Genetic Algorithms Applied to a Mission Routing Problem

THESIS

James B. Olsan
Captain, USAF

AFIT/GCE/ENG/93-12

AFIT/GCE/ENG/93-12

Genetic Algorithms Applied to a Mission Routing Problem

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

James B. Olsan, B.S.

Captain, USAF

December, 1993

Acknowledgements

I would like to thank the members of my committee for their knowledge and help. Maj Gregg H. Gunsch provided me with knowledge of artificial intelligence which aided my general understanding of search techniques. Lt Col William H. Hobart furnished me with a knowledge of parallel processing, which created a foundation for my understanding of parallel algorithms. Dr Gary B. Lamont, my advisor, integrated my knowledge of search techniques with parallel processing. With respect to the mission routing problem, he prompted me to look deeper into genetic algorithms while at the same time exposing me to the broader picture of all search techniques.

James B. Olsan

Table of Contents

List of Figures

Abstract

This thesis applies genetic algorithms to a mission routing problem.

The mission routing problem involves determining an aircraft's best route between a staging base and a target. The goal is to minimize the route distance and the exposure to radar. Potential routes are mapped to a 3-dimensional mesh where the nodes correspond to checkpoints in the route and the arcs correspond to partial paths of the route. Each arc is weighted with respect to distance and exposure to radar.

A genetic algorithm is a probabilistic search technique loosely based on theories of biological evolution. Genetic crossover and "survival of the fittest" are the basis of a genetic algorithm's control structure and can be used for general problems. Encoding and evaluation of the data structure is problem specific.

This thesis focuses on approaches to mapping the mission routing problem's mesh to a data structure which the genetic algorithm's control structure can effectively and efficiently manipulate. Three broad methods are proposed: Bounded Progress, Free Progress, and Restricted Progress. The Bounded Progress method uses a tightly-coupled mesh while the Free Progress method uses a loosely-coupled mesh. The Restricted Progress method is a hybrid of the other two methods.

Genetic Algorithms Applied to a Mission Routing Problem

I. Introduction.

This investigation proposes to use serial and parallel genetic algorithms to solve a mission routing problem. This introductory chapter presents background information, the problem, scope of the problem, the approach, and assumptions effecting the approach. This chapter concludes with an overview of this thesis.

1.1 Background

This section provides background information needed to understand the thesis problem. This information includes material on combinatorial problems, the mission routing problem, approaches to combinatorial problems, genetic algorithms, and parallel processing.

1.1.1 Combinatorial Optimization Problems . Garey and Johnson (21:83) define a combinatorial optimization problem as having a set of instances (D_π), a set of candidate solutions (S_π) for each instance (I), and an objective function ($m_\pi(I, \sigma)$) to evaluate each candidate solution for each instance. A brute force method to find the optimal solution for a combinatorial problem is to evaluate each candidate solution and pick the one with the optimal (minimum or maximum) objective function. The larger the set of candidate solutions, the longer this method takes.

Order-of analysis is used to analyze time and space requirements of an algorithm to solve a problem. Brassard and Bratley (2:6) define the order-of an algorithm as $t(n)$ such that all instances of a problem can be solved within $ct(n)$ seconds, where c is a positive constant. Let n represent the size of an instance I. If the brute force method previously described is used to solve a problem, then $t(n)$ is proportional to the size of the candidate solution set S_π, or $t(|S_\pi(I)|)$.

Problems in which the candidate solution set size ($S_\pi(I)$) increases much faster than the size of the instance ($t(n) \propto c^n$ or $t(n) \propto n!$) attract considerable attention in algorithm

research. These problems may take unacceptable time to solve. This is because they are required to search for a solution and the search space exhibits this combinatorial explosion (39:40). These combinatorial problems are characterized by parameters which take on discrete, finite values which are combined to create candidate solutions (21:8) (37:1). The Traveling Salesman Problem and the Shortest Path Problem are two instances of combinatorial problems. A detailed example of the TSP easily illustrates the combinatorial explosion of the search space. The Shortest Path Problem is used to offer another example of a combinatorial problem and is used throughout the thesis to describe a mission routing problem.

1.1.1.1 Traveling Salesman Problem (TSP). Given a fully connected graph of nodes and weighted arcs, the TSP determines the shortest path which starts and ends at a given node and traverses the other nodes exactly once (21:4) (37:11) (39:40) (40:288-292).

Example: Given a list of four cities (A, B, C, D), find the shortest distance of a route which travels through each city. Additionally, require that the route start and stop from city A. The following distances are known:

- distance between A and B = 3
- distance between A and C = 4
- distance between A and D = 5
- distance between B and C = 3
- distance between B and D = 100
- distance between C and D = 2

Solution: Enumerate all possible route combinations and determine their associated distances. A B C D A represents the route from A to B, B to C, C to D and D to A.

1. A B C D A : 3 + 3 + 2 + 5 = 13
2. A B D C A : 3 + 100 + 2 + 4 = 109
3. A C B D A : 4 + 3 + 100 + 5 = 112
4. A C D B A : 4 + 2 + 100 + 3 = 109
5. A D B C A ; 5 + 100 + 3 + 4 = 112
6. A D C B A ; 5 + 2 + 3 + 3 = 13

The shortest route would be enumerations 1 (A B C D A) and 6 (A D C B A), each with a total distance of 13.

The traveling salesman problem, requires a search for an optimal solution. First it generates six possible routes. Then, it searches through each possible route to determine the shortest distance. For this problem the search size was small, only six possible routes. However as the number of cities increase, the size of the search space grows combinatorially (order-of = $(n-1)!$, where n= number of cities in the problem). At ten cities this problem has a search space of 362,880; at 20 cities, $1.2x10^{17}$; at 30 cities, $8.84x10^{30}$. This problem quickly becomes unmanageable. Thirty cities is a small instance size of the TSP, but the candidate solution set is just too large to check each and every candidate.

1.1.1.2 Shortest-Path Problem (SPP). Given a graph of weighted arcs and nodes, find the shortest path between two given nodes (21:214). This is similar to the TSP in that a shortest path is desired. It is unlike the TSP in that the start/end nodes aren't the same and not all nodes must be visited.

1.1.2 Mission Routing Problem. Mission routing is a type of combinatorial problem of practical importance to USAF. Assume a fleet of aircraft and multiple targets. Each vehicle is assigned a group of targets to visit. The problem then is to determine the order in which to visit the targets. This type of mission routing problem maps to the traveling salesman problem where the staging base represents the starting/ending city and each target represents a city to visit. The distances between cities is redefined as the distance/detectability among the staging base and targets. However this distance/detectability among the base and targets is not yet defined.

Defining the distance/detectability among the base and targets maps to the shortest-path problem. The starting/ending node corresponds to two targets or one target and the staging base. The other nodes on the 3-dimensional mesh correspond to the intermediate route points (in 3-dimensional space) between the two targets or the staging base and target. The arcs between nodes correspond to distance/detectability between intermediate route points.

All of this produces a large search space (candidate solutions) for the mission routing problem. Before solving the first combinatorial problem (determining target order), multiple combinatorial sub-problems (best route among targets/staging base) must be solved.

1.1.3 Approaches to Combinatorial Problems. This section presents two broad approaches to solving combinatorial problems. Deterministic algorithms make all decisions in a deterministic manner, while probabilistic algorithms make some decisions based on probability. Deterministic approaches may guarantee a best solution (or a solution within a specified tolerance of the best solution), but may take too long to solve (as provided by the TSP example in section 1.1.1). Probabilistic methods aren't guaranteed to find the best solution, but they may run in less time than deterministic methods. The tradeoff between using these two methods relates solution quality to solution time (2:223-228).

Deterministic methods include general purpose branch and bound algorithms and more specialized methods such as greedy algorithms (38). Branch and bound algorithms work on general search problems, while greedy algorithms can be applied to any search problem. The production of optimal results depends on the problem. The branch and bound algorithms methodologically check the entire search space of the problem. Applications of heuristics may allow some of the search space to be searched implicitly, instead of explicitly, which reduces the algorithm search time. Greedy algorithms produce optimal results if the problem domain is such that local optima correspond to global optima.

Probabilistic algorithms produce solutions based on probabilistic selection of solutions. (2:223-228). Widely used probabilistic techniques include random search, simulated annealing, and evolutionary strategies such as evolutionary algorithms and genetic algorithms. Random search is just random guessing of solutions. Although it is unlikely to produce an optimal, or even good solution, it does execute quickly and can be used to provide known bounds to other approaches.

Simulated annealing starts out as a random search, but eventually narrows its focus to the most promising areas (2). After finding some promising areas to search, the algorithm stays within that search area with some probability.. When this probability is low, then

random search occurs. When this probability is high, then a greedy technique is employed within the local search area.

Evolutionary strategies are loosely based on biological evolutionary theories and attempt to evolve solutions in a manner similar to the way organisms have evolved. Evolutionary algorithms and genetic algorithms are two types of evolutionary strategies. Historically, evolutionary algorithms have focused on mutation as the primary operator, while genetic algorithms have focused on crossover as the primary operator (14:618-623). However, both algorithms are starting to make more use of both operators.

1.1.4 Genetic Algorithms. Genetic Algorithms (GA) are probabilistic algorithms loosely based on theories of evolution and natural selection (22). GAs generate a pool of random solutions. The number of these solutions is much smaller than the search space. From this pool, the GA produces successive generations through evaluation, reproduction, and mating. Evaluation determines the fitness of individual solutions. Good solutions have better fitness while weak solutions have poorer fitness. Reproduction uses fitness information in order to reproduce the next generation. The proportion of good solutions should increase in each successive generation, while the proportion of weak solutions should decrease. Once the GA creates a new generation, some solutions mate to produce slightly different solutions. The objective is that two good solutions combine their good characteristics to form an even better solution. After a number of generations, a good solution evolves. Although at first, this may sound like a random generation of solutions, GAs have a mathematical foundation and can be shown to produce good solutions in a reasonable time under proper conditions. Appendix A contains a detailed description of genetic algorithms. Items of particular importance to this thesis are described in chapter II.

1.1.5 Parallel Processing. In the past, improvements in electronics provided speed-up of the processing capability (speed). The speeds of electronic devices are approaching physical limits. The use of parallel processing provides a means to continue the increase in processing capability (35:1-5) (13:1-2). However, the problem must be properly decomposed to make effective use of all processors. Data and control provide two major

areas of problem decomposition. With data decomposition, the same control structure (algorithm) is put on each processor and the data is spread to each processor. In control decomposition, the algorithm is divided into tasks and the tasks are mapped to different processors. The data then flows through the processors as required by the algorithm.

Genetic algorithms efficiently use parallel processing architectures (43) by using data decomposition. Appendix B describes parallel genetic algorithms. Items of particular importance are described in chapter II.

1.2 Problem.

The investigation applies genetic algorithms to a mission routing problem. The goal is to determine various ways to encode routes are such that the control structure of the genetic algorithm works effectively and efficiently with the data structure of the route encodings.

1.3 Scope.

The portion of the mission routing problem addressed is the determination of the best route between a staging base and a target. Once this portion is accomplished, the results also apply to finding the best route between two targets. The primary focus is on route encoding. Two specific mission routing scenarios are used to implement route encodings. No attempt is made to fine tune genetic variables (mutation rate, crossover rate, selection operator, and others) for optimal performance . If fine tuning is to be done, a large number of various problems should be used to fine tune a genetic algorithm so that the fine tuning applies to the general problem and not a specific problem.

The reasoning behind the selection of this problem lies behind the desire to merge AFIT algorithm research with a real-world application. In the past several years, genetic algorithm research has been conducted at AFIT (18) (36) (41), focusing on the genetic algorithms alone. This previous research recommended that genetic algorithms be applied to area of interest to the US Air Force such as the mission routing problem. This problem

has been the focus of AFIT research using a parallel branch and bound algorithm (29) (17).

1.4 Approach.

The approach taken focuses on researching route encodings. Implementation makes extensive use of existing design, code, and test data. Five major objectives are addressed:

- Analyze the limited mission routing domain
- Propose genetic encodings of routes
- Select the two most promising encodings
- Develop encodings on serial processing system
- Port encodings to a parallel processing system

The approach makes extensive use of software engineering practices such as object-oriented design and code reuse. Sparcstations (serial) and the Intel iPSC/2 (parallel, eight-node hypercube) are available at AFIT to compare algorithms.

1.5 Assumptions.

These assumptions provide the foundation upon which route encodings are investigated.

- The GENESIS simple genetic algorithm operates correctly.
- The AFIT parallel adaptation of GENESIS operates correctly.
- Environment routines developed by Grimm (29) are correct.

The GENESIS code was developed by Grefenstette and is widely used in the genetic algorithm research community (28). Although other genetic algorithms are available such as OOGA (10) and Papagena (43), AFIT research has used GENESIS. Since this effort focuses on application development instead of genetic algorithm development, GENESIS is used.

1.6 Layout of the Thesis.

This thesis consists of six chapters. This first chapter provides background information and defines the general problem of the thesis research. Chapter II discusses current literature involved in developing this thesis. Chapter III analyzes the problem, presents the high level design of the solution, and narrows the focus of the research problem. Chapter IV presents the low-level design of the narrowed research problem. Chapter V provides the framework for experiments used. Finally, chapter VI presents conclusions and recommendations derived from the research effort.

II. Literature Review.

This chapter summarizes current literature related to genetic algorithms and mission routing. The literature is grouped in three major areas: mission routing, genetic algorithms, and vehicle routing.

2.1 Mission Routing.

Grimm (29) defined a simple mission routing problem and used a parallelized A* algorithm to solve it. The problem is to select a route from a staging base to a target such that the pilot can complete the mission with least probability of radar detection. The environment consists of six radars on a 19 x 25 x 15 grid. The grid includes terrain elevation to provide for a 3-dimensional search space. The data structures used by Grimm are based on a 3-dimensional mesh. Although a fully connected graph would be more desirable from an accuracy standpoint, a 3-dimensional mesh is a good way to implement the terrain and radar models (since the airspace and terrain are in three dimensions). Fully connected graphs require too much memory (n^2 instead of $2n$, where n is the number of nodes in the graph). Grimm compensates for this loss of accuracy by proving that the best distance found with a mesh is no greater than 1.4 times the best distance found with a fully connected graph.

2.2 Genetic Algorithms.

Section 1.1.4 briefly describes a genetic algorithm. Appendix A provides a more detailed description and Goldberg (22) provides a complete description of a simple genetic algorithm.

Enhancements have been proposed to Holland's genetic algorithm, such as Goldberg's own Messy Genetic Algorithm (25) and genetic operators to deal with combinatorial problems (42) (22) (31) (49) (10) (26) . Most of these approaches deal with the ordering aspect of combinatorial problem solutions. Davis (10) describes the problem of using traditional crossover operators with combinatorial problems. They tend to produce illegal solutions. For example, let A B C D and A C D B represent two solutions to a four city Traveling

Salesman Problem. Let an operator perform crossover at a point between the second and third cities (B&C for the first encoding, C&D for the second encoding). The result of the crossover is A B D B and A C C D. Both of these are invalid because one city is visited twice while one city isn't visited at all.

Starkweather provided a comparison of six different combinatorial operators (42) on a traveling salesman problem and a scheduling problem. The six operators are Edge Recombination (49), Order Crossover 1 (10), Order Crossover 2 (10), Partially Mapped Crossover (PMX) (26), Cycle Crossover, and Position Based Crossover. All operators can create two children (child 1, child 2) from two parents (parent 1, parent 2). Except for Edge Recombination, all operators start child 1 child by copying one or more genes from parent 1. Then, the operators complete child 1 by filling in the missing genes of the child based on the other parent (or both parents). Creating child 2 follows a similar process with parent 2 taking the place of parent 1. The determination of which genes to copy from the parent and the method of completing the child distinguishes the five operators. The sixth operator, Edge Recombination, pools both parent's genes and then creates children from the pool without regard to which parent contributed the gene. Starkweather showed that no single operator worked best (or worst) for both the traveling salesman problem and the scheduling problem.

As mentioned in Section 1.1.2, the mission routing problem is a combinatorial problem. Although the ordering of intermediate route points is important, this is not at the heart of the mission routing problem. Unlike the travelling salesman problem, a shortest path problem doesn't require all nodes in a graph to be visited. The heart of the problem lies in the selection of the proper subset of intermediate points available. Although the order of points in the subset is important, determination of the subset itself is more important.

One operator that may have application with the mission routing problem has been developed by Davidor (8). Davidor proposes an analogous crossover operator for use with robotic arm movement. The problem is to find the best path for a robotic arm to take between source and destination locations. Although random paths are easily generated,

Davidor finds that crossover between two paths only makes sense if the cross point occurs at an intersection of the two paths. Section 3.2.5 explains this in more detail.

2.3 Vehicle Routing.

Thangiah (44, 45, 46, 48) and Blanton (32) have used genetic algorithms to solve a Vehicle Routing Problem with Time Windows (VRPTW). Results show genetic algorithms work faster and produce equal, if not better, results than other types of algorithms. VRPTW involves a fleet of vehicles, a set of stops to make, and constraints of how much each vehicle can hold and when each stop must be made. School bus routing, mail delivery, and airline scheduling are examples of VRPTW. These problems involve assigning each vehicle in a centrally based fleet to a set of delivery/pickup locations. This can be be mapped to a mission routing problem of a single staging base with multiple aircraft, multiple targets (more than one target per aircraft), time constraints on targets, and weapon capacity of an aircraft.

Thangiah (44) (46) uses a sectoring method to create routes. Think of the location of the vehicle fleet at the center of a circle. All vehicle stops are located within the area of the circle. The circle is divided into sectors bounded by two lines from the circle's center to the circle's boundary. The number of genes in chromosome is equal to the number of vehicles. Each allele corresponds to a unique sector. No sectors overlap and the circle's entire area must be covered. Each chromosome is evaluated based on minimizing distance for all vehicles. This evaluation itself equates to a number of traveling salesman problems, where each vehicle must determine its best route for its assigned cities.

Thangiah(47) also researches a problem more similar to Grimm's (29) involving a single aircraft, a single target, and a hostile environment. Thangiah's problem environment is mapped to a 2-dimensional mesh. The encoding indicates movement in the mesh in one of eight directions (north, northeast, east, ...). Since eight is a power of two, this encoding maps nicely to a solution string where each gene is defined by three bits. The number of genes in a given solution correspond to the number of intermediate points in the route. The problem with this encoding is that it usually doesn't end at the target. This is accounted for by incorporating a heuristic in the fitness function which rates the route not only on

distance and threat exposure, but also by how close it comes to the target. The mapping of this 2-dimensional representation to 3 dimensions isn't clean (3 dimensions require 26 directions instead of 8), but it could be done. This type of encoding deserves further analysis.

Alliot (1) uses a genetic algorithm for an air traffic control problem. Given an airspace, aircraft enter at one point and desire to exit at another point. The goal is to route the aircraft in such a way that actual exit point is closest to the desired exit point. The constraint is to prevent aircraft collisions. Although the evaluation of the constraint is different than Thangiah's (avoid collision versus avoid detections), the implementation is similar to Thangiah's except that relative directions are used (keep going straight, turn left 45°, turn right 90°, ...).

Neither Alliot's nor Thangiah's approach to combinatorial problems includes specialized combinatorial operators. They encode solutions such that traditional operators can be used. Each gene represents the direction of an aircraft for a fixed distance. Since routes are defined as a series of directions (as opposed to coordinates), traditional operators don't create invalid solutions.

Cohoon (7) and Davidor (9) show that multiple genetic algorithms running in parallel produce better results than a single genetic algorithm. The rationale behind this performance is niching, which is described in Appendix B. If migration of solutions between genetic algorithms is controlled, this promotes diversity and helps prevent premature convergence. Deb (12) describes this improvement by using niching. Niching theory is explained in detail in Appendix B. Multiple, diverse populations are better than one homogeneous population because the diversity allows the entire search space to be more thoroughly searched. Because more of the problem space is searched and some communication does occur among sub-populations, all sub-populations benefit.

2.4 Summary Discussion.

Robustness is a quality associated with genetic algorithms (22). Robustness is the ability to work well with a wide variety of problems, in exchange for not performing

optimally for any one problem. The control structure of genetic algorithms shouldn't have to be tailored to specific problems. Although Goldberg has proposed specialized operators (25) (26), he has also suggested that genetic algorithms shouldn't be fine-tuned too much (23). Goldberg believes many attempts to improve on genetic algorithms lose sight of robustness. Combinatorial operators illustrate this point. The operators discussed require modification of the basic crossover principle. Different types of problems require different types of operators. Ideally, a single operator should be used for all. As discussed previously, this won't work. Simple operators on some encodings of combinatorial problems often produce invalid results. More emphasis should be placed on the encoding of solutions which allow conventional operators to be used. This emphasis on encoding solutions (routes) provides the heart of the remainder of this thesis.

III. Analysis and Design Methodology

This chapter presents the analysis and design process of applying genetic algorithms to a mission routing problem. Section 3.1 analyzes the mission routing problem, while section 3.2 discusses genetic algorithm design in relation to the mission routing problem. Section 3.3 explains design decisions made in applying a genetic algorithm to mission routing. Section 3.4 proposes a simple niching scheme to use with parallel processing.

3.1 Mission Routing Problem.

This section presents the mission routing problem used for purposes of this research. This section starts with a general, "English" description, progresses to a mathematical model of the problem, and finally shows how the mathematical models are used. Specifically, section 3.1.1 provides an overview of the specific mission routing problem. Section 3.1.2 discusses the three models required (terrain, radar environment, and route). Section 3.1.3 defines evaluation of the route model with respect to the radar environment model.

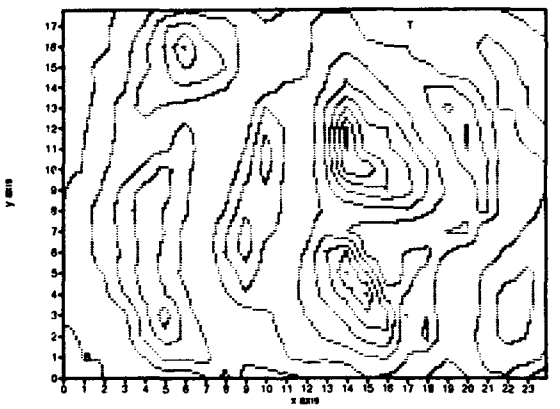


Figure 1. Problem Terrain

3.1.1 Description. For purposes of this investigation, the mission routing problem consists of determining the best route for an aircraft to take between its staging base and a single target. Criteria for determining the best route are minimizing distance traveled and avoiding radar detection. This is the problem researched by Grimm and Droddy (29) (17).

The air tasking order (ATO) (29) defines the staging base and target locations. It also sets restrictions on minimum altitude above ground, minimum altitude, and maximum altitude.

Figure 1 depicts the terrain developed by Grimm (29). The area represents a 25,000 by 19,000 square foot area. Each contour line represents a change of 1000 feet in elevation. Figure 2 depicts the radar environment developed by Grimm (29). The area inside each

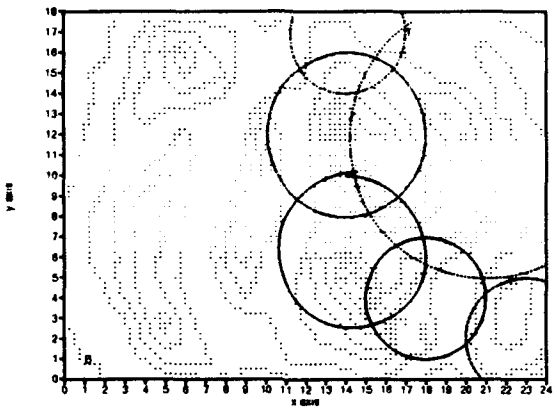


Figure 2. Problem Terrain with Radar Coverage

circle represents a detection zone associated with a radar at the center of the circle. The more powerful the radar, the larger its radius of detection. The circle represents a detection probability of 0.0 while the center of the circle represents a detection probability of 1.0. The detection probability of the area in the circle increases linearly from the circle to its center.

Figure 2 depicts a 2-dimensional area of influence, but the actual radar environment is 3-dimensional. Since radar detection requires line-of-sight, an aircraft may be within a radar's detection distance, but still avoid detection by "hiding" behind hills and mountains which prevent line-of-sight.

A route starts at its staging base (B) and end at the target (T). The search space in constrained by six limits: north, south, east, west, maximum altitude, and minimum altitude. The terrain area provides four limits (east, west, north, and south). The ATO provides the maximum altitude. Lastly, the minimum altitude is provided by a combination of the terrain elevation, ATO minimum altitude, and ATO minimum altitude above ground.

The route is evaluated on its distance and exposure to radar. The distance is just the total distance the route traverses between the staging base and target. Radar cost is determined by dividing the route into segments, calculating the radar detection probability for each segment, and summing the radar detection probabilities over all segments. This doesn't provide an overall radar detection probability, but it does provide a metric base on total exposure to radar.

3.1.2 Representation. The mission routing problem requires three models: one for the terrain, one for the radar, and one for the route. All models are described for completeness; however, most of the development centers around the route model. All models require a three dimensional grid system. For ease of illustration, the models are described in two dimensions (the same dimensionality of paper!).

Although all three models are required, the route model most affects the ability of the genetic algorithm to perform. The terrain and radar models are used for evaluation purposes, hence important for the fitness function. However, it is the route which is evaluated and genetically manipulated – the key operation of a genetic algorithm.

All models require a grid reference system (grid is used as both as a 2-dimensional square and a 3-dimensional cube – the meaning should be clear by the context). A square grid coordinate system is used for simplicity of explanation and implementation. Other coordinate systems (hex, irregular grids) may be more appropriate for modeling (especially radar and terrain); however, they are beyond the scope of this thesis. This thesis focuses on encoding grid coordinates in a manner independent of the type of grid used.

Terrain. The terrain is a surface in space. Although it can represent items such as ground cover (forest, water, desert), this effort uses it solely for elevation information. A surface is composed of an infinite number of points in space. The terrain is approximated with a set of points. Figure 3 shows the terrain overlaid with a 25 x 19 grid. For each grid point $((x, y), 1 \leq x \leq 25, 1 \leq y \leq 19)$, a maximum elevation is associated. This creates a set of points (x,y,f(x,y)) which define the terrain.

Let,

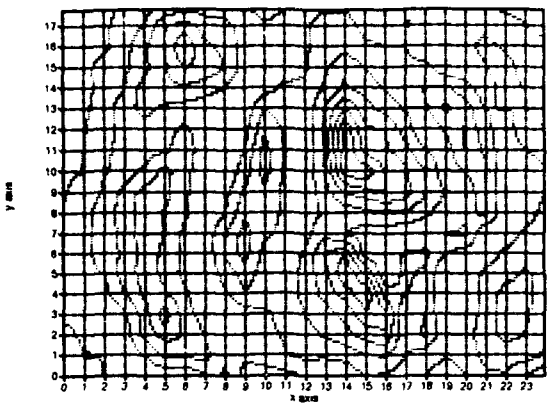


Figure 3. Problem Terrain Decomposed into a Mesh

x: represent the x coordinate

y: represent the y coordinate

then,

$f(x, y)$: represent z coordinate

Radar. The radar environment is a volume in space. A volume is composed of an infinite number of points. Associated with each point is a radar cost. This cost is based on the ability of all radars to detect an aircraft at that point.

For any single radar, its detection ability is based on its strength, distance of the point from the radar, and an un-obstructed line of sight from the radar to the point. Aircraft orientation with respect to the radar also is a factor, but is not considered for reasons of simplicity. This is an evaluation problem, not an encoding problem. Let,

(x, y, z): represent a point in space

rc: represent radar cost at (x,y,z)

$f_{r_i}(x, y, z)$: represent a the detection probability function for radar i at (x, y, z)

then,

$rc = \sum_i f_{r_i}(x, y, z)$

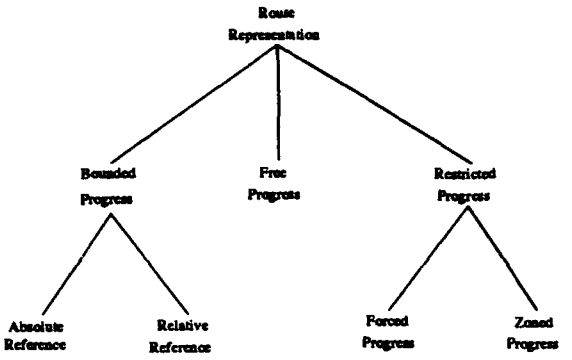


Figure 4. Route Representation Taxonomy

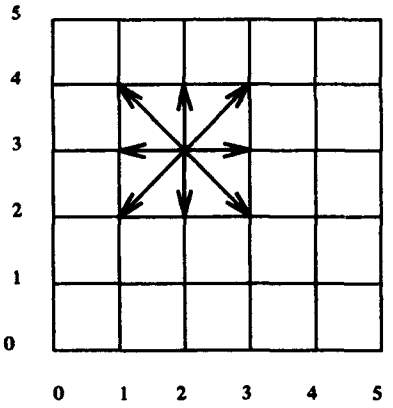


Figure 5. 2-Dimensional Bounds

Route. The route is a continuous curve in space. It starts at the staging base and ends at the target (actually, an altitude above the target). A curve is composed of an infinite number of points. It is approximated with a finite ordered set of points. The first point represents the staging base. The last point represents the target. The route is approximated with line segments between consecutive points in the set.

Figure 4 illustrates a taxonomy of route representation models discussed in this section. The top three models differ on how far away consecutive checkpoints can be. The bounded progress requires consecutive checkpoints to be adjacent to each other. A free progress places no bounds on consecutive checkpoints. A restricted progress is a loosely bound, hybrid approach of the previous two approaches.

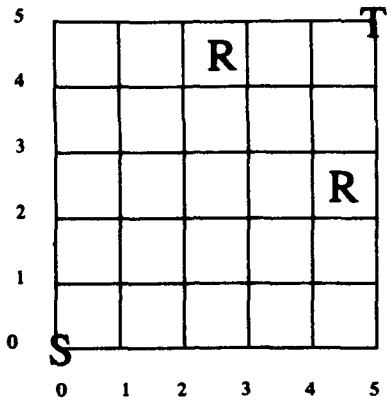


Figure 6. 2-Dimensional Grid

1. Bounded Progress. Representation of a route requires that two consecutive points map to adjacent grid points in space. Given a point on a 2-dimensional surface, the next point must be one of only eight grids immediately surrounding it (see figure 5). Similarly, for 3-dimensional space, the next point can be only one of 26 other points. The number of points which constitute a valid route is variable. For example, reference figure 7. Route 1 takes 5 points while route 2 takes 10 points. The bounded

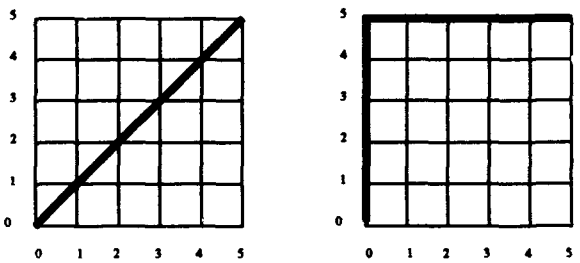


Figure 7. Bounded Progress Paths of Different Lengths

progress system lends itself to two reference systems, absolute and relative.

(a) Absolute. Any point on figure 6 can be referenced by an ordered pair: the first element represents the direction along one dimension, the second element represents the direction along the other dimension. Add another element, to create an ordered triple, and the any triple can represent any grid. In order for the set to constitute a valid route, each pair of consecutive points must correspond to adjacent points in space.

(b) Relative. This generalization is based on methods used by Thangiah (47) and Alliot (1). Any grid on figure 6 can be referenced with a single element and its previous location. The element represents which direction to go to next. It takes on one of eight values (the values could correspond to direction headings - N, NW, W, SW, S, SE, E, NE). For 3-dimensions, 26 values (directions) are required. Unlike the absolute reference, there is no danger of representing non-consecutive points on the grid. However, there is danger that the route could extend past the boundaries of the grid - a situation which the absolute reference representation doesn't allow.

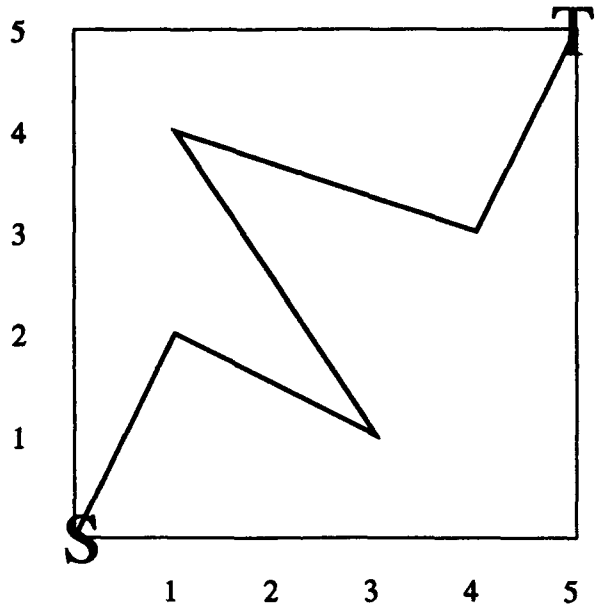


Figure 8. Free Space

2. Free Progress. Figure 8 is a 2-dimensional terrain. Unlike the Bounded Progress approach, construction of a route doesn't impose any restrictions (except that the route must be within the boundaries of the defined space). Two consecutive points can appear anywhere above (not below) the terrain. Whereas the distance between points in the Bounded Progress routes are fairly constant $(1.0, \sqrt{2}, \sqrt{3})$, the distance between points in the free system may not be constant. Unlike the grid approach, any route can be specified by a constant number of checkpoints. No attempt is made to break down the free progress model into absolute and relative representations

(as done for the Bounded Progress model). Nothing new would be added except complexity.

3. Restricted Progress. This approach combines a desire for checkpoints to be close to each other (Bounded Progress) with the ability to represent a route with a fixed number of checkpoints (Free Progress). Generally, the checkpoints near the beginning of the route should be close to the staging base, and the checkpoints towards the end of the route should be close to the target.

   (a) Forced Progress. This method forces progress towards the target along either the x or y coordinate axis. The route is specified by an ordered set of pairs (x,z) or (y,z) – the first element of the pair represents the axis *not* chosen for forced progress. The nth pair in the set refers to a point n units along the forced axis.

   (b) Zoned Progress. This method creates zones of selection from which a checkpoint may fall. The first zone represents a zone immediately around the staging base. The second zone represents a zone around the staging base larger than the first. The middle zones span the entire terrain area. The later zones then narrow their focus towards the target. See figure 9 for a six zone example.

   The three grids on the upper portion of figure 9 represent the expanding zones. The three zones show a progression from a zone immediately around the staging base $(0,0)$ to the entire grid. The lower three grids of figure 9 represent the contraction of zones from the entire grid to the target $(5,5)$. The upper left grid presents a small zone in which the first checkpoint may occur. The upper center grid presents an expanded zone which includes the first zone. The second checkpoint can occur anywhere within this zone. The upper right zone represents the full expansion of the search space around the staging base. The three bottom grids in figure 9 represent then the contraction of the zones from the entire grid to the area immediately around the target. The lower left grid represents the entire search space around the target (this also happens to be the same as th upper right zone). The bottom middle grid shows a zone col-

lapsing towards the target. The bottom right represents the area immediately surrounding the target.

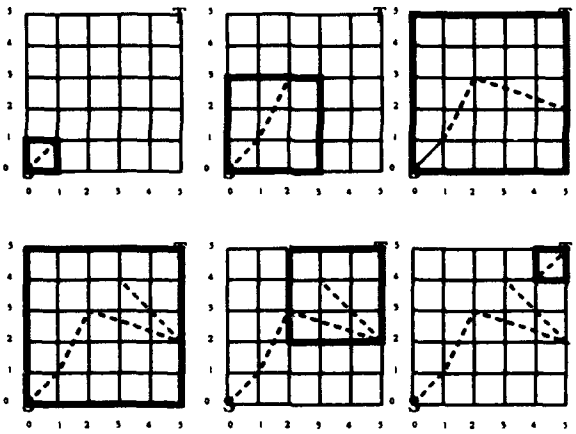


Figure 9. Zones of Checkpoint Selection

3.1.3 Evaluation. As mentioned earlier, the route is to be optimized with respect to minimum distance and minimum radar detection. Let

n = *number of checkpoints over route*

d_i = distance of between checkpoint $i-1$ and i

rp_i = radar detection probability between checkpoint i and $i-1$

w_d = weight of distance criteria

w_r = weight of radar detection criteria

Constraints include

$w_d + w_r = 1.0$

$0 \leq i \leq n$

checkpoint 0 is the staging base

checkpoint i *is the target*

$$\text{evaluation} = w_d * \sum_{i=0}^{i=n} d_i + w_r * \sum_{i=1}^{i=n} d_i * rp_i \tag{1}$$

which is equivalent to

$$\text{evaluation} = \sum_{i=1}^{i=n} d_i * (w_d + (w_r * rp_i)) \tag{2}$$

Distance. The first term in equation 1 represents the total distance of the route. The route is represented as an ordered set of points. The distance of the entire route is just the sum of all the distances between consecutive checkpoints. The importance of the distance with respect to all other criteria is specified by w_d.

Radar Detection. The second term in equation 1 represents the total radar cost. This is the sum of the radar costs associated with each route interval. The cost of each route interval ($d_i * rp_i$) is just the distance of the interval multiplied by the radar detection probability for that interval. This is trivial if only one detection probability applies to the interval. Otherwise, the radar cost requires more thought. If an interval spans more than one radar detection probability, each radar probability must be summed in proportion to the relative amount of distance the route traverses in the radar's probability zone. The importance of radar with respect to all other criteria is specified by w_r.

3.2 Genetic Algorithm.

This section describes the implications of the models discussed previously as used with a genetic algorithm. An "off the shelf" genetic algorithm should require only encoding and evaluation by the user. For this reason, most of the discussion centers around the encoding and evaluation of a genetic algorithm. However, some encoding schemes do impact the use of the other operators; so, they are discussed for completeness.

3.2.1 Encoding. This section discusses encoding the route representation models in section 3.1.2 as genetic strings. A route is a sequence of checkpoints. Each checkpoint is represented by three dimensions: x, y, and z. The details of encoding binary digits to route points are covered under detailed design. For this section, assume x, y, and z refer to decoded coordinates. For a single mission routing problem, the staging base and target are fixed; so, they're not explicitly part of the route.

Bounded Progress. For both bounded progress representations, each gene in a chromosome corresponds to a route point. Consecutive genes in a chromosome map to adjacent points on a 3-dimensional grid representing air space. Since different routes may have different numbers of checkpoints, each chromosome must be able to have a a variable number of genes. Either a genetic algorithm should be able to handle varying length chromosomes or a chromosome size should be declared which is large enough to accommodate the worst-case longest path.

The range of values of the absolute reference genes depends on the size of the air space. Since this range may vary with the problem, the gene should be encoded with a binary string of a length which depends on the range of the coordinate being encoded.

Relative reference genes are limited to 26 values representing relative direction. These genes may represent one of these 26 values with either a binary encoding (not convenient, since 26 is not a power of two) or a 26 character alphabet.

Free Progress. This is the same as the Bounded Progress approach using absolute reference. The difference is that the Free Progress allows, but doesn't require each solution to contain the same number of checkpoints; so ,the number of FreeRoutePoints can be constant.

Restricted Progress. The Restricted Progress representations are similar to the free progress representations in that they allow each chromosome to contain the same number of checkpoints. The Restricted Progress representation is more restrictive than the Free Progress representation because the checkpoints represented are constrained. The method of restriction differs between the zoned progress and the forced progress representation.

1. Zoned Progress. This is similar to the Free Progress and Absolute Reference representations in that each gene is explicitly mapped to an (x, y, z) coordinate. It is also like the Free Progress representation in that a fixed number of genes (checkpoints) make up the chromosome. It is unlike these representations in that each gene is restricted in the value it may take. Furthermore, this restriction varies with the gene's position in the chromosome (its loci). The genes at both ends of the chromosome are constrained most. They correspond to areas immediately around the route's starting

and ending locations. The genes in the middle are unconstrained. They may take on values anywhere in the search space.

2. Forced Progress. Like other representations discussed, each gene corresponds to a checkpoint. Unlike the other representations, each gene and its loci represent the coordinate of a checkpoint. While the other representations (except Relative Representation) explicitly encode the three coordinate values in a gene, the Forced Progress only encodes two of the three coordinate values in a gene. The third coordinate value is implicitly encoded as the gene's loci. The gene's loci corresponds to the checkpoint's distance from the starting point along one of the coordinate axis. This means that the number of genes in a Forced Progress Representation is equal to the distance between the staging base and target along one coordinate axis.

3.2.2 Evaluation. The evaluation was discussed in the analysis of the mission routing problem. This section discusses how each genetic encoding is evaluated with respect to the evaluation function established in section 3.1.3. The evaluation function measures the distance of each route and the radar detection probability.

None of the encodings are evaluated directly. The evaluation function is based on a Absolute Progress Route. Each encoding goes through a number of processes to convert the particular encoding to an Absolute Progress route. Encodings are made up of chromosomes of genes, while the Absolute Progress route is a sequence of 3-dimensional checkpoints.

In this section, encoding refers to the chromosome made up of genes and a genetic alphabet (usually 1's and 0's), while route refers to a sequence of 3-dimensional points (checkpoints) along the route.

Bounded Progress. The Absolute Representation encoding requires only one step to be converted to an Absolute Representation route. Each gene in a chromosome is converted to a 3-dimensional coordinate. The Absolute Progress Route is created by converting each gene in the chromosome.

The Relative Representation encoding requires that each gene be converted to a direction. After a sequence of directions is extracted from the encoding, this sequence

is converted to an Absolute Representation by calculating coordinate values using the coordinate of the staging base and the relative direction to determine absolute coordinates.

Free Progress. The Free Progress encoding requires that a sequence of three dimensional coordinates be extracted from chromosomes to create a Free Progress route. For each consecutive coordinate in the Free Progress route, new coordinates are inserted so that this new list creates an Absolute Route representation. This can be viewed as interpolating between consecutive Free Progress checkpoints.

Restricted Progress. The Zoned Representation encoding requires that a sequence of three dimensional coordinates be extracted from the chromosome to create a Zoned Progress route. This route is then processed the same as the Free Progress route.

3.2.3 Initialization. This section discusses the impact each type of encoding has on the initialization phase of a genetic algorithm. The Bounded Progress requires use of specialized initialization operators, while the Free Progress and Restricted Progress encodings allow use of the standard initialization operator (random).

Bounded Progress. Because the values of the genes in Bounded Progress encodings represent values which are dependent on the values of adjacent genes, the Bounded Progress encodings are best implemented using a specialized initialization operator.

1. Absolute Reference. Given a random generation of checkpoints, it is unlikely that consecutive points are located adjacently on the 3-dimensional mesh [1] ; therefore, a specialized operator is needed to generate adjacent points. From any point, one of 26 adjacent points is chosen. Although the choice of 26 points offers a randomness desired in genetic algorithms, the algorithm must at the same time guide the route toward the target. Without a guiding algorithm, it is also unlikely that a random route stops at its target. Yet, this guiding algorithm also reduces the randomness desired.

[1] Since any interior point on a 3-dimensional mesh has only 26 adjacent points, the probability of a point chosen at random is $\frac{26}{n}$. For the simple problem used in this thesis, $n \approx 5000$ which generates a probability of less than 1%.

2. Relative Reference. Unlike the Absolute Reference, a random initialization produces a route of consecutive points. Like the Absolute Reference, it is unlikely the route ends at the target unless a guiding algorithm is used. It is also likely that a random generation of directions could produce a route which goes outside the limits. A penalty function could be used to account for routes which go outside the boundaries and/or don't end at the target. A specialized initialization operator could also be used.

Free Progress, Restricted Progress (Zoned and Forced). A standard initialization operator may be used. The encodings allow for all genes to take on the range of values associated with each coordinate axis. This occurs because the value of each gene can be considered independent of all others. The genes are dependent only in that their order implies the sequence of the route.

3.2.4 Selection. The encoding approaches don't impact the selection operator.

3.2.5 Crossover . This section discusses the impact each type of encoding has on the crossover operator of a genetic algorithm. The Bounded Progress requires use of specialized crossover operators, while the Free Progress and Restricted Progress encodings allow use of the standard crossover operators.

Bounded Progress. Because of the independence between genes, as describe earlier for initialization, the Bounded Progress encodings require use of specialized crossover operators.

1. Absolute Reference. A specialized operator is required since a crossover may produce a route with non-adjacent checkpoints – see figure 10. Two types of operators are possible. One operator would only allow crossover at points where two routes intersect, as in figure 11. This conflicts with the standard procedure of choosing a random crosspoint, but Davidor (8) has proposed a similar operator. The other operator would "correct" the route, by taking the "broken" route and inserting checkpoints to

create a valid route – see figure 12. This has the impact of introducing new genetic material. Again, this is not consistent with the standard genetic algorithm [2]

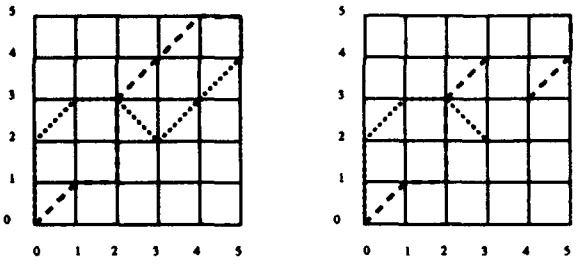


Figure 10. Absolute Reference: Invalid Crossover

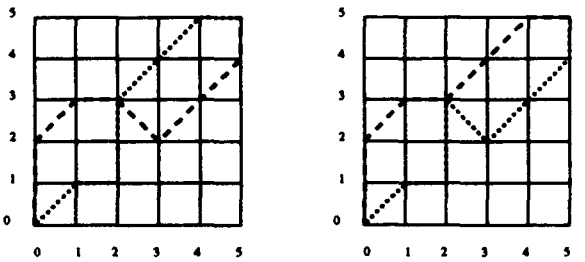


Figure 11. Absolute Reference: Non-random Crossover Point

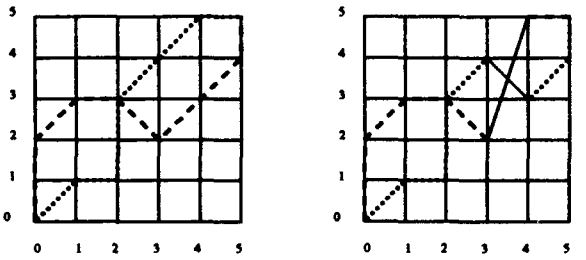


Figure 12. Absolute Reference: Repaired Children

2. Relative Reference. Although a standard crossover operator can be utilized, a specialized operator similar to figure 11 would preserve genetic information. If the standard crossover operator is used, it produces a route of adjacent points. Unless it occurs at an intersection, the child routes won't end at the target and may even go out of bounds – see figure 13.

[2]The standard genetic algorithm introduces new material in only one of two ways: through random initialization and mutation. Crossover only takes existing genetic material and re-arranges it.

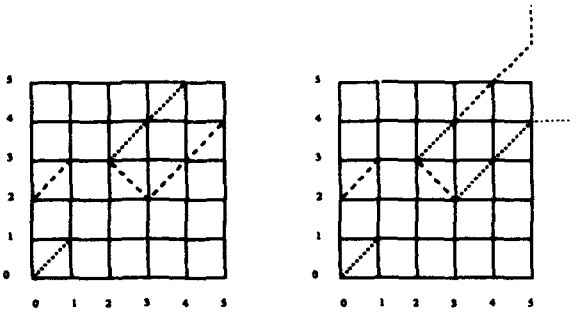


Figure 13. Relative Crossover

Free Progress and Restricted Progress (Zoned and Forced). The standard crossover operator may be used. The impact of the standard crossover operator on the Free Progress encoding is similar to the repairing of a "broken" route described under Bounded Progress, Absolute Reference. The difference is that the repair is implicitly done since consecutive checkpoints aren't required to be adjacent – see figure 14.

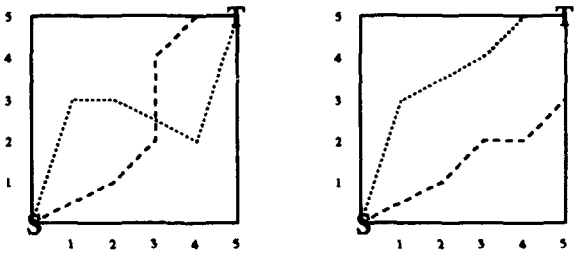


Figure 14. Free Progress Crossover

3.2.6 Mutation. The Bounded Progress encodings require a specialized mutation operator, while the Free Progress and Restricted Progress encodings don't require specialized mutation operators. As with the initialization and crossover operators, this is based on the fact that Bounded Progress encodings have dependent genes, while Free Progress and Restricted Progress encodings have independent genes. The reasoning is similar to that discussed for the initialization and crossover.

3.3 Design Decisions.

This section describes design decisions made on choosing an encoding for the mission routing problem and evaluating the encoding. Two of the design goals are to use the

standard genetic algorithm (no specialized operators) and to provide compatibility with other approaches to mission routing at AFIT. These design goals lead to the choice of Free Progress and Forced Progress as encoding/decoding schemes.

3.3.1 Bounded Progress. The Bounded Progress representations both offer an easy way to understand encoding; however there exist problems associated with the use of a standard genetic algorithm on these encodings. As discussed in section 3.2, specialized initialization, crossover, and mutation operators are required; so Bounded Progress is not used as an encoding scheme.

3.3.2 Free Progress. The Free Progress representations are compatible with the standard genetic algorithms. This encoding scheme requires no specialized genetic operators. However this approach is not directly compatible with the A* approach to mission routing at AFIT (17) (29). This is because the current A* approach uses an evaluation model based on a Bounded Progress representation. A further step of decoding would turn a Free Progress representation into a Bounded Progress representation by using a heuristic to connect consecutive Free Progress route points with intermediate Bounded Progress route points. The net impact is that the initialization, selection, crossover, and mutation operators manipulate a Free Progress encoding, while the evaluation operator is based on a Bounded Progress representation.

3.3.3 Restricted Progress. Like the Free Progress encoding, the Restricted Progress encoding schemes requires no specialized operators, but isn't directly compatible with the current A* approach. Both Restricted Progress schemes provide an opportunity for shorter encodings.

1. Forced Progress. In the Forced Progress encoding, one dimension of a coordinate is not explicitly encoded. This makes the chromosome small which may allow the genetic algorithm to find a good solution with a smaller population and fewer generations. The downside is that required forward progress along one axis prevents backtracking along that axis, which actually reduces the scope of the problem. Al-

though reducing the scope of the problem may be desirable, this encoding method forces it even when it is not desired by the user.

2. Zoned Progress. Unlike the Forced Progress model, the Zoned Progress representation allows for backtracking within a zone. It has the impact of guiding a route from a base to a target while allowing some randomness. It also requires substantial development of an encoding/decoding algorithm development. This development requires determining how many zones to use, how large to make each zone, and how to map each zone to a bit representation within the airspace environment. Because of the difficulty of the algorithm development, this encoding scheme is not pursued.

3.4 Niching

Niching can be allowed to occur naturally by using subpopulations easily implemented with an island model genetic algorithm (11). It can also be forced by purposely slanting the population toward certain solutions. This research attempts a trivial niche forcing scheme. The evaluation of a route depends on its distance and its exposure to radar. The relative importance of these two criteria depends on the problem. The niching scheme assigns each sub-population a different criteria weighting so that each sub-population is actually working on a different problem. The niching theory says that although each subpopulations focuses on a slightly different problem, they still interact by sharing solutions.

3.5 Summary

Two encoding schemes were selected for implementation: Forced Progress and Free Progress. For the remainder of this thesis, Restricted Progress encoding is equated with Forced Progress encoding (just a terminology issue). Factors governing the selection of these encodings include compatibility with existing software and minimization of new software.

Current software structures (17) (29) assume a bounded route representation. Although Forced Progress and Free Progress representations are the encodings genetically manipulated, they are converted into the Bounded Progress representation to simplify the

evaluation process. Implementing the Bounded Progress or Zoned Progress representations require significant software development; so, they weren't used. The Bounded Progress required modification to the genetic algorithm itself, while the selected encodings allow use of the existing genetic algorithm. The Zoned Progress representation doesn't require genetic algorithm modification, but would require a sophisticated decoding routine.

IV. Low-Level Design.

This chapter presents the low level design of six models required for this thesis. Four of six models, Air Tasking Order, Terrain, Evaluation, and Radar, adapt Grimm's code (29). Two models, Free Progress and Restricted Progress involve two route representations and are original designs. For each model, this chapter describes the requirements, data structure, and control structure. All these models are used in the evaluation step of a genetic algorithm. The Evaluation model implements the evaluation requirement of the genetic algorithm. Since no other modifications are made to the rest of the genetic algorithm (GENESIS software), no low level design is presented for the rest of the genetic algorithm.

The C programming language is used for implementation, since GENESIS and Grimm's software are written in C. Although an object oriented programming (OOP) language is not used for implementation, the low level design does reflect an object oriented design (OOD). The C code is written in a way to partially mimic OOD by compiling each model separately and requiring all inter-module references to use access functions. Appendix D contains the source listings for all modules discussed in this section.

As described in section 1.1.5, genetic algorithms are easily parallelized. Appendix B describes genetic algorithms in general terms which show the potential for parallelization. Appendix C uses UNITY (5) to formally describe a genetic algorithm. UNITY is a high-level description language which uses first-order predicate logic and temporal logic. It describes algorithms in a way to show an algorithm's inherent parallelism without requiring a direct mapping to a specific architecture. Although it doesn't require mapping to a specific parallel architecture, it does allow for it. A parallel version of GENESIS has been implemented on a hypercube and is available at AFIT (18).

4.1 Air Tasking Order.

The Air Tasking Order (ATO) defines two of the six bounds of the 3-dimensional search space: the maximum altitude and lower altitude.

Requirements. The ATO model must provide the user with a way to input the ATO information and then provide that same information to other modules.

Data. The ATO consists of six attributes:

1. Mission Name
2. Target Location (xyz coordinates)
3. Staging Base Location (xyz coordinates)
4. Minimum Altitude with respect to sea level
5. Minimum Altitude with respect to ground
6. Maximum Altitude with respect to sea level

Control. The ATO control structure is simple. An initialization function reads the ATO information from disk (based on an ATO file name given) and stores the information. Six access functions allow other modules to retrieve any of the six data attributes.

4.2 Terrain.

The Terrain model defines four of the six bounds of the 3-dimensional search space: the north, east, west, and south boundaries. It also assists in defining a fifth bound; the lower altitude bound is partially determined by the terrain elevation.

Requirements. The Terrain model must provide the user with a way to input terrain information and provide that data to other modules.

Data. The Terrain consists of five data items:

1. Maximum x. The x-axis notation is used to represent the east-west direction. A x value of 0 represents the western bound. This makes the Maximum x the number of units east of the western bound.

2. Maximum y. This is similar to Maximum x, except y is used to represent the north/south direction. A y value of 0 represents the southern bound. This makes the Maximum y the number of units north of the southern bound.
3. Terrain. This data item provides the ground elevation of any unit defined within the bounds set by Maximum x and Maximum y. This is implemented with an x by y matrix of elevations.
4. Minimum z. This is the minimum ground elevation of the given terrain.
5. Maximum z. This is the maximum ground elevation of the given terrain.

Control. An initialization routine reads the data from a file and stores it for use by other modules. The Minimum and Maximum z data items are derived as the elevations are read from the file into the matrix. Access functions are provided for each of the data items. Additionally, a Get z Range function is provided. Given an (x, y) coordinate, this function returns the number of units between the elevation at that coordinate and the Maximum Altitude (as defined in the ATO). This provides other modules with the altitude limits (upper and lower) for any given (x, y) coordinate.

4.3 Radar.

The Radar model defines the threat environment with which one criterion of a route is evaluated.

Requirements. The Radar model must provide the user a method to input a radar environment and provide radar detection probabilities for any unit in the given 3-dimensional search space.

Data. The radar detection probabilities are read from a file and stored in an (x, y, z) matrix. The file contains the number of units for x, y, and z.

Control. An initialization function reads the data from a file and stores it for use by other models. The only access function provides a radar detection probability for a given (x, y, z) coordinate.

4.4 Free Progress Route.

The Free Progress Route provides a route structure, which is what is genetically manipulated by the genetic algorithm. Recall that Free Progress Routes differ from Bounded Progress Routes in that the Free Progress Routes don't require adjacent points in the route to correspond to adjacent points in the search space. Instead, the route is interpolated between non-adjacent points. This model provides a method to determine the route structure length, decode the structure, and convert the structure into a route of adjacent route points.

Requirements. The Free Progress Route model must be able to generate a structure of appropriate length. This structure is then used by the genetic algorithm. The Free Progress Route model must be able to accept a structure from the genetic algorithm and generate a valid route. This valid route must start at the staging base, end at the target, and contain intermediate points such that adjacent points in the route correspond to adjacent points in the search space.

Data. The Free Progress Route maintains only one data item. This is the Structure Length, which defines the number of bits of the genetic structure for the problem. Search space size (north/south, east/west, elevation bounds) and the number of checkpoints determine the Structure Length.

Control. The access functions include

1. Initialize Structure. This function determines the number of bits needed to represent the (x, y, z) search space and the number of checkpoints.
2. Get Structure Length. This is an access function which returns the structure length as determined by the Initialize structure function.
3. Decode Structure. This function decodes the genetic structure into a Free Progress Route.
4. Expand Route. This function provides the method to convert a Free Progress Route into a Bounded Progress Route. This is done by interpolating between Free Progress

Route points and generating/inserting points in order to make it a Bounded Progress Route.

4.5 Restricted Progress Route.

The Restricted Progress Route is similar to the Free Progress Route just described. The major difference is that the genetic structures represent only two dimensions of the search space. The third dimension is computed by setting the number of checkpoints equal to the number of units between the staging base and target along one axis.

Requirements. The requirements are the same as the Free Progress model. The only difference is that the structure length and structure interpretations must be based on a Restricted Progress Route representation.

Data. Like the Free Progress Route, the Structure Length is the only data item accessed by other modules.

Control. The access functions include

1. Initialize Structure. This function determines the number of bits needed to represent the yz search space, given that the x search space is restricted to the number of units between the target and staging base.
2. Get Structure Length. Like its counterpart in the Free Progress Route model, this is an access function which returns the structure length as determined in the Initialize Structure function.
3. Decode Structure. This function decodes the genetic structure into a Free Progress Route. This access function differs from its counterpart in the Free Progress Route. Besides interpreting the genetic bits, it also computes the third (x) coordinate.
4. Expand Route. Like its counterpart in the Free Progress Route model, this function converts a Free Progress Route into a Bounded Progress Route. This is done by interpolating between Free Progress Route points and generating/inserting points in order to make it a Bounded Progress Route.

4.6 Evaluation.

The Evaluation provides the interface between the five other models in this chapter and the rest of the genetic algorithm. Given genetic material, the Evaluation returns an evaluation (fitness) of that genetic material.

Requirements. The evaluation must provide a method to evaluate genetic material from the genetic algorithm.

Data. Evaluation maintains no data for external access.

Control. Evaluation provides two functions:

1. Initialize MR (Mission Routing) File. The genetic algorithm must use this function as a part of its initialization process. This function initializes the five other models based on inputs read from a MR file. This file contains the file names for the Terrain, Radar, and ATO. If the Free Progress Route is to be used, the MR file contains the number of checkpoints to be used. Finally, the MR file contains the evaluation weighting criteria.
2. Evaluate. This function accepts genetic material, uses the previously described models to evaluate the genetic material, and returns the fitness of the genetic material.

4.7 Summary.

This chapter has presented six models used for the evaluation step of a genetic algorithm. The Evaluation model provides the single interface to the genetic algorithm. The Evaluation model makes use of four models to determine the fitness of genetic material. This genetic material is mapped into a route via either the Free Progress Route model or the Restricted Progress Route model. The ATO model defines the endpoints of a route and the altitude bounds of the search space. The Terrain model provides the other four bounds on the search space. The Radar model provides the threat environment against which the route is evaluated.

V. Experiments and Evaluation

The experiments fall into two broad categories. The two categories exhibit how well the central problem was addressed. Recall that the central problem is to research various encodings to effectively and efficiently use a genetic algorithm to solve a mission routing problem. One category involves examining the efficiency and effectiveness of the Free Progress/Forced Progress representations on a single processor system. The other category involves examining the efficiency and effectiveness of the Forced Progress representation on a multiprocessor system and comparing the results with a single processor system.

This chapter describes the experiment metrics, data, plan, and results. Specifically, section 5.1 discusses how the experimental results are measured. Section 5.2 discusses the data against which the experiments are performed. Section 5.3 discusses the different scenarios used for experiments. Section 5.4 displays the results of the experiments.

5.1 Metrics.

Metrics provide the capability to compare experiments. Solution quality, number of trials, and execution time provide the two metrics used for these experiments. Quality measures effectiveness, while the number of trials and execution time measures efficiency. Solution quality refers to the evaluation of the best route found. As discussed in section 3.1.3, the evaluation is based on the route's distance and its total exposure to radar. For time, the number of trials is used for experiments using the Sun Sparcstations. The GENESIS uses trials to describe the number of times new solutions are generated. This is directly proportional to the number of genetic operations performed. This is used instead of time because execution time varies due to uncontrollable factors such as network response time and the number of other processes using the same processor in the experiment.

5.2 Input Data.

The data used for the experiments is the same used by Grimm (29). The same data was used so that the solution qualities found by Grimm could be used as a baseline to

evaluate the solution qualities generated by genetic algorithms. Appendix E contains the input data used.

5.2.1 Terrain. The terrain represents a mountainous area. These mountainous areas are meant to complicate the search process in that routes which "hug" the mountains may hide from some radars. The terrain is bounded by 25 units on the x-axis and 19 units on the y-axis. Associated with each xy unit is a elevation ranging between 1585 and 6000.

5.2.2 Radar. The radar environment represents the threat by six radars. The radar environment is represented by a 25 by 19 by 15 unit matrix of radar detection probabilities. These probabilities are based on the combined influence of all six radars. As in the terrain model, 25 refers to the number of units along the x-axis and 19 to the number of units along the y-axis. The 15 refers to the number of units along the z-axis (altitude). The number of units along the z-axis is in thousands of feet. The radar environment is defined for below ground since the radar model starts at 0 elevation. Although this may seem inefficient, it is necessary so the evaluation algorithm can use terrains which may be at sea level.

5.2.3 Air Tasking Order. Two Air Tasking Orders (ATO) are used. AFIT-1 represents a staging base at (1,1,5) and a target at (17,17,8). AFIT-GO represents a staging base at (1,1,5) and a target at (24,16,8). Both ATOs represent the same staging base in non-hostile territory, but AFIT-GO uses a target further into hostile territory than AFIT-1. Both ATOs require a minimum altitude of 4 units above sea level and 0 units above ground.

5.3 Test Plan.

The test plan consists of two phases. The first phase, using a single processor system, compares the Free Progress and Forced Progress encodings. The second phase studies niching on a parallel processing system using Forced Progress encoding.

5.3.1 Single Processor. Forty-eight single processor experiments were run which represent permutations of the following three categories:

- Problem specified by ATO: AFIT-1 or AFIT-GO
- Use of seeded population: Seeded or Unseeded
- Encoding type: Free Progress or Forced Progress
- Population size: 50, 100, 200, 400, 800, 1600

Each experiment executes for 100,000 trials. The best solution quality is recorded approximately every 100 trials. Although genetic algorithms are probabilistic, GENESIS provides a pseudo random number generator which allows for repeatable experiments (the output is always the same, as long as all inputs are the same). Therefore, multiple test runs are not used since they all would create identical outputs.

Two problems, AFIT-1 and AFIT-GO, respectively represent one easy and one hard problem. One represents a target at the edge of radar coverage (AFIT-1) and the other represents a target deep into radar coverage (AFIT-GO). A greedy technique is used to find a solution with which to seed the initial population. The decision to study seeded populations came from discovering that some unseeded populations did not converge to a value as well as a greedy solution. Population sizes were chosen to represent a wide range of sizes. The decision to use 100,000 trials is based on the fact that the largest population size (1600) converges to values within 10% of one another. The radar weighting criteria was set at 0.8, the same Grimm used. This allows a solution quality comparison with Grimm's results.

Crossover and mutation rates where kept constant at 0.6 and 0.001 respectively. These were chosen only because of their default values in the GENESIS setup routine.

5.3.2 Multiple Processors. Two types of experiments were run on multiple processors. One type varied population size while keeping the epoch rate at five generations. The other type ran a single experiment with a population size of 400, epoch size of five, but varied the weighting criteria from 0 to 1.0 by assigning different subpopulations a unique weighting criteria.

5.4 Results.

5.4.1 Single Processor. Figures 15, 16, 17, and 18 show the best solution found for each of the 48 experiments. The only clear pattern is that the Free Progress Route always performs better (lower is better) than the Forced Progress Route. Grimm's best solutions are 7182 for AFIT-1 and 9664 for AFIT-GO. The best solutions from all genetic algorithm runs are 8371 for AFIT-1 and 9251 for AFIT-GO. The genetic algorithm didn't find a better answer for AFIT-1, but it did for AFIT-GO. Analysis of the results may suggest potential benefits and costs of each encoding. These are explained in the next chapter.

Encoding Type	Population Size	Best Solution Found
Free Progress	1600	10089
Free Progress	800	8510
Free Progress	400	8371
Free Progress	200	9623
Free Progress	100	9312
Free Progress	50	9468
Forced Progress	1600	12018
Forced Progress	800	10494
Forced Progress	400	10622
Forced Progress	200	10049
Forced Progress	100	9594
Forced Progress	50	11598

Figure 15. Data for AFIT-1, Unseeded Population

5.4.2 Multiple Processors. Multiple processor (niching) results (see figure 19 show no benefit of forced niching over natural niching. Although routes are found for four different radar weights, the 0.8 weight did not out perform the results from free niching (10851).

5.5 Summary.

This chapter has presented the raw data of experiments. Metrics were established, test inputs defined, test cases explained, and the results presented. The next chapter hypothesizes some reasons behind the results.

Encoding Type	Population Size	Best Solution Found
Free Progress	1600	9054
Free Progress	800	8593
Free Progress	400	9049
Free Progress	200	8939
Free Progress	100	8921
Free Progress	50	8921
Forced Progress	1600	11843
Forced Progress	800	11557
Forced Progress	400	11413
Forced Progress	200	11438
Forced Progress	100	11471
Forced Progress	50	12043

Figure 16. Data for AFIT-1, Seeded Population

Encoding Type	Population Size	Best Solution Found
Free Progress	1600	14695
Free Progress	800	9729
Free Progress	400	10654
Free Progress	200	10688
Free Progress	100	10332
Free Progress	50	10424
Forced Progress	1600	19311
Forced Progress	800	18421
Forced Progress	400	16346
Forced Progress	200	15254
Forced Progress	100	15618
Forced Progress	50	15350

Figure 17. Data for AFIT-GO, Unseeded Population

Encoding Type	Population Size	Best Solution Found
Free Progress	1600	9809
Free Progress	800	9397
Free Progress	400	9251
Free Progress	200	12287
Free Progress	100	12159
Free Progress	50	12287
Forced Progress	1600	15787
Forced Progress	800	15701
Forced Progress	400	15382
Forced Progress	200	15141
Forced Progress	100	15329
Forced Progress	50	15230

Figure 18. Data for AFIT-GO, Seeded Population

Radar Weight	Best Solution
0.0	33978
0.5	20283
0.8	11347
1.0	4398

Figure 19. Multiple Processor Results

VI. Conclusions and Recommendations.

6.1 Single Processor Conclusions.

The discussions in this section reference four figures. In each figure the staging base is in the lower left and the target is in the upper right. Figure 20 displays Grimm's best solution for AFIT-GO. Figures 21, 23, and 24 are based on a genetic algorithm run on AFIT-GO using a population size of 50 (chosen for ease of data generation of the charts) run for 5000 trials (enough for the population to converge). Although the problems were run in three dimensions and the displays are in two dimensions, the displays should be able to illustrate what is happening with the genetic algorithms.

The Forced Progress Route representation has a certain order to it. Each checkpoint moves closer to the target along one axis. Although this prevents any backtracking along the axis, it is hoped that this backtracking isn't needed. Yet this "forced" progress isn't consistent with the standard genetic algorithm's random initialization.

The Free Progress Route does provide this truly random initial population. At first this might appear undesirable. For example, assume the first route point is near the target; the second near the staging base; the third near the target; the fourth near the staging base. A truly random population allows this. This proposed path certainly is inefficient. But the processes of selection eliminates this poor route. Additionally, just one of the links in this bad route could provide valuable genetic material to another solution during crossover.

Figures 21 and 22 represent all the routes generated during the initialization phase for the Free Progress and Forced Progress routes. Although a small population size is used, it still provides enough genetic material to almost completely cover the search space. Notice that the Forced Progress initialization routes cover less of the search space (particularly at the left and right ends) than the Free Progress search space. This is because the Forced Progress route can only search in the forward x direction from the staging base to the target, and not past the target.

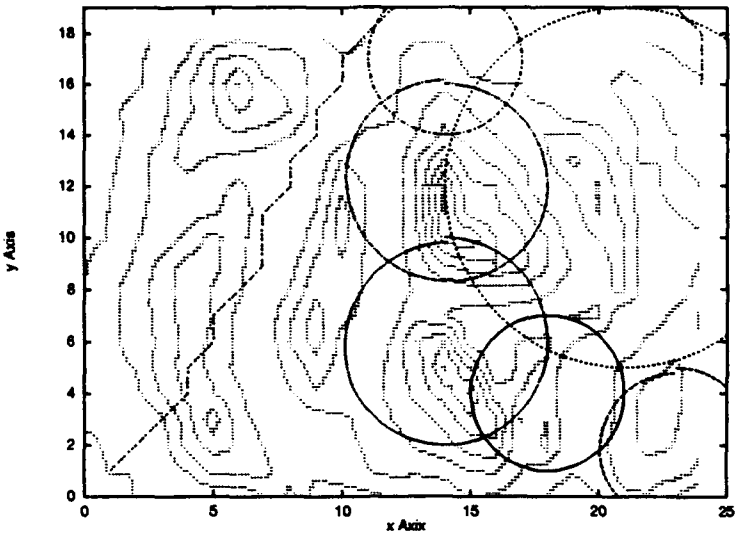


Figure 20. Grimm's Best Solution for AFIT-GO

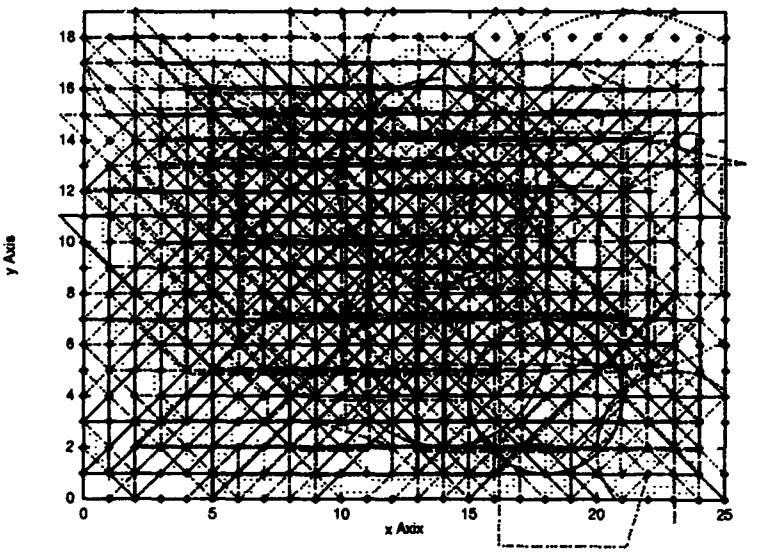


Figure 21. Free Route Initial Population Solutions for AFIT-GO

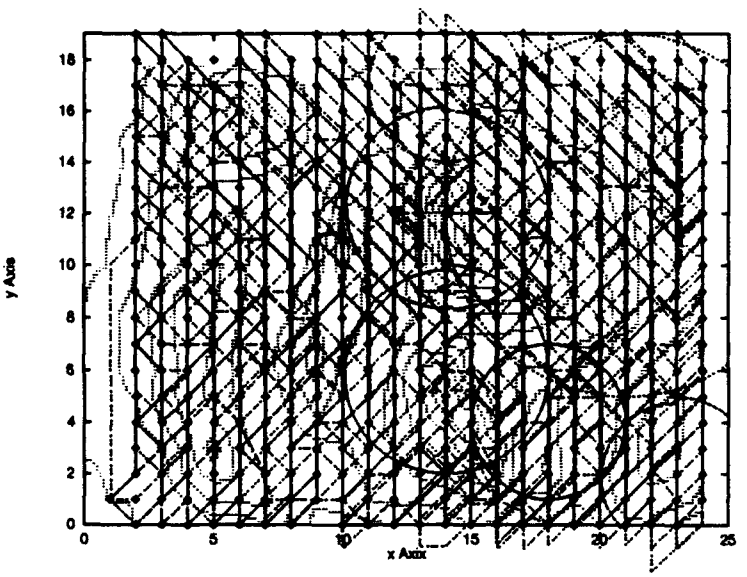


Figure 22. Forced Route Initial Population Solutions for AFIT-GO

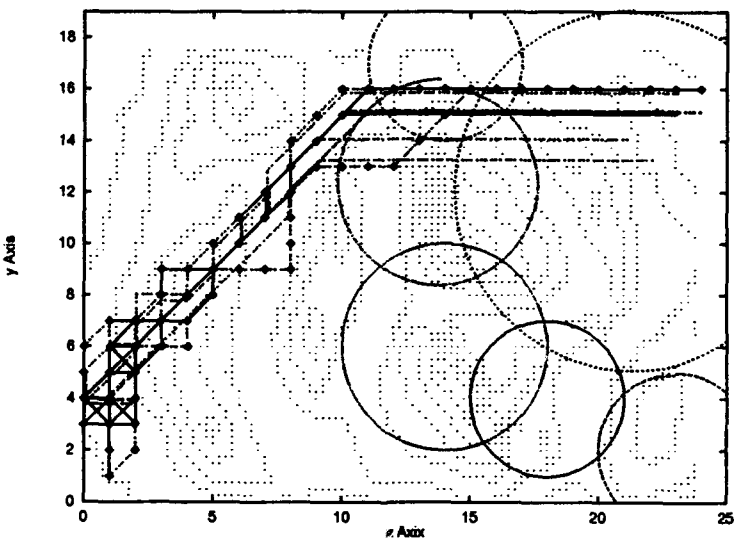


Figure 23. Free Route Final Population Solutions for AFIT-GO

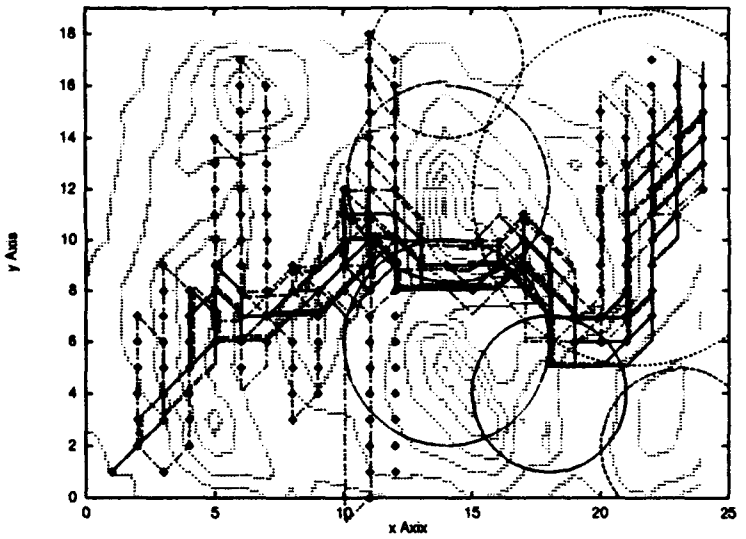


Figure 24. Forced Route Final Population Solutions for AFIT-GO

Figures 23 and 24 represent all the routes in the final populations of the Free Progress and Forced Progress generations. Note that despite the randomness of the Free Progress initialization, it converges to a smoother set of routes than the Forced Progress routes.

6.2 Multiple Processor Conclusions.

As stated in the previous chapter, the forced niching used provides no added benefit. The free niching produced a slightly better solution with the same number of generations and population. This is not to say that forced niching should not be used. The implemented niching scheme is too simple. More research into mission route planning techniques may provide some niching ideas, such as niches which "hug" the terrain. Of course, forced niching isn't required to achieve the benefits of parallel processing. The easily parallelized genetic algorithm achieves an almost linear speedup by just letting natural niches form.

Unrelated to niching, but perhaps a better way to utilize parallel processing would be to decompose the search space and use a genetic algorithm to determine best routes within a search space. Then apply a separator decomposition algorithm (6) (34) to combine routes from appropriate areas of the search space.

6.3 Recommendations.

Three major recommendations are suggested. The first one deals with determining the scalabilty of genetic algorithms with respect to mission routing problems of different sizes. The second recommendation deals with using a more realistic evaluation function. The final one deals with further exploration of encoding methodology.

6.3.1 Genetic Algorithm Scalability. Previous research shows that execution of genetic algorithms scale well with respect to population size and number of processors. The only reason not to use parallel processing architectures is a lack of availability or a small population size. However, the scalability of the mission routing problem itself is left unanswered. Since the relationship of population size and gene length to the number of generations required for convergence is not established with any theory, the only way to determine this is through empirical data. The terrain size, along with the mission (distance between staging base and target), determine the search space size. This requires a collection of many different missions terrains of varying sizes.

6.3.2 Evaluation Function. The evaluation function used is overly simplistic. For this research, this is not a problem because the focus is to develop route encodings. Once acceptable encodings are determined that work on a wide variety of mission routing problems, more realistic evaluation functions should be used. Integration of other evaluation functions into the genetic algorithm should be straight-forward, providing the evaluate routes in a Bounded Progress format.

6.3.3 Encoding Methodologies. The encoding methodologies implemented did find solutions within 20% of Grimm's results. These encodings can probably produce better results if time is taken to fine tune the genetic algorithm parameters. This fine tuning shouldn't be done until the encodings are used on a broader range of problems; otherwise, the fine tuning may be too problem specific. Although these encodings do produce material which a standard genetic algorithm can manipulate, perhaps specialized operators should be used. Specifically, Davidor's analogous crossover operator (8) should be considered.

6.4 Summary.

This chapter presented conclusions on single and multiple processor results and made recommendations for further research. The randomness inherently possible in the Free Progress route encoding suggests that it is better suited to find good routes than the Forced Progress route encoding. This superiority is represented by the better solution quality it consistently achieves, along with a visually smoother set of routes when mapped against the terrain. The use of forced radar weight niches showed no improvement in solution quality over the use of natural niches. Future research in the area of genetic algorithms should first examine the use of specialized operators to use directly with Bounded Route encodings. Once this is done, the three types of encodings (Bounded, Free, and Restricted progress) should be examined using a broad class of mission routing problems (with more realistic evaluation functions) to determine the best type of encoding and to allow fine tuning of the genetic algorithm.

Appendix A. Simple Genetic Algorithms.

This appendix provides an overview of simple genetic algorithms (4).

A.1 General Process

John Holland of the University of Michigan first developed genetic algorithms during the 1960s. He first published *Adaptation in Natural and Artificial Systems* (30) in 1975. David Goldberg, a student of Holland, published *Genetic Algorithms in Search, Optimization, and Machine Learning* in 1989. Most of the material in this section is based on chapter 1 of Goldberg's book.

This section describes genetic algorithms, generally and technically. A simple example illustrates the major concepts.

General Description. Genetic algorithms are computer-based search algorithms which are gaining in popularity for applications in a wide variety of problem domains. The primary distinction between genetic algorithms and other types of computer-based search algorithms is the manner in which the search space of candidate answers is traversed. Traditional search algorithms can be categorized as either calculus-based, enumerative, or random. Calculus-based algorithms seek to either indirectly derive the optimal solution by solving a set of equations, or directly find the maximum of the function by moving in the direction of the highest slope. An enumerative algorithm, which is only suitable for finite domains, simply enumerates or tests every point within the problem domain. A random search technique is similar to enumeration; however, only random points within the domain are tested.

Genetic algorithms differ from traditional search algorithms in that the search strategy employed by genetic algorithms is based on the Darwinian process of natural selection or survival of the fittest. The candidate domain is encoded as a population of string structures representative of chromosomes. Through the application of "genetic operators" to an initial population of string structures, a new generation is formed, called offspring. The strings with the highest fitness values will be kept for breeding the next generation. The basic genetic algorithm consists of at least three operators—reproduction, crossover, and

mutation. Reproduction employs a selection strategy to determine which strings will survive or be copied over into the next generation. The purpose of the crossover operator is to create new strings using pieces of the strings from the previous generation. A typical crossover operator mates a pair of strings by randomly selecting a crossover point along the length of the string and swapping the string sequence from the crossover point to the end of the string. The mutation operator randomly selects a string within the population and alters part of the string. Just as in nature, the mutation operator is applied only periodically. By applying the genetic algorithm operators to an initial population and simulating the process of natural selection over many generations, a final population of the fittest strings is formed.

Technical Description. This section describes the data and control structures of a genetic algorithm. Section A.4 provides pseudo code of the following descriptions.

Data Structure. This section describes terminology associated with the data structure of a genetic algorithm. The main data structure is a single population. This population is composed of individual structures. Each structure is composed of one or more strings (often just one string). A string is composed of features. Each feature (identified by its location in the string) is assigned a feature value. The encoding of feature values in the string determine the value of the string.

In discussing genetic algorithms, genetic terminology is often used interchangeably with genetic algorithm terminology (presented in previous paragraph). A genotype defines the makeup of an organism just as the structure defines a solution. A genotype is composed of multiple chromosomes just as a structure is defined by one or more strings . A gene defines a particular characteristic of an organism (such as eye color) just as a feature defines some portion of a solution. A particular gene in a chromosome is defined by its locus, just as the string position defines the feature in a string. Each gene takes on the value of an allele (such as green for eye color) just as each feature of a string takes on a value (often defined by a binary alphabet — 0 or 1). The phenotype provides a high-level description of the genotype (this genotype is a frog) just as a parameter set provides a

high-level description of a structure (this solution contains two parameters between the value of 0 and 63).

Control Structure. The genetic algorithm is of polynomial order complexity with a finite space requirement determined by the population size. Although Goldberg (24) suggests that there is an optimal population size which depends upon the length of the string, there is no fixed dependence between the length of the string and the population size. Experimental evidence, however, suggests that an insufficient population size may adversely affect solution quality (22)(36). The terms in the order of the genetic algorithm reflect the length of the string as well as the number of strings in the population. An entire cycle of the genetic algorithm is executed up to a maximum number of generations specified by the user. The basic pseudo code for a genetic algorithm is presented below.

```
initialize population
calculate fitness for all members of the population
for i = 1 to max_number_of_generations (m)
  for j = 1 to population_size (n)
    crossover
    evaluate fitness
    mutation
  end loop
  selection
end loop
```

The various genetic operators each have associated maximum complexities although complexity of an actual implementation may differ. The crossover operator selects two strings from the population pool (n), picks a random location along the length of the string, and then swaps the two tails of the parent strings which follow the randomly selected crossover point. This portion can be considered to be $O(l)$ since it needs to traverse the length of the string.

The fitness function includes a call to decode the string representation into the value in the problem domain. This decode call could be an $O(1)$ or an $O(l)$ function, depending upon the string representation scheme and the programming environment capabilities. Evaluating the fitness function should be an $O(1)$ operation since it simply substitutes the decoded string values, and evaluates the objective function. However, for complex

problems, the evaluation of the fitness has a lower bound of the order of the objective function —the functions being optimized.

Mutation also could be an $O(1)$ or an $O(n)$ operation depending upon the particular implementation and also the mutation strategy being used. Studies have shown good results are obtained with a mutation rate of once for every thousand string position transfers (22).

The selection function has an $O(n)$ complexity since it must implicitly or explicitly evaluate each member of the population to determine which strings will be carried on to the next generation. Actual implementations of the selection operator may be of $O(n^2)$ complexity, such as a roulette wheel approach biased according to the fitness of the strings.

This makes the complexity of the entire algorithm

$$O(m * max(n * max(l, \text{fitness function}, n), n^2))$$

which is equal to

$$O(m * n * max(l, \text{fitness function}, n)).$$

Examples. This section illustrates the use of genetic algorithms on two example problems. The first problem illustrates the detailed process of encoding, initialization, selection, and crossover on a single variable function. The second example illustrates encoding of a two-variable function from which the generalization to multi-variable problems can be drawn.

A.1.0.1 Single-variable Function. This section illustrates the operation of the initialization, selection, and crossover operators on a simple maximization problem of

$$f = x^2.$$

In real life, a genetic algorithm would not be used to solve this problem, but it does provide a simple way to explain genetic operators.

Restrict x to integer values between 0 and 31. An obvious encoding would be a string of 5 binary digits. This string corresponds to a binary integer. For example

1 0 0 0 0

represents 16.

The initialization phase just involves randomly assigning 0s and 1s to string positions. For simplicity, assume a small population of 4 strings. Let the initialization phase produce the following strings:

0 1 1 0 1
1 1 0 0 0
0 1 0 0 0
1 0 0 1 1

The above strings have the respective values of 13, 24, 8, 19. The evaluation (fitness=x^2) is then is then 169, 576, 64, and 361. Selection requires the computation of an average fitness which is

$$(169 + 576 + 64 + 361)/4 = 293.$$

For a simple selection algorithm, the number of copies a string receives in a new population is its

$$\text{expected count} = \frac{\text{string fitness}}{\text{average fitness}}.$$

The expected count is rounded to an integer value which gives the following expected counts for the respective strings above

$169/293 = .58 = 1$
$576/293 = 1.97 = 2$
$64/293 = 0.22 = 0$
$361/293 = 1.23 = 1$

The net change is that string 3 is deleted and replaced with a copy of string 2. The new population is

```
0 1 1 0 1
1 1 0 0 0
1 1 0 0 0
1 0 0 1 1
```

Crossover occurs between strings 1 and 2 with a random crosspoint of 4

```
0 1 1 0 | 1
1 1 0 0 | 0
```

which produces

```
0 1 1 0 0
1 1 0 0 1
```

Crossover occurs between strings 3 and 4 with a random crosspoint of 2

```
1 1 | 0 0 0
1 0 | 0 1 1
```

which produces

```
1 1 0 1 1
1 0 0 0 0
```

This makes the new population

```
0 1 1 0 0
1 1 0 0 1
1 1 0 1 1
1 0 0 0 0
```

These new strings have respective values of 12, 25, 27, and 16 which evaluate to 144, 625, 729, and 256. The new average fitness is

$$(144 + 625 + 729 + 256)/4 = 439.$$

Note how the average fitness increased from 293 to 439 with just one cycle of reproduction and crossover.

A.1.0.2 Multi-variable Function. The traditional method for determining the maximum and minimum points of a function requires setting the differential of the function equal to zero. This method is certainly more reliable for functions which are clearly differentiable; however, for complex function involving tens or even hundreds of variables, direct differentiation becomes intractable. The relatively simple function below will demonstrate the manner in which a genetic algorithm can be used to optimize a simple function. This basic method can be used to apply genetic algorithms to functions of any order of complexity.

Example function: $f(x, y) = 2(x^2 - 10y) + 5(e^y - x)$

The first step involved in applying genetic algorithms to any problem is to develop the encoding of the search space over which the genetic algorithm explores for solutions. In the case of functional optimization, the search space is the domain of the variables contained within the function. The above example contains two variables, x and y. Let us further simplify the problem by considering only positive integer values ranging from 0 to 63 as the domain of x and y. To encode the search space as a string of binary characters, the first 7 digits of a 14 digit string can be arbitrarily assigned to represent the value of x, and the second half of the string to represent the value of y. Now, any point (x, y) can be represented in the genetic algorithm as a single 14 digit binary string.

The second step is to implement the means to evaluate the fitness of each member of a population of strings which represent candidate solutions to the function being optimized. Assuming the function can be legally evaluated, the fitness value of each population member is simply the value of the function evaluated at the point (x, y) represented by the binary string. Through the searching action of the genetic algorithm, the population of strings evolves to conform to better and better solutions, eventually finding the optimal or near optimal solution to the function.

A.2 Theory

This section introduces schema notation of genetic algorithms. Then the fundamental theory shows that genetic algorithms produce increasingly better populations. This material is extracted from Merkle's thesis (36:18-20).

Schema. Goldberg develops an estimate for the performance of the SGA(22:28-33). Theoretical analysis of GA performance makes extensive use of *schemata*, or similarity templates. Schemata are strings composed of characters taken from the genetic alphabet, with the addition of the "don't care" character. A schema thereby describes a subset of the potential solutions. For example, the schema 1******* represents the set of all 8-bit strings which contain a 1 in the first position. Likewise, the schema 1******0 represents the set of all 8-bit strings which begin with a 1 and end with a 0.

The defining length, $\delta(H)$, of a schema is the "distance" between the index of the first specified position and the index of the last specified position. For example, $\delta(1*****0*) = 7 - 1 = 6$, while $\delta(1*******) = 1 - 1 = 0$. The *order* of a schema H, which is denoted $o(H)$, is the number of specified positions in the schema. For example, $o(1*******) = 1$, while $o(11111111) = 8$.

The schema concept can be extended to apply to absolute and relative ordering problems. Following Kargupta(33), an absolute ordering schema defines a set of valid permutation strings. For example, the absolute o-schema ! 1 ! 5 ! ! represents the set of all permutation strings for which the second and fourth positions contain alleles 1 and 5, respectively. This o-schema is distinct from the standard schemata * 1 * 5 * * in that the former requires that the string represent a valid permutation, while the latter does not.

Following Goldberg(22), Kargupta uses $rs^{t}(H)$ to denote the set of all valid permutation strings in which the alleles specified in H occur in the specified order. For example, $rs^{6}(1,5)$ represents all permutation strings of length 6 in which the allele 5 occurs after the allele 1.

Fundamental Theorem. Defining the average fitness of a string matching a schema H to be $f(H)$, the average population fitness to be $\bar{f}$, and the number of strings in a

population at time t which match the schema to be $m(H,t)$, the effect of the reproduction operator is

$$m(H,t+1) = m(H,t)\frac{f(H)}{\bar{f}} \tag{3}$$

Noting that crossover disrupts a schema only when the crossover point occurs within the defining length of the schema, the probability of survival under crossover for a schema in a string of length l is

$$p_s \geq 1 - p_c\frac{\delta(H)}{l-1} \tag{4}$$

where p_c is the probability of crossover and the inequality is used to reflect the fact that crossover may not actually disrupt the schema even when the crossover point is within the defining length.

The probability of survival for the above schema under the mutation operator then can be estimated as

$$p_{m_s} \approx 1 - o(H)p_m, p_m \ll 1 \tag{5}$$

where p_m is the probability of mutation. Combining these results and omitting negligible terms gives an estimate for the expected number of examples of a schema in the next generation:

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m\right] \tag{6}$$

This is referred to as the Fundamental Theorem of Genetic Algorithms, and can be interpreted as stating that "short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations" (22:33). This result also goes by the name of the Schema Theorem.

A.3 Applications

A genetic algorithm provides solutions to search problems. Since GAs require no knowledge of the problem, they are well suited for problems for which no known algorithm exists. Genetic algorithms may also be beneficial to problems in which all known algorithms take unacceptable time to execute.

This section divides search problems into two major categories: functional optimization and combinatorial optimization. An example of a functional optimization is finding the minimum of $f(x,y) = 100 * (x^2 - y)^2 + (1 - x)^2$. An example of a combinatorial optimization problem is finding the shortest circuit which contains all vertices in a fully connected graph (traveling salesman problem).

Continuous, infinite search space characterize functional optimization problems. Discrete, finite search spaces characterize combinatorial problems. Although a functional problem may have an infinite search space, any computer realization eventually requires the search space to be discretized to a finite domain. The size of the search space then becomes a function of the desired accuracy, in addition to the number of parameters.

Functional Optimization. There are many approaches to functional optimization. Differential algorithms use brute force mathematics to derive the minimum of the function. This approach, however, only works when the function is differentiable. For non-differentiable functions, gradient-based or hill-climbing algorithms can be used. A gradient-based approach uses a greedy algorithm to direct the search in the most promising direction. The greedy algorithm operates on the basis of local decisions to guide the search toward the globally optimal solution. This approach works fine for simpler functions, but does not perform as well on complex functions containing many minima. Consequently, a more robust search strategy must be applied to avoid being trapped by local minima. Monte Carlo random search, simulated annealing, and genetic algorithms are search-based algorithms which are applicable to optimization of complex functions.

The search space for a function consists of the domain of the variables contained in the function to be optimized. The solution will be the set of n values, where n is the number of variables in the function, and $f(v_1, v_2, v_3, ..., v_n)$ is either a maximum or minimum solution depending on the type of optimization being performed.

Solution of $f(x_1, x_2, x_3, ..., x_n) = [v_1, v_2, v_3, ..., v_n]$

The application of genetic algorithms to functional optimization problems involves two things; the encoding of the domain or search space as a genetic string, and the imple-

mentation of the fitness function, which is simply the function to be optimized. The genetic algorithm evaluates the fitness of each member of a population of strings and awards an increasing number of copies to strings of above average fitness, while decreasing the overall number of strings with below average fitness. The effects of crossover serve to combine the positive aspects of two strings into one solution. Eventually, the solutions which correspond to the strings in the population, reach a point of optimal or near optimal solution quality.

Combinatorial Optimization. Combinatorial problems attract much attention within the Genetic Algorithm community. The Fifth International Conference on Genetic Algorithms (20) published papers on vehicle routing, traveling salesman problem, and set partitioning. An entire session of the conference focused on scheduling problems. This section presents two combinatorial problems which represent different aspects of combinatorial encodings.

A.3.0.3 Task/Process Assignment. In this problem, m tasks t_i are to be assigned to n processors p_j in a way to minimize completion time of all tasks. Let the string consist of m genes which can take on one of n alleles. Each gene's locus (i) corresponds to a task (t_i) while the gene's value (j) represents the processor (p_j) to which the task is assigned. For illustration, assume a problem of six tasks ($m = 6$) to be assigned to four processors ($n = 4$). A possible encoding is

```
1 4 3 2 1 4
```

This encoding states that

t_1 assigned to p_1
t_2 assigned to p_4
t_3 assigned to p_3
t_4 assigned to p_2
t_5 assigned to p_1
t_6 assigned to p_4

The user provides a fitness function to evaluate the encoding and the GA does the rest.

A.3.0.4 Traveling Salesman Problem. Some combinatorial problems present problems to the traditional GA approach (22:170). These problems are characterized as solutions which represent order. A five city traveling salesman problem illustrates the problem with crossover. Let the locus of a gene represent the order in which cities (specified by the value of the gene) are visited. The first gene represents the starting city, the second gene represents the second city visited, ... The fifth gene represents the fifth city visited and return to the first city is implied. Let the following strings represent two solutions in a population.

```
Parent 1 = 1 2 3 4 5
Parent 2=  1 3 5 4 2
```

and let the crossover point be 2 (a point between the second and third genes).

```
Parent 1 = 1 2 | 3 4 5
Parent 2=  1 3 | 5 4 2
```

The result of this crossover is

```
Child 1 = 1 2 5 4 2
Child 2=  1 3 3 4 5
```

. Note that both children represent invalid solutions. Child 1 visits city 2 twice and doesn't visit city 3. Child 2 visits city 3 twice and doesn't visit city 2 at all.

Several approaches exist to fix this problem. One approach uses a heuristic to repair any broken children. Another approach uses a penalty function to decrease the fitness of any invalid children.

A.4 Computer Program Development

This section provides pseudo code based on appendix C of Goldberg's book (22). That appendix provides Pascal code for a simple genetic algorithm. Some simplifications where made for clarity: statistical reporting was eliminated and the data structure was modified to more resemble an object-oriented design. Note that the GENESIS code widely used is slightly different in implementation.

Data Definition.

```
allele-type:        boolean
chromosome-type:    array[1..max-length] of allele
fitness-type:       real
individual-type:
  record of
    chromosome:         chromosome-type
    fitness:            fitness-type
  end record
pop-type:
  record of
    individual:         array[1..pop-size] of individuals
    sum-fitness:        real
    size:               integer
    chromosome-size:    integer
  end record
```

Control Loop.

```
pop:            pop-type
old-pop:        pop-type
mate1:          chromosome-type
mate2:          chromosome-type
child1:         chromosome-type
child2:         chromosome-type
BEGIN
  USER SETS pop.size, pop.chromosome-size
  initialize(pop)
  FOR EACH GENERATION
    old-pop= pop
    SET pop.size= 0
    WHILE pop.size < old-pop.size
      mate1= select(pop)
      mate2= select(pop)
      crossover(mate1,mate2,child1,child2)
      pop.individual[pop.popsize]=   child1
      pop.individual[pop.popsize+1]= child2
      pop.size=  popsize+2
    ENDWHILE
  NEXT GENERATION
END
```

Evaluation.

```
input/output
  individual:  individual-type
BEGIN
  individual.fitness= FITNESS(individual.chromosome)
END
```

Initialize.

```
input/output
  pop:  pop-type
BEGIN
  INITIALIZE i=0
  FOR i= 1 to pop.size
    FOR j= 1 to  pop.chromosome-size
      SET pop.individual[i].chromosome[j]= RANDOM(0,1)
    NEXT j
  NEXT i
END
```

Select.

```
input
  pop:              pop-type
  target-fitness:   fitness-type
  slot:             fitness type
output
  individual:       individual-type
BEGIN
  SET target-fitness= RANDOM(0..pop.sum-fitness)
  INITIALIZE slot= 0
  INITIALIZE i=0
  WHILE slot < target-fitness
    i= i + 1
    slot=  slot + pop.individual[i].fitness
  ENDWHILE
  RETURN(pop.individual[i])
END
```

Crossover.

```
  input
    pop:       pop-type
    mate1:     chromosome-type
    mate2:     chromosome-type
  output
    child1:    chromosome-type
    child2:    chromosome-type
BEGIN
  IF CROSSOVER DESIRED (BASED ON CROSSOVER PROBABLITY)
    SET crosspoint= RANDOM(1..pop.chromosome-length -1)
  ELSE
    SET crosspoint= pop.chromosome-length
  ENDIF
  FOR j=1 to crosspoint
    child1.chromosome[j]= mutation(mate1.chromosome[j])
    child2.chromosome[j]= mutation(mate2.chromosome[j])
  NEXT crosspoint
  FOR j= crosspoint+1 to pop.chromosome-length
    child1.chromosome[j]= mutation(mate2.chromosome[j])
    child2.chromosome[j]= mutation(mate1.chromosome[j])
  NEXT j
  evaluate(child1)
  evaluate(child2)
END
```

Mutation.

```
  input/output
    allele:  allele-type
BEGIN
  IF MUTATION DESIRED (BASED ON MUTATION PROBABLITY)
    allele= not allele
  ELSE
    allele= allele
  ENDIF
  RETURN(allele)
END
```

A.5 Software Available

Software packages described in this section comes from a compilation in *Parallel Genetic Algorithms: Theory and Applications* (19). These packages are available on thor (the AFIT parallel network) under

/usr/genetic/Software.

GENESIS. GENESIS (GENEtic Search Implementation System) was written by John Grefenstette in 1981. Since 1985 it has been widely distributed in the research community. It is written in C and has been implemented on Sun Sparcstations and IBM compatible PCs.

OOGA. OOGA (Object-Oriented Genetic Algorithm) was written by Lawrence Davis to support his *Handbook of Genetic Algorithms* (10). It is written in Lisp.

Splicer. Splicer was written for NASA/Johnson Space Center. It is written in C and developed on Apple Macintosh. It has been ported to Sun Sparcstations using X-windows interface.

Appendix B. Parallel Genetic Algorithms.

This appendix provides an overview of simple genetic algorithms (3).

B.1 General Process - Three GA Models

The simple genetic algorithm described previously doesn't lend itself to parallelization (except for farming, explained in section B.2.0.5). However, a generalization of the simple model does lend itself to data parallelization. This section describes the general GA model and then discusses three instantiations of the model. The following material is based on material from *Parallel Genetic Algorithms: Theory & Applications* (16).

Generalization of Genetic Algorithm Model. Genetic algorithms are based on natural selection and natural genetics (22). In the simple genetic algorithm described previously, the selection and genetic re-combination were global. However, in nature, selection and genetics aren't global. Animals in Australia don't survive because they are fitter than animals on the European, Asian, African, American, and Australian continents. They survive because they are the fittest on the Australian continent. Similarly, they don't mate with animals on other continents. On the other hand, a long-term view might consider migration effects of animals between continents; so,perhaps global operators aren't out of the question.

The generalized model allows varying degrees of locality. Let the population be laid out in a mesh network as in figure 25. Each node of the mesh represents an individual. The locality of any individual to all other individuals is based on the shortest-path from the particular individual to each of the other individuals.

Instantiations of the Generalized Model.

Simple Genetic Algorithm (SGA). This is the same model explained previously. The location of an individual on the mesh has no effect on selection and genetic operators. Selection and crossover are based on the global population. This global nature may allow a single individual to quickly dominate the population.

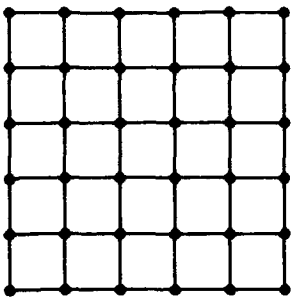


Figure 25. Population of Individuals

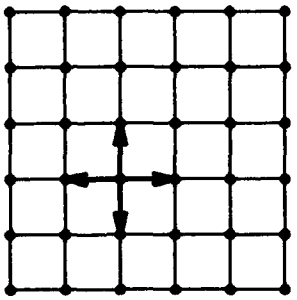


Figure 26. Radius of Influence of the Cellular Model

Cellular. In the cellular model, selection and genetics are based on an individual and it's four adjacent neighbors (see figure 26). Unlike the SGA model, a single individual can't quickly dominate a population. Initially, it may only dominate it's four immediate neighbors. Then it's neighbors can dominate their neighbors. In this way a good individual diffuses through the population. Since this diffusion takes multiple generations, this gives time for other individuals to develop. In effect, this allows niches to form in different areas of the mesh.

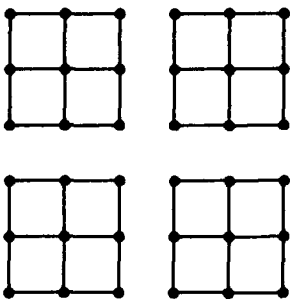


Figure 27. Subpopulations of an Island Model

Island. In the island model, the mesh is partitioned into n subpopulations. Within each subpopulation, the algorithm is the same as the SGA except a migration operator is added. This migration operator allows subpopulations to exchange genetic material which may enhance their populations. Within any subpopulation, it is possible that a single individual may quickly dominate; however, it won't immediately influence the other subpopulations (until migration occurs). Figure 27 shows the population of figure 25 divided into four subpopulations.

B.2 Theory - Conjecture

Niching.

Niching Theory. While optimizing over a search space which contains many near optimal solutions, simple genetic algorithms have been found to be susceptible to genetic drift (15)(27). Genetic drift happens when the stochastic error associated with the genetic operators, in conjunction with competing schemata corresponding to different peaks within the search space, causes the population to converge to one of the peaks (12). To overcome this pitfall, several implementations of genetic algorithms have been developed which attempt to maintain multiple, stable subpopulations on different peaks within the search space (11)(22).

The theoretical foundation for maintaining separate subpopulations is based on the limited resources model which can be observed among natural organisms. In nature, organisms rarely compete for the same resource; rather, they coexist while exploiting different resources within the same environment. The relative size of the population of a species corresponds to the availability of the resource which it exploits. The balance of the number of individuals among any one species is maintained by the fact that all resources are limited; if a resource becomes over utilized, starvation will correct the imbalance until an equilibrium is reached. The resulting system produces stable subpopulations of species which exploit their own region or niche within the overall environment.

The analogy between natural systems and genetic algorithms demonstrates the benefits gained by allowing the search space to be explored simultaneously within several

different niches of the search space. The number of population members allocated to each niche is proportional to the relative fitness of the niche, corresponding to the availability of the resource. The balance of the number of individuals allocated to the various niches is maintained throughout the search process in two ways. The relative fitness of one niche can decline with respect to other niches which prove to yield better solutions, thus causing a proportionally smaller number of individuals to be allocated to the niche with lesser performance. The competition for limited resources also affects the probability of survival of any one individual. If a large portion of the population is exploiting one area of the search space, the survival rate of any one individual will be proportionally smaller, even though the area of exploitation may correspond to a niche of high fitness. This competition for limited resources provides the genetic algorithm with the mechanism to prevent the population from converging upon one solution.

B.2.0.5 Models, Niching, Parallelization. There are basically two methods for incorporating a niche strategy in genetic algorithms. The first, proposed by DeJong (15), is called the crowding scheme which implements a generalized form of preselection to determine which population member is replaced by a new offspring. The idea is to replace an individual with a similar individual of higher fitness, thus preserving population diversity. Similarity is based on a bit-by-bit comparison of two strings to determine the number of bits in common. The crowding scheme works by preventing members of one niche from dominating the members of another niche.

The second method used to create niches in the search space is called the sharing scheme. The sharing scheme acts as a fitness scaling factor. The fitness of individuals are scaled according to the number of other similar individuals which are considered to be exploiting the same region or niche within the search space. In this scheme a distance metric is used to scale the fitness of an individual such that its expected survival rate is proportional to the number and proximity of neighboring individuals. Consequently, when a large number of individuals inhabit the same region of the search space, their corresponding fitness will be reduced, resulting in a proportionally smaller number of individuals to be allocated to that region of the search space. Likewise, when an individual

of relatively low fitness inhabits a sparsely populated region of the search space, its fitness will be scaled up, thus increasing its chances of survival.

The source of the distance metric used in the sharing scheme has led to two different implementations. The distance used to quantify the proximity of neighboring individuals can be determined at either the phenotypic level or the genotypic level. The phenotypic level refers to the decoded parameter search space. For a problem with n parameters, the distance is usually calculated as the Euclidean distance between two points in n-dimensional space. The distance at the genotypic level simply corresponds to the Hamming distance between two strings of the population.

Parallel implementations of genetic algorithms provide a perfect setting for encouraging the formation of niches within the search space. The subpopulations present on each node of a multiprocessor environment provide a natural mechanism to implicitly search separate niches of the search space, as opposed to explicitly forcing the formation of niches within a single population in a serial genetic algorithm implementation. However, some communication may be necessary between nodes in order to prevent a significant amount of duplication between subpopulations.

Farming . There are several different approaches to the decomposition of genetic algorithms on a parallel computer architecture. The approach should be chosen with the following considerations. The speed of communications between processors, the complexity of the fitness function, the size of the population, and the number of nodes available for execution. These considerations will determine both the manner in which the genetic algorithm is decomposed, and also the level of acceptable communications overhead. Farming is one approach to parallel decomposition which can be applied to genetic algorithms; however, the cost of communications usually makes farming an undesirable model for decomposing genetic algorithms in all but a few unique situations.

The speed of communications between processors can best be represented as a ratio of computational time versus communication time. In this format, the amount of computational work required in order to keep a processor busy between communications can be readily extracted. From this ratio, it is apparent that with any type of distributed memory

architecture, there is a significant benefit in terms of execution time if the level of required communications between processors can be minimized. Consequently, the most common method for decomposing a genetic algorithm on a distributed memory architecture is the island model, where the overall population is distributed among the processors and a separate copy of the genetic algorithm is executed on each of the processors. In the island model, each processor is responsible for evaluating the fitness of its subpopulation, and there is no inter-processor communication required to evaluate the fitness of the population members. The only communication present in this type of model is the information exchanged between processors in order to manage the overall population.

In contrast, a farming model distributes the fitness function among the processors, while a host node executes the genetic algorithm using a single population. The genetic algorithm can distribute the evaluation of the population across processors. The implementation is accomplished by distributing a copy of the evaluation function to each of the nodes such that the argument of the function is a population member which is sent by the host node. After evaluating the fitness of the population member, the processor node returns the fitness value to the host processor. The evaluation of the population with each generation is accomplished by sending out the population members to the network of processors, and then recording the fitness of each population member as it is returned back through the network. Each processor is responsible for evaluating $\frac{n}{p}$ population members where n is the size of the overall population and p is the number of processors. The communications overhead of this approach is substantial since an order n communications are necessary at every generation in order to evaluate the fitness of the population members. The only substantial difference between the farming approach and the island model is the farming model maintains a single population where as the island model contains separate subpopulations. Since the communications overhead is so large for the farming model, the only practical situation which would warrant its use would be when the resulting subpopulation sizes for an island model would be insufficient to prevent convergence. Given a costly evaluation function and a large number of available nodes, it may not be possible to increase the overall population size to a point where the subpopulation size ($\frac{n}{p}$) is sufficient.

However, even with an expensive evaluation function, it would be difficult to justify the farming model over an island model executed on a smaller number of nodes.

B.3 Parallel Genetic Algorithm Applications

Multi-modal Problems. Maximization of a sine curve is a good example of a multi-modal function: there is more than one optimal solution. However, since sine curves possess a well understood search space, genetic algorithms would not be used. In real world problems, the search space is not so well defined. It may not be known if it is a multi-modal function. It may be known that it is multi-modal, but the number of equal modes may not be known. If solutions are desired for more than one mode, parallel GAs which use niching should be considered.

Although the term multi-modal function is often used, multi-modal combinatorial problems exist as well. Consider a shortest path problem. Two equal shortest paths (between two vertices) may take fundamentally different routes. Each route corresponds to a mode.

Time-intensive Fitness Functions. There exists a growing number of parallelized algorithms which have been developed to solve or optimize some problem, given the necessary parameters or starting conditions. A unique capability of genetic algorithms is to encode some parameter space, and then find an optimal or near optimal point in the parameter space which minimizes or maximizes a function of the parameters. For example, consider the task scheduling problem. Many algorithms have been developed to search for an optimal schedule, given the tasks to be accomplished, their execution times, and any other constraints. All of these algorithms take the approach that tasks are known and fixed; however, a designer or programmer may be interested in knowing what types of tasks results in good schedules.

By encoding the parameter space of a search problem, and using an application program as a fitness function, a genetic algorithm can be used to find a set of input parameters which result in a good solution to the problem application. The result is a search contained within a search. The benefit of this type of embedded search is to gain insight into the

problem rather than simply solving the problem for particular circumstances. If parallelized code already exists for a particular application, a genetic algorithm can be executed on a single processor, while using the remaining processors to evaluate the fitness of the population members.

Given a parallelized algorithm which already efficiently utilizes a given parallel architecture, the simplest way to incorporate a genetic algorithm search on the parameter space of the parallel application is to host the genetic algorithm on a single processor and evaluate the members of the population one at a time, using the existing application software. Consider a second example, the mission routing problem. The mission routing problem starts with a defined terrain, an objective or target, and resources such as fuel or weapons. Although a mission routing algorithm may initially be designed to find an optimal or near optimal route to a target, the algorithm could instead be used to provide an objective function for determining the optimal weapons load in order to accomplish the mission. Embedding the route planning algorithm within a resource allocation model could produce combinations of weapons loads and routes which might otherwise have been overlooked.

B.4 Software Available

Parallel GENESIS. A parallel version of GENESIS implemented on an Intel Hypercube is available on thor.

AFIT Messy. A messy genetic algorithm was implemented by Dymek (18) and Merkle (36) as part of their thesis work at AFIT. It is written in C for the Intel Hypercube.

GAME. The Genetic Algorithm Manipulation Environment (GAME) is being developed by the European community (43). It is written in object-oriented C for IBM Compatible PCs and Sun Sparcstations. The current version runs only in a serial mode. Future versions are to allow simulation of parallel genetic algorithms on serial computers as well as execution of parallel genetic algorithms on parallel architectures.

Appendix C. UNITY Description of Genetic Algorithm

This appendix presents a UNITY description of a genetic algorithm and discussions on the invariant, fixed point, progress, and reachability of fixed point.

Chandy and Misra's UNITY language (5) is used to display inherent parallelism in algorithm. It is not a programming language, but a high level description language based on predicate logic. Although it can be used for serial algorithms, its strength lies in its ability to describe processes which may and may not be executed in parallel. Its proof system shows whether the algorithm can be guaranteed to produce its desired state. This is especially helpful in avoiding deadlock on parallel systems. The details of UNITY statements are not described here, but may be found in Chandy and Misra's text on parallel processing (5).

Program Genetic

```
declare
type allele-type is 0,1

type structure-type is record
  gene : array[1..MAX-STRUCT-SIZE] of allele-type
  fitness : fitness-type;
end record;

type solutions-type is record
  structures: array[1..SUB-POP-SIZE] of structure-type
end record;

type subpop-type is record
  solutions: array[0..MAX-GEN] of solutions-type;
  gen : integer;
    done : array[0..MAX-GEN] of boolean;
```

```
    the-best: integer;
end record;

pop : array[1..MAX-SUBPOP] of subpop-type;
the-best : structure-type;    {the best solution found}
epoch-size : integer;   {number of generations per epoch}

initially
the-best = NULL;
⟨|| ∀i : 1 ≤ i ≤ MAX-SUBPOP ::
  pop[i].solutions[0] = initialize(pop[i].solutions[0]);
  pop[i].gen = 1;
  ⟨|| ∀gen : 0 ≤ gen ≤ MAX-GEN ::
    pop[i].done[gen] = TRUE if gen=0;
    pop[i].done[gen] = FALSE if gen≠0;
  ⟩
⟩
assign
⟨|| gen :   1 ≤ gen ≤ MAX-GEN ::
  ⟨|| ∀i : 1 ≤ i ≤ MAX-SUBPOP ::
    pop[i].solutions[gen] := crossover(mutate(select(pop[gen-1])))
    pop[i].gen := pop[i].gen + 1
    pop[i].the-best := evaluate(pop[i].solutions[gen])
    pop[i].done[gen] := TRUE
  ⟩ if pop[i].done[gen-1] ∧
    ¬ pop[i].done[gen]
  the-best := max(pop[i].the-best);
⟩
[]
⟨|| ∀i : 1 ≤ i ≤ MAX-SUBPOP ::
```

exchange(pop[i].solution[gen])

if pop[i].gen mod epoch-size = 0∧

pop[i].done[gen-1] ∧¬ pop[i].done[gen]

⟩

end {Genetic}

Within any subpopulation, the transformation of one generation into the next is inherently sequential. This only happens when a generation is done but its succeeding generation is not done. However, subpopulations may perform this single transformation in parallel with other subpopulations.

Invariant. The invariant is that no new generation can be generated until its previous generation is done and solutions have been exchanged with other subpopulations. Solutions aren't exchanged for every generation, just every x generations where x is known as the epoch size.

$$I \equiv pop[i].done[gen-1] \vee \neg pop[i].done[gen] \tag{7}$$

Proof of the invariant involves proving it is stable and that the initial conditions lead to the invariant. The initial conditions set generation 0 of all populations equal to TRUE and all other generations of all populations to FALSE. This meets the invariant since

$$I \equiv pop[i].done[0] \vee \neg pop[i].done[gen] \tag{8}$$

holds for all generations. The invariant is stable in that the only operation which changes pop[i].done[gen] to true requires pop[i].done[gen-1] to be false. No operations changes pop[i].done[gen] back to FALSE.

Fixed Point. FP is derived from the assignment section by taking the conjunction of the negative of all conditions

$$FP \equiv \neg[pop[i].done[gen-1] \wedge \neg pop[i].done[gen]]$$

$\wedge$

$\neg[pop[i].genmodepoch - size = 0 \wedge$

$pop[i].done[gen - 1] \wedge$

$\neg pop[i].done[gen]]$

which reduces to

$FP \equiv \neg pop[i].done[gen - 1] \vee$

$pop[i].done[gen]]$

Progress. Metric for progress is *gen*. This is straight-forward. The genetic algorithm is designed for successive generations. Since *gen* can only be increased in value, all that is required is to show that the condition for it to increase is met. That condition is the same as the invariant; so progress does take place.

Fixed Point is Reachable. To show that FP is reachable, it must be shown that the initial conditions lead to FP. It is known that *gen* increases monotonically if only there exists a population which is not done, but whose successor is done. This means that successive generations are marked as done as *gen* increases to $MAX\text{-}POP\text{-}SIZE$. At this point all generations are done so $pop[i].done[gen]$ is always true.

Mapping to a Distributed Memory Architecture. Mapping the UNITY genetic algorithm description only requires the use of communication channels for the exchange operator. For purposes of a communication channel, a number of structures are chosen. These structures are part of the solutions. The level of detail of the given genetic algorithm does not map to any communications. The exchange assignment

exchange(pop[i].solutions[gen])

needs to go to another level of detail

$\langle \forall i \langle \exists j, k, l ::$

pop[i].solutions[gen].structure[j] =

pop[j].solutions[k].structure[l];

$\rangle\rangle$

where pop[i].solutions[gen].structure[j,l] represents the communication variable.

Appendix D. Source Code.

This appendix provides source listing of code modified in order to use a genetic algorithms to solve a mission routing problem.

D.1 Mission Routing

```
/*-------------------------------------------------------------
mr.h
  This provides data types used by mission routing
objects:
  POINT_type which represents a three dimensional
    cartesian point
-------------------------------------------------------------*/
typedef struct {
  int x;
int y;
int z;
}POINT_type;
```

```
#include <stdio.h>
#include <math.h>
#include "mr.h"
/*-------------------------------------------------------------
author:  Brent Olsan
date: 19 Sep 93
file:   ato.c
title:  air tasking order (ATO) object
function:  read ato data from a file and provide to othe
objects via access functions.  The ato file used is passed
as an input parameter to this routine in the initialization

history:  Makes use of Grimm's code

other modules used:
  terrain:  GetScale used to get the scale of the z axix
-------------------------------------------------------------*/
      POINT_type  Base;
      POINT_type  Target;
      int         MinRelAlt;
      int         MinAbsAlt;
      int         MaxAbsAlt;
      char        Mission[20];
      char        AltType[4];
/*-------------------------------------------------------------
Access Functions
-------------------------------------------------------------*/
char *GetMission()
{
  return(Mission);
}
POINT_type GetBase()
{
  return (Base);
}
POINT_type GetTarget()
{
  return (Target);
}
GetMinRelAlt()
{
  return(MinRelAlt);
}
GetMinAbsAlt()
```

```
{
  return(MinAbsAlt);
}
GetMaxAbsAlt()
{
  return(MaxAbsAlt);
 }
char *GetAltType()
{
  return(AltType);
}
/*-------------------------------------------------------------
Initialization Function
-------------------------------------------------------------*/
InitializeATO(ATOfile)
     char     *ATOfile;
{
  FILE *fATO, *fopen();
  if ((fATO = fopen(ATOfile, "r")) == NULL) {
     printf("error on ATO file%\n");}
 fscanf (fATO, "%s", Mission);
 fscanf (fATO, "%d", &Target.x);
 fscanf (fATO, "%d", &Target.y);
 fscanf (fATO, "%d", &Target.z);
 fscanf (fATO, "%d", &Base.x);
 fscanf (fATO, "%d", &Base.y);
 fscanf (fATO, "%d", &Base.z);
 fscanf (fATO, "%d", &MinRelAlt);
 fscanf (fATO, "%d", &MinAbsAlt);
 fscanf (fATO, "%d", &MaxAbsAlt);
 fscanf (fATO, "%s", AltType);
 fclose (fATO);
 Base.z=Base.z/GetScale();
 Target.z= Target.z/GetScale();
 printf("ATO File Initialized\n");
 }
```

D.3 Terrain

```
#include <stdio.h>
#include <math.h>
#include "extern.h"
#include "mr.h"
/*-------------------------------------------------------------
author:  Brent Olsan
date: 19 Sep 93
file:   terrain.c
title:  terrain object
function:  read terrain data from a file and provides to
objects via access functions.  The terrain file used is passed
as an input parameter to this routine in the initialization

history:  Makes use of Grimm's code

other modules used:
  ato:   uses GetMaxAbsAlt to get maximum absolute
    altitude
-------------------------------------------------------------*/
int terrain[1000][1000];
int maxx;
int maxy;
        int     minz;
int     maxz;
        int     scale;
/*-------------------------------------------------------------
Access Functions
-------------------------------------------------------------*/
GetElevation(Point)
POINT_type  Point;
{
return(terrain[Point.x][Point.y]);
}

GetMaxX()
{
return maxx;
}
GetMaxY()
{
return maxy;
}
GetMinZ()
{
```

```
return minz;
}
GetMaxZ()
{
return maxz;
}
GetRangeZ(Point)
POINT_type  Point;
{
  return ((GetMaxAbsAlt()-terrain[Point.x][Point.y]));
}
GetScale()
{
return scale;
}

/*-------------------------------------------------------------
Initialization Function
-------------------------------------------------------------*/
InitializeElevation(Infile)
        char *Infile;
{
int maxz;
int x;
int y;

char c;

FILE *fp,*fopen();

if ((fp = fopen(Infile, "r")) == NULL) {
printf("Input can't open %s\n",Infile);
}

fscanf(fp,"%d %d %d %d",&maxx,&maxy,&maxz,&scale);
fscanf(fp,"%c",&c);

maxz= 0;
minz=10000;
for (y=0;y<maxy;y++) {
  for (x=0;x<maxx;x=x++) {
    fscanf(fp,"%d",&terrain[x][y]);
    if (terrain[x][y] < minz)  minz=terrain[x][y];
    if (terrain[x][y] > maxz)  maxz=terrain[x][y];
    }
```

```
fscanf(fp,"%c",&c);
}
fclose(fp);
}
```

D.4 Radar

```
#include <stdio.h>
#include <math.h>
#include "extern.h"
#include "mr.h"
/*-------------------------------------------------------------
author:  Brent Olsan
date: 19 Sep 93
file:   radar.c
title:  radar object
function:  read radar data from a file and provide to other
objects via access functions.  The radar file used is passed
as an input parameter to this routine in the initialization

history:  Makes use of Grimm's code

other modules used: none
-------------------------------------------------------------*/
float       radar_matrix[26][19][16];
float       RadarWeight;

POINT_type  point;

/*-------------------------------------------------------------
Access Function
-------------------------------------------------------------*/
float GetRadarDetection(point)
POINT_type  point;
{
  return(radar_matrix[point.x][point.y][point.z]);
}
/*-------------------------------------------------------------
Initialization Function
-------------------------------------------------------------*/

InitializeRadar(Infile,InRadarWeight)
        char *Infile;
        float   InRadarWeight;
{
int x;
int y;
int     z;
int     i;
int     j;
int     k;
```

```
int     scale;
float    detection;
char c;

FILE *fp,*fopen();

RadarWeight=  InRadarWeight;
if ((fp = fopen(Infile, "r")) == NULL) {
printf("Input can't open %s\n",Infile);
}
printf("radar open\n");
fscanf (fp, "%d", &x);
fscanf (fp, "%d", &y);
fscanf (fp, "%d", &z);
fscanf (fp, "%d", &scale);

for (k = 0; k < z; k++) {
  for (j = 1; j <= y; j++) {
    for (i = 1; i <= x; i++) {
      fscanf (fp, "%f", &detection);
      radar_matrix [i][j][k] = detection;
    }
  }
}
fclose (fp);
printf("Radar Initialized\n");
      }
```

D.5 Free Structure

```
#include <stdio.h>
#include <math.h>
#include "mr.h"
/*-------------------------------------------------------------
author:  Brent Olsan
date: 19 Sep 93
file:   structure.c
title:  free progress route structure
functions:
  public:
    InitializeStructure:  sets structure length
    DecodeStructure: converts chromosome into a
      array of route points
    ExpandRoute:  converts an array of free route
      points into an array of bounded route points
    Access function to retrieve public data
  private:
    GetMinAlt:  Determine minimum altitude for any
       location
    GetBits: coverts genesis internal coding to an
       array of 0/1 integers
    DecodeBitStructure:  converts a 0/1 integer array
       to an array of route points
    DecodeBits:  converts binary representation to an
      integer
    DecodeStructure:  Converts a free     e encoding
      into an array of free route points
other modules used:
  terrain:
    GetScale
    GetElevation
    GetMaxX
    GetMaxY
    GetMinZ
    GetRangeZ
  ato:
    GetBase
    GetMinAbsAlt
    GetMinRelAlt
    GetMaxAbsAlt
    GetTarget
  ctoi (GENESIS module)
-------------------------------------------------------------*/
      POINT_type  Base;
```

```
        POINT_type  Target;
        POINT_type  GetBase();
        POINT_type  GetTarget();
        int         length;
        float         xscale;
        float       yscale;
        int         xbits;
        int         ybits;
        int         zbits;
        float       zscale;
        int         xdir;
        int         nockpts;

/*-------------------------------------------------------------
Access Functions
-------------------------------------------------------------*/
GetNoCkpts()
{
  return(nockpts);
}
GetXScale()
{
  return((int)xscale);
}
GetXDir()
{
  return(xdir);
}
GetXBits()
{
  return(xbits);
}
GetYBits()
{
  return(ybits);
}
GetYScale()
{
  return((int)yscale);
}
GetZBits()
{
  return(zbits);
}
GetZScale()
```

```
{
  return((int)zscale);
}

GetStructureLength()
{
  return(length);
}

/*-------------------------------------------------------------
GetMinAlt
  Determines the minimum altitude by returning the maximum of
  two values:
    Minimum Absolute altitude from the ato object
    Minimum Relative altitude from the ato object + terrain height
-------------------------------------------------------------*/
GetMinAlt()
{
  int  MinAlt;
  if (GetMinAbsAlt() < (GetMinRelAlt()+GetMinZ())) {
    MinAlt=  GetMinAbsAlt();}
  else{
    MinAlt=  GetMinRelAlt() + GetMinZ();}

  return(MinAlt);
}
/*-------------------------------------------------------------
Initialization Function
  Determines the number of bits for each coordinate (x,y,z) and
the scale (maximum number representable) for each coordinate.
  Determines the structure length by multiplying the number of
checkpoints times the number of bits for each checkpoint (the
number of checkpoints is an input parameter)
-------------------------------------------------------------*/
InitializeStructure(NoCheckPoints)
int  NoCheckPoints;
{
/*  printf("Initialize Structure\n");*/
  Base=      GetBase();
  Target=    GetTarget();

  xbits=     ceil((log((float)GetMaxX())/log(2.0)));
  xscale=    pow(2.0,(float)xbits)-1.0;
  ybits=     ceil((log((float)GetMaxY())/log(2.0)));
  yscale=    pow(2.0,(float)ybits)-1.0;
```

```
  zbits=  ceil(log((float)(GetMaxAbsAlt() -
          GetMinAlt())/GetScale())/log(2.0));
  zscale=    pow(2.0,(float)zbits)-1.0;

  nockpts=    NoCheckPoints;
  length=    NoCheckPoints*(xbits+ybits+zbits);
  printf("Structure Initialized\n");
}
/*-------------------------------------------------------------
GetBits
-------------------------------------------------------------*/
GetBits(Structure,BitStructure,length)
char  Structure[];
int  BitStructure[];
int  length;
{
  int  i;
  for (i=0;i<length;i++) {
     BitStructure[i]= Ctoi(&Structure[i],1);
   }
}
/*-------------------------------------------------------------
DecodeBitStructure
  takes an array of 0/1 integers and produces and array of
route points
-------------------------------------------------------------*/
DecodeBitStructure(BitStructure,Route)
  int  BitStructure[];
  POINT_type Route[];
{
  int i;
  Route[0]=  GetBase();
  for (i=0; i < GetNoCkpts();i++) {
    Route[i+1].x=
DecodeBits(&BitStructure[i*(xbits+ybits+zbits)],xbits)/
xscale*GetMaxX();
    Route[i+1].y=
DecodeBits(&BitStructure[i*(xbits+ybits+zbits)+xbits],ybits)/
yscale*GetMaxY();
    Route[i+1].z=ceil((float)
DecodeBits(&BitStructure[i*(xbits+ybits+zbits)+xbits+ybits],zbits)/
(float)zscale*(float)GetRangeZ(Route[i+1])/
(float)GetScale()+(float)GetElevation(Route[i+1])/(float)GetScale());
  }
  Route[i+1]=GetTarget();
```

```
}
/*-------------------------------------------------------------
DecodeBits
  Converts an array of bits into an integer value
-------------------------------------------------------------*/
DecodeBits(BitStructure,bits)
  int *BitStructure;
  int bits;
{
  int value;
  int i;
  value=0;
  for (i=0;i<bits;i++) {
      value=  value + value + *BitStructure;
    *BitStructure++;
  }
  return(value);
}

/*-------------------------------------------------------------
DecodeStructure
  Takes a GENESIS genetic structure and converts it into
an array of free route points
-------------------------------------------------------------*/

DecodeStructure(Structure,Route)
char        Structure[];
POINT_type  Route[];
{
  int i;
  int j;
  int BitStructure[1000];

  GetBits(Structure,BitStructure);
  DecodeBitStructure(BitStructure,Route);
}
/*-------------------------------------------------------------
ExpandRoute
  Takes an array of free route points, interpolates between
consecutive points, and produces an array of bounded route
points
-------------------------------------------------------------*/
ExpandRoute(FreeRoute,BoundRoute,length)
POINT_type  FreeRoute[];
POINT_type  BoundRoute[];
```

```
int         *length;
{
  int  ifr;
  int  ibr;
  int  delta;

  int  i;
  ibr= 0;

  BoundRoute[0]= FreeRoute[0];
  for (ifr=0; ifr < GetNoCkpts()+1;ifr++) {
    while ((BoundRoute[ibr].x  != FreeRoute[ifr+1].x) ||
           (BoundRoute[ibr].y  != FreeRoute[ifr+1].y) ||
           (BoundRoute[ibr].z  != FreeRoute[ifr+1].z)) {

      delta=  FreeRoute[ifr+1].x - BoundRoute[ibr].x;
      if (delta !=0) delta=  abs(delta)/delta;
      BoundRoute[ibr+1].x= BoundRoute[ibr].x + delta;

      delta=  FreeRoute[ifr+1].y - BoundRoute[ibr].y;
      if (delta !=0) delta=  abs(delta)/delta;
      BoundRoute[ibr+1].y= BoundRoute[ibr].y + delta;

      delta=  FreeRoute[ifr+1].z - BoundRoute[ibr].z;
      if (delta !=0) delta=  abs(delta)/delta;
      BoundRoute[ibr+1].z= BoundRoute[ibr].z + delta;
      while (BoundRoute[ibr+1].z < GetElevation(BoundRoute[ibr+1])/1000) {
        BoundRoute[ibr+1].z++;
      }

      if (BoundRoute[ibr+1].z < GetElevation(BoundRoute[ibr+1])/1000) {
printf("bad route Route.z %d Elevation: %d\n",
  %BoundRoute[ibr+1].z,GetElevation(BoundRoute[ibr+1])); }
      ibr++;
    }
  }
  ibr++;
  *length=ibr;
}
```

D.6 Forced Structure

```
#include <stdio.h>
#include <math.h>
#include "mr.h"
/*-------------------------------------------------------------
author:  Brent Olsan
date: 19 Sep 93
file:   structure.c
title:  forced progress route structure
functions:
  public:
    InitializeStructure:  sets structure length
    DecodeStructure: converts chromosome into a
      array of route points
    ExpandRoute:  converts an array of free route
      points into an array of bounded route points
    Access function to retrieve public data
  private:
    GetMinAlt:  Determine minimum altitude for any
       location
    GetBits: coverts genesis internal coding to an
       array of 0/1 integers
    DecodeBitStructure:  converts a 0/1 integer array
       to an array of route points
    DecodeBits:  converts binary representation to an
      integer
    DecodeStructure:  Converts a free route encoding
      into an array of free route points
other modules used:
  terrain:
    GetScale
    GetElevation
    GetMaxX
    GetMaxY
    GetMinZ
    GetRangeZ
  ato:
    GetBase
    GetMinAbsAlt
    GetMinRelAlt
    GetMaxAbsAlt
    GetTarget
  ctoi (GENESIS module)
-------------------------------------------------------------*/
```

```
      POINT_type  Base;
      POINT_type  Target;
      POINT_type  GetBase();
      POINT_type  GetTarget();
      int         length;
      int         xscale;
      float       yscale;
      int         ybits;
      int         zbits;
      float       zscale;
      int         xdir;

/*-------------------------------------------------------------
Access Functions
-------------------------------------------------------------*/
GetXScale()
{
  return((int)xscale);
}
GetXDir()
{
  return(xdir);
}
GetYBits()
{
  return(ybits);
}
GetYScale()
{
  return((int)yscale);
}
GetZBits()
{
  return(zbits);
}
GetZScale()
{
  return((int)zscale);
}
GetStructureLength()
{
  return(length);
}

/*-------------------------------------------------------------
```

```
GetMinAlt
  Determines the minimum altitude by returning the maximum of
  two values:
    Minimum Absolute altitude from the ato object
    Minimum Relative altitude from the ato object + terrain height
-------------------------------------------------------------*/
GetMinAlt()
{
  int  MinAlt;
  if (GetMinAbsAlt() < (GetMinRelAlt()+GetMinZ())) {
    MinAlt=  GetMinAbsAlt();}
  else{
    MinAlt=  GetMinRelAlt() + GetMinZ();}

  return(MinAlt);
}
/*-------------------------------------------------------------
Initialization Function
  Determines the number of bits for two coordinates (y,z) and
the scale (maximum number representable) for each coordinate.
Calculates the number of checkpoints by taking the difference
between the staging base and target and subtracting 1.
  Determines the structure length by multiplying the number of
checkpoints times the number of bits for each checkpoint.
-------------------------------------------------------------*/
InitializeStructure()
{
  Base=      GetBase();
  Target=    GetTarget();
  xscale=    abs(Target.x - Base.x) - 1;
  xdir=      abs(Target.x - Base.x)/(Target.x - Base.x);
  ybits=     ceil((log((float)GetMaxY())/log(2.0)));
  yscale=    pow(2.0,(float)ybits)-1.0;
  zbits=  ceil(log((float)(GetMaxAbsAlt() -
  GetMinAlt())/GetScale())/log(2.0));

  zscale=    pow(2.0,(float)zbits)-1.0;
  length=    xscale*(ybits+zbits);
  printf("Structure Initialized\n");
}
/*-------------------------------------------------------------
GetBits
-------------------------------------------------------------*/
GetBits(Structure,BitStructure,length)
char  Structure[];
```

```
int  BitStructure[];
int  length;
{
  int  i;
  for (i=0;i<length;i++) {
     BitStructure[i]= Ctoi(&Structure[i],1);
   }
}
/*-------------------------------------------------------------
DecodeBitStructure
  takes an array of 0/1 integers and produces and array of
route points
-------------------------------------------------------------*/
DecodeBitStructure(BitStructure,Route)
  int  BitStructure[];
  POINT_type Route[];
{
  int i;
  Route[0]=  GetBase();
  for (i=0; i < xscale;i++) {
    Route[i+1].x= (i+1)*GetXDir() + GetBase().x;
    Route[i+1].y= DecodeBits(&BitStructure[i*(ybits+zbits)],ybits)/
yscale*GetMaxY();
    Route[i+1].z=
ceil((float)DecodeBits(&BitStructure[i*(ybits+zbits)+ybits],zbits)/
(float)zscale*(float)GetRangeZ(Route[i+1])/
(float)GetScale()+(float)GetElevation(Route[i+1])/
(float)GetScale());
/*    printf("%d %d %d\n",Route[i].x,Route[i].y,Route[i].z);*/
  }
  Route[i+1]=GetTarget();
}
/*-------------------------------------------------------------
DecodeBits
  Converts an array of bits into an integer value
-------------------------------------------------------------*/
DecodeBits(BitStructure,bits)
  int *BitStructure;
  int bits;
{
  int value;
  int i;
  value=0;
  for (i=0;i<bits;i++) {
      value=  value + value + *BitStructure;
```

```
/*    value+= *BitStructure*((int)pow(2.0,(float)(bits-i-1)));*/
    *BitStructure++;
  }
  return(value);
}
/*-------------------------------------------------------------
DecodeStructure
  Takes a GENESIS genetic structure and converts it into
an array of free route points
-------------------------------------------------------------*/
DecodeStructure(Structure,Route)
char        Structure[];
POINT_type  Route[];
{
  int i;
  int j;
  int BitStructure[1000];

  GetBits(Structure,BitStructure);
  DecodeBitStructure(BitStructure,Route);
}
/*-------------------------------------------------------------
ExpandRoute
  Takes an array of free route points, interpolates between
consecutive points, and produces an array of bounded route
points
-------------------------------------------------------------*/
ExpandRoute(FreeRoute,BoundRoute,length)
POINT_type  FreeRoute[];
POINT_type  BoundRoute[];
int         *length;
{
  int  ifr;
  int  ibr;
  int  delta;

  int  i;
  ibr= 0;

  BoundRoute[0]= FreeRoute[0];
  for (ifr=0; ifr < GetXScale()+1;ifr++) {
    while ((BoundRoute[ibr].x  != FreeRoute[ifr+1].x) ||
           (BoundRoute[ibr].y  != FreeRoute[ifr+1].y) ||
           (BoundRoute[ibr].z  != FreeRoute[ifr+1].z)) {
```

```
      delta=  FreeRoute[ifr+1].x - BoundRoute[ibr].x;
      if (delta !=0) delta=  abs(delta)/delta;
      BoundRoute[ibr+1].x= BoundRoute[ibr].x + delta;

      delta=  FreeRoute[ifr+1].y - BoundRoute[ibr].y;
      if (delta !=0) delta=  abs(delta)/delta;
      BoundRoute[ibr+1].y= BoundRoute[ibr].y + delta;

      delta=  FreeRoute[ifr+1].z - BoundRoute[ibr].z;
      if (delta !=0) delta=  abs(delta)/delta;
      BoundRoute[ibr+1].z= BoundRoute[ibr].z + delta;
      while (BoundRoute[ibr+1].z < GetElevation(BoundRoute[ibr+1])/1000) {
        BoundRoute[ibr+1].z++;
      }

      if (BoundRoute[ibr+1].z < GetElevation(BoundRoute[ibr+1])/1000) {
printf("bad route Route.z %d Elevation: %d\n",
  %BoundRoute[ibr+1].z,GetElevation(BoundRoute[ibr+1])); }
      ibr++;
    }
  }
  ibr++;
  *length=ibr;
}
```

D.7 Evaluation

```
#include <math.h>
#include "mr.h"
#include "extern.h"
/*-------------------------------------------------------------
author:  Brent Olsan
date: 19 Sep 93
file:   mr.free.c
title:  free progress route evaluation object
functions:
  InitializeMR:  Initializes all mission routing objects
  InitializeEvaluation:  Initializes radar weight for the
    evaluation function
  eval:  returns evaluation of a GENESIS genetic structure
  distance:  return cost of a bounded route
other modules used:
  terrain
  radar
  ato
  structure
        history:  distance routine adapted from Grimm's code
-------------------------------------------------------------*/
float   RadarWeight;
/*-------------------------------------------------------------
InitializeMR
  This routine initializes all mission routing objects based
on information in a mission routing file.
-------------------------------------------------------------*/
InitializeMR(MRFile)
  char    MRFile[20];
{
  char    TerrainFile[20];
  char    RadarFile[20];
  char    ATOFile[20];
  float   RadarWeight;
  int     NoCkpts;
  FILE    *fp,  *fopen();
        if ((fp = fopen(MRFile, "r")) == NULL){
  printf("Input can't open %s ",MRFile);}

        fscanf(fp,"%s",TerrainFile);
        printf("Terrain File is %s\n",TerrainFile);
InitializeElevation(TerrainFile);

        fscanf(fp,"%s",RadarFile);
```

```
        printf("Radar File is %s\n",RadarFile);
InitializeRadar(RadarFile);

        fscanf(fp,"%s",ATOFile);
        printf("ATO File is %s\n",ATOFile);
InitializeATO(ATOFile);

        fscanf(fp,"%d",&NoCkpts);
        printf("Number of Checkpoints is %d\n",NoCkpts);
InitializeStructure(NoCkpts);

        fscanf(fp,"%f",&RadarWeight);
        printf("Radar Weight is %1.2f\n",RadarWeight);
InitializeEvaluation(RadarWeight);

        fclose(fp);
}

/*-------------------------------------------------------------
InitializeEvaluation
  Intializes the evaluation function with the proper weighting
  for radar detection
-------------------------------------------------------------*/
InitializeEvaluation(InRadarWeight)
float  InRadarWeight;
{
  RadarWeight=  InRadarWeight;
}
/*-------------------------------------------------------------
eval
  this is the function called by the GENESIS genetic algorithm.
  this routine executes the following process
    convert GENESIS structure to an array of 0/1 integers
    convert the 0/1 array to an array of free route points
    convert the array of free route points to an array of bounded
      route points
    uses distance to evaluate the bound route point array
-------------------------------------------------------------*/
double eval(buff, length)
register char buff[];
int length;
{

int i;
int     k;
```

```
int     BoundRouteLength;
float   distance();
POINT_type  FreeRoute[10000];
POINT_type  BoundRoute[10000];
int         BitStructure[1000];

GetBits(buff,BitStructure,length);
        DecodeBitStructure(BitStructure,FreeRoute);
ExpandRoute(FreeRoute,BoundRoute,&BoundRouteLength);

return((double)(distance(BoundRoute,BoundRouteLength)));
      }
/*-------------------------------------------------------------
distance
  this route takes an array of bounded route points, sums
the distance of the entire route, and sums the radar detection
probability of the entire route and then returns a weighted
value of the radar detection and distance
-------------------------------------------------------------*/
float distance(Route,length)
POINT_type  Route[];
int         length;
{

        float   GetRadarDetection();
int i;
int     x;
int     y;
int     z;
int     oldx;
int     oldy;
int     oldz;
float   dist;
float   cost;
oldx= Route[0].x;
old  Route[0].y;
oldz= Route[0].z;
cost=   0.0;
for (i=1;i<length; i++) {
  dist=  sqrt((float)((Route[i].x -oldx) * (Route[i].x - oldx) +
      (Route[i].y -oldy) * (Route[i].y - oldy) +
      (Route[i].z -oldz) * (Route[i].z - oldz))) *
 GetScale();

  oldx=  Route[i].x;
```

```
  oldy=  Route[i].y;
  oldz=  Route[i].z;
  cost+=  (1.0-RadarWeight)*dist +
GetRadarDetection(Route[i])*RadarWeight*dist;
}
return(cost);
      }
```

Appendix E. Test Data.

All data in this appendix is the same used by Grimm (29).

E.1 Air Tasking Orders

```
AFIT-1
17  17  8000
 1   1  5000
0
4000 14000 MSL



AFIT-GO
24  16  8000
 1   1  5000
0
4000 14000 MSL
```

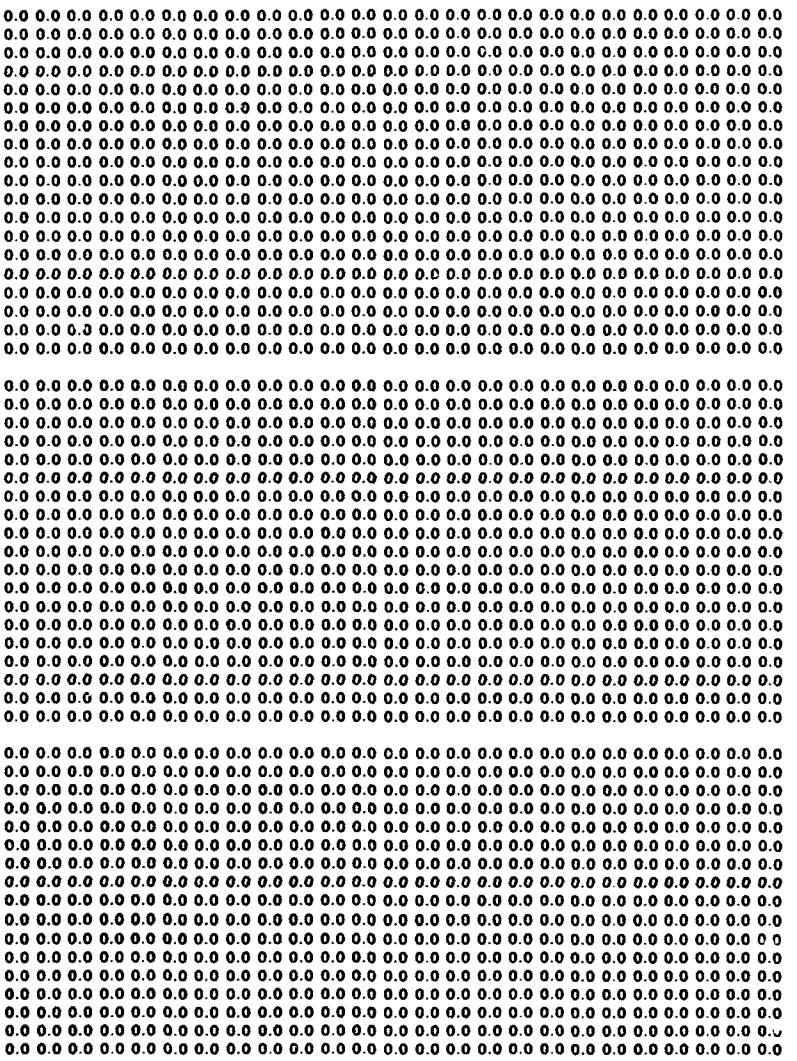
E.2 Terrain

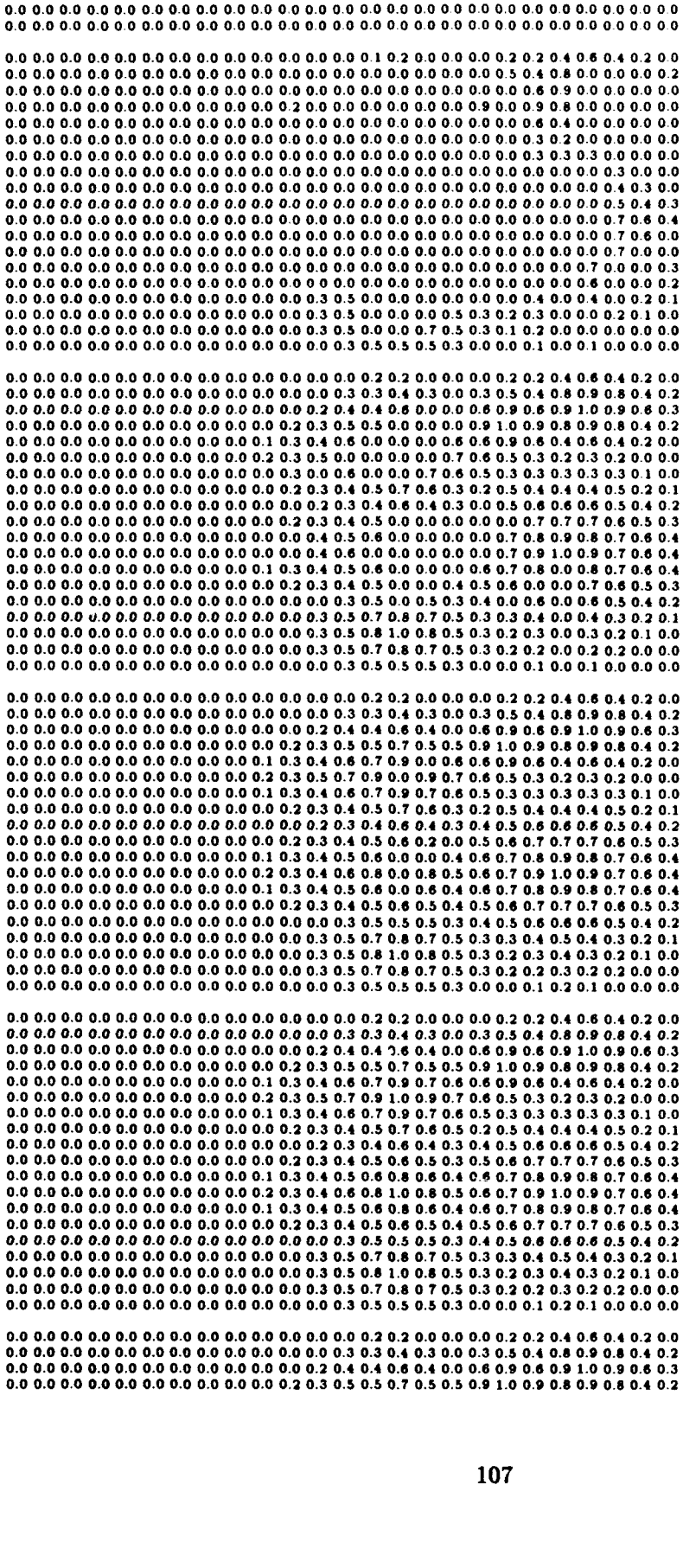
25 19 1000 1700 1775 2005 2010 2000 2000 2020 2450 3150 2350 2270 2320 2505 2590
2650 3000 2890 2675 2500 2270 2500 2905 2960 2470 2000 1700 2000 2020 2070 2495 3120
3110 3135 2750 2550 2690 2980 3100 3050 3150 3455 3535 3020 2705 2510 2700 3150 3200
3210 2890 1815 2100 2090 2200 3170 3700 3465 2995 2520 2610 2890 3045 3285 3580 3805
4200 4410 3050 3620 2500 2735 3335 3670 3480 3105 2090 2110 2170 2310 3250 4305 3250
2750 2600 2700 2875 3190 3500 3800 4120 4885 4705 2995 3485 2515 2625 3215 3740 3595
3085 2100 2200 2300 2995 3590 3555 3100 2515 2900 3470 3275 3315 3605 4195 4700 5500
3755 3495 3340 2560 2600 3200 3700 3710 3100 2250 2300 2530 3215 3700 3420 2850 2500
3205 3700 3300 3005 3710 4300 5500 4500 3150 3600 3400 2505 2605 3130 3305 3600 3200
2100 2370 2690 3090 3700 3430 2895 2510 3500 4205 3360 3005 3560 4500 5000 3300 3115
3550 3650 3000 2700 2700 3100 3300 3460 2050 2405 2600 3200 3700 3500 2880 2600 3395
4200 3610 3360 3300 3900 3515 3270 3250 3050 3095 3500 3590 2500 2700 3100 3540 2000
2320 2775 3220 3790 3460 3080 2690 3000 3900 3775 3420 3190 3600 3620 3795 3815 3910
3590 3225 3400 3585 2600 2700 3225 1950 2200 2545 3140 3710 3700 3120 2650 2770 3600
3900 3400 3070 3790 4500 4750 5010 4690 4050 3470 3390 3600 2500 2700 3000 1900 2015
2450 2720 3100 3680 3200 2650 2550 3225 4200 3450 3120 3980 5700 5920 5000 4600 4000
3200 3500 3600 2550 2700 2910 1850 2000 2350 2700 2800 2950 3220 2790 2650 3000 4200
3450 3100 4100 6000 4800 4595 4200 3500 3200 4000 3400 2590 2990 3200 1800 1955 2290
2650 2800 2800 3100 2800 2600 2755 3890 3450 3100 4000 6000 4600 4525 3770 3000 3550

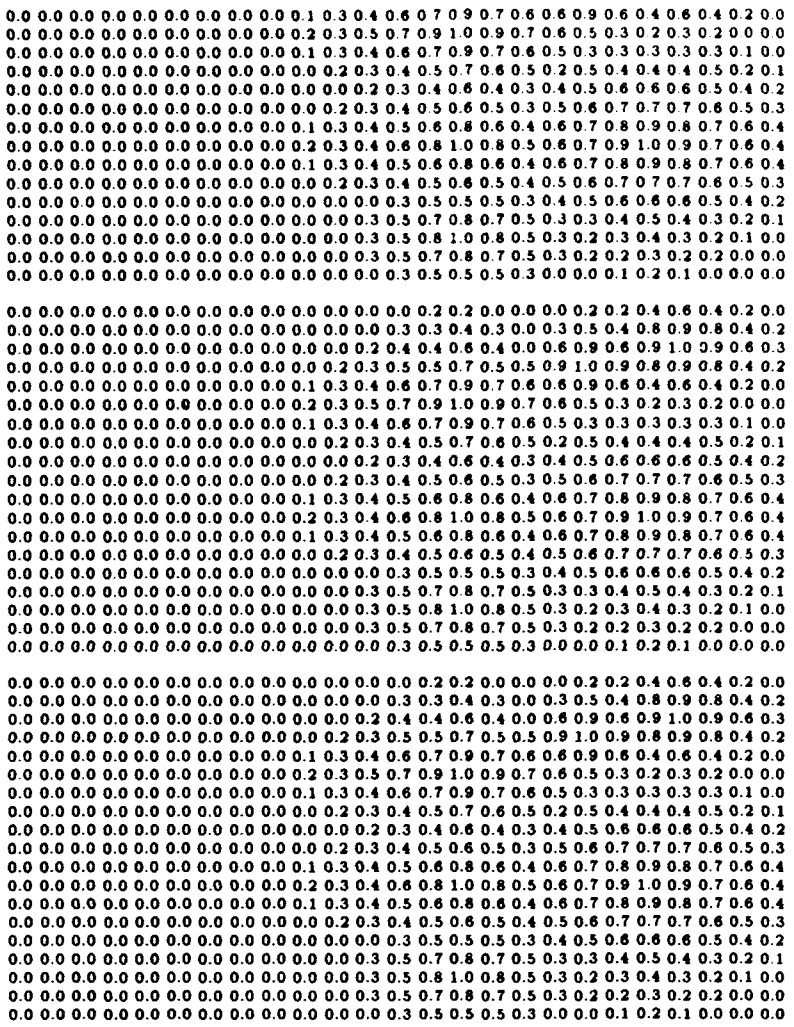
4050 3200 2640 3400 2990 1750 1900 2190 2800 3150 2900 2770 2700 2600 2540 3500 3225
3100 4000 5000 4200 3840 3450 3400 4120 3760 2975 3020 3040 2650 1710 1870 2100 2450
3200 3390 3400 3200 3000 2540 2780 3090 3100 3555 4105 3800 3500 3115 3540 3605 3205
2700 3210 3000 2650 1680 1810 2220 2350 2850 3400 4000 3900 3500 2810 2605 2760 2930
3400 3820 3500 3140 3200 3150 2560 2680 2950 3285 2710 2600 1625 1710 1900 2235 2900
3500 4500 3600 3385 2810 2515 2600 2900 3250 3370 3110 3000 3000 2500 2750 2750 3200
2885 2640 2550 1600 1700 1880 2210 2500 3430 3715 3300 3200 2800 2550 2580 2900 3290
3250 2800 2800 2560 2605 2600 2785 2810 2620 2500 2400 1585 1645 1700 2170 2260 2500
2770 2700 2700 2465 2320 2500 2740 2890 2905 2775 2540 2580 2600 2540 2510 2475 2425
2395 2320

E.3 Radar

25 19 51 1000







0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

Bibliography

1. Alliot, Jean-Marc and others. "Genetic Algorithms for solving Air Traffic Control conflicts." *IEEE Conference on Artificial Intelligence for Applications*. 338–344. IEEE, 1993.
2. Brassard, Giles and Paul Bratley. *Algorithmics: Theory and Practice* (First Edition). Englewood Cliffs NJ: Prentice Hall, 1988.
3. Brinkman, Don and Brent Olsan. *Parallel Genetic Algorithms*. Technical Report, Wright-Patterson AFB, OH: Air Force Institute of Technology, 1993.
4. Brinkman, Don and Brent Olsan. *Simple Genetic Algorithms*. Technical Report, Wright-Patterson AFB, OH: Air Force Institute of Technology, 1993.
5. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design: A Foundation*. Reading MA: Addison-Wesley Publishing Company, 1989.
6. Cohen, Edith. "Efficient parallel shortest-path in digraphs with a separator decomposition." *ACM Symposium on Parallel Algorithms and Architectures*. 57–67. ACM, 1993.
7. Cohoon, J. P. and others. "A Multi-population Genetic Algorithm for Solving the K-Partition Problem on Hyper-cubes." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 244–248. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.
8. Davidor, Yuval. "Analogous Crossover." *Proceedings of the Third International Conference on Genetic Algorithms*. 98–103. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.
9. Davidor, Yuval. "A Naturally Occurring Niche & Species Phenomenon: The Model and First Results." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 257–263. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.
10. Davis, Lawrence. *Handbook of Genetic Algorithms*. New York, New York: Van Nostrand Reinhold, 1991.
11. Deb, Kalyanmoy. *Genetic Algorithms in Multimodal Function Optimization*. MS thesis, University of Alabama Tuscaloosa, 1989.
12. Deb, Kalyanmoy and David E. Goldberg. "An Investigation of Niche and Species Formation in Genetic Function Optimization." *Proceedings of the Third International Conference on Genetic Algorithms*. 42–50. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.
13. DeCegama, Angel L. *Parallel Processing Architectures and VLSI Hardware*. Englewood Cliffs, NJ: Prentice Hall, Inc., 1989.
14. DeJong, Kenneth and William Spears. "On the State of Evolutionary Computation." *Proceedings of the 5th International Conference on Genetic Algorithms*. 618–623. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1993.
15. DeJong, Kenneth A. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.*. PhD dissertation, The University of Michigan, Ann Arbor MI, 1975.

16. Dorigo, Marco and Vittorio Maniezzo. "Parallel Genetic Algorithms: Introduction and Overview of Current Research." *Parallel Genetic Algorithms: Theory and Practice* edited by Joachim Stender, 5–42, Washington, DC: IOS Press, 1993.

17. Droddy, Vincent A. *Mission Route Planning Using Parallelized A* Search*. MS thesis, AFIT/GCS/ENG/93D-6, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1993.

18. Dymek, Capt Andrew. *An Examination of Hypercube Implementations of Genetic Algorithms*. MS thesis, AFIT/GCS/ENG/92M-02, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1992.

19. Filho, J. L. Riberio and P. Treleaven. "Genetic Algorithm Programming Environments." *Parallel Genetic Algorithms: Theory and Practice* edited by Joachim Stender, 65–83, Washington, DC: IOS Press, 1993.

20. Forrest, Stephanie, editor. *Proceedings of the Fifth International Conference on Genetic Algorithms*, San Mateo CA: Morgan Kaufmann Publishers, Inc., 1993.

21. Garey, M.R. and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Fransisco CA: W.H. Freeman and Company, 1979.

22. Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading MA: Addison–Wesley Publishing Company, 1989.

23. Goldberg, David E. "Zen and the Art of Genetic Algorithms." *Proceedings of the Third International Conference on Genetic Algorithms*. 80–85. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

24. Goldberg, David E., et al. "Genetic Algorithms, Noise, and Sizing of Populations," *Complex Systems*, *6*:333–362 (1992).

25. Goldberg, David E., et al. "Don't Worry, Be Messy." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 24–30. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

26. Goldberg, David E. and Robert Lingle. "Alleles, Loci, and the Traveling Salesman Problem." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. 154–159. Hillsdale NJ: Lawrence Erlbaum Associates, 1988.

27. Goldberg, David E. and Philip Segrest. "Finite Markov Chain Analysis of Genetic Algorithms." *Genetic Algorithms and Their Applications: Proceedings of the 2nd International Conference*. 1–8. Hillsdale NJ: Lawrence Erlbaum Associates, 1987.

28. Grefenstette, John J. *A User's Guide to Genesis*. Technical Report, Nashville TN: Vanderbilt University, 1986.

29. Grimm, Capt James J. *Solution to a Multicriteria Aircraft Routing Problem Utilizing Parallel Search Techniques*. MS thesis, AFIT/GCE/ENG/92D-04, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

30. Holland, John H. *Adaptation in Natural and Artificial Systems* (First mit press Edition). Cambridge, MA: MIT Press, 1992.

31. Homiafar, Abdollah, et al. "The N-Queens Problem and Genetic Algorithms." *Southeastcon Proceedings*. 262–267. New York, NY: IEEE Region 3, 1992.

32. Joe L. Blanton, Jr and Roger L. Wainwright. "Multiple Vehicle Routing with Time and Capacity Constraints using Genetic Algorithms." *Proceedings of the 5th International Conference on Genetic Algorithms*. 452–459. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1993.

33. Kargupta, Hillol, et al. *Ordering Genetic Algorithms and Deception*. Technical Report IlliGAL Report No. 92006, 117 Transportation Building, 104 South Mathews Avenue, Urbana, IL 61801: Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illinois at Urbana-Champaign, April 1992.

34. Klein, Philip. "On Gazit and Miller's parallel algorithm for planar separators: achieving greater efficiency through random sampling." *ACM Symposium on Parallel Algorithms and Architectures*. 43–49. ACM, 1993.

35. Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing*. Englewood Cliffs, NJ: Prentice Hall, 1992.

36. Merkle, Capt Laurence D. *Generalization and Parallelization of Messy Genetic Algorithms and Communication in Parallel Genetic Algorithms*. MS thesis, AFIT/GCE/ENG/92D-08, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

37. Moret, B.M.E and H.D. Shapiro. *Algorithms from P to NP*. Benjamin/Cummings Publishing Company, Inc., 1991.

38. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading MA: Addison–Wesley Publishing Company, 1985.

39. Rich, Elaine and Kevin Knight. *Artificial Intelligence* (2nd Edition). New York: McGraw Hill, 1991.

40. Rudeanu, Sergiv and Peter L. Hammer. *Boolean Methods in Operations Research and Related Areas*. New York: Springer-Verlag, 1968.

41. Sawyer, George Allen. *Extraction and Measurement of Multi-Level Parallelism in Production Systems*. MS thesis, AFIT/GCE/ENG/90D-04, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A230498).

42. Starkweather, T and others. "A Comparison of Genetic Sequencing Operators." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 69–76. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

43. Stender, Joachim, editor. *Parallel Genetic Algorithms: Theory and Practice*. Washington, DC: IOS Press, 1993.

44. Thangiah, Sam R, et al. "GIDEON: A Genetic Algorithm System for Vehicle Routing with Time Windows." *IEEE Conference on Artificial Intelligence for Applications*. 322–328. IEEE, 1991.

45. Thangiah, Sam R and Kendall E Nygard. "MICAH: A Genetic Algorithm System for Multi-Commodity Transshipment Problems." *IEEE Conference on Artificial Intelligence for Applications*. 240–246. IEEE, 1992.

46. Thangiah, Sam R and Kendall E Nygard. "School bus routing using genetic algorithms." *Proceedings of the SPIE Conference on Applications of Artificial Intelligence*. 387–398. Bellingham, Washington: SPIE, 1992.

47. Thangiah, Sam R and Kendall E Nygard. "Dynamic Trajectory Routing using an Adaptive Search Method." *ACM/SIGAPP Symposium on Applied Computing*. 131–138. ACM, 1993.

48. Thangiah, Sam R, et al. "Vehicle Routing with Time Deadlines using Genetic and Local Algorithms." *Proceedings of the Fifth International Conference on Genetic Algorithms*. 506–513. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1993.

49. Whitley, Darrell, et al. "Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator." *Proceedings of the Third International Conference on Genetic Algorithms*. 133–140. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

Vita

Captain James B. Olsan earned his bachelors degree in Computer Engineering from the University of Missouri - Columbia in 1987. He earned his commission through Air Force ROTC. Upon entering active duty in 1988, he was assigned as a Network Controller to the 6555th Aerospace Test Group, Cape Canaveral AFS, FL. He launched spacecraft on Space Shuttle, Titan, and Delta space launch boosters. He entered AFIT in 1992.

Permanent address: Merritt Island, FL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE
Genetic Algorithms Applied to a Mission Routing Problem

5. FUNDING NUMBERS

6. AUTHOR(S)
James B. Olsan, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/93D-12

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM Mathematics and Computer Sciences Directorate Air Force Office of Scientific Research Bolling AFB, Washington, D.C. 20332-6448

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis applies genetic algorithms to a mission routing problem.
The mission routing problem involves determining an aircraft's best route between a staging base and a target. The goal is to minimize the route distance and the exposure to radar. Potential routes are mapped to a 3-dimensional mesh where the nodes correspond to checkpoints in the route and the arcs correspond to partial paths of the route. Each arc is weighted with respect to distance and exposure to radar.
A genetic algorithm is a probabilistic search technique loosely based on theories of biological evolution. Genetic crossover and "survival of the fittest" are the basis of a genetic algorithm's control structure and can be used for general problems. Encoding and evaluation of the data structure is problem specific.
This thesis focuses on approaches to mapping the mission routing problem's mesh to a data structure which the genetic algorithm's control structure can effectively and efficiently manipulate. Three broad methods are proposed: Bounded Progress, Free Progress, and Restricted Progress. The Bounded Progress method uses a tightly-coupled mesh while the Free Progress method uses a loosely-coupled mesh. The Restricted Progress method is a hybrid of the other two methods.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Genetic Algorithms, Optimization, Parallel Processing	124
	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	UL