC reinitializes the current run and displays the main graphics screen (see figure 23). Otherwise, if the run can be smoothly continued from the point at which it was interrupted by the "O" key (but with new parameter values in place – ISAACA personality weight vectors, constrain thresholds, sensor and/or fire ranges, etc. all belong in this category of parameter values), ISAAC first queries the user whether such a continuation is desired, or whether the user nonetheless wishes to restart the run using the new parameter

values. If the user selects to restart the run rather than to continue the old one, ISAAC queries whether the user wants to restart using the original spatial distribution of red and blue forces (i.e., effectively using the "old" random number seed) or wishes to select a new random number seed to initialize the run.

Figure 35. Screen shot of the "on-the-fly" **Combat Parameter** change sub-menu



Combat Parameters

Pressing the "1" key (followed by **<ENTER>**) from the main menu that appears after interrupting a run by pressing the "O" hot-key, displays a sub-menu containing nine combat-related parameter choices (see figure 35):

- *<1> Size of Battlefield*, which prompts the user to adjust the size of the notional battlefield.
- *<2> Initial Distribution of forces*, which prompts the user to define a new initial configuration.
- *<3> Termination Condition*, which prompts the user to choose a new termination condition.
- *<4> Move Sampling Order*, which prompts the user to select a new sampling order (fixed or random sampling; see *Move Sampling Order*).

- <*5*> ***Combat Adjudication***, which prompts the user to specify the maximum number of simultaneously targetable enemy ISAACAs.
- <*6*> ***Reconstitution***, which prompts the user to decide whether the reconstitution option will be used. If "yes", then the user is further prompted for red and blue reconstitution times; see ***ISAACA Reconstitution***.
- <*7*> ***Terrain***, which prompts the user to decide whether the terrain option will be used. If "yes", then the user is further prompted for terrain block sizes and positions; see ***Terrain***.
- <*8*> ***Goal Positions***, which prompts the user to select new red and blue goal positions
- <*9*> ***Fratricide***, which prompts the user to decide whether the fratricide option will be used. If "yes", then the user is further prompted for red and blue fratricide radii and fratricide probability of hit; see ***ISAACA Fratricide***.

The user can change the value of as many parameters as desired. When finished, pressing *zero* (follows by **<ENTER>**) returns the user to the main *On-the-Fly Parameter Change* menu.

Red ISAACA Parameters

Pressing the "2" key (followed by **<ENTER>**) from the main menu that appears after interrupting a run by pressing the "O" hot-key, displays a sub-menu containing 27 red-ISAACA-related parameter choices:

- <*1*> ***Number of Red Forces***, which prompts the user to adjust the number of red ISAACAs. If there are more than one squad, the user is prompted to adjust the size of each squad and/or alter the number of squads.
- <*2*> ***Movement Range***, which prompts the user to select the movement range (either "1" or "2").
- <*3*> ***Personality***, which prompts the user to set the personality flag (either 1 for user-defined, or 2 for random). If the user-defined option is selected, the user is prompted for the values of the components of red's personality weight vector; see ***ISAACA Personalities***).
- <*4*> ***Weight*** w_1, which prompts the user to enter a value for red's alive and injured weights for moving toward alive red.
- <*5*> ***Weight*** w_2, which prompts the user to enter a value for red's alive and injured weights for moving toward alive blue.

- *<6> Weight* w_3, which prompts the user to enter a value for red's alive and injured weights for moving toward injured red.
- *<7> Weight* w_4, which prompts the user to enter a value for red's alive and injured weights for moving toward injured blue.
- *<8> Weight* w_5, which prompts the user to enter a value for red's alive and injured weights for moving toward red goal.
- *<9> Weight* w_6, which prompts the user to enter a value for red's alive and injured weights for moving toward blue goal.
- *<10> Weight* w_7, which prompts the user to enter a value for red's weight for moving toward the local commander (if the local commander option is set; see ***Local Command*** in ***Contents of Input Data File***).
- *<11> Weight* w_8, which prompts the user to enter a value for red's weight for obeying its local commander (if the local commander option is set; see ***Local Command*** in ***Contents of Input Data File***).
- *<12> Sensor Range*, which prompts the user to enter a value for red's sensor range.
- *<13> Fire Range*, which prompts the user to enter a value for red's fire range
- *<14> Communications*, which prompts the user to set red's communication option on or off.
- *<15> Communications Range*, which prompts the user to enter a value for red's communication range (see ***ISAACA Communication***).
- *<16> Communications Weight*, which prompts the user to enter a value for red's communication weight (see ***ISAACA Communication***).
- *<17> Movement Constraints*, which prompts the user to set red's movement constraint flag (1 meaning that constraints will be used, 0 that no further constraints will be used; see ***ISAACA Adaptability***).
- *<18> Threshold Range*, which prompts the user to set red's threshold range; see ***ISAACA Adaptability***.
- *<19> Advance Threshold*, which prompts the user to set red's advance threshold range; see ***ISAACA Adaptability***.
- *<20> Cluster Threshold*, which prompts the user to set red's cluster threshold range; see ***ISAACA Adaptability***.

- *<21> Combat Threshold*, which prompts the user to set red's combat threshold range; see ***ISAACA Adaptability***.
- *<22> Min Distance from Red*, which prompts the user to set red's minimum-distance-from-red constraint; see ***ISAACA Adaptability***.
- *<23> Min Distance from Blue*, which prompts the user to set red's minimum-distance-from-blue constraint; see ***ISAACA Adaptability***.
- *<24> Min Distance from Red Goal*, which prompts the user to set red's minimum-distance-from-red-goal constraint; see ***ISAACA Adaptability***.
- *<25> Probability of Shot*, which prompts the user to set red's single-shot probability.
- *<26> Max Number of Engagements*, which prompts the user to set red's maximum number of simultaneous enemy targets.
- *<27> Defense ('armor')*, which prompts the user to set red's alive and injured defensive strength (see ***Notional Defense in ISAACA Combat***).

Blue ISAACA Parameters

Pressing the "3" key (followed by **<ENTER>**) from the main menu that appears after interrupting a run by pressing the "O" hot-key, displays a sub-menu containing various blue-ISAACA-related parameter choices. Except for the fact that this sub-menu obviously references blue rather than red parameters, all **Blue ISAACA Parameters** menu choices have exactly the same meaning as their red counterparts, defined above.

Red Local Command Parameters

Pressing the "4" key (followed by **<ENTER>**) from the main menu that appears after interrupting a run by pressing the "O" hot-key, displays a sub-menu containing twenty red-local-command-related parameter choices:[18]

- *<1> Local command flag (on/off)*, which prompts the user to set red's local command flag ("1" means that local command will be used, while "0" that it will not be used; see ***Local Command***). If the local command flag is enabled, the user is prompted for the number of desired local commanders and to enter values for all local command parameters.

[18] Note that if there is currently more than one red local commander defined, the user is prompted to enter values for each local commander, in turn.

- *<2> **Patch type***, which prompts the user to set the red local commander's patch type ("1" means that the command area will be partitioned into 3-by-3 matrix of sub-blocks, while "2" means that it will be partitioned into 5-by-5 matrix of sub-blocks); see ***Local Command***.
- *<3> **Patch-Choice flag***, which prompts the user to set the red local commander's patch-choice flag ("1" means that if two or more patches yield the same penalty value, the local commander will choose among that set of same-penalty patches *randomly*, while "2" means that the actual patch chosen will be the one closest to the previously selected patch). See ***Local Command***.
- *<4> **Weight*** w_1, which prompts the user to enter a value for the red local commander's alive and injured weights for moving toward alive red.
- *<5> **Weight*** w_2, which prompts the user to enter a value for the red local commander's alive and injured weights for moving toward alive blue.
- *<6> **Weight*** w_3, which prompts the user to enter a value for the red local commander's alive and injured weights for moving toward injured red.
- *<7> **Weight*** w_4, which prompts the user to enter a value for the red local commander's alive and injured weights for moving toward injured blue.
- *<8> **Weight*** w_5, which prompts the user to enter a value for the red local commander's alive and injured weights for moving toward red goal.
- *<9> **Weight*** w_6, which prompts the user to enter a value for the red local commander's alive and injured weights for moving toward blue goal.
- *<10> **Sensor Range***, which prompts the user to enter a value for the red local commander's sensor range.
- *<11> **Local command radius***, which prompts the user to enter a value for the red local commander's local command radius. Recall that this effectively defines the size of one of the sub-blocks into which the command area is partitioned. See ***Local Command***.
- *<12> **ISAACAs under command***, which prompts the user to enter a value for the number of subordinate ISAACAs under the i^{th} red local commander's command.

- *<13> Threshold Range*, which prompts the user to enter a value for the red local commander's threshold range. See ***ISAACA Adaptability***.
- *<14> Advance Threshold*, which prompts the user to enter a value for the red local commander's advance threshold. See ***ISAACA Adaptability***.
- *<15> Cluster Threshold*, which prompts the user to enter a value for the red local commander's cluster threshold. See ***ISAACA Adaptability***.
- *<16> Combat Threshold*, which prompts the user to enter a value for the red local commander's combat threshold. See ***ISAACA Adaptability***.
- *<17> Alpha*, which prompts the user to enter a value for the red local commander's *alpha* command weight. See ***Local Command***.
- *<18> Beta*, which prompts the user to enter a value for the red local commander's *beta* command weight. See ***Local Command***.
- *<19> Delta*, which prompts the user to enter a value for the red local commander's *delta* command weight. See ***Local Command***.
- *<20> Gamma*, which prompts the user to enter a value for the red local commander's *gamma* command weight. See ***Local Command***.

Blue Local Command Parameters

Pressing the "5" key (followed by **<ENTER>**) from the main menu that appears after interrupting a run by pressing the "O" hot-key, displays a sub-menu containing various blue-local-command-related parameter choices Except for the fact that this sub-menu obviously references blue rather than red parameters, all **Blue Local Command Parameters** menu choices have exactly the same meaning as their red counterparts, defined above.

Red Global Command Parameters

This sub-menu is not yet implemented in the current version of ISAAC.

Blue Global Command Parameters

This sub-menu is not yet implemented in the current version of ISAAC.

Statistics Calculations

This sub-menu is not yet implemented in the current version of ISAAC.

Sample Runs

The ISAAC that is described in this report is an interim version of a "work in progress." It represents but a skeletal fragment of what will eventually become the "core engine" of a much more sophisticated set of tools. It is encouraging to note that, however, that even at this early juncture, ISAAC already has an impressive repertoire of self-organized emergent behaviors:

- Forward advance
- Frontal attack
- Local clustering
- Penetration
- Retreat
- Attack posturing
- Containment
- Flanking Maneuvers
- Defensive posturing
- "Guerilla-like" assaults
- Encirclement of enemy forces
- *many more ...*

Moreover, ISAAC frequently displays behaviors that appear to involve some form of "intelligent" division of red and blue forces to deal with local "firestorms" and skirmishes, particularly those forces whose personalities have been "evolved" (via the *Genetic Algorithm Evolver*) to perform a specific mission (see below). It must be remembered that such behaviors are not hard-wired-in but are effectively an emergent property of a decentralized and nonlinear local dynamics.

No one has yet provided a satisfactory formal definition of *emergence.* Loosely speaking, emergent behavior refers to the group behavior of two or more ISAACAs that arises from, but is also qualitatively different from, the collective interactions of the individual ISAACAs. For example, sample run #4 shows how a slow clockwise precession of a tight cluster of combatants locked in close combat can emerge out of the collective interactions of enemy ISAACAs. Sample run #7 shows how a seemingly well orchestrated "encircling" maneuver can spontaneously emerge out of the combined actions of very many ISAACAs, all of whom individually want only to fight the enemy and move toward the enemy's flag.

Figures 36 through 50 provide color "snapshots" of several sample runs using ISAAC. Table 7 also gives short descriptions. A majority of these

runs can be played back in their entirety by using the stand-alone "play-back" program **ISAAC_PB** (see table 4). The files **LOCALCMD.out**, **GLBALCMD.out** and **BATTLE1.out**, which were generated using a later version of ISAAC, must be played back by choosing option 2 on **ISAAC_CE**'s main options screen (see figure 24).

Note that while red and blue ISAACAs are appropriately colored red and blue in the following figures, ISAAC actually uses four colors for rendering runs on a computer screen: *bright red* for alive red ISAACAs, *bright blue* for alive blue ISAACAs, *dark red* for injured red ISAACAs and *dark blue* for injured blue ISAACAs.

Table 7. ISAAC output files corresponding to the sample runs shown in figures 36 through 50

Sample Run	Figure	ISAAC output file[1]	Brief Description
1	36	**MISMATCH.out**	One side "outmatches" the other
2	37	**FLUID_1.out**	Fluid-like collision between roughly equal force strengths
3	38	**FLUID_2.out**	Fluid-like collision between two large forces
4	39	**PRECESS.out**	Battle is constrained to a small pocket that slowly precesses in clockwise direction
5	40	**GOALDEF1.out**	Successful "goal-defense" by red
6	41	**GOALDEF2.out**	Unsuccessful "goal-defense" by blue; red's behavior is suggestive of an abrupt "phase transition"
7	42	**CIRCLE.out**	Red finds a way to encircle blue
8	43	**FIRESTM1.out**	Series of local firestorms
9	44	**FIRESTM2.out**	Same system as in **FIRESTM1.out** but blue's communications option is turned 'on'
10	45	**SENSOR.out**	How well does red do against blue as red's sensor range is systematically increased relative to that of blue?
11	47	**LOCALCMD.out**	Sample run with a very timid red local commander; blue has no command structure
12	49	**GLBALCMD.out**	Sample run with a very timid red global commander commanding three local commanders; blue has no command structure
13	50	**BATTLE.out**	A "mini-battle" with 400 ISAACAs per side

[1] Output files are provided on the accompanying disk.

Sample Run #1: MISMATCH.out

Snapshot views of the first sample run are shown in figure 36. A "play-back" of this sample can be viewed by running **ISAAC_PB** using **MISMATCH.out** as input.

The main points of interest in this file are:

1. *The very different sensor (= r_S) and fire ranges (= r_F) possessed by red and blue ISAACAs*: red ISAACAs have r_S=2 and r_F=1 while blue ISAACAs have r_S=7 and r_F=5. Thus blue can "see" much farther than red (effectively anything within a 14-by-14 "box" as compared to a 4-by-4 box by red), and can shoot from a much farther distance (anything within a 10-by-10 box as compared to a 1-by-1 box for red).

2. *While red's decision-making process is driven purely by its default personality weight vector* ($\vec{w}_{\text{red}} = (10, 40, 10, 40, 0, 50)$), *blue decides upon its moves by both using the same set of personality weights as red and by incorporating additional movement constraints.* Specifically, each blue ISAACA will choose to (i) move toward the red flag only when it is surrounded by at least 4 friendly ISAACAs (within a threshold range r_T=3 units), (ii) it will no longer move closer to friendly ISAACAs if it is surrounded by more than 10 friendly ISAACAs, and (iii) it will move toward engaging an enemy in combat only if it senses a force advantage of 7 friendly ISAACAs over the enemy. Moreover, blue ISAACA's try to maintain a minimum distance of 1 unit away from friendly ISAACAs and a minimum distance of 2 units away from enemy ISAACAs. The blue force can therefore be characterized as having a much better "situational awareness" than the red force (owing to its much large sensor range) and having a considerably less aggressive nature. The last part is so because while red ISAACAs will generally tend to engage all enemy ISAACAs within their sensor and fire ranges, blue ISAACAs will only choose to do so when they "know" that they have a significant local force advantage.

The initial configuration of the 100-by-100 notional battlefield at time t=1 is shown at the top of figure 36.

Snapshot views of the "battle" are provided for times t = 25, 35, 42, 65, 75, 85, 95, 110, and 127. Each of these frames tells a part of the unfolding story.

The frame at time t=25 shows that blue has progressed towards its "goal" (the red flag) farther than red has progressed towards its goal (the blue

flag). This is due mainly to red's tendency to "cluster" with nearby red ISAACAs. Blue ISAACAs, on the other hand, effectively put a clamp on clustering whenever the number of nearby friendlies exceeds a certain number (10 friendlies within a distance of 3 units). When this threshold is exceeded, blue ISAACAs are able to devote their full attention to moving toward their goal. It is difficult to see in the figure, but there is a single red ISAACA near the center of the frame that has moved quickly forward. It is able to do so because it was initially too far removed from friendly ISAACAs to cluster with them.

The frame at time t=35 shows that this single red ISAACA is met by the large cluster of forward moving blue ISAACAs. It is not entirely clear from this image, but as soon as the blue ISAACAs sense the presence of the lone enemy, they quickly move to surround and engage the enemy in combat. The effect of this rapid clustering around the enemy is more evident in the frame at time t=42, which shows the single red ISAACA completely surrounded by blue forces. Meanwhile, the remaining red ISAACAs are slowly moving in the direction of the blue flag. Because of their limited sensor range, none of these red forces has yet "seen" any enemy ISAACAs.

The frame at time t=65 shows that the large cluster of blue ISAACAs has destroyed the lone red ISAACA that was seen to have rushed quickly forward in the frames at t=25 and t=35. ISAAC's full screen view (such as the one shown in figure 23) would show at this point that blue has by this time managed, from long range, to kill 1 red ISAACA and injure another. However, because of red's inferior sensor and fire ranges, red has been unable to injure any blue forces thus far.

The frame at time t=75 shows a snapshot image of some "close-combat," and is the first image in which there is some overlap in the clusters of red and blue ISAACAs. At this point, red forces have been significantly depleted – a total of 5 have been killed and another 6 injured. No blue ISAACAs, however, have yet even been injured. This is again due to blue's superior sensor and fire ranges. While each red ISAACA can only engage a blue if that blue occupies an immediately adjacent site, blue ISAACAs can engage any red that is within a 10-by-10 box around it.

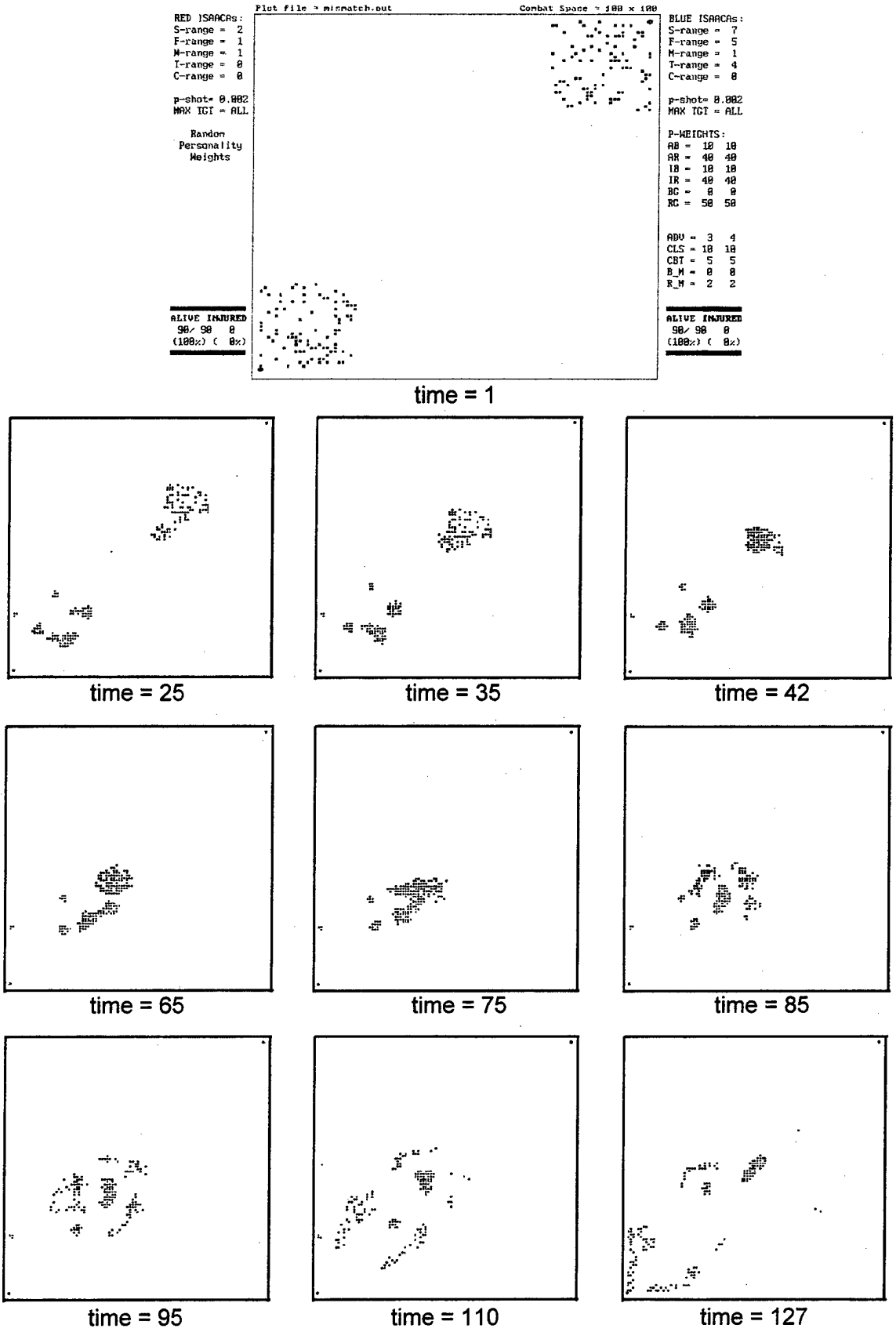
The frame at time t=85 shows that the large cluster of blue ISAACAs seen in the previous frames has broken up into several disjoint parts that appear to "surround" the red ISAACAs. Keep in mind, that as these blue forces surround their enemy, they continue moving toward the red flag. Moreover, as the blue ISAACAs continue moving forward they are able to maintain a significant power projection against the red ISAACAs. By time t=85, 8 red ISAACAs have been killed, and another 13 injured. At the same time, no blue ISAACAs have yet been injured.

The next two frames, at times t=110 and t=127, show the conclusion of this battle. Blue ISAACAs continue to "maneuver around" the red

forces, keeping their distance from and engaging their enemy from long range. The sequence ends at time t=127 when the first blue ISAACA has reached the red flag. At the end of the battle, blue has killed 16 red ISAACAs and injured 15; red has managed to injure only one blue ISAACA and kill none.

It is tempting to conclude from the last few frames of figure 36 that the blue side has devised an impromptu strategy to deal with the enemy. Blue's strategy *seems to be* to exploit its superior sensor and fire ranges by "intelligently" splitting up its force and continuing to advance toward the red goal while surrounding, and pummeling, the enemy from a long standoff range. Keep in mind, however, that this seemingly centrally-directed behavior (as is all the behavior seen in the succeeding figures), stems solely from a strictly *decentralized* dynamics.

Figure 36. Snapshot views of `MISMATCH.out`



Sample Run #2: FLUID_1.out

Figure 37 shows an example of a very different kind of "evolution." A "play-back" of this run can be viewed by running **ISAAC_PB** using **FLUID_1.out** as input.

The initial state of the battle is shown at the top of the figure, and shows that, as in sample run #1, the red and blue ISAACAs initially occupy random positions in diagonally opposite corners at time t=1. Notice that while the blue ISAACA parameters are essentially unchanged from sample run #1 (except for being even *less aggressive* this time around, by having a combat threshold of 7, and wanting to maintain a minimum distance of 2 units from red ISAACAs), the red force personality is very different from the first sample run. Unlike the case in the first sample run, where red's sensor and fire ranges were very small compared to that of blue, here red's sensor and fire ranges equal blue's. Moreover, while red's default personality was the same as blue's in sample run #1, here each red ISAACA's personality is completely random. Finally, red's decision-making process is also driven by additional constraints: each red ISAACA will choose to (i) move toward the blue flag only when it is surrounded by at least 1 friendly ISAACA (within a threshold range r_T=3 units), (ii) it will no longer move closer to friendly ISAACAs if it is surrounded by more than five friendly ISAACAs, and (iii) it will engage in combat only if it senses an equal relative local force strength. Red ISAACAs also wish to maintain a minimum distance of 2 units away from friendly ISAACAs and a minimum distance of 1 unit away from enemy ISAACAs. Both sides start out with force strengths of 90 ISAACAs.

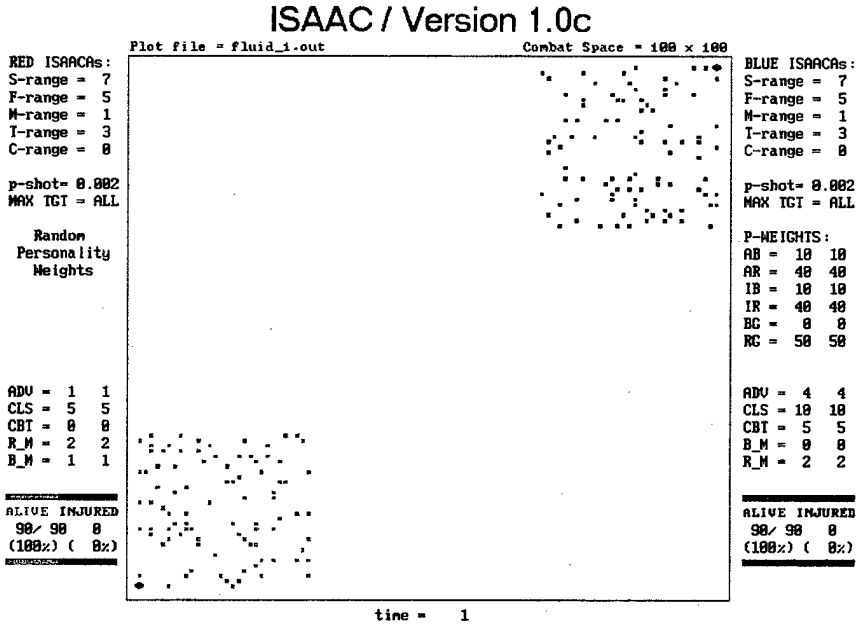
As the reader can see either by mentally reconstructing the actual "flow" of the battle from the six snapshots shown in figure 37 (which encompass events from times t=35 to t=155), or by actually viewing this sequence in its entirety by using **ISAAC_PB** to play back the file **FLUID_1.out**, combat between these two particular personalities proceeds as though it were a clash between two viscous fluids. Forces making up the two sides collide head-on but are dispersed and aligned along two narrow columns at time t=55. The two sides continue battling each other in this manner for a relatively large number of iterations (time t=55 to t=110 in the figure). Notice that, at time t=85, several blue ISAACAs have "found" a way to sneak around the bottom of red's column-like formation. In the next frame, at time t=110, this group can be seen advancing toward red's flag unchallenged, because it is unseen by the red ISAACAs making up red's central column. By time t=130, a cluster of red ISAACAs breaks away from what used to be the central column and advances towards blue's flag. Meanwhile, blue forces continue advancing toward red's flag at the bottom of the frame.

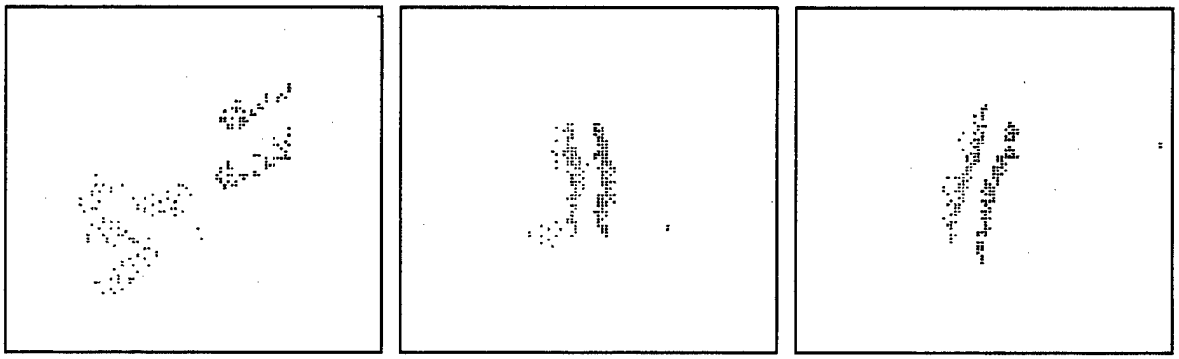
Note that red's column appears slightly more dispersed than does blue's. The reason for this is that whereas blues want to be as close to one another as other extant local conditions permit), reds want to maintain a minimum distance of D=2 units between themselves and other reds. This continual local "jostling for elbow room" renders red's column less finely structured.

An unexpected novel feature emerges in the frame at time t=155. The circular cluster of blue ISAACAs at the upper left of the lower right quadrant of the battlefield in fact makes up a small circularly rotating "vortex" that lasts for a few iterations. Another, smaller vortex forms later near the bottom of the battlefield (but is not shown in this figure).

At the end of the "battle" reproduced in figure 37, red ISAACAs have managed to kill 3 blues and injure another 12; blue ISAACAs have killed 4 reds and injured 10.

Figure 37. Snapshot views of `FLUID_1.out`





time = 25 time = 50 time = 75



time = 100 time = 125 time = 150

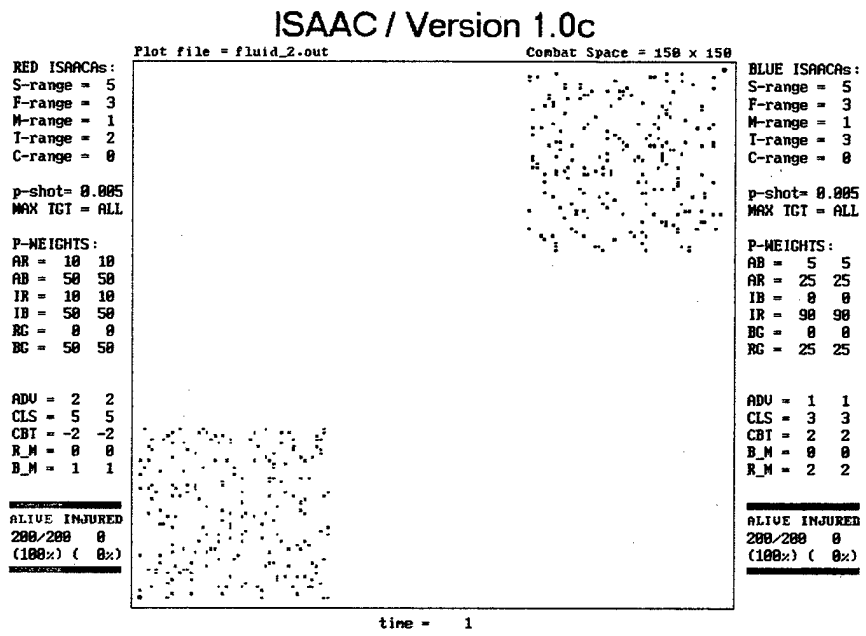
Sample Run #3: FLUID_2.out

Figure 38 shows a few snapshots taken from a play-back of the file **FLUID_2.out**.

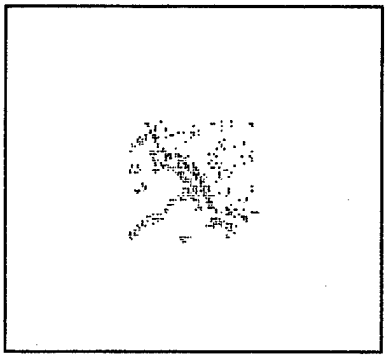
For this sample run – the first part of whose evolution is reminiscent of a collision between two viscous fluids (see ***Sample Run #2: FLUID_1.out*** for comparison) – there are 200 ISAACAs per side, and red and blue ISAACAs share the same sensor and fire ranges (equal to 5 and 3, respectively). From a personality-weight perspective, blue ISAACAs are relatively aggressive, in that they care little about "staying close" to friendly ISAACAs and are motivated most strongly by moving toward injured enemies. The red force, by comparison, distributes its motivation more evenly among alive and injured red and blue ISAACAs. Note – from the red and blue combat thresholds – that while red is more apt to start local skirmishes than blue, each is more prone to engage in combat than their counterparts in the previous example. Note also that because of their different advance threshold constraints, red and blue forces have different styles of moving toward their enemy's flag: blue moves as essentially one large cluster while red self-organizes into more ordered "columns."

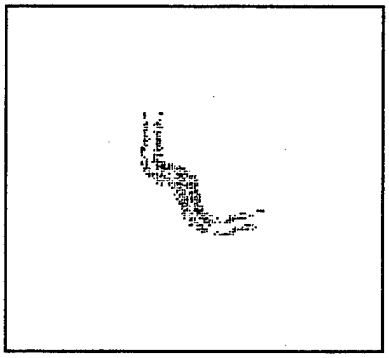
Figure 38 shows that, initially, the two forces collide in what is best described as "fluid-like" fashion. However, in contrast to the previous sample run's simple columnar style of collision – in which the two forces buttress up against each other along a single "boundary" before a few red and blue ISAACAs manage to "escape" from the top and bottom, respectively – the collision of the two forces in this sample run is more complex. Here, the combined mass appears to be arranged along a well-defined but nonlinear and undulating boundary, that strains under pressure near several "choke points." One of these choke points (near the bottom right of the combined mass) finally breaks around $t \sim 90$, splitting the combined mass into two disjoint parts. A second choke point (near the top left of the larger central mass) breaks around $t \sim 114$. In the immediately succeeding steps, clusters of red and blue ISAACAs combine and break apart a few more times before some manage to find their way to their enemy's flag.

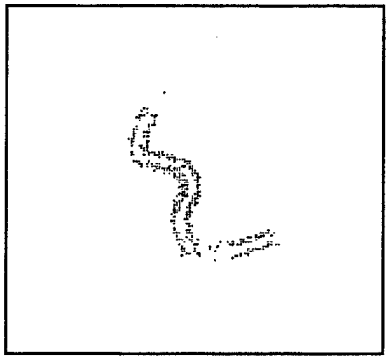
Figure 38. Snapshot views of `FLUID_2.out`

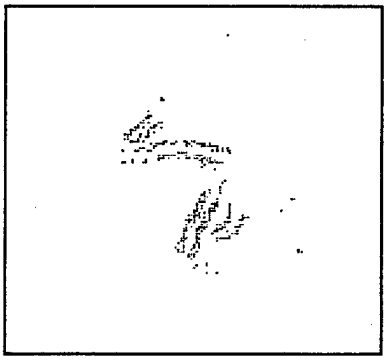



time = 25


time = 50


time = 75


time = 100


time = 125


time = 175

Sample Run #4: PRECESS.out

Figure 39 shows a few snapshots taken from a play-back of the file **PRECESS.out**, and represents the first simple example of an emergent behavior; namely, a slow clockwise precession of two forces locked in local combat.

In this sample run, the two sides are equipped with the same sensor and fire ranges (equal to 5 and 3, respectively), but have markedly different personalities and additional movement constraints. Red ISAACAs favor moving toward alive and injured reds (with relative weights of 25 and 75, respectively) over moving toward alive and injured blue forces (10 and 25, respectively). This means that red forces are 7-1/2 times more "concerned" with moving towards injured red forces than they are moving toward alive blue, and 3 times more concerned with moving towards injured red forces than toward injured blue forces. In contrast, blue ISAACAs are considerably more concerned with moving toward red forces than toward friendlies. In particular, blue ISAACAs are 3-1/2 times more concerned with moving towards alive red forces than either alive or injured blue forces, and 8 times as concerned with moving toward injured red forces.

Red and blue forces for this scenario also differ markedly in their respective sets of movement constraint conditions. Red forces, for example, advance toward the goal only if surrounded by at least 5 friendly forces within a threshold range r_T=2, continue moving toward friendly forces until surrounded by at least 10 reds, and move to engage an enemy ISAACA only if they sense a local numerical advantage of 4 ISAACAs over blue forces. Moreover, red forces wish to maintain a minimum distance of 3 from all enemy ISAACAs. In contrast, blue forces advance toward the red flag even when surrounded by a single friendly ISAACA within a threshold range r_T=3, they continue to cluster with friendly forces only until they are surrounded by at least 3 blues, and move to engage an enemy ISAACA even if they sense that they are locally *outnumbered* by the enemy by 5 ISAACAs. Moreover, they want to get as close to red forces as possible (i.e., the minimum distance is set to 0).

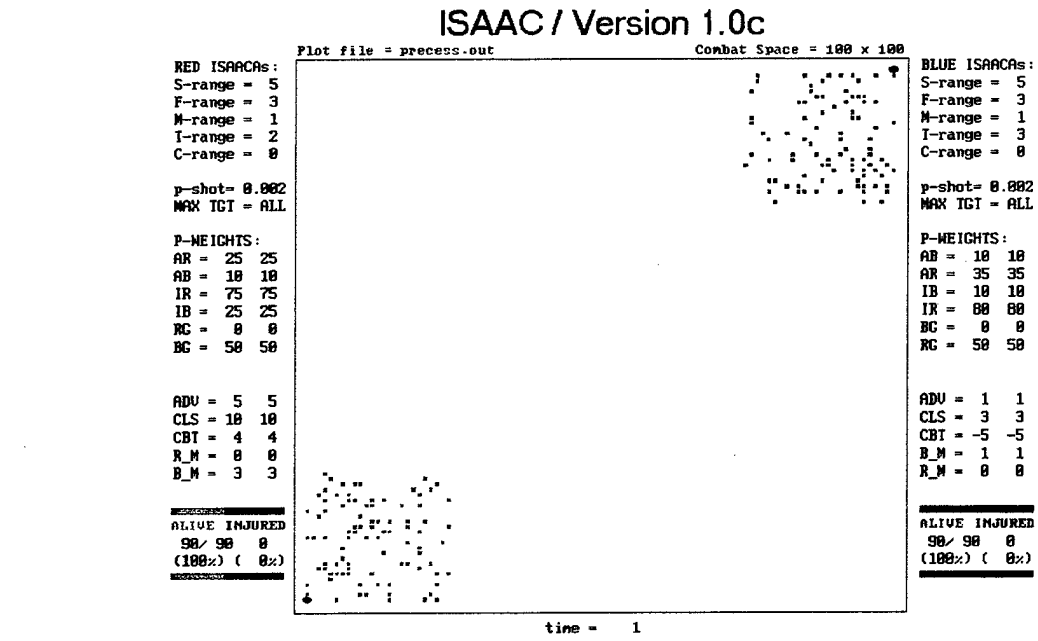
This stark contrast in personalities can be summarized by saying that, for this run, *reds want to avoid a fight as badly as blues want to start one.* And once a fight has started, and blues sense injured enemy ISAACAs, blues desire to "finish off" the enemy even more than they want to advance towards red's flag.

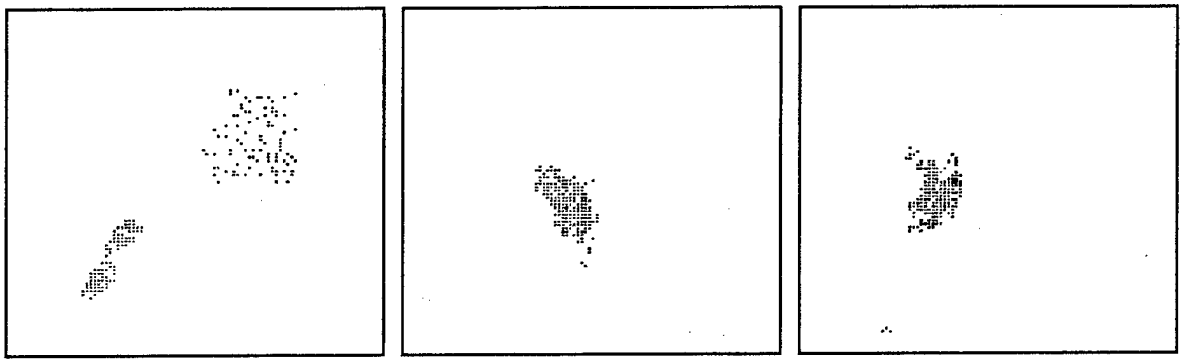
As in previous sample runs, red and blue ISAACAs are initially positioned in diagonally opposite corners (see the top frame at time t=1). The next frame shows the two sides meeting near the center at time t=37. Red forces are clustered into two advancing groups, while

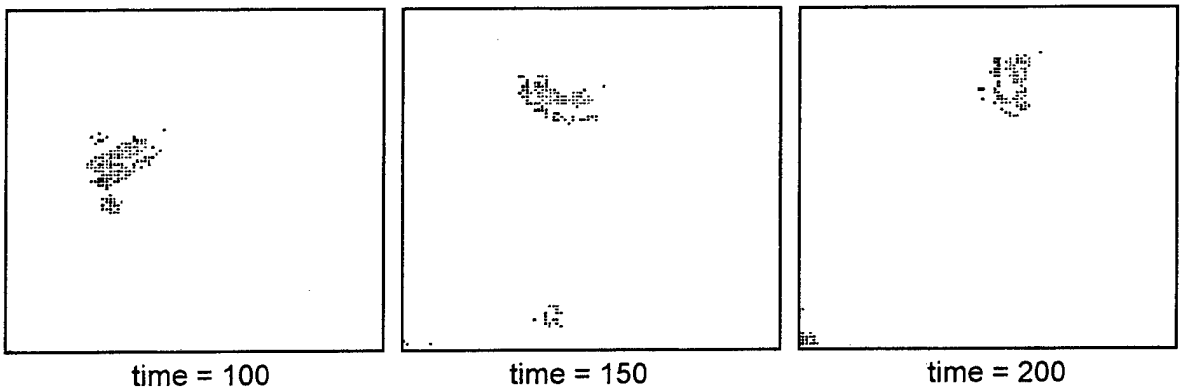
blue forces consist of one large, and widely dispersed, cluster. What is interesting about this particular sample run, and the way in which these two very different personalities "interact," is summarized by the last three frames of the battle, shown for times t=75, 125 and 250. Notice that, unlike any of the previous sample runs, here the two forces remain essentially *locked together* in local combat, moving slowly around the upper half of the battlefield. Except for a few stray "leakers" and an occasional group of a few blue ISAACAs that chooses to leave the main battle and head toward red's flag, there are no scattered skirmishes during this run. Almost all of the combat takes place within the large cluster of red and blue forces. Notice also that this slow processional of the locked-together cluster evolves over a relatively long time; the cluster remains well-formed even up to the very last frame shown for this run, showing the state of the battle at time t=250. It is also interesting to note that the collective motion of the locked-together cluster of red and blue ISAACAs is driven first by blue's desire to engage (and finish-off) red forces coupled with red's desire to flee, and later – as red gets closer to the blue flag – by red's desire to get to its goal (while still being chased by blues).

At the end of this "battle," red ISAACAs have managed to kill 25 blues and injure another 27; blue ISAACAs have killed 29 reds and injured 24. A large number of blues have also found their way to, and are clustered around, the red flag.

Figure 39. Snapshot views of `PRECESS.out`







Sample Run #5: GOALDEF1.out

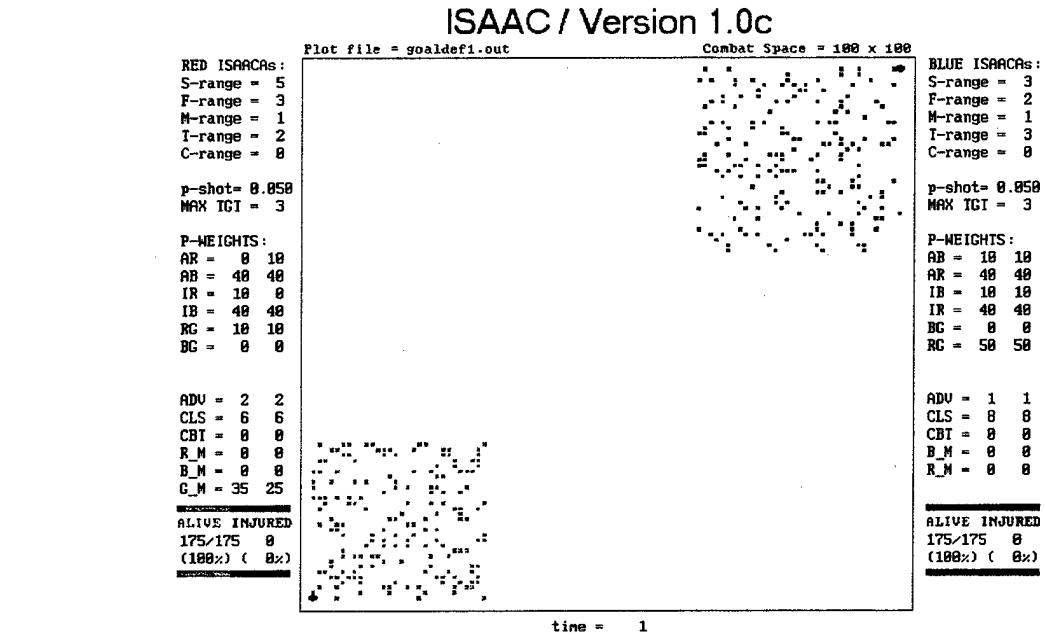
Figure 40 shows a few snapshots taken from a play-back of the file **GOALDEF1.out**.

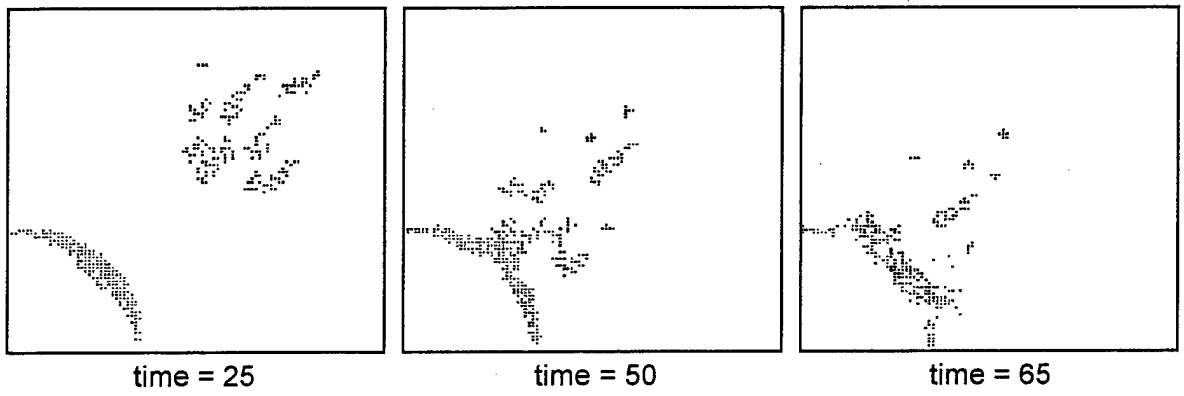
This, and the next, sample run illustrate how ISAAC can be used to explore simple "goal defense" scenarios. In this run; red defends its own flag with personality weight vector $\vec{w}_{\text{alive}} = (0, 40, 10, 40, 10, 0$. Notice that the *first* (i.e., $w_{\text{alive},1}$) and *last* (i.e., $w_{\text{alive},6}$) components of this weight vector are equal to zero: $w_{\text{alive},1} = 0$ means that red ISAACAs initially do not see other red ISAACAs; $w_{\text{alive},6} = 0$ means that the red forces are not "motivated" by moving toward the blue flag. Since $w_{\text{alive},5} > 0$, however, red ISAACAs are interested in staying near their own flag. In particular, since **G_M** = 35, red ISAACAs "desire" to be a distance D = 35 units away from their own flag (recall that the parameter **G_M** – listed near the bottom of red's constraint parameter data field – represents the desired distance from own goal; see ***Contents of ISAAC's Input Data File***). The first snapshot in figure 40 (for time t = 25) shows that red prepares for blue's attack by setting up its forces along a semi-circle surrounding its flag. Note that red is given an advantage over blue in terms of both its *sensor* ($r_{S,\text{red}} = 5$ compared with $r_{S,\text{blue}} = 3$) and *fire ranges* ($r_{F,\text{red}} = 3$ compared with $r_{F,\text{blue}} = 2$).

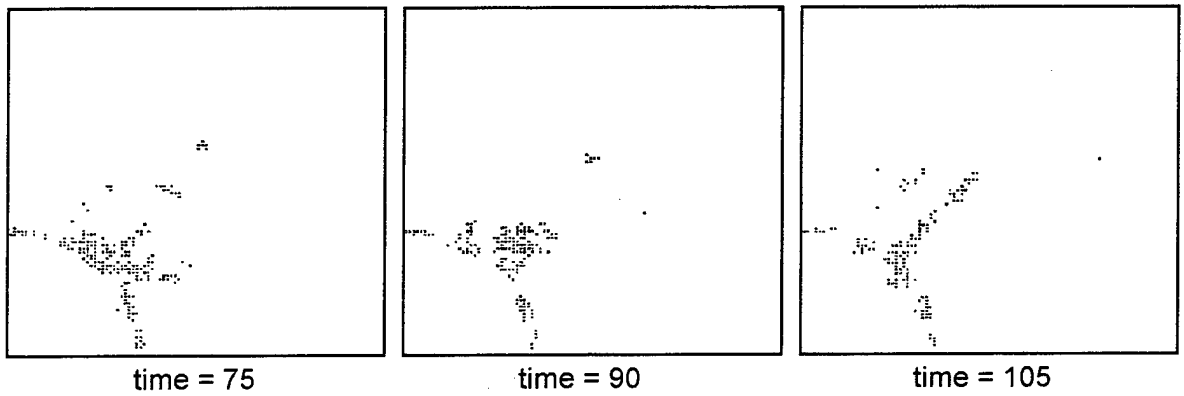
Subsequent snapshots of this sample run show that red is largely successful in defending its goal against blue forces. As soon as blue forces appear within red's sensor range (t = 50), red forces move out to intercept. As red:blue close combat ensues, and red's ranks are thinned near the center of the semi-circle, reinforcement red ISAACAs move in from the edges. As blue tries to penetrate the semi-circular defense posture, red forces move toward and surround the enemy.

Figure 40 shows that by time t = 125, red has successfully prevented any of the blue forces from reaching the red flag, and has in fact managed to kill most of the attacking force.

Figure 40. Snapshot views of `GOALDEF1.out`







Sample Run #6: GOALDEF2.out

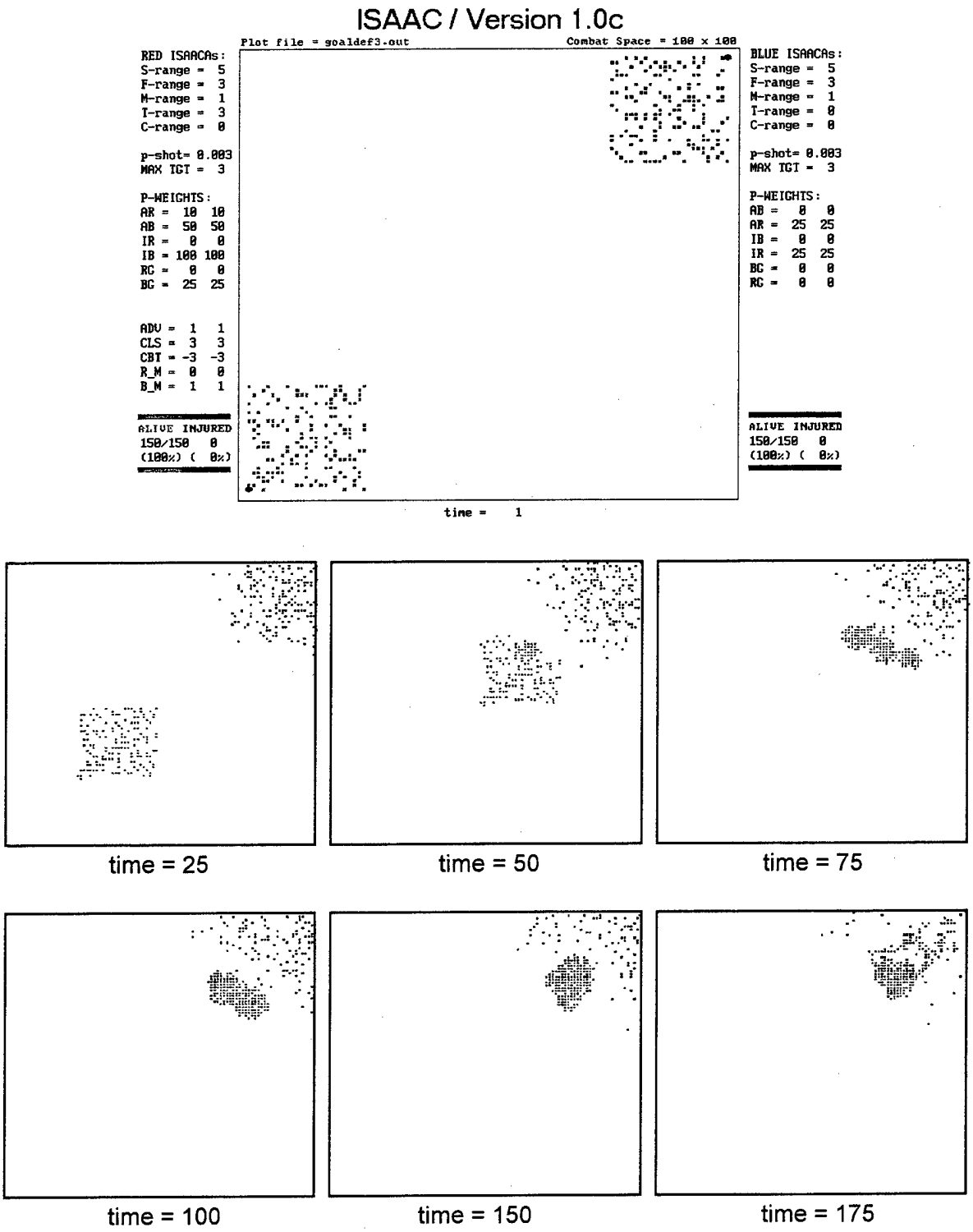
Figure 41 shows a few snapshots taken from a play-back of the file **GOALDEF2.out**.

This, and the previous, sample run illustrate how ISAAC can be used to explore simple "goal defense" scenarios. In this run, blue defends its own flag against a red attack. Note that blue has a very simple personality. In particular, since only the *second* (**AR** under **P-WEIGHTS** in figure 41) and *fourth* (**IR** under **P-WEIGHTS** in figure 32) components of blue's personality weight vector are non-zero, blue "sees" only enemy ISAACAs. Since, initially, all blue ISAACAs are only surrounded by other blue ISAACAs, until red forces come within range of the blue side, the blue ISAACAs effectively perform a "random walk" around their starting positions (since all possible moves incur exactly the same penalty). In contrast, the red force responds to both red and blue ISAACAs, though is "blind" to injured friendlies. Red and blue forces are endowed with equal *sensor* ($r_S = 5$) and *fire ranges* ($r_{F,red} = 3$), equal *single-shot probabilities* ($p = 0.005$) and can both simultaneously engage a maximum of 3 enemy targets. There are 150 ISAACAs per side.

What is interesting about this sample run is the unexpected, sudden *phase-transition*-like change of behavior that occurs a relatively long time "into" the close-combat that ensues near the blue flag. Upon reaching the outer area of blue's defensive posture (see snapshots for times t = 50 and t = 150), red at first fights blue in a tightly clustered formation. Red continues fighting in this cluster-mode for a relatively long time (see snapshots 75 through 150) until, suddenly, most of red's forces rapidly disperse outward and stream toward the blue flag.

The static snapshots shown in figure 41 do not do justice to the abruptness of this behavioral transition, which can be likened to turning on a "light" by a flicking a light switch. The abruptness of this transition is best appreciated by using **ISAAC_PB** to play back the file **GOALDEF2.out.**

Figure 41. Snapshot views of `GOALDEF2.out`



Sample Run #7: CIRCLE.out

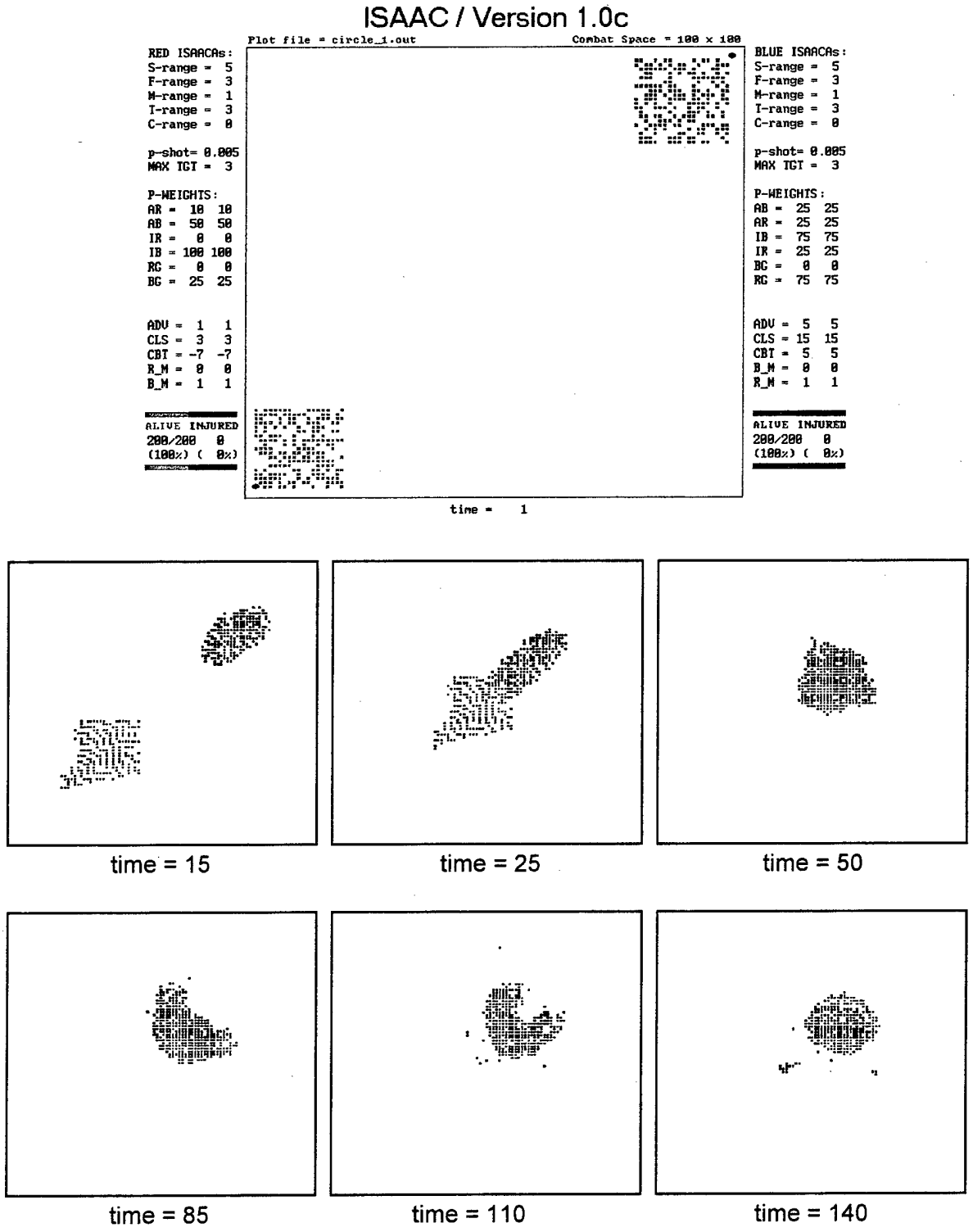
Figure 42 shows a few snapshots taken from a play-back of the file **CIRCLE.out**.

As can be seen from the parameter values shown in the snapshot of this run for time = 1, the red force is very *aggressive* (with a combat threshold of *negative* 7 and a strong propensity for moving toward *injured* blue) while the blue force is fairly *timid* (with a combat threshold of *positive* 5 and a propensity for moving toward *injured friendlies*). Red and blue forces are endowed with equal *sensor* ($r_S = 5$) and *fire ranges* ($r_{F,red} = 3$), equal *single-shot probabilities* ($p = 0.005$) and can both simultaneously engage a maximum of 3 enemy targets. There are 200 ISAACAs per side.

This sample run is noteworthy for two reasons: (1) the unexpected self-organized internal formation of red forces as they advance toward the blue flag (see snapshot for time t = 25), and (2) the emergent, and seemingly "intelligently orchestrated," encirclement of blue by red forces (see snapshots for times t = 85 through t = 140). Concerning the orderly fashion in which red ISAACAs march forward, keep in mind that this internal order arises *despite* the fact that ISAAC randomizes the order in which ISAACAs are sampled for making their moves on each iteration step. Note that this self-organized behavior is only partially revealed by the static snapshot in figure 42. In order to fully appreciate this point the user is urged to play back **CIRCLE.out** by using **ISAAC_PB**.

An interesting question to ask is "How should blue alter its personality (i.e., its "tactics") – *during the course of the battle* – in order to prevent being encircled by red forces?" While ISAAC can be used to explore the behavioral consequences of matching alternative fixed blue personalities against the same red force, ISAAC does not yet have the flexibility to explore the consequences of a dynamically changing personality (i.e., *meta*-personalities). Meta-personalities are planned to be incorporated into future versions of ISAAC.

Figure 42. Snapshot views of `CIRCLE.out`



Sample Run #8: FIRESTM1.out

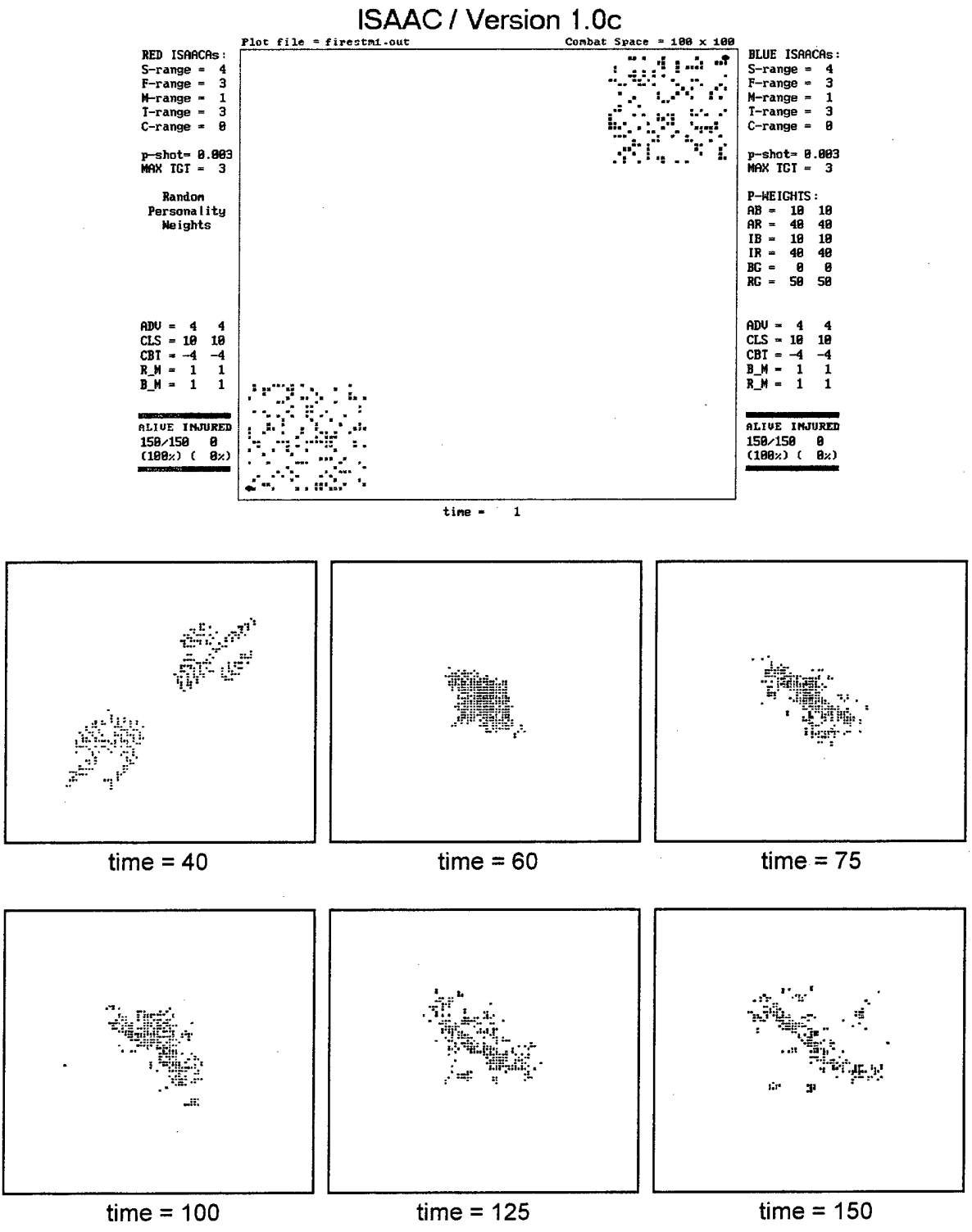
Figure 43 shows a few snapshots taken from a play-back of the file **FIRESTM1.out**.

Figures 43 and 44 are meant to be viewed together. Figure 44, which contains snapshots from a play-back of **FIRESTM2.out**, shows how exactly the same two force personalities shown in figure 43 unfold, except that blue ISAACAs are allowed to communicate with other blue ISAACAs.

As can be seen from the parameter values shown in the snapshot of this run for time = 1, the red and blue forces are virtually identical for this sample run: each is endowed with the same *sensor* ($r_S = 4$) and *fire ranges* ($r_{F,red} = 3$), each has the same *single-shot probabilities* ($p = 0.003$) and can simultaneously engage the same maximum of 3 enemy targets, and each obeys the same combat constraint conditions. Their personalities differ in that while each blue ISAACA is defined by the same personality weight vector $\vec{w} = (10, 40, 10, 40, 0, 50)$, each red ISAACA is defined by a different and random weight vector. There are 150 ISAACAs per side. Note that neither side uses communication.

The only noteworthy feature of this sample run is the overall, slightly disorganized, pattern of behavior that is to be contrasted with the behavior in figure 44. After the "collision" between the two forces at time t~60, the unfolding combat consists mainly of small, tightly clustered "firestorms" near the center of the battlefield. Neither side appears well organized, as both red and blue ISAACAs can be seen migrating from firestorm to firestorm. Now, skip ahead to figure 44, which shows the effects of endowing one side with an ability to communicate.....

Figure 43. Snapshot views of `FIRESTM1.out`



Sample Run #9: FIRESTM2.out

Figure 44 shows a few snapshots taken from a play-back of the file **FIRESTM2.out**.

Figures 43 and 44 are meant to be viewed together. Figure 43, which contains snapshots from a play-back of **FIRESTM1.out**, shows how exactly the same two force personalities shown in figure 44 unfold, except that *neither* the red nor blue forces have communications.

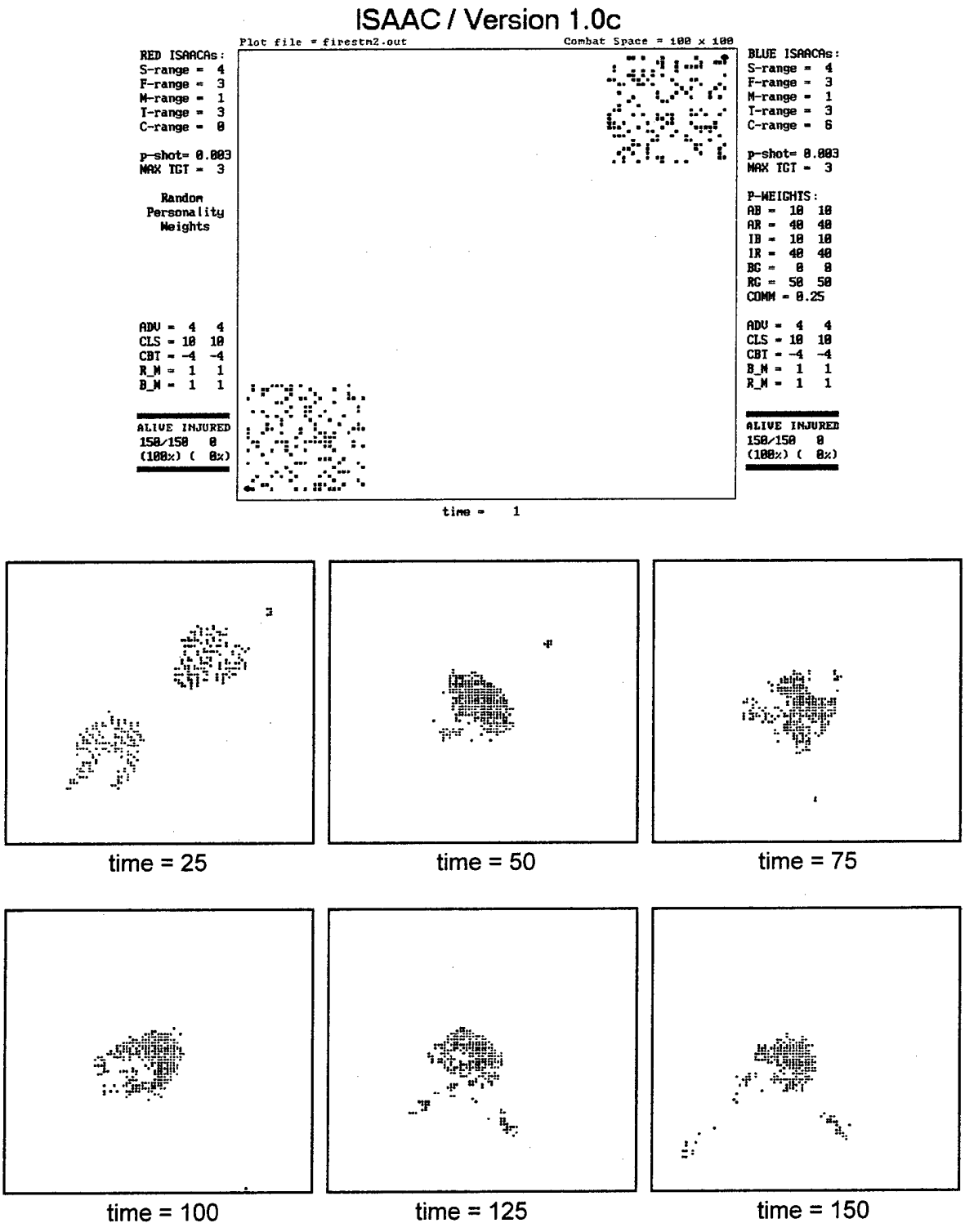
As can be seen from the parameter values shown in the snapshot of this run for time = 1, the red and blue forces are virtually identical for this sample run: each is endowed with the same *sensor* ($r_S = 4$) and *fire ranges* ($r_{F,red} = 3$), each has the same *single-shot probabilities* ($p = 0.003$) and can simultaneously engage the same maximum of 3 enemy targets, and each obeys the same combat constraint conditions. Their personalities differ in that while each blue ISAACA is defined by the same personality weight vector $\vec{w} = (10, 40, 10, 40, 0, 50)$, each red ISAACA is defined by a different and random weight vector. There are 150 ISAACAs per side.

The major difference between the red and blue forces in this sample run is that the blue ISAACAs are able to *communicate* with one another. In particular, blue's communications range (or C-RANGE) is equal to 6, and blue's communications weight $w_{comm} = 0.25$. This means that each blue ISAACA uses not only the information that is aware of in its own filed-of-view (out to a sensor range $r_S = 4$), but information communicated to it by all friendly ISAACAs within a range $r_{comm} = 6$. This additional information is assigned one-fourth the weight relative to information supplied by own-sensor. As in figure 43, the red side does not use communications.

Comparing the behavior as it unfolds in this sample run to that shown in figure 43, we see that the pattern this time is, overall, better organized. Instead of the small, tightly clustered "firestorms" that characterize the run in the absence of blue communications, here blue is able to maintain a strong organized central presence. No blue is allowed to stray too far from the area of the most intense combat, and few isolated "firestorms" appear, as they had in the earlier example.

It is interesting to note that while red undeniably "follows" blue's lead throughout the encounter (a fact that is much better illustrated by playing back the file **FIRESTM2.out** with **ISAAC_PB**) – that is to say, that blue initiates an action to which red immediately responds – red also appears to be better organized than in the previous example. This, despite the fact that red *does not use communications in either example.* The rudimentary "lesson learned" is that when one side unilaterally enhances its internal organization, that action may – ironically – enhance the apparent organization of *both sides.*

Figure 44. Snapshot views of `FIRESTM2.out`



Sample Run #10: SENSOR.out

Figure 45 shows snapshots at times t = 20, 40 and 60, taken from play-backs of files **SENSOR_1.out**, **SENSOR_2.out**, **SENSOR_3.out**, and **SENSOR_4.out**. These play-backs appear together here to illustrate the effects of *systematically increasing one side's (namely, red's) sensor range* while keeping the other side's parameters fixed.

The scenario for this sample run is as follows. The red force consists of N=100 ISAACAs, and is fairly aggressive, with a combat threshold of *negative* 4. The blue force consists of N=50 ISAACAs, and is less aggressive than red, with a combat threshold of *zero*. Both sides are endowed with the same *fire range* (r_F = 4), the same *single-shot probability* (p = 0.005) and can simultaneously engage the same maximum of 3 enemy targets. Note that the flags for this sample run are located near the middle of the left and right edges of the notional battlefield.

The four rows in figure 45, from top to bottom (appearing below the initial state shown at the top of the figure), show snapshots of runs in which red's sensor range is systematically increased in increments of two: $r_{S,\,red}$ = 3 for **SENSOR_1.out**; $r_{S,\,red}$ = 5 for **SENSOR_2.out**; $r_{S,\,red}$ = 7 for **SENSOR_3.out**; and $r_{S\,red}$ = 9 for **SENSOR_4.out**. Note that blue's sensor range, $r_{S,blue}$, *remains fixed* at $r_{S,blue}$ = 5 throughout.

The snapshots for **SENSOR_1.out** show that when red's sensor range is *less than* blue's, red is effectively able to "barrel" its way through the blue defenses into blue's flag, as it is unconcerned about – or, more likely, simply *unaware* of, because of its relatively short sensor range – the surrounding blue forces.

The snapshots for **SENSOR_2.out** show that when red's sensor range is set *equal to* blue's, red is no longer able to penetrate blue's defense as swiftly as in the previous run. Since red ISAACAs here have an longer sensor range, more of them are able to sense – and are therefore forced to respond to – the surrounding blue forces.

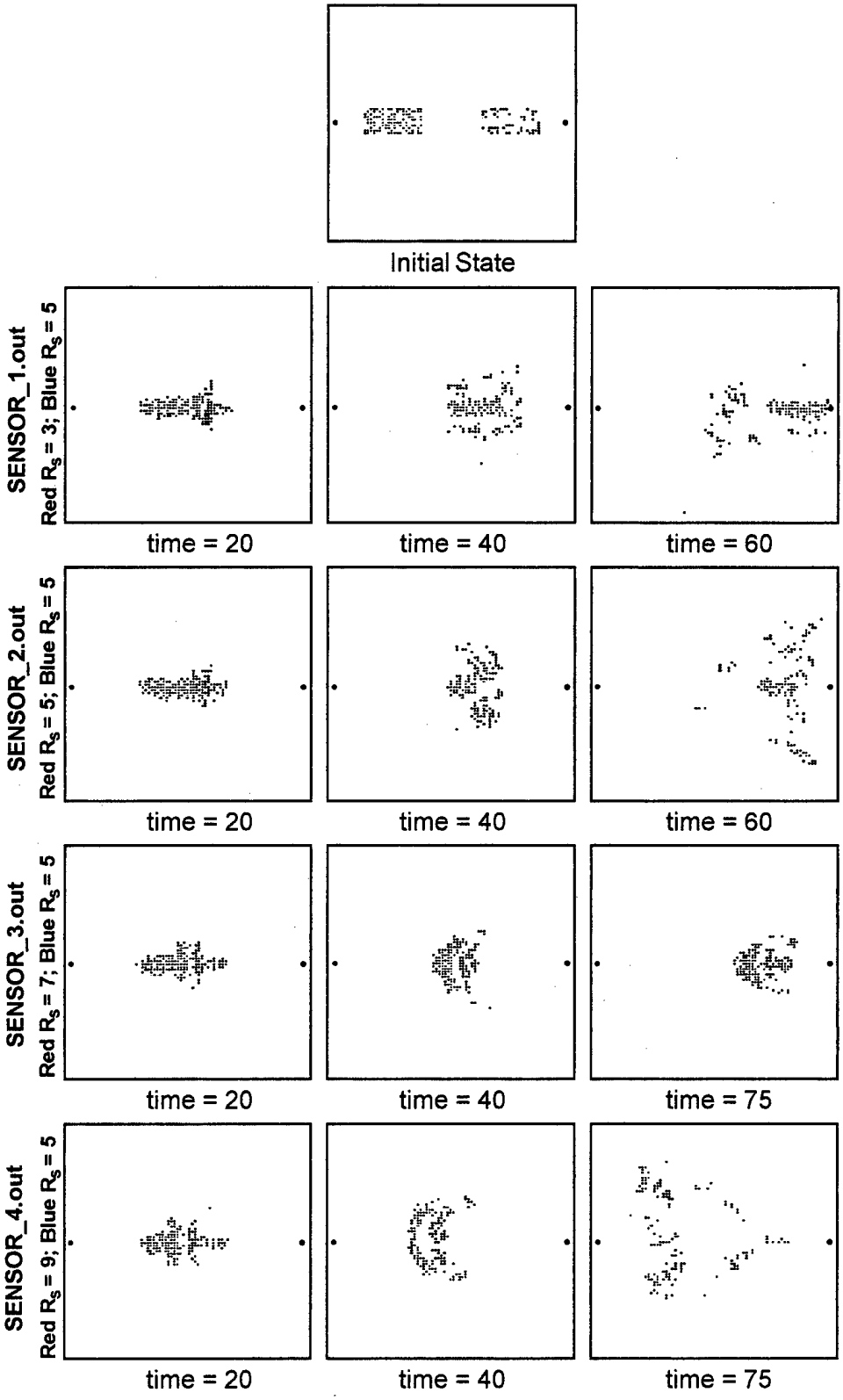
The snapshots for **SENSOR_3.out** show that when red's sensor range is *two units greater than* blue's, red is not only able to mass almost all of its forces on the blue flag (a later snapshot would reveal blue's flag completely enveloped by red forces by time t=100), but to defend its own flag from all blue forces as well. In this instance, the red force *knows enough about, and can respond quickly enough to* enemy action such that it is able to march into enemy territory effectively unhindered by enemy forces and "scoop up" blue ISAACAs as they are encountered.

What happens when red's sensor range is increased still further? One might intuitively guess that red can only do at least as well; certainly no worse. However, as the snapshots for **SENSOR_4.out** show, where red's sensor

range is increased to $r_{S\ red} = 9$, red actually does *worse* than it did in any of the preceding runs. "Worse" here means that red is less effective in (a) establishing a presence near the blue flag, and (b) defending blue's advance toward the red flag.

There is, perhaps, a fundamental lesson to be drawn from this simple example: *given that the resources and personalities of both sides remain fixed in a conflict, how "well" side X does over side Y does not necessarily scale monotonically with X's sensor capability.* As one side is forced to assimilate more and more information (with increasing sensor range), there will inevitably come a point where the available resources will be spread too thin and the overall fighting ability will therefore be curtailed. On the other hand, the deeper lesson here might be that as sensor range is increased – thereby increasing the amount of "information" that side X is forced to assimilate and respond to – *X's resources and/or tactics (i.e., "personality") must also be altered in order to ensure at least the same level of "mission success."*

Figure 45. Snapshot views of `SENSOR_X.out` (X=1,2,3,4)



Sample Run #11: LOCALCMD.out

This sample run illustrates a simple scenario in which the red force is endowed with *one local commander* (LC). The blue force is, as in all preceding examples, strictly decentralized.

Figure 46 shows a fragment of the input data file used to generate this run. This fragment is used to define the parameters of the red local commanders. Figure 46 shows that there are three LCs (**num_RED_comdrs** = 3). Each LC has 20 subordinate ISAACAs under his command ((1)**_R_undr_cmd** = 20)[19] and reacts only to enemy ISAACAs (**w2:alive_B** = **w4:injrd_B** = 35) and the enemy flag (**w6:B_goal** = 50). The LCs in this sample run are also fairly timid, both as individuals (with a combat threshold of *positive* 25) and as commanders (with *negative* command weights). Negative local command weights mean that the LCs tend to send their subordinate ISAACAs *away from* (rather than toward) areas in which the red forces are outnumbered by blue. Figure 47 shows a few snapshots taken from a play-back of the run stored in **LOCALCMD.out**.

Figure 46. Fragment of **LOCALCMD.dat** input data file

```
*******************************
* RED LOCAL COMMAND PARAMETERS
*******************************
RED_command_flag  1
num_RED_comdrs    3
R_patch_type      1
R_patch_flag      2
*
* local commander parameters
*
(1)_R_undr_cmd    20
(1)_R_cmnd_rad    2
(1)_R_SENSOR_rng  7
*
* local commander personality
*
(1)_w1:alive_R    0.000000
(1)_w2:alive_B    35.000000
(1)_w3:injrd_R    0.000000
(1)_w4:injrd_B    35.000000
(1)_w5:R_goal     0.000000
(1)_w6:B_goal     50.000000
*
* local commander constraints
*
(1)_R_ADV_range   4
(1)_ADVANCE_num   0
(1)_CLUSTER_num   0
(1)_COMBAT_num    25
*
* local command weights
*
(1)_R_w_alpha     -1.
(1)_R_w_beta      -1.
(1)_R_w_delta     -1.
(1)_R_w_gamma     -1.
*
* global command weights
*
(1)_w_obey_GC_def 1.
```

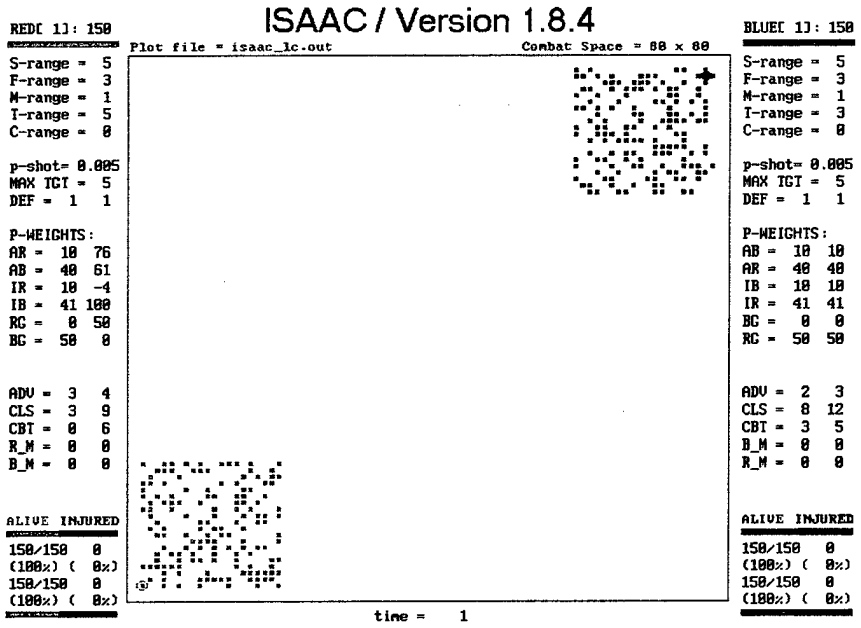
Figure 47 shows a few snapshots taken from a play-back of the file **LOCALCMD.out** (using **ISAAC_CE**). The images shown here can be

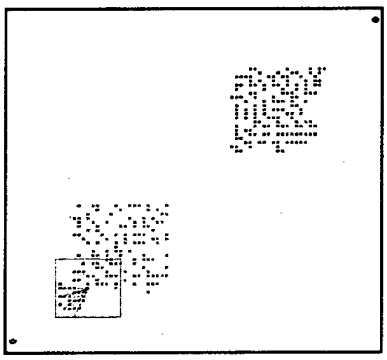
[19] The data file fragment in figure 46 shows only the parameters values of the *first* local commander. The other two local commanders are identical.

reproduced interactively by pressing the "C" hot-key toggle-switch (see *"Hot-Key" Menu*) a few times to display the LC's command area (i.e., the box surrounding the black "dot" representing the LC) and the LC's subordinate ISAACAs (i.e., the maroon colored "dots" that are "tethered" to the LC).

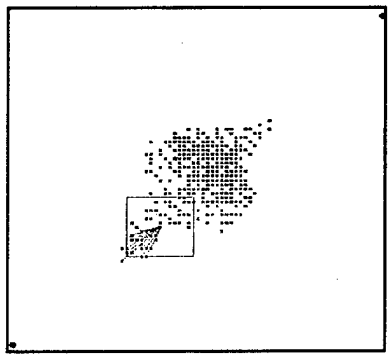
Figure 47 shows that as soon as the LC encounters enemy forces at the periphery of its command area (see snapshot at t = 30), it moves away from them (as well as the ensuing combat near the center of the notional battlefield), and directs its subordinates to follow. From that point on, the LC is able to stay clear of all enemy ISAACAs, and thereby steer its subordinate ISAACAs from harm's way. After most of the fighting near the center of the battlefield has ended, the LC finally "sees" a clear path toward the blue flag (see snapshot at t = 155).

Figure 47. Snapshot views of `LOCALCMD.out`

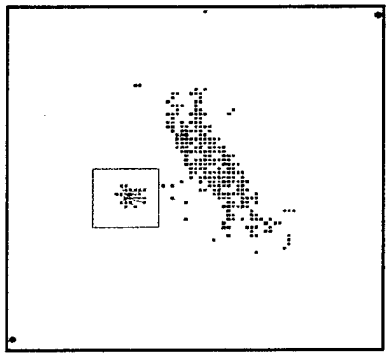




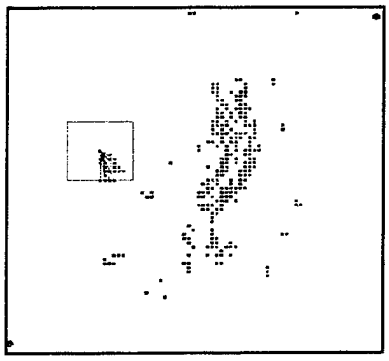
time = 15



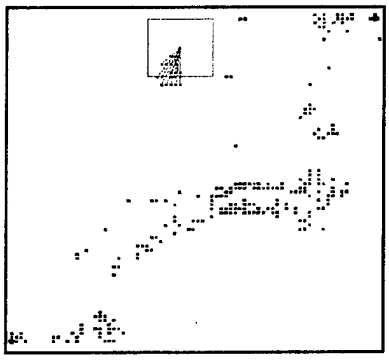
time = 30



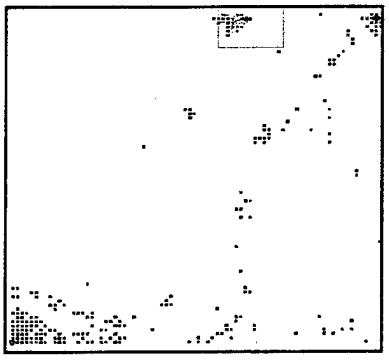
time = 60



time = 80



time = 105



time = 155

Sample Run #12: GLBALCMD.out

Figure 48 shows a fragment of the input data file used to generate sample run #12. This fragment is used to define the parameters of the red global commander. Figure 49 shows a few snapshots taken from a play-back of the run stored in **GLBALCMD.out**.

Figure 48. Fragment of **GLBALCMD.dat** input data file

```
*****************************
 RED GLOBAL COMMAND PARAMETERS
*****************************
ED_global_flag  1

                direction parameters

                                          C_fear_index     1.
C_w_alpha       1.
C_w_beta        1.
C_frac_R[1]     .3
C_frac_R[2]     .6
C_w_swath[1]    1.
C_w_swath[2]    1.
C_w_swath[3]    1.

                   help parameters

                                      C_max_red_f     2.5
C_help_radius   40
C_h_thresh      .1
C_rel_h_thresh  1.5
```

The initial state and almost all parameters defining the red and blue forces are the same as in the previous sample run (**LOCALCMD.out**). One important difference, however, is that whereas the red local commanders were all very timid in **LOCALCMD.out** – as exemplified by them all having *negative* command weights (see figure 46) – here the local commanders all have *positive* weights. This means that the local commanders in this sample run always send their subordinate ISAACAs *toward* (rather than *away from*, as in the previous sample run) the area in greatest need of red firepower.

In contrast, the global commander is very timid. Figure 48 shows that the GC's **GC_fear_index** is set equal to one, meaning that the GC will vector his local commanders toward the blue flag if and only if he finds a "sector" pointing at the blue flag that is completely free of blue ISAACAs.[20]

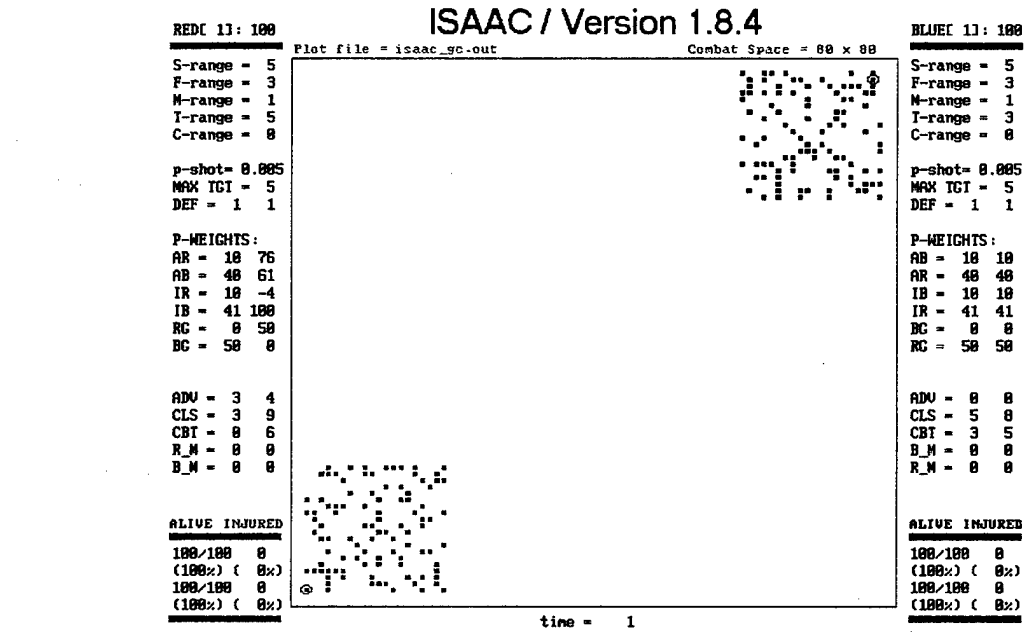
Figure 49 shows that the GC is able to keep his local commanders (and therefore their own subordinate ISAACAs) away from harm's way until the blue force moves close to the red flag. Note that the cluster of red

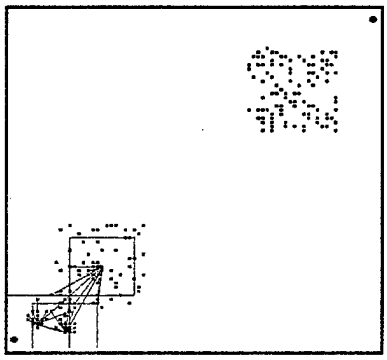
[20] "Sectors" and other parameters defining local and global command are discussed in **Command and Control**.

ISAACAs that collide with the blue force at time t = 25 are not under either local or global command.

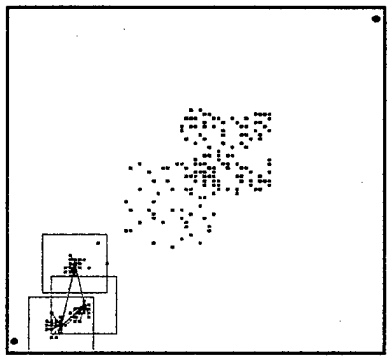
If **GLBALCMD.out** is played back interactively using **ISAAC_CE**, recall that the "C" hot-key acts as a toggle switch for displaying various command-related parameters on screen. Figure 49 shows both a command-box surrounding each local commander (representing the LC's field-of-view and area of responsibility) and a thin dark tether connecting the three LCs (reminding the user that each is under the command of a global commander).

Figure 49. Snapshot views of `GLBALCMD.out`

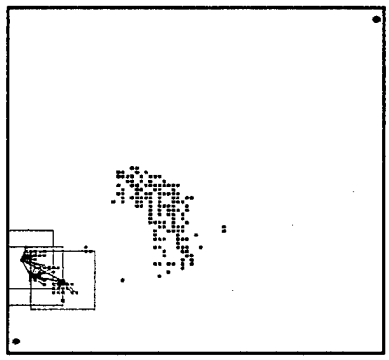




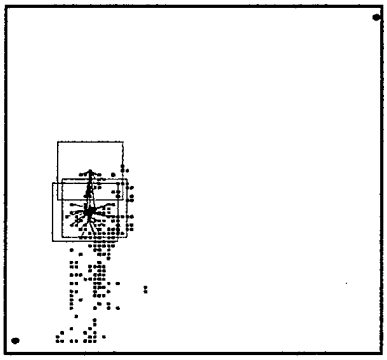
time = 10



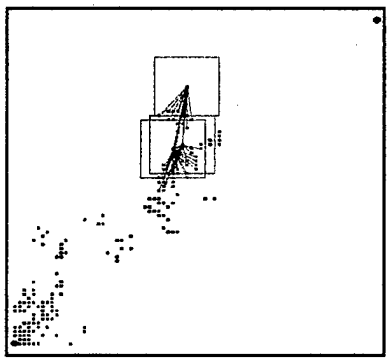
time = 25



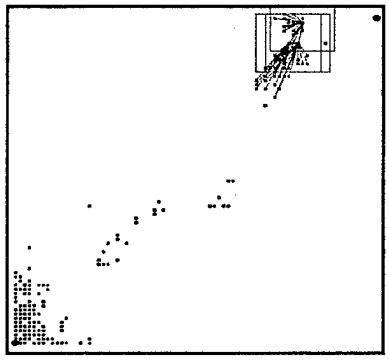
time = 50



time = 75



time = 100



time = 125

Sample Run #13: BATTLE1.out

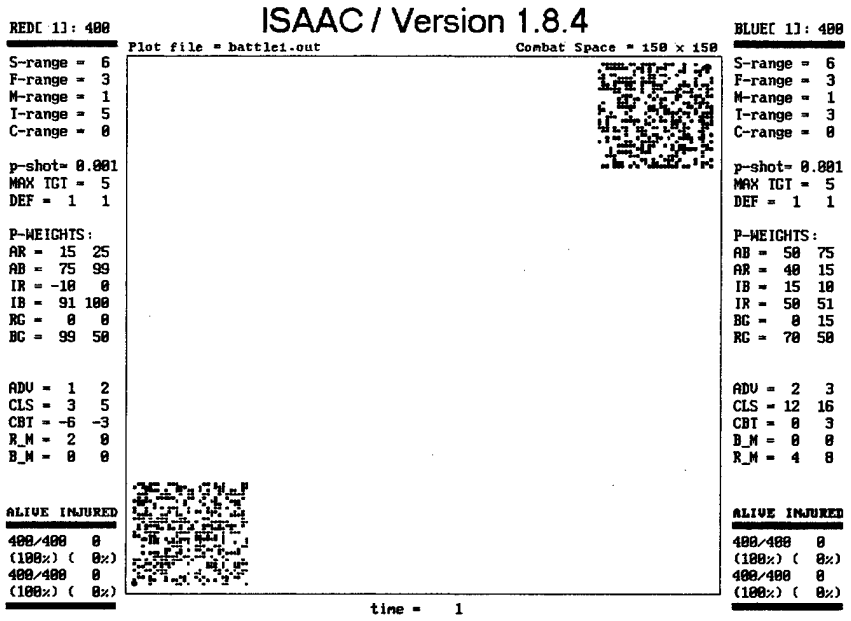
Figure 50 shows a few snapshots taken from a play-back of the file **BATTLE1.out**.

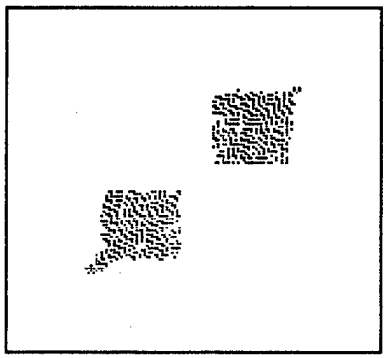
This last run is an example of what a relatively large ISAACian "battle" looks like. In this battle, a large number of ISAACAs (400 per side) engage first in close combat, then in an increasing number of smaller local skirmishes that eventually spread to all parts of the notional battlefield.

The red force is very aggressive, with a strong propensity for moving toward enemy forces and a combat threshold of *negative* 6 (see parameter values in the snapshot for time = 1). The blue force is less aggressive than red, with a slightly higher propensity for moving outward alive blue rather then alive red ISAACAs and a combat threshold of *zero*. Blue ISAACAs are also more prone to cluster than reds. Both sides are endowed with the same *sensor range* (r_S = 6), *fire range* (r_F = 3), the same *single-shot probability* (p = 0.001) and can simultaneously engage the same maximum of five enemy targets.

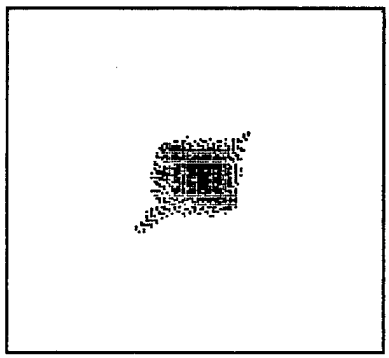
Figure 50 shows that, after the collision between the two forces takes place some time before the snapshot at time = 60, the battle evolves in essentially two distinct phases. The first phase consists of tight, close combat that – except for a slight "fraying" at the edges – is entirely confined to the central region of the battlefield. The second phase, which begins some time before the snapshot at time = 115, consists of an increasing number of much smaller-scale skirmishes spreading outward from what used to be the area of close combat. While not immediately clear from figure 50, there is complicated pattern of maneuvers on both sides, as red and blue ISAACAs continually shift over from one skirmish to another as the battle unfolds. This run may be viewed by using **ISAAC_CE** to play back the file **BATTLE1.out**.

Figure 50. Snapshot views of `BATTLE_1.out`

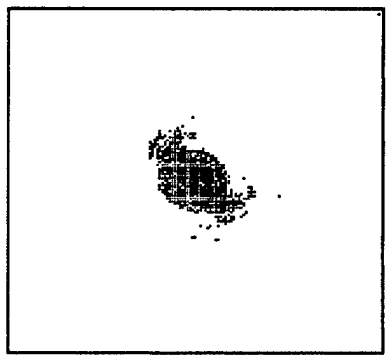




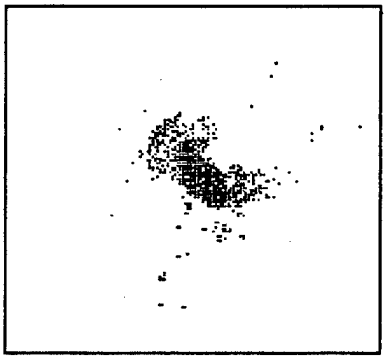
time = 40



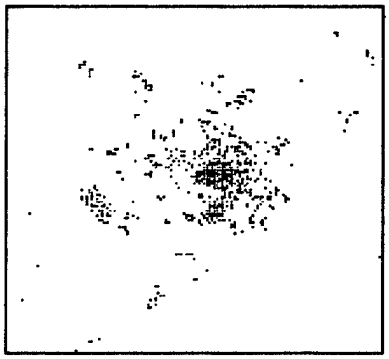
time = 60



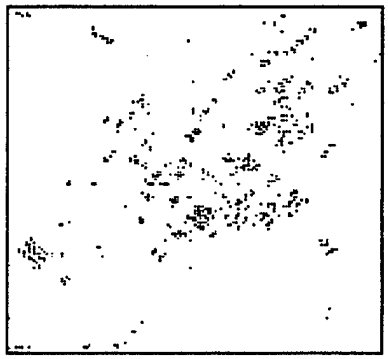
time = 85



time = 115



time = 150



time = 175

Data Collection

So far, we have discussed only how ISAAC can be used to display (or play back) the time evolution of a given data input file. In this simple visual mode of operation – which can be thought of as a purely *syntactic* representation of the unfolding behavior, in that it displays the exact symbology of what is happening and *no more* – the user chooses a new data file, or selectively alters the contents of an existing one, and runs ISAAC to display the resulting dynamics. While such interactive runs are obviously very useful in identifying a variety of emergent behaviors (see ***Sample Runs***), and can therefore be quite provocative – on a *qualitative* level – they do not by-themselves constitute any direct *quantitative* evidence of any sort of behavior. For this, other, more *semantically* oriented, measures are required; measures that provide not just a record of what is happening but an interpretation of why it is happening.

To this end, ISAAC provides a capability to (1) generate time series of various changing quantities describing the step-by-step evolution of a battle, and (2) keep track of certain measures of "how well" mission objectives are met at a battle's conclusion. The former (using built-in statistics measures; see below) yields quantitative snapshots of a battle as it unfolds in time; the latter (using a simple parameter-space mapping technique; see below) yields semi-quantitative measures of "success" at a mission's end.

Built-in Statistics

ISAAC is equipped with a rudimentary data collection capability. Specifically, ISAAC's *Core Engine* (i.e., the program **ISAAC_CE**; see table 4 in ***Overview of ISAAC***)[21] has facilities to calculate seven basic classes of information (each as a function of time; see discussion below):

- ***Class 1:*** *Force sizes*
- ***Class 2:*** *ISAACA interpoint distance distributions*
- ***Class 3:*** *ISAACA neighbor-number distributions*
- ***Class 4:*** *ISAACA:enemy-flag interpoint distance distributions*
- ***Class 5:*** *ISAACA cluster-size distributions*
- ***Class 6:*** *Center-of-mass positions*
- ***Class 7:*** *Spatial Entropy*

[21] ISAAC's single-squad version **ISAAC_SQ** has exactly the same data collection capabilities as the full multi-squad version.

Data collection is enabled by setting the **stat_flag** variable appearing in ISAAC's input data file equal to 1 (see ***Statistics Parameters*** in ***Contents of ISAAC's Data Input File***). The calculation of each of the above classes of statistics is toggled by individual parameter "flags" appearing in the statistics parameters section of ISAAC's input data file.

The total output of ISAAC's data collection routines (assuming all statistics flags are set equal to 1) is distributed among 21 consecutively numbered files **STATS_1.dat** through **STATS_21.dat**. Table 8 provides a brief description of their contents. A complete listing of the data fields appearing in each of these output files is given in ***Appendix E: STATS_X.dat Data Fields*** below.

In addition to these 21 output data files, the user also has the option of using the auxiliary *parameter-space mapper* program **ISAAC_PM** to effectively "map-out" the dynamical behavior over a two-dimensional slice of ISAAC's total N-dimensional parameter space using certain (user-defined) measures of behavior (see ***Taking 2D "Slices" of ISAAC's Parameter Space*** below).

Table 8. Description of ISAAC's data output files

Class of data	Appropriate "flag"[1]	Associated Output files
Force sizes	**stat_flag**	**STATS_1.dat**
ISAACA interpoint distance distributions	**interpoint_flag**	**STATS_2.dat – STATS_6.dat**
ISAACA:enemy-flag interpoint distance distributions	**interpoint_flag**	**STATS_7.dat** & **STATS_8.dat**
ISAACA neighbor-number distributions	**neighbors_flag**	**STATS_14.dat – STATS_19.dat** & **STATS_21.dat**
ISAACA cluster-size distributions	**cluster_1_flag** & **cluster_2_flag**	**STATS_10.dat – STATS_13.dat**
Center-of-mass positions	**center_mass_flag**	**STATS_20.dat**
Spatial entropy	**entropy_flag**	**STATS_9.dat**

[1] This refers to variables appearing in the statistics parameters section of ISAAC's data input file (see ***Contents of ISAAC's Data Input File***)

Classes of Data

As mentioned above, there are seven classes of statistical information that ISAAC is able to keep track of during a run. The following sections contain brief descriptions of each class.

Class 1: Force sizes

The first class of data consists of keeping track of basic red and blue ISAACA force strengths, measured as the remaining fractions of the original force size. Separate measures are provided for alive red, alive blue, injured red, injured blue and total red and total blue forces.

For data field descriptions see ***STATS_1.dat*** in ***Appendix E: STATS_X.dat Data Fields.***

Class 2: ISAACA interpoint distance distributions

The second class of data consists of keeping track of the averages and distributions of the distances between pairs of ISAACAs. Separate measures are provided for red:red pairs, red:blue pairs, blue:blue pairs and distances between red ISAACAs and the blue flag and blue ISAACAs and the red flag.

For data field descriptions see ***STATS_2.dat*** through ***STATS_6.dat*** in ***Appendix E: STATS_X.dat Data Fields.***

Class 3: ISAACA neighbor-number distributions

The third class of data consists of keeping track of the averages and distributions of the number of neighbors that red and blue ISAACAs have that are within a range R=1, 2, ..., 5 of them. Separate measures are provided for red, blue and all (either red or blue) ISAACAs near *red* ISAACAs, red, blue and all (either red or blue) ISAACAs near *blue* ISAACAs, and red and blue ISAACAs near both red and blue flags.

For data field descriptions see ***STATS_14.dat*** through ***STATS_19.dat*** and ***STATS_21.dat*** in ***Appendix E: STATS_X.dat Data Fields.***

Class 4: ISAACA:enemy-flag interpoint distance distributions

The fourth class of data consists of keeping track of the averages and distributions of the distances between red and blue ISAACAs and their enemy flags (i.e., between red ISAACAs and the blue flag and blue ISAACAs and the red flag).

For data field descriptions see ***STATS_7.dat*** and ***STATS_8.dat*** in ***Appendix E: STATS_X.dat Data Fields.***

Class 5: ISAACA cluster-size distributions

The fifth class of data consists of keeping track of the averages and distributions of the sizes of clusters of ISAACAs, using inter-cluster distance criteria of D=1 and D=2. (An inter-cluster distance criteria of D=d means that two ISAACAs that are within a distance d of each other are defined to belong to the same cluster.) Because this class of data provides an insight into the gross structural appearance of the entire

battlefield, it can be thought of as a crude pattern recognition measure. Another such measure is provided by spatial entropy (see below). Appendix F contains a heuristic description of the *Hoshen-Kopelman* algorithm used to calculate the cluster distribution.

For data field descriptions see ***STATS_10.dat*** through ***STATS_13.dat*** in ***Appendix E: STATS_X.dat Data Fields.***

Class 6: Center-of-mass positions

The sixth class of data consists of keeping track of the (x,y) coordinates of the center-of-mass position of the red, blue and total (i.e., red + blue) force, as well as the distances between the red and blue forces and both flags. The center of mass of the red force at time t, for example, is defined by coordinates $x_{red,\,CM}(t)$ and $x_{blue,CM}(t)$ given by

$$x_{\mathrm{red,CM}}(t) = \frac{1}{N_{\mathrm{red}}(t)} \sum_{i=1}^{N_{\mathrm{red}}(t)} x_{\mathrm{red},i}(t) \quad \text{and} \quad y_{\mathrm{red,CM}}(t) = \frac{1}{N_{\mathrm{red}}(t)} \sum_{i=1}^{N_{\mathrm{red}}(t)} y_{\mathrm{red},i}(t)\,,$$

where $x_{red,\,i}(t)$ and $y_{red,\,i}(t)$ are the x and y coordinates of the i^{th} red ISAACA at time t, respectively and $N_{red}(t)$ is the total number of red ISAACAs at time t.

For data field descriptions see ***STATS_20.dat*** in ***Appendix E: STATS_X.dat Data Fields.***

Class 7: Spatial entropy

The seventh class of data consists of keeping track of the *spatial entropy* of the configuration of the red, blue and total (i.e., red + blue) force.

Spatial entropy provides a measure of the degree of disorder of a battlefield state. For example, a large group of tightly clustered ISAACAs is relatively highly "organized" and therefore has low entropy. In contrast, a battlefield that consists of many and widely dispersed small groups of ISAACAs is relatively "disorganized" and thus has a high entropy.

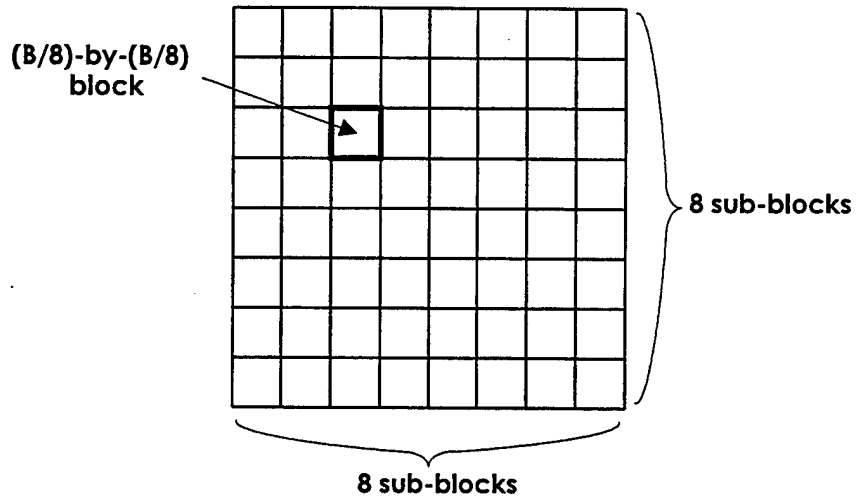
Spatial entropy $E = E(b)$, where b is the size of the sub-block of the (B/b)-by-(B/b) array of sub-blocks into which the battlefield is partitioned, and B is the length of the battlefield (see figure 51). ISAAC performs separate calculations using b = 4, 8 and 16. E(b) is defined as follows:

$$E(b) = -\frac{1}{2\log_2 b} \sum_{i=1}^{b^2} p_i(b)\,\log_2 p_i(b)\,,$$

where $p_i(b) = N_i(b) / N$, $\log_2 x$ is the logarithm *base-2* of x, N is the total number of ISAACAs (red + blue) on the battlefield, b^2 is the number of sub-blocks into which the battlefield is partitioned, and the factor appearing before the summation sign, $(2\log_2 b)^{-1}$, is a normalization constant. Note that $p_i(b)$ effectively gives the probability of finding an ISAACA in the i^{th} sub-block.

Observe that if a single sub-block contains all the points, then the spatial entropy has its minimal possible value: $E(b) = 0$. On the other hand, if the ISAACAs are all scattered throughout the battlefield in such a way that $p_i(b) = p = 1/b^2$ for all sub-blocks i, then the spatial entropy takes its maximal possible value: $E(b) = 1$. The closer the value of $E(b)$ is to *zero*, the "closer" the ISAACA distribution is to one that is tightly clustered near a single sub-block. The closer the value of $E(b)$ is to *one*, the "closer" the ISAACA distribution is to one that is completely scattered throughout the entire battlefield.

Figure 51. Battlefield partitioned into an 8-by-8 array of sub-blocks of size B/8



For data field descriptions see ***STATS_9.dat*** in ***Appendix E: STATS_X.dat Data Fields.***

Sample Output

Consider the fairly complex ISAAC evolution depicted in figures 52 and 53. The values of the parameters defining the single-squad red and blue forces are shown on the left and right of figure 52, respectively. There are 200 ISAACAs per side and both sides are relatively aggressive: red's *combat threshold*[22] being equal to -3 and blue's to -6. Note also that for this scenario, both *fratricide* and *reconstitution* options have been enabled, with a reconstitution time t_{recon} = 15 steps. The fratricide option means that whenever an ISAACA **X** targets an enemy ISAACA **Y** and *misses*, a friendly ISAACA **X'** that is near **Y** can inadvertently be targeted and hit by **X** instead. The reconstitution option means that if an alive (red or blue) ISAACA **X** is hit (either by enemy or friend) at time τ – so that **X** becomes *injured* – but is not hit during any time between $t = \tau+1$ and $t = \tau+t_{recon}$, **X**'s state is reconstituted back to *alive* at time $t = \tau+t_{recon}+1$.

Figure 53 shows a few snapshots of how the initial state shown in figure 52 unfolds in time. One can see that the behavior of these two "personalities" proceeds in essentially four stages. The first stage (referring roughly to times t = 1 through t = 20) consists of an initial internal jostling for position (on both sides) and a steady march toward the enemy corner. The second stage (between times t = 25 and t = 70) consists of "close combat" within a tight central cluster of closely packed red and blue ISAACAs. The third stage (between t = 80 and t = 120) is marked by a relatively rapid "expansion" of forces outward from the central region of the battlefield. The fourth, and final, stage (for times t > 140) consists mostly of small local skirmishes that are distributed throughout most of the battlefield.

Figures 54-a through 54-b provide a sampling of the kind of information contained in the statistics output files **STATS_1.dat** through **STATS_21.dat**. Figure 54-a, for example, shows a plot of blue force strength (expressed as a fraction of the original number of ISAACAs) as a function of time, plotted for t=1 through t=250 (note that the "snapshots" in figure 53 stop at t=150). Initially, and until the red and blue forces "collide" at time t~18, the blue ISAACAs are at their strongest. The bottom-most curve in figure 54-a, which shows the fraction of injured blues, begins to rise as soon as blue ISAACAs are targeted and hit by opposing reds. Notice that, because the reconstitution option has been enabled for this run (see above), the number of alive blues frequently increases during the course of the evolution.

Figure 54-b shows a plot of the average distance between red and blue ISAACAs as a function of time. The curve starts out at a distance d~57 that is equivalent to the initial separation distance between the starting

[22] See ***ISAACA Combat*** in ***Overview of ISAAC*** for a discussion of *combat threshold.*

"boxes" of the red and blue forces. As the two forces move toward their enemy's flag and thus approach one another, the average inter-force distance steadily diminishes, reaching a minimum as the two forces "collide" near the center of the battlefield. The relatively flat portion of the curve that appears between times t~25 and t~80 corresponds to the period of intense "close combat" discussed earlier and shown in early snapshots in figure 53. The curve in figure 54-b then begins a steep climb from values near ~15 (at $t < 80$) to values near ~35 (for $t > 130$), corresponding to the third stage of the evolution during which there is a relatively rapid "expansion" of forces outward from the central region of the battlefield. This third stage is next followed by a second plateau region (in which the average distance between red and blue ISAACAs is again relatively constant from t~130 to t~200).

Figure 54-c shows a plot of the average number of red ISAACAs that are within a distance D=1 (lower curve), D=3 (middle curve) or D=5 (upper curve) of blue ISAACAs. The largest relative numbers, of course, are found between the times t~25 and t~80 during which the scenario includes some very close combat near the central region of the battlefield. Compare the appearance of the plots in figure 54-b and figure 54-c during the later times t~150 through t~180, which is the interval of time shortly following the stage in which there is a rapid "expansion" of forces outward from the central region of the battlefield. We see that blue ISAACAs find themselves surrounded on average by a relatively greater number of enemy ISAACAs then before as the red forces effectively "organize" themselves for further combat after the "expansion." Notice that there is a second local rise in the average number of reds within blue corresponding to a small second "plateau" in figure 54-b between times t~215 and t~230.

Figure 54-d shows the trajectory of the blue forces' center-of-mass (COM) position. The path of the COM begins at (x,y) ~ (62,62), corresponding roughly to the center of the "box" containing the initial distribution of blue forces (see figure 52). Notice how this trajectory moves almost straight toward the red flag (which is located near the origin) but then *stops* and – due to the dynamics between red and blue forces – effectively *doubles back* toward the blue flag, never managing to get closer to the red flag than the coordinates (x,y) ~ (32,32).

Figure 54-e shows a plot of the spatial entropy as a function of time. Spatial entropy is here computed by partitioning the 80-by-80 battlefield into an 8-by-8 array of 10-by-10 sub-blocks, meaning that the "resolution" of this measure is effectively limited to 10-by-10 boxes. (see discussion in ***Spatial Entropy*** above). Recall that spatial entropy can be used as a crude pattern recognition tool: tight, relatively undispersed patterns having a low entropy; disorganized, scattered patterns having a high entropy. The measure is by no means perfect, but it can serve as useful descriptive tool to summarize and/or compare a large sampling of runs. In this case, we see that the relatively tightly clustered initial

distribution (red in one "box" and blue in another) yields an initial spatial entropy E ~ .46. It rises a bit for the first few iteration steps (in the *absence* of combat) due to the internal "jostling" of red and blue forces that widens the initial spread of forces. The entropy attains its minimal value E ~ .38 around t ~ 30 at the point at which the colliding red and blue forces have created the tightest cluster of combatants. The entropy then increases, reaching a maximum of E ~ .72 near t ~ 150 (the middle of the "plateau" region in figure 54-b) as the forces disperse throughout the battlefield. The entropy then falls back down to smaller values as the numbers of combating ISAACAs decreases and local skirmishes become better organized.

Figure 54-f provides an alternative pattern-recognition-like glimpse of the way in which the battle unfolds in time by plotting the average size of ISAACA clusters (with no distinction made between red and blue) as a function of time. Two ISAACAs that are within a distance d=1 apart of each other are said to belong to the same *cluster.* Since the initial condition consists of two far-separated random distributions of 200 ISAACAs per side, at time t = 0 the average cluster size is equal to 200. It drops as the opposing forces move toward their respective goals (and ISAACAs become temporarily separated as a result of internal "jostling" for position), and falls to a local minimum at t ~ 12, which is around the time the two forces first collide. The average cluster size begins to increase once again as the opposing forces "bunch up" near the center. It attains its largest local values between times t ~ 20 and t ~ 35, which correspond roughly to the period marking the most "intense fighting" of the scenario (as characterized, for example, by the steepest rise in the number of injured blues shown in the bottom-most curve in figure 54-a). As the forces are reduced by attrition and begin dispersing throughout the battlefield, the average cluster size steadily decreases. Notice, however, that the average cluster size begins decreasing at a significantly earlier time (t ~ 35) than the time at which forces begin "expanding" outward (at t ~ 80; see figure 54-b).

Figure 52. Parameters and screen shot of initial configuration for **BATTLE2.out**

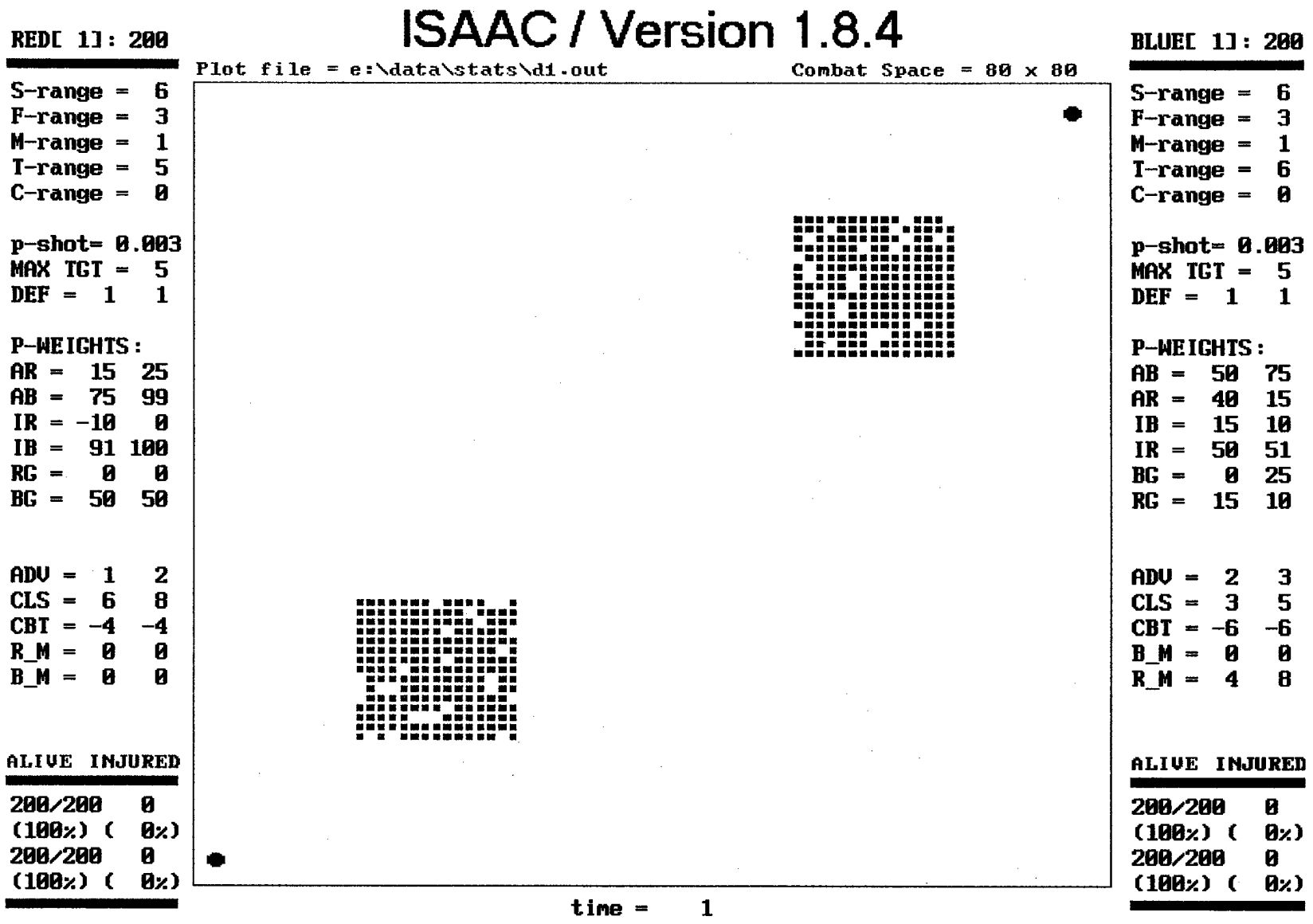


Figure 53. Snapshot views of `BATTLE2.out`

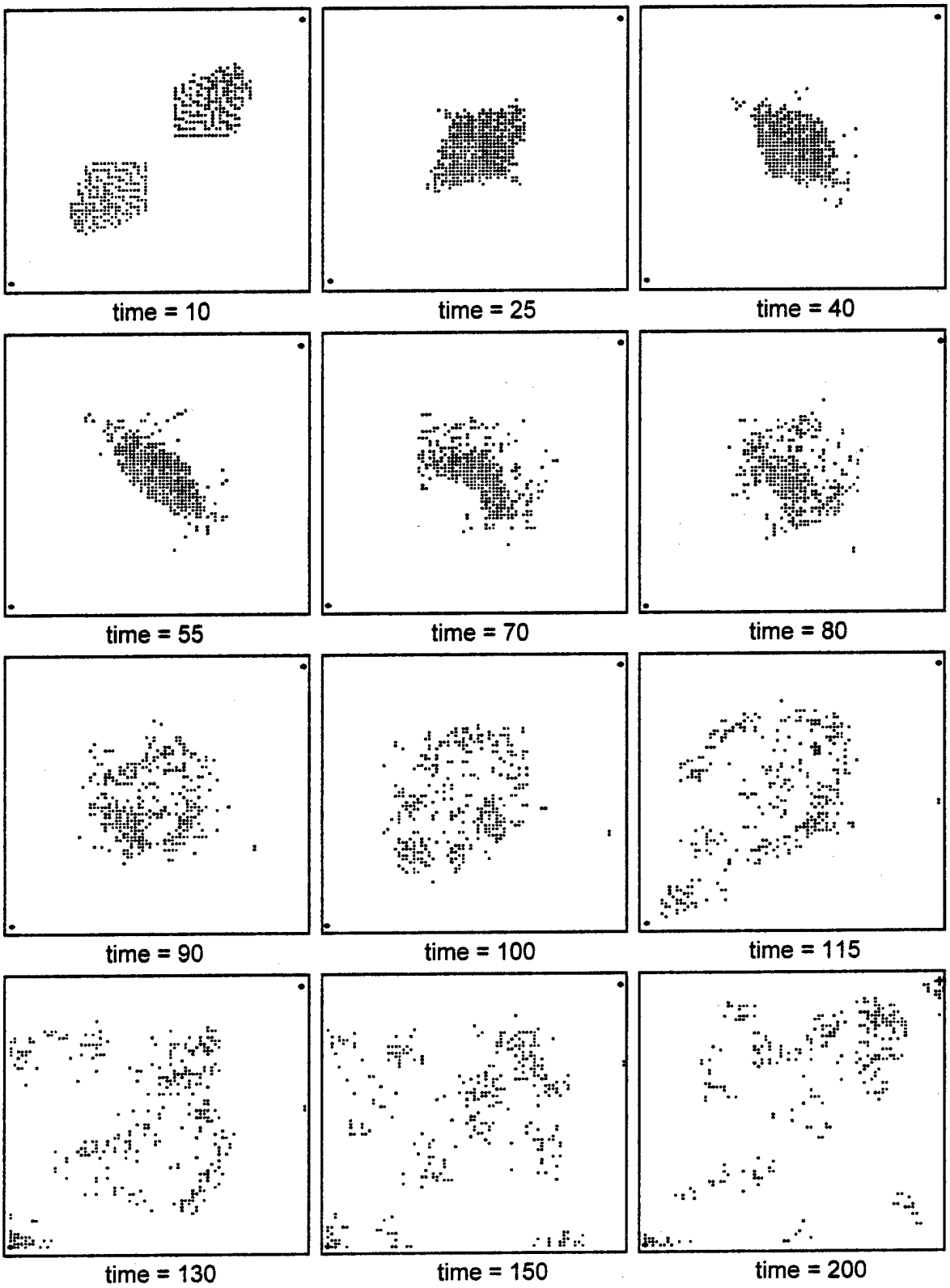
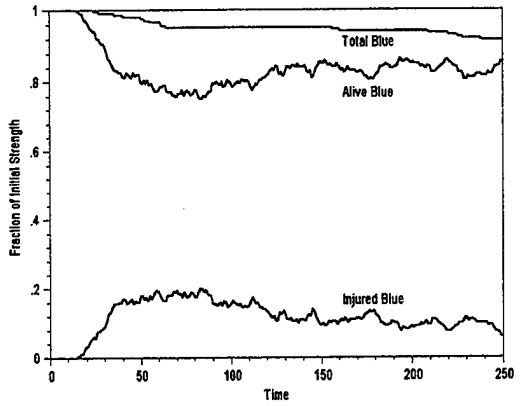
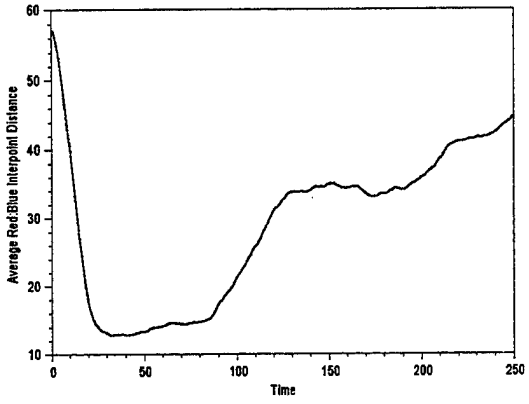


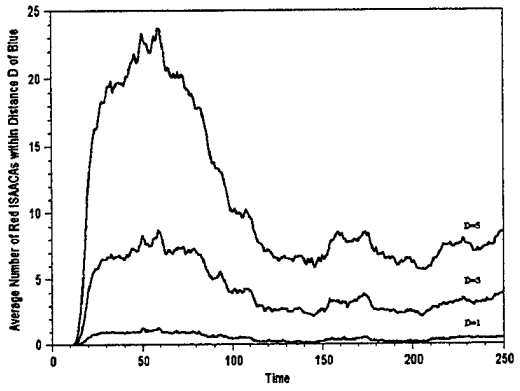
Figure 54. Sample statistics measures for **BATTLE2.out**



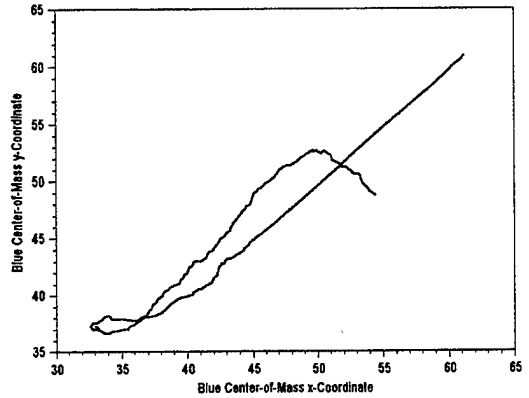
(a) Force Strengths



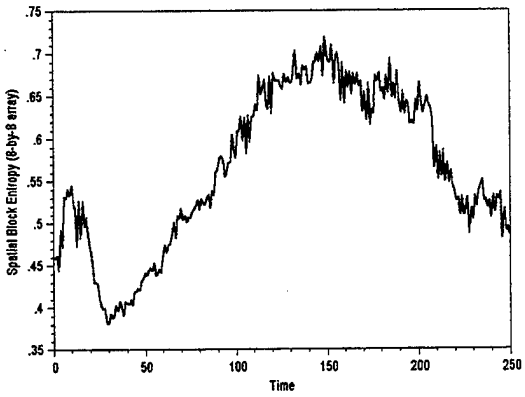
(b) Red:Blue Interpoint Distance



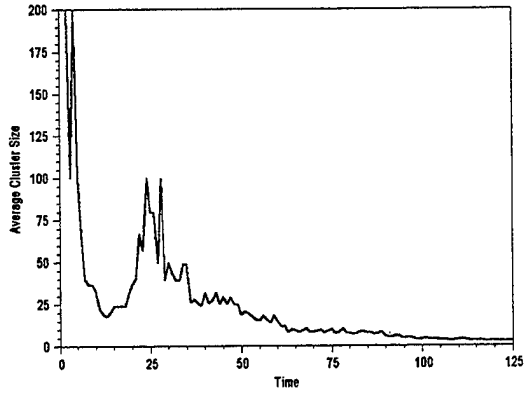
(c) Number of Red ISAACAs near Blue



(d) Blue Center-of-Mass Trajectory



(e) Spatial Entropy

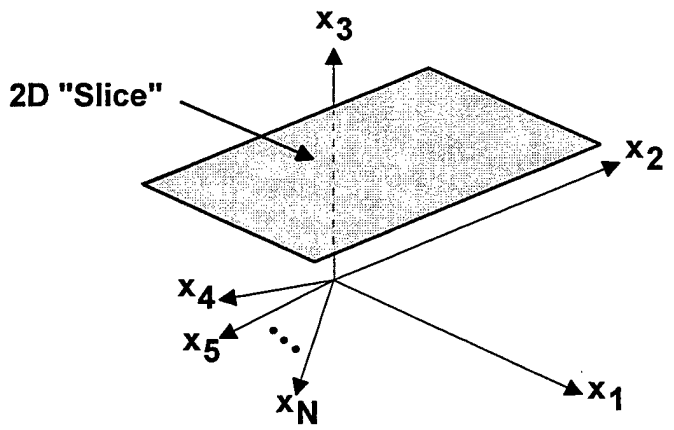


(f) Average Cluster Size

Taking 2D "Slices" of ISAAC's Parameter Space

The previous section discussed ISAAC's built-in data collection capability, which consists essentially of keeping track of various quantitative measures of behavior as a function of time. These measures include force strength, neighbor counts, ISAACA:ISAACA distance distributions, and center-of-mass position, and includes a few rudimentary pattern-recognition tools such as cluster-size distributions and estimates of spatial entropy. While this data arguably goes a long way to providing important additional insights into "what is happening" on the notional battlefield – beyond the purely *qualitative* picture that emerges from ISAACA's main graphics display alone – these data are nonetheless constrained by the same fundamental shortcoming as ISAAC's main graphics display. Namely, both provide glimpses of only a *single scenario*: a single set of parameters defining the red and blue forces, and a single spatial disposition of those forces. In order to gain insight into what ISAAC's *generic* behavior is like – i.e., in order to be able to answer questions such as *"What behavioral features are independent of initial force disposition?"* or *"How does the overall behavior change as I alter, say, blue's sensor range?"* – we must be able to (at least) (1) average over a set of initial conditions, and (b) compare objectively the behaviors resulting from differing sets of parameters.

Figure 55. Schematic of taking a two dimensional (x_1, x_2) "slice" through ISAAC's N-dimensional parameter space



To this end, a stand-alone *parameter-space mapping* program (**ISAAC_PM**; see table 4) provides a capability to effectively map out the behavior over a two-dimensional "slices" through ISAAC's N-dimensional parameter space (where N is a large number). Assume, for sake of argument, that each of the parameters that appears in ISAAC's input data file under the heading "ISAACA Parameters" represents a single independent "degree-of-freedom" (or axis) in the total ISAACA

parameter-space. Even if we were to do away with all parameters having to do with communications, all command and control functions, notional defense, fratricide and reconstitution, that still leaves us with a roughly 28-dimensional parameter space: 12 parameters for defining the individual components of the personality weight vector (6 alive + 6 injured) + 5 parameters for defining various ranges + 6 parameters for defining the 6 threshold conditions (3 alive + 3 injured) + 2 parameters for defining an ISAACA's single-shot probability and the maximum number of enemy targets that it can simultaneously target. Even granting that our assumption that each of these parameters can be treated as an independent parameter is obviously false (many parameters are interrelated and not all parameters are equally "important" in defining an ISAACA's overall behavior), an ISAACA's genome clearly "lives" in a very large dimensional space.

Now, from a conceptual stand-point, a complete "understanding" of ISAAC's overall dynamical behavior can be obtained via a complete, exhaustive sampling of all possible behaviors resulting from all possible combinations of all possible parameter values. From a practical stand-point, of course, such an ambitious research program is obviously much too computationally costly to represent a viable approach. Instead, we provide two tools for exploring the ISAACian parameter-space:

1. **Genetic Algorithm Evolutions.** The first tool consists of using a genetic algorithm to search through the available parameter space to find a set of parameters that lead to certain desired behaviors. This tool described in a later section (see ***Genetic Algorithm Evolutions of ISAACA Personalities*** below).

2. **Taking 2D "Slices" of Parameter Space.** The second tool, described immediately below, is essentially a *parameter-space mapper* that takes two-dimensional slices through the ostensibly N-dimensional parameter space, and provides measures of how well certain mission objectives are met (by one side) for a given combination of parameters. See figure 55.

As will become clear below, these two tools actually share some important features and use identical measures of "mission success" to either navigate through ISAAC's parameter space (in the case of genetic algorithms) or provide a quantitative assessment of what a particular (x,y) "point" in ISAAC's parameter space represents (in the case of parameter-space mapping).

The Basic Idea

The basic idea behind using the program **ISAAC_PM** to take 2D slices of ISAAC's parameter space is summarized as follows. First, by design, the blue force is *fixed* throughout a given run. That is to say, once the blue personality and combat parameters have all been defined – except for initial force disposition, which is always randomized at the beginning of a sample run – the blue side is "clamped," as it were, and remains unaltered throughout a run. The actual "slice" is taken through the red forces' parameter space.

To this end, the user defines red's personality and combat parameters in the usual way, except that two special parameters (of the user's choosing) – x and y – are identified to be the (x,y) coordinates over which the system's behavior will be sampled. To each (x,y) combination of variable parameters (all other red parameters remaining constant), the program associates a quantitative measure of "how well" the red ISAACAs have performed a user-defined *mission*, and averages this measure over a desired number of initial conditions (for both red and blue initial force disposition). A measure of "how well" the red force performs a given mission is provided by a well-defined mission *fitness*.

Because *missions* and *mission-fitness* are defined in the next section discussing genetic algorithms (see ***Mission Objective*** in ***Genetic Algorithm Evolutions of ISAACA Personalities***), we will only briefly introduce the two concepts here.

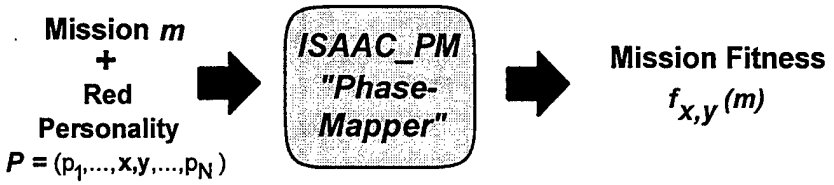
Mission

Missions – labeled m –are objective, quantifiable goals that the red force must attain within a certain time frame, and consist of one or more mission primitives. Mission primitives might include such objectives as *"get to blue flag as quickly as possible," "minimize red casualties,"* or *"maximize the ratio of blue to red casualties,"* among others.

Mission Fitness

Mission *fitness*, $f = f(m)$, is a numerical measure of how well the red force has performed its mission. It is defined so that f takes on real values between *zero* and *one*: $f(m) = 0$ meaning that the red force has been entirely *unsuccessful* in fulfilling mission m; $f(m) = 1$ meaning that the red force has been entirely *successful* in fulfilling mission m. How close the value of f is to zero or one indicates how poorly or how well, respectively, the red force has performed a given mission.

Figure 56. Schematic of how ISAAC_PM works



The output of the parameter-mapper program is thus a set of mission fitness values, $f_{x,y}(m)$, for each pair of sampled value of the (x,y) parameters defining the 2D "slice." Figure 56 shows a schematic.

Pseudo-code

ISAAC_PM couples a slightly older version of ISAAC's *Core Engine* with a basic data-collecting front-end that automatically loops through selected x and y red ISAACA parameters and averages the mission fitness over a user-specified number of red and blue initial conditions. In pseudo-code, the main components of this recipe appear as follows:

```
read PHASE.dat and P_ISAAC files
for x=xmin to xmax
    for y=ymin to ymax
        for initial_condition=1,icmax
            initialize for new run
            run ISAAC's Core Engine
            calculate_fitness(initial_condition)
        next initial_condition
        append new fitness to output file
    next y
next x
close output data file
```

Concise User's Guide to ISAAC_PM

The stand-alone parameter-space mapper program **ISAAC_PM** uses a slightly older version of ISAAC's core engine than the one that is described in detail in the section *Overview of ISAAC*. Specifically, the version of ISAAC that is embedded within **ISAAC_PM** allows only one squad per side and excludes all command and control structures. This minor deficiency will be remedied in future versions.

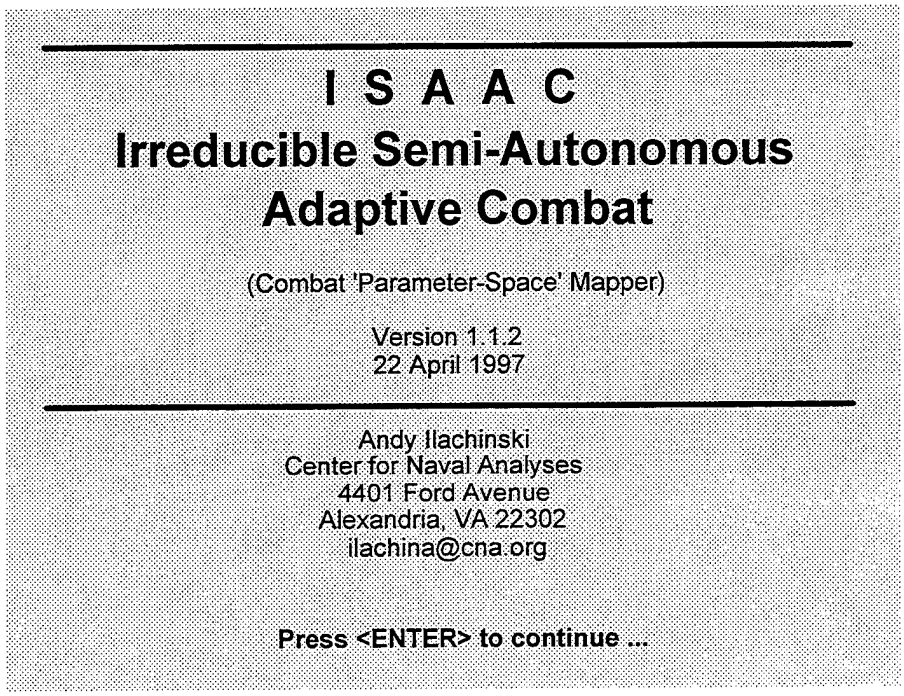
Starting ISAAC_PM

Assuming that the ISAAC "package" has been installed according to the instructions given in the section Installing ISAAC, the parameter-space mapper can be run by going to the appropriate subdirectory on the hard drive (say, **C:\ISAAC>**) and typing the command **ISAAC_PM**

followed by **<ENTER>** on the DOS command line. You will see the opening screen (figure 57), specifying the current version and build date of the program and a prompt to press **<ENTER>** to continue.

The next screen prompts for three files (see figure 58): (1) **P_ISAAC.dat**, which is the default name of the file that contains a truncated version of ISAAC's input data file (see ***Contents of ISAAC's Input File*** in ***Concise User's Guide to ISAAC***); (2) **PHASE.dat**, which is the default name of the file that contains ISAAC_PM-specific data entries needed to start the run (its contents are described below); and (3) **PHASEOUT.dat**, which is the file the user wishes to contain the output data generated by the impending run (see ***Contents of ISAAC_PM's Data Output File*** below).

Figure 57. **ISAAC_PM**'s opening screen



After naming the input and output files, the user is next prompted to select the x-coordinate of the two-dimensional "slice" that will be taken through ISAAC's parameter space (see figure 59). The user is asked to choose from a menu of twenty possible parameters.

Figure 58. **ISAAC_PM**'s file name prompt screen

```
SPECIFY INPUT FILES

ISAAC input (P_ISAAC.dat):          ?
PHASE input (PHASE.dat):            ?

SPECIFY OUTPUT FILE

PHASE output (PHASEOUT.dat):        ?
```

Figure 59. ISAAC_PM's prompt screen for specifying the x-coodinate

```
SPECIFY x-PARAMETER

<1>  Number of Red Forces
<2>  Weight w1 (Alive Red)
<3>  Weight w2 (Alive Blue)
<4>  Weight w3 (Injured Red)
<5>  Weight w4 (Injured Blue)
<6>  Weight w5 (Red Flag)
<7>  Weight w6 (Blue Flag)
<8>  Sensor Range
<9>  Fire Range
<10> Communications Range
<11> Communications Weight
<12> Threshold Range
<13> Advance Threshold
<14> Cluster Threshold
<15> Combat Threshold
<16> Min Distance from RED
<17> Min Distance from BLUE
<18> Min Distance from RED Goal
<19> Probability of Shot
<20> Max Number of Engagements

Selection (Quit = 0) ?
```

For example, suppose that the desired x-coordinate is "Number of Red Forces" (choice <1> in figure 59). After entering a "1" (followed by **<ENTER>**), the user is then prompted to specify the *minimum* value of this x-coordinate, the *maximum* value of the x-coordinate, and the *total number of desired samples* that will range from the minimal to maximal selected values during the course of the run (see figure 60). The same sequence of prompts is then repeated for the y-coordinate. Upon completing the y-coordinate related series of prompts, the program displays its main graphics screen and begins the run (see ***Sample ISAAC_PM Runs*** below).

Note that **ISAAC_PM** effectively reduces the dimensionality of ISAAC's total parameter space by setting *equal* the alive ($w_{i,alive}$) and injured ($w_{i,injured}$) versions of the same component of the personality weight vector, and doing the same for the advance, cluster, combat and min-distance thresholds (see ***ISAACA Adaptability*** in ***Overview of ISAAC***).

Figure 60. ISAAC_PM's range/sample prompt screen

```
x-coordinate: Red ISAACAs
Minimum Value =    ?
Maximum Value =    ?
# of Samples =     ?
```

Contents of ISAAC_PM's Data Input File: PHASE.dat

PHASE.dat contains a user-modifiable listing of variables that are used to control the execution of the parameter-space mapper front-end to ISAAC. It consists of three separate groups (see figure 61):

- **PHASE Parameters**, which consists of parameters specifying the number of red and blue initial spatial configurations to average over (**num_init_conds**), the maximum number of iterations allowed for any one run of a sample (**max_time_to_goal**), and a parameter that defines how rapidly the mission fitness function *f* falls off from its maximal to minimal values (**penalty_power**). For a detailed discussion of each of these entries, see ***Contents of ISAAC_GA's Data Input File: GA_ISAAC.dat***).
- **Penalty Weights**, which includes parameters that assign relative weight values to each of ten possible mission "primitives." These weights collectively define the red ISAACAs' "mission" *m* and therefore, implicitly, the mission fitness *f(m)* that **ISAAC_PM** will

"attach" to the 2D-slice coordinates (*x,y*). For details see the sections ***Mission Objective*** and ***Contents of ISAAC_GA's Data Input File: GA_ISAAC.dat*** in ***Genetic Algorithm Evolutions of ISAACA Personalities.***

- **Termination parameters,** which includes parameters specifying the exact conditions under which a given run will terminate. The four entries appearing in this group are defined in ***Contents of ISAAC_GA's Data Input File: GA_ISAAC.dat*** below).

Note that the parameters appearing in **PHASE.dat** (figure 62) are a *subset* of those appearing in the input data file used by the genetic algorithm front-end to ISAAC (**ISAAC_GA**, shown in figure 71). If the reader wishes to run the parameter-mapper program and/or alter any of **PHASE.dat**'s parameter values, he is urged to first read the section discussing the genetic algorithm front-end (***Genetic Algorithm Evolutions of ISAACA Personalities***).

Figure 61. Sample **PHASE.dat** input data file

```
************************************
*          PHASE parameters
************************************
num_initial_conds                50
max_time_to_goal                 125
penalty_power                    2
************************************
*       penalty weights (1-100)
************************************
w1_time_to_goal                  0
w2_friendly_loss                 10
w3_enemy_loss                    0
w4_red_to_blue_survival_ratio    0
w5_friendly_CM_to_enemy_flag     10
w6_enemy_CM_to_friendly_flag     0
w7_friendly_near_enemy_flag      0
w8_enemy_near_friendly_flag      0
w9_red_fratricide_hits           0
w10_blue_fratricide_hits         0
************************************
*       termination parameters
************************************
termination_code?                4
flag_containment_range           15
containment_number               10
red_CM_to_BF_frac                .5
```

Contents of ISAAC_PM's Data Output File: PHASEOUT.dat

Let *x* and *y* be the representative parameters defining the 2D "slice" of ISAAC's parameter space. The parameter *x*, for example, might be

"Number of Red ISAACAs" and y might be *"Probability of Shot;"* x and y can, of course, be set equal to any of the twenty parameters listed on **ISAAC_PM**'s prompt screens for specifying the x and y-coordinates; see figure 59.

Suppose that the user has decided to take $N_{x,\,samp}$ samples of x between the values $x_{min} = x_1$ and $x_{max} = x_{N_{x,samp}}$ (so that $x_{i+1} = x_i + \Delta$, where $\Delta = (x_{max} - x_{min}) / N_{x,\,samp}$), and to take $N_{y,\,samp}$ samples of y between the values $y_{min} = y_1$ and $y_{max} = x_{N_{y,samp}}$, so that $y_{i+1} = y_i + \Delta$, where $\Delta = (y_{max} - y_{min}) / N_{y,\,samp}$.

ISAAC_PM's output data file consists of the following three (unlabeled) columns, in the following format:

$$
\begin{array}{lll}
x_1 & y_1 & f_{x_1,y_1}(m) \\
\vdots & \vdots & \vdots \\
x_1 & y_{N_y} & f_{x_1,y_{N_y}}(m) \\
x_2 & y_1 & f_{x_2,y_1}(m) \\
\vdots & \vdots & \vdots \\
x_2 & y_{N_y} & f_{x_2,y_{N_y}}(m) \\
\vdots & \vdots & \vdots \\
x_{N_x} & y_1 & f_{x_N,y_1}(m) \\
\vdots & \vdots & \vdots \\
x_{N_x} & y_{N_y} & f_{x_N,y_{N_y}}(m)
\end{array}
$$

Sample Graphics Display

Once the user has selected the name of both input files and the parameter-mapper's output data file (see figure 58), **ISAAC_PM** runs through its initialization routine and displays the main graphics page.

A sample graphics page is shown in figure 62. Note that this figure assumes that the hot-keys **'B'** (for Battle-Space), and **'F'** (for Fitness) have both been pressed (see ***"Hot-Key" Menu*** below). The display is broken up into six separate regions:

- A **banner-display region**, located at the top of the display and containing a large bold font, which identifies the program and

release version, and a reminder of which two parameters have been selected as the *x* and *y* axes of the 2D "slice" for this run.

- A text-based **progress-report region**, located directly beneath the banner-display region, which provides an update of the progress made thus far during the run: *Sample* refers to the current sample number S being processed (expressed as a fraction of the total number of samples $S_{total} = N_x * N_y$: S/S_{total}), *Iteration* refers to the current initial condition *C* (expressed as a fraction of the total number of initial conditions that the program will average the fitness value over, C/C_{total}), Time refers to the current time step t of the sample that is being run for the C^{th} initial condition for the S^{th} sample (expressed as a fraction of the maximal allotted time for this run, t/t_{total}), and the *x*-value and *y*-value give the current values of the *x* and *y* parameters selected for this run, respectively.

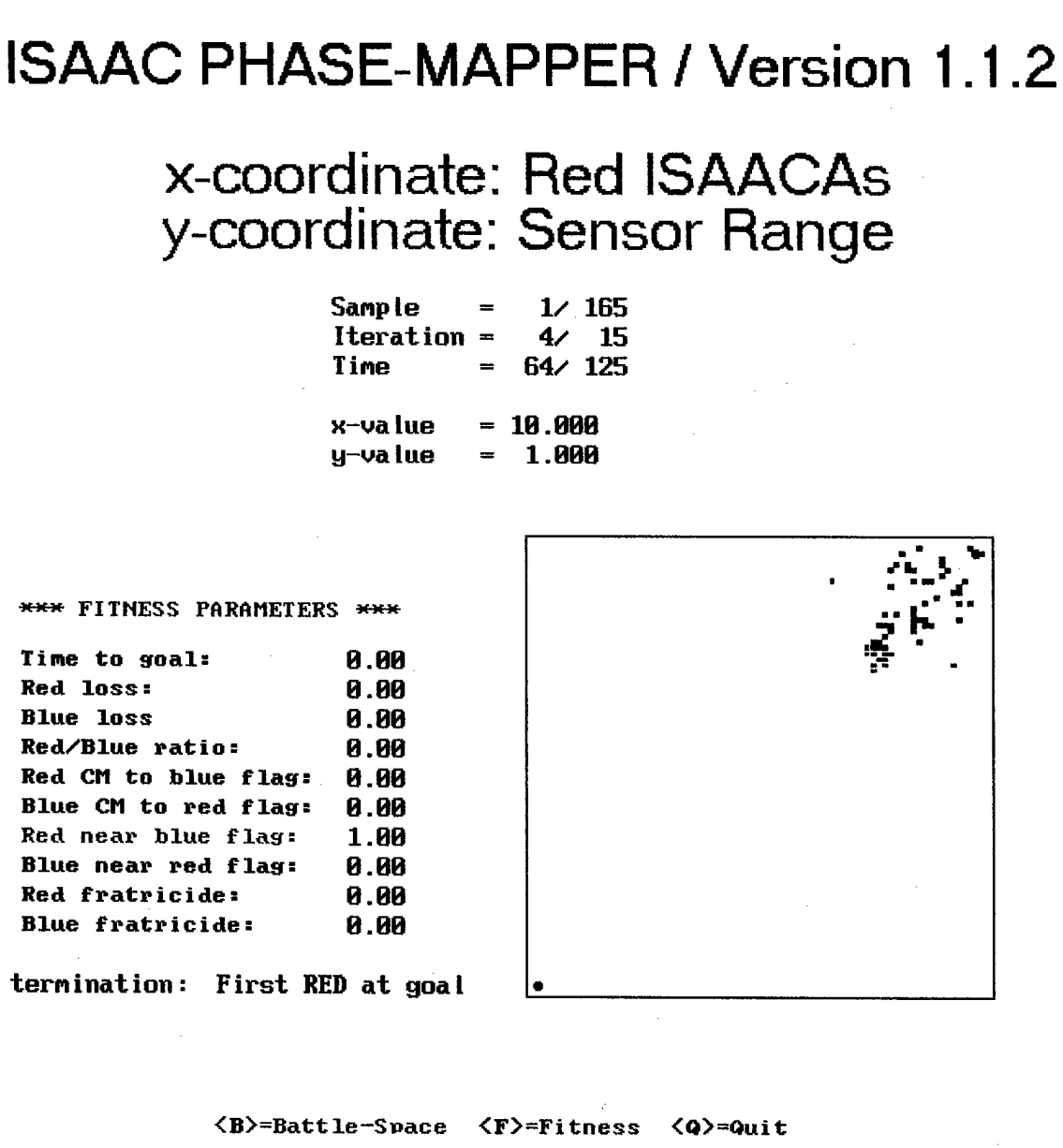
- A **battlefield region**, located near the bottom right of the display, which contains the battlefield view of state of the current sample.

- A **fitness-parameters region**, appearing to the bottom left of the battlefield, which contains a reminder of what mission fitness measure is being used for this run (see ***Mission Objective*** in ***Genetic Algorithm Evolutions of ISAACA Personalities***).

- A **"hot-key" menu region**, appearing at the bottom of the display, which contains a short menu of "hot keys" that the user can use to interrupt a run at any time to perform a variety of functions: **"B"** (for **B**attle-Space) toggles the graphics display of the notional battlefield,[23] **"F"** (for **F**itness) toggles the display of the mission objectives defining fitness for this run, along with a reminder of how each sample during this run is to be terminated, and **"Q"** (for **Q**uit) closes all output data files (saving intermediate results) and quits the program.

[23] Note that such a display is time-wise somewhat costly (slowing down apparent computation speed about 40%). Because speed is of the essence in genetic algorithm evolutions (see below), this option should be used sparingly to obtain glimpses of how a particular sample is doing.

Figure 62. ISAAC_PM's main graphics display



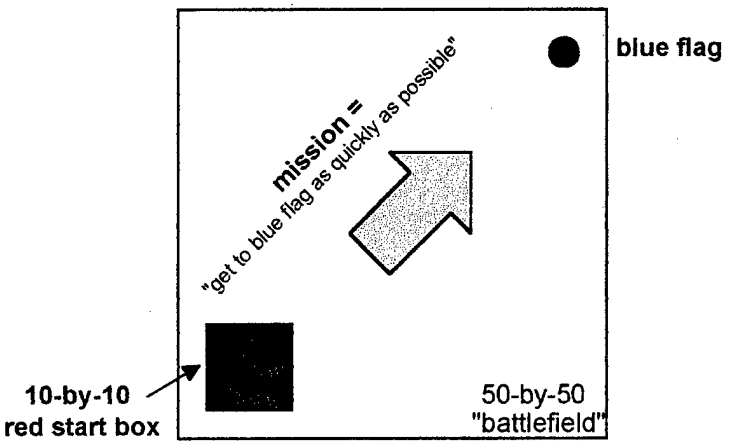
Sample Output

Sample #1: Forward Advance

One of the first emergent-like characteristics one notices about evolutions in ISAAC is that a cluster of mutually attracting ISAACAs – for example, a cluster composed of alive red ISAACAs, all of whom have a positive weight for moving toward alive red ISAACAs – typically does not move as quickly (toward the enemy goal, say) as the individual ISAACAs that make up that cluster would move toward the goal by themselves. In other words, a cluster attains an effective *cluster-velocity* $v_{cluster}$ that is in general not equal to the velocity v_{ISAACA} that the individual ISAACAs making up that cluster would have if they were not part of the cluster.

To see this using **ISAAC_PM**, consider a scenario that unfolds on a 50-by-50 notional battlefield (i.e., set **battle_size** = 50 in **P_ISAAC.dat**; see above) in which there are no blue ISAACAs at all (i.e., set **num_blues** = 0 in **P_ISAAC.dat**), red starts out confined to a 10-by-10 "box" near the lower-left corner of the battlefield, and red's mission is simply to "get to the blue flag as quickly as possible" (i.e., set **w1_time_to_goal** = 1 and all other weights **w2 - w10** to *zero* in **PHASE.dat**). In order to focus the scenario entirely on examining the effects of the cluster threshold, set the advance threshold (i.e., **ADVANCE_num** in **P_ISAAC.dat**) to zero, so that all red ISAACAs are able to move toward the blue flag even when they are locally "alone." Also, let the run terminate either when the first red ISAACA reaches the blue flag or when a maximum time t = 125 has been reached (i.e., set **termination_code** = 1 in **PHASE.dat**). See figure 63.

Figure 63. Schematic of initial state for sample run #1

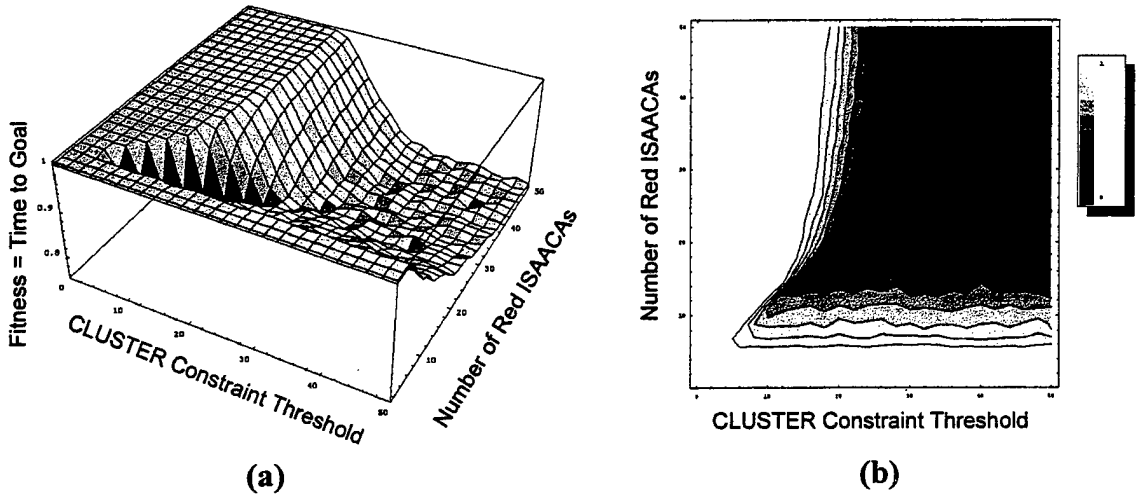


Since the red ISAACAs start in a "box" whose center is about 55 lattice sites away from the blue flag (positioned at coordinates (x,y) = (49,49)), the minimal number of iteration steps that it must take red to reach the

flag is also about 55 steps. The time may be somewhat smaller or larger than this value, depending on exactly where in the box the red ISAACA that reaches the blue flag first actually started its run from the start box. The mission fitness f for this scenario is equal to *one* if the red force reaches the blue flag as quickly as possible (that is, in a time that it would take red if red's red:red weights w_1 and w_3 were equal to zero). The fitness $f = 0$ if no red ISAACA reaches the blue flag before t = 125.

This first output sample, shown in figure 64, consists of showing the result of running **ISAAC_PM** using the *number of red ISAACAs* as the x-coordinate of the 2D "slice" (with the value of x ranging from 1 to 50) and the "*cluster constraint threshold*" as the y-coordinate (with the value of y ranging from 0 to 50). Figure 64-a shows a three-dimensional plot of the fitness $f(x,y)$ as a function of (x,y). Figure 64-b shows a contour plot of the same function, in which lighter shades of gray represent high fitness (near $f \sim 1$) and darker shades represent low fitness (near $f \sim 0$).

Figure 64. Output of **ISAAC_PM** for sample run #1



Notice that when the value of the cluster constraint, C, is very small (less than 5), the red force requires only the minimal time to get to the blue flag (i.e., $f \sim 1$), regardless of the number of red ISAACAs. This is because the ISAACAs effectively must spend little or no "time" on any "internal jostling" for position, and are able to focus entirely on moving toward the blue flag. However, higher values of C generally force the red ISAACAs to spend more time on this internal jostling. The result is that for scenarios in which there are more than five reds present, there is a critical dynamical threshold of the cluster threshold – call it C_c – such that for values of the constraint threshold $C < C_c$, $f \sim 1$ and for $C > C_c$, f is less than 1 and decreases with increases C. In other words, above a certain value of the constraint threshold, and depending on the

number of red ISAACAs, the red force becomes increasingly less adept at making it to the blue flag with increasing *C*.

Sample #2: Red Offense

As a second example of how **ISAAC_PM** can be used to explore a 2D slice of ISAAC's parameter space, consider a scenario in which red's *offensive* capability is tested. This time both sides start out with 50 ISAACAs each, and red's mission objective is to get *as many red ISAACAs within a distance D=12 of the blue flag as possible.* Blue defends with a personality defined by $\vec{w}_{\text{blue}} = (0, 10, 0, 10, 0, 0)$; i.e., blue "sees" only the enemy and does not distinguish between alive and injured reds. Blue's combat threshold is equal to *negative* 3 so that blue is fairly aggressive, and blue's sensor and fire ranges are $r_S = 4$ and $r_F = 3$, respectively. Red "attacks" with a more well-rounded personality, defined by the weight vector $\vec{w}_{\text{red}} = (10, 40, 10, 40, 0, 50)$. Red's sensor and fire ranges are equal to the blue forces'. Combat ensues for a maximum 125 iteration steps on a size 50-by-50 notional battlefield.

Figures 65-a and 65-b show the output of ISAAC_PM for this scenario, using *red's single-shot probability,* P_{red} as the *x*-coordinate of the 2D "slice" (with the value of *x* ranging from 0.001 to 0.015; note that $P_{blue} = 0.003$) and the "*combat threshold*" as the *y*-coordinate (with the value of *y* ranging from -25 to 25). Figure 65-a shows a three-dimensional plot of the fitness $f(x,y)$ as a function of (x,y). Figure 65-b shows a contour plot of the same function, in which lighter shades of gray represent high fitness (near $f \sim 1$) and darker shades represent low fitness (near $f \sim 0$).

Figure 65. Output of **ISAAC_PM** for sample run #2

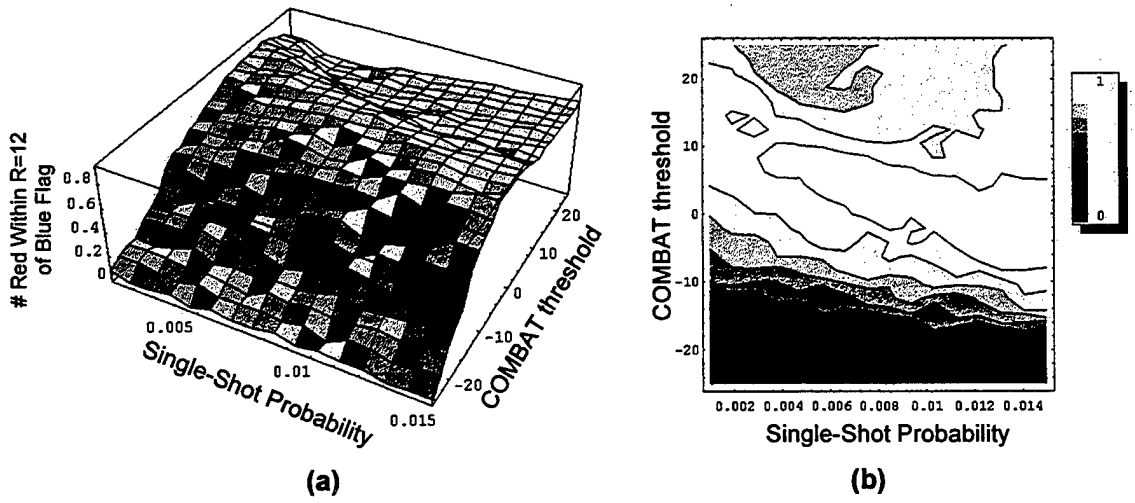


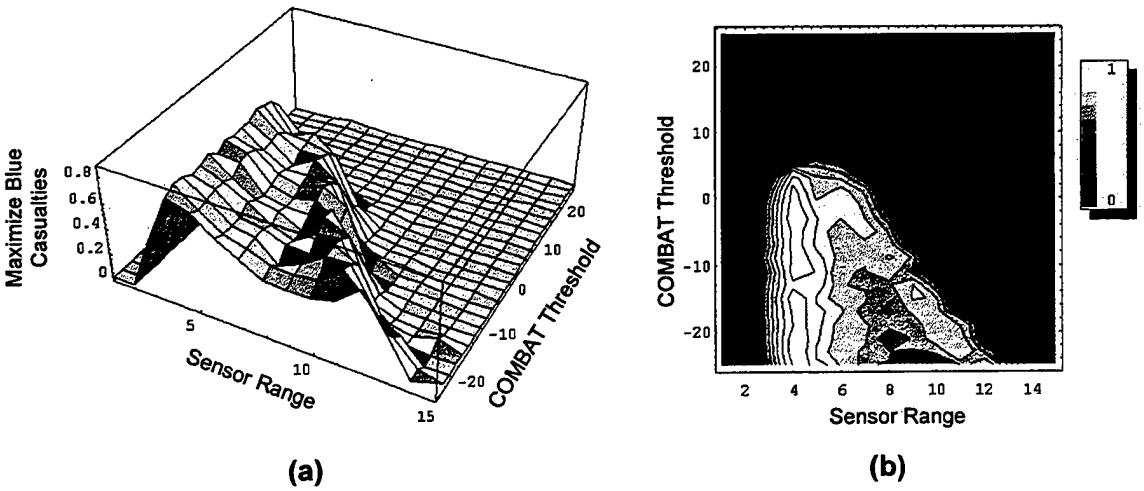
Figure 65 shows an interesting non-monotonic behavior. In words, for single-shot probabilities $P_{red} \gtrsim 0.00$, red performs this particular

mission "best" – as defined by the lighter colored regions in figure 65-b – by being neither too aggressive (with large negative values of combat threshold) nor too timid (with large positive values of combat threshold).

Sample #3: Red Defense

As a third, and final, example of how **ISAAC_PM** can be used to explore a 2D slice of ISAAC's parameter space, consider a scenario in which red's *defensive* capability is tested. As in the previous example, both sides start out with 50 ISAACAs each, but this time red's mission objective is to *maximize blue casualties near the red flag.* Red's fitness $f \sim 1$ only when red is able to kill all (or most) of the blue ISAACAs that approach the red flag. Red defends its flag with a personality defined by weight vector $\vec{w}_{red} = (10, 40, 10, 40, 0, 0)$, meaning that red ISAACAs "react" to both friendly and enemy ISAACAs, but that they do not "see" either flag. They are initially positioned close to the red flag, as in both previous sample runs (see figure 63). Blue attacks with a personality defined by $\vec{w}_{blue} = (10, 40, 10, 40, 0, 50)$, which is the same as red's except that blue also "sees" the red flag. Blue's sensor and fire ranges are $r_S = 4$ and $r_F = 3$, respectively, its combat threshold is equal to *negative* 3 (so that it is fairly aggressive), and its single-shot probability $P_{blue} = 0.005$. Combat ensues for a maximum 125 iteration steps on a size 50-by-50 notional battlefield.

Figure 66. Output of **ISAAC_PM** for sample run #3



Figures 66-a and 66-b show the output of **ISAAC_PM** for this scenario. Figure 66 uses *red's sensor range* as the *x*-coordinate of the 2D "slice" (with the value of x ranging from 1 to 15) and the "*combat threshold*" as the *y*-coordinate (with the value of y ranging from -25 to 25). Figure 66-a shows a three-dimensional plot of the fitness $f(x,y)$ as a function of

(x,y). Figure 66-b shows a contour plot of the same function, in which lighter shades of gray represent high fitness (near $f \sim 1$) and darker shades represent low fitness (near $f \sim 0$).

Figure 66 shows that the red force does not perform its mission very well anywhere within this particular 2D "slice" of the total parameter space, *except* within a relatively small, but well-defined, region (i.e., the "hump" in figure 66-a and the lighter-colored region in figure 66-b).

Genetic Algorithm Evolutions of ISAACA Personalities

As mentioned in the introduction to this paper, ISAAC has been designed with a view towards eventually encompassing three separate but mutually overlapping classes of run modes:

- ***Fixed Rules***, in which there is no "learning" and ISAAC is run using a fixed set of local rules applied at each time step. This mode is well suited for quickly playing simple *"What if?"* scenarios, and can be used to "search" for interesting emergent behavior.
- ***Fixed Strategies***, in which ISAAC is run using a fixed set of personalities (as in the *fixed rule* mode) but using rules that have been "evolved" off-line specifically to perform a given mission. Unlike the fixed rule mode, in which ISAAC is used to explore how an arbitrary set of parameter values unfolds in time, the fixed strategy mode involves a focused automated search for the personality "best" suited for performing some well-defined mission.
- ***Adaptive Learning***, in which ISAACAs use real-time heuristic adaptive learning strategies to "discover" new rules as scenarios unfold in time.

The ISAAC that has been described thus far has been firmly, and exclusively, rooted in the first – *fixed rule* – mode. The general "template" for using ISAAC has heretofore consisted of (1) choosing red and blue personalities (sensor and fire ranges, personality weight vectors and/or any additional movement constraint rules), (2) defining the initial spatial disposition of forces for the two sides, possibly throwing in some notional terrain to complicate the battlefield, and (3) running ISAAC to see what pattern of behavior unfolds.

From the point of view of trying to answer the basic question motivating this study – *"Can land combat be described as a complex adaptive system?"* (see ***Introduction***) – such a simple-minded "anecdotal" approach is well suited. Even the few illustrative runs shown in the ***Sample Runs*** section provide strong evidence that many "high-level" behaviors such as penetration, clustering, encirclement, defensive posturing, and so on, can actually be thought of as naturally emergent behavioral patterns stemming from a local, nonlinear and decentralized collective dynamics. The short answer to the above question is, therefore, a very defensible *"yes."* However, from a larger perspective, wherein one's

interest lies more in using a multiagent-based simulation of combat to develop an analyst's toolbox for identifying, exploring, and possibly exploiting emergent patterns of behavior, such purely anecdotal evidence cannot suffice. The real question is, *"Now that it has been demonstrated that land combat can be described as a complex system, so what? What do we do with this insight?"* This section describes the first baby-step that ISAAC can be taught to make toward addressing this real question: to use a Genetic Algorithm (GA) to evolve the "best" personality for performing a specific mission; i.e., to evolve tactics from the "ground up." This paper thus next describes ISAAC's first foray into a *fixed strategy* run mode.

Genetic Algorithms (GAs): Brief Overview

GAs are a class of heuristic search methods and computational models of adaptation and evolution based on natural selection. An overview of GAs, along with some sample problems, is provided in appendix B.

In nature, the search for beneficial adaptations to a continually changing environment (i.e., *evolution*) is fostered by the cumulative evolutionary knowledge that each species possesses of its forebears. This knowledge, which is encoded in the chromosomes of each member of a species, is passed from one generation to the next by a mating process in which the chromosomes of "parents" produce "offspring" chromosomes.

GAs mimic and exploit the genetic dynamics underlying natural evolution to search for optimal solutions of general combinatorial optimization problems. They have been applied to the Traveling Salesman Problem, VLSI circuit layout, gas pipeline control, the parametric design of aircraft, neural net architecture, models of international security, and strategy formulation.

While their modern form is derived mainly from John Holland's work in the 1960s [24], many key ideas such as using "selection of the fittest" like population-based selection schemes and using binary strings as computational analogs of biological chromosomes, actually date back to the late 1950s. More recent work is discussed by Goldberg [25], Davis [26] and Michalewicz [27] and in conference proceedings edited by Forrest [28]. A comprehensive review of the current state-of-the-art in genetic algorithms is given by Mitchell [29].

The basic idea behind GAs is very simple. Given a "problem" – which can be as well-defined as maximizing a function over some specified interval or as seemingly ill-defined and open-ended as evolution itself, where there is no a-priori discernible or fixed function to either maximize or minimize – GAs provide a mechanism by which the solution space to that problem is searched for "good solutions." Possible

solutions are encoded as *chromosomes* (or, sometimes, as sets of chromosomes), and the GA evolves one population of chromosomes into another according to their *fitness* by using some combination (and/ or variation) of the genetic operators of *reproduction, crossover* and *mutation.*

Each chromosome is usually defined to be a bit-string, where each bit position (or "locus") takes on one of two possible values (or "alleles"), and can be imagined as representing a single point in the "solution space." The fitness of a chromosome effectively measures how "good" a solution that chromosome represents to the given problem. Aside from its intentional biological roots and flavoring, GAs can be thought of as parallel equivalents of more conventional serial optimization techniques: rather than testing one possible solution after another, or moving from point to point in the solution phase-space, GAs move from entire populations of points to new populations.

Reproduction makes a set of identical copies of a given chromosome, where the number of copies depends on the chromosome's fitness. The *crossover* operator exchanges subparts of two chromosomes, where the position of the crossover is randomly selected, and is thus a crude facsimile of biological sexual recombination between two single-chromosome organisms. The *mutation* operator randomly flips one or more bits in the chromosome, where the bit positions are randomly chosen. The mutation rate is usually chosen to be small.

While reproduction generally rewards high fitness, and crossover generates new chromosomes whose parts, at least, come from chromosomes with relatively high fitness (this does not guarantee, of course, that the crossover-formed chromosomes will also have high fitness; see below), mutation seems necessary to prevent the loss of diversity at a given bit-position. For example, were it not for mutation, a population might evolve to a state where the first bit-position of each chromosome contains the value 1, with there being no chance of reproduction or crossover ever replacing it with a 0.

A solution search space together with a fitness function is called a *fitness landscape.* Eventually, after many generations, the population will, in theory, be composed only of those chromosomes whose fitness values are clustered around the global maximum of the fitness landscape.

The Basic GA Recipe

Although GAs, like cellular automata, come in many different flavors, and are usually fine-tuned in some way to reflect the nuances of a particular problem, they are all more or less variations of the following basic steps:

- *Step 1:* begin with a randomly generated population of chromosome-encoded "solutions" to a given problem
- *Step 2:* calculate the fitness of each chromosome, where fitness is a measure of how well a member of the population performs at solving the problem
- *Step 3:* retain only the fittest members and discard the least fit members
- *Step 4:* generate a new population of chromosomes from the remaining members of the old population by applying the operations *reproduction*, *crossover*, and *mutation* (see figure 91 in appendix B)
- *Step 5:* calculate the fitness of these new members of the population, retain the fittest, discard the least fit, and re-iterate the process

This basic five step algorithm will be adapted to simple "mission-specific" ISAAC scenarios in the next section.

Genetic Algorithms : Adapted to ISAAC

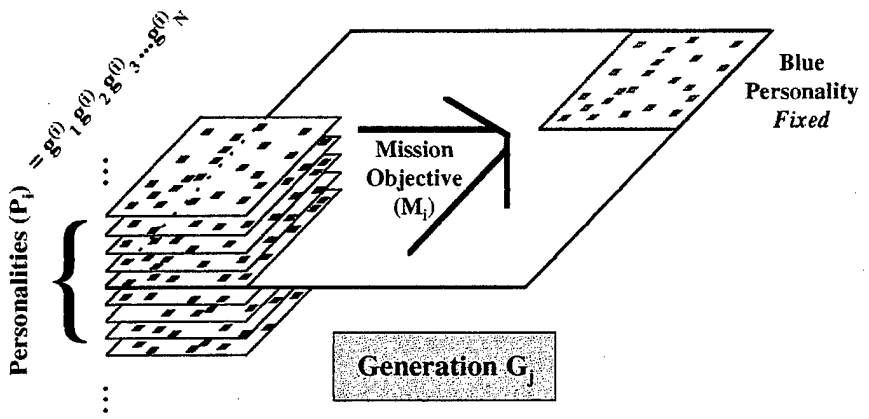
Figure 67 shows a schematic of the general kind of "GA problem" in ISAAC. In simplest terms (that will be made more precise shortly), the problem is this:

Given a fixed BLUE force (except for blue's initial spatial distribution on the battlefield), and a well-defined mission objective for RED, the GA's task is to evolve a personality for the RED force that is "best able" to satisfy the objective.

The phrase "best able" here means a red force that performs best according to some well-defined measure of mission fitness (see below).

To "solve" this problem, a pool of initially randomized red ISAACA personalities (shown at the bottom left of figure 67) is evolved in time according to the basic GA recipe defined above. Each personality is defined by a unique ISAACA chromosome, consisting of N genes (see below). The "fitness" of each chromosome (and, hence, each personality) is determined by how well that personality has performed a certain user-defined mission (see ***Mission Objective*** below).

Figure 67. Schematic of the GA "problem" in ISAAC



Personality Chromosome

The personality, $P_i(t)$, of the i^{th} red ISAACA in the personality pool at time t, is defined by a unique ISAACA chromosome, $C_i(t)$, defined by

$$C_i(t) = g_1^{(i)} g_2^{(i)} \cdots g_N^{(i)} ,$$

where $g_j^{(i)}$ is the j^{th} gene. In the current version of **ISAAC_GA**, there are a total of N=45 genes, though not all are necessarily used.

Table 9 provides a description of each of the 45 genes. Note that, unlike some common textbook GA examples (such as the illustrative example discussed in Appendix B), the chromosome is *not* a binary-valued string that consists only of 0's and 1's. Instead, each gene is *real-valued*, and any appropriate translations to integer values and or binary toggles (on/off) are performed automatically by the program.

For the most part, each gene encodes the value of a basic parameter defining the red force. For example, g_1 encodes red's sensor range, g_2 encodes red's fire range, and so on. Some genes – for example, the *odd* numbered genes between g_5 and g_{27} – encode the *sign* (+ or -) of the immediately preceding gene, but not the actual value. Thus, the actual value of each of the components of red's personality weight vector, for example, is actually encoded by two genes; one gene specifying the component's absolute value, the other gene defining its sign. Note also that the lighter colored genes (genes g_1 through g_{35}) are *always* used; that is, they are always a part of the genotype specifying the red force. The more darkly colored genes (genes g_{36} through g_{42} and genes g_{43}, g_{44} and g_{45}) are used only if certain software flags are set in the data input file for ISAAC_GA (see ***ISAAC_GA's User's Guide*** for details below).

Table 9. ISAACA Chromosome (see text)

Gene	Function
1	sensor range
2	fire range
3	threshold range
4	alive red:alive red weight (w1)
5	sign (+ or -) of w1
6	alive red:alive blue weight (w2)
7	sign (+ or -) of w2
8	alive red:injured red weight (w3)
9	sign (+ or -) of w3
10	alive red:injured blue weight (w4)
11	sign (+ or -) of w4
12	alive red:red flag (w5)
13	sign (+ or -) of w5
14	alive red:blue flag (w6)
15	sign (+ or -) of w6
16	injured red:alive red weight (w1')
17	sign (+ or -) of w1'
18	injured red:alive blue weight (w2')
19	sign (+ or -) of w2'
20	injured red:injured red weight (w3')
21	sign (+ or -) of w3'
22	injured red:injured blue weight (w4')
23	sign (+ or -) of w4'
24	injured red:red flag (w5')
25	sign (+ or -) of w5'
26	injured red:blue flag (w6')
27	sign (+ or -) of w6
28	alive ADVANCE threshold
29	alive CLUSTER threshold
30	alive COMBAT threshold
31	sign (+ or -) of gene 30
32	injured ADVANCE threshold
33	injured CLUSTER threshold
34	injured COMBAT threshold
35	sign (+ or -) of gene 34
36	min_dist_flag
37	alive red:red min distance
38	alive red:blue min distance
39	alive red:red flag min distance
40	injured red:red min distance
41	injured red:blue min distance
42	injured red:red flag min distance
43	size of initialization box
44	x-coordinate of initialization box
45	y-coordinate of initialization box

Mission Objective

The mission objective, or fitness, is a measure of how well red ISAACAs have performed a user-defined mission. Typical missions might be to *"get to blue flag as quickly as possible," "minimize red casualties," "maximize the ratio of blue to red casualties,"* and so on, or some combination of these.

More specifically, the user – who, for discussion purposes, can be thought of as a Supreme Commander (SC) – can assign up to ten weights – $0 \le w_1, w_2, \ldots, w_{10} \le 1$ – to represent the relative degree of importance afforded to a particular mission objective "primitive," m_i (see table 10 and discussion below). (At this introductory level, of course, the list of mission primitives is still fairly short and simple, though it is flexible enough to enable the user to define many non-trivial objectives.) The actual mission objective, or fitness function, M, is a weighted sum of mission primitives:

$$M = w_1 m_1 + w_2 m_3 + \cdots + w_{10} m_{10} .$$

Table 10. A description of GA weights

GA Weight	Mission Objective "Primitive"	Description
w_1	m_1	minimize time to goal
w_2	m_2	minimize red (i.e. friendly) casualties
w_3	m_3	maximize blue (i.e. enemy)casualties
w_4	m_4	maximize red-to-blue (i.e. friendly-to-enemy) survival ratio
w_5	m_5	minimize red (i.e. friendly) center-of-mass distance to blue (i.e. enemy) flag
w_6	m_6	maximize blue (i.e. enemy) center-of-mass distance to red (i.e. friendly) flag
w_7	m_7	maximize number of red (i.e. friendly) within an SC-defined distance of the blue (i.e. enemy) flag
w_8	m_8	minimize number of blue (i.e. enemy) within an SC-defined distance of the red (i.e. friendly) flag
w_9	m_9	minimize number of red fratricide "hits" (i.e. on friendly side)
w_{10}	m_{10}	maximize number of blue fratricide "hits" (i.e. on enemy side)

For example, a simple mission objective might be to *"get to the blue flag as quickly as possible,"* in which case $w_1 = 1$ and $M = m_1$. If, in addition, the

SC wishes to *"minimize red losses"* (defined by primitive m_2), but still cares more about minimizing the time it takes red to get to the blue flag than about casualties, the SC might set w_1 equal to 3/4 and $w_2 = 1\text{-}w_1 = 1/4$. The total mission fitness in this case is then given by $M = (3/4)\, m_1 + (1/4)\, m_2$. A more complicated mission objective might be to simultaneously satisfy several mission primitives:

- *get as many red ISAACAs within a certain range of the blue flag as possible* (defined by m_7)
- *keep blue forces as far from red flag as possible* (defined by m_6)
- *minimize red casualties* (defined by m_2)
- *maximize red to blue losses* (defined by m_4)
- *minimize red fratricide* (defined by m_9)

so that, if each of these primitives is afforded equal weight, the composite mission objective is in this case given by $M = (1/5)(m_2+m_4+m_6+m_7+m_9)$. Of course, not all such composite missions may make sense, or lead to red force personalities that are able to perform them to a desired (i.e., SC-prescribed) fitness level. It is the SC's task to ensure that mission objectives are both logically sound and amenable to "solution" (see ***GA Sample Runs***).

Before providing a bit more detail about each of these ten mission primitives, we first make a general technical comment concerning what "function" is really being maximized. The user may have noticed that half of the GA weights refer to mission primitives that involve functions whose values must be *minimized* (m_1, m_2, m_5, m_8 and m_9) and half refer to primitives that involve functions whose values must be *maximized* (m_3, m_4, m_6, m_7 and m_{10}). In fact, <u>*all*</u> mission primitives are represented within the program by a function that takes values between zero (corresponding to zero fitness) and one (corresponding to maximum fitness) and that the GA attempts to *maximize*. The result is that while it may be more intuitively natural to refer to some mission primitives (such as m_1 = "minimize time to goal") in terms of a quantity that must be minimized, in fact, all primitives are actually defined inside of the program in such a way that the GA consistently tries to maximize their fitness.

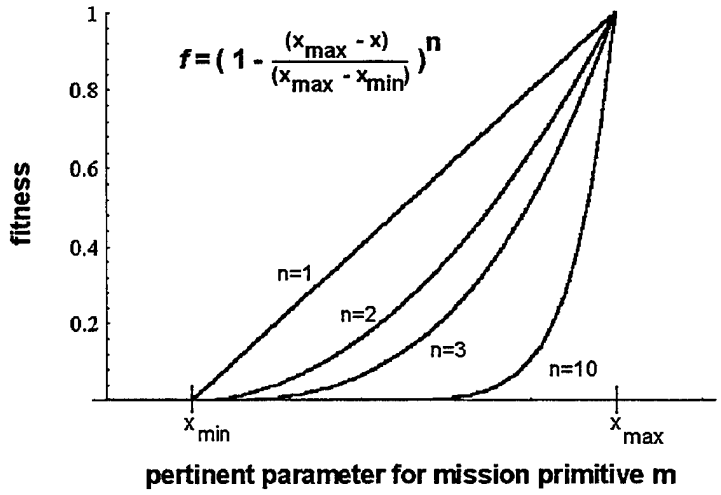
The general template for how each primitive is treated internally by the program is as follows. First, for each primitive m_i, the minimal ($=x_{min}$) and maximal ($=x_{max}$) possible values for the pertinent parameter x is identified. For example, for m_1 = "minimize time to goal" the pertinent variable is x = "time to goal;" for m_2 = "minimize red casualties" the pertinent variable is x = "number of red casualties," and so on. Next, a simple function $f = f(x)$ is defined that takes on values between zero and

one so that $f(x_{min}) = 0$ corresponds to "minimal fitness" and $f(x_{max}) = 1$ corresponds to "maximal fitness:"

$$f = \left(1 - \frac{(x_{max} - x)}{(x_{max} - x_{min})}\right)^n,$$

where the (the user-defined) power n (see **penalty_power** in ***Contents of ISAAC_GA's Input Data File***) determines how rapidly f falls off from its maximal to minimal fitness. In general, the closer the value of f is to the value 1, the "better" the red ISAACAs are said to have performed the particular mission primitive that f is the fitness function for. See figure 68.

Figure 68. Schematic for fitness function *f* (corresponding to mission primitive *m)* that is internally maximized by the program



The next few sections provide a self-contained reference for each of the ten mission primitives and the corresponding functions that the GA is asked to maximize.

Mission primitive m_1

The first mission primitive consists of minimizing the time to goal. The user can define an auxiliary condition triggering the *red-at-goal* flag by requiring that a certain number of red ISAACAs must be within a range R of the blue flag (see **termination_code** in ***Contents of ISAAC_GA's Input Data File***).

The pertinent parameter for m_1 is t_G = "time to goal." The minimal possible value of t_G, $t_{G,min}$, is determined by computing how much time it would take an ISAACA that is closest to the blue flag to get to the flag

if there were no other ISAACAs on the battlefield. The maximum value, $t_{G,max}$, is set by the user.

The fitness function for m_1 is given by $f_1 = ((t_{G,max} - t_G)/(t_{G,max} - t_{G,min}))^n$. Notice that if the red ISAACAs reach the blue flag only at the maximum allotted time, their mission fitness is *zero*. Conversely, if they reach the flag in the minimal possible time, their fitness is *one*.

Mission primitive m_2

The second mission primitive consists of minimizing the total number of red casualties.

The pertinent parameter for m_2 is R_t = "number of red ISAACAs at time t." Note that no distinction is made between alive or injured ISAACAs. The minimal possible value of R_t, R_{min}, is equal to zero, while the maximum possible value is $R_{max} = R_0$, or the total number of red ISAACAs at time t = 0.

The fitness function for m_2 is given by $f_2 = (R_T / R_0)^n$, where T is the termination time for the run (which can vary depending on the selected termination condition; see **termination_code** in ***Contents of ISAAC_GA's Input Data File***). Notice that if red suffers no losses at all, so that $R_T = R_0$, then $f_2 = 1$. Conversely, if all red ISAACAs are lost, then their mission fitness for this primitive is *zero*.

Mission primitive m_3

The third mission primitive consists of maximizing the total number of blue casualties.

The pertinent parameter for m_2 is B_t = "number of blue ISAACAs at time t." Note that no distinction is made between alive or injured ISAACAs. The minimal possible value of B_t, , B_{min}, is equal to zero, while the maximum possible value is $B_{max} = B_0$, or the total number of blue ISAACAs at time t = 0.

The fitness function for m_3 is given by $f_3 = (1 - B_T / B_0)^n$, where T is the termination time for the run (which can vary depending on the selected termination condition; see **termination_code** in ***Contents of ISAAC_GA's Input Data File***). Notice that if blue suffers no losses at all, so that $B_T = B_0$, then, from red's point of view, the mission fitness is minimal, and $f_3 = 0$. Conversely, if the red ISAACAs have successfully killed all blue ISAACAs – so that $B_T = 0$ – then their mission fitness is maximal, and $f_3 = 1$.

Mission primitive m_4

The fourth mission primitive consists of maximizing the ratio between red and blue casualties.

There are two pertinent parameters for m_4: R_t = "number of red ISAACAs at time t" and B_t = "number of blue ISAACAs at time t." Note that no distinction is made between alive or injured ISAACAs. The minimal possible values of R_t and B_t, are equal to zero, while the maximum possible values are $R_{max} = B_0$ and $B_{max} = B_0$, or the total number of red and blue ISAACAs at time t = 0.

The fitness function for m_4 is given by

$$f_4 = f_4(R_T, B_T) = \left(\frac{R_T/R_0}{B_T/B_0}\right)^n \left[\left(\frac{B_0-2}{B_0-1}\right)\left(\frac{R_T/R_0}{B_T/B_0}\right) + \frac{B_0}{B_0-1}\right]^{-n},$$

where T is the termination time for the run (which can vary depending on the selected termination condition; see **termination_code** in ***Contents of ISAAC_GA's Input Data File***). This fitness function is defined to give intuitively reasonable values for some extremal values. For example, $f_4(R_T=0, B_T) = 0$ for any value of B_T, reflecting the intuition that if the entire red side is killed, the mission fitness is minimal, regardless of the number of remaining blue forces. If the fraction of remaining forces is equal on both sides, so that, say, $R_T/R_0 = B_T/B_0 = \rho$, then $f_4(\rho R_0, \rho B_0) = 1/2$, reflecting the intuition that if red only keeps pace with blue casualties (but that, perhaps, both sides have suffered some casualties), the mission fitness lies somewhere halfway between its minimal and maximal values. Finally, if the number of red ISAACAs at time t = T is equal to the initial number of red ISAACAs while the number of blue ISAACAs has been reduced to one,[24] the mission fitness approaches its maximal possible value: $f_4(R_T = R_0, B_T = 1) \rightarrow 1$. Intermediate values of R_T and B_T yield values for f_4 between zero and one.

Mission primitive m_5

The fifth mission primitive consists of minimizing the average red center-of-mass distance to the blue flag.

The pertinent parameter for m_5 is d = "distance between the center-of-mass of the red force and the position of the blue flag at time t." The $x_{red,CM}$ and $y_{red,CM}$ coordinates of the red center-of-mass are defined by

$$x_{\mathrm{red,CM}}(t) = \frac{1}{R_t}\sum_{i=1}^{R_t} x_{\mathrm{red},i}(t) \quad \text{and} \quad y_{\mathrm{red,CM}}(t) = \frac{1}{R_t}\sum_{i=1}^{R_t} y_{\mathrm{red},i}(t),$$

where $x_{red,i}(t)$ and $y_{red,i}(t)$ are the x and y positions of the i^{th} red ISAACA at time t, respectively, and R_t is the number of red ISAACAs at time t.

[24] The value "1" is the minimal allowable value for B_0 in calculating f_4. In the event that $B_0=0$, B_0 is automatically set equal to 1.

The fitness function for m_5 is given by $f_5 = ((d_{max} - d_{ave})/d_{max})^n$, where $d_{max} = \sqrt{2}$ * **battle_size** (see ***General Battle Parameters*** in ***Contents of ISAAC Data Input File***) and d_{ave} is the average red center-of-mass distance to the blue flag:

$$d_{\text{ave}} = \frac{1}{T}\sum_{t=1}^{T}\left[\left(x_{\text{blue-flag}} - x_{\text{red},CM}(t)\right)^2 + \left(y_{\text{blue-flag}} - y_{\text{red},CM}(t)\right)^2\right],$$

where T is the termination time for the run (which can vary depending on the selected termination condition; see **termination_code** in ***Contents of ISAAC_GA's Input Data File***). Notice that if the red force is close to the blue flag at all times (so that $d_{ave} \sim 0$), then $f_5 \sim 1$. Conversely, if the red ISAACAs spend most of their time far from the blue goal (so that $d_{ave} \sim d_{max}$), then $f_5 \sim 0$.

Mission primitive m_6

The sixth mission primitive consists of maximizing the average blue center-of-mass distance to the red flag.

The pertinent parameter for m_6 is d = "distance between the center-of-mass of the blue force and the position of the red flag at time t." The $x_{blue,CM}$ and $y_{blue,CM}$ coordinates of the blue center-of-mass are defined by

$$x_{\text{blue,CM}}(t) = \frac{1}{B_t}\sum_{i=1}^{B_t} x_{\text{blue},i}(t) \quad \text{and} \quad y_{\text{blue,CM}}(t) = \frac{1}{B_t}\sum_{i=1}^{B_t} y_{\text{blue},i}(t),$$

where $x_{blue,i}(t)$ and $y_{blue,i}(t)$ are the x and y positions of the i^{th} blue ISAACA at time t, respectively, and B_t is the number of blue ISAACAs at time t.

The fitness function for m_6 is given by $f_6 = (d_{ave}/d_{max})^n$, where $d_{max} = \sqrt{2}$ * **battle_size** (see ***General Battle Parameters*** in ***Contents of ISAAC Data Input File***) and d_{ave} is the average blue center-of-mass distance to the red flag:

$$d_{\text{ave}} = \frac{1}{T}\sum_{t=1}^{T}\left[\left(x_{\text{red-flag}} - x_{\text{blue},CM}(t)\right)^2 + \left(y_{\text{red-flag}} - y_{\text{blue},CM}(t)\right)^2\right],$$

where T is the termination time for the run (which can vary depending on the selected termination condition; see **termination_code** in ***Contents of ISAAC_GA's Input Data File***). Notice that if the blue force is close to the red flag at all times (so that $d_{ave} \sim 0$), then $f_6 \sim 0$. Conversely, if the blue ISAACAs are forced, by red, to spend most of their time far from the red goal (so that $d_{ave} \sim d_{max}$), then $f_6 \sim 1$.

Mission primitive m_7

The seventh mission primitive consists of maximizing the number of red ISAACAs within an SC-defined distance of the blue flag.

The pertinent parameter for m_7 is $R_t(D)$= "number of red ISAACAs within a distance D of the blue flag at time t," where D is a user-specified parameter (see **flag_containment_range** in ***Contents of ISAAC_GA's Input Data File***). Note that no distinction is made between alive or injured ISAACAs. The minimal possible value of $R_t(D)$ is equal to zero, while the maximum possible value, R_{max}, clearly depends on D and is internally calculated by the program.

The fitness function for m_7 ($=f_7$) is given by a time average of $(R_t(D)/R_{max})^n$, averaged between t_{min} (corresponding to the earliest possible time that a red ISAACA could move to within a distance D of the blue flag) and t_{max} = T is the termination time for the run (which can vary depending on the selected termination condition; see **termination_code** in ***Contents of ISAAC_GA's Input Data File***):

$$f_7 = \frac{1}{T - t_{\min}} \sum_{t=t_{\min}}^{T} \left(\frac{R_t(D)}{R_{\max}} \right)^n .$$

Notice that if red is completely unable to penetrate blue's territory for the duration of the run, so that $R_t(D) = 0$ for times t, then $f_7 = 0$. Conversely, if red is able to maintain a constant presence within a distance D of blue's flag (which is likely only in the event that there are few or no blue forces defending the flag), then their mission fitness for this primitive approaches the value *one.*

Mission primitive m_8

The eighth mission primitive consists of minimizing the number of blue ISAACAs within an SC-defined distance of the red flag.

The pertinent parameter for m_8 is $B_t(D)$= "number of blue ISAACAs within a distance D of the red flag at time t," where D is a user-specified parameter (see **flag_containment_range** in ***Contents of ISAAC_GA's Input Data File***). Note that no distinction is made between alive or injured ISAACAs. The minimal possible value of $B_t(D)$ is equal to zero, while the maximum possible value, B_{max}, clearly depends on D and is internally calculated by the program.

The fitness function for m_8 ($=f_8$) is given by a time average of $(1 - B_t(D)/B_{max})^n$, averaged between t_{min} (corresponding to the earliest possible time that a blue ISAACA could move to within a distance D of the red flag) and t_{max} = T is the termination time for the run (which can vary depending on the selected termination condition; see **termination_code** in ***Contents of ISAAC_GA's Input Data File***):

$$f_7 = \frac{1}{T - t_{\min}} \sum_{t=t_{\min}}^{T} \left(1 - \frac{B_t(D)}{B_{\max}}\right)^n .$$

Notice that if blue is completely unable to penetrate red's territory for the duration of the run, so that $B_t(D) = 0$ for times t, then red may be said to have "succeeded in keeping blue away from its own flag" and $f_7 = 1$. Conversely, if blue is able to maintain a constant presence within a distance D of red's flag (which is likely only in the event that there are few or no red forces available to defend the flag), $B_t(D) \sim B_{max}$ and red's mission fitness for this primitive approaches *zero.*

Mission primitive m_9

The ninth mission primitive consists of minimizing the total number of red fratricide hits. This primitive is viable only if the **red_frat_flag** software "flag" is set equal to "1" in the **GA_DATA.dat** input data file, so that the red fratricide option during an ISAAC run is enabled (see below). Recall, also, that a fratricide "hit" is just that, a *hit,* and not necessarily a "kill." See ***Fratricide*** in ***ISAACA Combat.***

The pertinent parameter for m_9 is F_{red} = "total number of red fratricide hits during the run." (F_{red} is accumulated over the termination time T for the run, which can vary depending on the selected termination condition; see **termination_code** in ***Contents of ISAAC_GA's Input Data File***). The minimal possible value of F_{red} is obviously zero, while the maximum possible value is arbitrarily clamped at $F_{red,max} = R_0$, or the total number of red ISAACAs at time t = 0. If the actual value F_{red} exceeds $F_{red,max}$, F_{red} is internally redefined to equal $F_{red,max}$.

The fitness function for m_9 is given by $f_9 = (1 - F_{red} / R_0)^n$, Notice that if red suffers no fratricide hits at all, so that $F_{red} = 0$, then $f_9 = 1$. Conversely, if red ISAACAs are hit by friendly forces at least R_0 times, then red's overall mission fitness for this primitive is *zero.*

Mission primitive m_{10}

The tenth mission primitive consists of maximizing the total number of blue fratricide hits. This primitive is viable only if the **blue_frat_flag** software "flag" is set equal to "1" in the **GA_DATA.dat** input data file, so that the blue fratricide option during an ISAAC run is enabled (see below). Recall, also, that a fratricide "hit" is just that, a *hit,* and not necessarily a "kill." See ***Fratricide*** in ***ISAACA Combat.***

The pertinent parameter for m_{10} is F_{blue} = "total number of blue fratricide hits during the run." (F_{blue} is accumulated over the termination time T for the run, which can vary depending on the selected termination condition; see **termination_code** in ***Contents of ISAAC_GA's Input Data File***). The minimal possible value of F_{blue} is obviously zero, while the maximum possible value is arbitrarily clamped

at $F_{blue,max} = B_0$, or the total number of blue ISAACAs at time t = 0. If the actual value F_{blue} exceeds $F_{blue,max}$, F_{blue} is internally redefined to equal $F_{blue,max}$.

The fitness function for m_{10} is given by $f_{10} = (1 - F_{blue} / B_0)^n$, Notice that if red suffers no fratricide hits at all, so that $F_{blue} = 0$, then $f_{10} = 1$. Conversely, if red ISAACAs are hit by friendly forces at least B_0 times, then red's overall mission fitness for this primitive is *zero*.

ISAAC_GA's GA Recipe

ISAAC_GA couples a slightly older version of ISAAC's *Core Engine* with a basic GA algorithm adapted from [27] (see appendix D). In pseudo-code, the main components of this recipe appear as follows:

```
read GA_DATA and GA_ISAAC files
for generation=1,gmax
    for personality=1,pmax[25]
        decode chromosome
        for initial_condition=1,icmax
            run ISAAC's Core Engine
            calculate_fitness(initial_condition)
        next initial_condition
        calculate_mission_fitness()
    next personality
    find_the_best_personality()
    select_survivors_from_population()
    perform_single_point_crossover()
    perform_mutation()
    update_progress_report()
next generation
write best personality to file and close all data files
```

In words, **ISAAC_GA** first reads in two data files: **GA_DATA.dat** (that contains all variables pertaining to the GA) and **GA_ISAAC.dat** (that is a truncated form – appropriate for this slightly older version of the Core Engine – of the data input file described in ***Contents of ISAAC's Data Input File***). The contents of both files will be described below.

Next, the program uses a randomized pool of chromosomes to define the 1st generation of red personalities. For each such red personality, and for each of **icmax** initial spatial configurations of red and blue forces (remember that the blue personalities are fixed in **GA_ISAAC.dat**), the program then runs ISAAC's core engine to determine the mission fitness. After going through both loops, the program sorts the personalities according to their mission fitness values,

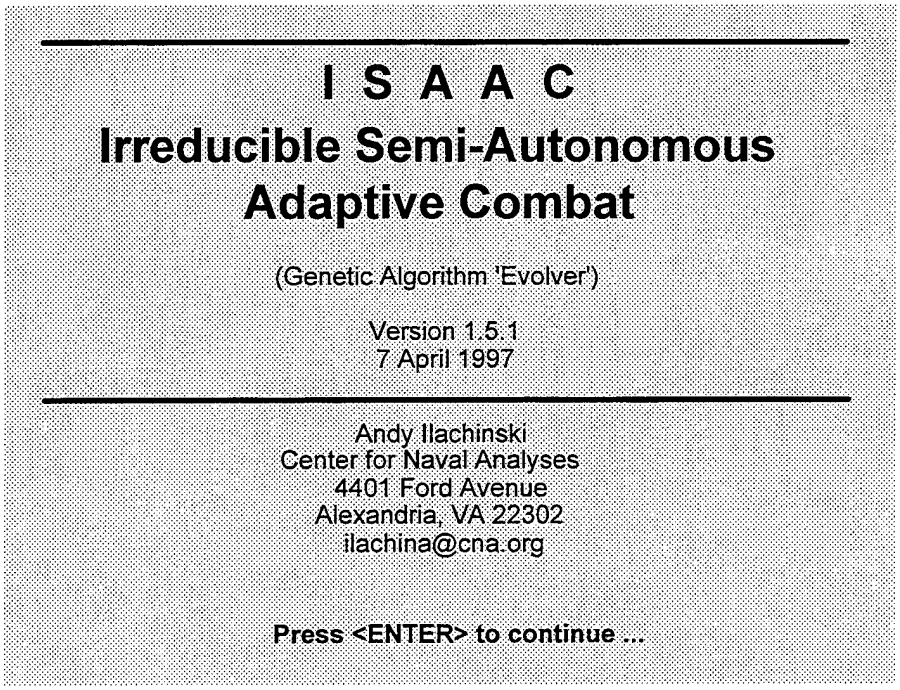
[25] Note that the total number of red personalities – i.e. population size – is defined by the variable **POPSIZE**, found in the header file **GA.h**; see appendix B and ***Concise User's Guide to ISAAC_GA***).

and selects some to be eliminated from the pool and others to breed. It then performs the basic GA functions of *crossover* and *mutation* (see Appendix B). Finally, having defined a new generation of red personalities, the whole process is continued until either the user interactively interrupts the evolution or the maximum allotted generation number has been reached.

Concise User's Guide to ISAAC_GA

As mentioned above, **ISAAC_GA** essentially provides a genetic algorithm "front-end" to a slightly older version ISAAC's *Core Engine* than the one described in the ***Overview of ISAAC*** section. Specifically, the version of ISAAC that is embedded within **ISAAC_GA** allows only one squad per side and excludes all command and control structures. This minor deficiency will be remedied in future versions.

Figure 69. **ISAAC_GA**'s opening screen



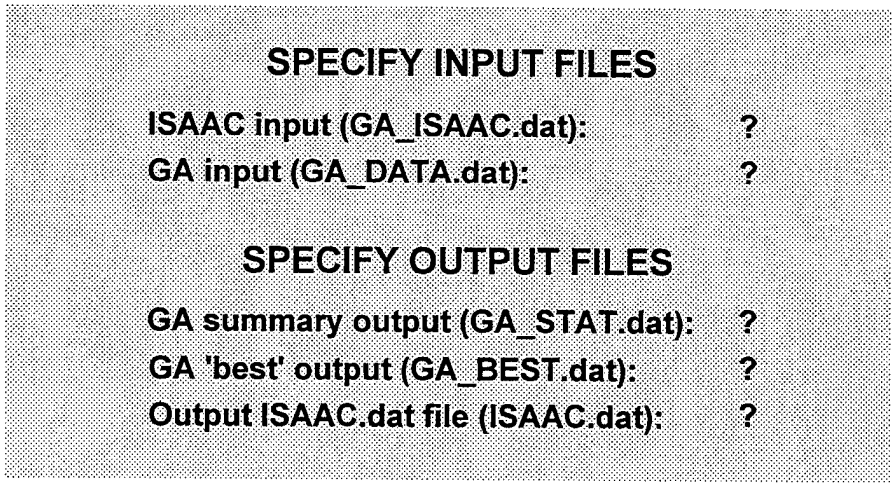
Starting ISAAC_GA

Assuming that the ISAAC "package" has been installed according to the instructions given in the previous section ***Installing ISAAC***, the genetic algorithm evolver can be run by going to the appropriate subdirectory on the hard drive (say, **C:/ISAAC>**) and typing the command **ISAAC_GA** followed by **<ENTER>** on the DOS command line. You will

see the opening screen (figure 69), specifying the current version and build date of the program and a prompt to press **<ENTER>** to continue.

The next screen prompts for a series of five file names (see figure 70): (1) **GA_ISAAC.dat**, which is the default name of the file that contains a truncated version of ISAAC's input data file (see ***Contents of ISAAC's Input Data File*** in ***A Concise User's Guide to ISAAC***); (2) **GA_DATA.dat**, which is the default name of the file that contain GA-specific data entries needed to start the run (its contents are described in the next section); (3) **GA_STAT.dat**, which is the file the user wishes to contain the statistical summary of the impending run (see ***Contents of ISAAC_GA's Statistics Output File: GA_STAT*** below); (4) **GA_BEST.dat**, which is the file the user wishes to contain a running record of all the "best" personalities as they are evolved by the program (see **best_personalities_to_file?** in ***Contents of ISAAC_GA's Data Input File: GA_DATA*** below); and (5) **ISAAC.dat**, which is a standard ISAAC data input file that the user wishes the program to write the best overall personality to (along with other fixed parameter entries from **GA_ISAAC.dat**) so that it can be run interactively using the *Core Engine*. Note that to run this file, the single-squad version of ISAAC must be used (see **ISAAC_SQ** in table 4).

Figure 70. **ISAAC_GA**'s file name prompt screen



Contents of ISAAC_GA's Data Input File: GA_DATA

GA_DATA.dat contains a user-modifiable listing of various GA-specific variables that are used to control the execution of the GA front-end to ISAAC. It consists of four separate sections (see figure 71):

- **GA Parameters**, which includes parameters specifying the maximum number of generations that the user desires to run, the number of red and blue initial spatial configurations to average over, software "flags" that toggle the use of specific ranges of genes in the personality chromosome, and so on.
- **GA Penalty Weights**, which includes parameters that assign relative weight values to each of the ten mission "primitives" defined above (see ***Mission Objective***).
- **Termination parameters,** which includes parameters specifying the exact conditions under which a given run will terminate.
- **ISAACA chromosome**, which includes parameters specifying the values of each of the 45 genes that make up a full chromosome defining a red personality

Short descriptions of each of the variables appearing in **GA_DATA.dat** are given below. These can be used as a reference guide for selectively altering this file's contents to tailor specific evolutions.

num_generations

This is the total number of generations that the user wants to run. A single generation consists of running ISAAC's core engine for each initial condition (see **num_initial_conds**) and each personality (where the total number of red personalities – i.e., population size – is defined by the variable **POPSIZE**, found in the header file **GA.h**; see Appendix B).

num_initial_conds

This is the total number of randomized initial spatial configurations of red and blue ISAACAs that will be averaged over in calculating a mission fitness for a given personality. The typical number of initial conditions used in the sample runs below are between 15 - 50.

max_time_to_goal

This variable sets a limit on the maximum number of iteration steps allowable per each run of the evolution. Depending on the termination condition (see **termination_code?**), a given run may end prior to the time specified in **max_time_to_goal**. Typical values for battlefield sizes of ~ 80-by-80 are between 100-150 iteration steps.

penalty_power

This variable refers to the power n used in defining the fitness function, f, for each of the ten mission primitives (see figure 68 in ***Mission Objective***). The value of **penalty_power** effectively determines how

rapidly f falls off from its maximal to minimal value (n=1 yields a linear fall-off, n=2 yields a quadratic fall-off, and so on).

best_personalities_to_file?

This software flag determines whether the program will automatically keep track of the best current personality (i.e., chromosome) during the evolution. If **best_personalities_to_file?** = 1, a user-specified file will contain a running tally of the best chromosomes for the entire run. In particular, whenever, after the first generation, the program finds a personality whose mission fitness exceeds that of the previously recorded personality it appends the appropriate data file with the better chromosome. Since the computational cost needed to perform this function is minimal, the user is encouraged to keep it always set equal to 1. If **best_personalities_to_file?** = 0, no updates of best personalities is made.

min_dist_genes_flag

This software flag controls the use of genes g_{36} through g_{42}, that define red's minimal distance constraints (see ***ISAACA Adaptability***). If **min_dist_genes_flag** = 1 these genes will be used in defining the red personalities, else they will not. Keep in mind that even if **min_dist_genes_flag** = 1, the program may itself determine that it would be "better" not to use any minimal distance constraints by finding an appropriate value of g_{36} = **min_dist_flag**.

initial_condition_genes_flag

This software flag controls the use of genes g_{43} through g_{45}, that define the size and x,y coordinates of red's initial spatial configuration. If **initial_condition_genes_flag** = 1 these genes will be used in defining red's initial condition, else they will not.

w1_time_to_goal

This variable defines the relative weight afforded to the 1st mission primitive (see ***Mission Objective***), that consists of *minimizing the time to goal.*

w2_friendly_loss

This variable defines the relative weight afforded to the 2nd mission primitive (see ***Mission Objective***), that consists of *minimizing the total number of red casualties.*

w3_enemy_loss

This variable defines the relative weight afforded to the 3rd mission primitive (see ***Mission Objective***), that consists of *maximizing the total number of blue casualties.*

w4_red_to_blue_survival_ratio

This variable defines the relative weight afforded to the 4th mission primitive (see ***Mission Objective***), that consists of *maximizing the ratio between red and blue casualties.*

w5_friendly_CM_to_enemy_flag

This variable defines the relative weight afforded to the 5th mission primitive (see ***Mission Objective***), that consists of *minimizing the cumulative distance between the center-of-mass of the red ISAACAs and the blue flag.*

w6_enemy_CM_to_friendly_flag

This variable defines the relative weight afforded to the 6th mission primitive (see ***Mission Objective***), that consists of *maximizing the cumulative distance between the center-of-mass of the blue ISAACAs and the red flag.*

w7_friendly_near_enemy_flag

This variable defines the relative weight afforded to the 7th mission primitive (see ***Mission Objective***), that consists of *maximizing the total number of red ISAACAs that are within a user-defined distance D* (see **flag_containment_range**) *of the blue flag.*

w8_enemy_near_friendly_flag

This variable defines the relative weight afforded to the 8th mission primitive (see ***Mission Objective***), that consists of *minimizing the total number of blue ISAACAs that are within a user-defined distance D* (see **flag_containment_range**) *of the red flag.*

w9_red_fratricide_hits

This variable defines the relative weight afforded to the 9th mission primitive (see ***Mission Objective***), that consists of *minimizing the total number of red fratricide hits.*

w10_blue_fratricide_hits

This variable defines the relative weight afforded to the 10th mission primitive (see ***Mission Objective***), that consists of *maximizing the total number of blue fratricide hits.*

termination_code?

This software flag controls how a run (for a given personality) will terminate. It can be assigned one of four integer values: 1, 2, 3 or 4. If **termination_code?** = 1, a run will terminate when the first red ISAACA reaches the blue flag. If **termination_code?** = 2, a run will terminate when the number of red ISAACAs within a range R =

flag_containment_range (see below) exceeds the threshold N = **containment_number** (see below). If **termination_code?** = 3, a run will terminate when the position of the red force's center-of-mass is closer to the blue flag than a threshold distance (defined by **red_CM_to_BF_frac**; see below). If **termination_code?** = 4, a run will terminate when the number of iterations t = **max_time_to_goal** (see above).

flag_containment_range

This variable sets a range around either the red or blue flags (depending on the values of other variables) which is used to count the number of ISAACAs *near a flag*. For example, if the relative weight for maximizing the number of red ISAACAs near the blue flag is nonzero (i.e., if the value of the variable **w7_friendly_near_enemy_flag** > 0), the value of **flag_containment_range** sets the pertinent range from the blue flag.

containment_number

If the termination flag is set for terminating a run when the number of red ISAACAs within a range R (= **flag_containment_range**) exceeds a certain threshold N – i.e., **if termination_code?** = 2; see above – N is specified by the variable **containment_number**.

red_CM_to_BF_frac

If the termination flag is set for terminating a run when the position of the red force's center-of-mass is closer to the blue flag than a threshold distance D – i.e., **if termination_code?** = 3; see above – D is specified by the variable **red_CM_to_BF_frac**.

ISAACA Chromosome Entries: gene[i]

The remaining entries – **gene[1]** through **gene[45]** – define not the values of the individual genes of a red personality chromosome but the *minimum* and *maximum* values that those genes are actually allowed to take in the program. For example, the first entry,

```
gene[1]:S_range          1,10
```

means that the first gene, corresponding to red's sensor range, can only take on values between 1 and 10. Note that the minimum and maximum entries for genes that correspond to the *signs* (+ or -) of other variables (such as **gene[5]:w1_alive_sign**) are 0 and 1, respectively. Either of these values can actually be set equal to any real value between 0 and 1. The sign is determined internally by generating a random number between 0 and 1, comparing this number to the gene "value" (also between 0 and 1 if the file shown in figure 71 is used), and

choosing the "+" sign if the random number > gene value, else choosing the "-" sign. A greater or lesser likelihood of choosing "+" versus "-" can therefore be regulated by selecting appropriate minimum and maximum entries for a given sign gene.

Figure 71. Sample ISAAC_GA data input file

```
************************************
*          GA parameters
************************************
num_generations                 250
num_initial_conds               50
max_time_to_goal                125
penalty_power                   2
best_personalities_to_file?     1
min_dist_genes_flag             0
initial_condition_genes_flag    0
************************************
*     penalty weights (1-100)
************************************
w1_time_to_goal                 0
w2_friendly_loss                10
w3_enemy_loss                   0
w4_red_to_blue_survival_ratio   0
w5_friendly_CM_to_enemy_flag    10
w6_enemy_CM_to_friendly_flag    0
w7_friendly_near_enemy_flag     0
w8_enemy_near_friendly_flag     0
w9_red_fratricide_hits          0
w10_blue_fratricide_hits        0
************************************
*       termination parameters
************************************
termination_code?               4
flag_containment_range          15
containment_number              10
red_CM_to_BF_frac               .5
************************************
*        ISAACA chromosome
************************************
gene[1]:S_range                 1,10
gene[2]:F_range                 1,10
gene[3]:C_range                 1,10
gene[4]:w1_alive                0,100
gene[5]:w1_alive_sign           0,1
gene[6]:w2_alive                0,100
gene[7]:w2_alive_sign           0,1
gene[8]:w3_alive                0,100
gene[9]:w3_alive_sign           0,1
gene[10]:w4_alive               0,100
gene[11]:w4_alive_sign          0,1
gene[12]:w5_alive               0,100
gene[13]:w5_alive_sign          0,1
gene[14]:w6_alive               0,100
gene[15]:w6_alive_sign          0,1
gene[16]:w1_injured             0,100
gene[17]:w1_injured_sign        0,1
gene[18]:w2_injured             0,100
gene[19]:w2_injured_sign        0,1
gene[20]:w3_injured             0,100
gene[21]:w3_injured_sign        0,1
gene[22]:w4_injured             0,100
gene[23]:w4_injured_sign        0,1
gene[24]:w5_injured             0,100
gene[25]:w5_injured_sign        0,1
gene[26]:w6_injured             0,100
gene[27]:w6_injured_sign        0,1
gene[28]:ADV_alive              0,20
gene[29]:CLS_alive              0,50
gene[30]:CBT_alive              0,50
gene[31]:CBT_alive_sign         0,1
gene[32]:ADV_injured            0,20
gene[33]:CLS_injured            0,50
gene[34]:CBT_injured            0,50
gene[35]:CBT_injured_sign       0,1
gene[36]:min_dist_flag          0,1
gene[37]:R_R_min_dist_alive     0,10
gene[38]:R_B_min_dist_alive     0,10
gene[39]:R_R_goal_min_alive     0,40
gene[40]:R_R_min_dist_injured   0,10
gene[41]:R_B_min_dist_injured   0,10
gene[42]:R_R_goal_min_injured   0,40
gene[43]:initial_box_size       1,50
gene[44]:initial_box_center_x   1,30
gene[45]:initial_box_center_y   1,30
```

Contents of ISAAC_GA's Data Output Files

As mentioned earlier, **ISAAC_GA** generates three output files:

- **ISAAC.dat**, containing a truncated version of ISAAC's input data file that can be read-in and run *as-is* by the single-squad version of ISAAC, **ISAAC_SQ**
- **GA_BEST.dat**, containing a running record of all chromosomes (and corresponding mission fitnesses) of the "best" personalities that were found during the evolution
- **GA_STAT.dat**, containing a statistical summary of the complete run (see below)

GA_STAT.dat

An annotated sample of the data fields appearing in **GA_STAT.dat** is shown in figure 72.

The file opens with the start date, start time and a listing of the mission "weights" that will be used during the run to calculate the mission fitness. There is also a reminder of the termination condition that the user has selected for this run (see discussion under **termination_code** in ***Contents of ISAAC_GA's Data Input File: GA_DATA***). This is all written to **GA_STAT.dat** prior to starting the actual run. Once the run begins, and at the conclusion of each generation (see pseudo-code in ***ISAAC_GA's GA Recipe*** above), the program continually updates this file with a summary of how well the evolution is proceeding. This summary consists of the generation number last completed, best and worst mission fitnesses found thus far (including all previous generations), best and worst mission fitnesses found during the immediately preceding generation, and the average fitness of the current population (+/- standard deviation).

Once the run is completed (whether *automatically*, by completing a full sweep through all user-requested generations or *interactively*, by a user by pressing the 'Q' (for *Quit*) key), the program records the parameter values defining the best overall red personality that was found during the run, along with its mission fitness value. The file concludes with a record of time, date and total elapsed time for the run.

Figure 72. Sample contents of the output file GA_STAT.dat

```
start date: 04/11/97
start time: 15:52:25

Fitness Parameters:

time to goal:          0.00
red loss:              0.00
blue loss:             0.00
red CM to blue flag:   0.00
blue CM to red flag:   0.00
red near blue flag:    1.00
blue near red flag:    0.00
red fratricide:        0.00
blue fratricide:       0.00

termination condition:  t_max=100

 generation    best       worst     best      worst    average   ave + standard   ave - standard   standard
   number      value      value     value     value    fitness     deviation        deviation      deviation
            (overall) (overall)   (gen)     (gen)     (gen)
     1         0.026     0.000     0.026     0.000     0.018        0.027            0.009          0.009
     2         0.048     0.000     0.048     0.000     0.031        0.045            0.017          0.014
     3         0.055     0.000     0.055     0.000     0.035        0.051            0.018          0.016
     4         0.058     0.000     0.058     0.000     0.028        0.042            0.013          0.015
     5         0.065     0.000     0.065     0.000     0.037        0.050            0.024          0.013
     6         0.065     0.000     0.052     0.000     0.036        0.052            0.020          0.016
     7         0.083     0.000     0.083     0.000     0.043        0.063            0.022          0.020
     8         0.127     0.000     0.127     0.000     0.041        0.059            0.024          0.017
     9         0.127     0.000     0.105     0.000     0.051        0.078            0.024          0.027
    10         0.127     0.000     0.110     0.000     0.063        0.095            0.032          0.032

Simulation completed

Best personality:
S-range = 9
F-range = 4
C-range = 9
w1_a    = 20.900
w2_a    = -8.300
w3_a    = 7.000
w4_a    = -23.500
w5_a    = -42.300
w6_a    = 21.600
w1_i    = -9.700
w2_i    = 67.000
w3_i    = 41.400
w4_i    = -50.600
w5_i    = -68.300
w6_i    = 54.000
ADV_a   = 7
CLS_a   = 49
CBT_a   = 30
ADV_i   = 12
CLS_i   = 23
CBT_i   = 48
R_R_a   = 0.00
R_B_a   = 0.00
R_RG_a  = 0.00
R_R_i   = 0.00
R_B_i   = 0.00
R_RG_i  = 0.00

Best fitness=0.1934

start date: 04/12/97
start time: 22:00:36
elapsed time: 108491 seconds
```

Start Time/Data

Mission Fitness "Weights"

summary of best,worst and average mission fitness for each generation

Parameters defining the "best" overall personality

Mission fitness of "best" personality

End Time/Data and Elapsed time for run

Sample Graphics Display

Once the user has selected the name of all input and output data files (see figure 70), **ISAAC_GA** runs through its initialization routine and displays the main graphics page.

A sample graphics page is shown in figure 73. Note that this figure assumes that the hot-keys **'B'** (for Battle-Space), **'C'** (for Chromosome) and **'F'** (for Fitness) have all been pressed (see ***"Hot-Key" Menu*** below). The display is broken up into six separate regions:

- A **banner-display region**, located at the top of the display and containing a large bold font, which identifies the program and release version, and the generation that is currently being processed.
- A text-based **fitness-summary region**, located directly beneath the banner-display region, which provides an up-to-date statistical summary of the genetic evolution (see below).
- A **battlefield region**, located near the bottom center of the display, which contains the battlefield view of what the current red personality is actually doing.
- A **fitness-parameters region**, appearing to the left of the battlefield, which contains a reminder of what mission fitness measure is being used for this run (see ***Mission Objective***).
- A **personality region**, appearing to the right of the battlefield, which contains parameters defining the *current* and *overall* best red ISAACA personality.
- A **"hot-key" menu region**, appearing at the bottom of the display, which contains a short menu of "hot keys" that the user can use to interrupt a run at any time to perform a variety of functions (see below).

Fitness Summary

The fitness-summary region, located directly beneath the banner-display region (see figure 73), consists of three columns of information that provides an up-to-date statistical summary of the genetic evolution.

The first (or left-most) column contains, from top to bottom, the number of the personality P (i.e., or chromosome) that the program is currently processing (expressed as a fraction of the total number of personalities in the genetic pool, P/P_{total}), followed by the current initial condition C (expressed as a fraction of the total number of initial

conditions that the program will average the fitness value over, C/C_{total}), followed by the current time step t of the sample that is being run for the C^{th} initial condition for the P^{th} personality (expressed as a fraction of the maximal allotted time for this run, t/t_{total}).

The second (or middle) column contains, from top to bottom, the value of the fitness of the $(n-1)^{st}$ personality, where the nth personality is the one currently being run (the value P/P_{total} appearing at the top of the left-most column), the fitness of the "best" personality of the immediately preceding generation, and the fitness of the "worst" personality of the immediately preceding generation.

The third (or right-most) column contains, from top to bottom, the average fitness value of the preceding generation, the fitness of the overall "best" personality that has been found thus far (up to and including the immediately preceding generation), and the fitness of the "worst" personality thus far (up to and including the immediately preceding generation).

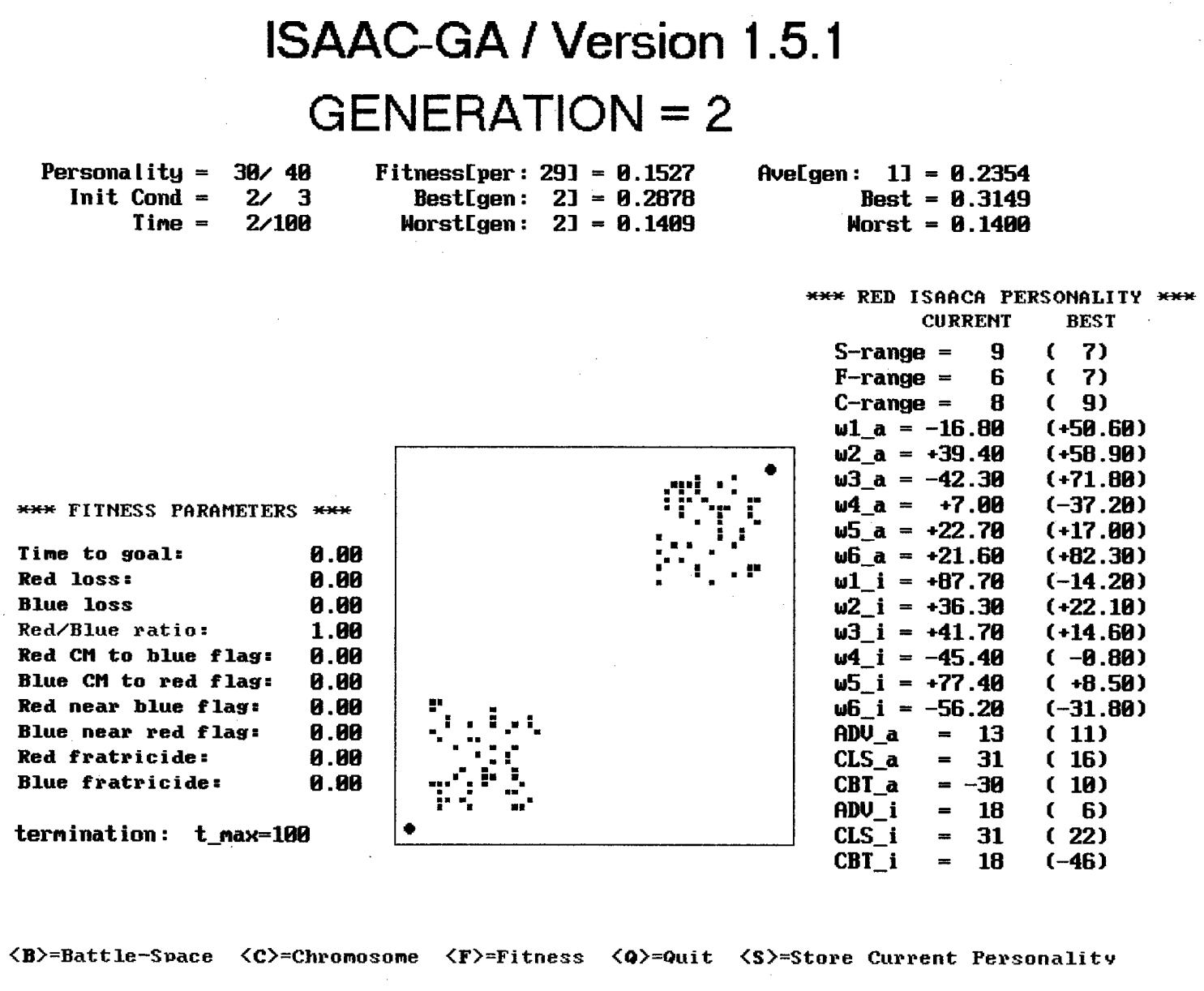
"Hot-Key" Menu

The colored words at the bottom of the battlefield comprise a short menu of (black-colored) "hot keys" that the user can use to interrupt a run at any time to perform a specific function. There are five functions, accessed by the following keys (and defined according to the order in which they appear, left to right, on screen):

- **"B"** (for **B**attle-Space): toggles the graphics display of the notional battlefield. Note that such a display is time-wise somewhat costly (slowing down apparent computation speed about 40%). Because speed is of the essence in genetic algorithm evolutions (see below), this option should be used sparingly to obtain glimpses of how a particular personality is doing.
- **"C"** (for **C**hromosome): toggles the display of the parameter values defining the current and best personality on the right-hand-side of the display.
- **"F"** (for **F**itness): toggles the display of the mission objectives defining fitness for this run, along with a reminder of how each sample during this run is to be terminated.
- **"Q"** (for **Q**uit): closes all output data files (saving intermediate results) and quits the program.
- **"S"** (for **S**tore Current Personality): stores the parameters defining the red personality that is currently being processed to a data file that can then be read-in and run as-is by the single-squad version of ISAAC's core engine (i.e., **ISAAC_SQ**). In practice,

one "sees" an interesting behavior taking place on the notional battlefield (having accessed the graphical display of the battlefield by first pressing the "B" hot-key; see above) and then presses the "S" hot-key to record the personality that is responsible for that behavior.

Figure 73. ISAAC_GA's main graphics display



Sample Runs

In this section we present a few illustrative sample runs using the genetic algorithm front-end to ISAAC's core engine.

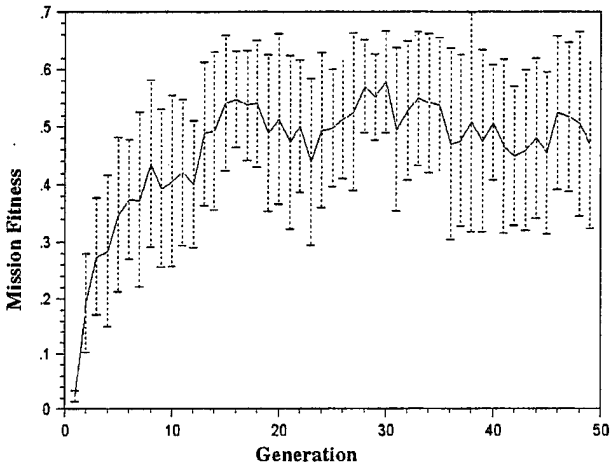
Typical Run-Times

In order to give the reader an idea of how much processing time a GA run requires (so that the values of pertinent variables can be adjusted according to available hardware – and patience!), consider a "typical" scenario in which 50 red and 50 blue ISAACAs engage in notional battle on an 80-by-80 battlefield. If the genetic "pool" is populated with 50 personalities and the program is asked to average over 25 initial conditions, typical run times on a 166 MHz Pentium computer average about ***5 - 6 hrs for 50 generations***. Keep in mind, also, that these values represent fairly small scenarios, and are used here for illustrative purposes only. Research caliber runs require 100 or more personalities, 50 – 100 initial conditions to average over and larger force sizes (~100 or more per side). Such scenarios currently take well over 24 hrs to run (or to reach a satisfactory "saturation" level; see below) on a Pentium-class computer. A major programming focus for future versions of the genetic algorithm evolver will be to improve the required run times.

Typical Learning Curves

Figure 74 shows a typical learning curve for scenarios such as the one outlined above. Recall that mission fitness is a number between zero and one; numbers close to *zero* representing *"low fitness"* (according to the specific mission objective selected for a given run), and numbers close to *one* representing *"high fitness."* The bars in figure 74 represent the lower and upper bounds of mission fitness for a given generation, as defined by the absolute deviation from the fitness average.

Figure 74. Typical GA learning curve



While learning curves will, of course, be different for different runs, some basic features are common to most curves. For example, because all runs begin with a random population of personalities, the initial gene pool tends to be relatively poor at performing whatever mission has been specified for a given run. The mission fitness at t = 1 will therefore be typically low. As the GA sorts the personalities according to their fitness values, and evolves this pool, the mission fitness will generally rise; fairly quickly at first, then eventually saturating at a value that represents the effective fitness maximum for a given mission objective. Since not all objectives are equally as amenable to a GA "solution" – in fact, some may not be "solvable" at all given the parameter space available to the GA – the value of the mission fitness at which any given curve saturates may not be as close to the value *one* as the user *a-priori* desires. The learning curve shown in figure 74 saturates near $f \sim .5$ at t ~14, which is a fairly typical characteristic saturation time for scenarios with 50 ISAACAs per side and a gene pool consisting of 50 personalities.

Figures 76 through 79 provide color "snapshots" of several sample GA runs using **ISAAC_GA**. Table 11 also gives short descriptions. All of these runs can be played back in their entirety by using the stand-alone "play-back" program **ISAAC_SQ** (see table 4).

Table 11. ISAAC output files corresponding to the sample GA runs shown in figures 76 through 79

Sample GA Run	Figure	ISAAC output file[1]	Brief Description
1	76	**AWAY_1.out**	Red's mission: *keep blue away as far from red flag as possible for t=100 steps*; "best personality"
		AWAY_2.out	Red's mission: same as in **AWAY_1.out**; 2nd "best personality"
		AWAY_3.out	Red's mission: same as in **AWAY_1.out**; Blue is more aggressive; red uses different "tactic"
		AWAY_BAD.out	Red's mission: same as in **AWAY_1.out**; Early "bad" personality
2	77	**GOAL_2.out**	Red's mission: *get to blue flag and minimize red casualties*; all red survive but few red get to blue flag (i.e., relatively low fitness)
	77	**GOAL_4.out**	Red's mission: same as in **GOAL_2.out**; 92% red survive and almost all red get to blue flag (i.e., high fitness)
	78	**GOAL_6.out**	Red's mission: same as in **GOAL_2.,out;** Red's tactic is to *"weaken center then go around blue defenses"*
	79	**GOAL_1.out**	Red's mission: same as in **GOAL_2.,out;** Red's tactic is to *"spread and weaken, then surge through the thinned blue defenses"*

[1] Output files are provided on the accompanying disk.

Sample GA Run #1

Figure 76 shows a few snapshots taken from a play-back of the files **AWAY_1.out**, **AWAY_2.out**, **AWAY_3.out** and **AWAY_BAD.out** (using **ISAAC_SQ**). The first three files represent the "best" GA-evolved red personalities for performing the following mission: *"Keep blue ISAACAs as far away from the red flag as possible, for as long as possible (up to a maximum 100 iteration steps)."* This means that the mission fitness f will be close to its maximal value *one* only if red is able to keep all blue ISAACAs pinned near their own flag (at a point farthest from the red flag) for the entire duration of the run, and f will be near its minimal value *zero* if red allows blue ISAACAs to advance completely unhindered toward the red flag. For comparison, the last file, **AWAY_BAD.out**, contains an early "bad" red-personality that performs this particular mission poorly. Combat unfolds on a 40-by-40 notional battlefield, with 35 ISAACAs per side. The GA is run using a "pool" of 50 red personalities for 50 generations, and each personality is averaged over 25 initial spatial dispositions. Figure 75 shows a fragment of **ISAAC_GA**'s input data file for this run (see ***Contents of ISAAC_GA's Data Input File: GA_DATA***).

Figure 75. Fragment of **GA_DATA.dat** input data file for Run #1

```
************************************
*
*          GA parameters
*
************************************
num_generations                 50
num_initial_conds               25
max_time_to_goal                100
penalty_power                   2
best_personalities_to_file?     1
min_dist_genes_flag             0
initial_condition_genes_flag    0
************************************
*
*     penalty weights (1-100)
*
************************************
w1_time_to_goal                  0
w2_friendly_loss                 0
w3_enemy_loss                    0
w4_red_to_blue_survival_ratio    0
w5_friendly_CM_to_enemy_flag     0
w6_enemy_CM_to_friendly_flag     10
w7_friendly_near_enemy_flag      0
w8_enemy_near_friendly_flag      10
w9_red_fratricide_hits           0
w10_blue_fratricide_hits         0
************************************
*
*      termination parameters
*
************************************
termination_code?                4
flag_containment_range           12
containment_number               10
red_CM_to_BF_frac                .5
```

The *best* red personality that the GA was able to find for this mission appears in **AWAY_1.out**. The snapshots of this run (taken at times t=25, 50 and 100), show that red is very successful at keeping blue forces away from its own flag. In fact, the closest that red permits blue ISAACAs from approaching the red flag – during the entire allotted run time of 100 iteration steps – is some point roughly near midfield. In words, the "tactic" here seems to be – from red's perspective – *"fight all enemy ISAACAs in sight while moving toward the enemy flag slowly enough to compel the enemy to keep following."* Note that this tactic is fairly robust, in the sense that if the battle is initialized with a different spatial disposition of red and blue forces (while keeping all personality parameters fixed), red would perform this particular mission about as well.[26]

The *second best* red personality, **AWAY_2.out** (whose snapshots are taken at times t=25, 50 and 90) shows a slightly less successful, but innovative, "tactic." Here, just as in **AWAY_1.out**, red ISAACAs initially move away from their own goal to meet the advancing blue forces (see time = 25). Once combat ensues, however, any red ISAACAs that find themselves locally isolated "double back" toward the red flag to regroup with other reds and thereby form an impromptu secondary defense against possible blue leakers. Because a few blue ISAACAs manage to fight their way near the red flag at later times (see snapshot for time = 90), the overall tactic is not as successful as the one used in **AWAY_1.out**.

The snapshots for **AWAY_3.out** show the tactic used by the best red personality found by the GA after the blue force is made a bit more *aggressive* (by increasing blue's personality weight for moving toward red – i.e., weight components $w_{blue,2}$ and $w_{blue,4}$; see ISAACA Personalities – by 50%). Red's new tactic is completely different, and in fact proves to be even more successful (from a mission fitness point of view) than the tactics used in the previous two examples. Here, red quickly "spreads out" to cover as much territory as possible and "strikes" the enemy as soon as the blue ISAACAs come within view. As their territorial coverage is thinned either through attrition or movement toward the blue flag, other red ISAACAs (namely, those previously positioned near the periphery of the battlefield) move inward to fill any voids. This tactic succeeds in not only preventing any blue ISAACAs from reaching the red flag, but manages to push most of the surviving blue force back toward its own flag! As the case with previous tactics, this tactic is also fairly robust, and is not a strong function of the initial spatial disposition of red and blue forces.

The last example in figure 76, **AWAY_BAD.out**, shows a few snapshots from an early "bad" red personality. The mission and blue personality are the same as in **AWAY_3.out**. Initially, red appears to behave as it does in **AWAY_3.out**, as red ISAACAs quickly disperse outward to cover a large area. But, because at this early junction in the "evolution," the

[26] The reader can verify this fact by running **AWAY_1.dat** using **ISAAC_SQ**, and by randomizing the initial conditions by pressing the "N" (i.e. ra<u>N</u>dom) hot-key.

GA has not yet had the time to "fine tune" all of red's genes, red is in this instance unable to prevent blue ISAACAs from penetrating deeply into red territory.

Sample GA Run #2

Figures 77 through 79 show a few GA-evolved "tactics" for the following mission: *"Get to the blue flag as quickly as possible while minimizing red casualties."* Except for the values appearing in the ***penalty weights*** section of **ISAAC_GA.dat**, the GA's input data file for this sample run is the same as the one shown in figure 75. The ***penalty weights*** section must be amended so that all weights are zero except for w_1 and w_7:

```
w1_time_to_goal              10
w7_friendly_near_enemy_flag  10
```

As in the previous example, the GA is run using a "pool" of 50 red personalities for 50 generations, and each personality is averaged over 25 initial spatial dispositions.

Figure 77 shows snapshots of the evolution of two early red "attack tactics." Red's first tactic (stored in the play-back file **GOAL_2.out)** is to station its forces out of reach of blue's fire power, and then – after creating an "opening" on blue's right flank by slowly drawing out a few enemy ISAACAs – to send a small section of reds toward and around that opening. While, in the end, all reds survive, the overall mission has not been a particularly successful one (from the mission fitness point of view) because only a relatively few reds have actually made it to the blue flag.

Red's second tactic (stored in the play-back file **GOAL_4.out**, and illustrated by the sequence of snapshots appearing at the bottom of figure 77) proves to be more successful. Here, red again first advances toward blue's defensive position, maintaining a station at a far enough range so as to avoid blue's fire power (see snapshot for time = 25). Then, after a relatively long period of time during which there is much "undulating" (or random "posturing") on both sides, red exploits a perceived weakening in blue's forces (near the center region of blue's defensive position) and quickly *strikes*, sending most of its force through the blue defenses and toward the enemy flag. As blue ISAACAs counterattack and surround the penetrating red ISAACAs, a second squad of reds penetrates through a newly created "hole" in blue's defense. The result, from red's point of view, is that 92% of the initial force has successfully penetrated through to the blue flag.

Figure 78 shows snapshots of a third red tactic for this same mission. The run can also be played back in its entirety by using **ISAAC_SQ** (i.e., the single-squad version of ISAAC; see table 4) to "play back" the file

GOAL_6.out. The tactic exploits (or *sacrifices!)* a few red ISAACAs at the front of the advancing red force. The snapshots for times 15 and 20 show that as most of the force splits into two groups and moves off toward blue's left and right flanks, a few red ISAACAs (those that are originally near the center of the split) proceed to move forward and penetrate blue's defense. By enticing blue to counterattack red's penetration (by sending forces toward the middle), red effectively dilutes blue's strength along the outer edges of its defensive station. This, in turn, creates "openings" on both sides of blue's defense through which the two separate groups into which red had earlier split can now move virtually unopposed. The snapshot for time 90 shows that red has successfully penetrated through to the blue flag well before the maximum allotted time for this run has expired (t_{max} =100).

Snapshots of the fourth, and final, sample red tactic for this same mission is shown in figure 79. This run can be viewed by using **ISAAC_SQ** to "play back" the file **GOAL_1.out**. Red's tactic here is again, as in the previous example, to exploit a few red ISAACAs at the front of the advancing red force. This time, however, red does not need to sacrifice these ISAACAs. Instead, red uses them to split apart blue's forces in order to temporarily "weaken" the center region of blue's defense. As soon as this center region is sufficiently weakened, red quickly penetrates through to the blue flag. What is surprising is the robustness of red's personality with respect to this tactic. Red is more often than not able to successfully employ the same general tactic against an arbitrary blue initial force disposition.

What is most surprising about many of these runs is that *the red force appears to task different ISAACAs with different missions, despite the fact that each red ISAACA is endowed with exactly the same personality!* Thus, in figure 79, the higher-level tactic "use the two forward positioned ISAACAs to weaken the enemy's center" emerges out of the collective interactions of the same low-level decision rules: an apparent centralized order induced by decentralized local dynamics.

Figure 76. Snapshot views of GA-evolved personalities for scenario GA_1

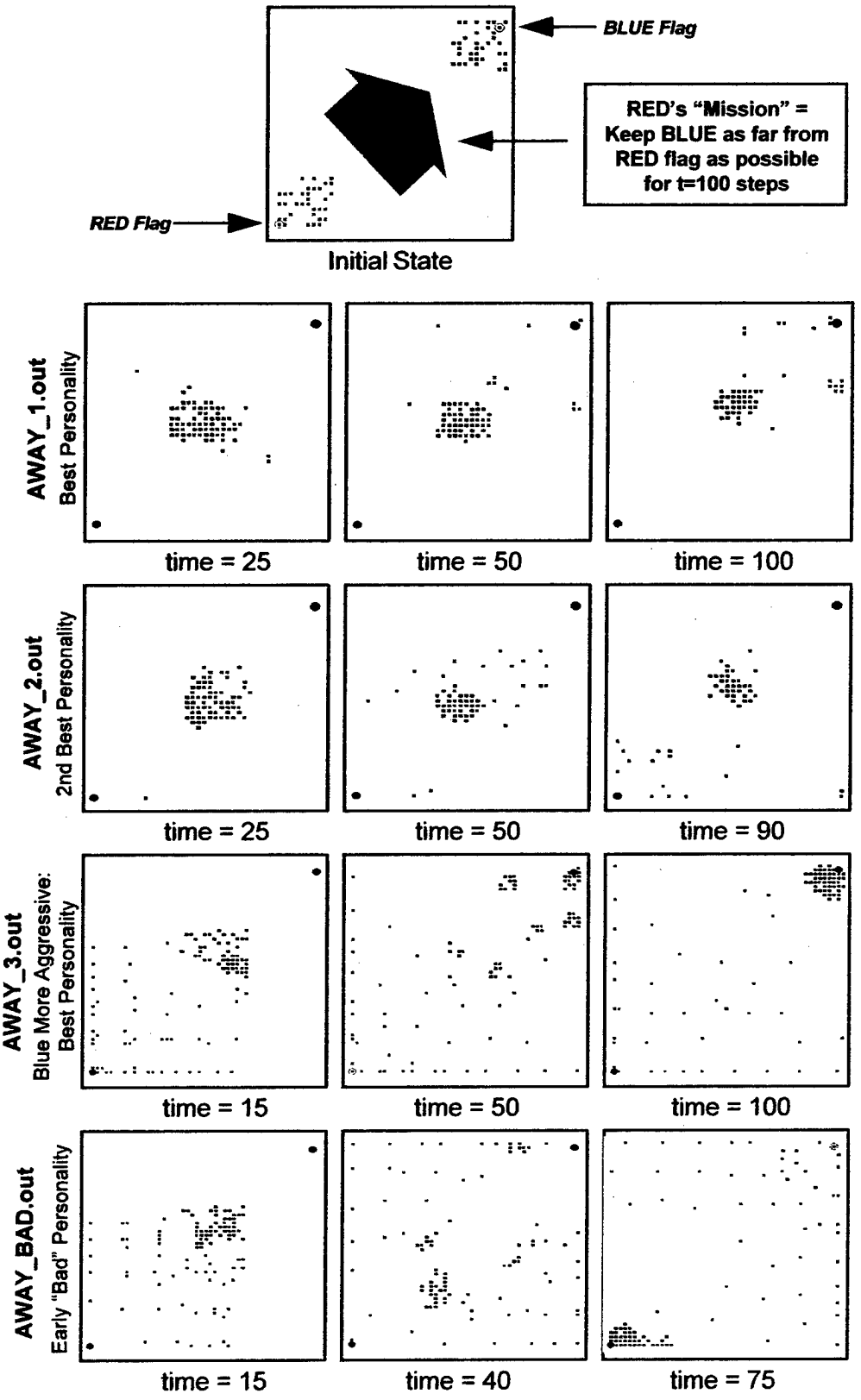


Figure 77. Snapshot views of GA-evolved personalities for scenario GA_2

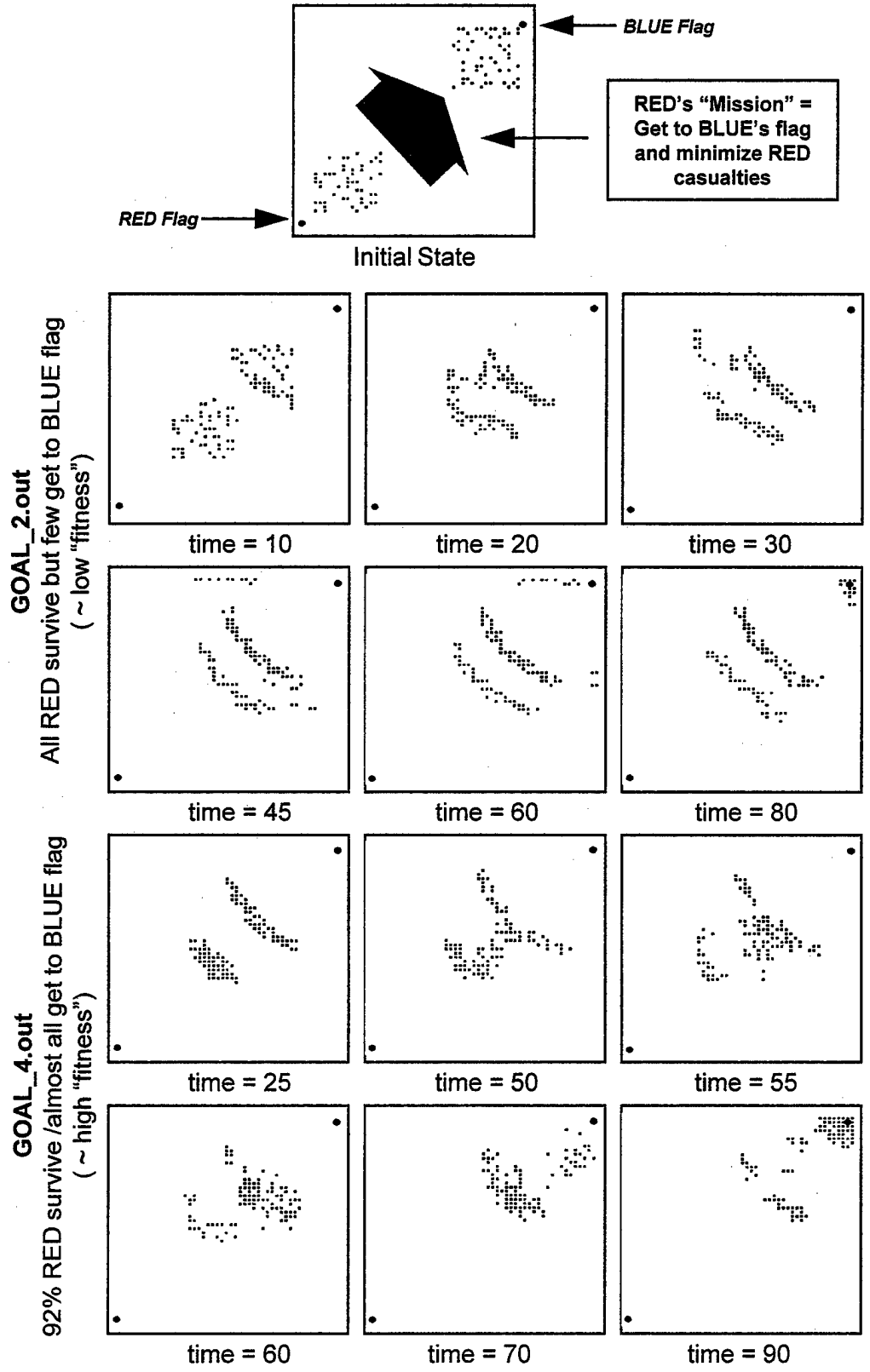


Figure 78. Snapshot views of GA-evolved personalities for scenario GA_2

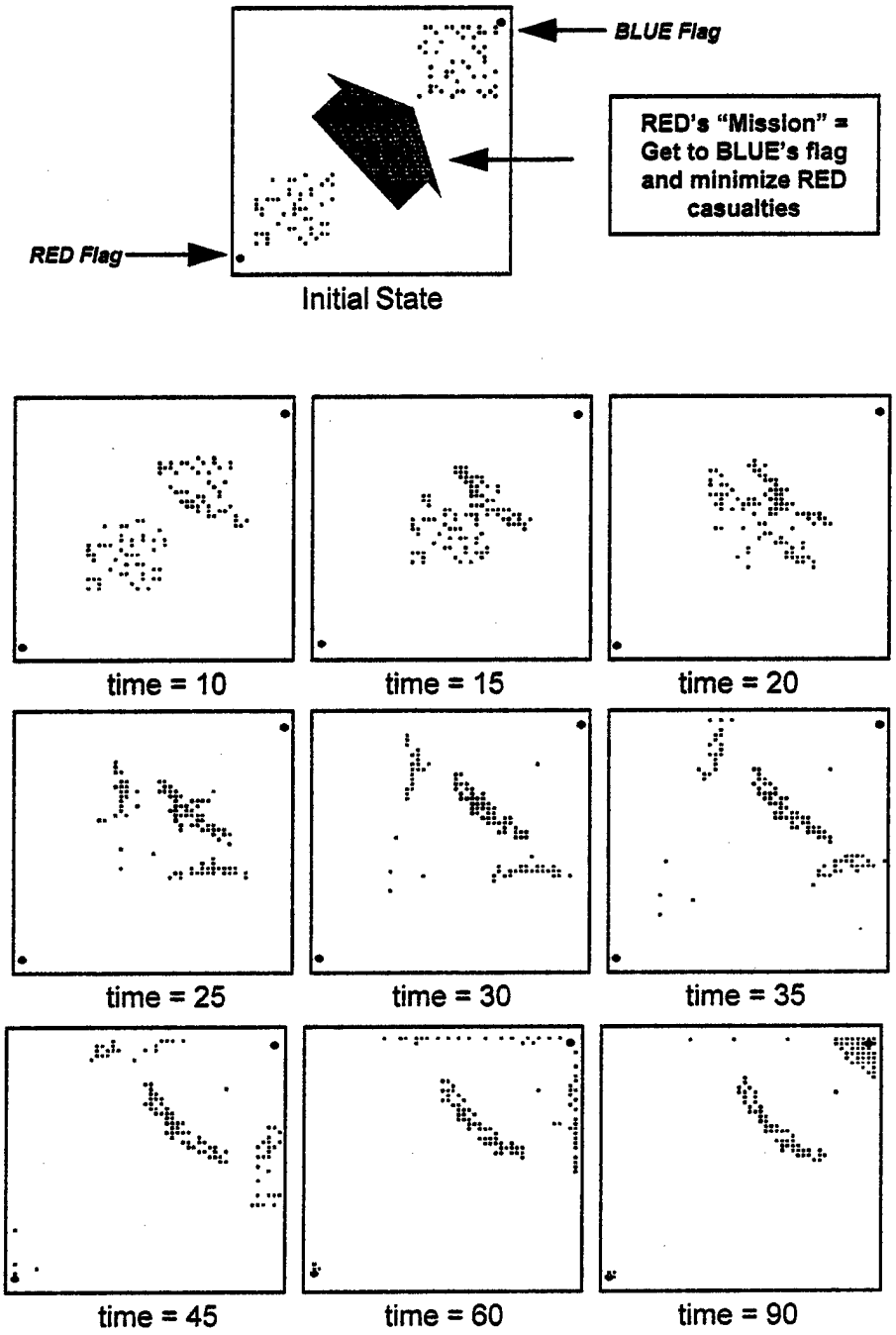
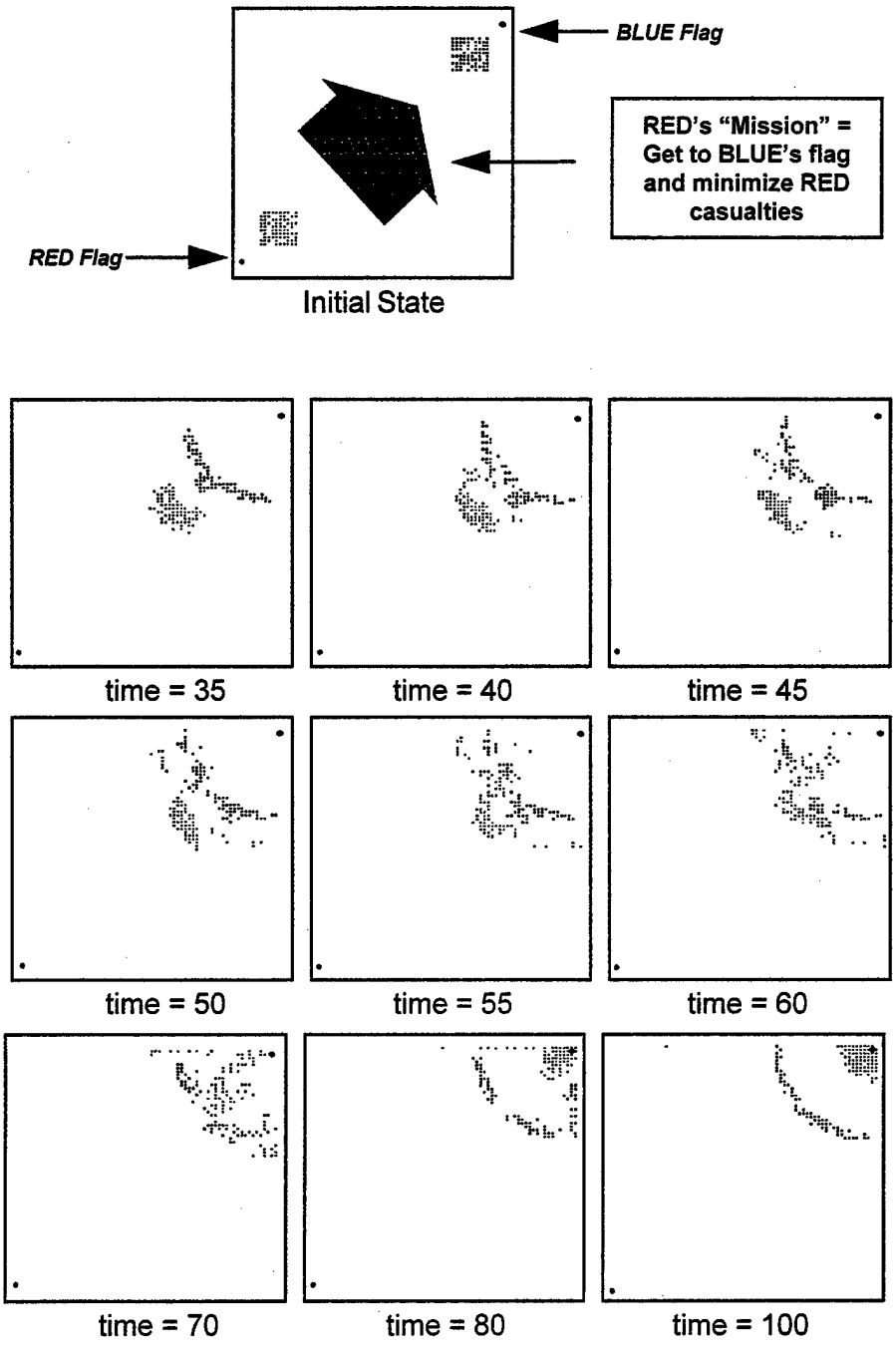


Figure 79. Snapshot views of GA-evolved personalities for scenario GA_2



Future Enhancements to ISAAC

Table 1 (see page 8) lists nine generic properties of complex adaptive systems and briefly summarizes their relevance to land combat. While the details of the given comparison can be debated, as can the actual characteristics that the table purports are shared by most complex adaptive systems, the fact that it makes any sense at all to compare some of the more obvious properties of land combat with those of complex systems is in itself significant. It suggests that, in principle, land combat ought to be amenable to precisely the same methodological course of study as any other complex adaptive system, such as the stock market, a natural ecology, or the human brain.

The obvious first step to take in such a course of study is to develop a complex systems theoretic model of land combat "from the ground up" (no pun intended!), making full use of the modeling and simulation tools that have been developed by the complex systems theory community in recent years to understand the behaviors of models of many real complex adaptive systems. This means, in particular, that a description of land combat must be developed that fundamentally derives not from the solution of a set of Lanchester equations (though, depending on the problem, this may be an entirely appropriate course to take), but from a context in which the dynamics of combat are driven by (1) a medium to large number of semi-autonomous agents, (2) agents that are able to adapt — intelligently and in real-time — to changing conditions, and (3) agents that filter, assimilate, and react only to local information.

The version of ISAAC described in this paper represents a tentative first step toward developing an inherently complex systems theoretic model of land combat, and is motivated by a desire to extend the largely conceptual links between complex systems theory and land combat, as outlined in table 1, to forge a set of practical connections as well.

Assuming land combat *can* be described as a complex adaptive system, the second — and ultimately most important — step is to determine the universal high-level emergent behaviors that result from such a description. The ultimate goal for ISAAC is for it to form the backbone of a general purpose complex systems theoretic analyst's toolbox for identifying, exploring, and possibly exploiting emergent collective patterns of behavior on the battlefield.

Future versions of ISAAC will include many enhancements to the "core engine" described in this paper. These enhancements will both provide a greater sense of realism and enrich the overall battlefield environment. There are five general categories of future enhancements:

- Basic enhancements to ISAAC's "Core Engine"
- Memory and learning
- Nested ISAAC dynamics
- Data collection enhancements
- Enhancements to GA evolution

Basic Enhancements to the "Core Engine"

Basic enhancements will include (but not be limited to):

- More realistic ISAACA state-space
- Enhanced offensive and defensive capabilities
- An enhanced command and control structure
- Enhanced Personality "Value-Systems"
- Greater "Depth" to, and Variety of, Local Moves
- Added environmental realism
- Enhanced Combat Adjudication.

More Realistic ISAACA State-Space

ISAACAs currently exist in one of only three possible states: *alive, injured,* and *dead.* Each ISAACA can therefore be encoded as a simple scalar element: equal to 2, say, if alive, equal to 1 if injured and equal to 0 if no longer "playing." In future versions of ISAAC, the inner state-space of each ISAACA will be enhanced by several additional factors, and will thus effectively be described as a vector quantity. These enhancements will include:

- *Health*
- *Morale*
- *Combat quality*
- *Experience*
- *Movement vectors*
- *Meta-personality templates*

An ISAACA's *health* at time t will reflect its overall combat-readiness. *Morale* – meaning "a spirit, as of dedication to a common goal, that unites a group"[27] – will be a basic measure of unit cohesion and general "fighting spirit." It will increase with (perceived) local combat "success" and decrease with increasing damage (to a single unit and/or a unit's squad).

Combat quality will be a general measure of how "well" a given ISAACA performs its own mission, where "well" is measured relative to an ISAACAs maximum combat characteristics. Higher combat quality will assure a higher offensive "mission success" rate. For example, firepower and maneuverability distributions might be weighted more towards the higher end, say, using tail-end weighted beta-distributions. Combat quality will increase with an ISAACA's *experience,* with higher quality assuring a lesser degradation of morale under adverse conditions.

Since, in the current version of ISAAC, ISAACAs respond only to the static configuration of nearby ISAACAs, direction and speed are completely ignored. Future versions of ISAAC that incorporate a local memory (see below) will also encode an ISAACA's speed and direction in a *movement vector.*

Factors such as health, morale, combat quality, and experience can all be used to define generalized ISAACA "profiles," or meta-personality rule-templates, for associating given personality types with given local contexts. In the current version of ISAAC, each ISAACA is endowed with a single *fixed* personality (as defined by its personality weight vector) and, perhaps, a few additional *fixed* constraints (such as advance, cluster, and combat thresholds). In future versions, ISAACAs will be endowed with *context-dependent meta-personality templates* that specify what fixed personalities (and what additional constraints) will be used at what time.

An increased state-space also allows for individual ISAACAs to be ranked according to their defensive vulnerability (see next subsection).

Enhanced Offensive and Defensive Capabilities

In the current version of ISAAC, ISAACAs have a *single notional offensive weapon* (characterized by a single-shot probability and a constraint on the maximum number of enemies that can be targeted at one time) and a *single defensive capability* (characterized by the "number of hits" required to degrade a state from alive to injured or from injured to "killed"). Future versions of ISAACAs will include a wider range of both offensive and defensive capabilities, including:

---

[27] *American Heritage Dictionary,* Third Edition, CD-ROM version 3.6, Houghton Mifflin Company, 1994.

- *a mix both short- and long-range weapons,* with appropriate context-sensitive rules (both local and command-related) prescribing their use
- *weapon store,* thus removing the unrealistic current infinite store of required weapons and ammunition
- *aim accuracy*
- *probability of kill,* p_{kill}, *that is different from single-shot "hit" probability,* p_{ss}, and will depend on individual targeted enemy units
- *terrain camouflage,* wherein terrain effects are included in both defensive posturing strategies and offensive capability (see below)
- *weapon-specific vulnerability.*

ISAAC will also include some form of *reinforcement,* wherein the user (i.e., Supreme Commander) will define the manner in which injured and/or killed ISAACAs will be replaced with "fresh" (i.e., alive) combatants.

An Enhanced Command and Control Structure

As with almost all other components of the current version of ISAAC, the rules defining command and control are very crude and not very realistic. On the local command level, for example, these rules consist essentially of providing a common reference point for clustering and issuing local movement vectors to subordinate ISAACAs. On the global command level, these rules consist of vectoring the movement of local commanders toward the enemy flag by using certain global information (that is not directly accessible by the local commanders themselves) and prescribing the manner in which local commanders may "help" other local commanders. These rules are currently very ad hoc, and represent but one way among many to accomplish the same basic tasks.

An enhanced, more realistic command and control rule structure must better respect the self-similar manner in which decisions are made on each level of the hierarchy. Namely, the action of each decision-maker (whether it is an ISAACA, a local commander, global commander or the supreme commander) is predicated on that decision-maker answering exactly the same series of fundamental questions:

- **Question 1:** What is my current goal (which may depend on context and therefore change as the battle unfolds)?
- **Question 2:** What resources are at my disposal?

- **Question 3:** How do I best make use of my available resources to satisfy my current goal?

For example, an individual ISAACA, which represents the simplest agent populating the ISAACian landscape, answers this series of questions as follows:

- **ISAACA Answer 1:** to get to the blue flag (as an example)
- **ISAACA Answer 2:** my only resource is my ability to move (either a distance 1 or 2, depending on my movement range, or to stay in my current position)
- **ISAACA Answer 3:** I must minimize my local penalty function (which is defined as a weighted-sum of my current and projected distances to nearby ISAACAs and my own and enemy flags).

Similarly, a local commander currently answers the same series of questions as follows:

- *LC Answer 1:* to get to the blue flag (as an example)
- **LC Answer 2:** the individual ISAACAs that make up my squad
- **LC Answer 3:** I order all of my subordinate ISAACAs to move toward the center of that "patch" in my local command area that minimizes my local command penalty function

It is clear that the local commander's answer #3 is, at best, only a "first approximation" of the solution to what is, in reality, a very complicated optimization problem. In the current version of ISAAC, each LC orders *all* of its subordinate ISAACAs to move toward the *same point.* While an optimization, of sorts, is performed, it consists only of determining what patch in the LC's command area to send all of the LCs subordinates to. This is done purely for expediency, and does at all represent the "best" approach.

A more realistic – albeit more time-consuming – approach is for the LC to *issue individually tailored movement orders to each of its subordinates*; each order being deduced by solving a local optimization problem. That is to say, the LC's local decision problem really consists of finding the "best" possible combination of moves for each of its subordinates, given the overall state (consisting of both friendly and enemy ISAACAs) within its local command area. Ideally, such a decision must also be based on the LC's *prediction* of future states, making LC decisions very chess-like, at least locally (see ***Nested ISAAC Dynamics***).

Similarly, a more realistic decision problem that must be solved by a global commander is to find the "best" possible combination of moves (including criteria, or "templates" for LC-issued orders to their own subordinates) for each of its subordinate local commanders, given the overall (perceived) state of the battlefield.

In summary, future versions of ISAAC will include a more robust command and control dynamical structure that consists of level-specific "decisions" based on solving local optimization problems.

Enhanced Personality "Value-Systems"

The early version of ISAAC described in this paper introduces the potentially powerful idea of basing the local decision-making process on individual *personalities*. Personalities, however, are thus far defined fairly crudely, and currently consist solely of assigning relative weights to the desire to move closer toward alive or injured red and blue ISAACAs and either of the two "flags."

More sophisticated personal "value-systems" can be imagined. For example, provision might be made to allow individual ISAACAs to more explicitly weigh tradeoffs between risk versus potential (i.e., perceived) payoff. The personality weight vector can also be generalized to allow for user specification of different sets of personality "types." For example, a certain number of ISAACAs on a given side can be declared "defenders," and another "attackers."

In the current version of ISAAC, each ISAACA is endowed with at most a single "goal," namely a propensity to move toward or away from the enemy flag (as defined by an appropriate component of its personality weight vector). If the local command option is enabled, individual ISAACAs may also be ordered by their local commander to give some weight to moving towards temporarily issued "local goals." In future versions of ISAAC, individual ISAACAs will act according to unique, but *time* and *context dependent* goals. In particular, ISAACAs will be able to alter their own personalities as a function of both experience and surrounding conditions. For example, local goals can depend on an ISAACA's position within the battlefield; ISAACAs near their own flag may assume a more "defensive" role (than that defined by their default personality) while an ISAACA's "offensive" drive may increase as it approaches the enemy flag.

Generalized Personality Matrix

Currently, the most general ISAACA personality – defined by weight vector $\vec{w}$ – is a function only of *state* (i.e., either alive or injured) and *squad* (if there is more than one). That is, each ISAACA can be assigned two different weight vectors (and, thus, two different personalities) – one for when it is in the alive state and one for when it is injured – and

these two personalities can be different for ISAACAs belonging to different squads. All ISAACAs belonging to the same squad, however, are assigned exactly the same weight vector. Moreover, in the case of a multi-squad force, the actual propensities for moving toward or away from other ISAACAs (as defined by $\vec{w}$) do not themselves depend on the squad that those other ISAACAs belong to. In other words, ISAACAs are currently unable to distinguish among ISAACAs that belong to different squads. Communication is also "blind" to squad labels in this sense.

In future versions of ISAAC, personality weight vectors will be generalized in two ways:

- $\vec{w}$ will be a function of the complete vector quantity that characterizes an ISAACA's inner-state space (i.e., health, morale, experience, etc; see ***More Realistic ISAACA State Space***). This generalization will greatly enrich the dynamical range of personality-driven responses that an individual ISAACA can make.
- The actual propensities for moving toward or away from other ISAACAs (as defined by $\vec{w}$ for ISAACA X, say) will themselves depend on what X *perceives* to be the properties of those other ISAACA's. In other words, the way in which X responds to an ISAACA Y (or incorporates Y into its local penalty calculation) will depend on the properties of Y that ISAACA X chooses to associate with Y (or that X *senses* are possessed by Y).

Hostility Rings

In the current version of ISAAC, ISAACAs treat all other ISAACAs within their sensor range *equally*, except for assigning different relative weights to ISAACAs of different types (alive friendlies, alive enemies, etc.). In future versions of ISAAC, ISAACAs will also be surrounded by concentric "hostility rings" (or annuli) defining the relative degree of importance assigned to neighboring ISAACAs. For example, a set of enemy ISAACAs S_1 that are closer to ISAACA X than another set S_2 may be assigned a higher weight (or priority). (See division of global command sectors into hostility rings in ***GC Command of Autonomous LC Movement.***)

Recall sample run #10 (see figure 45), which illustrates the effect of increasing red's sensor range relative to blue's. In discussing this run, we speculated that as side X is forced to assimilate more and more information (with increasing sensor range), there inevitably comes a point at which X's overall fighting ability is effectively curtailed because X's available resources are spread too thinly. This is assuming, of course, that X's resources and/or tactics (i.e., "personality") remain

fixed. An obvious question to ask regarding this example, is "How ought X to adapt its personality in order to perform (its mission) at least as well using increased sensor capability?" Part of the solution might depend on each ISAACA having the ability to *prioritize* all of the information contained within its sensor field. A genetic algorithm can then be coupled with this extended local dynamical parameter space to search for "optimal" local prioritization schemes.

Another possibility is to pattern a more sophisticated internal value-system after Smith's "Calculus of Ethics" [30].

Greater "Depth" to, and Variety of, Local Moves

In the current version of ISAAC, ISAACAs are, at any time t, constrained to move to one of either 8 (if the movement range $r_M = 1$) or 24 (if the movement range $r_M = 2$) *nearest neighbor* sites. In future versions, ISAACAs will not only be allowed to move over greater distances (thereby effectively adding a velocity parameter to the overall parameter specification list of a given ISAACA; see ***More Realistic ISAACA State Space***), but will also be able to develop local *tactics* and *strategies* of projected sequences of moves (see ***Memory and Learning***).

For example, as the environment is enhanced to include various types of terrain and obstacles (see **Added Environmental Realism**), ISAACAs will be forced to weigh factors beyond the simple "proximity to enemy and friendly force" factor that is currently the sole determinant of its local penalty function. Local move decisions will, in future versions, require some form of rule-based "tactics" to dynamically integrate such factors as line-of-sight, the "passability" of a given terrain type, and degree of camouflage.

Added Environmental Realism

The addition of terrain and other obstacles to the environment simultaneously adds a layer of complexity to the kinds of local moves ISAACAs can take and increases the level of sophistication of local tactics and strategies. Different kinds of terrain will include...

- flat/rough
- road
- forest
- hill
- river
- minefields

- structure (bridges, storage, bunkers, etc.)

Terrain will also be characterized by

- *altitude* (which will affect line-of-sight)
- *movement* (i.e., *"passability"*) *index*, which will be a function of the kind of ISAACA occupying a given battlefield cell
- *camouflage* (or *"fog"*) *index*, which will affect visibility and/or identifiability)

A simple way to implement a fog-index is to add two rule "templates": (1) *Visibility Rule-Template* = ISAACAs located in, say, a forest cell C_{forest}, are made visible only to ISAACAs that are immediately adjacent to C_{forest}, and (2) *Defensive Enhancement Rule-Template* = provide a specified fractional increase to the defensive strength of all ISAACAs within a cell of a given terrain type.

Enhanced Combat Adjudication

In the current version of ISAAC, combat is resolved in a very crude manner. Each ISAACA is given an opportunity to "fire" at any enemy ISAACA that is positioned within that ISAACA's fire range. If an ISAACA is shot by an enemy ISAACA (with a user-specified probability), its current state is degraded either from alive to injured or from injured to killed. In future versions of ISAAC, combat adjudication will be enhanced in at least two ways:

1. The addition of more realistic *lethality contours* surrounding each ISAACA. For example, some ISAACAs may have a greater forward firepower or have a firepower that diminishes with range to target.

2. The addition of *selective power projection;* i.e., the ability to tailor an engagement strategy to a local context (see below).

The enhanced set of engagements strategies will include:

- **Direct ISAACA <—> ISAACA fire** – in which ISAACA X "sees" an enemy Y (and vice-versa) and both engage in one-on-one combat. The outcome is determined probabilistically, as in the current version, but takes into account weapon strength, range, morale, defender's strength and visibility.

- **Area fire** – in which an ISAAC X "knows" or suspects that an enemy ISAACA Y is located within an area A (consisting of, say, an N-by-N array of battlefield "cells") and blindly fires at a random cell or cells in A.
- **Collective fire** – in which a set of ISAACAs – $X_1, X_2, \ldots, X_n$ – coordinate their fire into a patch of enemy territory (an area of size A at range R).

Targeting Strategies

In the current version of ISAAC, each ISAACA *individually* decides to target a randomly selected set of enemy targets (up to the user-specified maximum number allowed) that are located within a fire range r_F of their position. In future versions, all power projection and targeting strategies will be decided in a more "intelligent" fashion by incorporating information about enemy defenses, position, movement vectors, perceived health, morale, combat quality, and so on.

Locally, ISAACAs will weigh such tradeoffs as targeting less capable but closer enemy units (that may therefore be more likely to be "hit") vice targeting more capable enemy units that are located farther away (and that may therefore be less likely to be "hit" if targeted). On a local command level, local commanders will coordinate fire among its subordinate ISAACAs by issuing *targeting priorities* and *engagement strategies* (direct, area, or collective; see above).

Memory and Learning

As is true of any complex adaptive system, land combat consists not just of mindless combatants following some prescribed set of rules, but of intelligent and *adaptive* combatants who, over time, can both learn from their past mistakes and modify the default rule set that they initially entered combat with.

In the current version of ISAAC, however, ISAACAs are very limited in their ability to modify their default personalities. Their adaptability consists essentially of being able to slightly alter their default personalities according to a set of local threshold constraints, measured with respect to a user-specified threshold range (see ***ISAACA Personalities***). For example, while certain ISAACA's might have a personality that drives them to always move toward friendly ISAACAs (according to some positive relative weight), they might also be driven by an auxiliary constraint condition that effectively clamps that default positive weight to *zero* whenever they are surrounded by a threshold number of friendly forces. In this way, their weights — and therefore, their personalities — adapt to local contexts.

But this adaptation is clearly very basic. While weights may be either set to zero or have their sign flipped (from positive to negative or vice versa), their actual relative values never change. Instead — for maximum growth potential — ISAACAs need to be able to both adaptively change their entire personality structure and learn from their past experiences.

One simple way to augment ISAAC's current adaptability algorithm is to define *meta*-personalities that would, for example, either increase the relative weight for moving toward an enemy as the distance to enemy forces decreases (to define a class of "strongly aggressive" forces), or increase the relative disparity between moving toward an enemy and moving toward the enemy's goal as the distance to that goal decreases (to define a personality class that becomes more eager to attain the goal as it gets closer to it). (See discussion in ***More Realistic ISAACA State-Space***).

Another, more powerful way to make ISAACAs more flexible in adapting to their environment is to incorporate some form of *memory.*

Memory

ISAACAs currently have no memory. At each time step, they assimilate the information within their sensor's field-of-view and either choose to "do nothing" or move into some adjacent site. All previous moves are "forgotten," and no "anticipated" future moves (such as might be part of a projected series of moves, or strategy) affect their decision-making process.

Future versions of ISAAC will include an ISAACA memory. Each ISAACA will be able to store, retrieve, and incorporate into its decision-making process a memory of a certain number of its most recent moves. Moreover, ISAACAs will be able to retrieve a certain number of past configurations within their sensor's field-of-view; i.e., the actual positions of all friendly and enemy ISAACAs within their sensor range. Memory of the past successes and failures of other ISAACAs, and those of the strategies of previous local and global commanders, will also be considered.

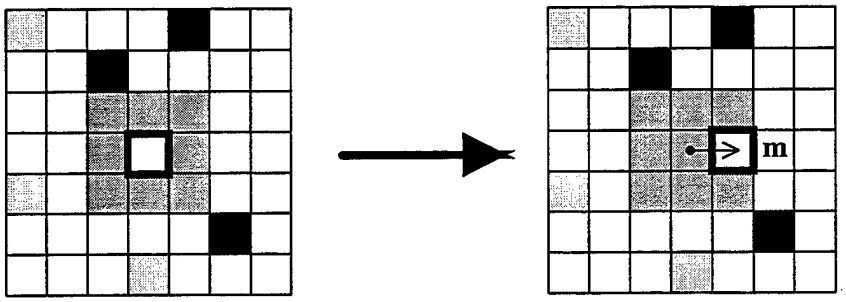
A memory capability — particularly when coupled with a more expansive set of possible moves that ISAACAs will be allowed to select from on any given time step (see ***Greater "Depth" to, and Variety of, Local Moves*** above) — leads naturally to the development of *tactics* and *strategies.* For example, individual moves will no longer be defined (as they are now) solely in terms of a local penalty minimization, but will involve both (1) a minimization of *projected penalty* (wherein an ISAACA will select a move that is based, in part, on an anticipated sequence of moves, $(m_1,m_2,...m_n)$, and their consequences), and (2) a consideration

of the efficacy of past moves made in similar configurational contexts (see discussion below).

Neural-Network-Derived Move Selection

In the current version of ISAAC, move selection is based on a few simple local rules. These rules are "hard-wired" in at run time. They also make only a very limited use of the total information within an ISAACA's sensor range. In particular, an ISAACA does not explicitly take into account the exact disposition of forces in its field-of-view, basing its decision to move, instead, on only the absolute or relative *numbers* of nearby ISAACAs. A more powerful approach is to enable ISAACAs to make better use of all of the information that is locally accessible by them, which includes knowing *force positions* as well as *force strength.* A powerful tool with which this can be accomplished is the *neural network* (see pages 104-116 of [2]).

Figure 80. Illustration of a neural-net assigned move m (at time t+1) given a local state S at time t



State S at time t

Neural net prescribes move **m** for time t+1

The idea would be to teach an ISAACA (in either supervised or unsupervised fashion) to associate a particular move (= **m**) with a given overall local state (= ***S***). The local state includes the numbers and positions of all friendly and enemy forces. In later versions of ISAAC, ***S*** will also include terrain and other obstacles.

Reinforcement Learning

Reinforcement learning is the problem that any autonomous adaptive agent faces when learning a strategy via trial-and-error interactions with its environment. There are two general strategies for attacking this problem [31]: (1) search through the space of all possible behaviors to find the one that performs "best" in a given environment (this is the approach taken by genetic algorithms); or (2) find a well-defined method of assigning credit to individual actions taken in response to

given states of the environment. The latter strategy, provided by reinforcement learning theory, involves a wide variety of techniques: greedy strategies, randomized techniques, adaptive heuristics, Q-learning, Bayesian reasoning techniques, and temporal-difference learning, among many others.

In each case, the underlying idea is the same. An ISAACA is connected to its environment via its *sensor* (which feeds information to the ISAACA out to within a range r_C) and *action*, which, in the general case, will involve strategies consisting of several projected moves. At each iteration step, an ISAACA receives input about the current state of its environment and chooses a particular move as output. Since this action obviously changes the state of the environment (of which each ISAACA is an integral part), one can imagine that — once a move is made — there is either an implicit or explicit *reinforcement signal* that is fed back to the ISAACA, telling it how "good" its move really was. For example, "Did it really get closer to enemy ISAACAs, as it wanted?" Or, "Did it mistakenly get farther and lose sight of some enemy forces?" An ISAACA ought to choose moves (and form strategies) — essentially, map actions to states (as depicted in figure 80) — that tend to *maximize some long-term measure of reinforcement.*

Note that this is very different from, say, supervised neural-net learning, which involves teaching an agent to associate input and output pairs by learning a test set of "training facts." In reinforcement learning, an ISAACA may be given an immediate "reward" after making a move (thumbs up/thumbs down, or some other measure), but it is *not* told explicitly which move would have been the best one to take. The ISAACA must come up with an optimal strategy by itself, using only its experience. Its task (as well as the designer's) is made more difficult still, by the fact that the criteria for assigning an appropriate reinforcement signal for current or past actions is far from trivial. For example, it is generally difficult to decide which one move (or set of moves), out of a sequence of moves that ends in a high-payoff end-state, was actually the "best" one to take, and is therefore the move (or moves) to which the highest "reward" ought to be assigned.[28] It is also not always clear what the "end-state" is, or how to long to wait to make an assignment. One class of techniques that is designed to deal with this problem is the so-called method of *temporal differences* [32]. This class takes its name from the fact that it consists essentially of adjusting the values of states according to differences between the immediate reward and the estimated value of the next state.

A recent, and in many ways remarkable, application of reinforcement learning to game playing is Tesauro's temporal difference algorithm for backgammon [33]. Since backgammon has on the order of 10^{20} possible states, it is impractical to use a brute-force search strategy.

[28] This general problem is known as the *Credit-Assignment Problem,* and is discussed by many authors. For a discussion, see [31].

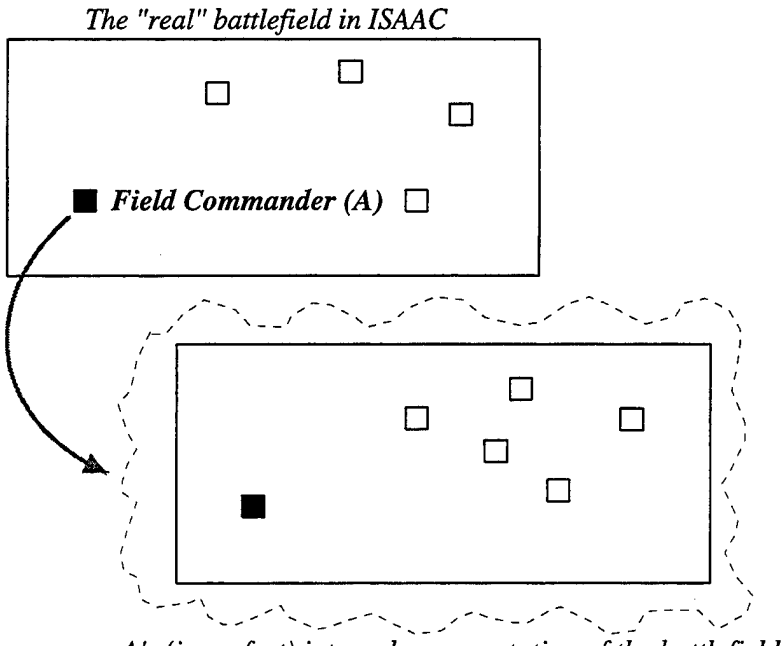
Tesauro's model (TD-Gammon), which also uses a backpropagation neural-net to aid the temporal difference learning algorithm, has learned to play backgammon so well that, at the time of this writing, it is considered to be one of the best "players" in the world.

Current design plans for future versions of ISAAC include building some form of reinforcement learning techniques into each ISAACA's decision-making process.

Nested ISAAC Dynamics

Agents in a real complex adaptive system can be expected to behave and adapt according to some internal model that they have constructed for themselves of what they *believe* their environment is really like. In particular, field commanders base their command decisions, in part, on what their intelligence support tells them has happened thus far on the battlefield and what the enemy's current order-of-battle is, and, in part, on their *own best intuition of how events will unfold in the future.*

Figure 81. A schematic representation of "nesting" in ISAAC



Sometimes, if the environment is simple enough, such models of potential futures are fixed and simple; sometimes, if the environment is complex, agents need to actively construct hypothetical models and test them against a wide variety of assumptions about initial states and rules and so forth. What must eventually be added to ISAAC is the ability to

use nested representations of the unfolding combat to allow each local and global commander to essentially create and manage simulations of the entire battlefield. Commanders must thus be able to base their decisions and behavior on their simulated picture of the battle (see figure 81).

Note that this split-level nesting allows a myriad of fundamental command and control questions to be asked:

- *What is the real "value" of information?*
- *How can I quantify the effects of incomplete knowledge?*
- *How can I exploit what I know the enemy does not know about me?*
- *How does my combat effectiveness degrade with decreasing reliability of information?*
- *etc. ....*

(For a further discussion of these issues, see the concluding section – ***Two Closing Speculations***). It should also be noted that a nested dynamics such as the one described here is also a basic design feature of the Santa Fe Institute's more general-purpose *Swarm* modeling system (see ***Recent Examples of Agent-Based Simulations*** in the ***Introduction***).

Data Collection Enhancements

To facilitate a *quantitative* vice purely qualitative understanding of the unfolding patterns of combat, ISAAC provides a rudimentary set of data collection tools. These tools currently consist of (1) time series plots of various changing quantities describing the step-by-step evolution of a given battle, and (2) measures of "how well" certain primitive mission objectives are met at a battle's conclusion. The first group of tools (using built-in statistics measures) yields quantitative snapshots of a battle as it unfolds in time; the second group (using a simple parameter-space mapping technique) yields semi-quantitative measures of "success" at a mission's end. Details are given in ***Data Collection.***

As ISAAC's core engine is enhanced in future versions, so too will ISAAC's core set of data collection tools be enhanced to provide a better quantitative understanding of the overall ISAACian dynamics. It is impossible to predict the precise form that these data collection enhancements will take, except to say that they will obviously be developed alongside (and therefore complement) whatever enhancements are made to ISAAC's core engine. Below, we give a feel for the kind of enhancements that can be made by briefly discussing three enhancements that were planned for the current version but were not included for lack of time:

- Trajectory-difference Measures
- Combat Entropy
- Activity Maps

The general spirit behind making any future enhancements to ISAAC's data collection capability will be to incorporate ideas from Tier-IV of the "Eight Tiers of Applicability" discussed in [2] (see Table 2). Recall that Tier-IV applications consist of using nonlinear-dynamics and complex systems theoretic inspired measures to describe the complexity of combat; i.e., power-law scaling, Lyapunov exponents, entropic measures, attractor reconstruction techniques, relativistic information, etc.

Trajectory-Difference Measures

Military historians are fond of citing examples from past conflicts to argue that "were *it not for factor X*" an outcome of a battle might have been very different. Factor X can be a "few more good men," "greater will to fight," or a "field commander's intuition." While such arguments may or may not be directly strengthened by ISAAC, ISAAC can be used to explore such issues by measuring differences between two (or more) trajectories.

Specifically, future versions of ISAAC will include a facility to *visually display the difference between two evolutions*, using different sets of initial conditions, combat personalities, and/or force strengths. One technique – called a *trajectory-difference plot* (TDP) – is to color a pixel at position (x,y) if and only if the information at (x,y) is *different* for two runs, A and B. Assuming the initial spatial dispositions for red and blue are the same for A and B, and that the factor X is "small" (that is, it does not immediately have a pronounced effect on the overall evolution), the TDP will initially be mostly blank. Colored areas will begin appearing as A's and B's trajectories diverge. In this way, one will be able to directly explore such questions as *"What difference will it make to increase my force strength by 10%?"* or *"What difference will it make to be more aggressive (or defensive)?"*

Combat Entropy

Carvalho-Rodriques [34] has suggested using entropy, as computed from casualty reports, as a predictor of combat outcomes. Whether or not combat can be described as a complex adaptive system, it may still be possible to describe it as a dissipative dynamical system (see [1], page 28). As such, it is not unreasonable to expect entropy, and/or entropy

production, to act as a predictor of combat evolution. Carvalho-Rodriques defines his casualty-based entropy E by

$$E_i = \frac{C_i}{N_i} \log \frac{1}{C_i/N_i}$$

where C_i represents the casualty count (in absolute numbers) and N_i represents the force strength of the i^{th} adversary (either red or blue). It is understood that both C_i and N_i can be functions of time.

The plot of the functional form $E(x) = x \log (1/x)$, where $x = C_i / N_i$, has a peak at about 0.37. One could interpret this to mean that once C_i/N_i goes beyond the peak, "it is as if the combat capability of the system ... declines, signifying disintegration of the system itself."[29]

Woodcock and Dockery [35] provide strong evidence that casualty-based entropy is a useful predictor of combat. They base this on analysis on both time-independent and time-dependent combat data derived from detailed historical descriptions of 601 battles from circa 1600 to 1970, exercise training-data obtained from the National Training Center and historical records of the West-Wall campaign in World-War II and Inchon campaign during the Korean war.

They find that plots of E_a (attacker entropy) versus E_d (defender entropy) are particularly useful for illustrating the overall combat process:

- *Region I:* a low entropy region corresponding to low casualties and ambiguous outcomes. Initial phases of a battle pass through this region, with the eventual success or failure for a given side depending on the details of the trajectory in this entropic space
- *Region II:* a region of high entropy for the defender and low entropy for the attacker indicates the attacker wins
- *Region III:* a region of ambiguous outcomes, like region I, region III represents high attrition with outcomes depending on the direction of the trajectory. (Woodcock and Dockery indicate that only simulated combat appears able to reach this region.[30])
- *Region IV:* an analogue of region II, where the entropy roles are reversed and the defender wins.

Woodcock and Dockery further suggest that the measurement and display of coupled casualty and reinforcement rates may be a first step towards quantifying the *battle tempo.* "The tempo is then seen to

---

[29] Reference [35], page 197.
[30] Reference [35], page 223.

characterize, not the physical rate of advance (the usual connection), but rather the rate of structural breakdown of the fighting force."[31]

We note, in closing, that Carvalho-Rodriques's definition of a casualty-based entropy is but one possible definition. One could alternatively use generalizations of the Renyi-entropy, Kolmogorov-Sinai entropy, or topological entropy, among many other definitions. Despite the seeming simplicity of the basic idea, there is strong evidence to suggest that entropy will play a fundamental role in understanding the underlying dynamical processes of war.

Activity Maps

In the current version of ISAAC, individual ISAACAs adapt to changing conditions in a very simple manner. Adaptability is essentially confined to either altering the sign of, or zeroing out completely, one or more components of an ISAACA's personality weight vector. Future versions will include a more sophisticated meta-personality-driven dynamics, in which personalities become full functions of local context (see ***More Realistic ISAACA State-Space***).

An important insight into the dynamics of an unfolding battle may be gained by examining a battle's *activity map*; i.e., a map not of the positions (and movement vectors) of individual combatants (such as is currently provided by ISAAC's simple graphical display of individual ISAACA positions), but a map that represents *the manner in which each combatant is adapting to changes in his local environment.*

An activity map of areas in which the combatants' default rule set is adequate for dealing with local conditions will show little or no activity. An activity map of areas in which the combatants' meta-personalities are repeatedly used to alter their default rules sets will highlight a high activity level. Areas of higher activity may be correlated with local dynamical conditions that are particularly "sensitive to perturbations" and/or with far-from-equilibrium behavioral patterns. ISAAC can then be used to explore such questions as *"How can I (as a local commander) introduce a set of combat conditions that will keep the enemy in a 'highly active' state while maintaining a relatively stable state for my own forces?"*

An activity map can also be thought as a particular example of a more general *decision-space map*, in which sites on the battlefield are colored coded to represent the "decision-flow" of ISAACAs and/or local and global commanders. One can imagine using ISAAC to address such questions as *"What kinds of decisions does an ISAACA make?"; "When?"; "Why?"; "What local dynamics and patterns tend to disrupt an enemy's decision-making capability?"*, among others.

---

[31] Reference [35], page 227.

Enhancements to GA Evolution

Reference [2][32] discusses several recent attempts to use genetic algorithms to "evolve" strategy and tactics, including deriving tank tactics, and to act as the critical dynamical component of tactical decision aids. As one might suspect, genetic algorithms also figure prominently in design plans for future versions of ISAAC.

Future versions of ISAAC will be able to be run in any of three distinct modes:

1. *Fixed Rules*, which involves no learning and consists of applying a set of fixed local cellular-automata-like rules as they are implemented in the current version of ISAAC.

2. *Fixed strategies*, which consists of applying a fixed set of adaptive personalities and/or strategies that are found (i.e., "evolved") prior to run-time.

3. *Adaptive learning*, which will consist of real-time adaptive learning strategies as the system evolves. In this mode, ISAACAs will act according to both fixed personalities, rule sets, and strategies and according to rules and strategies that they are able to "discover" as they evolve.

Depending on what mode the user chooses to run ISAAC, the second step will consist of using a genetic algorithm to find — or "evolve" — the "best fit" ISAACAs for a given scenario. In mode 2, the genetic algorithm is asked to search for the best mix of personalities and/or strategies to use against a particular opponent, or opponent type. Once this mix is found, the personalities and strategies are clamped and one then proceeds with the actual run from which sample data can be extracted for analysis. In mode 3, the genetic algorithm is an active part of a given run, and is used as an integral dynamic component providing real-time adaptability.

In either case, the genetic algorithm is used to search the enormous range of possible attributes of an ISAACA for the "right mix" of parameters that define a desired force capability. The objective function that defines what is meant by "right mix" is defined at the user's discretion. (See, however, the last section of this paper for speculations on how ISAAC itself can be used to suggest alternative objective functions.)

In the current version of **ISAAC_GA** (i.e., the stand-alone genetic algorithm "front-end" to ISAAC's core engine), the GA is used to evolve

[32] See pages 85-94.

a pool of red personalities to perform a given mission against a *fixed* blue force. The only variability that is allowed on the blue side is its spatial disposition. (Recall that the GA automatically averages over a user-specified number of initial conditions.) Moreover, red's mission "fitness" is defined as a weighted function of certain mission "primitives" (such as "minimizing time to enemy flag", "minimizing casualties," and so on) that are defined entirely from red's perspective. Future versions will enhance this basic GA engine in several ways:

1. The GA recipe (see ***The Basic GA Recipe*** in ***Genetic Algorithm Evolutions of ISAACA Personalities***) will be enhanced to speed up both convergence and execution times. The current program is of proof-of-concept caliber only, and can be improved upon in many ways.

2. The full (i.e., multi-squad), rather than truncated, version of ISAAC will be used so that the GA can search for optimal squad-specific sizes and personalities and include both local-command and global-command-related parameters in its search space.

3. Red's current red-centric mission fitness will be generalized to include blue-primitives. Instead of defining red's mission fitness by focusing entirely on how well red performs, the user will also be given the option of including primitives defining how well (or badly) blue performs. For example, red may assume a specific mission for blue, and then include an assessment of how well blue performs that mission as a part of an assessment of its own mission. Red thus effectively will be given an ability to consider not just how well it is doing (i.e., to maximize its own fitness), but how badly blue is doing, from red's perspective (to simultaneously minimize blue's fitness).

4. The GA will be used to evolve personalities for both the entire force (as it does currently) and *individual* ISAACAs.

Future development of ISAAC's GA search capability is also likely to borrow from Hillis' coupled-GA strategy [6]. Since this strategy is discussed at the end of appendix B, we will here only outline the approach. The idea is to set up not one but *two* interacting genetic algorithm populations, one population consisting of "solutions" (or, in Hillis' original formulation, *hosts*), and the other consisting of "problems" (or, *parasites*). Having the two populations interact effectively sets up an "arms-race" between the two populations. While the hosts are trying to find better and better ways to sort the problems, the parasites are trying to make the hosts less and less adept at sorting the problems by making the problems "harder."

The interaction between the two populations dynamically alters the form of the fitness function. Just as the hosts reach the top of a fitness "hill," the parasites deform the fitness landscape so that the hill becomes a "valley" that the hosts are then forced to find ways to climb out of and start looking for new peaks. When the population of programs finally reaches a hill that the parasites cannot find a way to turn into a valley, the combined efforts of the co-evolving hosts and parasites has found a global optimum. Thus, the joint, coupled, population pools are able to find better solutions quicker than the evolutionary dynamics of populations consisting of sorting programs alone.

The application to ISAAC is conceptually straightforward. The idea is to apply genetic algorithms not to just one side of a conflict, or to use genetic algorithms to find "optimal" combat tactics for fixed sets of constraints and environments, but to use *joint, coupled, pools of populations*, one side of which represents a set of tactics or strategies to deal with specific scenarios, and the other side of which seeks ways to alter the environment in ways that make it harder and harder for those tactics or strategies to work. Thus, future versions of **ISAAC_GA** will be able to search not just in the red personality space for a fixed blue force, but in a joint *red:blue personality space* in which blue's "mission" is to make it as hard as possible for red to succeed.

What Is ISAAC Useful For?

ISAAC has been developed primarily to address the basic question: *"To what extent is land combat a self-organized emergent phenomenon?"* As such, its intended use is not as a full system-level model of combat but as an interactive toolbox (or "conceptual playground") in which to explore high-level emergent behaviors arising from various low-level (i.e., individual combatant and squad-level) "interaction rules." The idea behind ISAAC is not to model in detail a specific piece of hardware (M16 rifle, M101 105mm howitzer, *etc.*), but to provide an understanding of the fundamental behavioral tradeoffs involved among a large number of notional variables.

Because ISAAC takes a *bottom-up, synthesist* approach to the modeling of combat – vice the traditional *top-down,* or *reductionist* approach – ISAAC's conceptual focus is very different from the focus of most conventional models. For example, models based on differential equations homogenize the properties of entire populations and ignore the spatial component altogether. Partial differential equations – by introducing a physical space to account for troop movement – fare somewhat better, but still treat the agent population as a continuum. In contrast, ISAAC consists of a discrete heterogeneous set of spatially distributed individual agents (i.e., combatants), each of which has its own characteristic properties and rules of behavior. These properties can also change (i.e., adapt) as an individual agent evolves in time.

Most traditional models focus on looking for equilibrium "solutions" among some set of (pre-defined) aggregate variables. The LEs themselves are effectively *mean-field* equations (in the parlance of physics), in which certain variables such as the attrition rate are assumed to represent an entire force and the outcome of a battle is said to be "understood" when the equilibrium state has been explicitly solved for. In contrast, ISAAC focuses on understanding the kinds of emergent patterns that might arise while the overall system is *out of equilibrium.*

In ISAAC, the "final outcome" of a battle – as defined, say, by measuring the surviving force strengths – takes second stage to exploring how two forces might "co-evolve" during combat. A few examples of the profoundly *non-equilibrium* dynamics that characterizes much of real combat include: the sudden "flash of insight" of a clever commander that changes the course of a battle; the swift flanking maneuver that surprises the enemy; and the serendipitous confluence of several far-separated (and unorchestrated) events that lead to victory. These are the kinds of behavior that Lanchesterian-based models are in principle incapable of even addressing. ISAAC represents a first step toward being able to explore such questions.

ISAAC is designed to allow the user to explore the evolving patterns of macroscopic behavior that result from the collective interactions of individual agents, as well as the feedback that these patterns might have on the rules governing the individual agents' behavior. While this preliminary version of ISAAC can do no more than suggest new ways of thinking about some old issues, it is encouraging to note that, even at this early juncture, ISAAC already has an impressive repertoire of emergent behaviors:

- Forward advance
- Frontal attack
- Local clustering
- Penetration
- Retreat
- Attack posturing
- Containment
- Flanking Maneuvers
- Defensive posturing
- "Guerilla-like" assaults
- Encirclement of enemy forces
- *many more ...*

Moreover, ISAAC frequently displays behaviors that appear to involve some form of "intelligent" division of red and blue forces to deal with local "firestorms" and skirmishes, particularly those forces whose personalities have been "evolved" (via a genetic algorithm) to perform a specific mission. It must be remembered that such behaviors are not hard-wired-in but are effectively an emergent property of a decentralized and nonlinear local dynamics.

The ultimate goal is for ISAAC to become a fully developed complex systems theoretic analyst's toolbox for identifying, exploring and possibly exploiting emergent collective patterns of behavior on the battlefield.

The payoff of using ISAAC, or some other multiagent-based model of land combat, is a radically new – and decidedly non-Lanchesterian – way of looking at some fundamental issues of land warfare. Specifically, ISAAC is being designed to help analysts ...

- Understand how all of the different elements of combat fit together in an overall "combat phase space"
- Understand the out-of-equilibrium patterns of behavior vice the approach to equilibrium states stressed by most conventional models

- Identify and explore emergent collective patterns of behavior on the battlefield
- Understand the effects of information uncertainties, inaccuracies, and time-delays
- Assess the value of information: *How can I exploit what I know the enemy does not know about me?*
- Explore tradeoffs between centralized and decentralized command-and-control (C2) structures: *Are some C2 topologies more conducive to information flow and attainment of mission objectives than others? What do the emergent forms of a self-organized C2 topology look like?*
- Provide a natural arena in which to explore consequences of various qualitative characteristics of combat (unit cohesion, morale, leadership, etc.)
- Study the general efficacy of combat doctrine and tactics
- Explore emergent properties and/or other "novel" behaviors arising from low-level rules (even doctrine if it is well encoded)
- Capture universal patterns of combat behavior by focusing on a reduced set of critical drivers
- Suggest likelihood of possible outcomes as a function of initial conditions
- Provide near-real-time tactical decision aids by providing a "natural selection" (via a genetic algorithm) of tactics and/or strategies for a given combat scenario.

Furthermore, ISAAC provides a natural arena in which to explore the Clausewitzian "fog-of-war," or the effects of uncertainties and/or inaccuracies of intelligence data and of time-delays in reporting information. More important, from an *Information Warfare* perspective, ISAAC provides a framework for quantifying the "value" of information on a battlefield. ISAAC can, in principle, be used to explore the consequences of given (personality-defined) force and/or weapon mixes. It can also be used to re-examine traditional measures of combat effectiveness and define requirements for what might loosely be called *nonlinear data collection*, which refers to data that capture the continuously evolving relationships among all of the interdependent components of combat (as compared with more static measures — such as force attrition — commonly used by conventional models).

Before illustrating how ISAAC (or its future versions) can be used to explore three sample issues in land warfare, we first briefly describe what a typical "new sciences" approach really entails.

How is Work in the "New Sciences" Actually Done?

There is a popular misconception that complex systems theory is a well-defined science; that it consists of some canned set of software routines ready to be downloaded from, say, *Microsoft's* WWW site, and directly unleashed on whatever "complex problem" happens to strike one's fancy. This cannot be further from the truth. The reality is that much of what goes under the name of "complex systems theory" actually consists of a hodgepodge of on-the-fly hand-crafted and tinkered techniques and approaches that say more about the research style of a particular complex systems "theorist" than they do about the how the new sciences are practiced as a whole. There is certainly no existing complex systems theory model *per se* that can be ported over to describe land combat. The current crop of models are either specifically tailored to particular problems or are general purpose simulators (like the Santa Fe Institute's SWARM programming language) that must be carefully tuned to apply to specific systems.

In fact, most "new sciences" research is generally practiced by following these basic steps (these steps are <u>not</u> meant to be taken facetiously!):

Step 1 – Think of an interesting question to ask regarding the behavior of a real system (or find a real system to study)

Step 2 – Simplify the problem as much as possible without losing the "essence" of the system

Step 3 – Write a program to simulate the individual agents of the system, following simple rules with specified interactions

Step 4 – "Play" (i.e., interact) with the simplified models of the system

Step 5 – ***Sit back and watch for patterns***; run the program many times to build up statistics and an intuition for when and how different patterns emerge

Step 6 – Develop theories about how the real system behaves

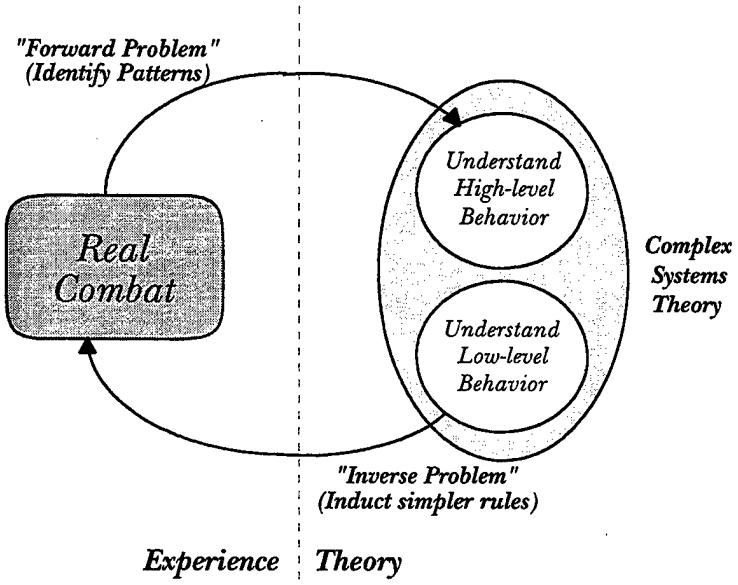
Step 7 – Tinker with the model, change prameters, identify sources of behavioral changes, simplify it even further

Step 8 – Repeat steps 4 through 7!

The critical steps – steps four through seven – are highlighted in gray. The most important step is step five: *sit back and watch for patterns!* Much of the early work with trying to understand the behavior of a system consists of finding ways to spot overall trends and patterns in the behavior of a system while continually interacting and "playing" with

"toy-models" of the system. If one is serious about applying the "new sciences" to land warfare, one must be ready to rethink some of the conventional strategies and approaches to modeling systems.

Figure 82. Interplay between experience and theory in the forward- and inverse-problems of complex systems theory



Another important element of the basic approach of complex system theory to understanding the behavior of complex systems is that the *forward*-problem and *inverse*-problem must both be studied simultaneously (see figure 82), and that the interplay between experience and theory is never overlooked.

The forward-problem consists essentially of observing either real-world behavior or the behaviors of a model of a complex system with the objective being to identify any emergent high-level behavioral patterns that the system might possess. The inverse problem deals with trying to induct a set of low-level rules that describe observed high-level behaviors. Starting with observed data, the goal here is to find something interesting to say about the properties of the source of the data. The forward problem is therefore concerned with theoretical tools that are used to *identify patterns*, while the inverse problem is concerned with tools that are used to *induct low-level rules* (or models) that generate the observed high-level behaviors.

A lengthier discussion of modeling and simulation and how it pertains to land warfare appears in [1] and [2]. Thoughtful discussions about the general use of models are given by Denning [36] and Casti [37].

Sample Issues

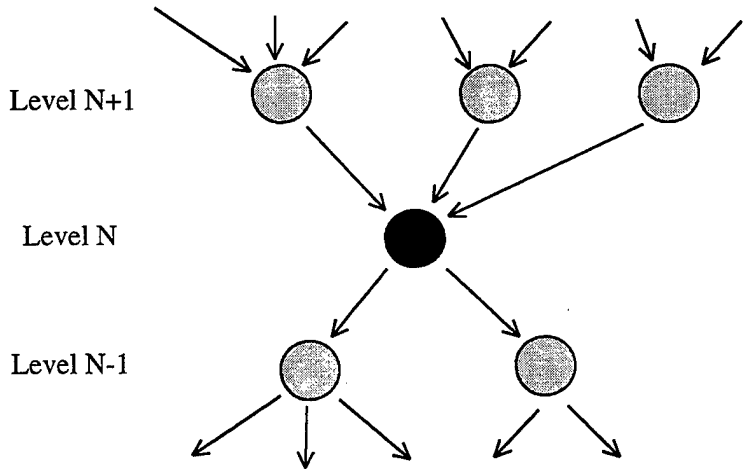
Below, I briefly discuss how a multiagent-based model such as ISAAC can help explore three sets of fundamental issues:

1. *Centralized versus decentralized command and control structures*
2. *The role of the "human element" in combat modeling*
3. *The relationship among all of the dynamical elements of combat.*

Centralized Versus Decentralized Command and Control Structures

In its simplest run-mode, ISAACian dynamics is strictly decentralized: ISAACAs do not communicate with other ISAACAs and all ISAACAs base their decisions on information that is strictly local to their sensor's field-of-view. In this mode, ISAAC represents a simple "toy-model" view of a strictly decentralized combat. ISAAC's built-in command and control structure can be used to explore the consequences of having a centralized versus decentralized C2 structure. Moreover, because of ISAAC's simple design, more probing questions regarding, for example, how the overall efficacy of a given command and control system depends on its hierarchical structure (i.e., connectivity) can also be addressed.

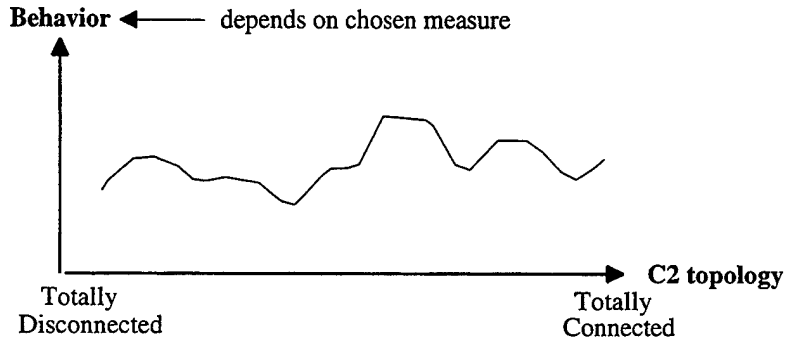
Figure 83. A schematic representation of an ISAACA C^2 structure



Recall that each ISAACA lives on a simple two-dimensional lattice, and is free (within constraints) to move in all directions up to a specified range. Each ISAACA is also "connected" (i.e., can communicate with)

other ISAACAs on an underlying graph that determines the command and control hierarchy. That is, each ISAACA on a level N of the hierarchy receives information and/or orders from ISAACAs on the next higher level (level N+1) and disseminates information and/or orders to ISAACAs on the next lower level (level N-1). Each ISAACA, on any level, can *receive* information, *send* information, and *act* on information, appropriate to all ISAACAs on the given level. ISAACAs on the lowest level (level 1) represent units that physically interact with enemy units in the lattice environment (see figure 83).

Figure 84. Behavior (arbitrary measure) as a function of C^2 structure



Imagine a topology landscape, ranging from the trivial *totally disconnected* graph on one end, to the (equally as trivial) *totally connected* graph on the other, with all possible graphs (and therefore all possible C^2 structures in between); see figure 84. The problem is to determine how the overall behavior changes (either qualitatively or quantitatively) as the landscape is systematically swept from one end to the other:

- *How does the topology affect behavior?*
- *Are some topologies more conducive to information flow and attainment of objectives than others?*

A Self-Organized C^2 Structure?

A more speculative multiagent-based approach to command and control is to see whether a decentralized variant of ISAAC possessing an ability to *evolve* (whether via a genetic algorithm or some other means) can be designed so that a command control structure *emerges on its own*!

If an ISAACA's chromosome encodes a sufficiently large volume of "possibility space" to include the evolution of local communication, it is entirely conceivable that — with the right fitness function guiding the overall ecology of ISAACAs — the system will by itself find that the most "efficient" use of the information available on the battlefield entails

effectively establishing a command and control structure. ISAAC may also be able to point out alternative C^2 structures that are more suitable for dealing with given scenarios.

The "Human Element" of Combat Modeling

One of the most significant shortcomings of conventional Lanchester-equation-based models of combat is their almost total disregard for the human (i.e., psychological and/or decision-making capability) factor. That this problem continues to plague most conventional combat models, of course, is due not to a lack of effort on the part of the modeling and simulation community, but to the fact that the problem may be fundamentally unsolvable. The problem of predicting what a given individual will do in a given situation, armed with a certain set of true or untrue facts, is already a hard enough problem. The problem of predicting what an individual will do in a given situation when that individual must base his decision not just on simple "facts" but also on what he thinks other individuals will do as a result of his impending decision (and who themselves, in turn, base their decisions on what they think others will choose to do, and so on...), is essentially an *impossible* problem to solve, at least with the mathematical tools currently available. And yet, this nested nonlinear decision-making process is what arguably drives much of the behavior on the real battlefield.

Most honest efforts to incorporate this all-important "human element" into models of combat take a more-or-less traditional Artificial Intelligence (AI) approach: they either rely on the decision-making capability of expert systems or incorporate some form of fuzzy logic into the overall decision-making process. The differences between traditional AI approaches and the multiagent-based approach that uniquely characterizes a complex systems theoretic approach to land combat modeling were described earlier in this paper (see ***Agent-Based Models*** in the ***Introduction***). What the design philosophy of ISAAC, in particular, brings to this problem is a natural context in which to describe combat as consisting of many mutually interacting elementary combatants, each reacting to local environmental stimuli and information according to a *quantifiable internal value system.* ISAAC thus offers a very natural complex systems theoretic arena in which to examine what high-level behaviors might emerge from *adaptive-personality-driven local dynamics.*

The ISAAC testbed consists essentially of a medium-sized "ecology" of elementary adaptive combatants that simultaneously act as both predator and prey. Each ISAACA acts according to a locally devised strategy that is based in part on its local perception/knowledge, in part on communicated nonlocal information, and in part on its forecast of enemy action. The resulting "combat ecology" consists of "local actions predicated on anticipated local actions predicated on..." and so on.

Now, while it would be foolish to suppose that *any* model (at least in the foreseeable future) can model exactly the dynamics of any one human, or that of a few humans working in concert, it is not hard to imagine modeling the effects of *interactions among many humans.* The viability of the whole of social science depends on this fact. And, as superbly demonstrated by Epstein's and Axtell's agent-based *Sugarscape* model of social systems [16], this supposition can go a long way indeed to "explaining" many system-wide behaviors that were heretofore believed to be "too complex" to understand. A useful analogy is the game of darts: while one cannot know in advance where an individual dart will hit a dartboard, one is reasonably well assured of attaining a Gaussian "hit pattern" distribution after throwing a few hundred darts.

ISAAC's fundamentally bottom-up, synthesist approach allows a land combat analyst to explore a variety of "unconventional" *personality*-driven questions:

- Which personalities/personality-mixes are more (or less) conducive to generating coherent (or incoherent) collective patterns of behavior?
- Given that a force must engage an enemy characterized by a given personality, which personalities are best suited for performing given missions?
- Which personalities tend to generate high (or low) entropy?
- For which regions in ISAACA's parameter-space are the emergent patterns stable (or unstable)?
- Are there regions in ISAACA's parameter-space that are sensitive to small perturbations (or chaotic), and might there be a way to exploit this in combat (as in selectively driving an opponent into these more sensitive regions of phase space)?
- What is the "optimal" platoon size for a given mission?
- *many others ...*

Combat "State- Space"

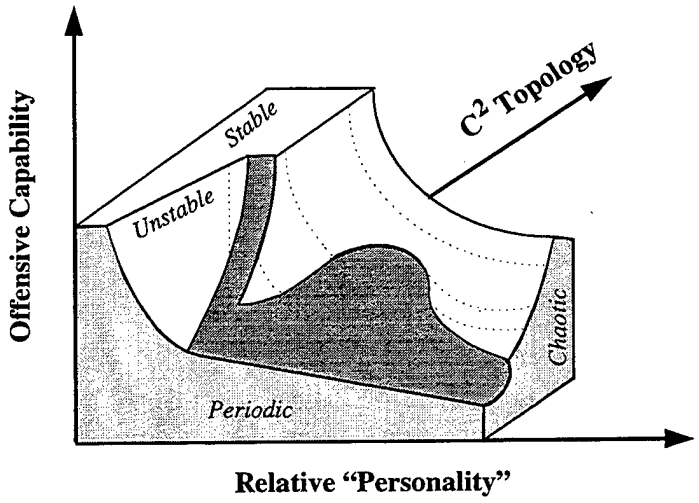
ISAAC provides a natural framework within which it is possible to talk about a combat "state-space." By this I mean a hypothetical N-dimensional space — spanned by the parameters that define each ISAACA and define the complete battlefield — within which the states of the evolving battle trace out particular trajectories.

One of the ultimate goals in designing ISAAC is to provide a basic set of tools to explore this state space. Once the behaviors in a sufficiently large volume of this state-space are mapped out — fixed point behavior, periodic states, and (as is likely, given the nonlinearity of the underlying dynamics) chaotic behavior — the range of possible questions that one can ask about the general behavior of this adaptive-agent-driven combat "ecology" becomes enormous.

One could directly ask, for example, for the kinds of "personality mixes" that are best-suited to deal with a known (or unknown) enemy. One could explore tradeoffs between personality and weapon-mix, or the value of having a "risk-taking" commander in charge of a group of ISAACAs of a given personality class. One could ask about the kinds of tactics and strategies that can be used, for a given mix of friendly and enemy force types, to coerce the combat trajectory to swing more toward a region of state-space that is more conducive for a "win," or toward a region within which the enemy's state generally becomes less stable.

In short, the act of mapping out the combat state-space represents a step toward obtaining a greater intuitive sense about how *all* of the different elements of combat fit together. The stand-alone parameter-space mapper program, **ISAAC_PM**, provides a glimpse of two-dimensional "slices" of this larger phase space (see ***Taking 2D "Slices" of ISAAC's Parameter Space***).

Figure 85. Schematic representation of the combat phase space



In the prologue to this report, artificial life is introduced as an attempt to understand *life-as-we-know-it* by exploring a larger context of *life-as-it-could-be*. ISAAC is then introduced as a tentative first step toward furthering our current understanding of the fundamental principles of

land warfare by providing an exploratory vehicle for examining a larger context of *combat-as-it-could-be.* Among the important questions to ask in this context is, *"What regions in ISAAC's phase-space are Lanchesterian-like?"*

Miscellaneous Issues and Questions

We conclude this report by discussing a few miscellaneous issues:

- *Selfishness vs. Atruism*
- *Self-Organized Criticality in Combat*
- *Scaling Problem*
- *Self-Organized Information?*

Selfishness vs Altruism

A fundamental problem in natural evolution and the evolution of social cultures is to understand the relationship between selfishness and altruism. On an individual level, the problem is to understand what contexts drive individuals to act selfishly vice cooperatively. On a more global level, there is the question of whether local selfishness - that is, a set of purely selfishly motivated individual acts designed to maximize an immediate payoff - can induce globally cooperative behaviors.

Axelrod and Hamilton [38], among others, has studied this problem by applying game theory to the idealized *Prisoner's Dilemma* problem. In the prisoner's dilemma problem, the fitness measure (or the "payoff matrix" in the parlance of game-theory) is such that individuals maximize their fitness collectively in the long term by *cooperatively ignoring* the behavior that would maximize their fitness in the short term. If the game is played once, individuals do best by not cooperating (i.e., defecting). If the game is played repeatedly, however, individuals do best by cooperating with one another. Axelrod and Hamilton have been able to show that, in the context of Darwinian evolution, locally selfish behavior can lead to global cooperation.

While this issue is philosophically intriguing in the abstract, it takes on an added dimension in the context of military command and control. Each level of a command and control hierarchy consists of entities that are driven by locally selfish goals. On the lowest (i.e., individual combatant) level such a purely "selfish" goal might simply be to stay alive. In ISAAC, each ISAACA selfishly chooses the best move, where "best" is interpreted purely locally. On a local command level, each local commander selfishly wants to issue the best local movement vectors to its subordinates. The global commander simply wants to "win the battle." Entities on the various levels must not only assimilate and

react to information that is local to them, but must do so on time scales that are appropriate for their level. Yet, if the command and control structure is to be a viable one, all of these individually and locally selfish acts must successfully combine to achieve a collective goal (or mission). The *system* (i.e., the entire combat force) must therefore carefully weigh short-term success (both perceived and real) and long-term strategy and fitness.

Such a view of command and control immediately leads to some interesting issues. For example, it is not immediately obvious that a combat force will perform its mission well when all of its constituent combatants are purely selfishly driven. Basic questions include ...

- *When is it appropriate (from a purely dynamical systems point of view) to sacrifice local fitness (i.e., to accept less-than-optimal local fitness) in order to achieve a greater (i.e., higher level) measure of mission success?*
- *What are the effects of dissent in command; that is, how is the overall mission affected by having one local commander "at odds" with another or with his global commander?*
- *What are the general effects of homogeneity (and heterogeneity) of ISAACA personalities?*

Kauffman's Patches

In this context, one could also explore the applicability of Kauffman's *patch-optimization* procedure [39]. The idea is to attack an optimization problem (that consists of many interacting parts) by dividing it into a quilt of nonoverlapping "patches" and optimizing "selfishly" within each patch. Kauffman's work suggests that while patches may find local "solutions" that are harmful to the whole, the overall process nonetheless often succeeds in finding solutions that are "good" for the entire system. Because command and control obviously involves overlapping (vice Kauffman's *non*-overlapping) patches, it is clear that this *patch-optimization* procedure – if it applies at all – must be somehow amended. However, because the basic approach resonates so strongly with the "locally selfish"-driven dynamics that underlies much of command and control, it is certainly worth exploring.

Self-Organized Criticality in Combat

Recall that *self-organized criticality* (SOC) is the idea that dynamical systems with many degrees of freedom naturally self-organize into a critical state in which the same events that brought that critical state into being can occur in all sizes, with the sizes being distributed according to a power-law.[33]

[33] See pages 101-107 of [1].

"Criticality" here refers to a concept borrowed from thermodynamics. Thermodynamic systems generally get more ordered as the temperature is lowered, with more and more structure emerging as cohesion wins over thermal motion. Thermodynamic systems can exist in a variety of phases – gas, liquid, solid, crystal, plasma, etc. – and are said to be critical if poised at a phase transition. Many phase transitions have a critical point associated with them, that separates one or more phases. As a thermodynamic system approaches a critical point, large structural fluctuations appear despite the fact the system is driven only by local interactions. The disappearance of a characteristic length scale in a system at its critical point, induced by these structural fluctuations, is a characteristic feature of thermodynamic critical phenomena and is universal in the sense that it is independent of the details of the system's dynamics.

The kinds of structures SOC seeks to describe the underlying mechanisms for look like equilibrium systems near critical points but are not near equilibrium; instead, they continue interacting with their environment, "tuning themselves" to a point at which critical-like behavior appears. In contrast, thermodynamic phase transitions usually take place under conditions of thermal equilibrium, where an external control parameter such as temperature is used to tune the system into a critical state.

Introduced in 1988, SOC is arguably the *only* existing holistic mathematical theory of self-organization in complex systems, describing the behavior of many real systems in physics, biology and economics. It is also a universal theory in that it predicts that the global properties of complex systems are independent of the microscopic details of their structure, and is therefore consistent with the "the whole is greater than the sum of its parts" approach to complex systems. Put in the simplest possible terms, SOC asserts that *complexity is criticality*. That is to say, that SOC is nature's way of driving everything towards a state of maximum complexity.

In general, SOC appears to be prevalent in systems that have the following properties:

- they have many degrees of freedom
- their parts undergo strong local interactions
- the number of parts is usually conserved
- they are driven by being slowly supplied with "energy" from an exogenous source
- energy is rapidly dissipated within the system

In systems that have these properties, SOC itself is characterized by

- a self-organized drive towards the critical state
- intermittently triggered ("avalanche"-style) release of energy in the critical state
- sensitivity to initial conditions (i.e., the trigger can be very small)[34]
- the critical state is maintained without any external "tuning"

The critical state is an attractor for the dynamics: systems are inexorably driven toward it for a wide variety of initial conditions. Frequently cited examples of SOC include the distribution of earthquake sizes, the magnitude of river flooding, and the distribution of solar flare x-ray bursts, among others. Conway's Life-game CA-rule (see ***Appendix A***), which is a crude model of social interaction, appears to self-organize to a critical state when driven by random mutations. Another vivid example of SOC is the extinction of species in natural ecologies. In the critical state, individual species interact to form a coherent whole, poised in a state far out of equilibrium. Even the smallest disturbances in the ecology can thus cause species to become extinct. Real data show that there are typically many small extinction events and few large ones, though the relationship does not quite follow the same linear power-law as it does for avalanches.

Is war, as suggested by Bak and Chen [40], perhaps a self-organized critical system? A simple way to test for self-organized criticality is to look for the appearance of any characteristic power-law distributions in a system's properties. Richardson [41] and Dockery and Woodcock [35] have examined historical land combat attrition data and have both reported the characteristic linear power-law scaling expected of self-organized critical systems. Richardson examined the relationship between the frequency of "deadly quarrels" versus fatalities per deadly quarrel using data from wars ranging from 1820 to 1945. Dockery and Woodcock used casualty data for military operations on the western front after Normandy in World War II and found that the log of the number of battles with casualties greater than a given number C also scales linearly with log(C).

---

[34] Sensitivity to initial conditions is usually a trademark of chaos in dynamical systems. Unlike fully chaotic systems, however, in which nearby trajectories diverge exponentially, the distance between two trajectories in systems undergoing SOC grows at a much slower (power-law) rate. Systems undergoing SOC are therefore only "weakly chaotic." There is an important difference between fully developed chaos and weak chaos: fully developed chaotic systems have a characteristic time scale beyond which it is impossible to make predictions about their behavior; no such time scale exists for weakly chaotic systems, so that long-time predictions may be possible.

ISAAC's data collection ability – particularly its cluster-counting routines (see ***Data Collection***) – can be used to search for evidence SOC-like behavior, and for the combination of parameters (if any) that it appears in ISAAC's N-dimensional parameter space.

Scaling Problem

As has been emphasized repeatedly throughout this report, ISAAC is designed to be nothing more than a "conceptual playground" in which to explore certain fundamental issues of land warfare; or a tool by which to take a baby step beyond simple metaphor in discussions concerning the applicability of complex systems theory to land warfare. ISAAC is certainly not intended to be used as a system-level model of real combat. Nonetheless, there is a question that needs to be raised about how ISAAC might - or might not - *scale with force size.* For example, it is not immediately obvious that whatever patterns of behavior one observes by running ISAAC using, say, 100 ISAACAs per side, to represent some small-scale "conflict," necessarily scales to display the same patterns (or warrant the same conclusions to be drawn from) running ISAAC using, say, 1000 ISAACAs per side.

Self-Organized Information?

The statements "I understand this system" or "I understand how this system behaves," are, unfortunately, very commonly misunderstood to be synonymous with "I can model this system."

Here is (only a slightly exaggerated) form of the conventional wisdom's "party-line:" *Once I have intelligently put together a model or simulation of a system, the "problem" of understanding how that system behaves is effectively "solved." While I must still, of course, observe the behavior of the model, most of the dirty work has already been accomplished. The true solution lies in modeling; everything else is mop-up work!*

In fact, this kind of reasoning is dangerously false. Suppose a physical system's behavior is complicated enough to warrant the development of a model (or models) in order for me to try to understand it. I will either come up with a model that captures none of the behavior of the real system — in which case, I have failed, and must start over — or I will succeed in reproducing some (or most) of the real system's complicated dynamics. In the second case, my model will have attained a degree of "realism" that convinces me that it can be used as a surrogate system to study the behavior of the real system. While one can argue that the act of capturing the essence of the real system in a model deepens one's understanding of that system purely as a result of having effectively simplified the definition of the system, this act — by itself — does not necessarily lead to better understanding. If the model's behavior is as "complicated" as that of the system it purportedly captures the essence

of, one is still faced with the problem of how to understand the model's behavior.

In [2][35], a computer model of natural evolution called *Tierra* was cited as an example of a case where a model captures a real system's behavior so well that the task of understanding its own behavior is far from trivial. In *Tierra*'s case, the unraveling of basic evolutionary phenomena such as a rich diversity of species, symbiosis, parasitism, para-parasitism, and so on, comes at a substantial price of having to do a considerable amount of "field work" with the computer model and its outputs just to understand what is "really happening" on a given run.

The point of this short discussion is to remind the reader that the "problem" ISAAC is designed to address (if not by its current, skeletal, version then by its future, more complete and mature, form) actually consists of *two* separate issues:

- **Issue #1: To Find a Proper *Complex Systems Theoretic* Model Testbed.** This issue involves the actual design of a multiagent-based "toy-combat" ecology, which involves all kinds of questions regarding adaptability, communication, evolution, combat and so on. For example, how does an individual ISAAC determine its strategy? What are the appropriate genomes? How much memory is needed? How is a strategy defined? How do the actions of low-level combatants differ from higher-level ones? ... This first issue is "solved" when one provides an answer to the question, *"Has ISAAC captured the critical drivers that determine the patterns of behavior of real combat?"* This report discusses the first tentative steps that have been taken toward addressing this question.
- **Issue #2: To Provide an Analyst's Toolbox for Exploring Emergent Patterns of Behavior.** The second issue involves the understanding of what is actually going on within the toy-battlefield once ISAAC begins evolving. The question here is, "Now that (a "mature" version of) ISAAC is up and running — orders are being sent down echelon via a realistic C2 structure, combatants adapt and react according to appropriate local tactics and both short- and long-term strategies, and so on — *how do we make sense of what ISAAC is really doing?*" As indicated in the discussion above, this important question can be asked of *any* dynamical system, whether it is real or simulated. Apart from applying the arsenal of tools and mathematical descriptions used by nonlinear dynamics and complex systems theory to understand behaviors of complex systems (see [1] and [2]), the

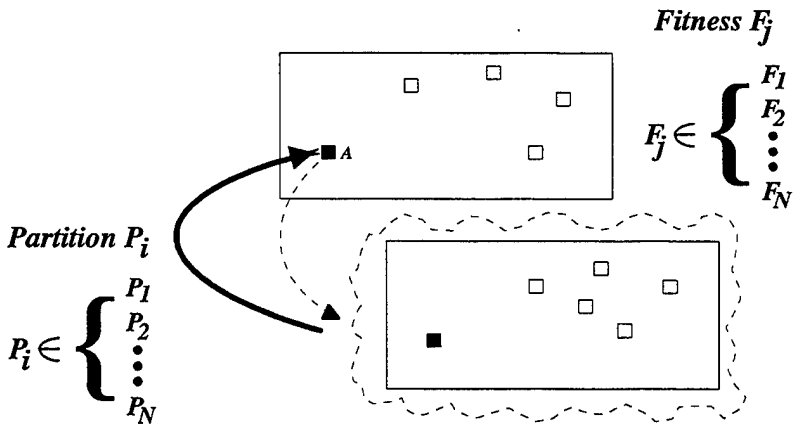
[35] See page 35.

speculative suggestion made here involves using ISAAC itself to address the problem (see discussion below).

I have already discussed using genetic algorithms as the engine driving both individual learning and long-term tactics and strategy acquisition. A third — and conceptually most far-reaching — use of genetic algorithms involves using them to search through the (enormously high-dimensional) space of *all possible ways of filtering, assimilating, and exploiting information to make command decisions.*

Consider a future version of ISAAC whose dynamics include an embedded command and control structure in which local and global commanders base their decisions (in part) on a nested layering of combat scenarios (see ***Nested ISAAC Dynamics***). What is implicit in such a model is that the local and global commanders have each found a way (or a way has been defined for them) to characterize whatever information is relevant for their command decision. Now, on the one hand, it is entirely reasonable to "hard-wire" in hypothetical characterizations, using the knowledge and experience of real commanders. This is, in fact, what is most commonly done in conventional AI-based warfare models. On the other hand, if the tools and potential power of complex systems theory are to be used to their fullest, ISAAC gives us a unique opportunity to inquire about what the relevant bits of information really are.

Figure 86. A schematic representation of how genetic algorithms may be used to find the "best" partitioning of the combat information-space



The idea is use a genetic algorithm (or some other heuristic search tool) to explore all possible ways in which a local commander can characterize and use the information describing (what he believes is) the overall state of the battle. By a characterization, I mean literally any well-defined compression of whatever information goes into defining

the overall state of the system. Conventional measures include unit formations, order-of-battle, distance to goal, friendly and enemy attrition rates, and so on. Less conventional measures might include a combat entropy, or be defined along the lines suggested by relativistic information theory (see pages 75-79 in [2]), etc.

Call any such effective compression of relevant battlefield information a *partitioning* (= P) of that combat state-space. Call the degree to which an objective (or set of objectives) that drives a commander's decision-making process has been attained the *combat fitness function* (= F), so that a higher fitness equates to being "closer to" the ultimate objective. Conceptually, a commander's task may be described as the problem of identifying, assimilating and exploiting the most appropriate P for attaining a given F. Figure 86 shows a schematic of this basic idea.

For example, a local commander might "discover" the fact that one of the most important pieces of information describing the state of the overall battle — and which therefore plays a significant role in his decision-making process — is information regarding the *distribution of adaptability* on the battlefield. That is to say, information that does not describe the state of the battle, *per se* (i.e., what forces are engaging the enemy where), but the manner in which his forces are adapting to local conditions. A local commander can then build on his experiential knowledge of such *meta*-patterns by associating local combat conditions with the right mix of adaptive ISAACAs to deal with those conditions. Some conditions may warrant a force consisting of rather fixed, rigid ISAACAs that are "optimized" for a particular kind of skirmish but are otherwise relatively inflexible and nonadaptive. Other conditions might require a more robust adaptive force, that can quickly assimilate changes to their local environment and modify their response accordingly. A genetic-algorithm-based approach to filtering all possible forms of information may help the local (and global) commander to identify and map the right local conditions to appropriate blends of force personalities and types

Epilogue: On the use of simulations .....

"Although it is true that, in the 300 years since Newton, most of theoretical science has been done using the rigorous, analytical approach, the reason for that is simply that that is the only kind of science *could* be done ... The lack of computational power meant that researchers could only answer questions that had clean, elegant solutions ... It is only now that we have the ability to do complex calculations and simulations that we are discovering that a great many systems seem to have an inherent complexity that cannot be simplified ... After another 300 years, we will no doubt feel as comfortable using computer simulations to analyze nature as scientists today feel using Newton's laws of motion to describe the trajectory of a falling stone."

– Glenn W. Rowe, *Theoretical Models in Biology: The Origin of Life, the Immune System, and the Brain* (Clarendon Press)

Appendix A: A Brief Primer on Cellular Automata

Cellular automata (CA) are a class of spatially and temporally discrete, deterministic mathematical systems characterized by local interaction and an inherently parallel form of evolution. First introduced by von Neumann in the early 1950s to act as simple models of biological self-reproduction, CA are prototypical models for complex systems and processes consisting of a large number of identical, simple, locally interacting components. The study of these systems has generated great interest over the years because of their ability to generate a rich spectrum of very complex patterns of behavior out of sets of relatively simple underlying rules. Moreover, they appear to capture many essential features of complex self-organizing cooperative behavior observed in real systems.

Although much of the theoretical work with CA has been confined to mathematics and computer science, there have been numerous applications to physics, biology, chemistry, biochemistry, and geology, among other disciplines. Some specific examples of phenomena that have been modeled by CA include fluid and chemical turbulence, plant growth and the dendritic growth of crystals, ecological theory, DNA evolution, the propagation of infectious diseases, urban social dynamics, forest fires, and patterns of electrical activity in neural networks. CA have also been used as discrete versions of partial differential equations in one or more spatial variables.

The best sources of information on CA are conference proceedings and collections of papers, such as the one's edited by Boccara [42], Gutowitz [43], and Wolfram [44,45]. An excellent review of how CA can be used to model physical systems is given by Toffoli and Margolus [46].

While there is an enormous variety of particular CA models – each carefully tailored to fit the requirements of a specific system – most CA models usually possesses these five generic characteristics:

- *discrete lattice of cells:* the system substrate consists of a one-, two- or three-dimensional lattice of cells
- *homogeneity:* all cells are equivalent
- *discrete states:* each cell takes on one of a finite number of possible discrete states
- *local interactions:* each cell interacts only with cells that are in its local neighborhood

- *discrete dynamics:* at each discrete unit time, each cell updates its current state according to a transition rule taking into account the states of cells in its neighborhood

Example #1: One-dimensional CA

For a one-dimensional CA, the value of the ith cell at time t – denoted by $c_i(t)$ – evolves in time according to a "rule" F that is a function of $c_i(t)$ and other cells that are within a range r (on the left and right) of $c_i(t)$:

$$c_i(t) = F[c_{i-r}(t-1), c_{i-r+1}(t-1), ..., c_{i+r-1}(t-1), c_{i+r}(t-1)].$$

Since each cell takes on one of k possible values – that is, $c_i \in \{0, 1, ..., k-1\}$– the rule F is completely defined by specifying the value assigned to each of the k^{2r+1} possible (2r+1)-tuple configurations for a given range-r neighborhood:

$c_{i-r}(t-1)$	···	$c_i(t-1)$	···	$c_{i+r}(t-1)$	$c_i(t)$
0		0		0	F(0,0,...,0)
0		0		1	F(0,0,...,1)
⋮		⋮		⋮	⋮
k		k		k	F(k,k,...,k)

Since F itself assigns any of k values to each of the k^{2r+1} possible (2r+1)-tuples, the total number of possible rules is an exponentially increasing function of both k and r. For the simplest case of nearest neighbors (range r=1) and k=2 (c_i = 0 or 1), for example, there are 2^8=256 possible rules. Increasing the number of values each cell can take on to k=3 (but keeping the radius at r=1) increases the rule-space size to $3^{3^3} \approx 7 \bullet 10^1$.

Figure 87 shows the time evolution of a nearest-neighbor (radius r=1) rule where c is equal to either 0 or 1. The row of eight boxes at the top of the figure shows the explicit rule-set, where – for visual clarity – a box has been arbitrarily colored "black" if the value c=1 and "white" if c=0. For each combination of three adjacent cells in generation 0, the rule F assigns a particular value to the next-generation center cell of the triplet. Beginning from an initial state (at time=0) consisting of the value zero everywhere except the center site, that is assigned the value 1, F is applied synchronously at each successive time step to each cell of the lattice. Each generation is represented by a row of cells and time is oriented downwards. The first image shows a blowup of the first five generations of the evolution. The second shows 300 generations. The figure illustrates the fact that simple rules can generate considerable complexity.

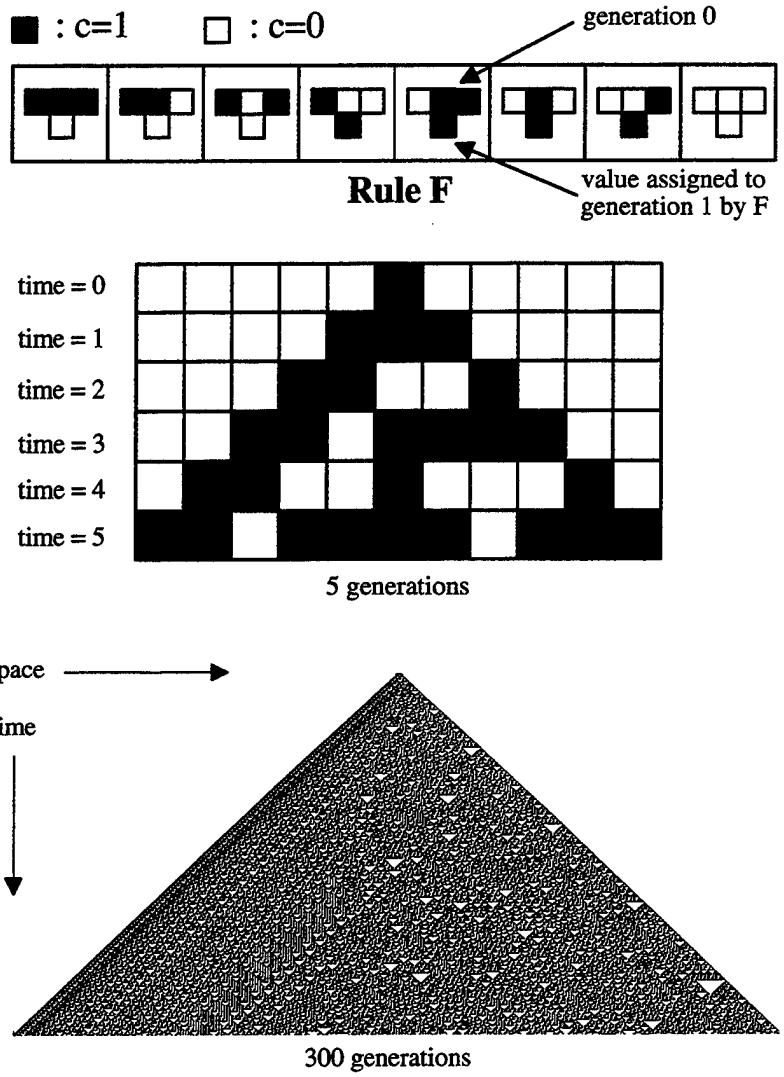
The space-time pattern generated from a single nonzero cell by this particular rule has a number of interesting properties. For example, it consists of a curious mixture of ordered behavior along the left-hand-side and what appears to be disordered behavior along the right-hand-side, separated by a corrugated boundary moving towards the left at a "speed" of about 1/4 cells per "clock" tick. In fact, it can be shown that, despite starting from an obviously non-random initial state and evolving according to a fixed deterministic rule, the temporal sequence of vertical values is completely random. Systems having the ability to deterministically generate randomness from non-random input are called *autoplectic systems.*

In general, the behavior of CA is strongly reminiscent of the kinds of behavior observed in continuum dynamical systems, with simple rules yielding steady-state behaviors consisting of fixed points or limit cycles, and complex rules giving rise to behaviors that are analogous to deterministic chaos. In fact, there is extensive empirical evidence suggesting that patterns generated by all (one-dimensional) CA evolving from disordered initial states fall into one of only four basic behavioral classes:

- *Class 1:* evolution leads to a homogenous state, in which all cells eventually attain the same value
- *Class 2:* evolution leads to either simple stable states or periodic and separated structures
- *Class 3:* evolution leads to chaotic nonperiodic patterns
- *Class 4:* evolution leads to complex, localized propagating structures

All CA within a given class yield qualitatively similar behavior. While the behaviors of rules belonging to the first three rule classes bear a strong resemblance to those observed in continuous systems – the homogenous states of class 1 rules, for example, are analogous to fixed-point attracting states in continuous systems, the asymptotically periodic states of class 2 rules are analogous to continuous limit cycles and the chaotic states of class 3 rules are analogous to strange attractors – the more complicated localized structures emerging from class 4 rules do not appear to have any obvious continuous analogues (although such structures are well characterized as being soliton-like in their appearance).

Figure 87. Example of a one-dimensional CA



Example #2: Conway's Life

"Its probable, given a large enough Life space, initially in a random state, that after a long time, intelligent self-reproducing animals will emerge and populate some parts of the space." –John H. Conway

Perhaps the most widely known CA is the game of Life, invented by John H. Conway, and popularized extensively by Martin Gardner in his "Mathematical Games" department in Scientific American in the early 1970s.

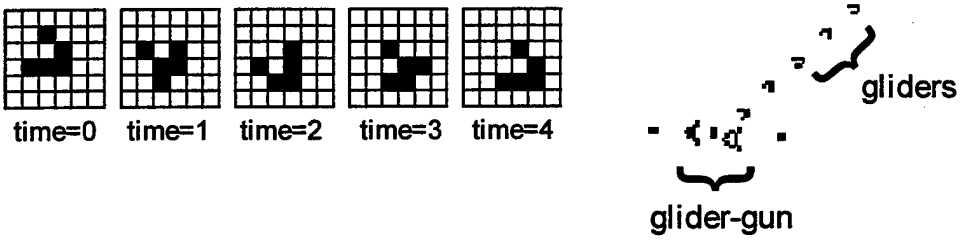
Life is "played" using the 9 nearest neighboring sites of any site on a two-dimensional lattice, and consists of (1) seeding a lattice with some

pattern of "live" and "dead" cells, and (2) simultaneously (and repeatedly) applying the following three rules to each cell of the lattice at discrete time steps:

- *Birth:* replace a previously dead cell with a live one if exactly 3 of its neighbors are alive
- *Death:* replace a previously live cell with a dead one if either (1) the living cell has no more than one live neighbor (i.e., it dies of isolation), or (2) the living cell has more than three neighbors (i.e., it dies of overcrowding)
- *Survival:* retain living cells if they have either 2 or 3 neighbors

One of the most intriguing patterns in Life is an oscillatory propagating pattern known as the "glider." Shown on the left-hand-side of figure 88, it consists of 5 "live" cells and reproduces itself in a diagonally displaced position once every four iterations. When the states of Life are projected onto a screen in quick succession by a fast computer, the glider gives the appearance of "walking" across the screen. The propagation of this pseudo-stable structure can also be seen as a self-organized emergent property of the system. The right-hand-side of figure 88 shows a still-frame in the evolution of a pattern known as a "glider-gun," which shoots-out a glider once every 30 iteration steps.

Figure 88. Glider patterns in Conway's Life



What is remarkable about this very simple appearing rule is that one can show that it is capable of universal computation. This means that with a proper selection of initial conditions (i.e., the initial distribution of "live" and "dead" cells), Life can be turned into a general purpose computer. This fact fundamentally limits the overall predictability of Life's behavior.

The well known Halting Theorem, for example, asserts that there cannot exist a general algorithm for predicting when a computer will halt its execution of a given program [47]. Given that Life is a universal computer – so that the Halting Theorem applies – this means that one cannot, in general, predict whether a particular starting configuration

of live and dead cells will eventually die out. No shortcut is possible, even in principle. The best one can do is to sit back and patiently await Life's own final outcome.

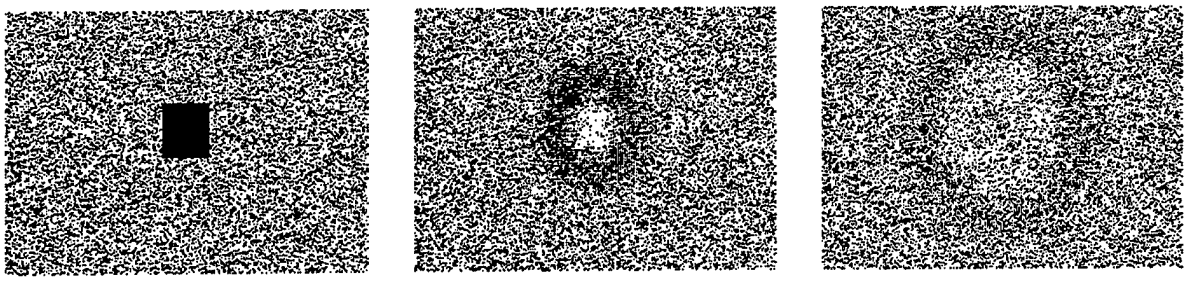
Put another way, this means that if you want to predict Life's long-term behavior with another "model" or by using, say, a partial differential equation, you are doomed to fail from the outset because its long-term behavior is effectively unpredictable. Life – like all computationally universal systems – defines the most efficient simulation of its own behavior.

Example #3: Lattice Gases

Lattice gases are micro-level rule-based simulations of macro-level fluid behavior. Lattice-gas models provide a powerful new tool in modeling real fluid behavior. The idea is to reproduce the desired macroscopic behavior of a fluid by modeling the underlying microscopic dynamics.

It can be shown that three basic ingredients are required to achieve an emergence of a suitable macrodynamics out of a discrete microscopic substrate: (1) local thermodynamic equilibrium, (2) conservation laws, and (3) a "scale separation" between the levels at which the microscopic dynamics takes place (among kinetic variables living on a micro-lattice) and the collective motion itself appears (defined by hydrodynamical variable on a macro-lattice). Another critical feature is the symmetry of the underlying lattice.

Figure 89. Two-dimensional lattice-gas simulation of a fluid



While there are many variants of the basic model, one can show that there is a well-defined minimal set of rules that define a lattice-gas system whose macroscopic behavior reproduces that predicted by the Navier-Stokes equations[36] *exactly*. In other words, there is critical "threshold" of rule size and type that must be met before the continuum fluid behavior is matched, and once that threshold is

[36] The Navier-Stokes equations are a set of analytically intractable coupled nonlinear partial differential equations describing fluid flow.

reached the efficacy of the rule-set is no longer appreciably altered by additional rules respecting the required conservation laws and symmetries.

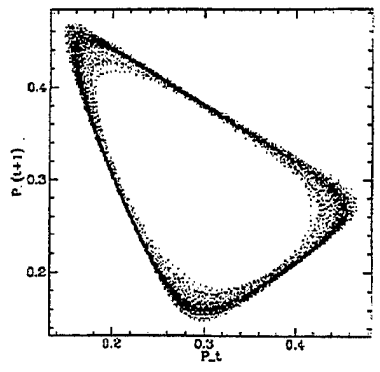
Figure 89 shows a few snapshots of the evolution of a two-dimensional lattice gas starting from an initial condition in which there is a tightly packed region of particles at the center of the lattice. Notice how this central region expands rapidly outward, and is very reminiscent of the effect a dropped stone has on an initially stagnant pool of water. The most striking feature is the circular sound wave, which is circular despite the fact that the microscopic dynamics takes place on a square lattice. The lattice gas "rules" thus force a symmetry that is not present in the microscopic dynamics to emerge on the macro-scale.

Example #4: Collective Behavior in Higher Dimensions

Chate and Manneville[37] have examined a wide variety of cellular automata that live in dimensions four, five and higher. They found many interesting rules that while being essentially featureless *locally*, nonetheless show a remarkably ordered *global* behavior.

Figure 90, for example, plots the probability that a cell has value 1 at time t+1 – labeled P_{t+1} – versus the probability that a cell had value 1 at time t – labeled P_t – four a particular four dimensional cellular automaton rule. The rule itself is unimportant, as there are many rules that display essentially the same kind of behavior. The point is that while the behavior of this rule is locally featureless – its space-time diagram would look like static on a television screen – the global density of cells with value 1 jumps around in quasi-periodic fashion. We emphasize that this quasi-periodicity is a global property of the system, and that no evidence for this kind of behavior is apparent in the local dynamics.

Figure 90. Collective behavior of a four dimensional CA



[37] H. Chate and P. Manneville, *Europhysc Letters*, Volume 14, 1991, 409.

Appendix B: A Brief Primer on Genetic Algorithms

Genetic algorithms (GAs) are a class of heuristic search methods and computational models of adaptation and evolution based on natural selection.

In nature, the search for beneficial adaptations to a continually changing environment (i.e., evolution) is fostered by the cumulative evolutionary knowledge that each species possesses of its forebears. This knowledge, which is encoded in the chromosomes of each member of a species, is passed from one generation to the next by a mating process in which the chromosomes of "parents" produce "offspring" chromosomes.

GAs mimic and exploit the genetic dynamics underlying natural evolution to search for optimal solutions of general combinatorial optimization problems. They have been applied to the Traveling Salesman Problem, VLSI circuit layout, gas pipeline control, the parametric design of aircraft, neural net architecture, models of international security, and strategy formulation.

While their modern form is derived mainly from John Holland's work in the 1960s [24], many key ideas such as using "selection of the fittest" like population-based selection schemes and using binary strings as computational analogs of biological chromosomes, actually date back to the late 1950s. More recent work is discussed by Goldberg [25], Davis [26] and Michalewicz [27] and in conference proceedings edited by Forrest [28]. A comprehensive review of the current state-of-the-art in genetic algorithms is given by Mitchell [29].

The basic idea behind GAs is very simple. Given a "problem" – which can be as well-defined as maximizing a function over some specified interval or as seemingly ill-defined and open-ended as evolution itself, where there is no a-priori discernible or fixed function to either maximize or minimize – GAs provide a mechanism by which the solution space to that problem is searched for "good solutions." Possible solutions are encoded as chromosomes (or, sometimes, as sets of chromosomes), and the GA evolves one population of chromosomes into another according to their fitness by using some combination (and/ or variation) of the genetic operators of crossover and mutation. A solution search space together with a fitness function is called a fitness landscape. Eventually, after many generations, the population will, in theory, be composed only of those chromosomes whose fitness values are clustered around the global maximum of the fitness landscape.

Genetic Operators

Each chromosome is usually defined to be a bit-string, where each bit position (or "locus") takes on one of two possible values (or "alleles"), and can be imagined as representing a single point in the "solution space." The fitness of a chromosome effectively measures how "good" a solution that chromosome represents to the given problem. Aside from its intentional biological roots and flavoring, GAs can be thought of as parallel equivalents of more conventional serial optimization techniques: rather than testing one possible solution after another, or moving from point to point in the solution phase-space, GAs move from entire populations of points to new populations.

Figure 91. Schematic representation of the basic GA operators

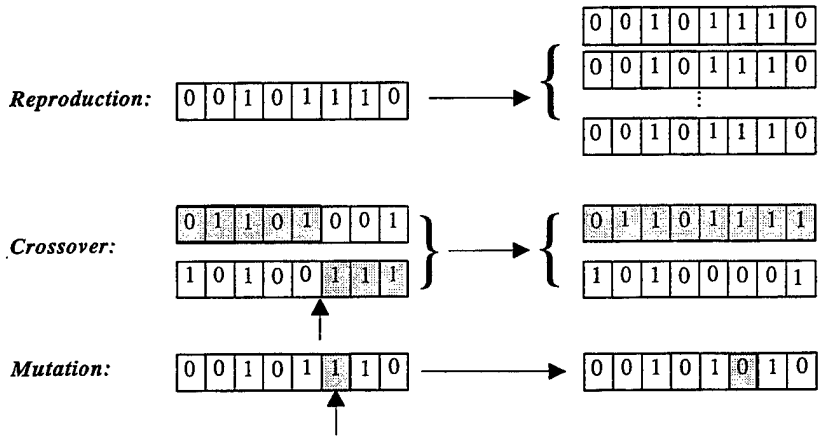


Figure 91 shows examples of the three basic genetic operations of *reproduction*, *crossover* and *mutation*, as applied to a population of 8-bit chromosomes. Reproduction makes a set of identical copies of a given chromosome, where the number of copies depends on the chromosome's fitness. The crossover operator exchanges subparts of two chromosomes, where the position of the crossover is randomly selected, and is thus a crude facsimile of biological sexual recombination between two single-chromosome organisms. The mutation operator randomly flips one or more bits in the chromosome, where the bit positions are randomly chosen. The mutation rate is usually chosen to be small.

While reproduction generally rewards high fitness, and crossover generates new chromosomes whose parts, at least, come from chromosomes with relatively high fitness (this does not guarantee, of course, that the crossover-formed chromosomes will also have high fitness; see below), mutation seems necessary to prevent the loss of diversity at a given bit-position. For example, were it not for mutation, a

population might evolve to a state where the first bit-position of each chromosome contains the value 1, with there being no chance of reproduction or crossover ever replacing it with a 0.

The Basic GA Recipe

Although GAs, like CA, come in many different flavors, and are usually fine-tuned in some way to reflect the nuances of a particular problem, they are all more or less variations of the following basic steps:

- *Step 1:* begin with a randomly generated population of chromosome-encoded "solutions" to a given problem
- *Step 2:* calculate the fitness of each chromosome, where fitness is a measure of how well a member of the population performs at solving the problem
- *Step 3:* retain only the fittest members and discard the least fit members
- *Step 4:* generate a new population of chromosomes from the remaining members of the old population by applying the operations reproduction, crossover, and mutation (see figure 91)
- *Step 5:* calculate the fitness of these new members of the population, retain the fittest, discard the least fit, and re-iterate the process

Example: Function Maximization

As a concrete example, suppose our problem is to maximize the fitness function $f(x) = x^2$, using six 6-bit chromosomes of the form $C=(c_1,c_2,...,c_6)$, where each c_i is equal to either 0 or 1. C's fitness, f(C), is determined by first converting its binary representation into a base-10 equivalent value and squaring: $f(C)=(c_1+2c_2+2^2c_3+2^3c_4+2^4c_5++2^5c_6)^2$.

The first step is to construct six random bit-strings representing the initial population:

$C_1 = (101101)$	$C_2 = (010110)$	$C_3 = (111001)$
$C_4 = (101011)$	$C_5 = (010001)$	$C_6 = (011101)$

These chromosomes have fitness values of 2025, 484, 3249, 1849, 289 and 841, respectively. The average fitness is 1456. By luck of the fitness-scaled draw, where the number of copies of a given chromosome

is determined according to its fitness, scaled by the average fitness of the entire population, three copies of C_3 are made for the next population (owing to its relatively high fitness), one copy each for chromosomes C_1, C_4 and C_6 and none for the remaining chromosomes. These copies form the mating population.

Next, we randomly pair up the new chromosomes, and perform the genetic crossover operation at randomly selected bit-positions – chromosomes $C_1$and C_4 exchange their last three bits, C_2 and C_6 exchange their last four bits, and C_3 and C_5 exchange their last bit:

C_1 exchange with C_4 at bit 3:	(101.101) x (111.001)	→	(101001)
C_2 exchange with C_6 at bit 2:	(11.1001) x (01.1101)	→	(111101)
C_3 exchange with C_5 at bit 5:	(11100.1) x (10101.1)	→	(111001)
C_4 exchange with C_1 at bit 3:	(111.001) x (101.101)	→	(111101)
C_5 exchange with C_3 at bit 5:	(10101.1) x (11100.1)	→	(101011)
C_6 exchange with C_2 at bit 2:	(01.1101) x (11.1001)	→	(011001)

Finally, we mutate each bit of the resulting chromosomes with some small probability – say $p_{mutation}$=0.05. In our example we find that values of the 5th bit in C_4 and 6th bit in C_5 are flipped. The resulting strings make up our 2nd generation chromosome population. By chance, the first loop through the algorithm has successfully turned up the most-fit chromosome – C_4=(111111) –> $f(C_4) = 63^2 = 3969$ – but in general the entire procedure would have to be repeated many times to approach the "desired" solution.

The table below summarizes the above steps:

Initial Population	Initial Fitness	Expected Copies	Actual Copies	Mating Population	Crossover Operation[1]	Mutation Operation	New Fitness
(101101)	2025	1.4	1	(101101)	(134)->(101001)	(101001)	1681
(010110)	484	0.3	0	(111001)	(226)->(111101)	(111101)	3481
(111001)	3249	2.2	3	(111001)	(355)->(111001)	(111001)	3249
(101011)	1849	1.3	1	(111001)	(431)->(111101)	(111111)	3969
(010001)	289	0.2	0	(101011)	(553)->(101011)	(101010)	1764
(011101)	841	0.6	1	(011101)	(622)->(011001)	(011001)	625

[1] The crossover operator (xyz) means that chromosomes C_x and C_z exchange bits at the y^{th} bit.

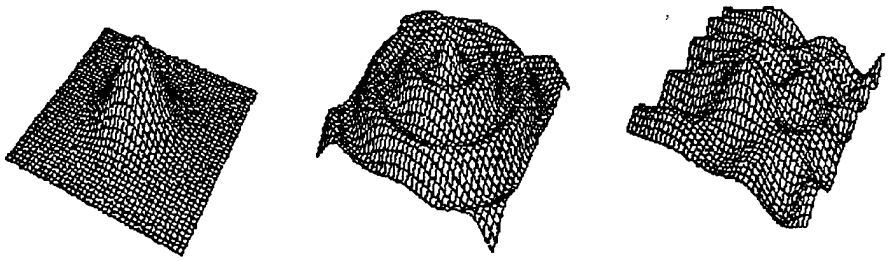
The Fitness Landscape

A solution search space, **x**, together with a fitness function, $f(\mathbf{x})$, make up what is called a *fitness landscape*. The term "landscape" comes from

visualizing a three-dimensional geographical landscape consisting of heights h=f(x,y) of a two-dimensional location **x**=(x,y). Particular problems, of course, may involve an arbitrary number of dimensions, but it is still helpful to keep this simple image in mind. The term "fitness" comes from Darwinian biology, and refers to the fitness of an individual to survive as a function either of its *phenotype* (or higher-level properties and/or behaviors) or its *genotype* (or lower-level genetic code).

Biological fitness is generally very difficult to define since it is usually a complicated (and changing!) function of the interactions between an organism and other organisms and interactions between the organism and its environment. In a biological context and/or biology-based setting (such as in studies of artificial life), the fitness landscape is also often referred to as an *adaptive landscape*. Other "fitness functions," which, depending on the particular problem, may be considerably easier to define than their biological cousins, include energy and free-energy landscapes from physics and chemistry, and cost (or objective) functions from combinatorial optimization problems in computer science.

Figure 92. Sample forms of fitness landscapes



As our simple geographical landscape metaphor might suggest, a variety of fitness landscapes are possible, each with their own strengths and weaknesses when it comes to "submitting" to a GA solution: completely flat landscapes, landscapes with a single isolated minimum and/or maximum, landscapes having several minima and/or maxima with equal heights, or landscapes with many unequal and irregularly spaced local minima and/or maxima.

Since GAs, like other combinatorial optimization schemes (such as *simulated annealing*), depend essentially on their "hill-climbing" ability to ascend (or descend) towards the desired global maximum (or minimum), how successful the climb – and hence, the approach to the solution – will be, depends on what the landscape looks like. What a

given landscape looks like, in turn, depends strongly on its metric; that is, on the function d=d(**x**,**x**') that is used to measure the distance between any two points **x** and **x**'. Since GAs tend to keep nearby bits near each other, embedded correlations among subsets of a chromosome's genes can sometimes be exploited to produce a "natural ordering" for the given landscape.

Landscapes with a single smoothly increasing "bump,"' such as the one shown in figure 92-a, for example, are usually amenable to any systematic climb towards larger values. On the other hand, landscapes with a single isolated maximum that sits on an otherwise even-leveled surface may not be so easy to "solve," because at no point on the surface is there a clue as to which direction to proceed in to move towards the maximum. More "rugged" landscapes, such as those shown in figures 92-b and 92-c, with their multiple, and in the case of figure 92-c, irregularly spaced and sized, local maxima, may present even greater challenges to "hill-climbing" methods. An excellent review of optimization on rugged landscapes is given by Palmer [48]. Kauffman ([49,50]) discusses the biological implications of rugged fitness landscapes.

How Do GAs Work?

While GAs are very simple to describe and implement on a computer, their behavior can be quite complex. There are a number of fundamental questions concerning how GAs work, not all of which have been completely answered. The first, and obvious, question is how do they manage to work at all? Given the vast number of possible genotypes of a size N "solution" ($=2^N$), it is not immediately clear why *any* finite search-strategy – be it serial, parallel, hill-climbing or whatever – should ever consistently come close to the desired solution in a reasonable time, particularly for large N. Since the efficacy of an optimization scheme depends strongly on the fitness landscape, one would also like to characterize the kinds of fitness landscapes that are most amenable to a GA solution. It is also important to explore ways in which GAs differ from more traditional hill-climbing methods like gradient-ascent. Are all such methods, GAs included, equally adept at "solving" the same sorts of problems? Or are different methods best suited for specific kinds of problems? If so, how are these problems, and presumably their fitness landscapes, different from one another? While it would take us too far afield to explore these and other important questions in any great depth, we will briefly discuss a notion that most formal studies of the theory behind GAs begin with: the *building-block hypothesis.*

The Building-Block Hypothesis

An heuristic explanation of why GA work – called the building-block hypothesis [25, 24] – is based on the idea that good solutions tend to be formed out of sets of good building-blocks (or *schemas*). GAs discover these solutions by assigning higher fitness-levels to – and therefore tending to retain, over the course of successive generations – sets of strings containing good schemas.

By schema, we mean templates, or forms for particular kinds of strings. For example, the schema S=(1****0), where * is a "wildcard" that stands for either bit-value 0 or 1, represents the template for all length-6 chromosomes whose first bit $\beta_1=1$ and last bit $\beta_6=1$. In this case, since the schema contains one fixed bit and the distance between the outer most fixed bits is 5, S is said to be an *order*-1 schema with *defining length* $\delta=5$.

The above example, in which we used length-6 chromosomes to maximize the function $f(x)=x^2$, illustrates why schema can be thought of as simple building-blocks of "fit" genes. In that example, any chromosome of the form (1*****) is obviously more fit than (0*****), and thus forms a basic building block out of which the best "solutions" must be constructed.

Now, to be sure, not every possible subset of the solution-space can be described as a schema. Simple counting shows that a length-N chromosome can have 2^N possible configurations, and therefore 2^{2^N} possible subsets, but only 3^N different schemas. Nonetheless, it is a central axiom of the building-block hypothesis that it is precisely the set of schemas that are effectively being processed by GAs.

The schema population can be estimated using a simple mean-field-like argument. Let s represent a schema in a size-K population P(t) at time t, and Z(P,t) instances of the schema at time t. Let f(s) be the fitness of the string s, $\bar{f}_S$ be the average fitness of instances of s at time t, and $\bar{f} = K^{-1}\Sigma_i$ be the average fitness of the population. Then the expected number of instances of s at time t+1, Z(P,t+1), is equal to

$$Z(P,t+1) = \sum_{s\in S} \frac{f(s)}{\bar{f}} = \bar{f}^{-1}\Sigma_{s\in S} f(s) = \frac{\bar{f}_S}{\bar{f}} Z(P,t)\,,$$

since, by definition, $\bar{f}_S = \Sigma_{s\in S} f(s)/Z(P,t)$.

This basic difference equation – known as the *Schema Theorem* [24] – expresses the fact that the sample representation of schemas whose average fitness remains above average relative to the whole population increases exponentially over time. As it stands, however, this equation

addresses only the reproduction operator, and ignores effects of both crossover and mutation.

A lower bound on the overall effect of crossover, which can both create and destroy instances of a given schema, can be estimated by calculating the probability, $p_C(S)$, that crossover leaves a schema S unaltered. Let p_c be the probability that the crossover operation will be applied to a string. Since a schema S will be destroyed by crossover if the operation is applied anywhere within its defining length, the probability that S will be destroyed is equal to $p_c*\delta(S)/(K-1)$, where $\delta(S)$ is the defining length of S. Hence, the probability of survival $p_S=1-p_c*\delta(S)/(K-1)$, and the expression for Z(P,t+1) takes the updated form:

$$Z(P,t+1) \geq \frac{f_S}{\bar{f}}\left(1-p_c\frac{\delta(S)}{K-1}\right)Z(P,t)$$

Finally, in order to also take into account the mutation operator, we note that the probability that a schema S survives under mutation is given by $p_M(S)=(1-p_m)^{O(S)}$, where p_m is the single-bit mutation probability and $O(S)$ is the number of fixed-bits (i.e., the order) of S. With this we can now express the *Schema Theorem* that (partially) respects the operations of reproduction, crossover and mutation:

$$Z(P,t+1) \geq \frac{f_S}{\bar{f}}\left(1-p_c\frac{\delta(S)}{K-1}\right)\left\{(1-p_m)^{O(S)}\right\}Z(P,t)$$

We conclude from this basic theorem that the sample representation of low-order schemas with above average fitness relative to the fitness of the population increases exponentially over time. (The fact that we have ignored possible crossover and/or mutation induced creations of previously nonexisting instances of schemas means only that the right hand side of the above equation represents a lower bound; the conclusion remains generally valid, as it stands.)

Dueling Parasites

We outline a potentially powerful generalization of the basic genetic algorithm introduced by Hillis [6], which may have a natural application to the modeling of combat.

Conventional genetic algorithms search for "solutions" to problems by "evolving" large populations of approximate solutions, each candidate solution represented by a chromosome. The genetic algorithm evolves one population of chromosomes into another according to their fitness using various genetic operators (such as crossover and mutation), and, eventually, after many generations, the population comes to consist only of the "most-fit" chromosomes.

This basic recipe is useful for finding near-optimal solutions for many kinds of problems. One of the major difficulties that *all solution schemes* for solving combinatorial optimization problems must contend with, however, is the classical problem of the search space containing local optima: once a search algorithm finds what it "thinks" is the global optimal solution, it is generally difficult for it to find ways to not be "locked into" the local optimum.

Hillis attacks this problem by exploiting host-parasite interactions among two coupled genetic algorithm populations. To illustrate the idea, consider his testbed system, which consists of finding a sorting algorithm for elements of a set of fixed size that requires the smallest number of comparisons and exchanges to be made among the elements. The overall problem is to design an efficient *sorting network*, which is a sorting algorithm in which the sequence of comparisons and exchanges is made in a predetermined order. A candidate sorting network, once defined (by a chromosome), is easy to test.

Now, Hillis' idea is to set up not one but *two* interacting genetic algorithm populations, one population consisting of "solutions," or sorting programs (the *hosts*), and the other consisting of "sorting problems" (the *parasites*). Having the two populations interact effectively sets up an "arms-race" between the two populations. While the hosts are trying to find better and better ways to sort the problems, the parasites are trying to make the hosts less and less adept at sorting the problems by making the problems "harder."

The interaction between the two populations dynamically alters the form of the fitness function. Just as the hosts reach the top of a fitness "hill," the parasites deform the fitness landscape so that the hill becomes a "valley" that the hosts are then forced to find ways to climb out of and start looking for new peaks. When the population of programs finally reaches a hill that the parasites cannot find a way to turn into a valley, the combined efforts of the co-evolving hosts and parasites has found a global optimum. Thus, the joint, coupled, population pools are able to find better solutions quicker than the evolutionary dynamics of populations consisting of sorting programs alone.

The application to combat modeling is conceptually straightforward. The idea is to apply genetic algorithms not to just one side of a conflict, or to use genetic algorithms to find "optimal" combat tactics for fixed sets of constraints and environments, but to use *joint, coupled, pools of populations*, one side of which represents a set of tactics or strategies to deal with specific scenarios, and the other side of which seeks ways to alter the environment in ways that make it harder and harder for those tactics or strategies to work.

Appendix C: Source Code for ISAAC

Below is the ANSI C source code for version 1.8.4 of **ISAAC_CE** (i.e., the *Core Engine*; see Table 4). Screen and graphics functions are those defined in **graph.h** of *Microsoft's* Visual C/C++ compiler for DOS (v1.52). Note that to run ISAAC using the parameter values appearing in the header file **ISAAC.h**, ISAAC requires 8-16 MB of extended memory and must thus be compiled using a DOS-extender such as Pharlap's *SDK 286\DOS-Extender*.

Header File

```
// Version number
#define ISAAC_VERSION "ISAAC / Version 1.8.4"

// maximum size of square battlefield
#define MAXFIELDSIZE 151

// maximum ISAACA sensor range
#define MAXSENSORRANGE 10

// maximum possible interpoint distance
#define MAXINTERPOINTDIST (int)(1.414214 * MAXFIELDSIZE)

// maximum number of local commanders
#define MAXCOMMANDNUM 25

// maximum number of ISAACAs that can be under
// the command of a local commander
#define MAXUNDERCOMMAND 100

// maximum number of ISAACs on either side
#define MAXISAACNUM 501

// maximum number of squads per side
#define MAXSQUADNUM 11

// maximum number of enemy ISAACs that can be
// located in the neighborhood of an ISAACA
#define MAXNEIGHBORNUM 2*MAXISAACNUM+1

// maximum number of 'obstacles' in battlefield
#define TERRAINMAXNUM 25

// maximum size of clusters to consider for
// calculating distribution
#define MAXCLUSTERSIZE 2*MAXISAACNUM+1
```

Structures

```
//*****************************************************************************************
//
//        - statistics                    : summary statistics and measures
//        - red_GC_parameters             : red gloabl commander parameters
//        - blue_GC_parameters            : blue global commander parameters
//        - red_command_parameters        : red localcommander parameters
//        - blue_command_parameters       : blue localcommander parameters
//        - battle_parameters             : battlefield/combat parameters
//        - red_parameters                : red ISAACA force parameters
//        - blue_parameters               : blue ISAACA force parameters
//
//*****************************************************************************************
struct statistics
{
int stat_flag;                          // =1 if statistics are to be computed, else 0
int interpoint_flag;                    // =1 if interpoint-dist to be calculated, else 0
int max_interpoint_dist;                // maximum distance for which interpoint dist != 0
int max_G_dist;                         // maximum distance for which ISAACA-goal dist != 0
float RG_dist[MAXINTERPOINTDIST];       // distribution of red-blueflag distances
float BG_dist[MAXINTERPOINTDIST];       // distribution of blue-redflag distances
float RR_interpoint[MAXINTERPOINTDIST]; // distribution of red-red interpoint distances
float BB_interpoint[MAXINTERPOINTDIST]; // distribution of blue-blue interpoint distances
float RB_interpoint[MAXINTERPOINTDIST]; // distribution of red-blue interpoint distances
float RG_dist_ave;                      // average of distribution of red-blueflag distances
float BG_dist_ave;                      // average of distribution of blue-redflag distances
float RR_interpoint_ave;                // average of distribution of RR interpoint distances
float BB_interpoint_ave;                // average of distribution of BB interpoint distances
float RB_interpoint_ave;                // average of distribution of RB interpoint distances
float RG_dist_adev;                     // average deviation of dist of red-blueflag distances
float BG_dist_adev;                     // average deviation of dist of blue-redflag distances
float RR_interpoint_adev;               // average deviation of dist of RR interpoint distances
float BB_interpoint_adev;               // average deviation of dist of BB interpoint distances
float RB_interpoint_adev;               // average deviation of dist of RB interpoint distances
float RG_dist_sdev;                     // standard deviation of dist of red-blueflag distances
float BG_dist_sdev;                     // standard deviation of dist of blue-redflag distances
float RR_interpoint_sdev;               // standard deviation of dist of RR interpoint distances
float BB_interpoint_sdev;               // standard deviation of dist of BB interpoint distances
float RB_interpoint_sdev;               // standard deviation of dist of RB interpoint distances
float RG_dist_var;                      // variance of dist of red-blueflag distances
float BG_dist_var;                      // variance of dist of blue-redflag distances
float RR_interpoint_var;                // variance of dist of RR interpoint distances
float BB_interpoint_var;                // variance of dist of BB interpoint distances
float RB_interpoint_var;                // variance of dist of RB interpoint distances
int entropy_flag;                       // =1 if spatial entropy is to be calculated, else =0
int block_xmin_1[17];                   // ith block x-min for 4x4 calc of entropy
int block_xmax_1[17];                   // ith block x-max for 4x4 calc of entropy
int block_ymin_1[17];                   // ith block y-min for 4x4 calc of entropy
int block_ymax_1[17];                   // ith block y-max for 4x4 calc of entropy
int block_xmin_2[65];                   // ith block x-min for 8x8 calc of entropy
int block_xmax_2[65];                   // ith block x-max for 8x8 calc of entropy
int block_ymin_2[65];                   // ith block y-min for 8x8 calc of entropy
int block_ymax_2[65];                   // ith block y-max for 8x8 calc of entropy
int block_xmin_3[257];                  // ith block x-min for 16x16 calc of entropy
int block_xmax_3[257];                  // ith block x-max for 16x16 calc of entropy
int block_ymin_3[257];                  // ith block y-min for 16x16 calc of entropy
int block_ymax_3[257];                  // ith block y-max for 16x16 calc of entropy
float red_entropy_1;                    // entropy measure of state using 4x4 array
float red_entropy_2;                    // entropy measure of state using 8x8 array
float red_entropy_3;                    // entropy measure of state using 16x16 array
float blue_entropy_1;                   // entropy measure of state using 4x4 array
float blue_entropy_2;                   // entropy measure of state using 8x8 array
float blue_entropy_3;                   // entropy measure of state using 16x16 array
float red_blue_entropy_1;               // entropy measure of state using 4x4 array
float red_blue_entropy_2;               // entropy measure of state using 8x8 array
float red_blue_entropy_3;               // entropy measure of state using 16x16 array
int cluster_1_flag;                     // =1 if D=1 clusters are to be found
int cluster_2_flag;                     // =1 if D=2 clusters are to be found
int number_of_clusters_1;               // total number of clusters (using D=1)
int number_of_clusters_2;               // total number of clusters (using D=2)
```

```
int clusters_1[MAXCLUSTERSIZE+1];                  // number of clusters of size i (using D=1)
int clusters_2[MAXCLUSTERSIZE+1];                  // number of clusters of size i (using D=2)
float cluster_1_ave;                               // average cluster size
float cluster_1_adev;                              // average deviation of cluster distribution
float cluster_1_sdev;                              // standard deviation of cluster distribution
float cluster_1_var;                               // variance of cluster distribution
float cluster_2_ave;                               // average cluster size
float cluster_2_adev;                              // average deviation of cluster distribution
float cluster_2_sdev;                              // standard deviation of cluster distribution
float cluster_2_var;                               // variance of cluster distribution
int neighbors_flag;                                // =1 if neighbor-routine is to be activated, else =0
float red_in_red_ave[MAXSENSORRANGE+1];            // average number of red ISAACAS in red at dist D
float blue_in_red_ave[MAXSENSORRANGE+1];           // average number of blue ISAACAS in red at dist D
float all_in_red_ave[MAXSENSORRANGE+1];            // average number of ISAACAS in red at dist D
float red_in_blue_ave[MAXSENSORRANGE+1];           // average number of red ISAACAS in blue at dist D
float blue_in_blue_ave[MAXSENSORRANGE+1];          // average number of blue ISAACAS in blue at dist D
float all_in_blue_ave[MAXSENSORRANGE+1];           // average number of ISAACAS in blue at dist D
float red_in_red_adev[MAXSENSORRANGE+1];           // average deviation of red ISAACAS in red at dist D
float red_in_red_sdev[MAXSENSORRANGE+1];           // standard deviation of red ISAACAS in red at dist D
float red_in_red_var[MAXSENSORRANGE+1];            // variance number of red ISAACAS in red at dist D
float blue_in_red_adev[MAXSENSORRANGE+1];          // average deviation of blue ISAACAS in red at dist D
float blue_in_red_sdev[MAXSENSORRANGE+1];          // standard deviation of blue ISAACAS in red at dist D
float blue_in_red_var[MAXSENSORRANGE+1];           // variance number of blue ISAACAS in red at dist D
float all_in_red_adev[MAXSENSORRANGE+1];           // average deviation of ISAACAS in red at dist D
float all_in_red_sdev[MAXSENSORRANGE+1];           // standard deviation of ISAACAS in red at dist D
float all_in_red_var[MAXSENSORRANGE+1];            // variance number of ISAACAS in red at dist D
float red_in_blue_adev[MAXSENSORRANGE+1];          // average deviation of red ISAACAS in blue at dist D
float red_in_blue_sdev[MAXSENSORRANGE+1];          // standard deviation of red ISAACAS in blue at dist D
float red_in_blue_var[MAXSENSORRANGE+1];           // variance number of red ISAACAS in blue at dist D
float blue_in_blue_adev[MAXSENSORRANGE+1];         // average deviation of blue ISAACAS in blue at dist D
float blue_in_blue_sdev[MAXSENSORRANGE+1];         // standard deviation of blue ISAACAS in blue at dist D
float blue_in_blue_var[MAXSENSORRANGE+1];          // variance number of blue ISAACAS in blue at dist D
float all_in_blue_adev[MAXSENSORRANGE+1];          // average deviation of ISAACAS in blue at dist D
float all_in_blue_sdev[MAXSENSORRANGE+1];          // standard deviation of ISAACAS in blue at dist D
float all_in_blue_var[MAXSENSORRANGE+1];           // variance number of ISAACAS in blue at dist D
int center_mass_flag;                              // =1 if center-of-mass is to be computed
float red_CM_x;                                    // x-coordinate of red's center-of-mass
float red_CM_y;                                    // y-coordinate of red's center-of-mass
float blue_CM_x;                                   // x-coordinate of blue's center-of-mass
float blue_CM_y;                                   // y-coordinate of blue's center-of-mass
float total_CM_x;                                  // x-coordinate of red and blue center-of-mass
float total_CM_y;                                  // x-coordinate of red and blue center-of-mass
int goal_stat_flag;                                // =1 if goal-statistics to be calculated, else 0
float red_in_BG[6];                                // number of red ISAACAs near blue flag (radius=1,..,5)
float blue_in_RG[6];                               // number of blue ISAACAs near red flag (radius=1,..,5)
int red_sensor_min;                                // minimum sensor range among red squads
int blue_sensor_min;                               // minimum sensor range among blue squads
};

struct red_GC_parameters
{
int red_GC_flag;                                   // =1 if red ISAACAs have a global commander, else =0
float swath_area[MAXCOMMANDNUM][17];               // contains 'swath' areas centered at (x,y) coor of LC
float swath_den_AB[MAXCOMMANDNUM][17];             // contains 'swath' densities of alive blue ISAACAs
float swath_den_IB[MAXCOMMANDNUM][17];             // contains 'swath' densities of injured blue ISAACAs
float red_GC_fear_index;                           // 0 <= f <= 1 = fear index; 0=no fear; 1= max fear
float red_gc_direction_wt[17];                     // default weights for direction 'orders'(1-16) to LCs
int red_GC_direction_x[17];                        // x-coordinate of possible GC goals for LCs
int red_GC_direction_y[17];                        // y-coordinate of possible GC goals for LCs
int red_GC_goal_x[MAXCOMMANDNUM];                  // x-coordinate of the ith red ISAACAs GC goal for LCs
int red_GC_goal_y[MAXCOMMANDNUM];                  // y-coordinate of the ith red ISAACAs GC goal for LCs
int red_GC_help_x[MAXCOMMANDNUM];                  // x-coordinate of the LC that the ith LC is to 'help'
int red_GC_help_y[MAXCOMMANDNUM];                  // y-coordinate of the LC that the ith LC is to 'help'
float red_GC_w_alpha;                              // global red command weight for alive blue density
float red_GC_w_beta;                               // global red command weight for injured blue density
float red_GC_frac_R[3];                            // fractions of range in which to weigh swath dens
float red_GC_w_swath[4];                           // swath weights to apply to the ith fractional radius
float red_GC_max_blue_factor;                      // max number of blues = max_factor*(# under command)
int red_GC_help_radius;                            // defines area about LC in which the LC may seek help
float red_GC_health_thresh;                        // threshold health for an LC to help another
float red_GC_rel_health_thresh;                    // threshold relative health for an LC to help another
};
```

```
struct blue_GC_parameters
{
int blue_GC_flag;                          // =1 if blue ISAACAs have a global commander, else =0
float swath_area[MAXCOMMANDNUM][17];       // contains 'swath' areas centered at (x,y) coor of LC
float swath_den_AR[MAXCOMMANDNUM][17];     // contains 'swath' densities of alive red ISAACAs
float swath_den_IR[MAXCOMMANDNUM][17];     // contains 'swath' densities of injured red ISAACAs
float blue_GC_fear_index;                  // 0 <= f <= 1 = fear index; 0=no fear; 1= max fear
float blue_gc_direction_wt[17];            // default weights for direction 'orders'(1-16) to LCs
int blue_GC_direction_x[17];               // x-coordinate of possible GC goals for LCs
int blue_GC_direction_y[17];               // y-coordinate of possible GC goals for LCs
int blue_GC_goal_x[MAXCOMMANDNUM];         // x-coor of the ith blue ISAACAs GC goal for LCs
int blue_GC_goal_y[MAXCOMMANDNUM];         // y-coor of the ith blue ISAACAs GC goal for LCs
int blue_GC_help_x[MAXCOMMANDNUM];         // x-coordinate of the LC that the ith LC is to 'help'
int blue_GC_help_y[MAXCOMMANDNUM];         // y-coordinate of the LC that the ith LC is to 'help'
float blue_GC_w_alpha;                     // global blue command weight for alive red density
float blue_GC_w_beta;                      // global blue command weight for injured red density
float blue_GC_frac_R[3];                   // fractions of range in which to weigh swath dens
float blue_GC_w_swath[4];                  // swath weights to apply to the ith fractional radius
float blue_GC_max_red_factor;              // max number of reds = max_factor*(# under command)
int blue_GC_help_radius;                   // defines area about LC in which the LC may seek help
float blue_GC_health_thresh;               // threshold health for an LC to help another
float blue_GC_rel_health_thresh;           // threshold relative health for an LC to help another
};

struct red_command_parameters
{
short red_command_color;
int num_red_commanders;                    // number of red local commanders
int red_command_flag;                      // =1 if red ISAACAs have local commanders, else =0
int red_command_patch;                     // =1 if 3-by-3 patch; =2 if 5-by-5
int red_patch_choice_flag;                 // if =0 then random patch choice, else min D to old
int red_command_radius[MAXCOMMANDNUM];     // command radius; area = 9 (2r+1)-by-(2r+1) patches
int red_command_R[MAXCOMMANDNUM];          // command radius of 'entire' command swath
int red_command[MAXISAACNUM];              // =2 if = local commander; 1 if under command; else 0
int reds_commander[MAXISAACNUM];           // =0 if not under command; label of commander if yes
int red_ISAACA_commander[MAXCOMMANDNUM];   // label (i=..irednum) of ith ISAACA's local commander
int red_ISAACA_label[MAXISAACNUM];         // label (i=..inumcmders) of ith ISAACA; =0 if not cmd
int red_num_under_command[MAXCOMMANDNUM];  // number of red ISAACAs under ith local command
int red_num_under_command_0[MAXCOMMANDNUM]; // # of red ISAACAs under ith local command (t=0)
int red_ISAACA_under_command[MAXCOMMANDNUM][MAXUNDERCOMMAND];
        // label (i=1..irednum) of jth ISAACA under the ith local commander's command
float red_min_command_dist;                // minimum ISAACA distance from their local commanders
float red_w_to_commander_def;              // red's relative weight to move towards commander
float red_w_obey_command_def;              // red's relative weight to comply with command order
float red_w_to_commander;                  // red's relative weight to move towards commander
float red_w_obey_command;                  // red's relative weight to comply with command order
int red_command_goal_x[MAXCOMMANDNUM];     // x-coordinate for red ISAACAs local goal
int red_command_goal_y[MAXCOMMANDNUM];     // y-coordinate for red ISAACAs local goal
int red_prior_goalx[MAXCOMMANDNUM];        // x-coor of prior red 'command' patch goal
int red_prior_goaly[MAXCOMMANDNUM];        // y-coor of prior red 'command' patch goal
int red_local_goal_x[MAXISAACNUM];         // x-coordinate of the ith red ISAACAs local goal
int red_local_goal_y[MAXISAACNUM];         // y-coordinate of the ith red ISAACAs local goal
float red_command_w_alpha[MAXCOMMANDNUM];  // local red command weight for (AR-AB)/(AR+IR)
float red_command_w_beta[MAXCOMMANDNUM];   // local red command weight for (AR-IB)/(AR+IR)
float red_command_w_delta[MAXCOMMANDNUM];  // local red command weight for (IR-AB)/(AR+IR)
float red_command_w_gamma[MAXCOMMANDNUM]; // local red command weight for (IR-IB)/(AR+IR)
float red_command_w1rdeff[MAXCOMMANDNUM];  // default weight for alive red --> alive red
float red_command_w2rdeff[MAXCOMMANDNUM];  // default weight for alive red --> alive blue
float red_command_w3rdeff[MAXCOMMANDNUM];  // default weight for alive red --> injured red
float red_command_w4rdeff[MAXCOMMANDNUM];  // default weight for alive red --> injured blue
float red_command_w5rdeff[MAXCOMMANDNUM];  // default weight for alive red --> red goal
float red_command_w6rdeff[MAXCOMMANDNUM];  // default weight for alive red --> blue goal
int red_command_adv[MAXCOMMANDNUM];        // local command red advance threshold
int red_command_clus[MAXCOMMANDNUM];       // local command red cluster threshold
int red_command_com[MAXCOMMANDNUM];        // local command red combat threshold
int red_command_srange[MAXCOMMANDNUM];     // local command red sensor range
int red_command_advrange[MAXCOMMANDNUM];   // range in which red # > threshold to advance
int red_subordinate_color_flag;            // paint subordinate ISAACAs different color if =1
float red_w_obey_GC_def[MAXCOMMANDNUM];    // default weight for LC to obey global commander
float red_w_obey_GC[MAXCOMMANDNUM];        // weight for LC to obey global commander
float red_w_help_LC_def[MAXCOMMANDNUM];    // weight for 'helping neighboring LCs (=1-w_goal)
float red_w_help_LC[MAXCOMMANDNUM];        // weight for 'helping neighboring LCs (=1-w_goal)
float red_LC_health[MAXCOMMANDNUM];        // the ith (i=1...numcmd) LC's health
};
```

```
struct blue_command_parameters
{
short blue_command_color;
int num_blue_commanders;                          // number of blue local commanders
int blue_command_flag;                            // =1 if blue ISAACAs have local commanders, else =0
int blue_command_patch;                           // =1 if 3-by-3 patch; =2 if 5-by-5
int blue_patch_choice_flag;                       // if =0 then random patch choice, else min D to old
int blue_command_radius[MAXCOMMANDNUM];           // command radius; area = 9 (2r+1)-by-(2r+1) patches
int blue_command_R[MAXCOMMANDNUM];                // command radius of 'entire' command swath
int blue_command[MAXISAACNUM];                    // =2 if = local commander; 1 if under command; else 0
int blues_commander[MAXISAACNUM];                 // =0 if not under command; label of commander if yes
int blue_ISAACA_commander[MAXCOMMANDNUM];  // label (i=1..ibluenum) of ith's local commander
int blue_ISAACA_label[MAXISAACNUM];               // label (i=..inumcmders) of ith ISAACA; =0 if not cmd
int blue_num_under_command[MAXCOMMANDNUM]; // number of blue ISAACAs under ith local command
int blue_num_under_command_0[MAXCOMMANDNUM];// # of blue ISAACAs under ith local command (t=0)
int blue_ISAACA_under_command[MAXCOMMANDNUM][MAXUNDERCOMMAND];
        // label (i=1..ibluenum) of jth ISAACA under the ith local commander's command
float blue_min_command_dist;                      // minimum ISAACA distance from their local commanders
float blue_w_to_commander_def;                    // blue's relative weight to move towards commander
float blue_w_obey_command_def;                    // blue's relative weight to comply with command order
float blue_w_to_commander;                        // blue's relative weight to move towards commander
float blue_w_obey_command;                        // blue's relative weight to comply with command order
int blue_command_goal_x[MAXCOMMANDNUM];           // x-coordinate for blue ISAACAs local goal
int blue_command_goal_y[MAXCOMMANDNUM];           // y-coordinate for blue ISAACAs local goal
int blue_prior_goalx[MAXCOMMANDNUM];              // x-coor of prior blue 'command' patch goal
int blue_prior_goaly[MAXCOMMANDNUM];              // y-coor of prior blue 'command' patch goal
int blue_local_goal_x[MAXISAACNUM];               // x-coordinate of the ith blue ISAACAs local goal
int blue_local_goal_y[MAXISAACNUM];               // y-coordinate of the ith blue ISAACAs local goal
float blue_command_w_alpha[MAXCOMMANDNUM];  // local bluecommand weight for (AB-AR)/(AB+IB)
float blue_command_w_beta[MAXCOMMANDNUM];   // local bluecommand weight for (AB-IR)/(AB+IB)
float blue_command_w_delta[MAXCOMMANDNUM];  // local bluecommand weight for (IB-AR)/(AB+IB)
float blue_command_w_gamma[MAXCOMMANDNUM];// local bluecommand weight for (IB-IR)/(AB+IB)
float blue_command_w1bdeff[MAXCOMMANDNUM];  // default weight for alive red --> alive red
float blue_command_w2bdeff[MAXCOMMANDNUM];  // default weight for alive red --> alive blue
float blue_command_w3bdeff[MAXCOMMANDNUM];  // default weight for alive red --> injured red
float blue_command_w4bdeff[MAXCOMMANDNUM];  // default weight for alive red --> injured blue
float blue_command_w5bdeff[MAXCOMMANDNUM];  // default weight for alive red --> red goal
float blue_command_w6bdeff[MAXCOMMANDNUM];  // default weight for alive red --> blue goal
int blue_command_adv[MAXCOMMANDNUM];              // local command blue advance threshold
int blue_command_clus[MAXCOMMANDNUM];             // local command blue cluster threshold
int blue_command_com[MAXCOMMANDNUM];              // local command blue combat threshold
int blue_command_srange[MAXCOMMANDNUM];           // local command red sensor range
int blue_command_advrange[MAXCOMMANDNUM];         // range in which red # > threshold to advance
int blue_subordinate_color_flag;                  // paint subordinate ISAACAs different color if =1
float blue_w_obey_GC_def[MAXCOMMANDNUM];          // default weight for LC to obey global commander
float blue_w_obey_GC[MAXCOMMANDNUM];              // weight for LC to obey global commander
float blue_w_help_LC_def[MAXCOMMANDNUM];          // weight for 'helping neighboring LCs (=1-w_goal)
float blue_w_help_LC[MAXCOMMANDNUM];              // weight for 'helping neighboring LCs (=1-w_goal)
float blue_LC_health[MAXCOMMANDNUM];              // the ith (i=1...numcmd) LC's health
};

struct battle_parameters
{
short goalcolor;
short boxcolor;
int squad_color_flag;                             //=1 if squads are color-highlighted, else =0
int default_speed;                                // =1 if run is FAST, =2 if run is SLOW
int ioutdata;                                     // output: 1=screen only; 2=file only; 3=both
int ichoice;                                      // run flag: 1=run engine; 2=playback file
int isize;                                        // user specified battlefield size
int initdist;                                     // initial force distribution flag
int ibattlebox_red_length_squad[MAXSQUADNUM];     // length of box containing initial distribution
int ibattlebox_red_width_squad[MAXSQUADNUM];      // width of box containing initial distribution
int ibattlebox_red_cen_x_squad[MAXSQUADNUM];      // x-coodinate of the center of red's initial box
int ibattlebox_red_cen_y_squad[MAXSQUADNUM];      // y-coodinate of the center of red's initial box
int ibattlebox_blue_length_squad[MAXSQUADNUM];    // length of box containing initial distribution
int ibattlebox_blue_width_squad[MAXSQUADNUM];     // width of box containing initial distribution
int ibattlebox_blue_cen_x_squad[MAXSQUADNUM];     // x-coodinate of the center of blue's initial box
int ibattlebox_blue_cen_y_squad[MAXSQUADNUM];     // y-coodinate of the center of blue's initial box
int itermcond;                                    // termination condition flag (1=goal; 2=none)
int imove_selection;                              // 1 = FIXED order; 2 = random order
int max_combat_flag;                              // 1=# of sim engmnts lmtd; 0=no limit
int terrain_flag;                                 // 1 = terrain to be used; 1 = no
```

```
int terrain_num;                                  // number of terrain block
int terrain_size[TERRAINMAXNUM];                  // radius of ith terrain block
int terrain_center_x[TERRAINMAXNUM];              // x-coordinate of the the ith block's center
int terrain_center_y[TERRAINMAXNUM];              // y-coordinate of the the ith block's center
int ioccupation[MAXFIELDSIZE][MAXFIELDSIZE];      // =2 if terrain, 1 if occupied, else 0
int reconstitution_flag;                          // if 0 then no reconstitution, else reconstitution on
int red_fratricide_flag;                          // =1 if red ISAACAs can accidentally kill red ISAACAs, else 0
int blue_fratricide_flag;                         // =1 if blue ISAACAs can accidentally kill blue ISAACAs, else 0
int red_frat_rad;                                 // radius surrounding targeted blue within which reds can be killed
int blue_frat_rad;                                // radius surrounding targeted red within which blues can be killed
int red_frat_count;                               // cummulative total of red fratricide 'hits'
int blue_frat_count;                              // cummulative total of blue fratricide 'hits'
float red_frat_prob;                              // probability that red is accidentally shot by red
float blue_frat_prob;                             // probability that blue is accidentally shot by blue
};

struct red_parameters
{
int num_red_squads;                               // number of red squads
int num_per_red_squad[MAXSQUADNUM];               // number of ISAACAs in the ith red squad
int squad[MAXISAACNUM];                           // number of the squad to which the ith ISAACA belongs
int display_red_squad;                            // which red squad to show the parameters of on screen
short redcolor;                                   // color code for alive red ISAACAs
short squadcolor;                                 // color code for highlighting individual squads
int redgoalx;                                     // x coordinate of red goal
int redgoaly;                                     // y coordinate of red goal
int redx[MAXISAACNUM];                            // x-coordinate of ith red ISAAC
int redy[MAXISAACNUM];                            // y-coordinate of ith red ISAAC
int rseer[MAXISAACNUM];                           // =1 if red sees red and =0 otherwise
int rseeb[MAXISAACNUM];                           // =1 if red sees blue and =0 otherwise
int rseercomm[MAXISAACNUM];                       // =1 if red sees red via COMM link
int rseebcomm[MAXISAACNUM];                       // =1 if red sees blue via COMM link
int rstatus[MAXISAACNUM];                         // =1 if alive, 1 if injured, 0 if dead
int ibinr[MAXISAACNUM];                           // number of blue isaacs in red isaac range
float w1red[MAXISAACNUM];                         // active weight for red --> alive red
float w2red[MAXISAACNUM];                         // active weight for red --> alive blue
float w3red[MAXISAACNUM];                         // active weight for red --> injured red
float w4red[MAXISAACNUM];                         // active weight for red --> injured blue
float w5red[MAXISAACNUM];                         // active weight for red --> red goal
float w6red[MAXISAACNUM];                         // active weight for red --> blue goal
int irednum;                                      // total number of red ISAACs
int irsrange[MAXISAACNUM];                        // red sensor range of ith ISAACA
int iredfrange[MAXISAACNUM];                      // red fire range of ith ISAACA
int irsrange_squad[MAXSQUADNUM];                  // red sensor range of the ith squad
int iredfrange_squad[MAXSQUADNUM];                // red fire range of the ith squad
float zshotbluebyreddef[MAXISAACNUM];             // probability that a red ISAAC shoots a blue
float zshotbluebyreddef_squad[MAXSQUADNUM];       // probability that a red ISAAC shoots a blue
int iperred;                                      // input flag for initial personality type
float w1rdeff_a[MAXSQUADNUM];                     // default weight for alive red --> alive red
float w2rdeff_a[MAXSQUADNUM];                     // default weight for alive red --> alive blue
float w3rdeff_a[MAXSQUADNUM];                     // default weight for alive red --> injured red
float w4rdeff_a[MAXSQUADNUM];                     // default weight for alive red --> injured blue
float w5rdeff_a[MAXSQUADNUM];                     // default weight for alive red --> red goal
float w6rdeff_a[MAXSQUADNUM];                     // default weight for alive red --> blue goal
float w1rdeff_i[MAXSQUADNUM];                     // default weight for injured red --> alive red
float w2rdeff_i[MAXSQUADNUM];                     // default weight for injured red --> alive blue
float w3rdeff_i[MAXSQUADNUM];                     // default weight for injured red --> injured red
float w4rdeff_i[MAXSQUADNUM];                     // default weight for injured red --> injured blue
float w5rdeff_i[MAXSQUADNUM];                     // default weight for injured red --> red goal
float w6rdeff_i[MAXSQUADNUM];                     // default weight for injured red --> blue goal
float red_w_a_max[MAXISAACNUM];                   // maximum absolute value of default red alive weights
float red_w_i_max[MAXISAACNUM];                   // maximum absolute value of default red injrd weights
float w1reddef_a[MAXISAACNUM];                    // default weight for alive red --> alive red
float w2reddef_a[MAXISAACNUM];                    // default weight for alive red --> alive blue
float w3reddef_a[MAXISAACNUM];                    // default weight for alive red --> injured red
float w4reddef_a[MAXISAACNUM];                    // default weight for alive red --> injured blue
float w5reddef_a[MAXISAACNUM];                    // default weight for alive red --> red goal
float w6reddef_a[MAXISAACNUM];                    // default weight for alive red --> blue goal
float w1reddef_i[MAXISAACNUM];                    // default weight for injrd red --> alive red
float w2reddef_i[MAXISAACNUM];                    // default weight for injrd red --> alive blue
float w3reddef_i[MAXISAACNUM];                    // default weight for injrd red --> injured red
float w4reddef_i[MAXISAACNUM];                    // default weight for injrd red --> injured blue
float w5reddef_i[MAXISAACNUM];                    // default weight for injrd red --> red goal
float w6reddef_i[MAXISAACNUM];                    // default weight for injrd red --> blue goal
```

```
int iredmovecont;                          // red movement constraint flag
int iradv_a_squad[MAXSQUADNUM];            // ith squad's red advance threshold
int iradv_i_squad[MAXSQUADNUM];            // ith squad's injured red advance threshold
int iradvrange_squad[MAXSQUADNUM];         // ith squad's range in which red # > threshold to advance
int irclus_a_squad[MAXSQUADNUM];           // ith squad's alive red cluster threshold
int irclus_i_squad[MAXSQUADNUM];           // ith squad's injured red cluster threshold
int ircom_a_squad[MAXSQUADNUM];            // ith squad's alive red combat threshold
int ircom_i_squad[MAXSQUADNUM];            // ith squad's injured red combat threshold
int iradv_a[MAXISAACNUM];                  // alive red advance threshold
int iradv_i[MAXISAACNUM];                  // injured red advance threshold
int iradvrange[MAXISAACNUM];               // range in which red # > threshold to advance
int irclus_a[MAXISAACNUM];                 // alive red cluster threshold
int irclus_i[MAXISAACNUM];                 // injured red cluster threshold
int ircom_a[MAXISAACNUM];                  // alive red combat threshold
int ircom_i[MAXISAACNUM];                  // injured red combat threshold
int iradvrange_min;                        // min red advance threshold for random constraints
int iradvrange_max;                        // max red advance threshold for random constraints
int iradv_a_min;                           // min alive red advance threshold for ran constraints
int iradv_a_max;                           // max alive red advance threshold for ran constraints
int iradv_i_min;                           // min injrd red advance threshold for ran constraints
int iradv_i_max;                           // max injrd red advance threshold for ran constraints
int irclus_a_min;                          // min alive red cluster threshold for ran constraints
int irclus_a_max;                          // max alive red cluster threshold for ran constraints
int irclus_i_min;                          // min injrd red cluster threshold for ran constraints
int irclus_i_max;                          // max injrd red cluster threshold for ran constraints
int ircom_a_min;                           // min alive red combat threshold for ran constraints
int ircom_a_max;                           // max alive red combat threshold for ran constraints
int ircom_i_min;                           // min injrd red combat threshold for ran constraints
int ircom_i_max;                           // max injrd red combat threshold for ran constraints
float zrfromrmindist_a_squad[MAXSQUADNUM]; // ith squad's minimum distance of alive red from red
float zrfromrgmindist_a_squad[MAXSQUADNUM]; // ith squad's minimum distance of alive red from red goal
float zbfromrmindist_a_squad[MAXSQUADNUM]; // ith squad's minimum distance of alive blue from red
float zrfromrmindist_i_squad[MAXSQUADNUM]; // ith squad's minimum distance of injured red from red
float zrfromrgmindist_i_squad[MAXSQUADNUM]; // ith squad's minimum distance of injured red from red goal
float zbfromrmindist_i_squad[MAXSQUADNUM]; // ith squad's minimum distance of injured blue from red
float zrfromrmindist_a[MAXISAACNUM];       // minimum distance of alive red from red
float zrfromrgmindist_a[MAXISAACNUM];      // minimum distance of alive red from red goal
float zbfromrmindist_a[MAXISAACNUM];       // minimum distance of alive blue from red
float zrfromrmindist_i[MAXISAACNUM];       // minimum distance of injured red from red
float zrfromrgmindist_i[MAXISAACNUM];      // minimum distance of injured red from red goal
float zbfromrmindist_i[MAXISAACNUM];       // minimum distance of injured blue from red
int iredmoverange_squad[MAXSQUADNUM];      // max movement radius for alive reds
int iredmoverange[MAXISAACNUM];            // max movement radius for alive reds
int red_max_eng_num[MAXISAACNUM];          // max # of simul engagements by red
int red_max_eng_num_squad[MAXSQUADNUM];    // max # of simul engagements by red
int red_COMM_flag;                         // if = 0 then COMMs NOT used for red, else yes
int ircommrange;                           // red communications range
float rcommweight;                         // red COMM weight (relative to w=1)
float rcommweight_def;                     // red default COMM weight
float zrsscale[MAXISAACNUM];               // scale factor for multiplying red penalty
int red_clock[MAXISAACNUM];                // internal red clock (for reconstitution)
int red_max_r_time;                        // maximum number of 'ticks' before reconstitution
int defense_flag;                          // =1 if defense "clock" (i.e. strength) to be used, else =0
int alive_defense_squad[MAXSQUADNUM];      // internal red clock (for defense) for ith squad
int injrd_defense_squad[MAXSQUADNUM];      // internal red clock (for defense) for ith squad
int alive_defense[MAXISAACNUM];            // internal red clock (for defense)
int injrd_defense[MAXISAACNUM];            // internal red clock (for defense)
int defense_clock[MAXISAACNUM];            // internal red clock (for defense)
};

struct blue_parameters
{
int num_blue_squads;                       // number of blue squads
int num_per_blue_squad[MAXSQUADNUM];       // number of ISAACAs in the ith blue squad
int squad[MAXISAACNUM];                    // number of the squad to which the ith ISAACA belongs
int display_blue_squad;                    // which blue squad to show the parameters of on screen
short bluecolor;                           // color code for alive blues
short squadcolor;                          // color code for highlighting individual squads
int bluegoalx;                             // x coordinate of blue goal
int bluegoaly;                             // y coordinate of blue goal
int bluex[MAXISAACNUM];                    // x-coordinate of ith blue ISAAC
int bluey[MAXISAACNUM];                    // y-coordinate of ith blue ISAAC
int bseer[MAXISAACNUM];                    // =1 if blue sees red and =0 otherwise
int bseeb[MAXISAACNUM];                    // =1 if blue sees blue and =0 otherwise
```

```
int bseercomm[MAXISAACNUM];                  // =1 if blue sees red via COMM link
int bseebcomm[MAXISAACNUM];                  // =1 if blue sees blue via COMM link
int bstatus[MAXISAACNUM];                    // =1 if alive, 1 if injured, 0 if dead
int irinb[MAXISAACNUM];                      // number of red isaacs in blue isaac range
float w1blue[MAXISAACNUM];                   // active weight for blue --> alive blue
float w2blue[MAXISAACNUM];                   // active weight for blue --> alive red
float w3blue[MAXISAACNUM];                   // active weight for blue --> injured blue
float w4blue[MAXISAACNUM];                   // active weight for blue --> injured red
float w5blue[MAXISAACNUM];                   // active weight for blue --> blue goal
float w6blue[MAXISAACNUM];                   // active weight for blue --> red goal
int ibluenum;                                // total number of blue ISAACs
int ibsrange[MAXISAACNUM];                   // blue sensor range of ith ISAACA
int ibluefrange[MAXISAACNUM];                // blue fire range of ith ISAACA
int ibsrange_squad[MAXSQUADNUM];             // red sensor range of the ith squad
int ibluefrange_squad[MAXSQUADNUM];          // red fire range of the ith squad
float zshotredbybluedef[MAXISAACNUM];        // probability that a blue ISAAC shoots a red
float zshotredbybluedef_squad[MAXSQUADNUM];  // probability that a blue ISAAC shoots a red
int iperblue;                                // input flag for initial personality type
float w1bdeff_a[MAXSQUADNUM];                // default weight for alive blue --> alive blue
float w2bdeff_a[MAXSQUADNUM];                // default weight for alive blue --> alive red
float w3bdeff_a[MAXSQUADNUM];                // default weight for alive blue --> injured blue
float w4bdeff_a[MAXSQUADNUM];                // default weight for alive blue --> injured red
float w5bdeff_a[MAXSQUADNUM];                // default weight for alive blue --> blue goal
float w6bdeff_a[MAXSQUADNUM];                // default weight for alive blue --> red goal
float w1bdeff_i[MAXSQUADNUM];                // default weight for injured blue --> alive blue
float w2bdeff_i[MAXSQUADNUM];                // default weight for injured blue --> alive red
float w3bdeff_i[MAXSQUADNUM];                // default weight for injured blue --> injured blue
float w4bdeff_i[MAXSQUADNUM];                // default weight for injured blue --> injured red
float w5bdeff_i[MAXSQUADNUM];                // default weight for injured blue --> blue goal
float w6bdeff_i[MAXSQUADNUM];                // def weight vector for injured blue --> red goal
float blue_w_a_max[MAXISAACNUM];             // max absolute value of default blue alive weights
float blue_w_i_max[MAXISAACNUM];             // max absolute value of default blue injurd weights
float w1bluedef_a[MAXISAACNUM];              // default weight for alive blue --> alive blue
float w2bluedef_a[MAXISAACNUM];              // default weight for alive blue --> alive red
float w3bluedef_a[MAXISAACNUM];              // default weight for alive blue --> injured blue
float w4bluedef_a[MAXISAACNUM];              // default weight for alive blue --> injured red
float w5bluedef_a[MAXISAACNUM];              // default weight for alive blue --> blue goal
float w6bluedef_a[MAXISAACNUM];              // default weight for alive blue --> red goal
float w1bluedef_i[MAXISAACNUM];              // default weight for injrd blue --> alive blue
float w2bluedef_i[MAXISAACNUM];              // default weight for injrd blue --> alive red
float w3bluedef_i[MAXISAACNUM];              // default weight for injrd blue --> injured blue
float w4bluedef_i[MAXISAACNUM];              // default weight for injrd blue --> injured red
float w5bluedef_i[MAXISAACNUM];              // default weight for injrd blue --> blue goal
float w6bluedef_i[MAXISAACNUM];              // default weight for injrd blue --> red goal
int ibluemovecont;                           // blue movement constraint flag
int ibadv_a_squad[MAXSQUADNUM];              // ith squad's blue advance threshold
int ibadv_i_squad[MAXSQUADNUM];              // ith squad's injured blue advance threshold
int ibadvrange_squad[MAXSQUADNUM];           // ith squad's range in which blue # > threshold to advance
int ibclus_a_squad[MAXSQUADNUM];             // ith squad's alive blue cluster threshold
int ibclus_i_squad[MAXSQUADNUM];             // ith squad's injured blue cluster threshold
int ibcom_a_squad[MAXSQUADNUM];              // ith squad's alive blue combat threshold
int ibcom_i_squad[MAXSQUADNUM];              // ith squad's injured blue combat threshold
int ibadv_a[MAXISAACNUM];                    // alive blue advance threshold
int ibadv_i[MAXISAACNUM];                    // injured blue advance threshold
int ibadvrange[MAXISAACNUM];                 // range within which blue # > threshold to advance
int ibclus_a[MAXISAACNUM];                   // alive blue cluster threshold
int ibclus_i[MAXISAACNUM];                   // injured blue cluster threshold
int ibcom_a[MAXISAACNUM];                    // alive blue combat threshold
int ibcom_i[MAXISAACNUM];                    // injured blue combat threshold
int ibadvrange_min;                          // min blue advance threshold for random constraints
int ibadvrange_max;                          // max blue advance threshold for random constraints
int ibadv_a_min;                             // min alive blue advance threshold for ran constrnts
int ibadv_a_max;                             // max alive blue advance threshold for ran constrnts
int ibadv_i_min;                             // min injrd blue advance threshold for ran constrnts
int ibadv_i_max;                             // max injrd blue advance threshold for ran constrnts
int ibclus_a_min;                            // min alive blue cluster threshold for ran constrnts
int ibclus_a_max;                            // max alive blue cluster threshold for ran constrnts
int ibclus_i_min;                            // min injrd blue cluster threshold for ran constrnts
int ibclus_i_max;                            // max injrd blue cluster threshold for ran constrnts
int ibcom_a_min;                             // min alive blue combat threshold for ran constrnts
int ibcom_a_max;                             // max alive blue combat threshold for ran constrnts
int ibcom_i_min;                             // min injrd blue combat threshold for ran constrnts
int ibcom_i_max;                             // max injrd blue combat threshold for ran constrnts
float zbfrombmindist_a_squad[MAXSQUADNUM];   // ith squad's minimum distance of alive blue from blue
```

```
float zbfrombgmindist_a_squad[MAXSQUADNUM];   // ith squad's minimum distance of alive blue from blue goal
float zrfrombmindist_a_squad[MAXSQUADNUM];    // ith squad's minimum distance of alive red from blue
float zbfrombmindist_i_squad[MAXSQUADNUM];    // ith squad's minimum distance of injured blue from blue
float zbfrombgmindist_i_squad[MAXSQUADNUM];   // ith squad's minimum distance of injured blue from blue goal
float zrfrombmindist_i_squad[MAXSQUADNUM];    // ith squad's minimum distance of injured red from blue
float zbfrombmindist_a[MAXISAACNUM];          // minimum distance of alive blue from blue
float zbfrombgmindist_a[MAXISAACNUM];         // minimum distance of alive blue from blue goal
float zrfrombmindist_a[MAXISAACNUM];          // minimum distance of alive red from blue
float zbfrombmindist_i[MAXISAACNUM];          // minimum distance of injured blue from blue
float zbfrombgmindist_i[MAXISAACNUM];         // minimum distance of injured blue from blue goal
float zrfrombmindist_i[MAXISAACNUM];          // minimum distance of injured red from blue
int ibluemoverange_squad[MAXSQUADNUM];        // max movement radius for alive blues
int ibluemoverange[MAXISAACNUM];              // max movement radius for alive blues
int blue_max_eng_num[MAXISAACNUM];            // max # of simul engagements by blue
int blue_max_eng_num_squad[MAXSQUADNUM];      // max # of simul engagements by blue
int blue_COMM_flag;                           // if = 0 then COMMs NOT used for blue, else yes
int ibcommrange;                              // blue communications range
float bcommweight;                            // blue COMM weight (relative to w=1)
float bcommweight_def;                        // blue default COMM weight
float zbsscale[MAXISAACNUM];                  // scale factor for multiplying blue penalty
int blue_clock[MAXISAACNUM];                  // internal blue clock (for reconstitution)
int blue_max_r_time;                          // maximum number of 'ticks' before reconstitution
int defense_flag;                             // =1 if defense "clock" (i.e. strength) to be used, else =0
int alive_defense_squad[MAXSQUADNUM];         // internal blue clock (for defense) for ith squad
int injrd_defense_squad[MAXSQUADNUM];         // internal blue clock (for defense) for ith squad
int alive_defense[MAXISAACNUM];               // internal blue clock (for defense)
int injrd_defense[MAXISAACNUM];               // internal blue clock (for defense)
int defense_clock[MAXISAACNUM];               // internal blue clock (for defense)
};
```

Main Module

```
//*****************************************************************************************************
//
// ISAAC.C - Irreducible Semi-Autonomous Adaptive Combatant
//
// Core engine (MS Visual C++ v1.52)
// Version 1.8.4
//
// Andy Ilachinski
// Center for Naval Analyses
// 4401 Ford Avenue
// Alexandria, VA 22302
// (703) 824-2045
// ilachina@cna.org
//
//*****************************************************************************************************

//*****************************************************************************************************
// ISAAC_A.C contains main() function
//
// All other function definitions appear in auxiliary files:
//
//      FILE            FUNCTION                            DESCRIPTION
//
//   ISAAC_B1.C  INITIALIZE_FIELD            initialize battlefield parameters
//   ISAAC_B2.C  SCREENDATA                  dump data to screen
//
//   ISAAC_C.C   ADAPT_RED_ISAACA_WEIGHTS    adapts red ISAACA weights
//   ISAAC_C.C   ADAPT_BLUE_ISAACA_WEIGHTS   adapts blue ISAACA weight
//   ISAAC_C.C   ADAPT_RED_LC_WEIGHTS        adapts red local commander weights
//   ISAAC_C.C   ADAPT_BLUE_LC_WEIGHTS       adapts blue local commander weights
//   ISAAC_C.C   ADAPT_RED_GC_WEIGHTS        adapts red global commander weights
//   ISAAC_C.C   ADAPT_BLUE_GC_WEIGHTS       adapts blue global commander weights
//
//   ISAAC_D.C   RED_SENSOR                  determines what the ith red ISAACA sees within sensor
//   ISAAC_D.C   BLUE_SENSOR                 determines what the ith blue ISAACA sees within sensor
//   ISAAC_D.C   RED_COMMAND_SENSOR          determines what the ith red local commander sees
//   ISAAC_D.C   BLUE_COMMAND_SENSOR         determines what the ith blue local commander sees
//   ISAAC_D.C   BLUEINRED                   determines number of blues within red sensor
//   ISAAC_D.C   REDINBLUE                   determines number of reds within blue sensor
//
//   ISAAC_E1.C  COMPUTEREDPENALTY           calculates penalty for each red move possibility
//   ISAAC_E1.C  COMPUTEBLUEPENALTY          calculates penalty for each blue move possibility
//   ISAAC_E2.C  COMPUTEREDPENALTY_GC        calculates penalty for red assuming GC flag on
//   ISAAC_E3.C  COMPUTEBLUEPENALTY_GC       calculates penalty for blue assuming GC flag on
//
//   ISAAC_F.C   COMPUTEREDPENALTY_COMM      calculate red-move penalty assuming COMM is 'on'
//   ISAAC_F.C   CCOMPUTEBLUEPENALTY_COMM    calculate blue-move penalty assuming COMM is 'on'
//
//   ISAAC_G.C   MOVERED                     moves all red ISAACAs (updates lattice positions)
//   ISAAC_G.C   MOVEBLUE                    moves all blue ISAACAs (updates lattice positions)
//
//   ISAAC_H1.C  RED_COMM_INFO               determine what ISAACAs are within red's COMM range
//   ISAAC_H1.C  BLUE_COMM_INFO              determine what ISAACAs are within blue's COMM range
//   ISAAC_H2.C  RED_PROMOTIONS              adjudicate red local commander promotions
//   ISAAC_H2.C  BLUE_PROMOTIONS             adjudicate blue local commander promotions
//
//   ISAAC_I.C   COMBAT                      adjudicates combat (assuming ALL enagements)
//   ISAAC_I.C   COMBAT_2                    adjudicates combat (assuming enagement threshold set)
//
//   ISAAC_J.C   RED_LOCAL_COMMAND_1         red local commanders set local goals for 3-by-3 patch
//   ISAAC_J.C   BLUE_LOCAL_COMMAND_1        blue local commanders set local goals for 3-by-3 patch
//
//   ISAAC_K1.C  RED_LOCAL_COMMAND_2         red local commanders set local goals for 5-by-5 patch
//   ISAAC_K2.C  BLUE_LOCAL_COMMAND_2        blue local commanders set local goals for 5-by-5 patch
//
//   ISAAC_L.C   UPDATEPICTURE               update graphics screen ith new red and blue positions
//
//   ISAAC_M1.C  INPUT_FILE_DATA             read input from data file
//   ISAAC_M2.C  INPUT_SCREEN_DATA           input data from screen prompts
//   ISAAC_M3.C  WRITE_DATA_FILE             Write current parameter values to data file
//   ISAAC_M3.C  WRITE_OUT_FILE              Open 'play-back' (*.out) file
//   ISAAC_M3.C  WRITE_INTERPOINT            Write interpoint-distances to statistics file
//   ISAAC_M3.C  WRITE_1_CLUSTER             Write cluster distributions to statistics file (using D=1
//   ISAAC_M3.C  WRITE_2_CLUSTER             Write cluster distributions to statistics file (using D=2
//   ISAAC_M3.C  WRITE_RR_NEIGHBORS          Write red-in-red distirbutions to statistics file
//   ISAAC_M3.C  WRITE_BB_NEIGHBORS          Write blue-in-blue distirbutions to statistics file
//   ISAAC_M3.C  WRITE_RB_NEIGHBORS          Write red-in-blue distirbutions to statistics file
//   ISAAC_M3.C  WRITE_BR_NEIGHBORS          Write blue-in-red distirbutions to statistics file
//   ISAAC_M3.C  WRITE_AR_NEIGHBORS          Write all-in-red distirbutions to statistics file
//   ISAAC_M3.C  WRITE_AB_NEIGHBORS          Write all-in-blue distirbutions to statistics file
//
//   ISAAC_N1.C  PROMPT_SCREEN               display choices for 'on-the-fly' parameter changes
//   ISAAC_N2.C  (Miscellaneous functions)   menu structures, etc. for PROMPT_SCREEN
//
//   ISAAC_O.C   PLAYBACK                    "plays-back" previously recorded //.out files
//
//   ISAAC_P.C   ABS_FLOAT                   returns absolute value of a float
```

```
//   ISAAC_P.C   GETRANDOM                 get a random number between a and b
//   ISAAC_P.C   RAN1                      uniform random generator from 'Numerical Recipes'
//   ISAAC_P.C   SIGNUM                    sign (+1,-1, or 0) of a float
//   ISAAC_P.C   NOMEM                     returns 'insufficient memory to run' message and exits
//
//   ISAAC_Q.C   RED_SWATH_AREA            calculates the area of each of 16 'swaths' centered
//                                         at the current (x,y) coordinates of red local commander
//   ISAAC_Q.C   BLUE_SWATH_AREA           calculates the area of each of 16 'swaths' centered
//                                         at the current (x,y) coordinates of blue local commander
//
//   ISAAC_R1.C  RED_SWATH_DENSITY         calculates the density of blue ISAACAs in each of 16
//                                         swath' centered at the current (x,y) coordinates of
//                                         red local commander
//   ISAAC_R2.C  BLUE_SWATH_DENSITY        calculates the area of of red ISAACAs in each of 16
//                                         swaths centered at the current (x,y) coordinates of
//                                         blue local commander
//
//   ISAAC_S1.C  RED_GLOBAL_COMMAND        red global commanders set 'direction' goals for LCs
//   ISAAC_S2.C  BLUE_GLOBAL_COMMAND       blue global commanders set 'direction' goals for LCs
//
//   ISAAC_T1.C  INTERPOINT_DIST           calculates R-R, B-B, R-B and R,B-goal distance dists
//   ISAAC_T2.C  SPATIAL_ENTROPY           computes spatial entropy for 4x4, 8x8 and 16x16 blocks
//   ISAAC_T3.C  CLUSTER_1                 returns the distribution of clusters (D=1) and average size
//   ISAAC_T3.C  MOMENT                    returns mean ave, ave deviation and standard deviation
//   ISAAC_T4.C  CLUSTER_2                 returns the distribution of clusters (D=2) and average size
//   ISAAC_T5.C  NEIGHBORS                 returns the average number of ISAACAs at distance D
//   ISAAC_T5.C  CENTER_MASS               returns the center-of-mass of red, blue and total forces
//   ISAAC_T5.C  GOAL_STATS                returns the number of ISAACAs near enemy flag
//
//*************************************************************************************************

#include <isaac18.h>     // contains MAXIMUM sizes for battlefield and ISAACA arrays
#include <istrc18.h>     // contains COMMAND, BATTLE, RED and BLUE parameter structures
#include <string.h>
#include <float.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
#include <graph.h>
#include <io.h>
#include <malloc.h>

//*************************************************************************************************
//
//                              Allocate Memory for Structures
//
//*************************************************************************************************
    struct red_GC_parameters red_GC;
    struct blue_GC_parameters blue_GC;
    struct red_command_parameters red_command;
    struct blue_command_parameters blue_command;
    struct battle_parameters battle;
    struct red_parameters red;
    struct blue_parameters blue;
    struct statistics stats;

//*************************************************************************************************
//
//
//                                    FUNCTION PROTOTYPES
//
//
//*************************************************************************************************
void INPUT_FILE_DATA(unsigned char filename[30], struct statistics *ss, struct red_GC_parameters *rgcp,
                     struct blue_GC_parameters *bgcp, struct red_command_parameters *rcomp,
                     struct blue_command_parameters *bcomp, struct battle_parameters *batp,
                     struct red_parameters *redp, struct blue_parameters *bluep, long *idum);

void INPUT_SCREEN_DATA(struct red_GC_parameters *rgcp, struct blue_GC_parameters *bgcp,
                       struct red_command_parameters *rcomp, struct blue_command_parameters *bcomp,
                       struct battle_parameters *batp, struct red_parameters *redp,
                       struct blue_parameters *bluep, long *idum, int *iternum);

void WRITE_DATA_FILE(FILE *outdatafile, struct statistics *s, struct red_GC_parameters *rgcp,
                     struct blue_GC_parameters *bgcp, struct red_command_parameters *rcomp,
                     struct blue_command_parameters *bcomp, struct battle_parameters *batp,
                     struct red_parameters *redp, struct blue_parameters *bluep);

void WRITE_OUT_FILE(FILE *outdatafile, int red_GC_flag, int blue_GC_flag,
                    struct red_command_parameters *rcomp, struct blue_command_parameters *bcomp,
                    struct battle_parameters *batp, struct red_parameters *redp,
                    struct blue_parameters *bluep);

void INITIALIZE_FIELD(struct statistics *s, struct red_GC_parameters *rgcp,
                      struct blue_GC_parameters *bgcp, struct red_command_parameters *rcomp,
                      struct blue_command_parameters *bcomp, struct battle_parameters *batp,
                      struct red_parameters *redp, struct blue_parameters *bluep,
                      int iflag[5][5], long *idum);

void SCREENDATA(int itime, int idata, unsigned char filename[30], struct red_command_parameters *rcomp,
                struct blue_command_parameters *bcomp, struct battle_parameters *batp,
                struct red_parameters *redp, struct blue_parameters *bluep);

void ADAPT_RED_ISAACA_WEIGHTS(int i, int irinrnum, int ibinrnum, int iradvnum,
                              struct red_command_parameters *rcomp, struct red_parameters *redp);
```

```
void ADAPT_BLUE_ISAACA_WEIGHTS(int i, int ibinbnum, int irinbnum, int ibadvnum,
                               struct blue_command_parameters *bcomp, struct blue_parameters *bluep);

void ADAPT_RED_LC_WEIGHTS(int i, int irinrnum, int ibinrnum, int iradvnum,
                          struct red_command_parameters *rcomp,
                          struct red_parameters *redp);

void ADAPT_BLUE_LC_WEIGHTS(int i, int ibinbnum, int irinbnum, int ibadvnum,
                           struct blue_command_parameters *bcomp,
                           struct blue_parameters *bluep);

void ADAPT_RED_GC_WEIGHTS(int i, int irinrnum, int ibinrnum, int iradvnum,
                          struct red_GC_parameters *rgcp,
                          struct red_command_parameters *rcomp,
                          struct red_parameters *redp);

void ADAPT_BLUE_GC_WEIGHTS(int i, int irinrnum, int ibinrnum, int iradvnum,
                          struct blue_GC_parameters *bgcp,
                          struct blue_command_parameters *bcomp,
                          struct blue_parameters *bluep);

void ADAPTBLUEWEIGHTS(int i, int ibinbnum, int irinbnum, int ibadvnum,
                      struct blue_command_parameters *bcomp, struct blue_parameters *bluep);

void RED_COMM_INFO(int i, struct red_parameters *redp, struct blue_parameters *bluep);

void BLUE_COMM_INFO(int i, struct red_parameters *redp, struct blue_parameters *bluep);

void MOVERED (int i, int imrr, int imove, struct battle_parameters *batp,
              struct red_parameters *redp);

void MOVEBLUE (int i, int imbr, int imove, struct battle_parameters *batp,
               struct blue_parameters *bluep);

void RED_SENSOR(int *irinrnum, int *ibinrnum, int *iradvnum, int *ibinrinjnum,
                int *irinrinjnum, int i, struct red_parameters *redp, struct blue_parameters *bluep,
                int **ilblbinr);

void RED_COMMAND_SENSOR(int *irinrnum, int *ibinrnum, int *iradvnum, int *ibinrinjnum,
                        int *irinrinjnum, int i, struct red_command_parameters *rcomp,
                        struct red_parameters *redp, struct blue_parameters *bluep, int **ilblbinr);

int BLUEINRED( int i, struct red_parameters *redp, struct blue_parameters *bluep,
               int **ilblbinr );

void BLUE_SENSOR(int *ibinbnum, int *irinbnum, int *ibadvnum, int *irinbinjnum,
                 int *ibinbinjnum, int i, struct red_parameters *redp,
                 struct blue_parameters *bluep, int **ilblrinb);

void BLUE_COMMAND_SENSOR(int *ibinbnum, int *irinbnum, int *ibadvnum, int *irinbinjnum,
                         int *ibinbinjnum, int i, struct blue_command_parameters *bcomp,
                         struct red_parameters *redp, struct blue_parameters *bluep, int **ilblrinb);

int REDINBLUE( int i, struct red_parameters *redp, struct blue_parameters *bluep,
               int **ilblrinb);

float COMPUTEREDPENALTY(int i, int imrr, int *igoalflag, int irinrinjnum, int ibinrinjnum,
                        float zmin, int iflag[5][5], float z[5][5], struct red_command_parameters *rcomp
                        struct blue_command_parameters *bcomp, struct battle_parameters *batp,
                        struct red_parameters *redp, struct blue_parameters *bluep);

float COMPUTEREDPENALTY_COMM(int i, int imrr, int *igoalflag, int irinrinjnum,
                             int ibinrinjnum, float zmin, int iflag[5][5], float z[5][5],
                             struct red_command_parameters *rcomp, struct blue_command_parameters *bcomp,
                             struct battle_parameters *batp, struct red_parameters *redp,
                             struct blue_parameters *bluep);

float COMPUTEREDPENALTY_GC(int i, int imrr, int *igoalflag, int irinrinjnum,
                           int ibinrinjnum, float zmin, int iflag[5][5], float z[5][5],
                           struct red_GC_parameters *rgcp, struct blue_GC_parameters *bgcp,
                           struct red_command_parameters *rcomp, struct blue_command_parameters *bcomp,
                           struct battle_parameters *batp, struct red_parameters *redp,
                           struct blue_parameters *bluep);

float COMPUTEBLUEPENALTY(int i, int imbr, int *igoalflag, int irinbinjnum, int ibinbinjnum,
                         float zmin, int iflag[5][5], float z[5][5], struct red_command_parameters *rcomp
                         struct blue_command_parameters *bcomp, struct battle_parameters *batp,
                         struct red_parameters *redp, struct blue_parameters *bluep);

float COMPUTEBLUEPENALTY_COMM(int i, int imbr, int *igoalflag, int irinbinjnum,
                              int ibinbinjnum, float zmin, int iflag[5][5], float z[5][5],
                              struct red_command_parameters *rcomp, struct blue_command_parameters *bcomp,
                              struct battle_parameters *batp, struct red_parameters *redp,
                              struct blue_parameters *bluep);

float COMPUTEBLUEPENALTY_GC(int i, int imbr, int *igoalflag, int irinbinjnum, int ibinbinjnum,
                            float zmin, int iflag[5][5], float z[5][5], struct red_GC_parameters *rgcp,
                            struct blue_GC_parameters *bgcp, struct red_command_parameters *rcomp,
                            struct blue_command_parameters *bcomp, struct battle_parameters *batp,
                            struct red_parameters *redp, struct blue_parameters *bluep);

void COMBAT( struct battle_parameters *batp, struct red_parameters *redp,
             struct blue_parameters *bluep, long *idum, int **ilblbinr, int **ilblrinb);

void COMBAT_2( struct battle_parameters *batp, struct red_parameters *redp,
               struct blue_parameters *bluep, long *idum, int **ilblbinr, int **ilblrinb);
```

```
void UPDATEPICTURE(FILE *stat_data[22], int itime, int red_GC_flag, int blue_GC_flag,
                   struct red_command_parameters *rcomp, struct blue_command_parameters *bcomp,
                   struct battle_parameters *batp, struct red_parameters *redp,
                   struct blue_parameters *bluep, struct statistics *s);

int PROMPT_SCREEN(struct battle_parameters *batp, struct red_parameters *redp,
                  struct blue_parameters *bluep, struct red_GC_parameters *rgcp,
                  struct blue_GC_parameters *bgcp, struct red_command_parameters *rcomp,
                  struct blue_command_parameters *bcomp, struct statistics *s, long *idum);

void RED_LOCAL_COMMAND_1(int i_command, int itime, struct red_GC_parameters *rgcp,
                         struct red_command_parameters *rcomp, struct blue_command_parameters *bcomp,
                         struct battle_parameters *batp, struct red_parameters *redp,
                         struct blue_parameters *bluep, long *idum);

void BLUE_LOCAL_COMMAND_1(int i_command, int itime, struct blue_GC_parameters *bgcp,
                          struct red_command_parameters *rcomp, struct blue_command_parameters *bcomp,
                          struct battle_parameters *batp, struct red_parameters *redp,
                          struct blue_parameters *bluep, long *idum);

void RED_LOCAL_COMMAND_2(int i_command, int itime, struct red_command_parameters *rcomp,
                         struct blue_command_parameters *bcomp, struct battle_parameters *batp,
                         struct red_parameters *redp, struct blue_parameters *bluep, long *idum);

void BLUE_LOCAL_COMMAND_2(int i_command, int itime, struct red_command_parameters *rcomp,
                         struct blue_command_parameters *bcomp, struct battle_parameters *batp,
                         struct red_parameters *redp, struct blue_parameters *bluep, long *idum);

void RED_GLOBAL_COMMAND(struct red_GC_parameters *rgcp, struct red_command_parameters *rcomp,
                        struct red_parameters *redp, long *idum);

void BLUE_GLOBAL_COMMAND(struct blue_GC_parameters *bgcp, struct blue_command_parameters *bcomp,
                         struct blue_parameters *bluep, long *idum);

void RED_PROMOTIONS(struct red_command_parameters *rcomp, struct red_parameters *redp, long *idum);

void BLUE_PROMOTIONS(struct blue_command_parameters *bcomp, struct blue_parameters *bluep, long *idum);

void RED_SWATH_AREA(int isize, struct red_GC_parameters *rgcp,
                    struct red_command_parameters *rcomp, struct red_parameters *redp);

void BLUE_SWATH_AREA(int isize, struct blue_GC_parameters *bgcp,
                    struct blue_command_parameters *bcomp, struct blue_parameters *bluep);

void RED_SWATH_DENSITY(int isize, struct red_GC_parameters *rgcp,
                       struct red_command_parameters *rcomp, struct red_parameters *redp,
                       struct blue_parameters *bluep);

void BLUE_SWATH_DENSITY(int isize, struct blue_GC_parameters *bgcp,
                       struct blue_command_parameters *bcomp, struct red_parameters *redp,
                       struct blue_parameters *bluep);

unsigned char PLAYBACK(unsigned char plotfilename[30], int *idata, unsigned char filename[30],
                       int *ichoice);

void INTERPOINT_DIST(struct red_command_parameters *rcomp, struct blue_command_parameters *bcomp,
                     struct red_parameters *redp, struct blue_parameters *bluep, struct statistics *s);

void SPATIAL_ENTROPY(int isize, struct statistics *s, struct red_parameters *redp,
                     struct blue_parameters *bluep);

void CLUSTER_1(int isize, struct red_parameters *redp, struct blue_parameters *bluep,
               struct battle_parameters *batp, struct statistics *s);

void CLUSTER_2(int isize, struct red_parameters *redp, struct blue_parameters *bluep,
               struct battle_parameters *batp, struct statistics *s);

void NEIGHBORS(FILE *stat_data[22], int itime, struct red_parameters *redp, struct blue_parameters *bluep,
               struct statistics *s);

void CENTER_MASS(FILE *stat_data[22], int itime, struct red_parameters *redp,
                 struct blue_parameters *bluep, struct statistics *s);

void GOAL_STATS(FILE *stat_data[22], int itime, int isize, struct red_parameters *redp,
                struct blue_parameters *bluep, struct statistics *s);

void WRITE_INTERPOINT(int itime, struct statistics *s, FILE *stat_data[22]);

void WRITE_1_CLUSTER(int itime, struct statistics *s, FILE *stat_data[22]);

void WRITE_2_CLUSTER(int itime, struct statistics *s, FILE *stat_data[22]);

float abs_float(float x);

float getrandom(int x, int y, long *idum);

float ran1(long *idum);

void nomem();

//*********************************************************************************************
//
//                                    MAIN: RUN ISAAC
//
//*********************************************************************************************
void main()
```

```
{
   int i, ii, j;              // loop variables
   int imx, imy;              // loop variables
   int im, imc;               // labels for intermediate candidte moves
   int igoal;                 // termination flag; =1 --> red wins
   int igoalflag;             // if =1 then red wins; if =2 then blue wins
   int imove;                 // labels selected move (1 <= imove <= 9)
   int ibinrnum;              // number of blue ISAACs in red sensor range
   int irinbnum;              // number of red ISAACs in blue sensor range
   int irinrnum;              // number of red ISAACs in red sensor range
   int ibinbnum;              // number of blue ISAACs in blue sensor range
   int iradvnum;              // threshold number of reds to advance
   int ibadvnum;              // threshold number of blues to advance
   int irinrinjnum;           // number of injured red in red sensor
   int ibinbinjnum;           // number of injured blue in blue sensor
   int irinbinjnum;           // number of injured red in blue sensor
   int ibinrinjnum;           // number of injured blue in red sensor
   int playterm;              // return variable from PLAYBACK; 1=Quit
   int itime;                 // time counter
   int iternum;               // number of iterations to store in file
   int irun_choice;           // if 1 then continue, else new start
   int idata;                 // =1 = screen prompt; 2= read from datafile
   int imovecand[26];         // intermediate move candidates from which
                              // an ISAAC will choose an actual move
   int iflag[5][5];           // iflag=1 if a particular move represents a viable option
   int itrace;                // 1=trace "ON", else trace "OFF"
   int imrr;                  // = red.iredmoverange if alive, else = 1
   int imbr;                  // = blue.ibluemoverange if alive, else = 1
   int igraphtype;            // if =1 then VGA, if =2 then SVGA
   int save_flag;
   int jj, icount;
   int bluelabel_randomized[MAXISAACNUM];
   int redlabel_randomized[MAXISAACNUM];
   int **ilblrinb;    // jth red's (in blue's range) label
   int **ilblbinr;    // jth blue's (in red's range) label

   long idum;                 // random number seed (dummy 'carry-over' variable)

   float zmin;                // minimum seed used by penalty function
   float zmoveprob;
   float zran;                // variable to catch initial ran number call
   float z[5][5];             // intermediate expected penalty function

   unsigned char buffer, bufferq;
   unsigned char filename[30];      // name of input data file
   unsigned char outfilename[30];   // name of output file
   unsigned char snapshotfile[30];  // name of output file
   unsigned char plotfilename[30];  // name of plot file to be played-back
   unsigned char fondir[_MAX_PATH];
   unsigned char list[20];
   char bb[20];
   short xfon;

   FILE *outdatafile;
   FILE *stat_data[22];

   struct _fontinfo fi;
   struct _videoconfig vc;

//*****************************************************************************
//
//                         Allocate Memory for Matrices
//
//*****************************************************************************
    ilblrinb = (int**) _fmalloc( (MAXISAACNUM+1) * sizeof(int*) );
    if ( !ilblrinb ) nomem();
    for ( i = 0; i < (MAXISAACNUM+1); i++ )
    {
        ilblrinb[i] = (int*) _fmalloc( (MAXNEIGHBORNUM+1) * sizeof(int) );
        if ( !ilblrinb[i] ) nomem();
    }

    ilblbinr = (int**) _fmalloc( (MAXISAACNUM+1) * sizeof(int*) );
    if ( !ilblbinr ) nomem();
    for ( i = 0; i < (MAXISAACNUM+1); i++ )
    {
        ilblbinr[i] = (int*) _fmalloc( (MAXNEIGHBORNUM+1) * sizeof(int) );
        if ( !ilblbinr[i] ) nomem();
    }

//*****************************************************************************
//
//                           Register and Set Fonts
//
//*****************************************************************************
    if( _registerfonts( "sserife.FON" ) <= 0 )
    {
        _outtext( "Enter full path where .FON files are located: " );
        gets( fondir );
        strcat( fondir, "\\*.FON" );
        if( _registerfonts( fondir ) <= 0 )
        {
            _outtext( "Error: can't register fonts" );
            exit( 1 );
        }
    }
```

```
//*****************************************************************************
//
//                              Set graphics colors
//
//*****************************************************************************
    red_command.red_command_color = 14;
    blue_command.blue_command_color = 15;
    red.squadcolor = 14;
    blue.squadcolor = 15;
    red.redcolor = 12;
    blue.bluecolor = 9;
    battle.goalcolor = 10;
    battle.boxcolor = 3;
    blue_command.blue_subordinate_color_flag = 0;
    red_command.red_subordinate_color_flag = 0;

//*****************************************************************************
//
//                                Set Video Mode
//
//*****************************************************************************
        if ( !_setvideomode( _SRES16COLOR ) ){
             _setvideomode( _VRES16COLOR );
             _clearscreen( _GCLEARSCREEN );
        }

//*****************************************************************************
//
//                                Opening Screen
//
//*****************************************************************************
    _setbkcolor ( _BLUE );
    _clearscreen( _GCLEARSCREEN );
    _getvideoconfig( &vc );
    _setcolor( 15 );
    _moveto( 75, 80 );
    _rectangle_w( _GFILLINTERIOR, 100, 85, 700, 90 );
    _moveto( 565, 295 );
    _rectangle_w( _GFILLINTERIOR, 100, 270, 700, 275 );
    strcat( strcat( strcpy( list, "t'" ), "sserife"), "'");
    strcat( list, "h30w24b" );
    _getfontinfo( &fi );
    _setfont( list );
    _setcolor( 15 );
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "I S A A C" ) / 2);
    _moveto( xfon, 105 );
    _outgtext ("I S A A C");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Irreducible Semi-Autonomous" ) / 2);
    _moveto( xfon, 143 );
    _outgtext ("Irreducible Semi-Autonomous");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Adaptive Combat" ) / 2);
    _moveto( xfon, 173 );
    _outgtext ("Adaptive Combat");
    _unregisterfonts();
    _registerfonts( "oem10.FON" );
    strcat( strcat( strcpy( list, "t'" ), "oem10"), "'");
    strcat( list, "h30w24bv" );
    _setfont( list );
    _getfontinfo( &fi );
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Version 1.8.4" ) / 2);
    _moveto( xfon, 227 );
    _outgtext ("Version 1.8.4");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "10 April 1997" ) / 2);
    _moveto( xfon, 242 );
    _outgtext ("10 April 1997");
    _setcolor( 7 );
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Andy Ilachinski" ) / 2);
    _moveto( xfon, 300 );
    _outgtext ("Andy Ilachinski");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Center for Naval Analyses" ) / 2);
    _moveto( xfon, 315 );
    _outgtext ("Center for Naval Analyses");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "4401 Ford Avenue" ) / 2);
    _moveto( xfon, 330 );
    _outgtext ("4401 Ford Avenue");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Alexandria, VA 22302" ) / 2);
    _moveto( xfon, 345 );
    _outgtext ("Alexandria, VA 22302");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "ilachina@cna.org" ) / 2);
    _moveto( xfon, 360);
    _outgtext ("ilachina@cna.org");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Press <ENTER> to continue ....." ) / 2);
    _moveto( xfon, 550 );
    _outgtext ("Press <ENTER> to continue .....");

    _getch();

startagain:

    // set default trace to no trace
    itrace = 0;

    // initialize squad number to display on screen
    red.display_red_squad = 1;
    blue.display_blue_squad = 1;
    battle.squad_color_flag = 0; // default is to NOT highlight
```

```
    // initialize "playback" flag
    playterm = 0;

    // seed random number generator
    idum=-31415926;
    zran=ran1(&idum);

    if ( !_setvideomode( _SRES16COLOR ) ){
         _setvideomode( _VRES16COLOR );
         _clearscreen( _GCLEARSCREEN );
    }
    _setbkcolor ( _BLUE );
    _clearscreen( _GCLEARSCREEN );

    _unregisterfonts();
    _registerfonts( "sserife.FON" );
    strcat( strcat( strcpy( list, "t'" ), "sserife"), "'");
    strcat( list, "h30w24bv" );
    _getfontinfo( &fi );
    _setfont( list );
    xfon = (vc.numxpixels / 2) - (_getgtextextent( "SELECT RUN OPTION" ) / 2);
    _moveto( xfon, 55 );
    _outgtext ("SELECT RUN OPTION");
    _settextposition( 7, 35);
    printf("[1] Run ISAAC engine with new input");
    _settextposition( 8, 35);
    printf("[2] Playback old run");
    _settextposition( 9, 35);
    printf("[3] Quit");
    _settextposition( 11, 39);
    printf("? ");
    scanf("%i",&battle.ichoice);

    if (battle.ichoice == 3){
        exit(1);
    }

    if (battle.ichoice == 2){
       goto playback; // goto end of file
    }

    xfon = (vc.numxpixels / 2) - (_getgtextextent( "SPECIFY FORM OF INPUT" ) / 2);
    _moveto( xfon, 215 );
    _outgtext ("SPECIFY FORM OF INPUT");
    _settextposition( 17, 40);
    printf("[1] Prompt from screen                    ");
    _settextposition( 18, 40);
    printf("[2] Read from datafile                    ");
    _settextposition( 20, 44);
    printf("? ");
    scanf("%i",&idata);

    if (idata == 2){
       _settextposition( 22, 40);
       printf("File name ? ");
       scanf("%s", &filename);

read_data:

        //*******************************************************************************
        //
        //                          Read input from data file
        //
        //*******************************************************************************
        INPUT_FILE_DATA(filename, &stats, &red_GC, &blue_GC, &red_command, &blue_command,
                        &battle,  &red, &blue, &idum);

    }

    xfon = (vc.numxpixels / 2) - (_getgtextextent( "SPECIFY FORM OF OUTPUT" ) / 2);
    _moveto( xfon, 390 );
    _outgtext ("SPECIFY FORM OF OUTPUT");
    _settextposition( 28, 45);
    printf("[1] Terminal                                ");
    _settextposition( 29, 45);
    printf("[2] File                                    ");
    _settextposition( 30, 45);
    printf("[3] Both                                    ");
    _settextposition( 32, 49);
    printf("? ");
    scanf("%i",&battle.ioutdata);

    if (battle.ioutdata > 1){
       _settextposition( 34, 45);
       printf("File name ? ");
       scanf("%s", &outfilename);
    }

    if (idata == 2 && battle.itermcond == 2 && battle.ioutdata > 1){
        _settextposition( 35, 28 );
        printf("NUMBER OF ITERATIONS TO STORE IN OUTPUT FILE ? ");
        scanf("%i",&iternum);
    }
    else{
        iternum = 1000;
    }
```

```
    if (battle.ioutdata != 2){
        igraphtype = 2; // default graphics = SVGA
        if ( !_setvideomode( _SRES16COLOR ) ){
              _setvideomode( _VRES16COLOR );
              _clearscreen( _GCLEARSCREEN );
        }

    }

    _unregisterfonts();
    _registerfonts( "oem10.FON" );
    strcat( strcat( strcpy( list, "t'" ), "oem10"), "'");
    strcat( list, "h30w24bv" );
    _getfontinfo( &fi );
    _setfont( list );

   if (idata == 1){

        //*************************************************************************
        //
        //                      Prompt user for input data
        //
        //*************************************************************************
        //INPUT_SCREEN_DATA(&red_GC, &blue_GC, &red_command, &blue_command, &battle,
        //                  &red, &blue, &idum, &iternum);

        _clearscreen( _GCLEARSCREEN );

        _getvideoconfig( &vc );
        if (vc.numxpixels < 641) { // then use VGA values
            _settextposition( 10, 15);
        }
        else{
            _settextposition( 10, 25);
        }
        printf("SAVE PARAMETER VALUES TO DATA FILE (y=1/n=0) ? ");
        scanf("%i",&save_flag);

        if (save_flag == 1){
            //*****************************************************
            //
            //        Save current parameter set to data file
            //
            //*****************************************************
            if (vc.numxpixels < 641) {    // then use VGA values
                _settextposition( 12, 15);
            }
            else{
                _settextposition( 12, 25);
            }
            printf("     File name ? ");
            scanf("%s", &filename);

            if ( (outdatafile = fopen(filename, "wt")) == NULL ){
                  printf(" Cannot open data file.\n");
                  exit(1);
            }

            WRITE_DATA_FILE(outdatafile, &stats, &red_GC, &blue_GC, &red_command,
                            &blue_command, &battle, &red, &blue);
        }

   } // end if idata == 1

   //*********************************************************
   //
   //        if battle.ioutdata>1 then write data to file
   //
   //*********************************************************
   if (battle.ioutdata > 1){

        if ( (outdatafile = fopen(outfilename, "wt")) == NULL ){
            printf(" Cannot open data file.\n");
            exit(1);
        }

        WRITE_OUT_FILE(outdatafile, red_GC.red_GC_flag, blue_GC.blue_GC_flag,
                       &red_command, &blue_command, &battle, &red, &blue);
   }

   //*************************************************************************
   //
   //    start here if any options are changed on-the-fly using PROMPT_SCREEN()
   //
   //*************************************************************************

changeoption:

    _setbkcolor ( _BLACK );
    _clearscreen( _GCLEARSCREEN );

    //*************************************************************************
    //
    //                      Initialize combat battlefield
    //
    //*************************************************************************
```

```
    INITIALIZE_FIELD(&stats, &red_GC, &blue_GC, &red_command, &blue_command, &battle,
                     &red, &blue, iflag, &idum);

    //*************************************************************************
    //
    //                          Initialize time counter
    //
    //*************************************************************************
    itime = 0;

    //*************************************************************************
    //
    //               Dump data to screen and show initial configuration
    //
    //*************************************************************************
    if (battle.ioutdata != 2){
        SCREENDATA(itime, idata, filename, &red_command, &blue_command, &battle,
                   &red, &blue);
    }

    //*************************************************************************
    //
    //                       Are statistics to be tabulated?
    //
    //*************************************************************************
    if (stats.stat_flag == 1){
        _getvideoconfig( &vc );
        if ( (stat_data[1] = fopen("stats_1.dat", "wt")) == NULL ||
             (stat_data[2] = fopen("stats_2.dat", "wt")) == NULL ||
             (stat_data[3] = fopen("stats_3.dat", "wt")) == NULL ||
             (stat_data[4] = fopen("stats_4.dat", "wt")) == NULL ||
             (stat_data[5] = fopen("stats_5.dat", "wt")) == NULL ||
             (stat_data[6] = fopen("stats_6.dat", "wt")) == NULL ||
             (stat_data[7] = fopen("stats_7.dat", "wt")) == NULL ||
             (stat_data[8] = fopen("stats_8.dat", "wt")) == NULL ||
             (stat_data[9] = fopen("stats_9.dat", "wt")) == NULL ||
             (stat_data[10] = fopen("stats_10.dat", "wt")) == NULL ||
             (stat_data[11] = fopen("stats_11.dat", "wt")) == NULL ||
             (stat_data[12] = fopen("stats_12.dat", "wt")) == NULL ||
             (stat_data[13] = fopen("stats_13.dat", "wt")) == NULL){
               printf(" Cannot open data file.\n");
               exit(1);
        }
        if (vc.numxpixels < 641) { // then in VGA mode
            _setviewport(200,455,525,465);
            _clearscreen( _GVIEWPORT );
            _setviewport(1,1,639,479);
            _moveto( 205, 450 );
            _setcolor( 14 );
            _setgtextvector( 1, 0 );
            _outgtext("Statistics being calculated ... ");
            _setviewport(140,48,525,428);
        }
        else{
            _setviewport(260,575,680,590);
            _clearscreen( _GVIEWPORT );
            _setviewport(1,1,799,599);
            _moveto( 270, 577 );
            _setcolor( 14 );
            _setgtextvector( 1, 0 );
            _outgtext("Statistics being calculated ... ");
            _setviewport(120,50,680,550);
        }
    }

    ////////////////////////////////////////////////////////////////////////////////////////////////
    //
    //                                  START MAIN DYNAMICS LOOP
    //
    ////////////////////////////////////////////////////////////////////////////////////////////////

start:

    ++itime;  // increment time counter

     //*************************************************************************************************
     //
     //                              Calculate descriptive statistics
     //
     //*************************************************************************************************
     if (stats.stat_flag == 1){

         //
         // calculate ISAACA-ISAACA and ISAACA-goal interpoint distributions
         //
         if (stats.interpoint_flag == 1){
             INTERPOINT_DIST(&red_command, &blue_command, &red, &blue, &stats);
             WRITE_INTERPOINT(itime, &stats, stat_data);
         }

         //
         // calculate spatial entropy using different course-graining
         //
         if (stats.entropy_flag == 1){
             SPATIAL_ENTROPY(battle.isize, &stats, &red, &blue);
             fprintf(stat_data[9], "%3i  %5.4f  %5.4f  %5.4f  %5.4f  %5.4f %5.4f  %5.4f  %5.4f  %5.4f\n",
                     itime, stats.red_entropy_1, stats.blue_entropy_1, stats.red_blue_entropy_1,
```

```
                    stats.red_entropy_2, stats.blue_entropy_2, stats.red_blue_entropy_2,
                    stats.red_entropy_3, stats.blue_entropy_3, stats.red_blue_entropy_3);
        }

        //
        // find cluster size distribution (using D=1)
        //
        if (stats.cluster_1_flag==1){
            CLUSTER_1(battle.isize, &red, &blue, &battle, &stats);
            WRITE_1_CLUSTER(itime, &stats, stat_data);
        }

        //
        // find cluster size distribution (using D=2)
        //
        if (stats.cluster_2_flag==1){
            CLUSTER_2(battle.isize, &red, &blue, &battle, &stats);
            WRITE_2_CLUSTER(itime, &stats, stat_data);
        }

        //
        // find average number of neighboring ISAACAs at distance D <= R
        // (write to stats_14.dat, stats_15.dat,... , stats_19.dat)
        //
        if (stats.neighbors_flag == 1){
            NEIGHBORS(stat_data, itime, &red, &blue, &stats);
        }

        //
        // calculate ISAACA-ISAACA and ISAACA-goal interpoint distributions
        // (write to stats_20.dat)
        //
        if (stats.center_mass_flag == 1){
            CENTER_MASS(stat_data, itime, &red, &blue, &stats);
        }

        //
        // calculate goal statistics
        // (write to stats_21.dat)
        //
        if (stats.goal_stat_flag == 1){
            GOAL_STATS(stat_data, itime, battle.isize, &red, &blue, &stats);
        }
    }

    //*************************************************************************
    //
    //     Should order of move selection be shuffled during each iteration?
    //
    //*************************************************************************
     if (battle.imove_selection == 1){  // select moves in fixed order
        for (j=1; j<=red.irednum; ++j){
            redlabel_randomized[j] = j;
        }
     }
     else{
        //*********************************************************************
        //
        //      Randomize order in which to consider moves for red ISAACAs:
        //      'i' is the actual label and the array redlabel_randomized[j] = i
        //
        //*********************************************************************
        icount=0;
        for (j=1; j<=red.irednum; ++j){
            // select random label between 1 and red.irednum
newired:    i = (int)(getrandom( 0, red.irednum, &idum ))+1;
            // test to see if label has already been used
            for (jj=1; jj<=icount; ++jj){
                if (redlabel_randomized[jj] == i) goto newired;
            }
            ++icount;
            redlabel_randomized[j] = i;
        }
     }

     if (battle.imove_selection == 1){  // select moves in fixed order
        for (j=1; j<=blue.ibluenum; ++j){
            bluelabel_randomized[j] = j;
        }
     }
     else{
        //*********************************************************************
        //
        //      Randomize order in which to consider moves for blue ISAACAs:
        //      'i' is the actual label and the array bluelabel_randomized[j] = i
        //
        //*********************************************************************
        icount=0;
        for (j=1; j<=blue.ibluenum; ++j){
            // select random label between 1 and red.irednum
newiblue:   i = (int)(getrandom( 0, blue.ibluenum, &idum ))+1;
            // test to see if label has already been used
            for (jj=1; jj<=icount; ++jj){
                if (bluelabel_randomized[jj] == i) goto newiblue;
            }
            ++icount;
            bluelabel_randomized[j] = i;
```

```
        }
      }

    //*************************************************************************************************
    //
    //                    Scan for user interrupt:
    //
    //                    A --> set flag for measuring and recording run statistics
    //                    B --> toggle blue squad parameters to display
    //                    C --> paint 'command structure'
    //                    D --> open data file
    //                    E --> toggle red squad parameters to display
    //                    F --> fast screen update
    //                    H --> take a 'snapshot' of the current configuration
    //                    L --> close statistics files
    //                    N --> restart with random seed
    //                    O --> options for making 'on-the-fly' parameter changes
    //                    P --> open 'playback' (*.out) file
    //                    Q --> quit
    //                    R --> restart with same random seed (re-run)
    //                    S --> single-step screen update
    //                    T --> enable trace
    //                    U --> toggle: hightlight squads whose parameters are currently displayed
    //
    //*************************************************************************************************
wait: if( battle.default_speed == 2 && !_kbhit() ) goto wait;

      if( _kbhit() ){
        buffer = _getch();
options:    switch (buffer) {
            case 'u': // hightlight squads whose parameters are currently displayed
                if (battle.squad_color_flag == 0){
                    battle.squad_color_flag = 1; // highlight 'on'
                }
                else{
                    battle.squad_color_flag = 0; // highlight 'off'
                }
                _clearscreen( _GVIEWPORT );
                UPDATEPICTURE(stat_data, itime, red_GC.red_GC_flag,
                              blue_GC.blue_GC_flag,&red_command, &blue_command,
                              &battle, &red, &blue, &stats);
                goto wait;
                break;
            case 'e': // toggle red squad parameters to display
                if ( red.num_red_squads > 1 ){
                    red.display_red_squad =
                        (red.display_red_squad + 1) % (red.num_red_squads+1);
                    if (red.display_red_squad==0)red.display_red_squad=1;
                    _clearscreen( _GCLEARSCREEN );
                    SCREENDATA(itime, idata, filename, &red_command, &blue_command, &battle, &red, &blue);
                    UPDATEPICTURE(stat_data, itime, red_GC.red_GC_flag,
                                  blue_GC.blue_GC_flag,&red_command, &blue_command,
                                  &battle, &red, &blue, &stats);
                }
                goto wait;
                break;
            case 'b': //toggle blue squad parameters to display
                if ( blue.num_blue_squads > 1 ){
                    blue.display_blue_squad =
                        (blue.display_blue_squad + 1) % (blue.num_blue_squads+1);
                    if (blue.display_blue_squad==0)blue.display_blue_squad=1;
                    _clearscreen( _GCLEARSCREEN );
                    SCREENDATA(itime, idata, filename, &red_command, &blue_command, &battle, &red, &blue);
                    UPDATEPICTURE(stat_data, itime, red_GC.red_GC_flag,
                                  blue_GC.blue_GC_flag,&red_command, &blue_command,
                                  &battle, &red, &blue, &stats);
                }
                goto wait;
                break;
            case 'h': // take a 'snapshot' of the current configuration
                _getvideoconfig( &vc );
                if (vc.numxpixels < 641) { // then in VGA mode
                    _setviewport(200,455,525,465);
                    _clearscreen( _GVIEWPORT );
                    _setviewport(1,1,639,479);
                    _moveto( 205, 450 );
                    _setcolor( 14 );
                    _setgtextvector( 1, 0 );
                    _outgtext("  'Snapshot' file name ? ");
                    _settextposition( 29, 52 );
                }
                else{
                    _setviewport(260,575,680,590);
                    _clearscreen( _GVIEWPORT );
                    _setviewport(1,1,799,599);
                    _moveto( 270, 577 );
                    _setcolor( 14 );
                    _setgtextvector( 1, 0 );
                    _outgtext("  'Snapshot' file name ? ");
                    _settextposition( 37, 59 );
                }
                scanf("%s", &snapshotfile);
                if ( (outdatafile = fopen(snapshotfile, "wt")) == NULL ){
                    printf(" Cannot open data file.\n");
                    exit(1);
                }
```

```
        _clearscreen( _GCLEARSCREEN );
        SCREENDATA(itime, idata, filename, &red_command, &blue_command, &battle, &red, &blue);
        UPDATEPICTURE(stat_data, itime, red_GC.red_GC_flag, blue_GC.blue_GC_flag,&red_command,
                      &blue_command, &battle, &red, &blue, &stats);
        WRITE_OUT_FILE(outdatafile, red_GC.red_GC_flag, blue_GC.blue_GC_flag, &red_command,
                       &blue_command, &battle, &red, &blue);

        //
        // write current configuration
        //

        fprintf(outdatafile, "%i\n", itime);

        if (red_command.red_command_flag == 0) {
           for (i=1; i<=red.irednum; ++i) {
                fprintf(outdatafile, "%i   %i   %i\n", red.rstatus[i], red.redx[i], red.redy[i]);
           }
        }
        else{
            for (i=1; i<=red.irednum; ++i) {
                fprintf(outdatafile, "%i   %i   %i   %i\n", red.rstatus[i], red.redx[i],
                        red.redy[i], red_command.red_command[i]);
            }
            fprintf(outdatafile, "%i\n", red_command.num_red_commanders);
            for (i=1; i<= red_command.num_red_commanders; ++i){
                fprintf(outdatafile, "%i   %i   %i\n", red_command.red_ISAACA_commander[i],
                    red_command.red_command_R[i], red_command.red_num_under_command[i]);
                for (j=1; j<=red_command.red_num_under_command[i]; ++j){
                    fprintf(outdatafile, "%i\n", red_command.red_ISAACA_under_command[i][j]);
                }
            }
        }

        if (blue_command.blue_command_flag == 0) {
           for (i=1; i<=blue.ibluenum; ++i) {
                fprintf(outdatafile, "%i   %i   %i\n", blue.bstatus[i], blue.bluex[i], blue.bluey[i]);
           }
        }
        else{
            for (i=1; i<=blue.ibluenum; ++i) {
                fprintf(outdatafile, "%i   %i   %i   %i\n", blue.bstatus[i], blue.bluex[i],
                        blue.bluey[i], blue_command.blue_command[i]);
            }
            fprintf(outdatafile, "%i\n", blue_command.num_blue_commanders);
            for (i=1; i<= blue_command.num_blue_commanders; ++i){
                fprintf(outdatafile, "%i   %i   %i\n", blue_command.blue_ISAACA_commander[i],
                    blue_command.blue_command_R[i], blue_command.blue_num_under_command[i]);
                for (j=1; j<=blue_command.blue_num_under_command[i]; ++j){
                    fprintf(outdatafile, "%i\n", blue_command.blue_ISAACA_under_command[i][j]);
                }
            }
        }

       fclose(outdatafile);
       goto wait;
       break;
    case 'n': // restart with random seed
        // close statistics output files
        if (stats.stat_flag > 0){
            for (ii=1; ii<=13; ii++){
                fclose(stat_data[ii]);
            }
        }

        // set default trace to no trace
        itrace = 0;

        // initialize squad number to display on screen
        red.display_red_squad = 1;
        blue.display_blue_squad = 1;
        battle.squad_color_flag = 0;

        // re-set fratricide counters
        battle.red_frat_count=0;
        battle.blue_frat_count=0;

        // re-set defense counters
        if (red.defense_flag==0){
            for(ii=1;ii<=red.irednum;ii++)red.defense_clock[ii]=0;
        }
        if (blue.defense_flag==0){
            for(ii=1;ii<=blue.ibluenum;ii++)blue.defense_clock[ii]=0;
        }

        // reset command structure
        red_command.red_subordinate_color_flag = 0;
        blue_command.blue_subordinate_color_flag = 0;

        // initialize for run
        goto changeoption;
        break;
    case 'c': // paint 'command structure'
        if ( (red_GC.red_GC_flag == 1 && red_command.num_red_commanders > 1 ) ||
             (blue_GC.blue_GC_flag == 1 && blue_command.num_blue_commanders > 1) ){
            red_command.red_subordinate_color_flag =
                (red_command.red_subordinate_color_flag + 1) % 9;
            blue_command.blue_subordinate_color_flag =
```

```
            (blue_command.blue_subordinate_color_flag + 1) % 9;
        }
        else{
            if (red_command.red_command_flag == 1 ){
                red_command.red_subordinate_color_flag =
                    (red_command.red_subordinate_color_flag + 1) % 8;
            }
            if (blue_command.blue_command_flag == 1 ){
            blue_command.blue_subordinate_color_flag =
                (blue_command.blue_subordinate_color_flag + 1) % 8;
            }
        }
        _clearscreen( _GVIEWPORT );
        UPDATEPICTURE(stat_data, itime, red_GC.red_GC_flag,
                      blue_GC.blue_GC_flag,&red_command, &blue_command,
                      &battle, &red, &blue, &stats);
        goto wait;
        break;
    case 'd': // open data file
        idata = 2;  // set data input flag to read data file
        _getvideoconfig( &vc );
        if (vc.numxpixels < 641) { // then in VGA mode
            _setviewport(200,455,525,465);
            _clearscreen( _GVIEWPORT );
            _setviewport(1,1,639,479);
            _moveto( 205, 450 );
            _setcolor( 14 );
            _setgtextvector( 1, 0 );
            _outgtext("   Input-data file name ? ");
            _settextposition( 29, 52 );
        }
        else{
            _setviewport(260,575,680,590);
            _clearscreen( _GVIEWPORT );
            _setviewport(1,1,799,599);
            _moveto( 270, 577 );
            _setcolor( 14 );
            _setgtextvector( 1, 0 );
            _outgtext("   Input-data file name ? ");
            _settextposition( 37, 61 );
        }
        scanf("%s", &filename);
        _clearscreen( _GCLEARSCREEN );

        // set default trace to no trace
        itrace = 0;

        // initialize squad number to display on screen
        red.display_red_squad = 1;
        blue.display_blue_squad = 1;
        battle.squad_color_flag = 0;

        // Re-seed random number generator
        idum=-31415926;
        zran=ran1(&idum);

        // reset command structure
        red_command.red_subordinate_color_flag = 0;
        blue_command.blue_subordinate_color_flag = 0;

        //*************************************************************************
        //
        //                        Read input from data file
        //
        //*************************************************************************
        INPUT_FILE_DATA(filename, &stats, &red_GC, &blue_GC, &red_command,
                        &blue_command, &battle, &red, &blue, &idum);
        goto changeoption;
        break;
    case 'p': // open 'playback' (*.out) file
        // close statistics output files
        if (stats.stat_flag > 0){
            for (ii=1; ii<=13; ii++){
                fclose(stat_data[ii]);
            }
        }

        battle.ichoice = 2;  // set select run option flag to playback file
        _getvideoconfig( &vc );
        if (vc.numxpixels < 641) { // then in VGA mode
            _setviewport(200,455,525,465);
            _clearscreen( _GVIEWPORT );
            _setviewport(1,1,639,479);
            _moveto( 205, 450 );
            _setcolor( 14 );
            _setgtextvector( 1, 0 );
            _outgtext("  Playback-data file name ? ");
            _settextposition( 29, 54 );
        }
        else{
            _setviewport(260,575,680,590);
            _clearscreen( _GVIEWPORT );
            _setviewport(1,1,799,599);
            _moveto( 270, 577 );
            _setcolor( 14 );
            _setgtextvector( 1, 0 );
            _outgtext("  Playback-data file name ? ");
```

```
                    _settextposition( 37, 63 );
                }
                scanf("%s", &plotfilename);
                _clearscreen( _GCLEARSCREEN );
                goto playagain;
                break;
            case 's': // single-step screen update
                battle.default_speed = 2;
                break;
            case 'f': // fast screen update
                battle.default_speed = 1;
                break;
            case 'q':
                _getvideoconfig( &vc );
                if (vc.numxpixels < 641) { // then in VGA mode
                    _setviewport(200,455,525,465);
                    _clearscreen( _GVIEWPORT );
                    _setviewport(1,1,639,479);
                    _moveto( 205, 455 );
                    _setcolor( 14 );
                    _setgtextvector( 1, 0 );
                    _outgtext("        Run Terminated");
                }
                else{ // in SVGA mode
                    _setviewport(260,575,680,590);
                    _clearscreen( _GVIEWPORT );
                    _setviewport(1,1,799,599);
                    _moveto( 270, 575 );
                    _setcolor( 14 );
                    _setgtextvector( 1, 0 );
                    _outgtext("        Run Terminated");
                }
                if (battle.ioutdata > 1){
                    fclose(outdatafile);
                }
                if (stats.stat_flag > 0){
                    for (ii=1; ii<=13; ii++){
                        fclose(stat_data[ii]);
                    }
                }

                // reset command structure
                red_command.red_subordinate_color_flag = 0;
                blue_command.blue_subordinate_color_flag = 0;

opwaitq:        bufferq = _getch();
                switch (bufferq) {
                    case 'r':
                        goto playagain;
                        break;
                    case 'd':
                        buffer=bufferq;
                        goto options;
                        break;
                    case 'p':
                        buffer=bufferq;
                        goto options;
                        break;
                    case 'q':
                        goto startagain;
                        break;
                    default:
                        goto opwaitq;
                }
                break;
            case 'o': // options for making 'on-the-fly' parameter changes

                //*********************************************************************
                //
                //                  Display prompt screen and make changes
                //
                //*********************************************************************
                irun_choice = PROMPT_SCREEN(&battle, &red, &blue, &red_GC, &blue_GC,
                                            &red_command, &blue_command, &stats, &idum);

                // re-set fratricide counters
                battle.red_frat_count=0;
                battle.blue_frat_count=0;

                //*************************************************************************************
                //
                //    if irun_choice = 2 (start new run with new parameters) and battle.ioutdata>1
                //    then write new data to file
                //
                //*************************************************************************************
                if (irun_choice == 2 && battle.ioutdata > 1){

                     fclose(outdatafile);

                     if ( (outdatafile = fopen(outfilename, "wt")) == NULL ){
                         printf(" Cannot open data file.\n");
                         exit(1);
                     }

                     WRITE_OUT_FILE(outdatafile, red_GC.red_GC_flag, blue_GC.blue_GC_flag,
                        &red_command, &blue_command, &battle, &red, &blue);
```

```
        }

        if (irun_choice > 1){
            // close statistics output files
            if (stats.stat_flag > 0){
                for (ii=1; ii<=13; ii++){
                    fclose(stat_data[ii]);
                }
            }

            // set default trace to no trace
            itrace = 0;

            // initialize squad number to display on screen
            red.display_red_squad = 1;
            blue.display_blue_squad = 1;
            battle.squad_color_flag = 0;

            if (irun_choice == 3){
                // Re-seed using old seed value
                idum = -31415926;
                zran=ran1(&idum);
            }

            // re-set defense counters
            if (red.defense_flag==0){
                for(ii=1;ii<=red.irednum;ii++)red.defense_clock[ii]=0;
            }
            if (blue.defense_flag==0){
                for(ii=1;ii<=blue.ibluenum;ii++)blue.defense_clock[ii]=0;
            }

            // reset command structure coloring flags
            red_command.red_subordinate_color_flag = 0;
            blue_command.blue_subordinate_color_flag = 0;

            // initialize for next run
            goto changeoption;
        }
        else{
            _setbkcolor ( _BLACK );
            _clearscreen( _GCLEARSCREEN );
            if (battle.ioutdata != 2){

                  //*******************************************************
                  //
                  //             Update data on graphics screen
                  //
                  //*******************************************************

                  SCREENDATA(itime, idata, filename, &red_command,
                             &blue_command, &battle, &red, &blue);
            }
        }
        break;
    case 't':
        if (itrace == 0){
            itrace = 1; // trace 'on'
        }
        else{
            itrace = 0; // trace 'off'
        }
        break;
    case 'r': // restart with random seed #1 (re-run)
        // close statistics output files
        if (stats.stat_flag > 0){
            for (ii=1; ii<=13; ii++){
                fclose(stat_data[ii]);
            }
        }

        // re-set fratricide counters
        battle.red_frat_count=0;
        battle.blue_frat_count=0;

        // re-set defense counters
        if (red.defense_flag==0){
            for(ii=1;ii<=red.irednum;ii++)red.defense_clock[ii]=0;
        }
        if (blue.defense_flag==0){
            for(ii=1;ii<=blue.ibluenum;ii++)blue.defense_clock[ii]=0;
        }

        // set default trace to no trace
        itrace = 0;

        // initialize squad number to display on screen
        red.display_red_squad = 1;
        blue.display_blue_squad = 1;
        battle.squad_color_flag = 0;

        // initialize "playback" flag
        playterm = 0;

        // Re-seed random number generator
        idum=-31415926;
        zran=ran1(&idum);
```

```
        // reset command structure coloring flags
        red_command.red_subordinate_color_flag = 0;
        blue_command.blue_subordinate_color_flag = 0;

        // re-initialize all data
        INPUT_FILE_DATA(filename, &stats, &red_GC, &blue_GC, &red_command,
                        &blue_command, &battle, &red, &blue, &idum);

        // re-run
        goto changeoption;
        break;
    case 'l': // close statistics files
        for (ii=1; ii<=13; ii++){
            fclose(stat_data[ii]);
        }
        // reset stat_flag
        stats.stat_flag = 0;
        _getvideoconfig( &vc );
        if (vc.numxpixels < 641) { // then in VGA mode
            _setviewport(200,455,525,465);
            _clearscreen( _GVIEWPORT );
            _setviewport(140,48,525,428);
        }
        else{
            _setviewport(260,575,680,590);
            _clearscreen( _GVIEWPORT );
            _setviewport(120,50,680,550);
        }
        goto wait;
        break;
    case 'a': // set flag for measuring and recording run statistics
        _getvideoconfig( &vc );
        // stat_flag = 0 <---> no files open, no stats
        // stat_flag = 1 <--->    files open,    stats
        // stat_flag = 2 <--->    files open, no stats
        switch(stats.stat_flag){
                case 0:
                    stats.stat_flag = 1;
                    // open output stat files
                    if ( (stat_data[1] = fopen("stats_1.dat", "wt")) == NULL ||
                         (stat_data[2] = fopen("stats_2.dat", "wt")) == NULL ||
                         (stat_data[3] = fopen("stats_3.dat", "wt")) == NULL ||
                         (stat_data[4] = fopen("stats_4.dat", "wt")) == NULL ||
                         (stat_data[5] = fopen("stats_5.dat", "wt")) == NULL ||
                         (stat_data[6] = fopen("stats_6.dat", "wt")) == NULL ||
                         (stat_data[7] = fopen("stats_7.dat", "wt")) == NULL ||
                         (stat_data[8] = fopen("stats_8.dat", "wt")) == NULL ||
                         (stat_data[9] = fopen("stats_9.dat", "wt")) == NULL ||
                         (stat_data[10] = fopen("stats_10.dat", "wt")) == NULL ||
                         (stat_data[11] = fopen("stats_11.dat", "wt")) == NULL ||
                         (stat_data[12] = fopen("stats_12.dat", "wt")) == NULL ||
                         (stat_data[13] = fopen("stats_13.dat", "wt")) == NULL){
                            printf(" Cannot open data file.\n");
                            exit(1);
                    }
                    if (vc.numxpixels < 641) { // then in VGA mode
                            _setviewport(200,455,525,465);
                            _clearscreen( _GVIEWPORT );
                            _setviewport(1,1,639,479);
                            _moveto( 205, 450 );
                            _setcolor( 14 );
                            _setgtextvector( 1, 0 );
                            _outgtext("Statistics being calculated ... ");
                            _setviewport(140,48,525,428);
                    }
                    else{
                            _setviewport(260,575,680,590);
                            _clearscreen( _GVIEWPORT );
                            _setviewport(1,1,799,599);
                            _moveto( 270, 577 );
                            _setcolor( 14 );
                            _setgtextvector( 1, 0 );
                            _outgtext("Statistics being calculated ... ");
                            _setviewport(120,50,680,550);
                    }
                    break;
                case 1:
                    stats.stat_flag = 2;
                    _getvideoconfig( &vc );
                    if (vc.numxpixels < 641) { // then in VGA mode
                        _setviewport(200,455,525,465);
                        _clearscreen( _GVIEWPORT );
                        _setviewport(1,1,639,479);
                        _moveto( 200, 450 );
                        _setcolor( 14 );
                        _setgtextvector( 1, 0 );
                        _outgtext("Statistics calculations paused ... ");
                        _setviewport(140,48,525,428);
                    }
                    else{
                        _setviewport(260,575,680,590);
                        _clearscreen( _GVIEWPORT );
                        _setviewport(1,1,799,599);
                        _moveto( 260, 577 );
                        _setcolor( 14 );
                        _setgtextvector( 1, 0 );
```

```
                        _outgtext("Statistics calculations paused ... ");
                        _setviewport(120,50,680,550);
                    }
                    break;
                case 2:
                    stats.stat_flag = 1;
                    _getvideoconfig( &vc );
                    if (vc.numxpixels < 641) { // then in VGA mode
                        _setviewport(200,455,525,465);
                        _clearscreen( _GVIEWPORT );
                        _setviewport(1,1,639,479);
                        _moveto( 205, 450 );
                        _setcolor( 14 );
                        _setgtextvector( 1, 0 );
                        _outgtext("Statistics being calculated ... ");
                        _setviewport(140,48,525,428);
                    }
                    else{
                        _setviewport(260,575,680,590);
                        _clearscreen( _GVIEWPORT );
                        _setviewport(1,1,799,599);
                        _moveto( 270, 577 );
                        _setcolor( 14 );
                        _setgtextvector( 1, 0 );
                        _outgtext("Statistics being calculated ... ");
                        _setviewport(120,50,680,550);
                    }
                    break;
            }
            goto wait;
            break;
    }
}

//*****************************************************************************
//
//          Determine what blue isaacs are in red's neighborhood
//
//*****************************************************************************
for (i=1; i<=red.irednum; ++i) {
   ibinrnum = BLUEINRED( i, &red, &blue, ilblbinr );
}

//*****************************************************************************
//
//          Determine what red isaacs are in blue's neighborhood
//
//*****************************************************************************
for (i=1; i<=blue.ibluenum; ++i) {
   irinbnum = REDINBLUE( i, &red, &blue, ilblrinb);
}

//*****************************************************************************
//
//                    Adjudicate combat attrition
//
//*****************************************************************************
if (battle.max_combat_flag == 1){ // then no limit on number of
                                  // simultaneous engagements

   COMBAT( &battle, &red, &blue, &idum, ilblbinr, ilblrinb);
}
else{ // use routine that puts limit on the number of
      // simultaneous engagements
   COMBAT_2( &battle, &red, &blue, &idum, ilblbinr, ilblrinb);
}

//*****************************************************************************
//
//          Update local command structure and adjudicate local
//          command 'promotion' in case of combat 'kill'
//
//*****************************************************************************
if ( red_command.red_command_flag == 1 ){
   RED_PROMOTIONS(&red_command, &red, &idum);
}
if ( blue_command.blue_command_flag == 1 ){
   BLUE_PROMOTIONS(&blue_command, &blue, &idum);
}

if (battle.ioutdata != 2){

   //**************************************************************************
   //
   //          Update picture; first clear screen if trace is off
   //
   //**************************************************************************
   if (itrace == 0){
      _clearscreen( _GVIEWPORT );
   }

   //**************************************************************************
   //
   //                  Update picture on graphics screen
   //
   //**************************************************************************
   UPDATEPICTURE(stat_data, itime, red_GC.red_GC_flag, blue_GC.blue_GC_flag,
```

```
                    &red_command, &blue_command, &battle, &red, &blue, &stats);
  }

//*************************************************************************
//
//                        Output data to file
//
//*************************************************************************
if (battle.ioutdata > 1){
  //***********************************************************************
  //
  //    If termination condition is 1 (till one isaac reaches enemy goal) or
  //    it is 2 (continue till both sides exhausted) + the alloted time has not
  //    yet been reached then write current RED and BLUE states to file
  //
  //***********************************************************************
  if (   battle.itermcond == 1 ||
       ( battle.itermcond == 2 && itime <= iternum) ){

     if (battle.ioutdata == 2){
         _setviewport(1,200,799,300);
         _clearscreen( _GVIEWPORT );
         _setviewport(1,1,799,599);
         _setcolor( 15 );
         _unregisterfonts();
         _registerfonts( "sserife.FON" );
         strcat( strcat( strcpy( list, "t'" ), "sserife"), "'");
         strcat( list, "h30w24bv" );
         _getfontinfo( &fi );
         _setfont( list );
             xfon = (vc.numxpixels / 2) - (_getgtextextent("time = xxx") / 2);
         _moveto( xfon, 250 );
         _outgtext ("time = ");
         _outgtext( itoa( itime, bb, 10) );
         _unregisterfonts();
         _registerfonts( "oem10.FON" );
         strcat( strcat( strcpy( list, "t'" ), "oem10"), "'");
         strcat( list, "h30w24bv" );
         _getfontinfo( &fi );
         _setfont( list );
     }

     fprintf(outdatafile, "%i\n", itime);

     if (red_command.red_command_flag == 0) {
        for (i=1; i<=red.irednum; ++i) {
            fprintf(outdatafile, "%i   %i   %i\n", red.rstatus[i], red.redx[i], red.redy[i]);
        }
     }
     else{
         for (i=1; i<=red.irednum; ++i) {
             fprintf(outdatafile, "%i   %i   %i   %i\n", red.rstatus[i], red.redx[i],
                     red.redy[i], red_command.red_command[i]);
         }
         fprintf(outdatafile, "%i\n", red_command.num_red_commanders);
         for (i=1; i<= red_command.num_red_commanders; ++i){
             fprintf(outdatafile, "%i   %i   %i\n", red_command.red_ISAACA_commander[i],
                 red_command.red_command_R[i], red_command.red_num_under_command[i]);
             for (j=1; j<=red_command.red_num_under_command[i]; ++j){
                 fprintf(outdatafile, "%i\n", red_command.red_ISAACA_under_command[i][j]);
             }
         }
     }

     if (blue_command.blue_command_flag == 0) {
        for (i=1; i<=blue.ibluenum; ++i) {
            fprintf(outdatafile, "%i   %i   %i\n", blue.bstatus[i], blue.bluex[i], blue.bluey[i]);
        }
     }
     else{
         for (i=1; i<=blue.ibluenum; ++i) {
             fprintf(outdatafile, "%i   %i   %i   %i\n", blue.bstatus[i], blue.bluex[i],
                     blue.bluey[i], blue_command.blue_command[i]);
         }
         fprintf(outdatafile, "%i\n", blue_command.num_blue_commanders);
         for (i=1; i<= blue_command.num_blue_commanders; ++i){
             fprintf(outdatafile, "%i   %i   %i\n", blue_command.blue_ISAACA_commander[i],
                 blue_command.blue_command_R[i], blue_command.blue_num_under_command[i]);
             for (j=1; j<=blue_command.blue_num_under_command[i]; ++j){
                 fprintf(outdatafile, "%i\n", blue_command.blue_ISAACA_under_command[i][j]);
             }
         }
     }

  }
  else{
     igoal=3;
     goto goal;
  }
}

//*************************************************************************
//
//                RED local commanders generate ISAACA goals
//
//*************************************************************************
if (red_command.red_command_flag == 1){
```

```
    for (j=1; j<=red_command.num_red_commanders; ++j) {

        if (red_command.red_command_patch == 1){
            RED_LOCAL_COMMAND_1(j, itime, &red_GC, &red_command, &blue_command,
                                &battle, &red, &blue, &idum);
        }
        else{
            if (red_command.red_command_patch == 2){
                RED_LOCAL_COMMAND_2(j, itime, &red_command, &blue_command, &battle, &red,
                                    &blue, &idum);
            }
        }
    }
}

//*************************************************************************
//
//                  RED global commanders generate LC goals
//
//*************************************************************************
if (red_GC.red_GC_flag == 1){

     RED_SWATH_AREA(battle.isize, &red_GC, &red_command, &red);
     RED_SWATH_DENSITY(battle.isize, &red_GC, &red_command, &red, &blue);
     RED_GLOBAL_COMMAND(&red_GC, &red_command, &red, &idum);

}

//*************************************************************************
//
//                              UPDATE RED ISAACAs
//
//*************************************************************************

for (j=1; j<=red.irednum; ++j) {

   //**********************************************************************
   //
   //                          Get randomized label
   //
   //**********************************************************************
   i = redlabel_randomized[j];

   //**********************************************************************
   //
   //                Do only if red ISAAC is alive or injured
   //
   //**********************************************************************
   if ( red.rstatus[i] > 0 )  {

          //*****************************************************************************
          //
          //                      Get local goals from local commander
          //
          //*****************************************************************************
          if (red_command.red_command_flag ==  1 && // if command flag is set
              red_command.reds_commander[i] != 0 ){ // if ith ISAACA has a local commander
              red_command.red_local_goal_x[i] =
                    red_command.red_command_goal_x[red_command.reds_commander[i]];
              red_command.red_local_goal_y[i] =
                    red_command.red_command_goal_y[red_command.reds_commander[i]];
          }

          //*****************************************************************************
          //
          //                    What does the ith red isaac see?
          //
          //           - irinrnum      : number of reds in red
          //           - ibinrnum      : number of blue in red
          //           - iradvnum      : number of reds within advance range
          //           - ibinrinjnum   : number of injured blues
          //           - irinrinjnum   : number of injured reds
          //
          //*****************************************************************************
          irinrnum = 0;
          ibinrnum = 0;
          iradvnum = MAXISAACNUM;
          ibinrinjnum = 0;
          irinrinjnum = 0;

          if (red_command.red_command_flag ==  1 && // if command flag is set
              red_command.red_command[i] == 2 ){    // if ith ISAACA is a local commander
                RED_COMMAND_SENSOR(&irinrnum, &ibinrnum, &iradvnum, &ibinrinjnum,
                                   &irinrinjnum, i, &red_command, &red, &blue, ilblbinr);
          }
          else{ // use function for subordinate ISAACA
                RED_SENSOR(&irinrnum, &ibinrnum, &iradvnum, &ibinrinjnum, &irinrinjnum, i,
                           &red, &blue, ilblbinr);
          }

          //*****************************************************************************
          //
          //   Adapt red weights; i.e. determine values for red.w1red, red.w2red,
          //   red.w3red, red.w4red, red.w5red, red.w6red to be used for this time step
          //
          //*****************************************************************************
          if ( red_command.red_command_flag == 0 ||
```

```
     red_command.red_command_flag == 1 && red_command.red_command[i] < 2){
        ADAPT_RED_ISAACA_WEIGHTS(i, irinrnum, ibinrnum, iradvnum, &red_command,
                                 &red);
}

if ( red_command.red_command_flag == 1 && red_command.red_command[i] == 2){
        ADAPT_RED_LC_WEIGHTS(i, irinrnum, ibinrnum, iradvnum, &red_command,
                             &red);
}

if ( red_GC.red_GC_flag == 1 ){
        ADAPT_RED_GC_WEIGHTS(i, irinrnum, ibinrnum, iradvnum, &red_GC,
                             &red_command, &red);
}

//*************************************************************************
//
//              Are communications to be used between ISAACAs?
//
//*************************************************************************
if (red.red_COMM_flag != 0){ // if COMMs 'on' then get COMM data
    RED_COMM_INFO(i, &red, &blue);
}

//*************************************************************************
//
//          Compute expected penalty for each possible move;
//          isaac's move will be into square with least penalty
//
//*************************************************************************

igoalflag=0; // if remains equal to 0 then goal not reached

//*************************************************************************
//
//                      Initialize minimum sum value
//
//*************************************************************************
zmin = (float)(99999.);

// get movement range
imrr = red.iredmoverange[i];
if (red.rstatus[i] == 1) imrr = 1; // if injured, make sure max range equals 1


if (red.red_COMM_flag == 1){ // use 'COMM' routine
    zmin = COMPUTEREDPENALTY_COMM(i, imrr, &igoalflag, irinrinjnum, ibinrinjnum,
              zmin, iflag, z, &red_command, &blue_command, &battle, &red, &blue);
}
else{
    if ( red_GC.red_GC_flag == 0 ){
        zmin = COMPUTEREDPENALTY(i, imrr, &igoalflag, irinrinjnum, ibinrinjnum,
                  zmin, iflag, z, &red_command, &blue_command, &battle, &red, &blue);
    }
    else{
        zmin = COMPUTEREDPENALTY_GC(i, imrr, &igoalflag, irinrinjnum, ibinrinjnum,
                  zmin, iflag, z, &red_GC, &blue_GC, &red_command, &blue_command,
                  &battle, &red, &blue);
    }
}

if ( igoalflag == 1) {
  igoal = 1;
  goto goal;
}

//*************************************************************************
//
//     If zmin = 99999 then there are no viable moves --> do nothing
//
//*************************************************************************
if ( zmin == 99999. ){
//
// do nothing
//
  if (imrr == 1){
      imove = 5;
  }
  else{
      imove = 13;
  }
}
else{
//*************************************************************************
//
//         See what possible local moves correspond to zmin
//
//*************************************************************************
  imc = 0; // initialize local count variable
  for (imx = - imrr; imx <= imrr; ++imx) {
      for (imy = - imrr; imy <= imrr; ++imy) {
          if (iflag[imx + 2][imy + 2] == 1 &&
              z[imx + 2][imy + 2] == zmin){
              // add another candidate move to count
              ++imc;
              //                    select candidate move
              if (imrr == 1){
```

```
                        //
                        //      1 = (-1,+1)   |   2 = (0,+1)    |   3 = (+1,+1)
                        //      ---------------------------------------------
                        //      4 = (-1,0)    |   5 = (0,0)     |   6 = (+1,0)
                        //      ---------------------------------------------
                        //      7 = (-1,-1)   |   8 = (0,-1)    |   9 = (+1,-1)
                        //
                        imovecand[imc] = imx + 5 - 3 * imy;
                    }
                    else{
                        //
                        //      1 = (-2,+2)   |      .....      |   5 = (+2,+2)
                        //      ---------------------------------------------
                        //         .....      |   13 = (0,0)    |      .....
                        //      ---------------------------------------------
                        //      21 = (-2,-2)  |      .....      |  25 = (+2,-2)
                        //
                        imovecand[imc] = imx + 13 - 5 * imy;
                    }
                }
            }
        }

        //*************************************************************************
        //
        //      Actual move is randomly selected from among the imc candidates
        //
        //*************************************************************************
            if (imc == 1){
                imove = imovecand[1];
                }
            else{
                zmoveprob = (float)(0.0001 * getrandom(1,10000,&idum));
                for (im = 1; im < imc + 1; ++im) {
                    if (zmoveprob > (float)(im - 1) / (float)(imc) &&
                        zmoveprob <= (float)(im) / (float)(imc) ){
                        imove = imovecand[im];
                    }
                }
            }
        }

        //*************************************************************************
        //
        //          Move red to new square for which penalty is minimum
        //
        //*************************************************************************
          MOVERED (i, imrr, imove, &battle, &red);

    } // end if red.rstatus[i]!=0 test

} // end i = 1 to red.irednum loop

//*****************************************************************************
//
//                  BLUE local commanders generate local goals
//
//*****************************************************************************
if (blue_command.blue_command_flag == 1){
    for (j=1; j<=blue_command.num_blue_commanders; ++j) {
        if (blue_command.blue_command_patch == 1){
            BLUE_LOCAL_COMMAND_1(j, itime, &blue_GC, &red_command, &blue_command,
                                 &battle, &red, &blue, &idum);
        }
        else{
            if (blue_command.blue_command_patch == 2){
                BLUE_LOCAL_COMMAND_2(j, itime, &red_command, &blue_command,
                                     &battle, &red, &blue, &idum);
            }
        }
    }
}

//*****************************************************************************
//
//                  BLUE global commanders generate LC goals
//
//*****************************************************************************
if (blue_GC.blue_GC_flag == 1){

     BLUE_SWATH_AREA(battle.isize, &blue_GC, &blue_command, &blue);
     BLUE_SWATH_DENSITY(battle.isize, &blue_GC, &blue_command, &red, &blue);
     BLUE_GLOBAL_COMMAND(&blue_GC, &blue_command, &blue, &idum);

}

//*****************************************************************************
//
//                          UPDATE BLUE ISAACAs
//
//*****************************************************************************

for (j=1; j<=blue.ibluenum; ++j) {

   //*************************************************************************
   //
   //                          Get randomized label
```

```
//
//*************************************************************************
i = bluelabel_randomized[j];

//*************************************************************************
//
//               Do only if blue ISAAC is alive or injured
//
//*************************************************************************
if ( blue.bstatus[i] > 0 ){

        //*********************************************************************************
        //
        //                      Get local goals from local commander
        //
        //*********************************************************************************
        if (blue_command.blue_command_flag ==  1 &&// if command flag is set
            blue_command.blues_commander[i] != 0 ){// if ith ISAACA has a local commander
            blue_command.blue_local_goal_x[i] =
              blue_command.blue_command_goal_x[blue_command.blues_commander[i]];
            blue_command.blue_local_goal_y[i] =
              blue_command.blue_command_goal_y[blue_command.blues_commander[i]];
        }

        //*************************************************************************
        //
        //                  What does the ith blue isaac see?
        //
        //          - ibinbnum     : number of blues in blue
        //          - irinbnum     : number of reds in blue
        //          - ibadvnum     : number of blues within advance range
        //          - irinbinjnum  : number of injured reds
        //          - ibinbinjnum  : number of injured blues
        //
        //*************************************************************************
        ibinbnum = 0;
        irinbnum = 0;
        ibadvnum = MAXISAACNUM;
        irinbinjnum = 0;
        ibinbinjnum = 0;

        if (blue_command.blue_command_flag == 1 && // if command flag is set
            blue_command.blue_command[i] == 2 ){    // if ith ISAACA is a local commander
              BLUE_COMMAND_SENSOR(&ibinbnum, &irinbnum, &ibadvnum, &irinbinjnum, &ibinbinjnum,
                                  i, &blue_command, &red, &blue, ilblrinb);
        }
        else{ // use function for subordinate ISAACA
              BLUE_SENSOR(&ibinbnum, &irinbnum, &ibadvnum, &irinbinjnum, &ibinbinjnum, i,
                    &red, &blue, ilblrinb);
        }

        //*********************************************************************************
        //
        // Adapt blue weights; i.e. determine values for blue.w1blue, blue.w2blue,
        // blue.w3blue, blue.w4blue, blue.w5blue, blue.w6blue to be used for this time step
        //
        //*********************************************************************************
        if ( blue_command.blue_command_flag == 0 ||
             blue_command.blue_command_flag == 1 && blue_command.blue_command[i] < 2){
              ADAPT_BLUE_ISAACA_WEIGHTS(i, ibinbnum, irinbnum, ibadvnum, &blue_command,
                                        &blue);
        }

        if ( blue_command.blue_command_flag == 1 && blue_command.blue_command[i] == 2){
              ADAPT_BLUE_LC_WEIGHTS(i, ibinbnum, irinbnum, ibadvnum, &blue_command,
                                    &blue);
        }

        if ( blue_GC.blue_GC_flag == 1 ){
              ADAPT_BLUE_GC_WEIGHTS(i, ibinbnum, irinbnum, ibadvnum, &blue_GC,
                                    &blue_command, &blue);
        }

        //*************************************************************************
        //
        //              Are communications to be used between ISAACAs?
        //
        //*************************************************************************
        if (blue.blue_COMM_flag != 0){ // if COMMs 'on' then get COMM data
           BLUE_COMM_INFO(i, &red, &blue);
        }

        //*************************************************************************
        //
        //          Compute expected penalty for each possible move;
        //          isaac's move will be into square with least penalty
        //
        //*************************************************************************

        igoalflag=0; // if remains equal to 0 then goal not reached

        //*************************************************************************
        //
        //                    Initialize minimum sum value
        //
        //*************************************************************************
        zmin = (float)(99999.);
```

```
// get movement range
imbr = blue.ibluemoverange[i];
if (blue.bstatus[i] == 1) imbr = 1; // if injured, make sure max range equals 1

if (blue.blue_COMM_flag == 1){ // use 'COMM' routine
    zmin = COMPUTEBLUEPENALTY_COMM(i, imbr, &igoalflag, irinbinjnum, ibinbinjnum,
                                   zmin, iflag, z, &red_command, &blue_command,
                                   &battle, &red, &blue);
}
else{
    if ( blue_GC.blue_GC_flag == 0 ){
        zmin = COMPUTEBLUEPENALTY(i, imbr, &igoalflag, irinbinjnum, ibinbinjnum,
                                  zmin, iflag, z, &red_command, &blue_command,
                                  &battle, &red, &blue);
    }
    else{
        zmin = COMPUTEBLUEPENALTY_GC(i, imbr, &igoalflag, irinbinjnum, ibinbinjnum,
                                  zmin, iflag, z, &red_GC, &blue_GC, &red_command,
                                  &blue_command, &battle, &red, &blue);
    }
}

if ( igoalflag == 2) {
     igoal = 2;
     goto goal;
}

//*************************************************************************
//
//     If zmin = 99999 then there are no viable moves --> do nothing
//
//*************************************************************************
if ( zmin == 99999. ){
//
// do nothing
//
  if (imbr == 1){
      imove = 5;
  }
  else{
      imove = 13;
  }
}
else{
//*************************************************************************
//
//         See what possible local moves correspond to zmin
//
//*************************************************************************
  imc = 0;
  for (imx = -imbr; imx <= imbr; ++imx) {
      for (imy = -imbr; imy <= imbr; ++imy) {
        if (iflag[imx + 2][imy + 2] == 1 &&
          z[imx + 2][imy + 2] == zmin){
          // add another candidate move to count
          ++imc;
          //                  select candidate move
          if (imbr == 1){
              //
              //     1 = (-1,+1)   |   2 = (0,+1)    |   3 = (+1,+1)
              //     ---------------------------------------------
              //     4 = (-1,0)    |   5 = (0,0)     |   6 = (+1,0)
              //     ---------------------------------------------
              //     7 = (-1,-1)   |   8 = (0,-1)    |   9 = (+1,-1)
              //
              imovecand[imc] = imx + 5 - 3 * imy;
          }
          else{
              //
              //     1 = (-2,+2)   |     .....     |   5 = (+2,+2)
              //     ---------------------------------------------
              //         .....     |   13 = (0,0)  |       .....
              //     ---------------------------------------------
              //     21 = (-2,-2)  |     .....     |  25 = (+2,-2)
              //
              imovecand[imc] = imx + 13 - 5 * imy;
          }
        }
      }
  }

//*************************************************************************
//
//      Actual move is randomly selected from among the imc candidates
//
//*************************************************************************
    if (imc == 1){
        imove = imovecand[1];
    }
    else{
        zmoveprob = (float)(0.0001 * getrandom(1,10000,&idum));
        for (im=1; im<imc+1; ++im) {
            if (zmoveprob > (float)(im - 1) / (float)(imc) &&
                zmoveprob <= (float)(im) / (float)(imc) ){
                imove = imovecand[im];
            }
```

```
                    }
                  }
                } // end if zmin=99999

              //*************************************************************************
              //
              //          Move red to new square for which penalty is minimum
              //
              //*************************************************************************
                MOVEBLUE (i, imbr, imove, &battle, &blue);

        } // end if blue.bstatus[i]!=0 test

      } // end i = 1 to blue.ibluenum loop

      goto start;

       //*************************************************************************
       //
       //                          Run is terminated
       //
       //*************************************************************************
goal:
      if (battle.ioutdata > 1){
         fclose(outdatafile);
         if ( (battle.itermcond == 2 && itime == iternum) ||
              igoal == 3){
                 _getvideoconfig( &vc );
                 if (vc.numxpixels < 641) { // then use VGA value
                     _setviewport(200,455,525,465);
                     _clearscreen( _GVIEWPORT );
                     _setviewport(1,1,639,479);
                     _moveto( 205, 455 );
                     _setcolor( 14 );
                     _setgtextvector( 1, 0 );
                     _outgtext("          Run Complete");
                     battle.default_speed = 2;
                     goto wait;
                 }
                 else{
                     _setviewport(260,575,680,590);
                     _clearscreen( _GVIEWPORT );
                     _setviewport(1,1,799,599);
                     _moveto( 270, 575 );
                     _setcolor( 14 );
                     _setgtextvector( 1, 0 );
                     _outgtext("          Run Complete");
                     battle.default_speed = 2;
                     goto wait;
                 }

              buffer = _getch();
              if (buffer == 'r') {
                  // set default trace to no trace
                  itrace = 0;

                  // initialize squad number to display on screen
                  red.display_red_squad = 1;
                  blue.display_blue_squad = 1;
                  battle.squad_color_flag = 0;

                  // Re-seed random number generator
                  idum=-31415926;
                  zran=ran1(&idum);

                  // reset command structure coloring flags
                  red_command.red_subordinate_color_flag = 0;
                  blue_command.blue_subordinate_color_flag = 0;

                  // initialize for re-run
                  goto changeoption;
              }
              else{
                  _clearscreen( _GCLEARSCREEN );
                  goto startagain;
              }

         }
      }

      if (igoal==1){
         _getvideoconfig( &vc );
         if (vc.numxpixels < 641) { // then use VGA value
               _setviewport(200,455,525,465);
               _clearscreen( _GVIEWPORT );
               _setviewport(1,1,639,479);
               _moveto( 205, 455 );
               _setcolor( 14 );
               _setgtextvector( 1, 0 );
              _outgtext(" Run Complete...RED attains goal");
               battle.default_speed = 2;
               goto wait;
         }
         else{
              _setviewport(260,575,680,590);
              _clearscreen( _GVIEWPORT );
              _setviewport(1,1,799,599);
```

```
            _moveto( 270, 575 );
            _setcolor( 14 );
            _setgtextvector( 1, 0 );
            _outgtext(" Run Complete...RED attains goal");
             battle.default_speed = 2;
             goto wait;
         }

         buffer = _getch();
         if (buffer == 'r') {
              // set default trace to no trace
              itrace = 0;

              // initialize squad number to display on screen
              red.display_red_squad = 1;
              blue.display_blue_squad = 1;
              battle.squad_color_flag = 0;

              // Re-seed random number generator
              idum=-31415926;
              zran=ran1(&idum);

              // reset command structure coloring flags
              red_command.red_subordinate_color_flag = 0;
              blue_command.blue_subordinate_color_flag = 0;

              // initialize for re-run
              goto changeoption;
         }
         else{
              _clearscreen( _GCLEARSCREEN );
              goto startagain;
         }

      }
      else{
              if (igoal == 2){
              _getvideoconfig( &vc );
              if (vc.numxpixels < 641) { // then use VGA value
                   _setviewport(200,455,525,465);
                   _clearscreen( _GVIEWPORT );
                   _setviewport(1,1,639,479);
                   _moveto( 205, 455 );
                   _setcolor( 14 );
                   _setgtextvector( 1, 0 );
                  _outgtext("Run Complete...BLUE attains goal");
                   battle.default_speed = 2;
                   goto wait;
              }
              else{
                  _setviewport(260,575,680,590);
                  _clearscreen( _GVIEWPORT );
                  _setviewport(1,1,799,599);
                  _moveto( 270, 575 );
                  _setcolor( 14 );
                  _setgtextvector( 1, 0 );
                  _outgtext("Run Complete...BLUE attains goal");
                   battle.default_speed = 2;
                   goto wait;
              }

              buffer = _getch();
              if (buffer == 'r') {
                  // set default trace to no trace
                  itrace = 0;

                  // initialize squad number to display on screen
                  red.display_red_squad = 1;
                  blue.display_blue_squad = 1;
                  battle.squad_color_flag = 0;

                  // Re-seed random number generator
                  idum=-31415926;
                  zran=ran1(&idum);

                  // reset command structure coloring flags
                  red_command.red_subordinate_color_flag = 0;
                  blue_command.blue_subordinate_color_flag = 0;

                  // initialize for re-run
                  goto changeoption;
              }
              else{
                  _clearscreen( _GCLEARSCREEN );
                  goto startagain;
              }

      }
      }

      //*************************************************************************
      //
      //                    Playback previously recorded run
      //
      //*************************************************************************
playback:
```

```
    _unregisterfonts();
    _registerfonts( "oem10.FON" );
    strcat( strcat( strcpy( list, "t'" ), "oem10"), "'");
    strcat( list, "h30w24bv" );
    _getfontinfo( &fi );
    _setfont( list );
    _settextposition( 13, 35);
    printf("Plot-data file name ? ");
    scanf("%s", &plotfilename);

playback_option:

    if ( !_setvideomode( _SRES16COLOR ) ){
        _setvideomode( _VRES16COLOR );
        _clearscreen( _GCLEARSCREEN );
    }

playagain:

    playterm = PLAYBACK(plotfilename, &idata, filename, &battle.ichoice);

    if (playterm == 2){  // then read-in new input data
        goto read_data;
    }

    if (playterm == 3){  // then play-back new file
        goto playback_option;
    }

    _getvideoconfig( &vc );

    if (vc.numxpixels < 641) { // then use VGA value
        _setviewport(200,455,525,465);
        _clearscreen( _GVIEWPORT );
        _setviewport(1,1,639,479);
        _moveto( 205, 455 );
        _setcolor( 14 );
        _setgtextvector( 1, 0 );
        if (playterm == 1){
            _outgtext("        Run Terminated");
        }
        else{
            if (playterm == 0){
                _outgtext("          Run Complete");
            }
        }
    }
    else{
        _setviewport(260,575,680,590);
        _clearscreen( _GVIEWPORT );
        _setviewport(1,1,799,599);
        _moveto( 270, 575 );
        _setcolor( 14 );
        _setgtextvector( 1, 0 );
        if (playterm == 1){
            _outgtext("        Run Terminated");
        }
        else{
            if (playterm == 0){
                _outgtext("          Run Complete");
            }
        }
    }

opwait:
    buffer = _getch();
    if (buffer == 'r') {
        goto playagain;
    }
    else{
        if( buffer=='d' || buffer=='p' || buffer=='q') {
            goto options;
        }
        else{
            goto opwait;
        }
    }

}

//*****************************************************************************
//
//                          Allocate Memory for Structures
//
//*****************************************************************************
struct red_GC_parameters *get_r_GC(void)
{
    struct red_GC_parameters *p ;

    if ( (p = _fmalloc( sizeof(struct red_GC_parameters) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
    return p;
}
```

```
struct blue_GC_parameters *get_b_GC(void)
{
    struct blue_GC_parameters *p ;

    if ( (p = _fmalloc( sizeof(struct blue_GC_parameters) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
    return p;
}

struct red_command_parameters *get_r_com(void)
{
    struct red_command_parameters *p ;

    if ( (p = _fmalloc( sizeof(struct red_command_parameters) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
    return p;
}

struct blue_command_parameters *get_b_com(void)
{
    struct blue_command_parameters *p ;

    if ( (p = _fmalloc( sizeof(struct blue_command_parameters) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
    return p;
}

struct battle_parameters *get_bat(void)
{
    struct battle_parameters *p ;

    if ( (p = _fmalloc( sizeof(struct battle_parameters) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
    return p;
}

struct red_parameters *get_red(void)
{
    struct red_parameters *p;

    if ( (p = _fmalloc( sizeof(struct red_parameters) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
    return p;
}

struct blue_parameters *get_blue(void)
{
    struct blue_parameters *p;

    if ( (p = _fmalloc( sizeof(struct blue_parameters) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
    return p;
}

struct statistics *get_stats(void)
{
    struct statistics *p;

    if ( (p = _fmalloc( sizeof(struct statistics) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
    return p;
}
```

Function Modules

A short description of each function module that appears in ISAAC's main function (see ***Main Module*** above) is given in table 12.

Table 12. ISAAC functions

Function	File	Description
ABS_FLOAT	ISAAC_P.C	returns absolute value of a float
ADAPT_BLUE_GC_WEIGHTS	ISAAC_C.C	adapts blue global commander weights
ADAPT_BLUE_ISAACA_WEIGHT	ISAAC_C.C	adapts blue ISAACA weights
ADAPT_BLUE_LC_WEIGHTS	ISAAC_C.C	adapts blue local commander weights
ADAPT_RED_GC_WEIGHTS	ISAAC_C.C	adapts red global commander weights
ADAPT_RED_ISAACA_WEIGHTS	ISAAC_C.C	adapts red ISAACA weights
ADAPT_RED_LC_WEIGHTS	ISAAC_C.C	adapts red local commander weights
BLUE_COMMAND_SENSOR	ISAAC_D.C	determines what the ith blue local commander sees
BLUE_COMM_INFO	ISAAC_H1.C	determines what ISAACAs are within blue's COMM range
BLUE_GLOBAL_COMMAND	ISAAC_S2.C	blue global commanders set 'direction' goals for LCs
BLUEINRED	ISAAC_D.C	determines number of blues within red sensor
BLUE_LOCAL_COMMAND_1	ISAAC_J.C	blue local commanders set local goals for 3-by-3 patch
BLUE_LOCAL_COMMAND_2	ISAAC_K2.C	blue local commanders set local goals for 5-by-5 patch
BLUE_PROMOTIONS	ISAAC_H2.C	adjudicate blue local commander promotions
BLUE_SENSOR	ISAAC_D.C	determines what the ith blue ISAACA sees within sensor
BLUE_SWATH_AREA	ISAAC_Q.C	calculates the area of each of 16 'swaths' centered at the current (x,y) coordinates of blue local commander
BLUE_SWATH_DENSITY	ISAAC_R2.C	calculates the density of red ISAACAs in each of 16 swaths centered at the current (x,y) coordinates of blue local commander
CENTER_MASS	ISAAC_T5.C	returns the center-of-mass of red, blue and total forces
CLUSTER_1	ISAAC_T3.C	returns the distribution of clusters (D=1) and average size
CLUSTER_2	ISAAC_T4.C	returns the distribution of clusters (D=2) and average size

COMBAT	ISAAC_I.C	adjudicates combat (assuming ALL enagements)
COMBAT_2	ISAAC_I.C	adjudicates combat (assuming enagement threshold set)
COMPUTEBLUEPENALTY	ISAAC_E1.C	calculates penalty for each blue move possibility
COMPUTEBLUEPENALTY_COMM	ISAAC_F.C	calculate blue-move penalty assuming COMM is 'on'
COMPUTEBLUEPENALTY_GC	ISAAC_E3.C	calculates penalty for blue assuming GC flag on
COMPUTEREDPENALTY	ISAAC_E1.C	calculates penalty for each red move possibility
COMPUTEREDPENALTY_COMM	ISAAC_F.C	calculate red-move penalty assuming COMM is 'on'
COMPUTEREDPENALTY_GC	ISAAC_E2.C	calculates penalty for red assuming GC flag on
GETRANDOM	ISAAC_P.C	get a random number between a and b
GOAL_STATS	ISAAC_T5.C	returns the number of ISAACAs near enemy flag
INITIALIZE_FIELD	ISAAC_B1.C	initialize battlefield parameters
INPUT_FILE_DATA	ISAAC_M1.C	read input from data file
INPUT_SCREEN_DATA	ISAAC_M2.C	input data from screen prompts
INTERPOINT_DIST	ISAAC_T1.C	calculates R-R, B-B, R-B and R,B-goal distance dists
MOMENT	ISAAC_T3.C	returns mean ave, ave deviation and standard deviation
MOVEBLUE	ISAAC_G.C	moves all blue ISAACAs (updates lattice positions)
MOVERED	ISAAC_G.C	moves all red ISAACAs (updates lattice positions)
NEIGHBORS	ISAAC_T5.C	returns the average number of ISAACAs at distance D
NOMEM	ISAAC_P.C	returns 'insufficient memory to run' message and exits
PLAYBACK	ISAAC_O.C	"plays-back" previously recorded *.out files
PROMPT_SCREEN	ISAAC_N.C	display choices for 'on-the-fly' parameter changes
RAN1	ISAAC_P.C	uniform random generator from 'Numerical Recipes' (Cambridge University Press)
RED_COMMAND_SENSOR	ISAAC_D.C	determines what the ith red local commander sees
RED_COMM_INFO	ISAAC_H1.C	determines what ISAACAs are within red's COMM range

RED_GLOBAL_COMMAND	ISAAC_S1.C	red global commanders set 'direction' goals for LCs
REDINBLUE	ISAAC_D.C	determines number of reds within blue sensor
RED_LOCAL_COMMAND_1	ISAAC_J.C	red local commanders set local goals for 3-by-3 patch
RED_LOCAL_COMMAND_2	ISAAC_K1.C	red local commanders set local goals for 5-by-5 patch
RED_PROMOTIONS	ISAAC_H2.C	adjudicate red local commander promotions
RED_SENSOR	ISAAC_D.C	determines what the ith red ISAACA sees within sensor
RED_SWATH_AREA	ISAAC_Q.C	calculates the area of each of 16 'swaths' centered at the current (x,y) coordinates of red local commander
RED_SWATH_DENSITY	ISAAC_R1.C	calculates the density of blue ISAACAs in each of 16 swaths centered at the current (x,y) coordinates of red local commander
SCREENDATA	ISAAC_B2.C	dump data to graphics screen
SIGNUM	ISAAC_P.C	sign (+1,-1, or 0) of a float
SPATIAL_ENTROPY	ISAAC_T2.C	computes spatial entropy for 4x4, 8x8 and 16x16 blocks
UPDATEPICTURE	ISAAC_L.C	update graphics screen ith new red and blue positions
WRITE_1_CLUSTER	ISAAC_M4.C	Write cluster distributions (calculated using D=1) to files (stats_10.dat, stats_11.dat)
WRITE_2_CLUSTER	ISAAC_M4.C	Write cluster distributions (calculated using D=1) to files (stats_12.dat, stats_13.dat)
WRITE_DATA_FILE	ISAAC_M3.C	Write current parameter values to data file
WRITE_INTERPOINT	ISAAC_M4.C	Write interpoint distance distributions to file (stats_2.dat, ... stats_8.dat)
WRITE_OUT_FILE	ISAAC_M3.C	Open and write to 'play-back' (*.out) file
WRITE_RR_NEIGHBORS	ISAAC_M4.C	Write red-in-red distributions to file (stats_14.dat)
WRITE_BB_NEIGHBORS	ISAAC_M4.C	Write blue-in-blue distributions to file (stats_15.dat)
WRITE_RB_NEIGHBORS	ISAAC_M4.C	Write red-in-blue distributions to file (stats_16.dat)
WRITE_BR_NEIGHBORS	ISAAC_M4.C	Write blue-in-red distributions to file (stats_17.dat)

WRITE_AR_NEIGHBORS	ISAAC_M4.C	Write all-in-red distributions to file (stats_18.dat)
WRITE_AB_NEIGHBORS	ISAAC_M4.C	Write all-in-blue distributions to file (stats_19.dat)

Appendix D: Source Code for ISAAC_GA

Below is the ANSI C source code for version 1.5.1of **ISAAC_GA** (i.e., the stand-alone *Genetic Algorithm* evolver; see Table 4). Screen and graphics functions are those defined in **graph.h** of *Microsoft's* Visual C/C++ compiler for DOS (v1.52). Note that **ISAAC_GA** uses a slightly older version of ISAAC's core engine than the one listed in appendix C. Specifically, the version of ISAAC that is embedded within **ISAAC_GA** allows only *one squad per side* and *excludes all command and control structures.* All auxiliary functions are the same as those listed in Table 12.

Header File

```
#define ISAAC_VERSION "ISAAC-GA / Version 1.5.1"
#define MAXFIELDSIZE 101
#define MAXSENSORRANGE 10
#define MAXINTERPOINTDIST (int)(1.414214 * MAXFIELDSIZE)
#define MAXISAACNUM 126
#define MAXNEIGHBORNUM 2*MAXISAACNUM+1
#define TERRAINMAXNUM 25
#define MAXCLUSTERSIZE 2*MAXISAACNUM+1
#define POPSIZE 40              // maximum population size
#define INIT_COND_MAX 50        // number of ICs to average over in GA calculation
```

Structures

```
//*************************************************************************************************************************
//
//                          - mission_objective     : mission objective measures
//                          - battle_parameters     : battlefield/combat parameters
//                          - red_parameters        : red ISAACA force parameters
//                          - blue_parameters       : blue ISAACA force parameters
//
//*************************************************************************************************************************
struct mission_objective                            // define a member template of the population
{
double alpha_1;                                     // weight for "time to goal" measure
double alpha_2;                                     // weight for "total friendly loss" measure
double alpha_3;                                     // weight for "total enemy loss" measure
double alpha_4;                                     // weight for "ratio between surviving red and blue ISAACAs" measure
double alpha_5;                                     // weight for "distance between red CM and blue flag" measure
double alpha_6;                                     // weight for "distance between blue CM and red flag" measure
double alpha_7;                                     // weight for "number of blue near red flag" measure
double alpha_8;                                     // weight for "number of red near blue flag" measure
double alpha_9;                                     // weight for "number of red fratricide hits" measure
double alpha_10;                                    // weight for "number of blue fratricide hits" measure
double time_to_goal[INIT_COND_MAX+1];               // time that first friendly ISAAC arrives at enemy flag
double total_friendly_loss[INIT_COND_MAX+1];        // total number of friendly ISAACAs killed or injured
double total_enemy_loss[INIT_COND_MAX+1];           // total number of enemy ISAACAs killed or injured
double survival_ratio[INIT_COND_MAX+1];             // ratio between surviving RED and BLUE ISAACAs (R/(R0*B))
double red_CM_to_BF_dist[INIT_COND_MAX+1];          // distance between red CM and blue flag
double blue_CM_to_RF_dist[INIT_COND_MAX+1];         // distance between blue CM and red flag
```

Appendix E: STATS_X.dat Data Fields

```
double red_near_BF[INIT_COND_MAX+1];        // number of red ISAACAs near blue flag
double blue_near_RF[INIT_COND_MAX+1];       // number of blue ISAACAs near red flag
double red_fratricide[INIT_COND_MAX+1];     // number of red fratricide hits
double blue_fratricide[INIT_COND_MAX+1];    // number of blue fratricide hits
double near_range;                          // the radius that defines "near" for red_near_BF and blue_near_RF
double near_range_num;                      // number of friendly ISAACAs that must be within near_range of BF
double r_CM_f;                              // fractional distance threshold such that if r_CM/max < r_CM_f then stop
};

struct battle_parameters
{
short goalcolor;
short boxcolor;
int default_speed;                  // =1 if run is FAST, =2 if run is SLOW
int ioutdata;                       // output: 1=screen only; 2=file only; 3=both
int ichoice;                        // run flag: 1=run engine; 2=playback file
int isize;                          // user specified battlefield size
int initdist;                       // initial force distribution flag
int ibattlebox_red_length;          // length of box containing initial distribution
int ibattlebox_red_width;           // width of box containing initial distribution
int ibattlebox_red_cen_x;           // x-coodinate of the center of red's initial box
int ibattlebox_red_cen_y;           // y-coodinate of the center of red's initial box
int ibattlebox_blue_length;         // length of box containing initial distribution
int ibattlebox_blue_width;          // width of box containing initial distribution
int ibattlebox_blue_cen_x;          // x-coodinate of the center of blue's initial box
int ibattlebox_blue_cen_y;          // y-coodinate of the center of blue's initial box
int itermcond;                      // termination condition flag (1=goal; 2=none)
int imove_selection;                // 1 = FIXED order; 2 = random order
int max_combat_flag;                // 1=# of sim engmnts lmtd; 0=no limit
int terrain_flag;                   // 1 = terrain to be used; 1 = no
int terrain_num;                    // number of terrain block
int terrain_size[TERRAINMAXNUM];    // radius of ith terrain block
int terrain_center_x[TERRAINMAXNUM]; // x-coordinate of the the ith block's center
int terrain_center_y[TERRAINMAXNUM]; // y-coordinate of the the ith block's center
int ioccupation[MAXFIELDSIZE][MAXFIELDSIZE]; // =2 if terrain, 1 if occupied, else 0
int reconstitution_flag;            // if 0 then no reconstitution, else reconstitution on
int red_fratricide_flag;            // =1 if red ISAACAs can accidentally kill red ISAACAs, else 0
int blue_fratricide_flag;           // =1 if blue ISAACAs can accidentally kill blue ISAACAs, else 0
int red_frat_rad;                   // radius surrounding targeted blue within which reds can be killed
int blue_frat_rad;                  // radius surrounding targeted red within which blues can be killed
int red_frat_count;                 // cummulative total of red fratricide 'hits'
int blue_frat_count;                // cummulative total of blue fratricide 'hits'
float red_frat_prob;                // probability that red is accidentally shot by red
float blue_frat_prob;               // probability that blue is accidentally shot by blue
};

struct red_parameters
{
short redcolor;
int redgoalx;                       // x coordinate of red goal
int redgoaly;                       // y coordinate of red goal
int redx[MAXISAACNUM];              // x-coordinate of ith red ISAAC
int redy[MAXISAACNUM];              // y-coordinate of ith red ISAAC
int rseer[MAXISAACNUM];             // =1 if red sees red and =0 otherwise
int rseeb[MAXISAACNUM];             // =1 if red sees blue and =0 otherwise
int rseercomm[MAXISAACNUM];         // =1 if red sees red via COMM link
int rseebcomm[MAXISAACNUM];         // =1 if red sees blue via COMM link
int rstatus[MAXISAACNUM];           // =1 if alive, 1 if injured, 0 if dead
int ibinr[MAXISAACNUM];             // number of blue isaacs in red isaac range
float w1red[MAXISAACNUM];           // active weight for red --> alive red
float w2red[MAXISAACNUM];           // active weight for red --> alive blue
float w3red[MAXISAACNUM];           // active weight for red --> injured red
float w4red[MAXISAACNUM];           // active weight for red --> injured blue
float w5red[MAXISAACNUM];           // active weight for red --> red goal
float w6red[MAXISAACNUM];           // active weight for red --> blue goal
```

```
int irednum;                          // total number of red ISAACs
int irsrange;                         // red sensor range
int iredfrange;                       // red fire range
float zshotbluebyreddef;              // probability that a red ISAAC shoots a blue
int iperred;                          // input flag for initial personality type
float w1rdeff_a;                      // default weight for alive red --> alive red
float w2rdeff_a;                      // default weight for alive red --> alive blue
float w3rdeff_a;                      // default weight for alive red --> injured red
float w4rdeff_a;                      // default weight for alive red --> injured blue
float w5rdeff_a;                      // default weight for alive red --> red goal
float w6rdeff_a;                      // default weight for alive red --> blue goal
float w1rdeff_i;                      // default weight for injured red --> alive red
float w2rdeff_i;                      // default weight for injured red --> alive blue
float w3rdeff_i;                      // default weight for injured red --> injured red
float w4rdeff_i;                      // default weight for injured red --> injured blue
float w5rdeff_i;                      // default weight for injured red --> red goal
float w6rdeff_i;                      // default weight for injured red --> blue goal
float red_w_a_max[MAXISAACNUM];       // maximum absolute value of default red alive weights
float red_w_i_max[MAXISAACNUM];       // maximum absolute value of default red injrd weights
float w1reddef_a[MAXISAACNUM];        // default weight for alive red --> alive red
float w2reddef_a[MAXISAACNUM];        // default weight for alive red --> alive blue
float w3reddef_a[MAXISAACNUM];        // default weight for alive red --> injured red
float w4reddef_a[MAXISAACNUM];        // default weight for alive red --> injured blue
float w5reddef_a[MAXISAACNUM];        // default weight for alive red --> red goal
float w6reddef_a[MAXISAACNUM];        // default weight for alive red --> blue goal
float w1reddef_i[MAXISAACNUM];        // default weight for injrd red --> alive red
float w2reddef_i[MAXISAACNUM];        // default weight for injrd red --> alive blue
float w3reddef_i[MAXISAACNUM];        // default weight for injrd red --> injured red
float w4reddef_i[MAXISAACNUM];        // default weight for injrd red --> injured blue
float w5reddef_i[MAXISAACNUM];        // default weight for injrd red --> red goal
float w6reddef_i[MAXISAACNUM];        // default weight for injrd red --> blue goal
int iredmovecont;                     // red movement constraint flag
int iradv_a[MAXISAACNUM];             // alive red advance threshold
int iradv_i[MAXISAACNUM];             // injured red advance threshold
int iradvrange[MAXISAACNUM];          // range in which red # > threshold to advance
int irclus_a[MAXISAACNUM];            // alive red cluster threshold
int irclus_i[MAXISAACNUM];            // injured red cluster threshold
int ircom_a[MAXISAACNUM];             // alive red combat threshold
int ircom_i[MAXISAACNUM];             // injured red combat threshold
int iradvrange_min;                   // min red advance threshold for random constraints
int iradvrange_max;                   // max red advance threshold for random constraints
int iradv_a_min;                      // min alive red advance threshold for ran constraints
int iradv_a_max;                      // max alive red advance threshold for ran constraints
int iradv_i_min;                      // min injrd red advance threshold for ran constraints
int iradv_i_max;                      // max injrd red advance threshold for ran constraints
int irclus_a_min;                     // min alive red cluster threshold for ran constraints
int irclus_a_max;                     // max alive red cluster threshold for ran constraints
int irclus_i_min;                     // min injrd red cluster threshold for ran constraints
int irclus_i_max;                     // max injrd red cluster threshold for ran constraints
int ircom_a_min;                      // min alive red combat threshold for ran constraints
int ircom_a_max;                      // max alive red combat threshold for ran constraints
int ircom_i_min;                      // min injrd red combat threshold for ran constraints
int ircom_i_max;                      // max injrd red combat threshold for ran constraints
float zrfromrmindist_a;               // minimum distance of alive red from red
float zrfromrgmindist_a;              // minimum distance of alive red from red goal
float zbfromrmindist_a;               // minimum distance of alive blue from red
float zrfromrmindist_i;               // minimum distance of injured red from red
float zrfromrgmindist_i;              // minimum distance of injured red from red goal
float zbfromrmindist_i;               // minimum distance of injured blue from red
int iredmoverange;                    // max movement radius for alive reds
int red_max_eng_num;                  // max # of simul engagements by red
int red_COMM_flag;                    // if = 0 then COMMs NOT used for red, else yes
int ircommrange;                      // red communications range
float rcommweight;                    // red COMM weight (relative to w=1)
float rcommweight_def;                // red default COMM weight
```

```
float zrsscale;                    // scale factor for multiplying red penalty
int red_clock[MAXISAACNUM];        // internal red clock (for reconstitution)
int red_max_r_time;                // maximum number of 'ticks' before reconstitution
};

struct blue_parameters
{
short bluecolor;
int bluegoalx;                     // x coordinate of blue goal
int bluegoaly;                     // y coordinate of blue goal
int bluex[MAXISAACNUM];            // x-coordinate of ith blue ISAAC
int bluey[MAXISAACNUM];            // y-coordinate of ith blue ISAAC
int bseer[MAXISAACNUM];            // =1 if blue sees red and =0 otherwise
int bseeb[MAXISAACNUM];            // =1 if blue sees blue and =0 otherwise
int bseercomm[MAXISAACNUM];        // =1 if blue sees red via COMM link
int bseebcomm[MAXISAACNUM];        // =1 if blue sees blue via COMM link
int bstatus[MAXISAACNUM];          // =1 if alive, 1 if injured, 0 if dead
int irinb[MAXISAACNUM];            // number of red isaacs in blue isaac range
float w1blue[MAXISAACNUM];         // active weight for blue --> alive blue
float w2blue[MAXISAACNUM];         // active weight for blue --> alive red
float w3blue[MAXISAACNUM];         // active weight for blue --> injured blue
float w4blue[MAXISAACNUM];         // active weight for blue --> injured red
float w5blue[MAXISAACNUM];         // active weight for blue --> blue goal
float w6blue[MAXISAACNUM];         // active weight for blue --> red goal
int ibluenum;                      // total number of blue ISAACs
int ibsrange;                      // blue sensor range
int ibluefrange;                   // blue fire range
float zshotredbybluedef;           // probability that a blue ISAAC shoots a red
int iperblue;                      // input flag for initial personality type
float w1bdeff_a;                   // default weight for alive blue --> alive blue
float w2bdeff_a;                   // default weight for alive blue --> alive red
float w3bdeff_a;                   // default weight for alive blue --> injured blue
float w4bdeff_a;                   // default weight for alive blue --> injured red
float w5bdeff_a;                   // default weight for alive blue --> blue goal
float w6bdeff_a;                   // default weight for alive blue --> red goal
float w1bdeff_i;                   // default weight for injured blue --> alive blue
float w2bdeff_i;                   // default weight for injured blue --> alive red
float w3bdeff_i;                   // default weight for injured blue --> injured blue
float w4bdeff_i;                   // default weight for injured blue --> injured red
float w5bdeff_i;                   // default weight for injured blue --> blue goal
float w6bdeff_i;                   // def weight vector for injured blue --> red goal
float blue_w_a_max[MAXISAACNUM];// max absolute value of default blue alive weights
float blue_w_i_max[MAXISAACNUM]; // max absolute value of default blue injurd weights
float w1bluedef_a[MAXISAACNUM];    // default weight for alive blue --> alive blue
float w2bluedef_a[MAXISAACNUM];    // default weight for alive blue --> alive red
float w3bluedef_a[MAXISAACNUM];    // default weight for alive blue --> injured blue
float w4bluedef_a[MAXISAACNUM];    // default weight for alive blue --> injured red
float w5bluedef_a[MAXISAACNUM];    // default weight for alive blue --> blue goal
float w6bluedef_a[MAXISAACNUM];    // default weight for alive blue --> red goal
float w1bluedef_i[MAXISAACNUM];    // default weight for injrd blue --> alive blue
float w2bluedef_i[MAXISAACNUM];    // default weight for injrd blue --> alive red
float w3bluedef_i[MAXISAACNUM];    // default weight for injrd blue --> injured blue
float w4bluedef_i[MAXISAACNUM];    // default weight for injrd blue --> injured red
float w5bluedef_i[MAXISAACNUM];    // default weight for injrd blue --> blue goal
float w6bluedef_i[MAXISAACNUM];    // default weight for injrd blue --> red goal
int ibluemovecont;                 // blue movement constraint flag
int ibadv_a[MAXISAACNUM];          // alive blue advance threshold
int ibadv_i[MAXISAACNUM];          // injured blue advance threshold
int ibadvrange[MAXISAACNUM];       // range within which blue # > threshold to advance
int ibclus_a[MAXISAACNUM];         // alive blue cluster threshold
int ibclus_i[MAXISAACNUM];         // injured blue cluster threshold
int ibcom_a[MAXISAACNUM];          // alive blue combat threshold
int ibcom_i[MAXISAACNUM];          // injured blue combat threshold
int ibadvrange_min;                // min blue advance threshold for random constraints
int ibadvrange_max;                // max blue advance threshold for random constraints
```

```
int ibadv_a_min;                // min alive blue advance threshold for ran constrnts
int ibadv_a_max;                // max alive blue advance threshold for ran constrnts
int ibadv_i_min;                // min injrd blue advance threshold for ran constrnts
int ibadv_i_max;                // max injrd blue advance threshold for ran constrnts
int ibclus_a_min;               // min alive blue cluster threshold for ran constrnts
int ibclus_a_max;               // max alive blue cluster threshold for ran constrnts
int ibclus_i_min;               // min injrd blue cluster threshold for ran constrnts
int ibclus_i_max;               // max injrd blue cluster threshold for ran constrnts
int ibcom_a_min;                // min alive blue combat threshold for ran constrnts
int ibcom_a_max;                // max alive blue combat threshold for ran constrnts
int ibcom_i_min;                // min injrd blue combat threshold for ran constrnts
int ibcom_i_max;                // max injrd blue combat threshold for ran constrnts
float zbfrombmindist_a;         // minimum distance of alive blue from blue
float zbfrombgmindist_a;        // minimum distance of alive blue from blue goal
float zrfrombmindist_a;         // minimum distance of alive red from blue
float zbfrombmindist_i;         // minimum distance of injured blue from blue
float zbfrombgmindist_i;        // minimum distance of injured blue from blue goal
float zrfrombmindist_i;         // minimum distance of injured red from blue
int ibluemoverange;             // max movement radius for alive blues
int blue_max_eng_num;           // max # of simul engagements by blue
int blue_COMM_flag;             // if = 0 then COMMs NOT used for blue, else yes
int ibcommrange;                // blue communications range
float bcommweight;              // blue COMM weight (relative to w=1)
float bcommweight_def;          // blue default COMM weight
float zbsscale;                 // scale factor for multiplying blue penalty
int blue_clock[MAXISAACNUM];    // internal blue clock (for reconstitution)
int blue_max_r_time;            // maximum number of 'ticks' before reconstitution
};
```

Main Module

```
//*************************************************************************************************
//
// ISAAC_GA.C - Simple Genetic Algorithm 'Evolver' for ISAAC
//
// Adapted from Z. Michalewicz, GA + Data Structures = EP, Springer-Verlag, 2nd Edition
//
// MS Visual C++ v1.52
// Version 1.5.1
//
// Andy Ilachinski
// Center for Naval Analyses
// 4401 Ford Avenue
// Alexandria, VA 22302
// (703) 824-2045
// ilachina@cna.org
//
//*************************************************************************************************

#include <ga.h>          // contains MISSION_OBJECTIVE, BATTLE, RED and BLUE parameter structures
#include <string.h>
#include <float.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
#include <graph.h>
#include <io.h>
#include <malloc.h>
#include <time.h>
#include <process.h>

#define CHROM_LENGTH_MAX 45      // number of genes in chromosome defining an ISAACA personality
#define PXOVER 0.8               // probability of crossover
#define PMUTATION 0.1            // probability of mutation

struct genotype // define a member template of the population
{
  double gene[CHROM_LENGTH_MAX+1]; // a string of variables
  double fitness;                  // genotype's fitness
  double upper[CHROM_LENGTH_MAX+1];// genotype's variables upper bound
  double lower[CHROM_LENGTH_MAX+1];// genotype's variables lower bound
  double rel_fitness;              // relative fitness
  double cum_fitness;              // cumulative fitness
};

//*************************************************************************************************
//
//                              Allocate Memory for Structures
//
//*************************************************************************************************
    struct battle_parameters battle;
    struct red_parameters red;
    struct blue_parameters blue;
    struct genotype population[POPSIZE+1];    // population
    struct genotype newpopulation[POPSIZE+1]; // new population to replace the old generation
    struct mission_objective mission;         // mission objective fitness measure parameters
    struct red_parameters red;

//*************************************************************************************************
//
//
//                                    FUNCTION PROTOTYPES
//
//
//*************************************************************************************************
void INPUT_FILE_DATA(unsigned char filename[30], struct battle_parameters *batp,
                     struct red_parameters *redp, struct blue_parameters *bluep, long *idum);
```

Appendix E: STATS_X.dat Data Fields

```
void WRITE_DATA_FILE(FILE *datafile, struct battle_parameters *batp,
                     struct red_parameters *redp, struct blue_parameters *bluep);

void WRITE_FITNESS(FILE *ga_stat, struct mission_objective *m, int termination_code,
                   int max_time_to_goal);

void PROMPTS(void);

void WRITE_CHROM_TO_FILE(int g, FILE *ga_stat, int initial_condition_genes_flag);

void SHOW_FITNESS(struct mission_objective *m, int termination_code, int max_time_to_goal);

void SHOW_GRAPHICS(struct battle_parameters *batp,  struct red_parameters *redp,
                   struct blue_parameters *bluep, struct mission_objective *m, int termination_code,
                   int max_time_to_goal);

void SHOW_CHROMOSOME(int mem, int initial_condition_genes_flag);

void INITIALIZE_FIELD(struct battle_parameters *batp, struct red_parameters *redp,
                      struct blue_parameters *bluep, int iflag[5][5], long *idum);

void ADAPT_RED_ISAACA_WEIGHTS(int i, int irinrnum, int ibinrnum, int iradvnum,
                              struct red_parameters *redp);

void ADAPT_BLUE_ISAACA_WEIGHTS(int i, int ibinbnum, int irinbnum, int ibadvnum,
                               struct blue_parameters *bluep);

void ADAPTBLUEWEIGHTS(int i, int ibinbnum, int irinbnum, int ibadvnum,
                      struct blue_parameters *bluep);

void RED_COMM_INFO(int i, struct red_parameters *redp, struct blue_parameters *bluep);

void BLUE_COMM_INFO(int i, struct red_parameters *redp, struct blue_parameters *bluep);

void MOVERED (int i, int imrr, int imove, struct battle_parameters *batp,
              struct red_parameters *redp);

void MOVEBLUE (int i, int imbr, int imove, struct battle_parameters *batp,
               struct blue_parameters *bluep);

void RED_SENSOR(int *irinrnum, int *ibinrnum, int *iradvnum, int *ibinrinjnum,
                int *irinrinjnum, int i, struct red_parameters *redp, struct blue_parameters *bluep,
                int **ilblbinr);

int BLUEINRED( int i, struct red_parameters *redp, struct blue_parameters *bluep,
               int **ilblbinr );

void BLUE_SENSOR(int *ibinbnum, int *irinbnum, int *ibadvnum, int *irinbinjnum,
                 int *ibinbinjnum, int i, struct red_parameters *redp,
                 struct blue_parameters *bluep, int **ilblrinb);

int REDINBLUE( int i, struct red_parameters *redp, struct blue_parameters *bluep,
               int **ilblrinb);

float COMPUTEREDPENALTY(int i, int imrr, int *igoalflag, int irinrinjnum, int ibinrinjnum,
                        float zmin, int iflag[5][5], float z[5][5], struct battle_parameters *batp,
                        struct red_parameters *redp, struct blue_parameters *bluep);

float COMPUTEREDPENALTY_COMM(int i, int imrr, int *igoalflag, int irinrinjnum,
                             int ibinrinjnum, float zmin, int iflag[5][5], float z[5][5],
                             struct battle_parameters *batp, struct red_parameters *redp,
                             struct blue_parameters *bluep);

float COMPUTEBLUEPENALTY(int i, int imbr, int *igoalflag, int irinbinjnum, int ibinbinjnum,
                         float zmin, int iflag[5][5], float z[5][5], struct battle_parameters *batp,
                         struct red_parameters *redp, struct blue_parameters *bluep);

float COMPUTEBLUEPENALTY_COMM(int i, int imbr, int *igoalflag, int irinbinjnum,
                              int ibinbinjnum, float zmin, int iflag[5][5], float z[5][5],
                              struct battle_parameters *batp, struct red_parameters *redp,
                              struct blue_parameters *bluep);

void COMBAT( struct battle_parameters *batp, struct red_parameters *redp,
             struct blue_parameters *bluep, long *idum, int **ilblbinr, int **ilblrinb);

void COMBAT_2( struct battle_parameters *batp, struct red_parameters *redp,
               struct blue_parameters *bluep, long *idum, int **ilblbinr, int **ilblrinb);

void CENTER_MASS(int iterations, struct mission_objective *m, struct red_parameters *redp,
                 struct blue_parameters *bluep, int *termination_flag, int max_CM_dist);
```

Appendix E: STATS_X.dat Data Fields

```
void GOAL_STATS(int iterations, struct mission_objective *m, int isize, struct red_parameters *redp,
                struct blue_parameters *bluep, int *termination_flag, int blue_in_RG_max, int red_in_BG_max);

void SCREEN_UPDATE(int itime, int iterations, int mem, int generation, int num_generations,
                   int num_initial_conds, int max_time_to_goal, double avg, double best_val,
                   double worst_val, double *best_fitness_gen, double *worst_fitness_gen,
                   double pf);

void DECODE_BEST(struct red_parameters *redp, int min_dist_genes_flag,
                 int initial_condition_genes_flag);

float abs_float(float x);

float getrandom(int x, int y, long *idum);

float ran1(long *idum);

int SIGN(double x);

void nomem();

void initialize(void);
double randval(double, double);
void penalty(struct mission_objective *m, struct red_parameters *redp);
void keep_the_best(int CHROM_LENGTH);
void elitist(int CHROM_LENGTH);
void select(void);
void crossover(int CHROM_LENGTH);
void Xover(int,int,int CHROM_LENGTH);
void swap(double *, double *);
void mutate(int CHROM_LENGTH);
void progress_report(int generation, FILE* ga_stat, double *best_val, double *worst_val, double *avg,
                     double best_fitness_gen, double worst_fitness_gen);

//*************************************************************************************************
//
//                                        MAIN: RUN ISAAC
//
//*************************************************************************************************
void main()
{

   int i, j;                    // loop variables
   int imx, imy;                // loop variables
   int ired, iblue;
   int im, imc;                 // labels for intermediate candidte moves
   int igoalflag;               // if =1 then red wins; if =2 then blue wins
   int imove;                   // labels selected move (1 <= imove <= 9)
   int ibinrnum;                // number of blue ISAACs in red sensor range
   int irinbnum;                // number of red ISAACs in blue sensor range
   int irinrnum;                // number of red ISAACs in red sensor range
   int ibinbnum;                // number of blue ISAACs in blue sensor range
   int iradvnum;                // threshold number of reds to advance

   int ibadvnum;                // threshold number of blues to advance
   int irinrinjnum;             // number of injured red in red sensor
   int ibinbinjnum;             // number of injured blue in blue sensor
   int irinbinjnum;             // number of injured red in blue sensor
   int ibinrinjnum;             // number of injured blue in red sensor
   int itime;                   // time counter
   int actual_time;
   int termination_flag;        // =1 if number of red near BF > threshold is to terminate run
   int imovecand[26];           // intermediate move candidates from which
                                // an ISAAC will choose an actual move
   int iflag[5][5];             // iflag=1 if a particular move represents a viable option

   int imrr;                    // = red.iredmoverange if alive, else = 1
   int imbr;                    // = blue.ibluemoverange if alive, else = 1
   int jj, icount;
   int bluelabel_randomized[MAXISAACNUM];
   int redlabel_randomized[MAXISAACNUM];
   int __huge **ilblrinb;       // jth red's (in blue's range) label
   int __huge **ilblbinr;       // jth blue's (in red's range) label
   int mem;
   int iterations;
   int num_generations;         // maximum number of GA generations
   int num_initial_conds;       // number of initial conditions to average over
   int max_time_to_goal;
   int min_time_to_goal;
   int max_CM_dist;
   time_t start_time, finish_time;
```

```
   int range, blue_in_RG_max, red_in_BG_max;
   int termination_code;       // 1: stop w/1 reaches flag, 2: >N near flag, 3: CM < r_CM, 4: no stop
   int show_fitness_flag;      // toggle to show fitness (=1) or no (=0)
   int show_graphics_flag;     // toggle to show graphics in small window (=1) or no (=0)
   int show_chromosome_flag;   // toggle to show chromosome of current personality (=1) or no (=0)
   int best_flag;              // =1 if best personalities are to be written to file, else =0
   int min_dist_genes_flag;    // =1 if minimum distance genes (36-42) are to be used, else =0
   int initial_condition_genes_flag;// =1 if initial condition genes (43-45) are to be used, else =0
   int CHROM_LENGTH;           // actual chromosome length to be used during run
   int min_containment_time_red;  // minimum time in which RED can reach BLUE flag-containment area
   int min_containment_time_blue; // minimum time in which BLUE can reach RED flag-containment area
   int min_containment_time ; // minimum of the two possible containment times

   double elapsed_time;
   double power;
   double best_val;            // best population fitness
   double worst_val;           // best population fitness
   double avg;                 // avg population fitness
   double best_fitness_gen;    // best fitness during current generation
   double worst_fitness_gen;   // worst fitness during current generation
   double znorm;
   double zz, t1, t3, t7, t17;
   long idum;                  // random number seed (dummy 'carry-over' variable)

   float zmin;                 // minimum seed used by penalty function
   float zmoveprob;
   float zran;                 // variable to catch initial ran number call
   float z[5][5];              // intermediate expected penalty function
   float zx, zy;
   double lower_bound, upper_bound;
   double previous_best_fitness;

   unsigned char buffer;
   char bb[20];
   unsigned char datafilename[30];  // name of input ISAAC data file
   unsigned char filename[30];      // name of input data file
   unsigned char outfilename[30];   // name of output file
   unsigned char outdatafile[30];   // name of output ISAACA.dat file
   unsigned char currentper[30];    // name of output ISAACA.dat file to store current personality
   unsigned char bestfilename[30];  // name of output file containing best personalities
   unsigned char fondir[_MAX_PATH];
   unsigned char list[20];
   struct _fontinfo fi;
   struct _videoconfig vc;
   short xfon;

   int generation;              // current generation no.
   FILE *current;               // output file
   FILE *ga_stat;               // output file
   FILE *infile;                // input GA data file
   FILE *datafile;              // input ISAAC data file
   FILE *bestfile;              // output best personalities
   FILE *outdata;               // output best personalities
   unsigned char s[44];
   unsigned char dbuffer [9];
   unsigned char tbuffer [9];

//*****************************************************************************
//
//                        Allocate Memory for Matrices
//
//*****************************************************************************
    ilblrinb = (int**) _fmalloc( (MAXISAACNUM+1) * sizeof(int*) );
    if ( !ilblrinb ) nomem();
    for ( i = 0; i < (MAXISAACNUM+1); i++ )
    {
        ilblrinb[i] = (int*) _fmalloc( (MAXNEIGHBORNUM+1) * sizeof(int) );
        if ( !ilblrinb[i] ) nomem();
    }

    ilblbinr = (int**) _fmalloc( (MAXISAACNUM+1) * sizeof(int*) );
    if ( !ilblbinr ) nomem();
    for ( i = 0; i < (MAXISAACNUM+1); i++ )
    {
        ilblbinr[i] = (int*) _fmalloc( (MAXNEIGHBORNUM+1) * sizeof(int) );
        if ( !ilblbinr[i] ) nomem();
    }

//*****************************************************************************
//
//                          Register and Set Fonts
```

```
//
//*****************************************************************************
    if( _registerfonts( "sserife.FON" ) <= 0 )
    {
        _outtext( "Enter full path where .FON files are located: " );
        gets( fondir );
        strcat( fondir, "\\*.FON" );
        if( _registerfonts( fondir ) <= 0 )
        {
            _outtext( "Error: can't register fonts" );
            exit( 1 );
        }
    }

//*****************************************************************************
//
//                                  Set Video Mode
//
//*****************************************************************************
    if ( !_setvideomode( _SRES16COLOR ) ){
          _setvideomode( _VRES16COLOR );
          _clearscreen( _GCLEARSCREEN );
    }
    _clearscreen( _GCLEARSCREEN );

//*****************************************************************************
//
//                                  Opening Screen
//
//*****************************************************************************
    _setbkcolor ( _BLUE );
    _clearscreen( _GCLEARSCREEN );
    _getvideoconfig( &vc );
    _setcolor( 15 );
    _moveto( 75, 80 );
    _rectangle_w( _GFILLINTERIOR, 100, 85, 700, 90 );
    _moveto( 565, 295 );
    _rectangle_w( _GFILLINTERIOR, 100, 297, 700, 302 );
    strcat( strcat( strcpy( list, "t'" ), "sserife"), "'");
    strcat( list, "h30w24b" );
    _getfontinfo( &fi );
    _setfont( list );
    _setcolor( 15 );
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "I S A A C" ) / 2);
    _moveto( xfon, 105 );
    _outgtext ("I S A A C");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Irreducible Semi-Autonomous" ) / 2);
    _moveto( xfon, 143 );
    _outgtext ("Irreducible Semi-Autonomous");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Adaptive Combat" ) / 2);
    _moveto( xfon, 173 );
    _outgtext ("Adaptive Combat");
    _unregisterfonts();
    _registerfonts( "oem10.FON" );
    strcat( strcat( strcpy( list, "t'" ), "oem10"), "'");
    strcat( list, "h30w24bv" );
    _setfont( list );
    _getfontinfo( &fi );
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "(Genetic Algorithm 'Evolver')" ) / 2);
    _moveto( xfon, 227 );
    _outgtext ("(Genetic Algorithm 'Evolver')");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Version 1.5.1" ) / 2);
    _moveto( xfon, 257 );
    _outgtext ("Version 1.5.1");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "7 April 1997" ) / 2);
    _moveto( xfon, 272 );
    _outgtext ("7 April 1997");
    _setcolor( 7 );
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Andy Ilachinski" ) / 2);
    _moveto( xfon, 350 );
    _outgtext ("Andy Ilachinski");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Center for Naval Analyses" ) / 2);
    _moveto( xfon, 365 );
    _outgtext ("Center for Naval Analyses");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "4401 Ford Avenue" ) / 2);
    _moveto( xfon, 380 );
    _outgtext ("4401 Ford Avenue");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "Alexandria, VA 22302" ) / 2);
    _moveto( xfon, 395 );
    _outgtext ("Alexandria, VA 22302");
        xfon = (vc.numxpixels / 2) - (_getgtextextent( "ilachina@cna.org" ) / 2);
```

```
_moveto( xfon, 410);
_outgtext ("ilachina@cna.org");
    xfon = (vc.numxpixels / 2) - (_getgtextextent( "Press <ENTER> to continue ....." ) / 2);
_moveto( xfon, 550 );
_outgtext ("Press <ENTER> to continue .....");

_getch();

_clearscreen( _GCLEARSCREEN );

// prompt for file names
_unregisterfonts();
_registerfonts( "sserife.FON" );
strcat( strcat( strcpy( list, "t'" ), "sserife"), "'");
strcat( list, "h30w24bv" );
_getfontinfo( &fi );
_setfont( list );
xfon = (vc.numxpixels / 2) - (_getgtextextent( "Specify Input Files" ) / 2);
_moveto( xfon, 80 );
_setcolor( 15 );
_outgtext ("Specify Input Files");
_settextposition( 10, 30);
_setcolor( 7 );
printf("ISAAC input (ga_isaac.dat):        ");
scanf("%s", &datafilename);

_settextposition( 12, 30);
printf("GA input (ga_data.dat):            ");
scanf("%s", &filename);

xfon = (vc.numxpixels / 2) - (_getgtextextent( "Specify Output Files" ) / 2);
_moveto( xfon, 275 );
_setcolor( 15 );
_outgtext ("Specify Output Files");
_settextposition( 22, 30);
_setcolor( 7 );
printf("GA summary output (ga_stat.dat):   ");
scanf("%s", &outfilename);

// default is to NOT show graphics on screen
show_graphics_flag = 0;

// default is to NOT show fitness on screen
show_fitness_flag = 0;

// default is to NOT show chromosome on screen
show_chromosome_flag = 0;

// initialize built-in time counter
_tzset();
time( &start_time );

// open output update file
if ( (ga_stat=fopen(outfilename,"w")) == NULL) {
    printf(" Cannot open GA data file\n");
    exit(1);
}

_strdate( dbuffer );
fprintf( ga_stat, "start date: %s \n", dbuffer );
_strtime( tbuffer );
fprintf( ga_stat, "start time: %s \n", tbuffer );

// initialize
if ( (infile=fopen(filename,"r")) == NULL) {
    fprintf(ga_stat,"\nCannot open input file\n");
    exit(1);
}

// read data
fscanf(infile, "%s", s);
fscanf(infile, "%s", s);
fscanf(infile, "%s%s%s", s, s, s);
fscanf(infile, "%s", s);
fscanf(infile, "%s", s);
fscanf(infile,"%s%i",s, &num_generations);
fscanf(infile,"%s%i",s, &num_initial_conds);
fscanf(infile,"%s%i",s, &max_time_to_goal);
fscanf(infile,"%s%lf",s, &power);
fscanf(infile,"%s%i",s, &best_flag);
    CHROM_LENGTH=35;
```

Appendix E: STATS_X.dat Data Fields

```
fscanf(infile,"%s%i",s, &min_dist_genes_flag);
    if (min_dist_genes_flag==1)CHROM_LENGTH=42;
fscanf(infile,"%s%i",s, &initial_condition_genes_flag);
    if (initial_condition_genes_flag==1)CHROM_LENGTH=45;
fscanf(infile, "%s", s);
fscanf(infile, "%s", s);
fscanf(infile, "%s%s%s%s", s, s, s, s);
fscanf(infile, "%s", s);
fscanf(infile, "%s", s);
fscanf(infile,"%s%lf",s, &mission.alpha_1); // minimize time to goal
fscanf(infile,"%s%lf",s, &mission.alpha_2); // minimize number of friendly losses
fscanf(infile,"%s%lf",s, &mission.alpha_3); // maximize number of enemy losses
fscanf(infile,"%s%lf",s, &mission.alpha_4); // maximize survival ratio  of alive to enemy forces
fscanf(infile,"%s%lf",s, &mission.alpha_5); // minimize cummulative distance between red CM and blue flag
fscanf(infile,"%s%lf",s, &mission.alpha_6); // maximize cummulative distance between blue CM and red flag
fscanf(infile,"%s%lf",s, &mission.alpha_7); // maximize number of friendly forces near enemy flag
fscanf(infile,"%s%lf",s, &mission.alpha_8); // minimize number of enemy forces near friendly flag
fscanf(infile,"%s%lf",s, &mission.alpha_9); // minimize total number of friendly fratricide hits
fscanf(infile,"%s%lf",s, &mission.alpha_10);// maximize total number of enemy fratricide hits
fscanf(infile, "%s", s);
fscanf(infile, "%s", s);
fscanf(infile, "%s%s%s", s, s, s);
fscanf(infile, "%s", s);
fscanf(infile, "%s", s);
fscanf(infile,"%s%i",s, &termination_code);
    // normalize penalty weights
    if (termination_code == 4)mission.alpha_1=0; // do not minimize time_to_goal if no termination
    znorm = mission.alpha_1 + mission.alpha_2 + mission.alpha_3 + mission.alpha_4 +
            mission.alpha_5 + mission.alpha_6 + mission.alpha_7 + mission.alpha_8 +
            mission.alpha_9 + mission.alpha_10;
    mission.alpha_1 = mission.alpha_1 / znorm;
    mission.alpha_2 = mission.alpha_2 / znorm;
    mission.alpha_3 = mission.alpha_3 / znorm;
    mission.alpha_4 = mission.alpha_4 / znorm;
    mission.alpha_5 = mission.alpha_5 / znorm;
    mission.alpha_6 = mission.alpha_6 / znorm;
    mission.alpha_7 = mission.alpha_7 / znorm;
    mission.alpha_8 = mission.alpha_8 / znorm;
    mission.alpha_9 = mission.alpha_9 / znorm;
    mission.alpha_10 = mission.alpha_10 / znorm;
fscanf(infile,"%s%lf",s, &mission.near_range);
fscanf(infile,"%s%lf",s, &mission.near_range_num);
fscanf(infile,"%s%lf",s, &mission.r_CM_f);
fscanf(infile, "%s", s);
fscanf(infile, "%s", s);
fscanf(infile, "%s%s%s", s, s, s);
fscanf(infile, "%s", s);
fscanf(infile, "%s", s);
for (i=1; i<=CHROM_LENGTH; i++){
    fscanf(infile, "%s%lf,%lf", s, &lower_bound, &upper_bound);
      for (j=0; j<POPSIZE; j++){
           population[j].fitness=0;
           population[j].rel_fitness=0;
           population[j].cum_fitness=0;
           population[j].lower[i]=lower_bound;
           population[j].upper[i]=upper_bound;
           population[j].gene[i]=randval(population[j].lower[i], population[j].upper[i]);
      }
}
fclose(infile);

if (best_flag==1){
    _settextposition( 24, 30);
    printf("GA 'best' output (ga_best.dat):    ");
    scanf("%s", &bestfilename);
    if ( (bestfile=fopen(bestfilename,"w")) == NULL) {
        fprintf(ga_stat,"\nCannot open 'best personalities' output file\n");
        exit(1);
    }
    WRITE_FITNESS(bestfile, &mission, termination_code, max_time_to_goal);
}

_settextposition( 26, 30);
printf("Output ISAAC-dat file (isaac.dat): ");
scanf("%s", &outdatafile);
if ( (outdata = fopen(outdatafile, "w")) == NULL ){
    printf(" Cannot open ISAAC output data file.\n");
        exit(1);
}

_unregisterfonts();
```

Appendix E: STATS_X.dat Data Fields

```
    _registerfonts( "oem10.FON" );
    strcat( strcat( strcpy( list, "t'" ), "oem10"), "'");
    strcat( list, "h30w24bv" );
    _getfontinfo( &fi );
    _setfont( list );

    _clearscreen( _GCLEARSCREEN );

    WRITE_FITNESS(ga_stat, &mission, termination_code, max_time_to_goal);
    fprintf(ga_stat, " generation   best      worst    best     worst   average   ave + standard  ave - standard
standard\n");
    fprintf(ga_stat, "   number     value     value    value    value   fitness     deviation       deviation
deviation\n");
    fprintf(ga_stat, "            (overall) (overall)  (gen)    (gen)    (gen)\n");

    _setvideomode( _SRES16COLOR );
    _setviewport(1,1,799,599);
    _clearscreen( _GCLEARSCREEN );

    // seed random number generator
    idum=-31415926;
    zran=ran1(&idum);
    srand( (unsigned)time( NULL ) );

    // set generation counter to zero
    generation = 0;
    best_val = 0;   // overall best
    worst_val = 1.; // overall worst
    avg = 0;
    previous_best_fitness = 0;

    // display on-screen prompts
    PROMPTS();

    while(generation < num_generations){

        // display generation on screen
        _setviewport(1,33,799,85);
        _clearscreen( _GVIEWPORT );
        _setviewport(1,1,799,599);
        _setcolor( 15 );
        _unregisterfonts();
        _registerfonts( "sserife.FON" );
        strcat( strcat( strcpy( list, "t'" ), "sserife"), "'");
        strcat( list, "h30w24bv" );
        _getfontinfo( &fi );
        _setfont( list );
           xfon = (vc.numxpixels / 2) - (_getgtextextent("GENERATION = XXX") / 2);
        _moveto( xfon, 47 );
        _setcolor( 2 );
        _outgtext ("GENERATION = ");
        _outgtext( itoa( generation+1, bb, 10) );
        _unregisterfonts();
        _registerfonts( "oem10.FON" );
        strcat( strcat( strcpy( list, "t'" ), "oem10"), "'");
        strcat( list, "h30w24bv" );
        _getfontinfo( &fi );
        _setfont( list );

        if (generation > 0){
            select();      // select survivors from population
            crossover(CHROM_LENGTH); // perform a single-point crossover
            mutate(CHROM_LENGTH);    // mutate a gene
            // initialize ave fitness
            avg=0;
            // update progress-report
            progress_report(generation, ga_stat, &best_val, &worst_val, &avg,
                            best_fitness_gen, worst_fitness_gen);
            if(best_flag==1){ // then write best personality genes to file
                if (population[POPSIZE].fitness > previous_best_fitness){
                    fprintf(bestfile,"\n");
                    fprintf(bestfile,"Generation = %3i \n",generation);
                    fprintf(bestfile,"Fitness=%4.4f",population[POPSIZE].fitness);
                    WRITE_CHROM_TO_FILE(POPSIZE, bestfile, initial_condition_genes_flag);
                    previous_best_fitness = population[POPSIZE].fitness;
                }
            }
        }

        // initialize best and worst fitness for current generation
        best_fitness_gen = 0;
```

Appendix E: STATS_X.dat Data Fields

```
worst_fitness_gen = 1.;

////////////////////////////////////////////////////////////////////////////////
//
//                calculate "mission penalty" for nth generation;
//                i.e. run ISAAC
//
////////////////////////////////////////////////////////////////////////////////
for (mem=0; mem<POPSIZE; mem++){ // do for each member of the population

    // first read-in default values
    if ( (datafile=fopen(datafilename,"r")) == NULL) {
        printf(" Cannot open ISAAC data file\n");
        exit(1);
    }

    INPUT_FILE_DATA(datafilename, &battle, &red, &blue, &idum);

    fclose(datafile);

    // now re-define ISAACA force with genome-prescribed personality
    red.irsrange = (int)(population[mem].gene[1]);
      if (red.irsrange == 0)red.irsrange=1;
    red.iredfrange = (int)(population[mem].gene[2]);
      if (red.iredfrange == 0)red.iredfrange=1;
      // make sure F-range <= S-range
      if (red.iredfrange > red.irsrange)red.iredfrange = red.irsrange;
    red.iradvrange[1] = (int)(population[mem].gene[3]);
      if (red.iradvrange[1] == 0)red.iradvrange[1]=1;
    red.w1rdeff_a = (float)(population[mem].gene[4]);
      if (population[mem].gene[5] < .5)red.w1rdeff_a = -red.w1rdeff_a;
    red.w2rdeff_a = (float)(population[mem].gene[6]);
      if (population[mem].gene[7] < .5)red.w2rdeff_a = -red.w2rdeff_a;
    red.w3rdeff_a = (float)(population[mem].gene[8]);
      if (population[mem].gene[9] < .5)red.w3rdeff_a = -red.w3rdeff_a;
    red.w4rdeff_a = (float)(population[mem].gene[10]);
      if (population[mem].gene[11] < .5)red.w4rdeff_a = -red.w4rdeff_a;
    red.w5rdeff_a = (float)(population[mem].gene[12]);
      if (population[mem].gene[13] < .5)red.w5rdeff_a = -red.w5rdeff_a;
    red.w6rdeff_a = (float)(population[mem].gene[14]);
      if (population[mem].gene[15] < .5)red.w6rdeff_a = -red.w6rdeff_a;
    red.w1rdeff_i = (float)(population[mem].gene[16]);
      if (population[mem].gene[17] < .5)red.w1rdeff_i = -red.w1rdeff_i;
    red.w2rdeff_i = (float)(population[mem].gene[18]);
      if (population[mem].gene[19] < .5)red.w2rdeff_i = -red.w2rdeff_i;
    red.w3rdeff_i = (float)(population[mem].gene[20]);
      if (population[mem].gene[21] < .5)red.w3rdeff_i = -red.w3rdeff_i;
    red.w4rdeff_i = (float)(population[mem].gene[22]);
      if (population[mem].gene[23] < .5)red.w4rdeff_i = -red.w4rdeff_i;
    red.w5rdeff_i = (float)(population[mem].gene[24]);
      if (population[mem].gene[25] < .5)red.w5rdeff_i = -red.w5rdeff_i;
    red.w6rdeff_i = (float)(population[mem].gene[26]);
      if (population[mem].gene[27] < .5)red.w6rdeff_i = -red.w6rdeff_i;
    red.iradv_a[1] = (int)(population[mem].gene[28]);
    red.irclus_a[1] = (int)(population[mem].gene[29]);
    red.ircom_a[1] = (int)(population[mem].gene[30]);
      if (population[mem].gene[31] < .5)red.ircom_a[1] = -red.ircom_a[1];
    red.iradv_i[1] = (int)(population[mem].gene[32]);
    red.irclus_i[1] = (int)(population[mem].gene[33]);
    red.ircom_i[1] = (int)(population[mem].gene[34]);
      if (population[mem].gene[35] < .5)red.ircom_i[1] = -red.ircom_i[1];
    if (min_dist_genes_flag==1){
      if (population[mem].gene[36] > .5){ // the  use min dist genes (37-42)
          red.zrfromrmindist_a= (float)(population[mem].gene[37]);
          red.zbfromrmindist_a= (float)(population[mem].gene[38]);
          red.zrfromrgmindist_a= (float)(population[mem].gene[39]);
          red.zrfromrmindist_i= (float)(population[mem].gene[40]);
          red.zbfromrmindist_i= (float)(population[mem].gene[41]);
          red.zrfromrgmindist_i= (float)(population[mem].gene[42]);
      }
      else{
          red.zrfromrmindist_a= 0;
          red.zbfromrmindist_a= 0;
          red.zrfromrgmindist_a= 0;
          red.zrfromrmindist_i= 0;
          red.zbfromrmindist_i= 0;
          red.zrfromrgmindist_i= 0;
      }
    }
    else{
      population[mem].gene[36] = 0;
```

```
  population[mem].gene[37] = 0;
  population[mem].gene[38] = 0;
  population[mem].gene[39] = 0;
  population[mem].gene[40] = 0;
  population[mem].gene[41] = 0;
  population[mem].gene[42] = 0;
  red.zrfromrmindist_a= 0;
  red.zbfromrmindist_a= 0;
  red.zrfromrgmindist_a= 0;
  red.zrfromrmindist_i= 0;
  red.zbfromrmindist_i= 0;
  red.zrfromrgmindist_i= 0;
}

if (initial_condition_genes_flag==1){
    if ( population[mem].gene[43] <= sqrt(battle.isize)+1 ){
          battle.ibattlebox_red_length = (int)(sqrt(battle.isize)) + 2;
          battle.ibattlebox_red_width = (int)(sqrt(battle.isize)) + 2;
    }
    else{
        battle.ibattlebox_red_length = (int)(population[mem].gene[43]);
        battle.ibattlebox_red_width = (int)(population[mem].gene[43]);
    }
    battle.ibattlebox_red_cen_x = (int)(population[mem].gene[44]);
    battle.ibattlebox_red_cen_y = (int)(population[mem].gene[45]);
}
else{
    population[mem].gene[43] = 0;
    population[mem].gene[44] = 0;
    population[mem].gene[45] = 0;
}

// overwrite the ISAACA data file with new data
if ( (datafile = fopen(datafilename, "w")) == NULL ){
  printf(" Cannot open ISAAC data file.\n");
  exit(1);
}

WRITE_DATA_FILE(datafile, &battle, &red, &blue);

// now read-in and initialize using updated system values
if ( (datafile=fopen(datafilename,"r")) == NULL) {
    printf(" Cannot open ISAAC data file.\n");
    exit(1);
}

INPUT_FILE_DATA(datafilename, &battle, &red, &blue, &idum);

// find minimal possible time to goal
zx= (float)(red.redgoalx) -
      (float)(battle.ibattlebox_red_cen_x) + (float)(.5*battle.ibattlebox_red_length);
zy= (float)(red.redgoaly) -
      (float)(battle.ibattlebox_red_cen_y) + (float)(.5*battle.ibattlebox_red_width);
min_time_to_goal = (int)( sqrt( zx*zx + zy*zy) ) - 1;
max_CM_dist = (int)( 1.4142135 * battle.isize) + 1;

// find maximum number of near-goal ISAACAs
blue_in_RG_max = 0;
red_in_BG_max = 0;
range = (int)(mission.near_range);
for (i=blue.bluegoalx-range; i<=blue.bluegoalx+range; i++){
    for (j=blue.bluegoaly-range; j<=blue.bluegoaly+range; j++){
       if (i<=battle.isize && i>=1 && j<=battle.isize && j>=1) ++blue_in_RG_max;
    }
}
if (blue_in_RG_max > blue.ibluenum)blue_in_RG_max=blue.ibluenum;
for (i=red.redgoalx-range; i<=red.redgoalx+range; i++){
    for (j=red.redgoaly-range; j<=red.redgoaly+range; j++){
       if (i<=battle.isize && i>=1 && j<=battle.isize && j>=1) ++red_in_BG_max;
    }
}
if (red_in_BG_max > red.irednum)red_in_BG_max=red.irednum;

// find minimum time in which RED can reach BLUE flag-containment area
zx= (float)(red.redgoalx - range) -
      (float)(battle.ibattlebox_red_cen_x) + (float)(.5*battle.ibattlebox_red_length);
zy= (float)(red.redgoaly - range) -
      (float)(battle.ibattlebox_red_cen_y) + (float)(.5*battle.ibattlebox_red_width);
min_containment_time_red = (int)( sqrt( zx*zx + zy*zy) ) - 1;
zx= (float)(blue.bluegoalx + range) -
      ((float)(battle.ibattlebox_blue_cen_x) - (float)(.5*battle.ibattlebox_blue_length));
```

```
                zy= (float)(blue.bluegoaly + range) -
                    ((float)(battle.ibattlebox_blue_cen_y) - (float)(.5*battle.ibattlebox_blue_width));
                min_containment_time_blue = (int)( sqrt( zx*zx + zy*zy) ) - 1;
                min_containment_time = __min(min_containment_time_red, min_containment_time_blue);

                fclose(datafile);

                ////////////////////////////////////////////////////////////////////////////
                //
                //          Now run ISAAC and compute "mission objective" penalty
                //          average over num_initial_conds
                //
                ////////////////////////////////////////////////////////////////////////////
                for (iterations=1; iterations<=num_initial_conds; iterations++){

                     // initialize fratricide counters
                     battle.red_frat_count=0;
                     battle.blue_frat_count=0;

                     // initial GOAL_STATS counters
                     mission.red_near_BF[iterations] = 0;
                     mission.blue_near_RF[iterations] = 0;

                     // initialize combat battlefield
                     INITIALIZE_FIELD(&battle, &red, &blue, iflag, &idum);

                    // initialize time counter
                    itime = 0;

                    ////////////////////////////////////////////////////////////////////////////
                    //
                    //                          START MAIN DYNAMICS LOOP
                    //
                    ////////////////////////////////////////////////////////////////////////////
start:              ++itime;  // increment time counter

                    // dump progress report to screen
                    SCREEN_UPDATE(itime, iterations, mem, generation, num_generations,
                                  num_initial_conds, max_time_to_goal, avg, best_val,
                                  worst_val, &best_fitness_gen, &worst_fitness_gen,
                                  population[mem-1].fitness);

                    // check to see if graphics are to be displayed on screen
                    if (show_graphics_flag == 1){
                        SHOW_GRAPHICS(&battle, &red, &blue, &mission, termination_code,
                                      max_time_to_goal);
                    }

                    // check to see if GOAL_STATS are to be calculated and updated
                    if ( mission.alpha_7 != 0 || // friendly_near_enemy_F weight is not zero
                         mission.alpha_8 != 0 || // enemy_near_friendly_F weight is not zero
                         termination_code == 2   // terminate run if number of ISAACAs > N
                         ){ // do only if min_time_to_containment has been reached
                               if (itime >= min_containment_time){
                                   // initialize termination flag; if =1 after then = 1
                                   termination_flag = 0;
                                   GOAL_STATS(iterations, &mission, battle.isize, &red, &blue,
                                              &termination_flag, blue_in_RG_max, red_in_BG_max);
                                   // has termination condition been satisfied?
                                   if (termination_code == 2 && termination_flag == 1) goto goal;
                               }
                    }

                    // check to see if CENTER_MASS is to be calculated and updated
                    if ( mission.alpha_5 != 0 || // friendly_CM_to_enemy_flag weight is not zero
                         mission.alpha_6 != 0 || // enemy_CM_to_friendly_flag weight is not zero
                         termination_code == 3   // terminate run if CM within range r_CM
                         ){
                               // initialize termination flag; if =1 after then = 1
                               termination_flag = 0;
                               CENTER_MASS(iterations, &mission, &red, &blue, &termination_flag,
                                           max_CM_dist);
                               // has termination condition been satisfied?
                               if (termination_code == 3 && termination_flag == 1) goto goal;
                    }

                    // has maximum time been reached?
                    if (itime == max_time_to_goal) goto goal;

                     //*************************************************************************
                     //
```

```
                    //     Should order of move selection be shuffled during each iteration?
                    //
                    //*************************************************************************
                     if (battle.imove_selection == 1){  // select moves in fixed order
                        for (j=1; j<=red.irednum; ++j){
                            redlabel_randomized[j] = j;
                        }
                     }
                     else{
                        //*********************************************************************
                        //
                        //      Randomize order in which to consider moves for red ISAACAs:
                        //      'i' is the actual label and the array redlabel_randomized[j] = i
                        //
                        //*********************************************************************
                        icount=0;
                        for (j=1; j<=red.irednum; ++j){
                            // select random label between 1 and red.irednum
newired:                    i = (int)(getrandom( 0, red.irednum, &idum ))+1;
                            // test to see if label has already been used
                            for (jj=1; jj<=icount; ++jj){
                                if (redlabel_randomized[jj] == i) goto newired;
                            }
                            ++icount;
                            redlabel_randomized[j] = i;
                        }
                     }

                     if (battle.imove_selection == 1){  // select moves in fixed order
                        for (j=1; j<=blue.ibluenum; ++j){
                            bluelabel_randomized[j] = j;
                        }
                     }
                     else{
                        //*********************************************************************
                        //
                        //      Randomize order in which to consider moves for blue ISAACAs:
                        //      'i' is the actual label and the array bluelabel_randomized[j] = i
                        //
                        //*********************************************************************
                        icount=0;
                        for (j=1; j<=blue.ibluenum; ++j){
                            // select random label between 1 and red.irednum
newiblue:                   i = (int)(getrandom( 0, blue.ibluenum, &idum ))+1;
                            // test to see if label has already been used
                            for (jj=1; jj<=icount; ++jj){
                                if (bluelabel_randomized[jj] == i) goto newiblue;
                            }
                            ++icount;
                            bluelabel_randomized[j] = i;
                        }
                     }

                      if( _kbhit() ){
                        buffer = _getch();
                        switch (buffer) {
                            case 'c': // chromosome toggle
                                show_chromosome_flag = (show_chromosome_flag + 1) % 2;
                                if ( show_chromosome_flag==0){
                                    _setviewport(555,151,799,580);
                                    _clearscreen( _GVIEWPORT );
                                    _setviewport(1,1,799,599);
                                }
                                else{ // show chromosome
                                    SHOW_CHROMOSOME(mem, initial_condition_genes_flag);
                                }
                                break;
                            case 's': // store current chromosome to file
                                _setviewport(260,575,680,590);
                                _clearscreen( _GVIEWPORT );
                                _setviewport(1,1,799,599);
                                //_moveto( 270, 562 );
                                _setcolor( 14 );
                                xfon = (vc.numxpixels / 2) -
                                    (_getgtextextent( "Output-data file name ? " ) / 2);
                                _moveto( xfon, 562 );
                                _outgtext ("Output-data file name ? ");
                                //_outgtext("       Output-data file name ? ");
                                _settextposition( 36, 66 );
                                scanf("%s", &currentper);
                                if ( (current = fopen(currentper, "w")) == NULL ){
```

```
                printf(" Cannot open ISAAC output data file.\n");
                    exit(1);
                }
                // dump current personality to file
                WRITE_DATA_FILE(current, &battle, &red, &blue);
                //WRITE_CHROM_TO_FILE(mem, current, initial_condition_genes_flag);
                fclose(current);
                _setviewport(1,550,799,580);
                _clearscreen( _GVIEWPORT );
                _setviewport(1,1,799,599);
                break;
            case 'f': // graphics toggle
                show_fitness_flag = (show_fitness_flag + 1) % 2;
                if ( show_fitness_flag==0){
                    _setviewport(1,200,290,580);
                    _clearscreen( _GVIEWPORT );
                    _setviewport(1,1,799,599);
                }
                else{ // show fitness
                    SHOW_FITNESS(&mission, termination_code, max_time_to_goal);
                }
                break;
            case 'b': // graphics toggle
                show_graphics_flag = (show_graphics_flag + 1) % 2;
                if ( show_graphics_flag==0){
                    _setviewport(291,200,551,580);
                    _clearscreen( _GVIEWPORT );
                    _setviewport(1,1,799,599);
                }
                break;
            case 'q': // quit
                time( &finish_time );
                elapsed_time = difftime( finish_time, start_time );
                if (generation == 0){
                    keep_the_best(CHROM_LENGTH);  // identify "best" ISAACA personality
                }
                else{
                    elitist(CHROM_LENGTH);  // find the "best" ISAACA personality
                }

                // decode current best chromosome to ISAACA personality
                DECODE_BEST(&red, min_dist_genes_flag, initial_condition_genes_flag);

                WRITE_DATA_FILE(outdata, &battle, &red, &blue);

                // dump best current personality to file
                fprintf(ga_stat,"\n Interim best personality: \n");
                WRITE_CHROM_TO_FILE(POPSIZE, ga_stat, initial_condition_genes_flag);

                fprintf(ga_stat,"\n\n Best fitness=%4.4f",population[POPSIZE].fitness);
                fprintf(ga_stat,"\n\n");
                _strdate( dbuffer );
                fprintf( ga_stat,"start date: %s \n", dbuffer );
                _strtime( tbuffer );
                fprintf( ga_stat,"start time: %s \n", tbuffer );
                fprintf( ga_stat,"elapsed time: %6.0f seconds \n", elapsed_time );
                fclose(ga_stat);
                _setviewport(260,565,680,580);
                _clearscreen( _GVIEWPORT );
                _setviewport(1,1,799,599);
                _moveto( 270, 565 );
                _setcolor( 14 );
                _setgtextvector( 1, 0 );
                _outgtext("            Run Terminated");
                _getch();
                exit(1);
                if (best_flag==1)fclose(bestfile);
                break;
        }
    }

    //*************************************************************************
    //
    //          Determine what blue isaacs are in red's neighborhood
    //
    //*************************************************************************
    for (i=1; i<=red.irednum; ++i) {
       ibinrnum = BLUEINRED( i, &red, &blue, ilblbinr );
    }

    //*************************************************************************
```

```
//
//          Determine what red isaacs are in blue's neighborhood
//
//*****************************************************************************
for (i=1; i<=blue.ibluenum; ++i) {
   irinbnum = REDINBLUE( i, &red, &blue, ilblrinb);
}

//*****************************************************************************
//
//                    Adjudicate combat attrition
//
//*****************************************************************************
if (battle.max_combat_flag == 1){ // then no limit on number of
                                  // simultaneous engagements

   COMBAT( &battle, &red, &blue, &idum, ilblbinr, ilblrinb);
}
else{ // use routine that puts limit on the number of
      // simultaneous engagements
   COMBAT_2( &battle, &red, &blue, &idum, ilblbinr, ilblrinb);
}

//*****************************************************************************
//
//                              UPDATE RED ISAACAs
//
//*****************************************************************************

for (j=1; j<=red.irednum; ++j) {

   //************************************************************************
   //
   //                          Get randomized label
   //
   //************************************************************************
   i = redlabel_randomized[j];

   //************************************************************************
   //
   //                Do only if red ISAAC is alive or injured
   //
   //************************************************************************
   if ( red.rstatus[i] > 0 )  {

          irinrnum = 0;
          ibinrnum = 0;
          iradvnum = MAXISAACNUM;
          ibinrinjnum = 0;
          irinrinjnum = 0;

          RED_SENSOR(&irinrnum, &ibinrnum, &iradvnum, &ibinrinjnum, &irinrinjnum, i,
                          &red, &blue, ilblbinr);

          //*********************************************************************
          //
          //   Adapt red weights; i.e. determine values for red.w1red, red.w2red,
          //   red.w3red, red.w4red, red.w5red, red.w6red to be used for this time step
          //
          //*********************************************************************
          ADAPT_RED_ISAACA_WEIGHTS(i, irinrnum, ibinrnum, iradvnum, &red);

          //*********************************************************************
          //
          //              Are communications to be used between ISAACAs?
          //
          //*********************************************************************
          if (red.red_COMM_flag != 0){ // if COMMs 'on' then get COMM data
              RED_COMM_INFO(i, &red, &blue);
          }

          //*********************************************************************
          //
          //          Compute expected penalty for each possible move;
          //          isaac's move will be into square with least penalty
          //
          //*********************************************************************

          igoalflag=0; // if remains equal to 0 then goal not reached

          //*********************************************************************
```

```
//
//                      Initialize minimum sum value
//
//*****************************************************************************
zmin = (float)(99999.);

// get movement range
imrr = red.iredmoverange;
if (red.rstatus[i] == 1) imrr = 1; // if injured, make sure max range equals 1


if (red.red_COMM_flag == 1){ // use 'COMM' routine
    zmin = COMPUTEREDPENALTY_COMM(i, imrr, &igoalflag, irinrinjnum, ibinrinjnum,
              zmin, iflag, z, &battle, &red, &blue);
}
else{
    zmin = COMPUTEREDPENALTY(i, imrr, &igoalflag, irinrinjnum, ibinrinjnum,
                  zmin, iflag, z, &battle, &red, &blue);
}

if ( igoalflag == 1 && termination_code == 1) {
  goto goal;
}

//*****************************************************************************
//
//     If zmin = 99999 then there are no viable moves --> do nothing
//
//*****************************************************************************
if ( zmin == 99999. ){
//
// do nothing
//
  if (imrr == 1){
      imove = 5;
  }
  else{
      imove = 13;
  }
}
else{
//*****************************************************************************
//
//         See what possible local moves correspond to zmin
//
//*****************************************************************************
  imc = 0; // initialize local count variable
  for (imx = - imrr; imx <= imrr; ++imx) {
      for (imy = - imrr; imy <= imrr; ++imy) {
          if (iflag[imx + 2][imy + 2] == 1 &&
              z[imx + 2][imy + 2] == zmin){
              // add another candidate move to count
              ++imc;
              //                  select candidate move
              if (imrr == 1){
                  imovecand[imc] = imx + 5 - 3 * imy;
              }
              else{
                  imovecand[imc] = imx + 13 - 5 * imy;
              }
          }
      }
  }

//*****************************************************************************
//
//      Actual move is randomly selected from among the imc candidates
//
//*****************************************************************************
    if (imc == 1){
        imove = imovecand[1];
        }
    else{
        zmoveprob = (float)(0.0001 * getrandom(1,10000,&idum));
        for (im = 1; im < imc + 1; ++im) {
            if (zmoveprob > (float)(im - 1) / (float)(imc) &&
                zmoveprob <= (float)(im) / (float)(imc) ){
                imove = imovecand[im];
            }
        }
    }
```

```
        }

        //*************************************************************************
        //
        //          Move red to new square for which penalty is minimum
        //
        //*************************************************************************
          MOVERED (i, imrr, imove, &battle, &red);

   } // end if red.rstatus[i]!=0 test

} // end i = 1 to red.irednum loop

//*************************************************************************
//
//                          UPDATE BLUE ISAACAs
//
//*************************************************************************

for (j=1; j<=blue.ibluenum; ++j) {

   // get randomized label
   i = bluelabel_randomized[j];

   // do only if blue ISAAC is alive or injured
   if ( blue.bstatus[i] > 0 ){

          ibinbnum = 0;
          irinbnum = 0;
          ibadvnum = MAXISAACNUM;
          irinbinjnum = 0;
          ibinbinjnum = 0;

          BLUE_SENSOR(&ibinbnum, &irinbnum, &ibadvnum, &irinbinjnum, &ibinbinjnum, i,
                      &red, &blue, ilblrinb);

          // adapt blue weights
          ADAPT_BLUE_ISAACA_WEIGHTS(i, ibinbnum, irinbnum, ibadvnum, &blue);

          // are communications to be used between ISAACAs?
          if (blue.blue_COMM_flag != 0){ // if COMMs 'on' then get COMM data
             BLUE_COMM_INFO(i, &red, &blue);
          }

          // compute expected penalty for each possible move;

          // initialize minimum sum value
          zmin = (float)(99999.);

          // get movement range
          imbr = blue.ibluemoverange;
          if (blue.bstatus[i] == 1) imbr = 1; // if injured, make sure max range equals 1

          if (blue.blue_COMM_flag == 1){ // use 'COMM' routine
              zmin = COMPUTEBLUEPENALTY_COMM(i, imbr, &igoalflag, irinbinjnum, ibinbinjnum,
                                             zmin, iflag, z, &battle, &red, &blue);
          }
          else{
              zmin = COMPUTEBLUEPENALTY(i, imbr, &igoalflag, irinbinjnum, ibinbinjnum,
                                        zmin, iflag, z, &battle, &red, &blue);
          }

          // if zmin = 99999 then there are no viable moves --> do nothing
          if ( zmin == 99999. ){
          // do nothing
            if (imbr == 1){
                imove = 5;
            }
            else{
                imove = 13;
            }
          }
          else{
          // see what possible local moves correspond to zmin
            imc = 0;
            for (imx = -imbr; imx <= imbr; ++imx) {
                for (imy = -imbr; imy <= imbr; ++imy) {
                  if (iflag[imx + 2][imy + 2] == 1 &&
                     z[imx + 2][imy + 2] == zmin){
                     // add another candidate move to count
                     ++imc;
```

```
                                //                  select candidate move
                                if (imbr == 1){
                                    imovecand[imc] = imx + 5 - 3 * imy;
                                }
                                else{
                                    imovecand[imc] = imx + 13 - 5 * imy;
                                }
                              }
                            }
                        }

                    // actual move is randomly selected from among the imc candidates
                        if (imc == 1){
                            imove = imovecand[1];
                        }
                        else{
                            zmoveprob = (float)(0.0001 * getrandom(1,10000,&idum));
                            for (im=1; im<imc+1; ++im) {
                                if (zmoveprob > (float)(im - 1) / (float)(imc) &&
                                    zmoveprob <= (float)(im) / (float)(imc) ){
                                    imove = imovecand[im];
                                }
                            }
                         }
                      } // end if zmin=99999

                    // move red to new square for which penalty is minimum
                      MOVEBLUE (i, imbr, imove, &battle, &blue);

                  } // end if blue.bstatus[i]!=0 test

                } // end i = 1 to blue.ibluenum loop

                goto start;

                ////////////////////////////////////////////////////////////////////////////
                //
                //                    Run is terminated: calculate penalty
                //
                ////////////////////////////////////////////////////////////////////////////
goal:           actual_time = itime;
                if (itime < min_time_to_goal)itime = min_time_to_goal;

                // time to reach blue flag
                mission.time_to_goal[iterations] =
                    pow( (double)(max_time_to_goal - itime)/
                         (double)(max_time_to_goal - min_time_to_goal), power );
                // count reds
                ired=0;
                for (i=1; i<=red.irednum; i++){
                    if (red.rstatus[i] > 0){
                        ++ired;
                    }
                }
                // count blues
                iblue=0;
                for (i=1; i<=blue.ibluenum; i++){
                    if (blue.bstatus[i] > 0){
                        ++iblue;
                    }
                }

                // number of friendly ISAACAs remaining
                mission.total_friendly_loss[iterations] =
                    pow( (double)(ired)/(double)(red.irednum), power);

                // number of enemy ISAACAs killed
                mission.total_enemy_loss[iterations] =
                    pow( (double)(blue.ibluenum - iblue)/(double)(blue.ibluenum), power);

                // normalized ratio of surviving ISAACAs
                if (iblue==0) iblue=1;
                t1 = (double)(1.0/ (double)(red.irednum));
                t3 = (double)(1.0/(double)(iblue));
                t7 = (double)(1.0/(double)(blue.ibluenum-1.0));
                t17 = (double)(ired*t1*t3*blue.ibluenum/((blue.ibluenum-2.0)*t7*ired*t1*t3*blue.ibluenum+
                              blue.ibluenum*t7));
                mission.survival_ratio[iterations] = pow( t17 , power);

                // center of mass distance of RED from opposing flag
                if (mission.alpha_5 != 0){
```

```
                mission.red_CM_to_BF_dist[iterations] =
                    pow( (1. - mission.red_CM_to_BF_dist[iterations]/(double)(actual_time)), power);
            }
            else{
                mission.red_CM_to_BF_dist[iterations] = 0;
            }

            // center of mass distance of BLUE from opposing flag
            if (mission.alpha_6 != 0){
                mission.blue_CM_to_RF_dist[iterations] =
                    pow( mission.blue_CM_to_RF_dist[iterations]/(double)(actual_time), power);
            }
            else{
                mission.blue_CM_to_RF_dist[iterations] = 0;
            }

            // average number of RED ISAACAs near within opposing flag area
            // (take average over actual_time - min_containment_time
            if (mission.alpha_7 != 0){
                // average over evolution steps
                mission.red_near_BF[iterations] =
                    pow(mission.red_near_BF[iterations]/
                    (double)(actual_time- min_containment_time),power);
            }
            else{
                mission.red_near_BF[iterations] = 0;
            }

            // average number of BLUE ISAACAs near within opposing flag area
            // (take average over actual_time - min_containment_time
            if (mission.alpha_7 != 0 || mission.alpha_8 != 0){
                // average over evolution steps
                mission.blue_near_RF[iterations] =
                    pow( (1. - mission.blue_near_RF[iterations]/
                    (double)(actual_time- min_containment_time)),power);
            }
            else{
                mission.blue_near_RF[iterations] = 0;
            }

            // fratricide
            if (battle.red_fratricide_flag==1){
                zz = (double)(battle.red_frat_count)/(double)(red.irednum);
                if (zz > 1)zz=1.;
                mission.red_fratricide[iterations] = pow( 1. - zz, power);
            }
            else{
                mission.red_fratricide[iterations] = 0;
            }
            if (battle.blue_fratricide_flag==1){
                zz = (double)(battle.blue_frat_count)/(double)(blue.ibluenum);
                if (zz > 1)zz=1.;
                mission.blue_fratricide[iterations] = pow( zz, power);
            }
            else{
                mission.blue_fratricide[iterations] = 0;
            }

        } // iterations loop

          // calculate "mission fitness" (to maximize)
          // average over num_initial_conds for given personality
          population[mem].fitness = 0;
          for (iterations=1; iterations<=num_initial_conds; iterations++){
               population[mem].fitness = population[mem].fitness +
                    mission.alpha_1 * mission.time_to_goal[iterations] +
                    mission.alpha_2 * mission.total_friendly_loss[iterations] +
                    mission.alpha_3 * mission.total_enemy_loss[iterations] +
                    mission.alpha_4 * mission.survival_ratio[iterations] +
                    mission.alpha_5 * mission.red_CM_to_BF_dist[iterations] +
                    mission.alpha_6 * mission.blue_CM_to_RF_dist[iterations] +
                    mission.alpha_7 * mission.red_near_BF[iterations] +
                    mission.alpha_8 * mission.blue_near_RF[iterations] +
                    mission.alpha_9 * mission.red_fratricide[iterations] +
                    mission.alpha_10 * mission.blue_fratricide[iterations];
          }
          population[mem].fitness = population[mem].fitness / (double)(num_initial_conds);
    }

    if (generation == 0){
        keep_the_best(CHROM_LENGTH);  // identify "best" ISAACA personality
```

```
        }
        else{
            elitist(CHROM_LENGTH);  // find the "best" ISAACA personality
        }

        generation++;              // update generation counter
    }  // generation loop

    time( &finish_time );
    elapsed_time = difftime( finish_time, start_time );

    fprintf(ga_stat,"\n\n Simulation completed\n");
    fprintf(ga_stat,"\n Best personality: \n");

    WRITE_CHROM_TO_FILE(POPSIZE, ga_stat, initial_condition_genes_flag);

    fprintf(ga_stat,"\n\n Best fitness=%4.4f",population[POPSIZE].fitness);
    fprintf(ga_stat,"\n\n");
    _strdate( dbuffer );
    fprintf( ga_stat,"start date: %s \n", dbuffer );
    _strtime( tbuffer );
    fprintf( ga_stat,"start time: %s \n", tbuffer );
    fprintf( ga_stat,"elapsed time: %6.0f seconds \n", elapsed_time );
    fclose(ga_stat);
    if (best_flag==1)fclose(bestfile);

    // decode current best chromosome to ISAACA personality
    DECODE_BEST(&red, min_dist_genes_flag, initial_condition_genes_flag);

    WRITE_DATA_FILE(outdata, &battle, &red, &blue);

    fclose(outdata);

    _settextposition( 16, 30);
    printf("                Run Complete");

}

//*****************************************************************************
//
//                    Random number generator:
//                    Generates a value within bounds
//
//*****************************************************************************
double randval(double low, double high)
{
    double val;
    val=((double)(rand()%1000)/1000.0)*(high - low) + low;
    return(val);
}

//*****************************************************************************
//
//            keep the best member of the population
//            (the last entry in the array Population has a copy
//            of the best individual)
//
//*****************************************************************************
void keep_the_best(int CHROM_LENGTH)
{
    int mem;
    int i;
    int current_best;

    current_best=1; // initialize index of the best individual

    for (mem=0; mem<POPSIZE; mem++){
        if (population[mem].fitness > population[POPSIZE].fitness){
            current_best=mem;
            population[POPSIZE].fitness=population[mem].fitness;
        }
    }
    // once the best member in the population is found, copy the genes
    for (i=1; i<=CHROM_LENGTH; i++)
        population[POPSIZE].gene[i]=population[current_best].gene[i];
}

//*****************************************************************************
//
//      Elitist function: The best member of the previous generation is stored
//      as the last in the array. If the best member of the current generation
```

```
//      is worse then the best member of the previous generation, the latter one
//      would replace the worst member of the current population
//
//*************************************************************************************
void elitist(int CHROM_LENGTH)
{
    int i;
    double best, worst;             // best and worst fitness values
    int best_member, worst_member;  // indexes of the best and worst member

    best=population[0].fitness;
    worst=population[0].fitness;
    for (i=0; i<POPSIZE - 1; ++i){
          if(population[i].fitness > population[i+1].fitness){
                if (population[i].fitness >= best){
                      best=population[i].fitness;
                      best_member=i;
                }
                if (population[i+1].fitness <= worst){
                      worst=population[i+1].fitness;
                      worst_member=i + 1;
                }
          }
          else{
                if (population[i].fitness <= worst){
                      worst=population[i].fitness;
                      worst_member=i;
                }
                if (population[i+1].fitness >= best){
                      best=population[i+1].fitness;
                      best_member=i + 1;
                }
          }
    }
    // if best individual from the new population is better than
    // the best individual from the previous population, then
    // copy the best from the new population; else replace the
    // worst individual from the current population with the
    // best one from the previous generation

    if (best >= population[POPSIZE].fitness){
        for (i=1; i<=CHROM_LENGTH; i++){
           population[POPSIZE].gene[i]=population[best_member].gene[i];
        }
        population[POPSIZE].fitness=population[best_member].fitness;
    }
    else{
        for (i=1; i<=CHROM_LENGTH; i++){
           population[worst_member].gene[i]=population[POPSIZE].gene[i];
        }
        population[worst_member].fitness=population[POPSIZE].fitness;
    }
}

//*************************************************************************************
//
//             Selection function: Standard proportional selection for
//             maximization problems incorporating elitist model - makes
//             sure that the best member survives
//
//*************************************************************************************
void select(void)
{
    int mem, i, j;
    double sum=0;
    double p;

    // find total fitness of the population
    for (mem=0; mem<POPSIZE; mem++){
          sum += population[mem].fitness;
    }

    // calculate relative fitness
    for (mem=0; mem<POPSIZE; mem++){
          population[mem].rel_fitness= population[mem].fitness/sum;
    }
    population[0].cum_fitness=population[0].rel_fitness;

    // calculate cumulative fitness
    for (mem=1; mem<POPSIZE; mem++){
          population[mem].cum_fitness= population[mem-1].cum_fitness +
```

```
                                population[mem].rel_fitness;
    }

    // finally select survivors using cumulative fitness.
    for (i=0; i<POPSIZE; i++){
          p=rand()%1000/1000.0;
          if (p<population[0].cum_fitness){
                newpopulation[i]=population[0];
          }
          else{
                for (j=0; j<POPSIZE;j++){
                      if (p >= population[j].cum_fitness && p<population[j+1].cum_fitness){
                            newpopulation[i]=population[j+1];
                      }
                }
          }
    }

    // once a new population is created, copy it back
    for (i=0; i<POPSIZE; i++) population[i]=newpopulation[i];
}

//*****************************************************************************
//
//            Crossover operator:
//            carries out a single point crossover between two parents
//
//*****************************************************************************
void crossover(int CHROM_LENGTH)
{
    int mem, one;
    int first = 0; // count of the number of members chosen
    double x;

    for (mem=0; mem<POPSIZE; ++mem){
          x=rand()%1000/1000.0;
          if (x<PXOVER){
                ++first;
                if (first % 2 == 0){
                      Xover(one, mem, CHROM_LENGTH);
                }
                else{
                      one=mem;
                }
          }
    }
}

//*****************************************************************************
//
//                    performs crossover of two selected parents
//
//*****************************************************************************
void Xover(int one, int two, int CHROM_LENGTH)
{
    int i;
    int point; // crossover point

    point=(rand() % (CHROM_LENGTH - 1)) + 1;
    for (i=1; i<=point; i++){
         swap(&population[one].gene[i], &population[two].gene[i]);
    }
}

//*****************************************************************************
//
//                                  swap 2 variables
//
//*****************************************************************************
void swap(double *x, double *y)
{
    double temp;

    temp=*x;
    *x=*y;
    *y=temp;
}

//*****************************************************************************
//
//                                  Mutation operator
```

Appendix E: STATS_X.dat Data Fields

```
//
//*****************************************************************************
void mutate(int CHROM_LENGTH)
{
    int i, j;
    double lower_bound, hbound;
    double x;

    for (i=0; i<POPSIZE; i++){
          for (j=1; j<=CHROM_LENGTH; j++){
               x=rand()%1000/1000.0;
               if (x<PMUTATION){
                   // find the bounds on the variable to be mutated
                   lower_bound=population[i].lower[j];
                   hbound=population[i].upper[j];
                   population[i].gene[j]=randval(lower_bound, hbound);
               }
          }
    }
}

//*****************************************************************************
//
//                          Progress report (output to file)
//
//*****************************************************************************
void progress_report(int generation, FILE* ga_stat, double *best_val, double *worst_val,
                     double *avg, double best_fitness_gen, double worst_fitness_gen)
{
    int i;
    double stddev;              // std. deviation of population fitness
    double sum_square;          // sum of square for std. calc
    double square_sum;          // square of sum for std. calc
    double sum;                 // total population fitness

    sum=0.0;
    sum_square=0.0;

    for (i=0; i<POPSIZE; i++){
          sum += population[i].fitness;
          sum_square += population[i].fitness * population[i].fitness;
          if (population[i].fitness < *worst_val) *worst_val=population[i].fitness;
    }

    *avg=sum/(double)POPSIZE;
    square_sum=(*avg) * (*avg) * POPSIZE;
    stddev=sqrt((sum_square - square_sum)/(POPSIZE - 1));
    *best_val=population[POPSIZE].fitness;

    fprintf(ga_stat, " %5d       %6.3f    %6.3f   %6.3f   %6.3f   %6.3f       %6.3f          %6.3f        %6.3f\n",

            generation, *best_val, *worst_val, best_fitness_gen, worst_fitness_gen,
            *avg, *avg+stddev, *avg-stddev, stddev);
}

//*****************************************************************************
//
//                          Allocate Memory for Structures
//
//*****************************************************************************
struct battle_parameters *get_bat(void)
{
    struct battle_parameters *p ;

    if ( (p = _fmalloc( sizeof(struct battle_parameters) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
    return p;
}

struct red_parameters *get_red(void)
{
    struct red_parameters *p;

    if ( (p = _fmalloc( sizeof(struct red_parameters) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
```

```
    return p;
}

struct blue_parameters *get_blue(void)
{
    struct blue_parameters *p;

    if ( (p = _fmalloc( sizeof(struct blue_parameters) )) == NULL) {
       _clearscreen( _GCLEARSCREEN );
       printf( "Insufficient Memory to Run");
       exit(0);
    }
    return p;
}

/*****************************************************************************************/
/*                                                                                       */
/*                Write chromosome of best ISAACA personality to file                    */
/*                                                                                       */
/*****************************************************************************************/
void WRITE_CHROM_TO_FILE(int g, FILE *ga_stat, int initial_condition_genes_flag)
{
    int ii;

    if (initial_condition_genes_flag==1){
        fprintf (ga_stat,"\n initial box size = %i", (int)(population[g].gene[43]));
        fprintf (ga_stat,"\n           x-coor = %i", (int)(population[g].gene[44]));
        fprintf (ga_stat,"\n           y-coor = %i", (int)(population[g].gene[45]));
        fprintf (ga_stat,"\n");
    }
    fprintf (ga_stat,"\n S-range = %i",(int)(population[g].gene[1]));
    ii = (int)(population[g].gene[2]);
    if (ii > (int)(population[g].gene[1]))
        ii=(int)(population[g].gene[1]);
    fprintf (ga_stat,"\n F-range = %i",ii);
    fprintf (ga_stat,"\n C-range = %i",(int)(population[g].gene[3]));
    fprintf (ga_stat,"\n w1_a    = %3.3f",SIGN(population[g].gene[5])*population[g].gene[4]);
    fprintf (ga_stat,"\n w2_a    = %3.3f",SIGN(population[g].gene[7])*population[g].gene[6]);
    fprintf (ga_stat,"\n w3_a    = %3.3f",SIGN(population[g].gene[9])*population[g].gene[8]);
    fprintf (ga_stat,"\n w4_a    = %3.3f",SIGN(population[g].gene[11])*population[g].gene[10]);
    fprintf (ga_stat,"\n w5_a    = %3.3f",SIGN(population[g].gene[13])*population[g].gene[12]);
    fprintf (ga_stat,"\n w6_a    = %3.3f",SIGN(population[g].gene[15])*population[g].gene[14]);
    fprintf (ga_stat,"\n w1_i    = %3.3f",SIGN(population[g].gene[17])*population[g].gene[16]);
    fprintf (ga_stat,"\n w2_i    = %3.3f",SIGN(population[g].gene[19])*population[g].gene[18]);
    fprintf (ga_stat,"\n w3_i    = %3.3f",SIGN(population[g].gene[21])*population[g].gene[20]);
    fprintf (ga_stat,"\n w4_i    = %3.3f",SIGN(population[g].gene[23])*population[g].gene[22]);
    fprintf (ga_stat,"\n w5_i    = %3.3f",SIGN(population[g].gene[25])*population[g].gene[24]);
    fprintf (ga_stat,"\n w6_i    = %3.3f",SIGN(population[g].gene[27])*population[g].gene[26]);
    fprintf (ga_stat,"\n ADV_a   = %i",(int)(population[g].gene[28]));
    fprintf (ga_stat,"\n CLS_a   = %i",(int)(population[g].gene[29]));
    fprintf (ga_stat,"\n CBT_a   = %i",SIGN(population[g].gene[31])*(int)(population[g].gene[30]));
    fprintf (ga_stat,"\n ADV_i   = %i",(int)(population[g].gene[32]));
    fprintf (ga_stat,"\n CLS_i   = %i",(int)(population[g].gene[33]));
    fprintf (ga_stat,"\n CBT_i   = %i",SIGN(population[g].gene[35])*(int)(population[g].gene[34]));
    if (population[g].gene[36] < .5){ // do not use min_dist genes 37-42
        fprintf (ga_stat,"\n R_R_a   = 0.00");
        fprintf (ga_stat,"\n R_B_a   = 0.00");
        fprintf (ga_stat,"\n R_RG_a  = 0.00");
        fprintf (ga_stat,"\n R_R_i   = 0.00");
        fprintf (ga_stat,"\n R_B_i   = 0.00");
        fprintf (ga_stat,"\n R_RG_i  = 0.00");
    }
    else{
        fprintf (ga_stat,"\n R_R_a   = %f",population[g].gene[37]);
        fprintf (ga_stat,"\n R_B_a   = %f",population[g].gene[38]);
        fprintf (ga_stat,"\n R_RG_a  = %f",population[g].gene[39]);
        fprintf (ga_stat,"\n R_R_i   = %f",population[g].gene[40]);
        fprintf (ga_stat,"\n R_B_i   = %f",population[g].gene[41]);
        fprintf (ga_stat,"\n R_RG_i  = %f",population[g].gene[42]);
    }
    fprintf(ga_stat,"\n");
}

/*****************************************************************************************/
/*                                                                                       */
/*                          Write mission fitness measure to file                        */
/*                                                                                       */
/*****************************************************************************************/
void WRITE_FITNESS(FILE *ga_stat, struct mission_objective *m, int termination_code,
                   int max_time_to_goal)
{
```

```
    // print out fitness parameters
    fprintf( ga_stat,"\n");
    fprintf( ga_stat, "Fitness Parameters:\n");
    fprintf( ga_stat,"\n");
    fprintf( ga_stat, "time to goal:        %2.2lf\n",m->alpha_1);
    fprintf( ga_stat, "red loss:            %2.2lf\n",m->alpha_2);
    fprintf( ga_stat, "blue loss:           %2.2lf\n",m->alpha_3);
    fprintf( ga_stat, "red CM to blue flag: %2.2lf\n",m->alpha_4);
    fprintf( ga_stat, "blue CM to red flag: %2.2lf\n",m->alpha_5);
    fprintf( ga_stat, "red near blue flag:  %2.2lf\n",m->alpha_6);
    fprintf( ga_stat, "blue near red flag:  %2.2lf\n",m->alpha_7);
    fprintf( ga_stat, "red fratricide:      %2.2lf\n",m->alpha_8);
    fprintf( ga_stat, "blue fratricide:     %2.2lf\n",m->alpha_9);
    fprintf( ga_stat,"\n");
    switch (termination_code) {
        case 1:
            fprintf( ga_stat, "termination condition:  First RED at goal\n");
            break;
        case 2:
            fprintf( ga_stat, "termination condition:  N(RED)>%i w/R=%i\n",
                (int)(m->near_range_num), (int)(m->near_range));
            break;
        case 3:
            fprintf( ga_stat, "termination condition:  RED_CM < %2.2lfR_max\n", m->r_CM_f);
            break;
        case 4:
            fprintf( ga_stat, "termination condition:  t_max=%3i\n", max_time_to_goal);
            break;
    }
    fprintf( ga_stat,"\n");
}

/*****************************************************************************************/
/*                                                                                       */
/*               Write chromosome of best ISAACA personality to file                     */
/*                                                                                       */
/*****************************************************************************************/
void DECODE_BEST(struct red_parameters *redp, int min_dist_genes_flag,
                 int initial_condition_genes_flag)
{
              // now re-define ISAACA force with genome-prescribed personality
              redp->irsrange = (int)(population[POPSIZE].gene[1]);
                if (redp->irsrange == 0)redp->irsrange=1;
              redp->iredfrange = (int)(population[POPSIZE].gene[2]);
                if (redp->iredfrange == 0)redp->iredfrange=1;
                // make sure F-range <= S-range
                if (redp->iredfrange > redp->irsrange)redp->iredfrange = redp->irsrange;
              redp->iradvrange[1] = (int)(population[POPSIZE].gene[3]);
                if (redp->iradvrange[1] == 0)redp->iradvrange[1]=1;
              redp->w1rdeff_a = (float)(population[POPSIZE].gene[4]);
                if (population[POPSIZE].gene[5] < .5)redp->w1rdeff_a = -redp->w1rdeff_a;
              redp->w2rdeff_a = (float)(population[POPSIZE].gene[6]);
                if (population[POPSIZE].gene[7] < .5)redp->w2rdeff_a = -redp->w2rdeff_a;
              redp->w3rdeff_a = (float)(population[POPSIZE].gene[8]);
                if (population[POPSIZE].gene[9] < .5)redp->w3rdeff_a = -redp->w3rdeff_a;
              redp->w4rdeff_a = (float)(population[POPSIZE].gene[10]);
                if (population[POPSIZE].gene[11] < .5)redp->w4rdeff_a = -redp->w4rdeff_a;
              redp->w5rdeff_a = (float)(population[POPSIZE].gene[12]);
                if (population[POPSIZE].gene[13] < .5)redp->w5rdeff_a = -redp->w5rdeff_a;
              redp->w6rdeff_a = (float)(population[POPSIZE].gene[14]);
                if (population[POPSIZE].gene[15] < .5)redp->w6rdeff_a = -redp->w6rdeff_a;
              redp->w1rdeff_i = (float)(population[POPSIZE].gene[16]);
                if (population[POPSIZE].gene[17] < .5)redp->w1rdeff_i = -redp->w1rdeff_i;
              redp->w2rdeff_i = (float)(population[POPSIZE].gene[18]);
                if (population[POPSIZE].gene[19] < .5)redp->w2rdeff_i = -redp->w2rdeff_i;
              redp->w3rdeff_i = (float)(population[POPSIZE].gene[20]);
                if (population[POPSIZE].gene[21] < .5)redp->w3rdeff_i = -redp->w3rdeff_i;
              redp->w4rdeff_i = (float)(population[POPSIZE].gene[22]);
                if (population[POPSIZE].gene[23] < .5)redp->w4rdeff_i = -redp->w4rdeff_i;
              redp->w5rdeff_i = (float)(population[POPSIZE].gene[24]);
                if (population[POPSIZE].gene[25] < .5)redp->w5rdeff_i = -redp->w5rdeff_i;
              redp->w6rdeff_i = (float)(population[POPSIZE].gene[26]);
                if (population[POPSIZE].gene[27] < .5)redp->w6rdeff_i = -redp->w6rdeff_i;
              redp->iradv_a[1] = (int)(population[POPSIZE].gene[28]);
              redp->irclus_a[1] = (int)(population[POPSIZE].gene[29]);
              redp->ircom_a[1] = (int)(population[POPSIZE].gene[30]);
                if (population[POPSIZE].gene[31] < .5)redp->ircom_a[1] = -redp->ircom_a[1];
              redp->iradv_i[1] = (int)(population[POPSIZE].gene[32]);
              redp->irclus_i[1] = (int)(population[POPSIZE].gene[33]);
              redp->ircom_i[1] = (int)(population[POPSIZE].gene[34]);
                if (population[POPSIZE].gene[35] < .5)redp->ircom_i[1] = -redp->ircom_i[1];
```

```
            if (min_dist_genes_flag==1){
              if (population[POPSIZE].gene[36] > .5){ // the  use min dist genes (37-42)
                  redp->zrfromrmindist_a= (float)(population[POPSIZE].gene[37]);
                  redp->zbfromrmindist_a= (float)(population[POPSIZE].gene[38]);
                  redp->zrfromrgmindist_a= (float)(population[POPSIZE].gene[39]);
                  redp->zrfromrmindist_i= (float)(population[POPSIZE].gene[40]);
                  redp->zbfromrmindist_i= (float)(population[POPSIZE].gene[41]);
                  redp->zrfromrgmindist_i= (float)(population[POPSIZE].gene[42]);
              }
              else{
                  redp->zrfromrmindist_a= 0;
                  redp->zbfromrmindist_a= 0;
                  redp->zrfromrgmindist_a= 0;
                  redp->zrfromrmindist_i= 0;
                  redp->zbfromrmindist_i= 0;
                  redp->zrfromrgmindist_i= 0;
              }
            }
            else{
              population[POPSIZE].gene[36] = 0;
              population[POPSIZE].gene[37] = 0;
              population[POPSIZE].gene[38] = 0;
              population[POPSIZE].gene[39] = 0;
              population[POPSIZE].gene[40] = 0;
              population[POPSIZE].gene[41] = 0;
              population[POPSIZE].gene[42] = 0;
              redp->zrfromrmindist_a= 0;
              redp->zbfromrmindist_a= 0;
              redp->zrfromrgmindist_a= 0;
              redp->zrfromrmindist_i= 0;
              redp->zbfromrmindist_i= 0;
              redp->zrfromrgmindist_i= 0;
            }

            if (initial_condition_genes_flag==1){
                battle.ibattlebox_red_length = (int)(population[POPSIZE].gene[43]);
                battle.ibattlebox_red_width = (int)(population[POPSIZE].gene[43]);
                battle.ibattlebox_red_cen_x = (int)(population[POPSIZE].gene[44]);
                battle.ibattlebox_red_cen_y = (int)(population[POPSIZE].gene[45]);
            }
            else{
                population[POPSIZE].gene[43] = 0;
                population[POPSIZE].gene[44] = 0;
                population[POPSIZE].gene[45] = 0;
            }

}

/*****************************************************************************/
/*                                                                           */
/*                          Show chromosome                                  */
/*                                                                           */
/*****************************************************************************/
void SHOW_CHROMOSOME(int mem, int initial_condition_genes_flag)
{
    int ii1, ii2;

    _moveto( 525, 175 );
    _setcolor( 2 );
    _outgtext ("    *** RED ISAACA PERSONALITY ***");
    _moveto( 525, 190 );
    _setcolor( 2 );
    _outgtext ("             CURRENT    BEST");
    _settextposition( 14, 73 );
    printf("S-range = %3i   (%3i)",(int)(population[mem].gene[1]), (int)(population[POPSIZE].gene[1]));
    ii1 = (int)(population[mem].gene[2]);
    if (ii1 > (int)(population[mem].gene[1]))
        ii1=(int)(population[mem].gene[1]);
    ii2 = (int)(population[POPSIZE].gene[2]);
    if (ii2 > (int)(population[POPSIZE].gene[1]))
        ii2=(int)(population[POPSIZE].gene[1]);
    _settextposition( 15, 73 );
    printf("F-range = %3i   (%3i)",ii1, ii2);
    _settextposition( 16, 73 );
    printf("C-range = %3i   (%3i)",(int)(population[mem].gene[3]),(int)(population[POPSIZE].gene[3]));
    _settextposition( 17, 73);
    printf("w1_a = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[5])*population[mem].gene[4],
                                      SIGN(population[POPSIZE].gene[5])*population[POPSIZE].gene[4]);
    _settextposition( 18, 73 );
    printf("w2_a = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[7])*population[mem].gene[6],
                                      SIGN(population[POPSIZE].gene[7])*population[POPSIZE].gene[6]);
```

```
        _settextposition( 19, 73 );
        printf("w3_a = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[9])*population[mem].gene[8],
                                          SIGN(population[POPSIZE].gene[9])*population[POPSIZE].gene[8]);
        _settextposition( 20, 73 );
        printf("w4_a = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[11])*population[mem].gene[10],
                                          SIGN(population[POPSIZE].gene[11])*population[POPSIZE].gene[10]);
        _settextposition( 21, 73 );
        printf("w5_a = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[13])*population[mem].gene[12],
                                          SIGN(population[POPSIZE].gene[13])*population[POPSIZE].gene[12]);
        _settextposition( 22, 73 );
        printf("w6_a = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[15])*population[mem].gene[14],
                                          SIGN(population[POPSIZE].gene[15])*population[POPSIZE].gene[14]);
        _settextposition( 23, 73 );
        printf("w1_i = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[17])*population[mem].gene[16],
                                          SIGN(population[POPSIZE].gene[17])*population[POPSIZE].gene[16]);
        _settextposition( 24, 73 );
        printf("w2_i = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[19])*population[mem].gene[18],
                                          SIGN(population[POPSIZE].gene[19])*population[POPSIZE].gene[18]);
        _settextposition( 25, 73 );
        printf("w3_i = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[21])*population[mem].gene[20],
                                          SIGN(population[POPSIZE].gene[21])*population[POPSIZE].gene[20]);
        _settextposition( 26, 73 );
        printf("w4_i = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[23])*population[mem].gene[22],
                                          SIGN(population[POPSIZE].gene[23])*population[POPSIZE].gene[22]);
        _settextposition( 27, 73 );
        printf("w5_i = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[25])*population[mem].gene[24],
                                          SIGN(population[POPSIZE].gene[25])*population[POPSIZE].gene[24]);
        _settextposition( 28, 73 );
        printf("w6_i = %+6.2f   (%+6.2f)",SIGN(population[mem].gene[27])*population[mem].gene[26],
                                          SIGN(population[POPSIZE].gene[27])*population[POPSIZE].gene[26]);
        _settextposition( 29, 73 );
        printf("ADV_a   = %3i   (%3i)",(int)(population[mem].gene[28]),(int)(population[POPSIZE].gene[28]));
        _settextposition( 30, 73 );
        printf("CLS_a   = %3i   (%3i)",(int)(population[mem].gene[29]),(int)(population[POPSIZE].gene[29]));
        _settextposition( 31, 73 );
        printf("CBT_a   = %3i   (%3i)",SIGN(population[mem].gene[31])*(int)(population[mem].gene[30]),
                                       SIGN(population[POPSIZE].gene[31])*(int)(population[POPSIZE].gene[30]));
        _settextposition( 32, 73 );
        printf("ADV_i   = %3i   (%3i)",(int)(population[mem].gene[32]),(int)(population[POPSIZE].gene[32]));
        _settextposition( 33, 73 );
        printf("CLS_i   = %3i   (%3i)",(int)(population[mem].gene[33]),(int)(population[POPSIZE].gene[33]));
        _settextposition( 34, 73 );
        printf("CBT_i   = %3i   (%3i)",SIGN(population[mem].gene[35])*(int)(population[mem].gene[34]),
                                       SIGN(population[POPSIZE].gene[35])*(int)(population[POPSIZE].gene[34]));
}

int SIGN(double x)
{
    if (x<=.5){
        return -1;
    }
    else{
        return 1;
    }
}
```

Appendix E: STATS_X.dat Data Fields

This appendix provides a complete listing of the contents of each data field appearing in each of the 21 statistics output files that can be generated by ISAAC (see table 8 in ***Data Collection***). This information can be used to generate desired plots using a stand-alone plotting program.

STATS_1.dat

STATS_1.dat contains a summary of basic force strength measures and consists of the following seven fields:

- ***field 1:*** iteration step (i.e., time)
- ***field 2:*** fraction of remaining alive red ISAACAs
- ***field 3:*** fraction of remaining injured red ISAACAs
- ***field 4:*** fraction of remaining (alive + injured) red ISAACAs
- ***field 5:*** fraction of remaining alive blue ISAACAs
- ***field 6:*** fraction of remaining injured blue ISAACAs
- ***field 7:*** fraction of remaining (alive + injured) blue ISAACAs

STATS_2.dat

STATS_2.dat contains distributions of the number of ISAACA pairs less than or equal to a certain distance apart and consists of the following five fields:

- ***field 1:*** distance (D)
- ***field 2:*** iteration step (i.e., time)
- ***field 3:*** # of red-red pairs less than or equal to a distance (D) apart
- ***field 4:*** # of blue-blue pairs less than or equal to a distance (D) apart
- ***field 5:*** # of red-blue pairs less than or equal to a distance (D) apart

STATS_3.dat

STATS_3.dat contains distributions of the number of red and blue ISAACA less than or equal to a certain distance from their enemy flag. It consists of the following four fields:

- *field 1:* distance (D)
- *field 2:* iteration step (i.e., time)
- *field 3:* # of red ISAACAs that are ≤ distance (D) from the blue flag
- *field 4:* # of blue ISAACAs that are ≤ distance (D) from the red flag

STATS_4.dat

STATS_4.dat summarizes the actual interpoint distributions appearing in *field 3* of **STATS_2.dat** above by providing their averages and standard deviations. It consists of the following seven fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average of red-red interpoint distances
- *field 3:* red-red average + red-red absolute deviation
- *field 4:* red-red average - red-red absolute deviation
- *field 5:* absolute deviation of red-red interpoint distances
- *field 6:* standard deviation of red-red interpoint distances
- *field 7:* variance of red-red interpoint distances

STATS_5.dat

STATS_5.dat summarizes the actual interpoint distributions appearing in *field 4* of **STATS_2.dat** above by providing their averages and standard deviations. It consists of the following seven fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average of blue-blue interpoint distances
- *field 3:* blue-blue average + blue-blue absolute deviation
- *field 4:* blue-blue average - blue-blue absolute deviation

- *field 5:* absolute deviation of blue-blue interpoint distances
- *field 6:* standard deviation of blue-blue interpoint distances
- *field 7:* variance of blue-blue interpoint distances

STATS_6.dat

STATS_6.dat summarizes the actual interpoint distributions appearing in *field 5* of **STATS_2.dat** above by providing their averages and standard deviations. It consists of the following seven fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average of red-blue interpoint distances
- *field 3:* red-blue average + red-blue absolute deviation
- *field 4:* red-blue average - red-blue absolute deviation
- *field 5:* absolute deviation of red-blue interpoint distances
- *field 6:* standard deviation of red-blue interpoint distances
- *field 7:* variance of red-blue interpoint distances

STATS_7.dat

STATS_7.dat summarizes the actual interpoint distributions appearing in *field 3* of **STATS_3.dat** above by providing their averages and standard deviations. It consists of the following seven fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average of red:blue-flag interpoint distances
- *field 3:* red:blue-flag average + red:blue-flag absolute deviation
- *field 4:* red:blue-flag average - red:blue-flag absolute deviation
- *field 5:* absolute deviation of red:blue-flag interpoint distances
- *field 6:* standard deviation of red:blue-flag interpoint distances
- *field 7:* variance of red:blue-flag interpoint distances

STATS_8.dat

STATS_8.dat summarizes the actual interpoint distributions appearing in *field 4* of **STATS_3.dat** above by providing their averages and standard deviations. It consists of the following seven fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average of blue:red-flag interpoint distances
- *field 3:* blue:red-flag average + blue:red-flag absolute deviation
- *field 4:* blue:red-flag average - blue:red-flag absolute deviation
- *field 5:* absolute deviation of blue:red-flag interpoint distances
- *field 6:* standard deviation of blue:red-flag interpoint distances
- *field 7:* variance of blue:red-flag interpoint distances

STATS_9.dat

STATS_9.dat contains estimates of the *spatial entropy* (see ***Spatial Entropy*** above) of the distribution of red and blue ISAACAs. It consists of the following ten fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* red entropy (using a 4x4 array of 20x20 blocks)
- *field 3:* blue entropy (using a 4x4 array of 20x20 blocks)
- *field 4:* red-blue entropy (using a 4x4 array of 20x20 blocks)
- *field 5:* red entropy (using a 8x8 array of 10x10 blocks)
- *field 6:* blue entropy (using a 8x8 array of 10x10 blocks)
- *field 7:* red-blue entropy (using a 8x8 array of 10x10 blocks)
- *field 8:* red entropy (using a 16x16 array of 5x5 blocks)
- *field 9:* blue entropy (using a 16x16 array of 5x5 blocks)
- *field 10:* red blue entropy (using a 16x16 array of 5x5 blocks)

STATS_10.dat

STATS_10.dat contains the distribution of clusters of red and blue ISAACAs using an inter-cluster distance criterion of D=1 (see ***Appendix F: Cluster Counting Algorithm***). It consists of the following 14 fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* total number of clusters
- *field 3:* number of clusters of size N=1
- *field 4:* number of clusters of size N=2 through N=5
- *field 5:* number of clusters of size N=6 through N=10
- *field 6:* number of clusters of size N=11 through N=15
- *field 7:* number of clusters of size N=16 through N=20
- *field 8:* number of clusters of size N=21 through N=-25
- *field 9:* number of clusters of size N=26 through N=-30
- *field 10:* number of clusters of size N=31 through N=35
- *field 11:* number of clusters of size N=36 through N=40
- *field 12:* number of clusters of size N=41 through N=45
- *field 13:* number of clusters of size N=46 through N=50
- *field 14:* number of clusters of size N=51 through N=MAX

STATS_11.dat

STATS_11.dat contains the averages and deviations of sizes of clusters of red and blue ISAACAs using an inter-cluster distance criterion of D=1 (see ***Appendix F: Cluster Counting Algorithm***). It consists of the following seven fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average cluster size
- *field 3:* average cluster size + absolute deviation
- *field 4:* average cluster size - absolute deviation
- *field 5:* absolute deviation of average cluster size
- *field 6:* standard deviation of average cluster size
- *field 7:* variance of cluster size

STATS_12.dat

STATS_12.dat contains the distribution of clusters of red and blue ISAACAs using an inter-cluster distance criterion of D=2 (see ***Appendix F: Cluster Counting Algorithm***). It consists of the following 14 fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* total number of clusters
- *field 3:* number of clusters of size N=1
- *field 4:* number of clusters of size N=2 through N=5
- *field 5:* number of clusters of size N=6 through N=10
- *field 6:* number of clusters of size N=11 through N=15
- *field 7:* number of clusters of size N=16 through N=20
- *field 8:* number of clusters of size N=21 through N=-25
- *field 9:* number of clusters of size N=26 through N=-30
- *field 10:* number of clusters of size N=31 through N=35
- *field 11:* number of clusters of size N=36 through N=40
- *field 12:* number of clusters of size N=41 through N=45
- *field 13:* number of clusters of size N=46 through N=50
- *field 14:* number of clusters of size N=51 through N=MAX

STATS_13.dat

STATS_13.dat contains the averages and deviations of sizes of clusters of red and blue ISAACAs using an inter-cluster distance criterion of D=2 (see ***Appendix F: Cluster Counting Algorithm***). It consists of the following seven fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average cluster size
- *field 3:* average cluster size + absolute deviation
- *field 4:* average cluster size - absolute deviation
- *field 5:* absolute deviation of average cluster size
- *field 6:* standard deviation of average cluster size

- *field 7:* variance of cluster size

STATS_14.dat

STATS_14.dat contains the averages and deviations of the number of *red* ISAACAs within a range R=1,2,...5 of *red* ISAACAs. It consists of the following 21 fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average # of red within R=1 of red
- *field 3:* average # of red within R=1 of red + absolute deviation
- *field 4:* average # of red within R=1 of red - absolute deviation
- *field 5:* absolute deviation for range R=1
- *field 6:* average # of red within R=2 of red
- *field 7:* average # of red within R=2 of red + absolute deviation
- *field 8:* average # of red within R=2 of red - absolute deviation
- *field 9:* absolute deviation for range R=2
- *field 10:* average # of red within R=3 of red
- *field 11:* average # of red within R=3 of red + absolute deviation
- *field 12:* average # of red within R=3 of red - absolute deviation
- *field 13:* absolute deviation for range R=3
- *field 14:* average # of red within R=4 of red
- *field 15:* average # of red within R=4 of red + absolute deviation
- *field 16:* average # of red within R=4 of red - absolute deviation
- *field 17:* absolute deviation for range R=4
- *field 18:* average # of red within R=5 of red
- *field 19:* average # of red within R=5 of red + absolute deviation
- *field 20:* average # of red within R=5 of red - absolute deviation
- *field 21:* absolute deviation for range R=5

STATS_15.dat

STATS_15.dat contains the averages and deviations of the number of ***blue*** ISAACAs within a range R=1,2,...5 of ***blue*** ISAACAs. It consists of the following 21 fields:

- ***field 1:*** iteration step (i.e., time)
- ***field 2:*** average # of blue within R=1 of blue
- ***field 3:*** average # of blue within R=1 of blue + absolute deviation
- ***field 4:*** average # of blue within R=1 of blue - absolute deviation
- ***field 5:*** absolute deviation for range R=1
- ***field 6:*** average # of blue within R=2 of blue
- ***field 7:*** average # of blue within R=2 of blue + absolute deviation
- ***field 8:*** average # of blue within R=2 of blue - absolute deviation
- ***field 9:*** absolute deviation for range R=2
- ***field 10:*** average # of blue within R=3 of blue
- ***field 11:*** average # of blue within R=3 of blue + absolute deviation
- ***field 12:*** average # of blue within R=3 of blue - absolute deviation
- ***field 13:*** absolute deviation for range R=3
- ***field 14:*** average # of blue within R=4 of blue
- ***field 15:*** average # of blue within R=4 of blue + absolute deviation
- ***field 16:*** average # of blue within R=4 of blue - absolute deviation
- ***field 17:*** absolute deviation for range R=4
- ***field 18:*** average # of blue within R=5 of blue
- ***field 19:*** average # of blue within R=5 of blue + absolute deviation
- ***field 20:*** average # of blue within R=5 of blue - absolute deviation
- ***field 21:*** absolute deviation for range R=5

STATS_16.dat

STATS_16.dat contains the averages and deviations of the number of ***red*** ISAACAs within a range R=1,2,...5 of ***blue*** ISAACAs. It consists of the following 21 fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average # of red within R=1 of blue
- *field 3:* average # of red within R=1 of blue + absolute deviation
- *field 4:* average # of red within R=1 of blue - absolute deviation
- *field 5:* absolute deviation for range R=1
- *field 6:* average # of red within R=2 of blue
- *field 7:* average # of red within R=2 of blue + absolute deviation
- *field 8:* average # of red within R=2 of blue - absolute deviation
- *field 9:* absolute deviation for range R=2
- *field 10:* average # of red within R=3 of blue
- *field 11:* average # of red within R=3 of blue + absolute deviation
- *field 12:* average # of red within R=3 of blue - absolute deviation
- *field 13:* absolute deviation for range R=3
- *field 14:* average # of red within R=4 of blue
- *field 15:* average # of red within R=4 of blue + absolute deviation
- *field 16:* average # of red within R=4 of blue - absolute deviation
- *field 17:* absolute deviation for range R=4
- *field 18:* average # of red within R=5 of blue
- *field 19:* average # of red within R=5 of blue + absolute deviation
- *field 20:* average # of red within R=5 of blue - absolute deviation
- *field 21:* absolute deviation for range R=5

STATS_17.dat

STATS_17.dat contains the averages and deviations of the number of ***blue*** ISAACAs within a range R=1,2,...5 of ***red*** ISAACAs. It consists of the following 21 fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average # of blue within R=1 of red
- *field 3:* average # of blue within R=1 of red + absolute deviation
- *field 4:* average # of blue within R=1 of red - absolute deviation

- *field 5:* absolute deviation for range R=1
- *field 6:* average # of blue within R=2 of red
- *field 7:* average # of blue within R=2 of red + absolute deviation
- *field 8:* average # of blue within R=2 of red - absolute deviation
- *field 9:* absolute deviation for range R=2
- *field 10:* average # of blue within R=3 of red
- *field 11:* average # of blue within R=3 of red + absolute deviation
- *field 12:* average # of blue within R=3 of red - absolute deviation
- *field 13:* absolute deviation for range R=3
- *field 14:* average # of blue within R=4 of red
- *field 15:* average # of blue within R=4 of red + absolute deviation
- *field 16:* average # of blue within R=4 of red - absolute deviation
- *field 17:* absolute deviation for range R=4
- *field 18:* average # of blue within R=5 of red
- *field 19:* average # of blue within R=5 of red + absolute deviation
- *field 20:* average # of blue within R=5 of red - absolute deviation
- *field 21:* absolute deviation for range R=5

STATS_18.dat

STATS_18.dat contains the averages and deviations of the number of ***both red and blue*** ISAACAs within a range R=1,2,...5 of ***red*** ISAACAs. It consists of the following 21 fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average # of ISAACAs within R=1 of red
- *field 3:* average # of ISAACAs /w R=1 of red + absolute deviation
- *field 4:* average # of ISAACAs /w R=1 of red - absolute deviation
- *field 5:* absolute deviation for range R=1
- *field 6:* average # of ISAACAs /w R=2 of red
- *field 7:* average # of ISAACAs /w R=2 of red + absolute deviation
- *field 8:* average # of ISAACAs /w R=2 of red - absolute deviation

- *field 9:* absolute deviation for range R=2
- *field 10:* average # of ISAACAs /w R=3 of red
- *field 11:* average # of ISAACAs /w R=3 of red + absolute deviation
- *field 12:* average # of ISAACAs /w R=3 of red - absolute deviation
- *field 13:* absolute deviation for range R=3
- *field 14:* average # of ISAACAs /w R=4 of red
- *field 15:* average # of ISAACAs /w R=4 of red + absolute deviation
- *field 16:* average # of ISAACAs /w R=4 of red - absolute deviation
- *field 17:* absolute deviation for range R=4
- *field 18:* average # of ISAACAs /w R=5 of red
- *field 19:* average # of ISAACAs /w R=5 of red + absolute deviation
- *field 20:* average # of ISAACAs /w R=5 of red - absolute deviation
- *field 21:* absolute deviation for range R=5

STATS_19.dat

STATS_19.dat contains the averages and deviations of the number of ***both red and blue*** ISAACAs within a range R=1,2,...5 of ***blue*** ISAACAs. It consists of the following 21 fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* average # of ISAACAs within R=1 of blue
- *field 3:* average # of ISAACAs /w R=1 of blue + absolute deviation
- *field 4:* average # of ISAACAs /w R=1 of blue - absolute deviation
- *field 5:* absolute deviation for range R=1
- *field 6:* average # of ISAACAs /w R=2 of blue
- *field 7:* average # of ISAACAs /w R=2 of blue + absolute deviation
- *field 8:* average # of ISAACAs /w R=2 of blue - absolute deviation
- *field 9:* absolute deviation for range R=2

- *field 10:* average # of ISAACAs /w R=3 of blue
- *field 11:* ave # of ISAACAs /w R=3 of blue + absolute deviation
- *field 12:* ave # of ISAACAs /w R=3 of blue - absolute deviation
- *field 13:* absolute deviation for range R=3
- *field 14:* average # of ISAACAs /w R=4 of blue
- *field 15:* ave # of ISAACAs /w R=4 of blue + absolute deviation
- *field 16:* ave # of ISAACAs /w R=4 of blue - absolute deviation
- *field 17:* absolute deviation for range R=4
- *field 18:* average # of ISAACAs /w R=5 of blue
- *field 19:* ave # of ISAACAs /w R=5 of blue + absolute deviation
- *field 20:* ave # of ISAACAs /w R=5 of blue - absolute deviation
- *field 21:* absolute deviation for range R=5

STATS_20.dat

STATS_20.dat contains the center-of-mass (COM) coordinates of red and blue ISAACAs and the distances between the center-of-mass positions and red and blue flags. It consists of the following thirteen fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* red COM x-coordinate
- *field 3:* red COM y-coordinate
- *field 4:* distance between red COM x-coordinate and red flag
- *field 5:* distance between red COM x-coordinate and blue flag
- *field 6:* blue COM x-coordinate
- *field 7:* blue COM y-coordinate
- *field 8:* distance between blue COM x-coordinate and red flag
- *field 9:* distance between blue COM x-coordinate and blue flag
- *field 10:* total (red+blue) COM x-coordinate
- *field 11:* total (red+blue) COM y-coordinate
- *field 12:* distance between total COM x-coordinate and red flag

- *field 13:* distance between total COM x-coordinate and blue flag

STATS_21.dat

STATS_21.dat contains the number of red and blue ISAACAs within ranges R=1,2,...,5 of the red and blue flags, expressed as the fraction of the maximum possible number. It consists of the following eleven fields:

- *field 1:* iteration step (i.e., time)
- *field 2:* number of red within R=1 of blue flag
- *field 3:* number of blue within R=1 of red flag
- *field 4:* number of red within R=2 of blue flag
- *field 5:* number of blue within R=2 of red flag
- *field 6:* number of red within R=3 of blue flag
- *field 7:* number of blue within R=3 of red flag
- *field 8:* number of red within R=4 of blue flag
- *field 9:* number of blue within R=4 of red flag
- *field 10:* number of red within R=5 of blue flag
- *field 11:* number of blue within R=5 of red flag

Appendix F: Cluster Counting Algorithm

ISAAC's rudimentary data collection capability provides facilities to calculate seven basic classes of information; see ***Data Collection.*** The fifth class of data consists of keeping track of the averages and distributions of the sizes of *clusters* of ISAACAs. Because this class of data provides an insight into the gross structural appearance of the entire battlefield, it can be thought of as a crude pattern recognition measure. This appendix provides a brief heuristic description of the cluster counting algorithm (as implemented in **ISAAC_CE** and **ISAAC_SQ**) and includes the C source code listing of the function that implements this algorithm (i.e., **CLUSTER_1** in **ISAAC_T3.c**; see Table 12). The contents of the output files is described in

Heuristic Recipe

The cluster counting algorithm used by ISAAC is patterned after the *Hoshen-Kopelman* algorithm described in [50]. A heuristic description of this algorithm follows.

Given a distribution of ISAACAs spread throughout the battlefield, the "cluster counting problem" is to find an algorithm that assigns all ISAACAs within the same cluster the same label and gives different labels to ISAACAs belonging to different clusters. An ISAACA's *membership* within a cluster is defined by an inter-cluster distance criterion of either D=1 (meaning that two ISAACAs that are separated by *one* lattice cell belong to the same cluster) or D=2 (meaning that two ISAACAs that are separated by *one* or *two* lattice cells belong to the same cluster).

The approach is to scan the battlefield – cell by cell and row by row – assigning cluster-specific labels to occupied cells, and leaving empty cells alone. The first cluster that is encountered gets assigned the label L=1. Neighboring occupied cells, within the same row, get assigned the same label. As the scan continues to the right, occupied cells that are farther than D=1 (or D=2) from the right-most occupied cell of the first cluster are assigned the label L=2, and so on.

A potential problem occurs with this labeling scheme when ISAACAs belonging to the same cluster are inadvertently assigned different labels on different rows. Consider figure 93, which shows a fragment of the same cluster on two successive rows, and the result of applying the heuristic cluster-counting algorithm as thus far described (using D=1 as the inter-cluster distance criterion).

Figure 93 shows that the first cell (on the upper left), which is occupied, is immediately assigned the label L=1. The second cell is empty and is

left alone. The third cell contains an ISAACA that is farther from cell #1 than one unit, and is therefore assigned the label L=2. This labeling scheme continues through the second row until we reach the "question mark," at which point we have to make a decision. *What label should we assign to this cell?* According to the top row, the label is L=3. According to the second row, the label is L=2. The answer is that we choose L=2 to keep the total number of labels as small as possible. We must also relabel the cell with L=3 with L=2 to indicate that it is really part of the same cluster as defined by label L=2.

Figure 93. A fragment of a cluster of ISAACAs and the result of applying the cluster-counting heuristic



More generally, the *Hoshen-Kopelman* heuristic is to partition all labels into two groups: *good labels* and *bad labels.* The good labels refer to labels that correctly characterize distinct clusters. The bad labels refer to labels that appear at first to characterize a new cluster but are actually assigned to cells belonging to previously labeled clusters.

These two groups are kept track of by an additional array, the *label of labels* (called label_of_labels[i] in **CLUSTER_1** below). A good label, say l_g, is characterized by label_of_labels[l_g]=l_g. A bad label, say l_b, is characterized by label_of_labels[l_b]=x, where $x < l_b$. In this way, the battlefield is first scanned in the manner described above, and each cluster is assigned a initial label (or set of labels). After the scan is completed, the cells are relabeled according to the following logic:

- If the label of a cell is l_c, check to see if label_of_labels[l_c] = l_c: if yes, then the label is *good* (leave it alone); if no, then the label is bad, and label_of_labels[l_c] = l_c'.
- Check to see if label_of_labels[l_c'] = l_c': if yes, then the label is *good* (relabel the cell with initial label l_c with the label l_c'); if no, then the label is bad, and label_of_labels[l_c'] = l_c''.
- Continue in this way until the search ends with a good label, say l_g, and relabel the cell with that label.

Source Code

```
//*****************************************************************************
//
//                    returns the distribution of cluster sizes
//                   (using inter-cluster distance criteria of D=1)
//
//*****************************************************************************
void CLUSTER_1(int isize, struct red_parameters *redp, struct blue_parameters *bluep,
                struct battle_parameters *batp, struct statistics *s)

{
int i, j, ll, n;
int label_min, ix, iy;
int count[MAXCLUSTERSIZE+1];
int label_of_labels[MAXCLUSTERSIZE+1];
int **label;
float ep=0,ss;

//
// allocate memory for label
//
label = (int**) _fmalloc( (MAXISAACNUM+1) * sizeof(int*) );
if ( !label ) nomem();
for ( i = 0; i < (MAXISAACNUM+1); i++ ){
        label[i] = (int*) _fmalloc( (MAXISAACNUM+1) * sizeof(int) );
        if ( !label[i] ) nomem();
}

// initialize label matrix
for (i=1; i<=batp->isize; i++){
        for (j=1; j<=batp->isize; j++){
                // label < 0 if ISAACA is present, else 0
                label[i][j] = -batp->ioccupation[i][j];
        }
}

// initialize 'good'-label matrix
for (i=1; i<=MAXCLUSTERSIZE; i++){
        label_of_labels[i]=MAXCLUSTERSIZE;
}

// initialize cluster count variable to zero
n = 0;

// loop through battlefield
for (i=1; i<=batp->isize; i++){
        for (j=1; j<=batp->isize; j++){
                if (n==0){
                        if (label[i][j] < 0){ // then found first ISAACA without a label
                                n=1;
                                label[i][j]=1;
                                label_of_labels[1]=1;
                        }
                }
                else{ // see if i,j belongs to known clusters (at neighboring sites)
                        // if < 0 then ISAACA at i,j does not yet have a label
                        if (label[i][j] < 0){
                                // give it a temporary large label
                                label_min = MAXCLUSTERSIZE + 1;
```

```
// need to test these sites:
//
// (i-1, j-1) | (i, j-1) | xxxxxxx
// ------------------------
// (i-1, j) | I,J | xxxxxxx
// ------------------------
// (i-1, j+1) | xxxxxxx | xxxxxxx
//

// first try i-1, j-1
ix = i-1;
iy = j-1;
if (ix > 0 && ix <= batp->isize && iy > 0 && iy <=batp->isize){
  if (label[ix][iy] < label_min && label[ix][iy] > 0)
        label_min = label[ix][iy];
}
// try i, j-1
ix = i;
iy = j-1;
if (ix > 0 && ix <= batp->isize && iy > 0 && iy <=batp->isize){
  if (label[ix][iy] < label_min && label[ix][iy] > 0)
        label_min = label[ix][iy];
}
// try i-1, j
ix = i-1;
iy = j;
if (ix > 0 && ix <= batp->isize && iy > 0 && iy <=batp->isize){
  if (label[ix][iy] < label_min && label[ix][iy] > 0)
        label_min = label[ix][iy];
}
// try i-1, j+1
ix = i-1;
iy = j+1;
if (ix > 0 && ix <= batp->isize && iy > 0 && iy <=batp->isize){
  if (label[ix][iy] < label_min && label[ix][iy] > 0)
        label_min = label[ix][iy];
}

if (label_min < MAXCLUSTERSIZE + 1) {
        // give i,j the minimum valued label
        label[i][j] = label_min;

        // change all neighboring label_of_labels to label_min
        // first look at i-1, j-1
        ix = i-1;
        iy = j-1;
        if (ix > 0 && ix <= batp->isize &&
          iy > 0 && iy <=batp->isize){
                if ( label[ix][iy] > label_min &&
                  label_of_labels[label[ix][iy]] > label_min){
                        label_of_labels[label[ix][iy]] = label_min;
                }
        }
        // look at i, j-1
        ix = i;
        iy = j-1;
        if (ix > 0 && ix <= batp->isize &&
          iy > 0 && iy <=batp->isize){
                if ( label[ix][iy] > label_min &&
                  label_of_labels[label[ix][iy]] > label_min){
                        label_of_labels[label[ix][iy]] = label_min;
                }
        }
        // look at i-1, j
        ix = i-1;
```

Appendix F: Cluster Counting Algorithm

```
                                        iy = j;
                                        if (ix > 0 && ix <= batp->isize &&
                                          iy > 0 && iy <=batp->isize){
                                                if ( label[ix][iy] > label_min &&
                                                  label_of_labels[label[ix][iy]] > label_min){
                                                        label_of_labels[label[ix][iy]] = label_min;
                                                }
                                        }
                                        // look at i-1, j+1
                                        ix = i-1;
                                        iy = j+1;
                                        if (ix > 0 && ix <= batp->isize &&
                                          iy > 0 && iy <=batp->isize){
                                                if ( label[ix][iy] > label_min &&
                                                  label_of_labels[label[ix][iy]] > label_min){
                                                        label_of_labels[label[ix][iy]] = label_min;
                                                }
                                        }
                                }
                                else{
                                        // i,j could not be attached to old cluster; add a new cluster
                                        ++n;
                                        label[i][j]=n;
                                        label_of_labels[n]=n; // mark as a 'good' label
                                }
                        }
                }
        }
}

//
// find the array of real (i.e. 'good') labels
//
for (i=1; i<=batp->isize; i++){
        for (j=1; j<=batp->isize; j++){
                if ( label[i][j] > 0){
doagain:                        ll = label[i][j];
                                label[i][j] = label_of_labels[label[i][j]];
                                if (ll != label[i][j]) goto doagain;
                }
        }
}



//
// initialize count array
//
for (i=1; i<=MAXCLUSTERSIZE; ++i){
   count[i] = 0;
}

//
// get the size of each cluster (labeled by label[i][j] = 1,2,...n )
//
for (i=1; i<=batp->isize; i++){
        for (j=1; j<=batp->isize; j++){
                if ( label[i][j] > 0) ++count[ label[i][j] ];
        }
}

//
// initialize cluster distribution
//
for (i=1; i<=MAXCLUSTERSIZE; i++) s->clusters_1[i] = 0;
```

Appendix F: Cluster Counting Algorithm

```
//
// get cluster distribution
//
s->number_of_clusters_1 = 0;
for (i=1; i<=MAXCLUSTERSIZE; i++){
        ++s->clusters_1[count[i]];
        if (count[i]>0) ++s->number_of_clusters_1;
}

//*************************************************************
//
//          calculate averages and deviations
//
//*************************************************************
// initialize average cluster
s->cluster_1_ave = 0;

// calculate average cluster size
        for (i=1; i<=n; i++){
        s->cluster_1_ave = s->cluster_1_ave + (float)(count[i]);
}

if (s->number_of_clusters_1 < 2){
        s->cluster_1_adev = 0;
        s->cluster_1_var = 0;
        s->cluster_1_sdev = 0;
}
else{
        s->cluster_1_ave = s->cluster_1_ave / (float)(s->number_of_clusters_1);
        // calculate deviation
        s->cluster_1_adev = 0;
        s->cluster_1_var = 0;
        for (i=1; i<=n; i++){
           if (count[i]>0) {
                ss = (float)(count[i]) - s->cluster_1_ave;
                s->cluster_1_adev = s->cluster_1_adev + abs_float(ss);
                ep = ep + ss;
                s->cluster_1_var = s->cluster_1_var + (ss*ss);
           }
        }
        s->cluster_1_adev =  s->cluster_1_adev / (float)(s->number_of_clusters_1);
        s->cluster_1_var = (s->cluster_1_var - ep*ep/(float)(s->number_of_clusters_1))/
                                (float)((s->number_of_clusters_1-1));
        if( s->cluster_1_var < 0) s->cluster_1_var = 0 ;
        s->cluster_1_sdev=(float)(sqrt((double)(s->cluster_1_var)));
}

//*************************************************************
//
//                  free memory
//
//*************************************************************
for ( i = 0; i < (MAXISAACNUM+1); i++ ){
        _ffree(label[i]) ;
}
  _ffree(label);
}
```

Appendix G: Sample Input Data Files

This appendix contains sample data input files for **ISAAC_CE** (i.e., ISAAC's multi-squad core-engine), **ISAAC_GA** (i.e., the stand-alone genetic algorithm "evolver") and **ISAAC_PM** (i.e., the parameter-space "mapper").

Sample Data Input File for ISAAC_CE: ISAAC.dat

ISAAC.dat is the input data file for **ISAAC_CE**. The section ***A Concise User's Guide to ISAAC*** described the contents of this file. Note that this file *does not* include command and control related parameter values (described in pertinent sections of ***Contents of ISAAC's Input Data File***; see figures 28 and 29).

```
*****************************************
* GENERAL BATTLE PARAMETERS
*****************************************
battle_size          80
*
* initial distribution
*
init_dist            1
R_box_(l,w)          20,20 10,10 20,20 20,20 20,20 20,20 20,20 20,20 20,20 20,20
RED_cen_(x,y)        10,10 10,10 10,10 10,10 10,10 10,10 10,10 10,10 10,10 10,10
B_box_(l,w)          35,35 35,35 35,35 35,35 35,35 35,35 35,35 35,35 35,35 35,35
BLUE_cen_(x,y)       70,70 70,70 70,70 70,70 70,70 70,70 70,70 70,70 70,70 70,70
B_flag_(x,y)         79,79
R_flag_(x,y)         1,1
termination?         2
move_order?          2
combat_flag?         2
terrain_flag?        0
*
* fratricide parameters
*
red_frat_flag?       0
blue_frat_flag?      0
red_frat_rad         1
blue_frat_rad        1
red_frat_prob        0.001000
blue_frat_prob       0.001000
*
* reconstitution
*
reconst_flag?        0
RED_recon_time       10
BLUE_recon_time      10
*************************************
*  STATISTICS PARAMETERS
*************************************
stat_flag?           0
goal_stat_flag?      0
center_mass_flag?    0
interpoint_flag?     0
entropy_flag?        0
cluster_1_flag?      0
```

```
cluster_2_flag?      0
neighbors_flag?      0
*****************************************************
* RED GLOBAL COMMAND PARAMETERS
*****************************************************
RED_global_flag    0
*****************************************************
* BLUE GLOBAL COMMAND PARAMETERS
*****************************************************
BLUE_global_flag  0
*****************************************************
* RED LOCAL COMMAND PARAMETERS
*****************************************************
RED_command_flag    0
*****************************************************
* BLUE LOCAL COMMAND PARAMETERS
*****************************************************
BLUE_command_flag  0
*****************************************
*  RED ISAACA PARAMETERS
*****************************************
num_reds          100
squads            1
num_per_squad     100  25  25  25   0   0   0   0   0   0
M_RANGE           1    1   1   2   2   2   2   2   2   2
personality       1
*
* ALIVE personality weights
*
w1_a:R_alive_R    10.000  76.00  10.00  10.000  76.000  76.000  76.000  76.000  76.000  76.000
w2_a:R_alive_B    40.100  61.00 99.00  99.100  61.100  61.100  61.100  61.100  61.100  61.100
w3_a:R_injrd_R    10.100  10.100   0.100  -4.100  -4.100  -4.100  -4.100  -4.100  -4.100  -4.100
w4_a:R_injrd_B    40.500 99.500 99.000 99.000  99.500  99.500  76.000  76.000  76.000  76.000
w5_a:R_R_goal      0.000  0.000   0.000 16.100 16.100 -16.100 -16.100 -16.100 -16.100 -16.100
w6_a:R_B_goal     50.000 47.000 25.000 25.000 47.000  47.000  47.000  47.000  47.000  47.000
*
* INJURED personality weights
*
w1_i:R_alive_R    76.000  76.00  20.00  20.000  76.000  76.000  76.000  76.000  76.000  76.000
w2_i:R_alive_B    61.100  61.00 99.100 99.100  61.100  61.100  61.100  61.100  61.100  61.100
w3_i:R_injrd_R    -4.100  -4.100   0.100  -4.100  -4.100  -4.100  -4.100  -4.100  -4.100  -4.100
w4_i:R_injrd_B    99.500  99.500 99.000 99.000 99.500  99.500  76.000  76.000  76.000  76.000
w5_i:R_R_goal     50.100  0.100  0.000  16.100  16.100 -16.100 -16.100 -16.100 -16.100 -16.100
w6_i:R_B_goal     0.000  47.000 25.000 25.000  47.000  47.000  47.000  47.000  47.000  47.000
*
* ISAACA-LC weights
*
w7:R_loc_comdr    1.000000
w8:R_loc_goal     1.000000
*
* defense parameters
*
defense_flag      1
alive_strength    1 1 1 2 1 1 1 1 1 1
injured_strength  1 1 1 2 1 1 1 1 1 1
*
* sensor/fire ranges
*
S_RANGE           5  5  7  8  8  8  8  8  8  8
F_RANGE           3  3  3  3  8  8  8  8  8  8
*
* communications
*
COMM_flag         0
COMM_range        0
```

```
COMM_weight      0.000000
*
* movement constraints
*
movement_flag      1
C_RANGE            5  4  3  5  7  7  7  7  7  7
A:ADVANCE_num 3    3  3  0  0  1  1  1  1  1  1
A:CLUSTER_num      3  5  7  7  16 16 16 16 16 16
A:COMBAT_num       0  1  -5 -5 -1 -1 -1 -1 -1 -1
I:ADVANCE_num 4    4  0  0  9  9  9  9  9  9
I:CLUSTER_num      9  9  7  7  9  9  9  9  9  9
I:COMBAT_num       6  6  -5 -5 -16 -16 -16 -16 -16 -16
C_RANGE_(m,M)  1,4
A:ADV_(m,M)        0,4
A:CLUS_(m,M)       1,0
A:COMB_(m,M)       -3,3
I:ADV_(m,M)        0,4
I:CLUS_(m,M)       1,8
I:COMB_(m,M)       -3,3
A:R_R_min_dist     0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
A:R_B_min_dist     0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
A:R_R_goal_min     20.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
I:R_R_min_dist     0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
I:R_B_min_dist     0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
I:R_R_goal_min     10.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
*
* combat/engagement
*
shot_prob          0.005  0.007  0.010  0.005  0.005  0.005  0.005  0.005  0.005  0.005
R_max_eng_num 5    5  7  10  2  2  2  2  2  2
****************************************
* BLUE ISAACA PARAMETERS
****************************************
num_blues          100
squads             1
num_per_squad      100  15  23  0  0  0  0  0  0  0
M_RANGE            1  1  1  2  2  2  2  2  2  2
personality        1
*
* ALIVE personality weights
*
w1_a:B_alive_B     10.000  10.000  76.000  76.000  76.000  76.000  76.000  76.000  76.000  76.000
w2_a:B_alive_R     40.100 100.00  61.00  61.100  61.100  61.100  61.100  61.100  61.100  61.100
w3_a:B_injrd_B     10.000  0.000  0.000  -4.100  -4.100  -4.100  -4.100  -4.100  -4.100  -4.100
w4_a:B_injrd_R     40.500  99.500  76.000  76.000  99.500  99.500  76.000  76.000  76.000  76.000
w5_a:B_B_goal      0.000  0.000  0.000 -16.100 -16.100 -16.100 -16.100 -16.100 -16.100 -16.100
w6_a:B_R_goal      50.000  99.000  47.000  47.000  47.000  47.000  47.000  47.000  47.000  47.000
*
* INJURED personality weights
*
w1_i:B_alive_B     10.000  76.000  76.000  76.000  76.000  76.000  76.000  76.000  76.000  76.000
w2_i:B_alive_R     40.100  61.100  61.100  61.100  61.100  61.100  61.100  61.100  61.100  61.100
w3_i:B_injrd_B     10.100  -4.100  -4.100  -4.100  -4.100  -4.100  -4.100  -4.100  -4.100  -4.100
w4_i:B_injrd_R     40.500  99.500  76.000  76.000  99.500  99.500  76.000  76.000  76.000  76.000
w5_i:B_B_goal      0.100 -16.100 -16.100 -16.100 -16.100 -16.100 -16.100 -16.100 -16.100 -16.100
w6_i:B_R_goal      50.000  47.000  47.000  47.000  47.000  47.000  47.000  47.000  47.000  47.000
*
* ISAACA-LC weights
*
w7:B_loc_comdr     1.000000
w8:B_loc_goal      1.000000
*
* defense parameters
*
defense_flag       1
```

```
alive_strength      1 1 1 1 1 1 1 1 1 1
injured_strength    1 1 1 1 1 1 1 1 1 1
*
* sensor/fire ranges
*
S_RANGE             5  8  7  3  3  3  3  3  3  3
F_RANGE             3  4  3  3  3  3  3  3  3  3
*
* communications
*
COMM_flag           0
COMM_range          0
COMM_weight         0.000000
*
* movement constraints
*
movement_flag       1
C_RANGE             3  4  3  7  7  7  7  7  7  7
A:ADVANCE_num       2  0  0  1  1  1  1  1  1  1
A:CLUSTER_num       8  5  7 16 16 16 16 16 16 16
A:COMBAT_num        3 -5 -1 -1 -1 -1 -1 -1 -1 -1
I:ADVANCE_num       3  9  9  9  9  9  9  9  9  9
I:CLUSTER_num      12  9  9  9  9  9  9  9  9  9
I:COMBAT_num        5 -2 -16 -16 -16 -16 -16 -16 -16 -16
C_RANGE_(m,M)       1,4
A:ADV_(m,M)         0,4
A:CLUS_(m,M)        1,8
A:COMB_(m,M)        -3,3
I:ADV_(m,M)         0,4
I:CLUS_(m,M)        1,8
I:COMB_(m,M)        -3,3
A:R_R_min_dist      0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
A:R_B_min_dist      0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
A:R_R_goal_min      0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
I:R_R_min_dist      0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
I:R_B_min_dist      0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
I:R_R_goal_min      0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
*
* combat/engagement
*
shot_prob           0.005 0.010 0.010 0.005 0.005 0.005 0.005 0.005 0.005 0.005
B_max_eng_num       5  3  2  2  2  2  2  2  2  2
**********************************
* TERRAIN PARAMETERS
**********************************
(1)_size            2
(1)_center_x        32
(1)_center_y        50
(2)_size            5
(2)_center_x        44
(2)_center_y        44
(3)_size            2
(3)_center_x        58
(3)_center_y        50
```

Sample Data Input Files for ISAAC_GA

ISAAC_GA uses two input data files:

- **GA_ISAAC.dat**, which is the default name of the file that contains a truncated version of ISAAC's input data file (see ***Contents of ISAAC's Input Data File*** in ***A Concise User's Guide to ISAAC)***
- **GA_DATA.dat**, which is the default name of the file that contain GA-specific data entries needed to start the run (see ***Contents of ISAAC_GA's Input Data File*** in ***Genetic Algorithm Evolutions of ISAACA Personalities***).

GA_ISAAC.dat

Note that **GA_ISAAC.dat** defines single-squad red and blue forces and does not contain any command and control parameters.

```
*****************************************
* GENERAL BATTLE PARAMETERS
*****************************************
battle_size          50
*
* initial distribution
*
init_dist            1
R_box_(l,w)          15,15
RED_cen_(x,y)        10,10
B_box_(l,w)          15,15
BLUE_cen_(x,y)       40,40
B_flag_(x,y)         49,49
R_flag_(x,y)         1,1
termination?         2
move_order?          2
combat_flag?         2
terrain_flag?        0
*
* fratricide parameters
*
red_frat_flag?       0
blue_frat_flag?      0
red_frat_rad         1
blue_frat_rad        1
red_frat_prob        0.001000
blue_frat_prob       0.001000
*
* reconstitution
*
reconst_flag?        0
RED_recon_time       1
BLUE_recon_time      1
***********************************
*  STATISTICS PARAMETERS
***********************************
stat_flag?           0
```

```
goal_stat_flag?      0
center_mass_flag?    0
interpoint_flag?     0
entropy_flag?        0
cluster_1_flag?      0
cluster_2_flag?      0
neighbors_flag?      0
*****************************************************
* RED GLOBAL COMMAND PARAMETERS
*****************************************************
RED_global_flag    0
*****************************************************
* BLUE GLOBAL COMMAND PARAMETERS
*****************************************************
BLUE_global_flag   0
*****************************************************
* RED LOCAL COMMAND PARAMETERS
*****************************************************
RED_command_flag  0
*****************************************************
* BLUE LOCAL COMMAND PARAMETERS
*****************************************************
BLUE_command_flag 0
************************************
* RED ISAACA PARAMETERS
************************************
num_reds            50
M_RANGE             1
personality         1
*
* ALIVE personality weights
*
w1_a:R_alive_R      10.000000
w2_a:R_alive_B      40.000000
w3_a:R_injrd_R      10.000000
w4_a:R_injrd_B      40.000000
w5_a:R_R_goal        0.000000
w6_a:R_B_goal        0.000000
*
* INJURED personality weights
*
w1_i:R_alive_R      10.000000
w2_i:R_alive_B      40.000000
w3_i:R_injrd_R      10.000000
w4_i:R_injrd_B      40.000000
w5_i:R_R_goal        0.000000
w6_i:R_B_goal        0.000000
*
* ISAACA-LC weights
*
w7:R_loc_comdr      1.
w8:R_loc_goal       1.
*
* sensor/fire ranges
*
S_RANGE             1
F_RANGE             1
*
* communications
*
COMM_flag           0
COMM_range          0
COMM_weight         0.000000
*
* movement constraints
```

```
*
movement_flag       1
C_RANGE             3
A:ADVANCE_num       1
A:CLUSTER_num       5
A:COMBAT_num        -3
I:ADVANCE_num       1
I:CLUSTER_num       5
I:COMBAT_num        -3
C_RANGE_(m,M)       0,0
A:ADV_(m,M)         0,0
A:CLUS_(m,M)        0,0
A:COMB_(m,M)        0,0
I:ADV_(m,M)         0,0
I:CLUS_(m,M)        0,0
I:COMB_(m,M)        0,0
A:R_R_min_dist      0.000000
A:R_B_min_dist      0.000000
A:R_R_goal_min      0.000000
I:R_R_min_dist      0.000000
I:R_B_min_dist      0.000000
I:R_R_goal_min      0.000000
*
* combat/engagement
*
shot_prob           0.005000
R_max_eng_num       6
*************************************
* BLUE ISAACA PARAMETERS
*************************************
num_blues           50
M_RANGE             1
personality         1
*
* ALIVE personality weights
*
w1_a:B_alive_B      5.000000
w2_a:B_alive_R      99.000000
w3_a:B_injrd_B      0.000000
w4_a:B_injrd_R      99.000000
w5_a:B_B_goal       0.000000
w6_a:B_R_goal       50.000000
*
* INJURED personality weights
*
w1_i:B_alive_B      5.000000
w2_i:B_alive_R      99.000000
w3_i:B_injrd_B      0.000000
w4_i:B_injrd_R      99.000000
w5_i:B_B_goal       0.00
w6_i:B_R_goal       50.000000
*
* ISAACA-LC weights
*
w7:B_loc_comdr      1.
w8:B_loc_goal       1.
*
* sensor/fire ranges
*
S_RANGE             4
F_RANGE             3
*
* communications
*
COMM_flag           0
```

```
COMM_range      0
COMM_weight     0.000000


*
* movement constraints
*
movement_flag       1
C_RANGE             3
A:ADVANCE_num       0
A:CLUSTER_num       1
A:COMBAT_num        -3
I:ADVANCE_num       0
I:CLUSTER_num       1
I:COMBAT_num        -3
C_RANGE_(m,M)       0,0
A:ADV_(m,M)         40,0
A:CLUS_(m,M)        0,0
A:COMB_(m,M)        10,0
I:ADV_(m,M)         0,0
I:CLUS_(m,M)        40,0
I:COMB_(m,M)        0,0
A:B_B_min_dist      0.000000
A:B_R_min_dist      0.000000
A:B_B_goal_min      0.000000
I:B_B_min_dist      0.000000
I:B_R_min_dist      0.000000
I:B_B_goal_min      0.000000
*
* combat/engagement
*
shot_prob           0.005000
B_max_eng_num       6
*********************************
* TERRAIN PARAMETERS
*********************************
```

GA_DATA.dat

```
*****************************************
*
*           GA parameters
*
*****************************************
num_generations                 50
num_initial_conds               25
max_time_to_goal                100
penalty_power                   2
best_personalities_to_file?     1
min_dist_genes_flag             0
initial_condition_genes_flag    0
*****************************************
*
*       penalty weights (1-100)
*
*****************************************
w1_time_to_goal                 0
w2_friendly_loss                0
w3_enemy_loss                   0
w4_red_to_blue_survival_ratio   0
w5_friendly_CM_to_enemy_flag    0
w6_enemy_CM_to_friendly_flag    10
w7_friendly_near_enemy_flag     0
```

```
w8_enemy_near_friendly_flag    10
w9_red_fratricide_hits         0
w10_blue_fratricide_hits       0
*****************************************
*
*        termination parameters
*
*****************************************
termination_code?              4
flag_containment_range         12
containment_number             10
red_CM_to_BF_frac              .5
*****************************************
*
*          ISAACA chromosome
*
*****************************************
gene[1]:S_range                1,10
gene[2]:F_range                1,10
gene[3]:C_range                1,10
gene[4]:w1_alive               0,100
gene[5]:w1_alive_sign          0,1
gene[6]:w2_alive               0,100
gene[7]:w2_alive_sign          0,1
gene[8]:w3_alive               0,100
gene[9]:w3_alive_sign          0,1
gene[10]:w4_alive              0,100
gene[11]:w4_alive_sign         0,1
gene[12]:w5_alive              0,100
gene[13]:w5_alive_sign         0,1
gene[14]:w6_alive              0,100
gene[15]:w6_alive_sign         0,1
gene[16]:w1_injured            0,100
gene[17]:w1_injured_sign       0,1
gene[18]:w2_injured            0,100
gene[19]:w2_injured_sign       0,1
gene[20]:w3_injured            0,100
gene[21]:w3_injured_sign       0,1
gene[22]:w4_injured            0,100
gene[23]:w4_injured_sign       0,1
gene[24]:w5_injured            0,100
gene[25]:w5_injured_sign       0,1
gene[26]:w6_injured            0,100
gene[27]:w6_injured_sign       0,1
gene[28]:ADV_alive             0,20
gene[29]:CLS_alive             0,50
gene[30]:CBT_alive             0,50
gene[31]:CBT_alive_sign        0,1
gene[32]:ADV_injured           0,20
gene[33]:CLS_injured           0,50
gene[34]:CBT_injured           0,50
gene[35]:CBT_injured_sign      0,1
gene[36]:min_dist_flag         0,1
gene[37]:R_R_min_dist_alive    0,10
gene[38]:R_B_min_dist_alive    0,10
gene[39]:R_R_goal_min_alive    0,40
gene[40]:R_R_min_dist_injured  0,10
gene[41]:R_B_min_dist_injured  0,10
gene[42]:R_R_goal_min_injured  0,40
gene[43]:initial_box_size      1,50
gene[44]:initial_box_center_x  1,30
gene[45]:initial_box_center_y  1,30
```

Sample Data Input File for ISAAC_PM: PHASE.dat

ISAAC_PM uses two input data files:

- **P_ISAAC.dat**, that contains a truncated version of ISAAC's input data file (see ***Contents of ISAAC's Input File*** in ***Concise User's Guide to ISAAC***). This file has exactly the same form as **GA_ISAAC.dat** shown above. Like **GA_ISAAC.dat**, **P_ISAAC.dat** defines single-squad red and blue forces and does not contain any command and control parameters.
- **PHASE.dat**, that contains **ISAAC_PM**-specific data entries needed to start the run. It is essentially a truncated version of **GA_DATA.dat** (see above). See ***Contents of ISAAC_PM's Data Input File: Phase.dat.***

```
*****************************************
*
*       GA parameters
*
*****************************************
num_initial_conds                15
max_time_to_goal                 125
penalty_power                    2
*****************************************
*
*    penalty weights (1-100)
*
*****************************************
w1_time_to_goal                  0
w2_friendly_loss                 10
w3_enemy_loss                    0
w4_red_to_blue_survival_ratio    0
w5_friendly_CM_to_enemy_flag     0
w6_enemy_CM_to_friendly_flag     0
w7_friendly_near_enemy_flag      0
w8_enemy_near_friendly_flag      0
w9_red_fratricide_hits           0
w10_blue_fratricide_hits         0
*****************************************
*
*     termination parameters
*
*****************************************
termination_code?                4
flag_containment_range           15
containment_number               10
red_CM_to_BF_frac                .5
```

References

1. A. Ilachinski, *Land Warfare and Complexity, Part I: Mathematical Background and Technical Sourcebook* (U), Center for Naval Analyses Information Manual CIM-461, July 1996, Unclassified.

2. A. Ilachinski, *Land Warfare and Complexity, Part II: An Assessment of the Applicability of Nonlinear Dynamics and Complex Systems Theory to the Study of Land Warfare* (U), Center for Naval Analyses Research Memorandum CRM-68, July 1996, Unclassified.

3. F. W. Lanchester, "Aircraft in warfare: the dawn of the fourth arm – No. V, the principle of concentration," *Engineering*, Volume 98, 1914, 422-423. (Reprinted on pages 2138-2148, *The World of Mathematics*, Volume IV, edited by J. Newman, Simon and Schuster, 1956.)

4. J. V. Chase, "A mathematical investigation of the effect of superiority in combats upon the sea," 1902, reprinted in B. A. Fiske, The Navy as a Fighting Machine, Annapolis, MD: U.S. Naval Institute Press, 1988.

5. M. Osipov, "The Influence of the Numerical Strength of Engaged Forces in Their Casualties," translated by R. L. Helmbold and A. S. Rehm, *Naval Research Logistics*, Volume 42, No. 3, April 1995, 435-490.

6. W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," *Physica D*, Volume 42, 1990, 228-234.

7. A. Ilachinski, *A Mobile Cellular Automata Approach to Land Combat: A User's Guide to an Early Version of ISAAC* (U), Center for Naval Analyses Information Manual CIM-482, September 1996, Unclassified.

8. D. S. Hartley III and R. L. Helmbold, "Validating Lanchester's square law and other attrition models," *Naval Research Logistics*, Volume 42, May, 1995, 609-633.

9. H. K. Weiss, "Lanchester-type models of warfare," Proceedings of 1st Conference on Operations Research, Operations Research Society of America, 1957, 82-99.

10. J. Fain, "The Lanchester equations and historical warfare: an analysis of sixty world war II land engagements," in *Proceedings of the 34th Military Operations Research Symposium*, Alexandria, Virginia: Military Operations Research Society, 1975.

11. Richard D. Hooker, Jr., editor, *Maneuver Warfare: An Anthology*, Presidio, 1993.

12. C. G. Langton, editor, *Artificial Life: The Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems Held September, 1987 in Los Alamos, New Mexico*, Addison-Wesley, 1989.

13. C. Reynolds, "Flocks, herds, and schools: a distributed behavioral model," *Computer Graphics*, Volume 21, July, 1987, 25.

14. J. Deneubourg, , S. Goss, N. Franks, A. Sendova-Franks, C. Detrain and L. Chretien, "The dynamics of collective sorting robot-like ants and ant-like robots," pages 356-363 in *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior,* edited by Jean-Arcady Meyer and Stewart W. Wilson, MIT Press, 1991.

15. J. Casti, *Would-be Worlds: How Simulation is Changing the Frontiers of Science,* John Wiley and Sons, 1997..

16. J. M. Epstein and R. Axtell, *Growing Artificial Societies: Social Science From the Bottom Up*, MIT Press, 1996.

17. P. Maes, "Modeling Adaptive Autonomous Agents," *Artificial Life*, Volume 1, No. 1, 1994, 135-162.

18. L. Steels and R. Brooks, editors, *The Artificial Life Route to Artificial Intelligence,* Lawrence Erlbaum Associates, 1995.

19. P. Maes, "Behavior-based artificial intelligence," pages 2-10 in *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior,* edited by Jean-Arcady Meyer, H. L. Roitblat and Stewart W. Wilson, MIT Press, 1993.

20. N. Boccara, O. Roblin and M. Roger, "Automata network predator-prey model with pursuit and evasion," *Physical Review E*, Volume 50, No. 6, December 1994, 4531-4541.

21. O. Miramontes, R. Sole and B. Goodwin, "Collective behavior of random-activated mobile cellular automata," *Physica D*, Volume 63, 1993, 145-160.

22. B. R. Sutherland and A. E. Jacobs, "Self-organization and scaling in a lattice predator-prey model," *Complex Systems*, Volume 8, 1994, 385-405.

23. A. E. R. Woodcock, L. Cobb and J.T. Dockery, "Cellular Automata: A New Method for Battlefield Simulation," *Signal*, January, 41-50.

24. J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*, Second Edition, MIT Press, 1992.

25. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

26. L. Davis, editor, *Handbook of Genetic Algorithms*, von Nostrand Reinhold, 1991.

27. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 2nd Edition, Springer-Verlag, 1996.

28. S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kauffman, 1993.

29. M. Mitchell, "Genetic algorithms," pages 3-87 in *1992 Lectures in Complex Systems*, edited by L. Nadel and D. Stein, Addison-Wesley, 1993.

30. N. M. Smith, "A calculus for ethics: a theory of the structure of value," *Behavioral Science*, Volume 1, No. 2, April, 1956, 111-142, 186-211.

31. L. P. Kaelbling, M. L. Littman and A. W. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, Volume 4, 1996, 237-285.

32. R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, Volume 3, 1988, 9-44.

33. G. Tesauro, "Temporal difference learning and TD-Gammon," *Communications of the ACM*, Volume 38, No. 3, 1995, 58-67.

34. F. Carvalho-Rodrigues, "A proposed entropy measure for assessing combat degradation," *Jour. Opl. Res. Soc.* (UK), Volume 40, No. 8, 1989, 789-793.

35. J. T. Dockery and A. E. R. Woodcock, *The Military Landscape: Mathematical Models of Combat*, Cambridge, England: Woodhead Publishing Limited, 1993.

36. P. J. Denning, "Modeling Reality," *American Scientist*, November-December 1990, 495-498.

37. J. Casti, *Reality Rules*, Volumes I and II, John Wiley and Sons, 1992.

38. R. Axelrod and W. D. Hamilton, "The evolution of cooperation," *Science*, Volume 211, 1981, 1390-1396.

39. S. Kauffman, *At Home in the Universe: The Search for Laws of Self-Organization and Complexity*, Oxford University Press, 1995.

40. P. Bak and K. Chen, "Self-Organized Criticality," *Scientific American*, Volume 26, January 1991, 46-53.

41. L. F. Richardson, *Statistics of Deadly Quarrels*, Pitsburg, PA: Boxwood, 1960.

42. N. Boccara, E. Goles, S. Martinez, and P. Picco, *Cellular Automata and Cooperative Systems*, Conference Proceedings, Les Houches, France, June 22 – July 2, 1992, NATO ASI Series Volume 396, Kluwer Academic Publishers, 1993.

43. H. A. Gutowitz, editor, *Cellular Automata: Theory and Experiment*, Elsevier Science Publishers, 1990.

44. S. Wolfram, editor, *Theory and Applications of Cellular Automata*, World Scientific, 1986.

45. S. Wolfram, *Cellular Automata and Complexity: Collected Papers*, Addison-Wesley, 1994.

46. T. Toffoli and N. Margolus, *Cellular Automata Machines: a New Environment for Modeling*, MIT Press, 1987.

47. Michael R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

48. R. G. Palmer, "Optimization on rugged landscapes," pages 3-26 in *Molecular Evolution on Rugged Landscapes: Proteins, RNA and the Immune System*, edited by A. S. Perelson and S. A. Kauffman, Addison-Wesley, 1991.

49. S. Kauffman and S. Johnson, "Coevolution to the edge of chaos: coupled fitness landscapes, poised states and coevolutionary avalanches," pages 325-370 in *Artificial Life II: The Proceedings of the Workshop on Artificial Life Held February, 1990 in Santa Fe, New Mexico*, edited by C. G. Langton, C. Taylor, J. Doyne Farmer and S. Rasmussen, Addison-Wesley, 1992.

50. S. Kauffman, *Origins of Order: Self-Organization and Selection in Evolution*, Oxford University Press, 1993.

51. J. Hoshen and R. Kopelman, *Physical Review B*, Volume 14, 1976, 3428.

52. J. H. Holland, "Genetic Algorithms," *Scientific American*, Volume 278, No. 1, July, 1992, 66-72.

53. P. Tzionas, P. Tsalidis, and A. Thanailakis, "Three-dimensional minimum cost path planning using cellular automata architectures," *Mobile Robots*, Volume 7, 1992, 297.

54. J. Casti, "What if...", *New Scientist*, 13 July 1996, 36-40.

55. R. Leonhard, *The Art of Maneuver: maneuver-Warfare Theory and AirLand Battle*, Presidio, 1991.

56. W. S. Lind, *Maneuver Warfare Handbook*, Westview Press, 1985.

57. J. H. Holland, *Hidden Order: How Adaptation Builds Complexity*, Addison-Wesley Publishing Company, 1995.

58. E. Hillebrand and J. Stender, editors, *Many-Agent Simulation and Artificial Life*, IOS Press,1994.

59. M. J. wooldridge and N. R. Jennings, editors, *Intelligent Agents: ECAI-94 Workshop*, Springer-Verlag, 1995.

60. K. Nagel and S. Rasmussen, "Traffic at the edge of chaos," pages 222-235 in *Artificial Life IV*, edited by R. A. Brooks and P. Maes, MIT Press,1994.

61. M. Resnick, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*, MIT Press, 1994.

List of Figures

Figure 1. The force-on-force attrition "challenge" 6
Figure 2. *Artificial Life:* a possible new approach to the classic force-on-force attrition problem? 8
Figure 3. Evolution of a one-dimensional CA starting from a random initial state 10
Figure 4. Collective sorting by ant-like robots 12
Figure 5. Schematic of ISAAC's hierarchy of information levels 23
Figure 6. Putative two-dimensional "Combat Battlefield" in ISAAC 26
Figure 7. Various kinds of ranges that surround each ISAACA 28
Figure 8. Set of possible ISAACA moves from its current (x,y) position 32
Figure 9. General movement rule 33
Figure 10. Sample penalty calculation 33
Figure 11. Schematic of ISAACA *Meta*-Personality 34
Figure 12. Sample constraint rules 37
Figure 13. Blue X targets 3 Red ISAACAs 38
Figure 14. Schematic of a fratricide "hit" of X' by X 39
Figure 15. Schematic of ISAACA communications 40
Figure 16. Schematic representation of a ISAACA C^2 hierarchy 41
Figure 17. Local command 42
Figure 18. Local command (sample calculation) 45
Figure 19. Plot of health(LC) versus number of enemy ISAACAs 46
Figure 20. Rule for GC mediated LC-LC interaction 47
Figure 21. Global command 48
Figure 22. Rule for GC command of LC movement 49
Figure 23. ISAAC's opening screen 52
Figure 24. ISAAC's main option screen 53
Figure 25. ISAAC's second option screen 53
Figure 26. General battle parameters 56
Figure 27. Statistics Parameters 60

Figure 28. Blue global command parameters 62
Figure 29. Blue local command parameters 64
Figure 30. Blue ISAACA Parameters 70
Figure 31. Terrain parameters 82
Figure 32. ISAAC's main graphics display 84
Figure 33. The Attrition data field of ISAAC's main graphics display 88
Figure 34. Main menu of the "on-the-fly" parameter change option 91
Figure 35. Screen shot of the "on-the-fly" **Combat Parameter** change sub-menu 92
Figure 36. Sample Run #1 – **MISMATCH.out** 104
Figure 37. Sample Run #2 – **FLUID_1.out** 107
Figure 38. Sample Run #3 – **FLUID_2.out** 109
Figure 39. Sample Run #4 – **PRECESS.out** 112
Figure 40. Sample Run #5 – **GOALDEF1.out** 114
Figure 41. Sample Run #6 – **GOALDEF2.out** 116
Figure 42. Sample Run #7 – **CIRCLE.out** 118
Figure 43. Sample Run #8 – **FIRESTM1.out** 120
Figure 44. Sample Run #9 – **FIRESTM2.out** 122
Figure 45. Sample Run #10 – **SENSOR.out** 125
Figure 46. Fragment of **LOCALCMD.dat** input data file 126
Figure 47. Sample Run #11 – **LOCALCMD.out** 128
Figure 48. Fragment of **GLBALCMD.dat** input data file 129
Figure 49. Sample Run #12 – **GLBALCMD.out** 131
Figure 50. Sample Run #13 – **BATTLE1.out** 133
Figure 51. Battlefield partitioned into an 8-by-8 array of sub-blocks of size B/8 139
Figure 52. Opening screen for **BATTLE.out** 143
Figure 53. Snapshot views of **BATTLE.out** 144
Figure 54. Sample statistics measures for **BATTLE.out** 145
Figure 55. Schematic of taking a two dimensional (x_1, x_2) "slice" through ISAAC's N-dimensional parameter space 146
Figure 56. Schematic of how **ISAAC_PM** works 149
Figure 57. **ISAAC_PM**'s opening screen 150
Figure 58. **ISAAC_PM**'s file name prompt screen 151
Figure 59. **ISAAC_PM**'s prompt screen for specifying the x-coodinate 151

Figure 60. **ISAAC_PM**'s range/sample prompt screen 152
Figure 61. Sample **PHASE.dat** input data file 153
Figure 62. **ISAAC_PM**'s main graphics display 156
Figure 63. Schematic of initial state for sample run #1 157
Figure 64. Output of **ISAAC_PM** for sample run #1 158
Figure 65. Output of **ISAAC_PM** for sample run #2 159
Figure 66. Output of **ISAAC_PM** for sample run #3 160
Figure 67. Schematic of the GA "problem" in ISAAC 167
Figure 68. Schematic for fitness function *f* (corresponding to mission primitive *m)* that is internally maximized by the program 171
Figure 69. **ISAAC_GA**'s opening screen 178
Figure 70. **ISAAC_GA**'s file name prompt screen 179
Figure 71. Sample **ISAAC_GA** data input file 185
Figure 72. Sample contents of the output file **GA_STAT.dat** 187
Figure 73. **ISAAC_GA**'s main graphics display 191
Figure 74. Typical GA learning curve 192
Figure 75. Fragment of **GA_DATA.dat** input data file for Run #1 194
Figure 76. Sample GA Run #1 – **AWAY_1.out** 198
Figure 77. Sample GA Run #2 – **GOAL_2.out** 199
Figure 78. Sample GA Run #2 – **GOAL_6.out** 200
Figure 79. Sample GA Run #2 – **GOAL_1.out** 201
Figure 80. Illustration of a neural-net assigned move m (at time t+1) given a local state S at time t 214
Figure 81. A schematic representation of "nesting" in ISAAC 216
Figure 82. Interplay between experience and theory in the forward- and inverse-problems of complex systems theory 229
Figure 83. A schematic representation of an ISAACA C^2 structure 230
Figure 84. Behavior (arbitrary measure) as a function of C^2 structure 231
Figure 85. Schematic representation of the combat phase space 234
Figure 86. A schematic representation of how genetic algorithms may be used to find the "best" partitioning of the combat information-space 241
Figure 87. Example of a one-dimensional CA 248

Figure 88. Glider patterns in Conway's Life 249
Figure 89. Two-dimensional lattice-gas simulation of a fluid 250
Figure 90. Collective behavior of a four dimensional CA 251
Figure 91. Schematic representation of the basic GA operators 254
Figure 92. Sample forms of fitness landscapes 257
Figure 93. A fragment of a cluster of ISAACAs and the result of applying the cluster-counting heuristic 350

List of Tables

Table 1. Land combat as a complex adaptive system 2

Table 2. Eight tiers of applicability 4

Table 3. Comparison between traditional and agent-based approaches to complex systems modeling 17

Table 4. A listing of program "modules" making up ISAAC 20

Table 5. Components of the personality weight vector 30

Table 6. Tradeoff between available memory and some basic run-time parameters in ISAAC 51

Table 7. ISAAC output files corresponding to the sample runs shown in figures 36 through 50 100

Table 8. Description of ISAAC's data output files 136

Table 9. ISAACA Chromosome 168

Table 10. A description of GA weights 169

Table 11. ISAAC output files corresponding to the sample GA runs shown in figures 76 through 79 193

Table 12. ISAAC functions 299

Distribution list

Research Memorandum 97-61.10

Commanding General, Marine Corps Combat Development Center

CDR Whitener

NAS Oceana (VF-11)

Virginia Beach, VA 23456