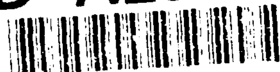
AD-A251 115



DTIC
S ELECTE D
JUN 4 1992
C

A HEURISTIC FOR DISCRETE SEARCH PROBLEMS WITH POSITIVE SWITCH COSTS

by

STEPHEN ROBERT RIESE

B.Arch., University of Notre Dame, 1982

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

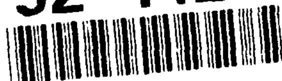
Department of Industrial Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

1992

92-14213



Approved by:

Major Professor

92 5

Statement A per telecon
Capt Jim Creighton TAPC/OPB-D
Alexandria, VA 22332-0411

NWW 6/3/92

Accession For
NTIS GRA&I ☒
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

A HEURISTIC FOR DISCRETE SEARCH PROBLEMS WITH POSITIVE SWITCH COSTS

by

STEPHEN ROBERT RIESE

B.Arch., University of Notre Dame, 1982

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Industrial Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1992

Approved by:

Major Professor

Table of Contents

List of Terms and Symbols iii

List of Tables and Figures iv

Acknowledgements vii

I INTRODUCTION

1.1 Introduction to the Problem 1-1

1.2 Investigation Strategy 1-3

II SEARCH THEORY

2.1 General 2-1

2.2 The Development of Search Theory 2-2

2.3 The Discrete Search Problem 2-8

2.4 Discrete Search with Switch Costs 2-15

III FORMAL STATEMENT OF THE PROBLEM

IV SOLUTION APPROACH AND COMPUTATIONS

4.1 The Heuristic 4-1

4.2 Methodology of Research 4-3

4.3 Computational Experiments 4-6

4.4 The Computer Model 4-12

V RESULTS AND DISCUSSION

5.1 Summary of Results 5-1

5.2 Distribution of Expected Costs 5-2

5.3 Determining a Measure of Performance 5-15

5.4 Behavior as a Function of Number of Cells . . 5-19

5.5 Behavior as a Function of Switch Costs . . . 5-21

5.6 Performance of the Heuristic 5-25

VI CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions 6-1

6.2 Recommendations for Future Research 6-2

VII REFERENCES AND BIBLIOGRAPHY

APP A COMPUTER SOURCE CODE

APP B ABSTRACT

Terms and Symbols used in this Thesis:

n : number of cells

p_i : probability that the target is in cell i

a_i : probability of overlooking the target, given that it is in cell i, and cell i is searched

b_i : probability of finding the target, given that it is in cell i, and cell i is searched ($b_i = 1 - a_i$)

c_i : cost of searching cell i

C_i : Cumulative cost of searching cells 1..i

m_{ji} : cost of moving from cell j to cell i

$M_{(i,k,s)}$: the number of times cell i is searched under policy s during the first k searches; see M_i

M_i : an abreviated form of $M_{(i,k,s)}$

s : a search policy, a sequence of cells ($s_1, s_2, s_3 \ldots$)

s' : a search policy constructed by a heuristic

s^* : the optimal search policy

R : a set of random policies

r : a random search policy

r' : from a group of random policies, the one with lowest expected cost

E[x] : the expected value of x

Tables and Figures in this Thesis:

Figure	Description	Page
1.1	Target Distribution in search for the lost nuclear submarine Scorpion.	1-4
2.1	A sample search problem	2-3
2.2	Why greedy algorithms do not always work . .	2-7
2.3	Four categories of search problems	2-8
2.4	Sample search problem with added parameters .	2-12
2.5	Computations of expected cost	2-14
2.6	Sample search problem with switch costs . . .	2-16
2.7	Two cell search problem	2-19
2.8	A hypothetical five cell problem	2-20
4.1	The concept of Tolerance	4-5
4.2	Description of Experiment 1	4-9
4.3	Description of Experiment 2	4-10
4.4	Description of Experiment 3	4-11
5.1	The truncated nature of the distribution of expected costs of policies	5-2
5.2	The behavior of the left tail in the distributions of expected costs	5-3
5.3	The gap between r' and s' as a function of n	5-4
5.4	Frequency diagram, $n = 5$, $m_{ji} = 0$	5-5
5.5	Frequency diagram, $n = 5$, $m_{ji} \in (1,4)$	5-5
5.6	Frequency diagram, $n = 5$, $m_{ji} \in (1,10)$	5-6
5.7	Frequency diagram, $n = 5$, $m_{ji} \in (90,100)$ . . .	5-6

5.8 Frequency diagram, n = 5, $m_{ji} \in (1,100)$ . . . 5-7

5.9 Frequency diagram, n = 10, $m_{ji} = 0$ 5-7

5.10 Frequency diagram, n = 10, $m_{ji} \in (1,4)$ 5-8

5.11 Frequency diagram, n = 10, $m_{ji} \in (1,10)$ . . . 5-8

5.12 Frequency diagram, n = 10, $m_{ji} \in (90,100)$ . . 5-9

5.13 Frequency diagram, n = 10, $m_{ji} \in (1,100)$ . . . 5-9

5.14 Frequency diagram, n = 25, $m_{ji} = 0$ 5-10

5.15 Frequency diagram, n = 25, $m_{ji} \in (1,4)$ 5-10

5.16 Frequency diagram, n = 25, $m_{ji} \in (1,10)$ . . . 5-11

5.17 Frequency diagram, n = 25, $m_{ji} \in (90,100)$ . . 5-11

5.18 Frequency diagram, n = 25, $m_{ji} \in (1,100)$ . . . 5-12

5.19 Frequency diagram, n = 50, $m_{ji} = 0$ 5-12

5.20 Frequency diagram, n = 50, $m_{ji} \in (1,4)$ 5-13

5.21 Frequency diagram, n = 50, $m_{ji} \in (1,10)$ . . . 5-13

5.22 Frequency diagram, n = 50, $m_{ji} \in (90,100)$ . . 5-14

5.23 Frequency diagram, n = 50, $m_{ji} \in (1,100)$ . . . 5-14

5.24 Example from Lossner and Wegener [1982] . . . 5-16

5.25 Estimating the Location Parameter, Gamma . . 5-17

5.26 Expected cost as a function of n 5-19

5.27 Number of searches as a function of n 5-20

5.28 Expected cost as a function of switch costs . 5-23

5.29 Number of searches as a function of switch costs 5-24

5.30 Performance of the heuristic as a function of switch costs 5-25

5.31 Performance of the heuristic as a function of n 5-26

5.32 Performance of the heuristic for 2-8 cells . 5-27

Acknowledgements

I express my appreciation for the efforts of all who have assisted me in this endeavor. My committee, Doctors Evangelos Triantaphyllou, John Boyer, and Shing Chang, was extremely helpful in providing me with both direction and understanding. My wonderful family was very understanding about the strange hours and odd habits that go along with graduate research. And, for granting me spiritual courage in the face of deadlines and my own shortcomings, I thank our Lord, Jesus, who provided his own, albeit paraphrased, example of search theory in action:

> *"And after two days of searching their caravan, Mary and Joseph could not find the Child; so they returned to Jerusalem, where they had last seen Him. After looking through the city, they heard of a child in the temple speaking with the priests and elders. Mary and Joseph went to the temple, where they found Jesus. 'Why have you done this to us?', they asked of Jesus. He replied: 'You did not employ an effective search strategy. You should have known that I would be doing the work of my Father with probability of .90. The other places you looked were unlikely to have produced an early success.' After this, Jesus accompanied his parents, and they never questioned Him about search theory again."*

[misquoted from Luke, c.70 A.D.]

This Thesis, as well as my graduate education, is fully funded by the United States Army as part of its Advanced Civil Schooling program.

I

INTRODUCTION

1.1 Introduction to the Problem.

A searcher is looking for a stationary target which is not attempting to avoid detection. The target is in one location of a finite set, I, of locations (cells), with an a priori probability p_i of being in cell i ($\sum p_i = 1, i \in I$). Associated with each cell is an overlook probability, a_i, which indicates the chance of not locating the target in cell i given that it is in cell i and cell i is searched. The complement of a_i is b_i, which is the probability of finding the target in cell i if it is in cell i and that cell is searched. The cost of searching cell i is c_i. Finally, there is a switch cost, m_{ji}, which represents the cost of moving to cell i from cell j between searches.

A search policy, s, is a sequence of cells in the order

they are to be searched $(s_1, s_2, ..., s_i, ...)$. A policy is optimal if the expected cost of finding the target is minimum, and is denoted s^*. When $m_{ji} > 0$, the problem of finding an optimal policy has been classified as NP-hard [Trummel and Weisinger, 1986]. Problems classified as NP-hard are considered very difficult in that a linear increase in the size of the problem (number of cells, for example) results in an exponential increase in the computational requirements to solve the problem. Our problem is to find s', a policy which is good, but not necessarily optimal.

When all switch costs are zero, the problem has a well documented and relatively straightforward solution. Black [1965], presents what is perhaps the most concise solution of the discrete search problem without switch costs. Matula [1964], gives a similar solution as well as conditions under which the optimal policy is **ultimately periodic** (u.p.). A policy which is u.p. begins with a transient portion T of finite length, followed by the periodic portion L, also of finite length. The portion L repeats for the remainder of the search. See also Section 2.3

The optimal policies for these problems without switch costs are interesting in that they always proceed by next searching the cell which has the highest ratio of probability of success to cost of search. Also, these policies are incrementally optimal as well as ultimately optimal. An **incrementally optimal** policy is one for which

$[s_1, s_2, ..., s_n]$ (where s_i is the cell searched on the i-th step of policy s), is optimal for all n. That is, it is optimal at each stage. An **ultimately optimal** policy is one that, when considered in its entirety, is optimal.

1.2 Research Strategy.

The computational complexity of the discrete search problem with switch costs is discussed in Chapter II. Lossner and Wegener [1982], present an optimal solution for this problem and a two-cell example. However, their method is computationally feasible only for small problems. There is a need for a method of finding a good, if not optimal, policy for much larger problems.

An example of the need to find search policies for larger problems is the 1968 search planning for the lost nuclear submarine Scorpion (See Figure 1.1). This problem has 175 cells, each with some probability of containing the target. The area where the submarine was last known to have been was divided into cells of about one square mile each. The number in each cell indicates the number of times that a computer simulation placed the submarine in that cell. The submarine was eventually found about 300 yards from the cell with the highest probability (location of the submarine is indicated with an asterisk).

An alternative to the computationally difficult optimal solution is to extend the solution for the problem without

NOTE: CONVERT NUMBERS TO PROBABILITIES BY DIVIDING BY 10,000. SHADING INDICATES MAGNITUDE AS FOLLOWS:

BLANK: 0 ≤ NUMBER ≤ 10
10 < NUMBER ≤ 100
100 < NUMBER < 1000
1000 < NUMBER ≤ 10,000

NOTE: * INDICATES LOCATION OF SCORPION.

			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O		
																5	7	1	
														3	11	14	24	6	
		1											5	26	35	22	26	9	1
		2										18	46	74	42	18	10	4	2
		3								8	60	140	99	45	20	4	2	1	1
		4	2	21	137	16	7	1	20	215	239	105	30	5	3	1	1	1	
		5 18	40	46	747	30	[illegible]	205	571	277	38	5	2	1	1	1			
	14	6 326	3	1	28	31	63	* 85	62	1	8	7	10	7	3	4			
1	359	7 175	174	[illegible]	282	245	82	71	65	35	27	9	12	6	5	4			
	24	8 25	42	82	297	230	129	115	61	33	14	14	10	6	2	5	1		
	17	9 25	20	20	20	19	55	99	46	30	14	15	3	5	1	6			
	2	10 13	14	25	20	24	45	34	27	19	15	5	7	5	5	1			
		11	7	13	12	9	1	3	3	11	14	5	4	3	2	1			
		12							1	4	4	10	5	4	1				
											1	3	2						
												3	2						

Figure 1.1. The initial target distribution for the 175 cells of the search for the U.S. nuclear submarine Scorpion. From Haley and Stone [1979].

switch costs to the one with switch costs in the form of a heuristic. Obviously, for zero switch costs, the heuristic is optimal. This heuristic is formally presented in Chapter IV.

To evaluate this heuristic, we need a measure of

accuracy. Since the optimal solution is generally too difficult to obtain, we will compare the policy constructed by the heuristic to the best of a very large sample of random policies. There are three parameters of interest for this study: the number of cells, range of search costs, and the range of switch costs.

The difficulty of search problems increases dramatically as the number of cells increases. As the number of cells increases, the number of possible search policies grows exponentially (explained in Chapter II). This makes it more difficult for a random policy to get close to the optimal. For this reason we will also test the heuristic with problems which involve different numbers of cells.

We expect the heuristic to perform well when the switch costs are low relative to the search costs. As the switch costs get larger we expect the heuristic to become less accurate. For this reason we will test the heuristic for problems with different ranges of switch costs, while the search costs remain fixed.

The form of the basic experiment is a 4x5 factorial with four different numbers of cells and five different ranges of switch costs. The search costs will remain fixed. One level of switch costs will be zero as a control. In the case of zero switch costs, the heuristic generates the optimal policy.

II

SEARCH THEORY

2.1 General.

The purpose of a search is to find something. We have all searched for something at some time: the keys to the car; a parking place; or a lost child. These simple examples have similarities to some important searches conducted recently: the search in 1966 for a lost hydrogen bomb near Spain; the 1968 search for a lost nuclear submarine near the Azores; the search in 1974 for underwater mines in the Suez Canal; and the search during the 1991 Gulf War for Iraqi SCUD missiles.

In all of the above examples, there was an object of the search (the *target*), different locations that the targets might occupy (the *locations*, or *cells*), probabilities associated with each cell that the target might be there(p_i), as well as other parameters that define the search problem. The

process of search planning is one of allocation of effort. The end product of this process is a *search policy*, that is, a sequence of locations to be searched.

Interestingly, the search for the lost car keys has parameters similar to those for the well defined problems; we usually just choose not to define the parameters before we look for lost keys. For example, common sense might tell us that the lost keys are probably in the drawer, where they are supposed to be (target = keys, location = drawer, p[drawer] = .75). The keys could also be in a coat pocket (location = coat, p[coat] = .2), or in the car itself (location = car, p[car] = .05). Given this simple example, our search policy would most likely start with: drawer, coat, car (see Figure 2.1).

Other parameters certainly come into play: the chance of not finding the target if it is in a location and you search in that location (*overlook probability*); the cost of searching a given cell (*search cost*); a cost of moving from one cell to another between searches (*switching or movement cost*); the possibility of a false target (*decoy*); the possibility of a *moving target*; and the possibility that the target may be trying to avoid detection (*a hider*).

2.2 The Development of Search Theory.

Search theory as an important part of Operations

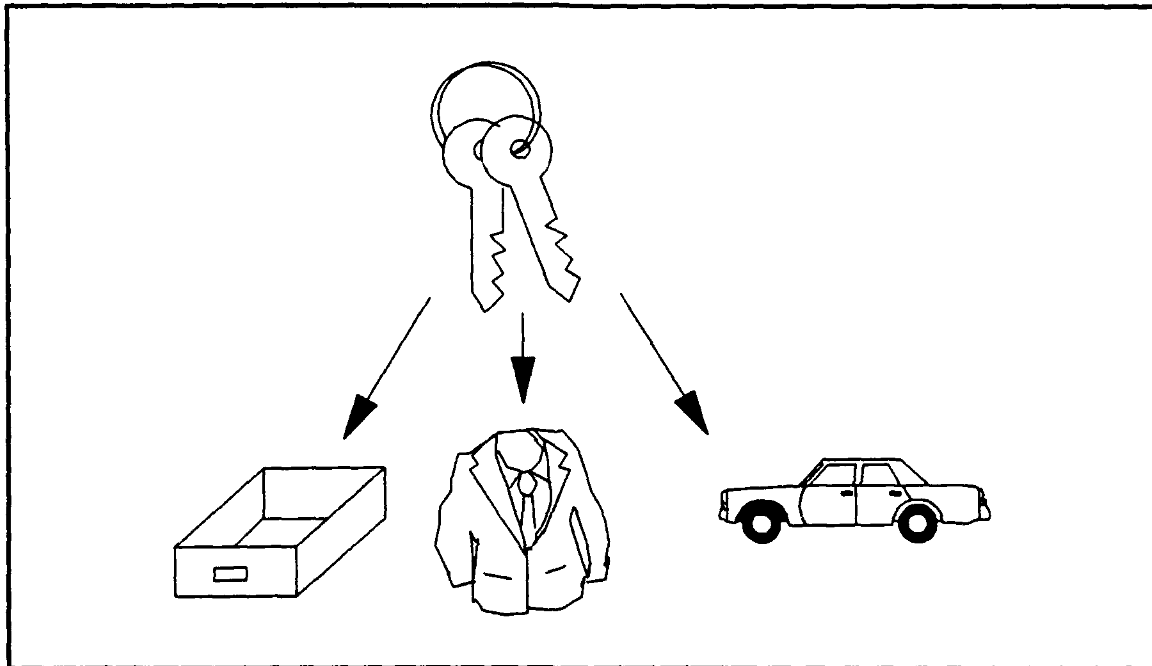


Figure 2.1. A sample search problem. The keys are known to be in one of three locations.

Research was born in the Anti-Submarine Warfare Operations Group (ASWORG) research during 1942-1945 in World War II. As outlined in Stone [1989], early efforts were directed to solving real problems with very immediate applications -- dealing with the German submarine threat.

Two texts are the cornerstones of search theory literature: *Search and Screening* by Bernard O. Koopman, [1946], revised in 1980, and *Theory of Optimal Search* by Lawrence D. Stone [1975]. Either one or both are listed as references in nearly all of the available literature.

Bernard Koopman was one of the key players in the ASWORG group during World War II. After the war the group became the Operations Research Group (ORG) and later the

Operations Evaluations Group (OEG). *Search and Screening* is one of three reports published by the OEG at the end of the war. It initially was classified, and thus was unavailable to the public. The book is written in a very applied manner and Koopman's association with the Navy is obvious from the technical terms and references throughout. In *Search and Screening*, Koopman defines the stage for search theory research. Virtually every aspect which might influence a searching operation is analyzed and presented in mathematical terms. These include many of the physical parameters of the equipment they had to work with such as the sweep width and sweep rate of a radar or the velocities of the searching craft and of the hider.

In 1956 and 1957 Koopman published a three part article in the *Operations Research* magazine which outlined much of the unclassified theory, [Koopman, 1956 and 1957]. While similar in content to parts of *Search and Screening*, the style of presentation was changed. The articles are more abstract and have similarities with most of the articles on search theory published since then. Most of the technical nautical terms are gone. With these articles, we can see search problems beginning to be divided into convenient categories.

Stone's *Theory of Optimal Search* presents most of the major advancements in search theory up to 1975. It deals primarily with stationary target problems and gives optimal

solutions for a wide variety of these problems. The work is mathematical in nature, heavy on proofs and light on examples. Stone's chapter on discrete search provides the optimal solution for our problem when there are no switch costs.

As indicated in Stone, 1989, search theory concerning stationary targets had reached a 'mature' state by about 1975, also the year in which Stone's book was published. By 'mature' he meant that most standard problems had been solved and that, in general, it would be difficult to significantly extend these results. This left the door open for search theory concerning moving targets. The literature since 1975 is concerned mostly with search problems involving moving targets.

The most significant result from the research in stationary target search was that, for a large group of problems, the optimal solutions are also incrementally optimal (optimal at each stage). To fit into this category, the problem must use **discrete search spaces** and have a **concave detection function.** Discrete search spaces, as opposed to a continuous search space, are well defined and independent of each other. A concave detection function is one in which the probability of detecting the target on a single search decreases the more often the cell is searched.

This means that in a discrete search for a stationary target with concave detection function, if we always search

the cell with highest ratio of probability of detection to cost then our overall policy will be optimal. This type of approach is known as *greedy*. Greedy means that at each stage the best option at that time is chosen. Greedy algorithms do not always give optimal results. Figure 2.2 gives an example of why we should be wary of greedy algorithms.

In 1964, David Matula extended the above finding to show that for a more restricted set of problems, that not only is the solution incrementally optimal but that it is also periodic after a determinable transient phase [Matula, 1964]. The restriction to guarantee periodicity concerns the overlook probabilities and is further discussed in section 2.3.

As with other areas of Operations Research, there are many facets of search theory. Many of the specific problems have existing optimal solutions which cannot generally be extended to similar problems. The problems divide naturally along two categories: whether or not the target is moving and whether or not the target is attempting to avoid detection. This natural division based on the behavior of the target is illustrated by Stone in [Chudnovsky and Chudnovsky, 1989] (See also Figure 2.3).

Whether a problem is one sided or two sided is determined by whether or not the target is trying to avoid detection. If the target is not an evading one, then the problem is one sided. That is, the target is either unaware

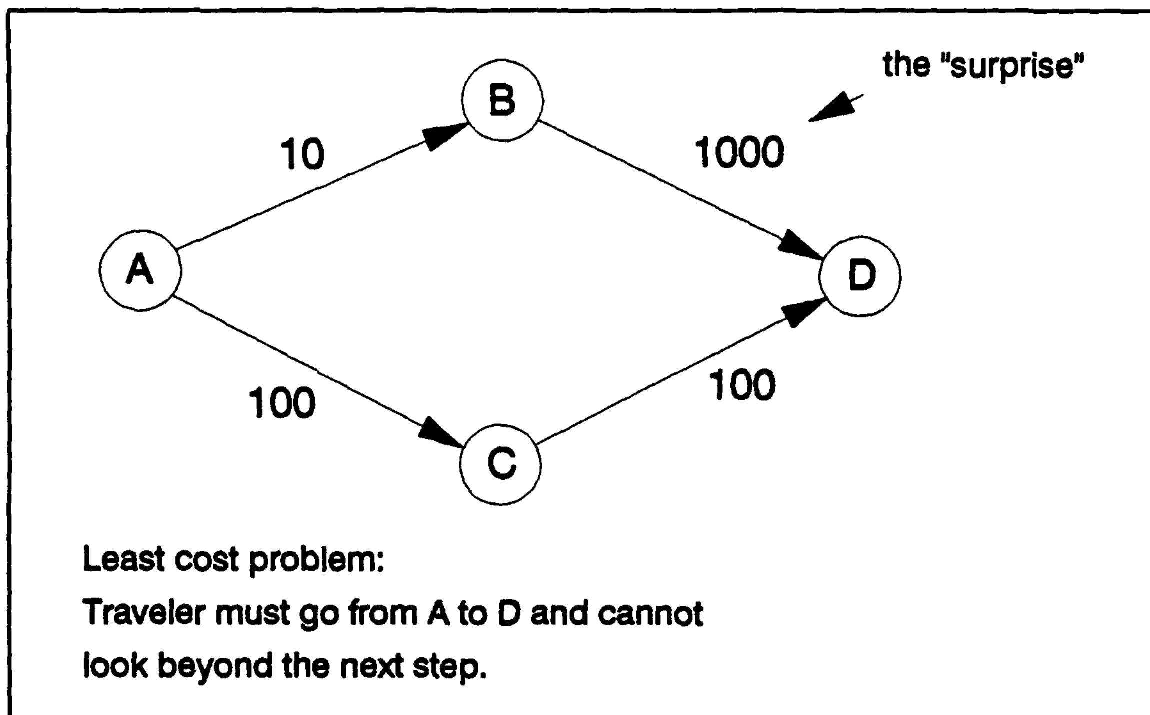


Figure 2.2. Why greedy algorithms don't always work.

of or unconcerned with the fact that he is being sought. An example here is of an energy company exploring for oil.

If the target is a conscious evader, the problem is a two sided one. In these problems the target (now referred to as a *hider* or *evader*) can position himself such that he minimizes the chance of detection. If the target is a non-moving evader (as in the case of a buried treasure) then the hider needs to choose only his initial hiding position. If the evader is moving (as in most military applications), then he may be able to update his position throughout the search process based on partial or complete information about the searcher.

Beyond these four general categories there are several

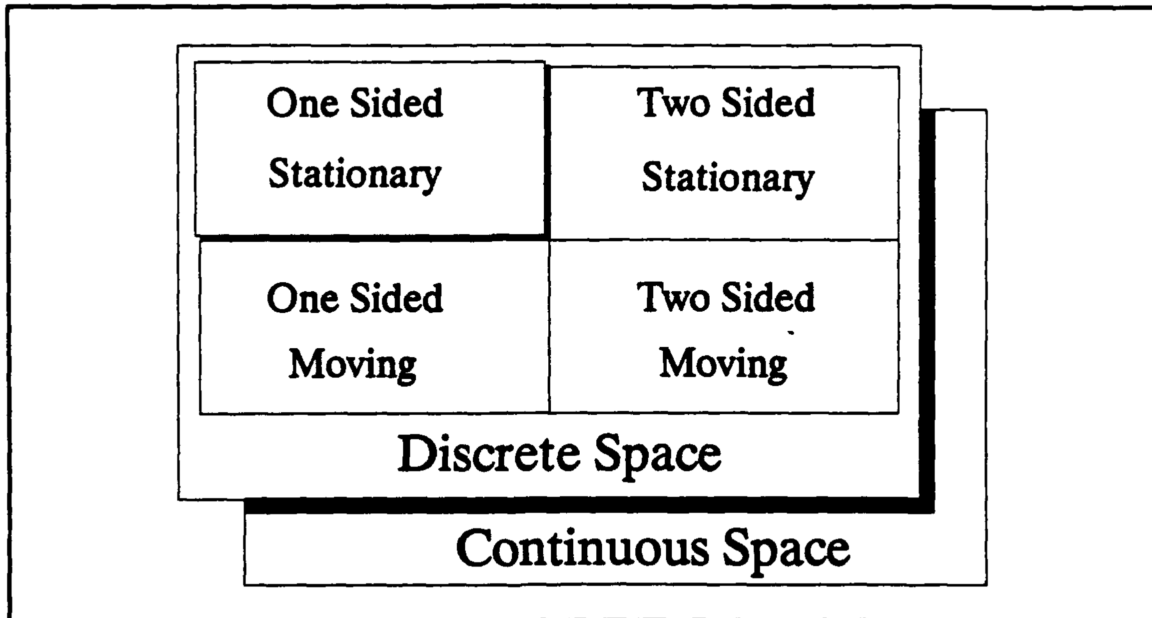


Figure 2.3. Four categories of search problems [Stone in Chudnovsky and Chudnovsky, 1989].

options which help define the problem. These include:

a. Discrete vs. continuous search effort.

b. Switch (movement) costs between searches.

c. Multiple targets.

d. False targets (decoys).

e. Limited resource searches.

Our search problem is a **one-sided, stationary target discrete problem** with a **single target** and **positive switch costs.**

2.3 The Discrete Search Problem.

In these problems discrete refers to how the search effort is applied to the problem. It is, perhaps easier to first consider the continuous search problem. Imagine a search for a submarine at sea. We could superimpose an

artificial grid over the ocean and apply our searches discretely to these cells. However, the more natural method considers the sea as a continuous field. We apply our searching effort (perhaps time of exposure to a passive radar), in any amount we choose and at any location we choose. The return from the search (amount of probability that the target is found), is in the form of an integral over the area swept by the radar and other parameters of the problem.

In a discrete search, we will apply our effort to cells, and the entire area within a cell is considered as one element. Associated with each cell is a search cost, and, if we search a cell, we pay the entire search cost. As we define them, the search costs do not change through time.

Discrete search problems generally have the following three common characteristics:

a) There is an *a priori probability distribution* of the target's location, designated as p. *A priori* here means that the probabilities are assigned by the searcher before the search begins, and are not adjusted during the search process. These probabilities may be determined by a computer model (as was the case for the missing nuclear submarine, [Richardson and Corwin in Haley and Stone, 1979]), experience, reconnaissance, seismic reports or some other method. In all cases, $\sum p_i = 1$.

b) There is a *decreasing detection function* which gives the probability of finding the target as a function of the searching effort applied to the cell. By decreasing, we mean that the more often a cell is searched, the less probable each subsequent search will be in locating the target. Thus, the first look in a cell is more likely to find the target in that cell than the second or third look in that cell.

The most common detection function is the exponential, although any concave function will work [Stone, 1975]. The form of this function in our problem involves two complimentary values. The first, a_i, is the overlook probability, or the probability that, if a target is in a cell, and you search that cell, you will not find the target. The second is b_i ($b_i = 1-a_i$), which is the chance of finding the target, given that the cell containing the target is searched. This function is generally seen as: $b_i a_i^{M_i}$, where M_i is the number of times that cell i has been previously searched. (See note on page iii about the abbreviated use of the symbol M_i.) This function is decreasing because M_i starts at zero and takes on increasing integer values, and $a_i < 1.00$.

c) A *cost* is associated with searching each cell, c_i. This cost might be time, money, or some other measure. In the case of searching for metallic nodules on the ocean

floor the cost depends on the depth at the point of the search [Haley and Stone, 1979]. In the search for the lost keys, the cost of searching the drawer might be high compared to the cost of searching the car. In looking through the drawer, the searcher might spend a couple minutes sorting through the contents, while a quick glance at the ignition switch will let the searcher know if he left the keys there.

In some discrete searches the cost depends on how many times the cell has been previously searched. Again, consider looking for the lost keys. The first search of the drawer might be a cursory one, the second search a more thorough one, and on the third search the entire contents are dumped on the table. In this case the search cost is a function of both the cell and of M_i, the number of times the cell has been previously searched. In our problem the search costs remain constant throughout the search process.

Figure 2.4 shows the lost keys example with the a, b, c, and p parameters added as well as the optimal policy and its cost. The values of p indicate the chance that the keys are in a particular location. These values all sum to one. The higher overlook probability for the drawer indicates that it would be more difficult to locate the keys there. The low overlook probability indicates that it would be difficult to overlook the keys if they were in the ignition.

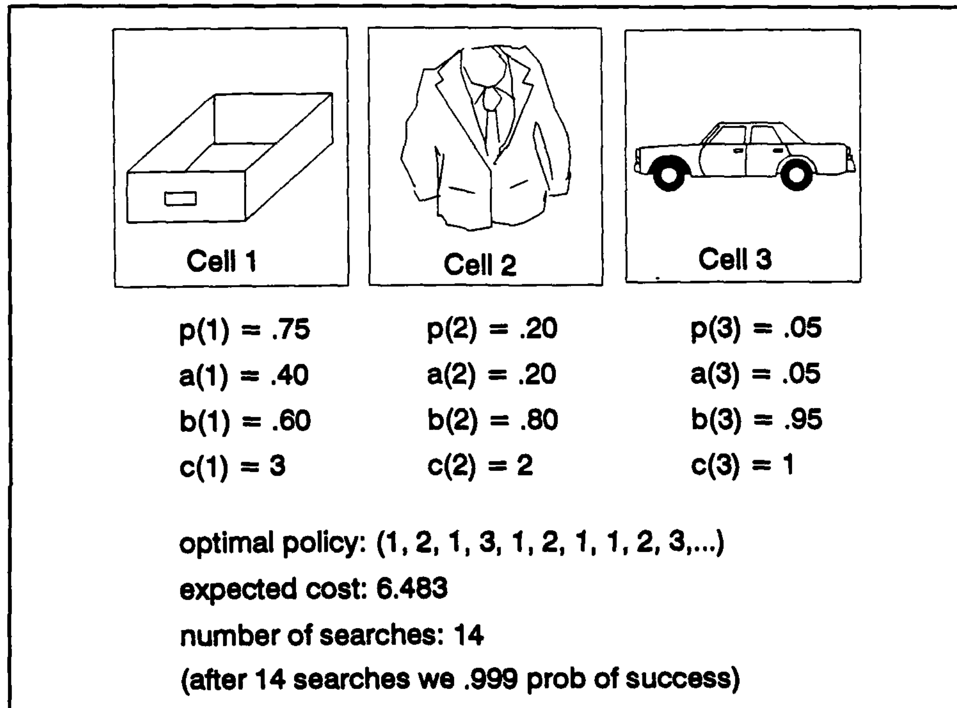


Figure 2.4 The sample search problem with some parameters added. Here there are no switch costs.

Also in Figure 2.4 is the beginning of the optimal policy. This policy means: "Search the drawer first. If that search is unsuccessful, then search the coat. If the search of the coat is unsuccessful, then search the drawer again, and so on..." The expected cost of this policy is 6.483 and it takes 14 searches to find the target with .9999 probability (tolerance = .0001; explained in Section 4.2).

Discrete search problems with these three parameters have straightforward solutions. They always proceed by next searching the cell for which the ratio $\frac{p_i b_i a_i^{M_i}}{c_i}$ is a maximum [Stone, 1975]. The ratio for each cell is computed at each

stage of the search. Since the detection function is decreasing, the ratio for a given cell decreases each time it is searched. Using this method, every cell will at some point become the best cell to search next [Stone, 1975].

There are **at least three interpretations of optimality** as it applies to search problems. The first, and the one we use here, is that an optimal search policy is the policy which has the **smallest expected cost** of locating the target. The second is that a policy is optimal if it **minimizes the number of searches** to locate a target. The third is that a policy is optimal if it returns the **highest probability** of finding the target within a limited budget. In problems in which c_i = a constant, the first two definitions are equivalent.

The expected cost of a policy is given by:

$$\sum_{k=1}^{\infty} C(k)\, p_i b_i a_i^{M_i}$$

where $C(k)$ is the cumulative costs of the first k searches and i represents the cell searched on the *k-th* search. How this sum is computed is shown for the first several stages of a search in Figure 2.5 (note that switch costs are included in this figure). The expected cost is discussed in Black, [1965]; Kadane in Haley and Stone, [1979]; Stone, [1975]; and others.

A policy that eventually finds the target is called

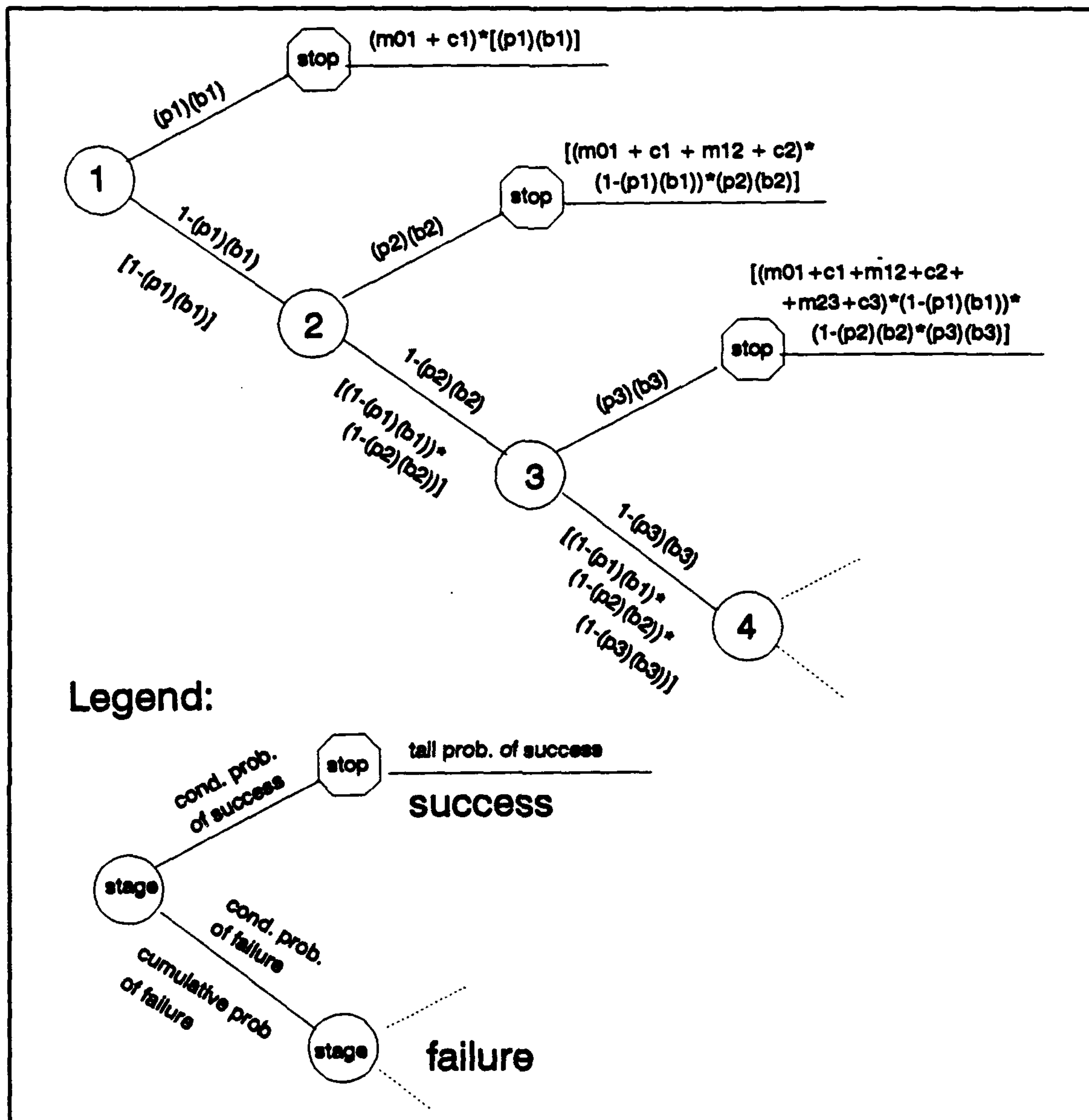


Figure 2.5 At each stage a search is either successful (stop) or unsuccessful (continue). The sum of the tail products from 1 to infinity is the expected cost of the policy.

successful. To guarantee that a policy is successful, each cell must be searched an infinite number of times. We assume that no p_i, a_i, b_i, or c_i has zero value. If a policy only searches a particular cell a finite number of times, then there remains some positive probability that the target

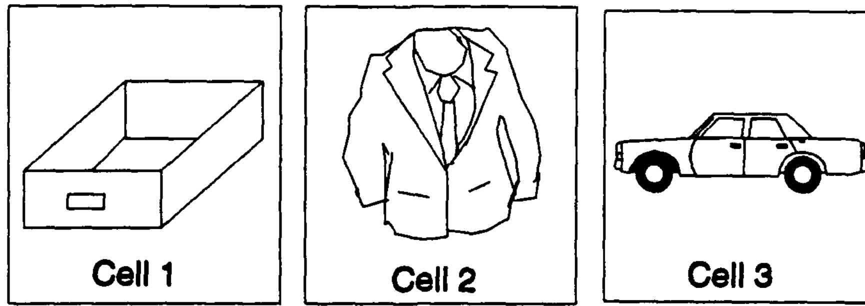
will not be found. Understandably, an optimal policy must be successful [Matula, 1964].

Matula, [1964], also presented a condition necessary and sufficient for an optimal discrete search policy to be ultimately periodic (u.p.). A periodic policy has a beginning transient portion (T) followed by a periodic sequence (L) which is then repeated until the target is found. The condition is that log a_i/log a_j must be rational for all $i,j \in I$. Furthermore, the lengths of the transient portion and of the periodic portion can be calculated from the problem parameters (a, b, c, p). This allows us to express an optimal policy of infinite length in finite form [Matula, 1964].

When positive switch costs are added to this discrete search problem, we no longer have a straightforward optimal solution. This is discussed in the next section.

2.4 Discrete Search with Positive Switch Costs

The addition of switch costs to the discrete search problem is a natural extension. For example, consider the search for deep sea oil. In addition to the cost of drilling at a location, there is a cost involved in moving from one location to another. In our lost keys example, time is consumed when moving between the three locations. These moving costs are known as switch costs. Figure 2.6 shows switch costs added to the lost keys example as well as



p(1) = .75	p(2) = .20	p(3) = .05
a(1) = .40	a(2) = .20	a(3) = .05
b(1) = .60	b(2) = .80	b(3) = .95
c(1) = 3	c(2) = 2	c(3) = 1

Initial moves:

m(0,1) = 1

m(0,2) = 1

m(0,3) = 4

switches:

from: \ to:	1	2	3
1	0	2	4
2	2	0	3
3	5	4	0

some policies with switch costs added:

type	policy	cost	searches
heuristic:	(1, 1, 2, 2, 1, 1, 3, 3, 1, 2,...)	9.458	14
random:	(1, 1, 3, 2, 1, 2, 3, 2, 2, 1,...)	11.343	27
random:	(2, 1, 1, 1, 3, 2, 2, 1, 1, 2,...)	10.715	18
random:	(1, 3, 3, 1, 1, 1, 3, 3, 2, 3,...)	16.583	32

(note: the heuristic is defined in section 4.1)

Figure 2.6. The sample search problem with switch costs.

several searching policies. The method used to generate the heuristic policy is described in section 4.1.

Very little has been published concerning search problems with switch costs. Gilbert [1959], is the first to discuss these costs. He presents strategies and examples for both zero and nonzero switch costs. These examples, however, are limited to two cells and are continuous, not discrete, as is our problem.

Onaga [1971], presented a solution for a positive switch cost problem in which the time parameter (search cost) is continuous. Although the target is confined to a cell, this is not a discrete search problem. In Onaga's problem the switch costs depend only on the destination cell, and not the cell last visited.

In 1986, Trummel and Weisinger discussed the complexity of problems very similar to ours. The problems are restricted in that a searcher can only move to adjacent cells between searches. They consider two cases: a finite horizon where effectiveness is measured in level of probability of detection; and an infinite horizon where effectiveness is measured in the expected cost of finding the target (our case). Their main result is that the first problem is NP-complete while the second problem is NP-hard. Wegener [1982], identified our switch cost problem as NP-hard. The computational requirements for NP-hard and NP-complete problems increases exponentially as the size of

the problem increases linearly and, in general, no simple solution exists. The group of NP-complete problems are interesting in that if a simple solution is found for one NP-complete problem, then all can be solved using the same solution.

Udo Lossner and Ido Wegener [1982] published an optimal solution to our problem. They show that, for the same conditions introduced by Matula, an optimal policy for this problem is also ultimately periodic. They give methods to calculate bounds on T and L and bounds on the number of times each cell appears in T and L.

The only general constraint on these periods is that the periodic element L must contain every cell at least once. If some cell were omitted from L, that cell would only be searched a finite number of times (the number of appearances in T). This violates the requirement that a policy visits each cell an infinite number of times for the policy to be successful.

Once the bounds for the number of appearances of each cell in T and L are determined, all possible policies that fit within these bounds are generated and compared against each other. For the two cell example given in the article, this results in a total of 96 policies generated and compared (See Figure 2.7). The number of policies to generate and compare grows exponentially as the number of cells in the problem grows.

Two Cell Example
of a Discrete Search with Switch Costs
[Lossner and Wegener, 1982]

$p_1 = 2/3$	$a_1 = 1/16$	$b_1 = 15/16$	$c_1 = 3$	$m_{01} = m_{21} = 4$
$p_2 = 1/3$	$a_2 = 1/2$	$b_2 = 1/2$	$c_2 = 2$	$m_{02} = m_{12} = 1$

$\log a_1/\log a_2 = 4$ is rational (therefore, $\log a_2/\log a_1$ is also rational); thus an ultimately periodic optimal policy exists.

Two cases must be considered: $s_1 = 1$ and $s_1 = 2$. By applications of several theorems, T and L are bounded for $s_1 = 1$ as follows:

Phase	Cell	Visits this Phase	Possible Choices
T	1	(1, 2)	2
	2	(2..9)	8
L	1	(1, 2)	2
	2	4 x L1	1
		Strategies to Compare:	32

For the case $s_1 = 2$, several applications of the various theorems yields the following bounds:

Phase	Cell	Visits this Phase	Possible Choices
T	1	(1, 2)	2
	2	(1..1)	2
L	1	(2, 9)	8
	2	(1, 2)	2
		Strategies to Compare:	64

These 96 possible policies are constructed and compared with the final result:

$T = 6 \quad L = 10$

$s^*_1 .. s^*_{16} = (1,2,2,2,2,2,1,1,2,2,2,2,2,2,2,2,2)$

Note that s^* is not optimal if switch costs are zero.

Figure 2.7. A two cell discrete problem with switch costs from Lossner and Wegener [1982].

Estimate for Five Cell Problems

This are hypothetical numbers based on an extrapolation of the Lossner/Wegener example problem.

First Iteration

Phase	Cell	Visits this Phase	Possible Choices
T	1	(1, 2)	2
	2	(2,..4)	3
	3	(1, 2)	2
	4	(3,..9)	7
	5	(1, 2)	2
L	1	(2,..4)	3
	2	(1, 2)	2
	3	(1, 2)	2
	4	(2,..6)	5
	5	(1,..3)	3

Policies to Compare: 30,240
(product of above)

Second Iteration

Phase	Cell	Visits this Phase	Possible Choices
T	1	(1, 2)	2
	2	(1, 1)	1
	3	(1,..4)	3
	4	(3,..9)	7
	5	(2,..4)	3
L	1	(1, 2)	2
	2	(1,..4)	5
	3	(1, 2)	2
	4	(2,..9)	8
	5	(1,..5)	5

Policies to Compare: 100,800
(product of above)

Total policies to generate = 30,240 (1st iter.)
+ 100,800 (2nd iter.)
= 131,040

Figure 2.8. Estimate of the number of policies to generate and compare using the Lossner/Wegener optimal procedure.

To see how complex the problem is, we assume that the range of the bounds remains the same for each cell as in the example problem (there is no indication that they behave otherwise). A hypothetical five cell problem requires over 100,000 policies to be generated and compared (see Figure 2.8). Continuing this growth, a ten cell problem would then require that over 10 billion policies be generated and the 175 cell problem would require over 1×10^{80} policies to be generated. At one million comparisons per second, this 175 cell problem would require several times the believed age of the universe to complete!

This complexity provides the motivation to find a solution which, while not guaranteed to be optimal, can produce good results quickly.

III

FORMAL STATEMENT OF THE PROBLEM

An object is located in one of a finite number of possibly noncontiguous, well defined, cells. An a priori probability distribution of the target within the cells is either known or assumed by a searcher. The target is not attempting to avoid detection and may not move while the search proceeds. Associated with each cell is:

a) a chance that if the target is in that cell, a search of that cell will overlook the target

b) a chance that if the target is in that cell, a search of that cell will locate the target

c) a cost for searching that cell

d) a cost involved in switching from one cell to another between searches.

The effort of the searcher is applied in discrete units

(a cell may not be partially searched). The search costs, switch costs, and the a priori distribution do not change throughout the search. The chance of locating the target, given that it is in the cell being searched (the detection function), is a decreasing function. That is, the more often a cell is searched unsuccessfully, the less likely that an additional search will result in finding the target.

The problem is to construct a search policy, a sequence of cells, which has a very low expected cost. If the policy has minimal expected cost then it is optimal.

The following notation is used:

n : number of cells

p_i : probability that the target is in cell i

a_i : probability of overlooking the target, given that it is in cell i, and cell i is searched

b_i : probability of finding the target, given that it is in cell i, and cell i is searched ($b_i = 1 - a_i$)

c_i : cost of searching cell i

C_k : Cumulative cost of searching cells $1..k$

m_{ji} : cost of moving from cell j to cell i

M_i : the number of times cell i has been previously searched (see also p.iii)

s : a search policy, a sequence of cells $(s_1, s_2, \ldots, s_k, \ldots)$

s' : a search policy constructed by the proposed heuristic

s^* : the optimal search policy

R : a set of random policies

r : a random search policy

r' : from R, the one with lowest cost

IV

SOLUTION APPROACH AND COMPUTATIONS

4.1 The Heuristic.

The complexity of finding an optimal solution (Chapter II and Trummel and Weisinger [1986]) motivates us to find a policy which is good, but not guaranteed to be optimal, and can be determined quickly. Our heuristic has the same complexity as the optimal solution for discrete problems without switching costs (complexity is n^2 where n is the number of cells). That is, the number of computations increases quadratically as n increases linearly.

The following rule constructs an optimal policy when $m_{ji} = 0$ for all cells: At each stage of the search, choose, as the next cell to search, the cell for which the ratio $\frac{p_i b_i a_i^{M_i}}{c_i}$ is a maximum. This type of policy is a greedy policy because at each stage it chooses the 'best' at that stage.

Switch costs are an extension to search problems without such costs. Thus, a natural source for our heuristic is the above rule for problems without switch costs. By adding the switch costs to the denominator, we maintain the 'benefit/cost' aspect of the rule. Our heuristic is then: At each stage of the search, choose, as the next cell to search, the cell for which the ratio $\frac{p_i\, b_i\, a_i^{M_i}}{c_i + m_{ji}}$ is a maximum. This is also a greedy policy, however, it is not guaranteed to be optimal.

The motivation for using this rule as the heuristic is that it is an extension of the optimal rule for discrete search without switch costs. Since no heuristics have been previously published for this problem, this extension is a natural place to start. In Chapter VII, we recommend some ideas for future research that build upon these results.

The heuristic has the same time complexity as the optimal solution for the zero movement cost problem. At each stage the number of 'benefit/cost' ratios to calculate equals n, the number of cells. The addition of movement costs to the problem does not directly affect the length of the search (number of stages until the target is found). It may increase or decrease the length of the search because the policies for two problems -- one with and one without movement costs, but otherwise identical -- are generally different. Because we are using expected cost as the measure of effectiveness, and not number of searches, we may

find policies which can find the target faster but have higher expected cost.

4.2 Methodology of Research.

Since we are suggesting a heuristic to construct search policies, we need some measure of how good the heuristic is. The optimal value, as discussed earlier, is too difficult to determine even for problems with very few cells. However we can generate a large number of random policies in very short (linear) time, and graph their expected costs on a frequency plot. When the expected cost of our heuristic (denoted as s'), is placed on the frequency plot with these random policies we will be able to see where, in the distribution of all possible policies, s' lies.

Since the heuristic is derived from an optimal method, we naturally expect for the heuristic to perform best when the conditions of the problem are close to the conditions under which the optimal method holds (ie, no switch costs). For example, the heuristic should perform better when $m_{ji} \epsilon$ (1,4) than when $m_{ji} \epsilon$ (90, 100). The reason for this is that large values of m_{ji} will tend to dominate small values of c_i. Since it is the switch costs which cause the greedy algorithm to not be optimal, the more dominant these become, the less effective we expect the heuristic to be. For this reason we will look at several ranges of switch costs.

Search problems become more complex as the number of cells increase. We are interested in the behavior of search problems and the heuristic for different numbers of cells, so we will look at several different numbers of cells.

The measure of effectiveness we have selected, expected cost of finding the target, requires an infinite horizon. Since this is impractical for computations, we introduce a *tolerance*, or threshold, to be used in evaluating policies. The tolerance is the probability that our policy will not find the target when all searches have been completed. (1-tolerance is the probability that we have found the target). By specifying a very low tolerance (.0001, for example), the expected cost we calculate will be very close to the true value. See Figure 4.1.

A more obvious, but incorrect, method to allow us to move from an infinite to a finite horizon would be to limit the number of stages (total number of searches, or the length of a policy). Why this method will not work is seen when we try to compare the expected costs of policies against each other. A good policy may be penalized while a poor policy may be unfairly rewarded when evaluating both using the same horizon.

To see this unfairness, consider a search problem in which we limit the total number of searches to 20. We generate policies using the heuristic and at random. Suppose that after the 20 searches, the heuristic policy

The Concept of Tolerance

This shows calculations on the sample problem from Chapter II (the lost keys).

stage	cell	c_i	$p_i b_i a_i^M$	cum p(success)	cum cost
1	1	3	.45	.45	1.35
2	2	2	.16	.61	2.15
3	1	3	.18	.79	3.59
4	3	1	.0475	.8375	4.0175
5	1	3	.0720	.9095	4.8815
6	2	2	.0320	.9415	5.3295
7	1	3	.0288	.9703	5.8191
8	1	3	.0115	.9818	6.0495
9	2	2	.0064	.9882	6.1903
10	3	1	.0024	.9906	6.2449
11	1	3	.0046	.9952	6.3647
12	2	2	.0013	.9965	6.4006
13	1	3	.0018	.9983	6.4577
14	1	3	.0007	.9991	6.4828

When cum p(success) > (1 - tolerance) = .999 we consider the cumulative cost to be a close approximation of the expected cost.

Figure 4.1. How tolerance is used to approximate an infinite sum with a finite sum.

would have found the target with .95 probability with an expected cost of 10. Suppose that the random policy, also after 20 searches, would have found the target with .65 probability with an expected cost of 5. If we say that the random policy is better because of the lower expected cost then we wrongly penalize the heuristic for choosing the higher probability cells.

In using the tolerance to allow us to evaluate policies, we avoid the previous unfairness. Therefore, all policies are carried out to the same degree of success. The tolerance allows us to approximate the infinite horizon.

The lower the tolerance, the closer the results are to the results from an infinite horizon.

4.3 Computational Experiments.

Computational results will be gathered from **three main types of experiments**:

1. Experiments to **determine the distribution** of possible values for the expected cost of search policies.

2. Experiments to **explore the behavior** of the expected cost and number of searches required as functions of n and m_{ji}.

3. Experiments with very large numbers of random policies to try to **get an estimate of the performance** of the heuristic.

While it is theoretically not necessary to run three separate experiments to gather the required data, it is necessary from a computational standpoint. In some cases we will generate results which stretch the abilities of the spreadsheet we use to analyze and plot the information. In each experiment we are looking at certain aspects of search problems and it is more convenient to run separate experiments than to run one massive experiment.

Some aspects are common to all experiments. For each value of n, the parameters p, a and c are generated at random following these rules:

a) $p_i < 1.00$

b) $\sum_{i=1}^{n} p_i = 1.00$

c) $a_i \leq .50$

d) $b_i = 1 - a_i$

e) $c_i \in [1, 10]$

These parameters are held constant while the switch cost ranges are varied (see specific experiment designs). For each n-m_{ji} combination, the m_{ji} parameters are generated at random within the specified range. Once these parameters are set for each of the tests, the heuristic policy is determined. The random policies are then generated and their expected costs computed, and the best random policy, r', is found.

The parameters are all drawn randomly from a uniform distribution within their specified range, with one exception. Because the target location probabilities, p, must sum to one, they are adjusted. A multiplier is introduced which keeps the sum of the probabilities equal to one.

The **first experiment** explores the shape of the distribution of expected costs of search policies. To do this we use a 4x5 factorial design with four levels of n, the number of cells, and 5 levels of switch cost ranges (See Figure 4.2). The range of the search costs is fixed because the absolute value of these costs does not have an impact on

the resulting policy. It is the relative value of the search costs, as compared to the switch costs, which has an impact. Since we need to only vary one of these in order to achieve the relative differences, we choose to adjust the switch costs because they are the main reason the problem is difficult to begin with.

The values for the numbers of cells are chosen to show how the problems and the heuristic behaves for small through relatively large size problems. We chose not to include larger numbers of cells (100+) because pilot tests indicated no significant additional information gained at the expense of more computational time.

An additional factor to consider in choosing numbers of cells is the average value of p_i. Since $\sum_{i=1}^{n} p_i = 1.00$, there is only 100% to divide among the p_i's. Problems with small n will have a higher average p_i. As n gets large, the average p_i gets small and the random policies become less effective as a measure of heuristic performance. The random policies become less effective because there is more likelihood that cells with low p_i get chosen at random.

The ranges of switch costs are chosen to have one with mean below the search costs: (1-4); one with the same range as the search costs: (1-10); one with mean much greater than the search costs: (90-100); and one with a broad range that encompasses the search costs: (1-100). Additionally, one

Design of Experiment 1

Design Parameters:

No. Cells	Search Costs	Switch Costs
n: 5	c_i: 1-10	m_{ji}: 0 (control)
10		1-4
25		1-10
50		1-100
		90-100

Replications: 1

Random Policies per Test: 10,000

Tolerance: .00001

Figure 4.2. Design of Experiment 1.

range of switch costs is set to zero as a control, since the heuristic is known to be optimal for these problems.

In the <u>second experiment</u> we are concerned with the expected cost and number of searches required as functions of n and of m_{ji}. This experiment is similar to the first one except that we generate fewer random policies per problem and that we also collect information about the number of searches. This tradeoff is an example of the computational limitations mentioned earlier. The factors and their levels remain the same as in the first experiment, however not all combinations are run (see Figure 4.3).

The <u>third experiment</u> is described in Figure 4.4. Here, we are after a good estimate of the performance of the heuristic. This means estimating s^* as closely as possible,

Design of Experiment 2

Part a: vary n, hold m_{ji} fixed at 0.

No. Cells	Search Costs	Switch Costs
n: 5	c_i: 1-10	m_{ji}: 0 (control)
10		
25		
50		

Part b: vary m_{ji} hold n fixed at 5.

No. Cells	Search Costs	Switch Costs
n: 5	c_i: 1-10	m_{ji}: 0 (control)
		1-4
		1-10
		1-100
		90-100

Replications: 1

Random Policies per Test: 2000

Tolerance: .00001

Responses of Interest: Expected Cost
Number of Searches

Figure 4.3. Design of Experiment 2.

which in turn means a very large number of random policies per problem. One million random policies will be generated and evaluated for each combination of factor-levels. For small n this is sufficient to almost guarantee that we get very close to, if not exactly, the optimal. For large n, however, one million is still a small fraction of the possible policies.

Because the chance of producing near optimal results is greater for small n, we add four additional levels of n,

Design of Experiment 3

Design Parameters:

No. Cells	Search Costs	Switch Costs
n: 2	c_i: 1-10	m_{ji}: 0 (control)
3		1-4
4		1-10
5		1-100
8		90-100
10		
25		
50		

Replications: 4

Random Policies per Test: 1,000,000

Tolerance: .001

Response of Interest: Expected Cost

Figure 4.4. Design of Experiment 3.

(2, 3, 4, 8), to the four from the previous experiment, (5, 10, 25, 50). The levels of m_{ji} will remain the same. To help reduce variance introduced by the randomness of the problems themselves, we will perform 4 replications of the experiment (chosen because of the number of available computers).

For this third experiment we lower the tolerance to .001 (from .00001 in the other two experiments). This lowering of the tolerance allows us to conduct this experiment in a reasonable period of time without significantly affecting the outcome of expected cost. Lowering the tolerance has a big impact on the number of

Lowering the tolerance has a big impact on the number of searches, but not on the expected cost.

4.4 The Computer Model.

The source code for the computer model is in Appendix A. It consists of two main programs, Search_1 and Search_2, and a menu unit. The programs are written in Turbo Pascal 6.0 for the IBM PC and compatibles. Where possible, the same notation is used in the program as is used in the text.

Search_1 is an interactive program where the user can change problem parameters and observe the heuristic's performance against random policies. The user can change the following parameters: number of cells, search cost range, move cost range, number of random policies generated and number of different problems to generate with the same parameters. There is also an output option which allows the user to put the results into a text file.

Search_2 is the program which generated the main data used here. It is not interactive, but, if the user has a Turbo Pascal compiler, the program can be altered to generate data for problems with different parameters.

The calculations are, for the most part, straightforward. The one part which may seem odd is that the expected value for a policy is calculated incrementally rather than all at once after the policy is created. This is done with the same reasoning that caused us to introduce

the tolerance earlier. Since we want to compare all policies against the same standard, they must be evaluated until they meet tolerance (in this case 0.00001). For some policies this is achieved quickly while others require over a thousand searches.

For this reason, the expected cost of a policy is updated as each new cell to be searched is chosen (by the rule for the heuristic policy or at random for the random policies). As the expected cost is updated, the unconditional probability of success for searching the current cell is subtracted from the running total. When this total becomes less than the tolerance, the search stops and the current value of the expected cost of finding the target is used to approximate the true expected cost. Separate experiments have shown that the difference between the estimate and the true expected cost is, at most, in the tenth significant digit.

The program Search_2 was used to generate the results in the three experiments. Slight alterations concerning the parameters and output options were made for the different experiments. The rest of the code remains the same throughout the experiments.

V

RESULTS AND DISCUSSION

5.1 Summary of Results.

The discussion of the results from the three main experiments follow these four steps:

a. The results from experiment 1 are plotted to determine characteristics of the **distributions of expected costs** (Section 5.2).

b. Insight from the previous step is used to form a measure of the **performance of the heuristic**, that is, a way of quantifying the heuristic's effectiveness (Section 5.3).

c. The results of experiment 2 are used to determine the **behavior of search problems** as functions of n and of the range of m_{ji} (Sections 5.4 and 5.5).

d. The results of experiment 3 are used, along with observations from the other results, to **estimate the performance of the heuristic** (Section 5.6).

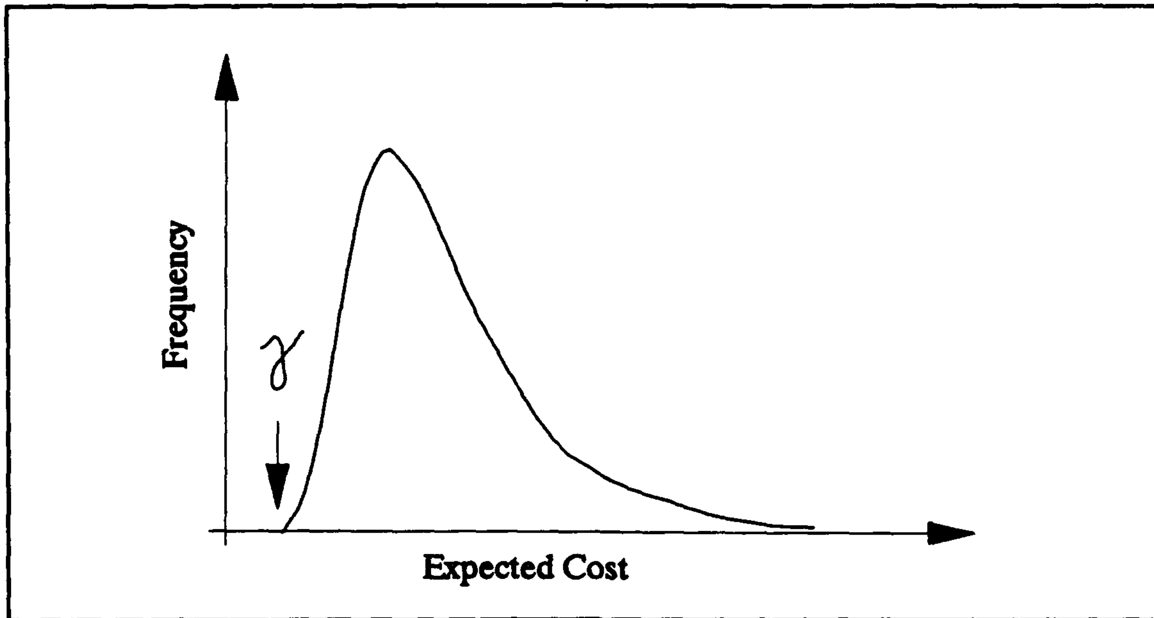


Figure 5.1. Diagram showing the truncated nature of these distributions.

5.2 Distribution of Expected Costs.

The primary observation from the histograms, Figures 5.4 through 5.23, is that the distribution of expected costs is mound shaped and reasonably well behaved. Because there exists a nonzero expected cost which is optimal, the distribution is also truncated. That is, there is zero probability of having an expected cost of between zero and s^* (see Figure 5.1). For these discussions, s^*, s' and r' will refer to expected costs and to not the policies themselves.

The distributions are noticeably skewed to the right. In all cases the right tail approaches the x-axis gradually, while it is difficult to discern a behavior for the left tail. For small n the distribution drops off abruptly on the left. As n gets larger, the left approach to the x-axis

is less abrupt and the distributions less skewed. For small n, the value of s' is close to r'. In two cases, Figures 5.5 and 5.8, $r' < s'$. As n gets large the gap $r' - s'$ grows.

The two observations in the preceding paragraph are shown graphically in Figure 5.2. The importance of Figure 5.2 is that the position of s' in the distribution is consistent with where we would expect to find s^*. The four control tests ($m_{ji} = 0$), confirm that s^* behaves in the same manner as s'. That is, both s' and s^* are at the extreme left end of the distribution, near or at the point of truncation.

The gap, $r' - s'$, mentioned above is charted in Figure 5.3. This gap behaves consistently with the generalizations about the shape of the distributions given above. When the

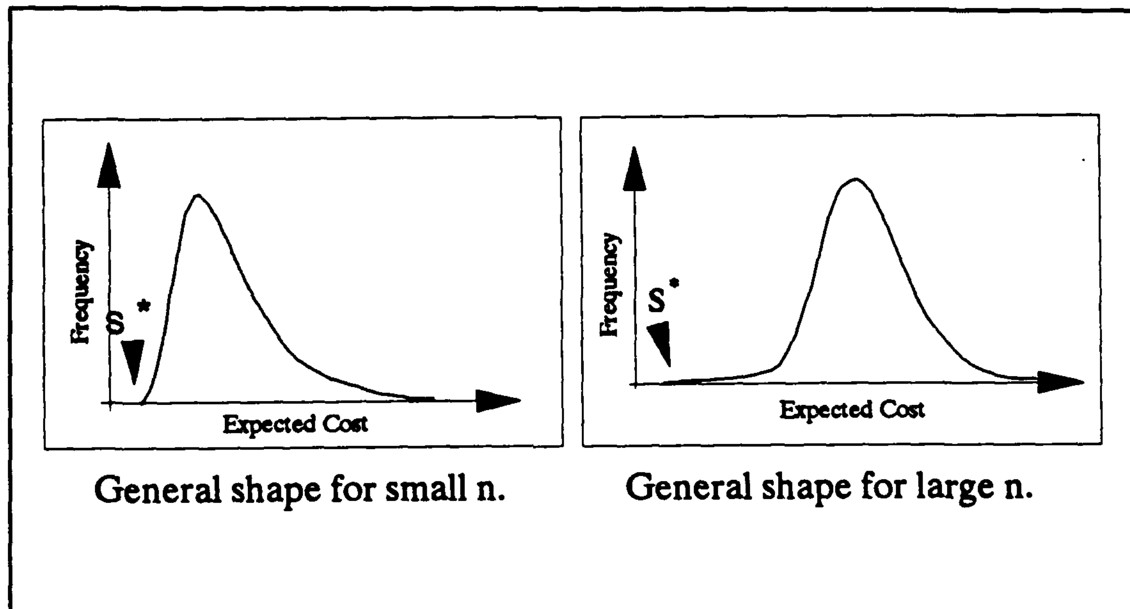


Figure 5.2. The behavior of the left tail in the distributions of expected costs.

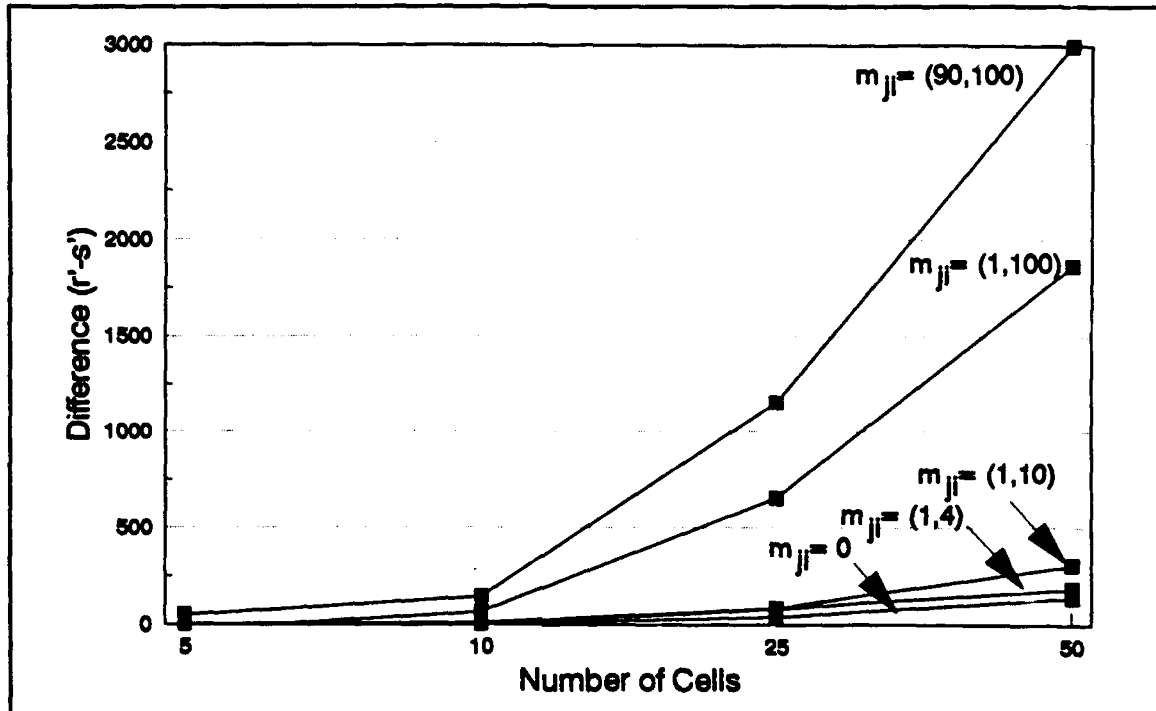


Figure 5.3. The gap between r' and s'.

left tail descends sharply, there is a sizeable percentage of the distribution "close" to the x-intercept. Random policies should, therefore, be easy to generate near to this point. However, when the left tail becomes more asymptotical, it is more difficult for random policies to occur in that neighborhood.

From these frequency diagrams, we see that the heuristic is good because its expected cost is lower than the best random policy in 18 of 20 tests, each with 10,000 random policies. We see that the heuristic is not always optimal, because at least two policies were found that had a lower expected cost. Additionally, the distributions are well behaved and the test results are consistent with the control runs.

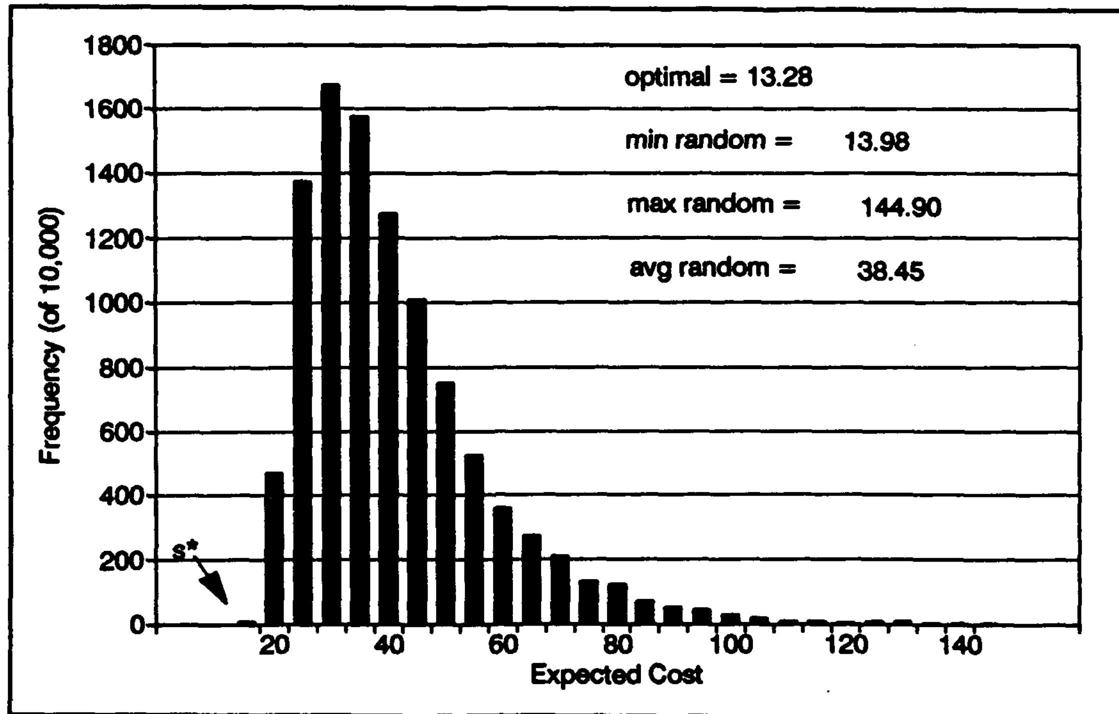


Figure 5.4. Frequency diagram for a problem with 5 cells, search costs 1-10, and no switch costs.

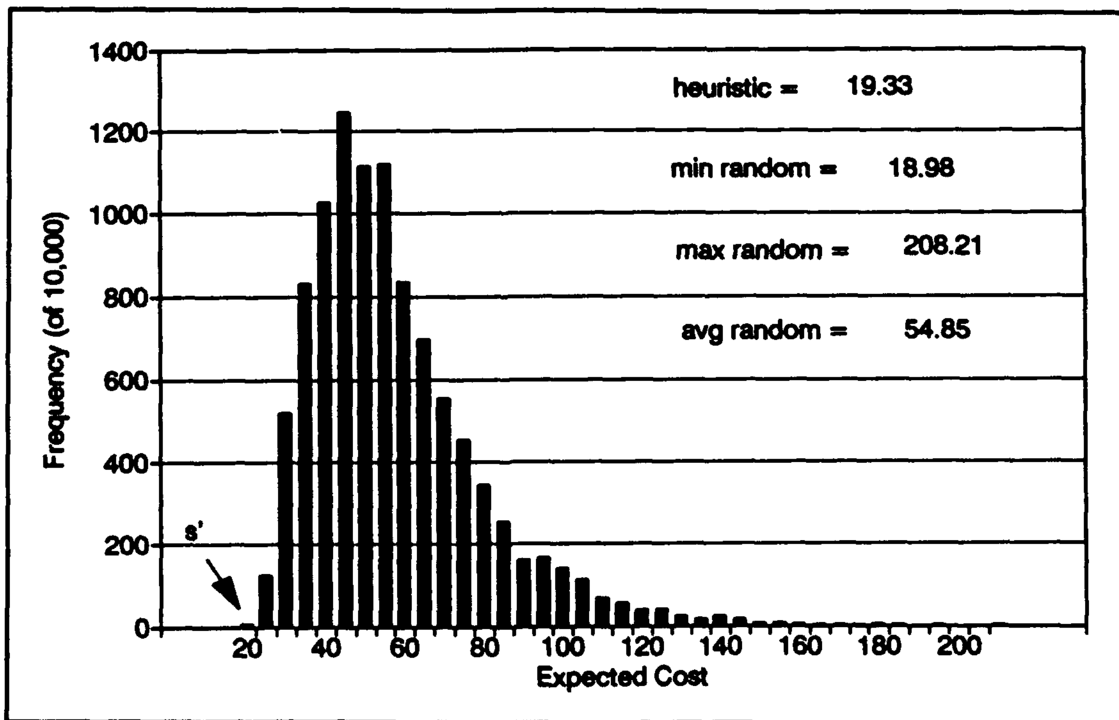


Figure 5.5. Frequency diagram for a problem with 5 cells, search costs 1-10, and switch costs 1-4.

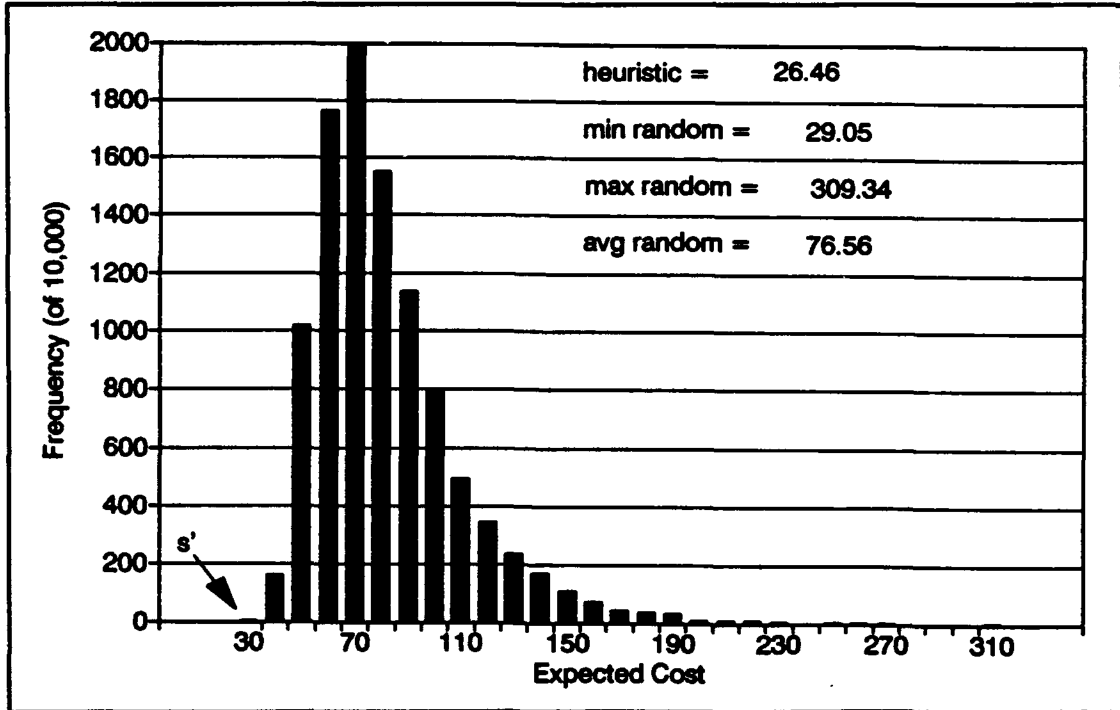


Figure 5.6. Frequency diagram for a problem with 5 cells, search costs 1-10, and switch costs 1-10.

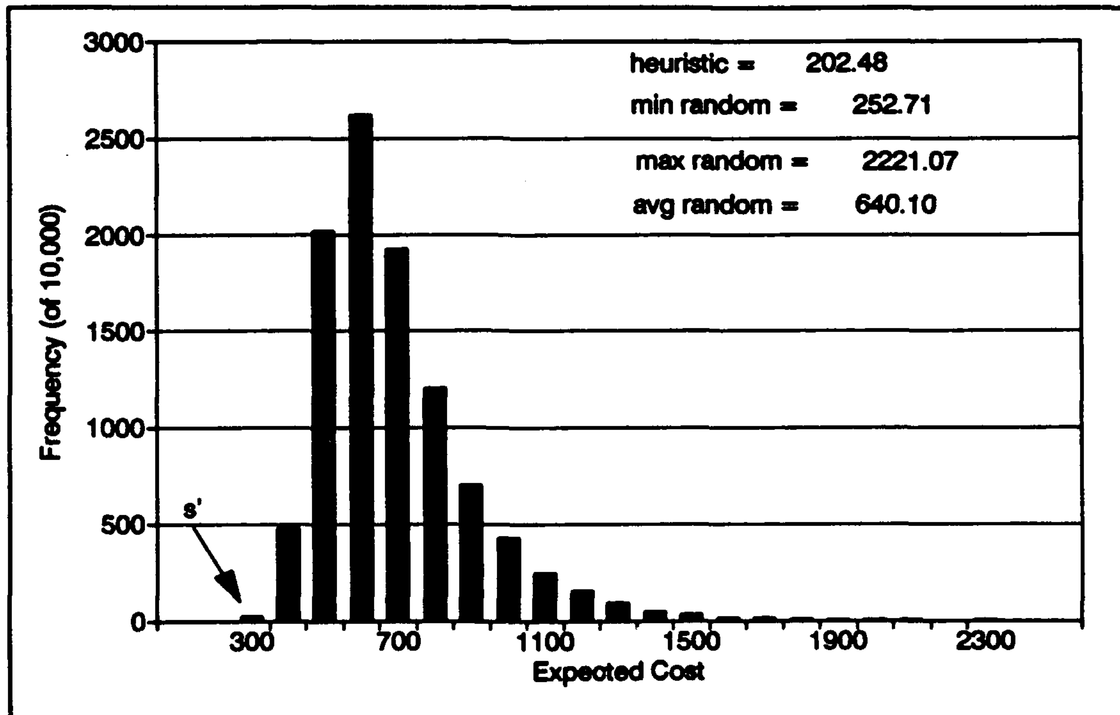


Figure 5.7. Frequency diagram for a problem with 5 cells, search costs 1-10, and switch costs 90-100.

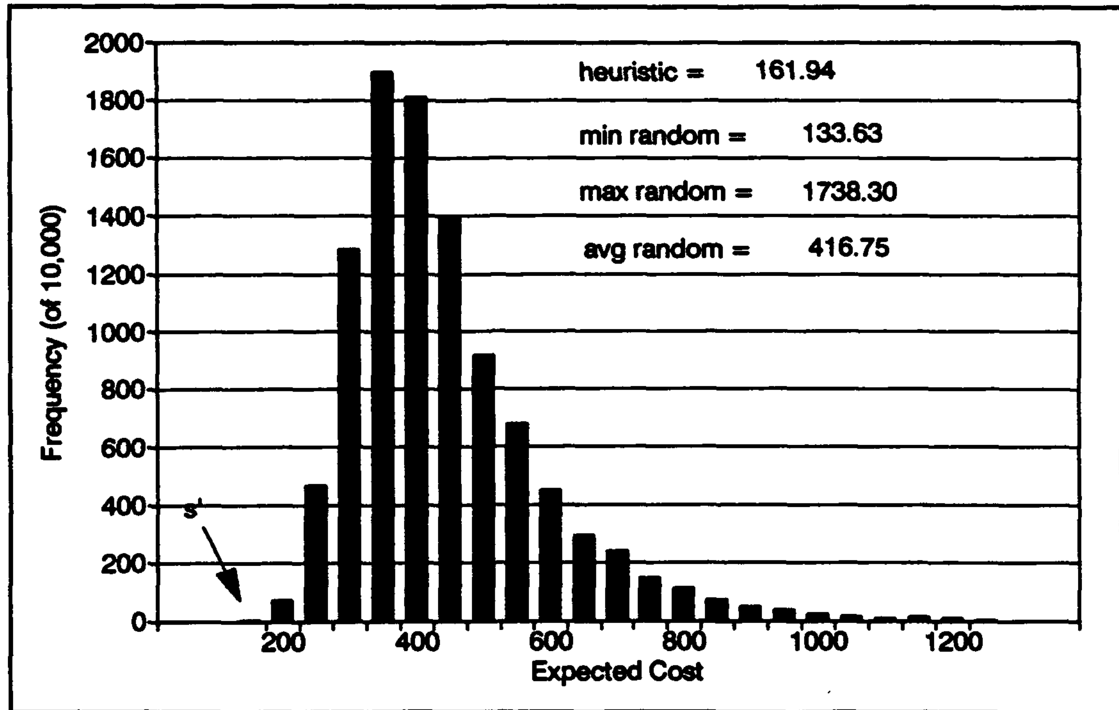


Figure 5.8. Frequency diagram for a problem with 5 cells, search costs 1-10, and switch costs 1-100.

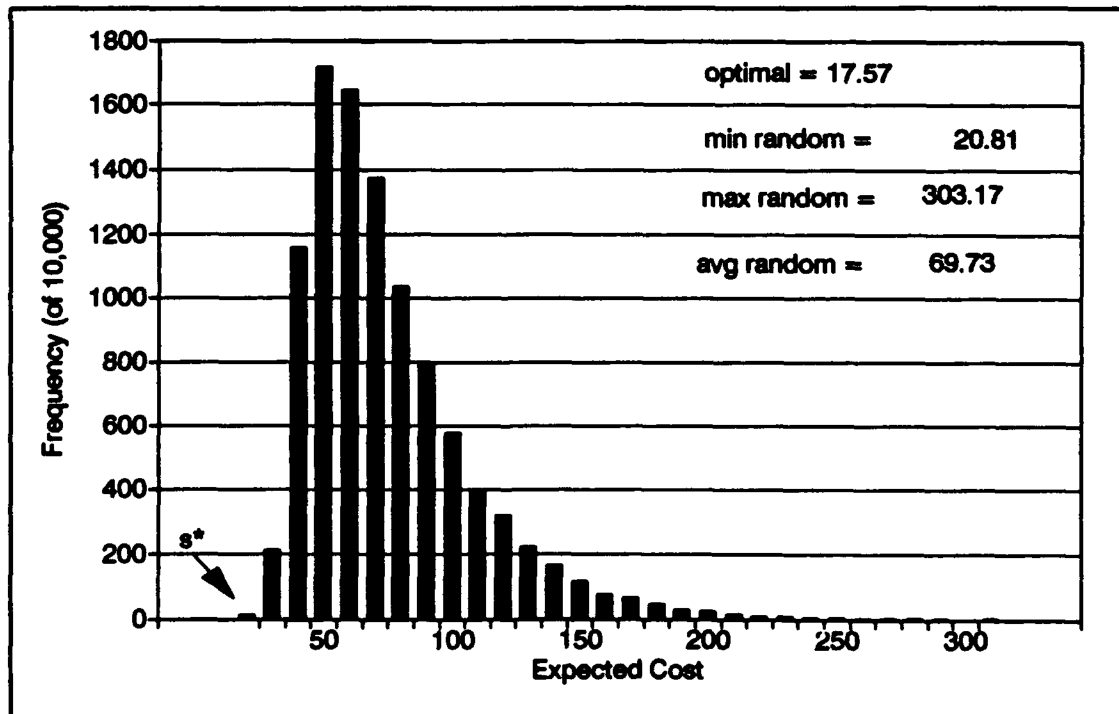


Figure 5.9. Frequency diagram for a problem with 10 cells, search costs 1-10, and no switch costs.

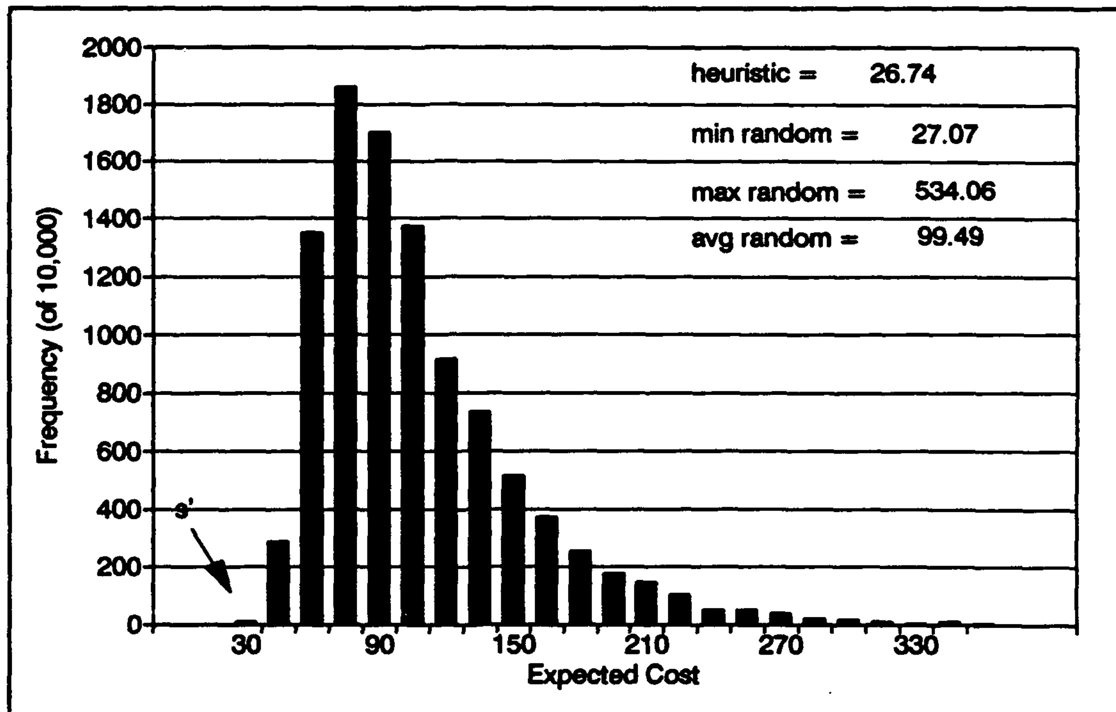


Figure 5.10. Frequency diagram for a problem with 10 cells, search costs 1-10, and switch costs 1-4.

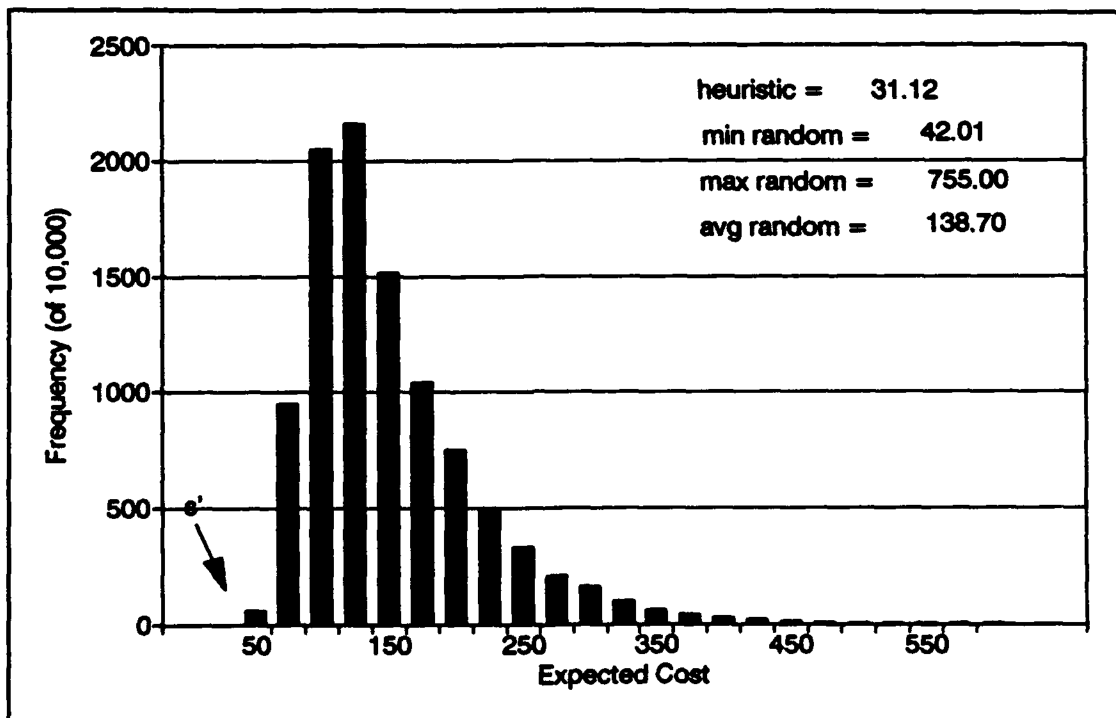


Figure 5.11. Frequency diagram for a problem with 10 cells, search costs 1-10, and switch costs 1-10.

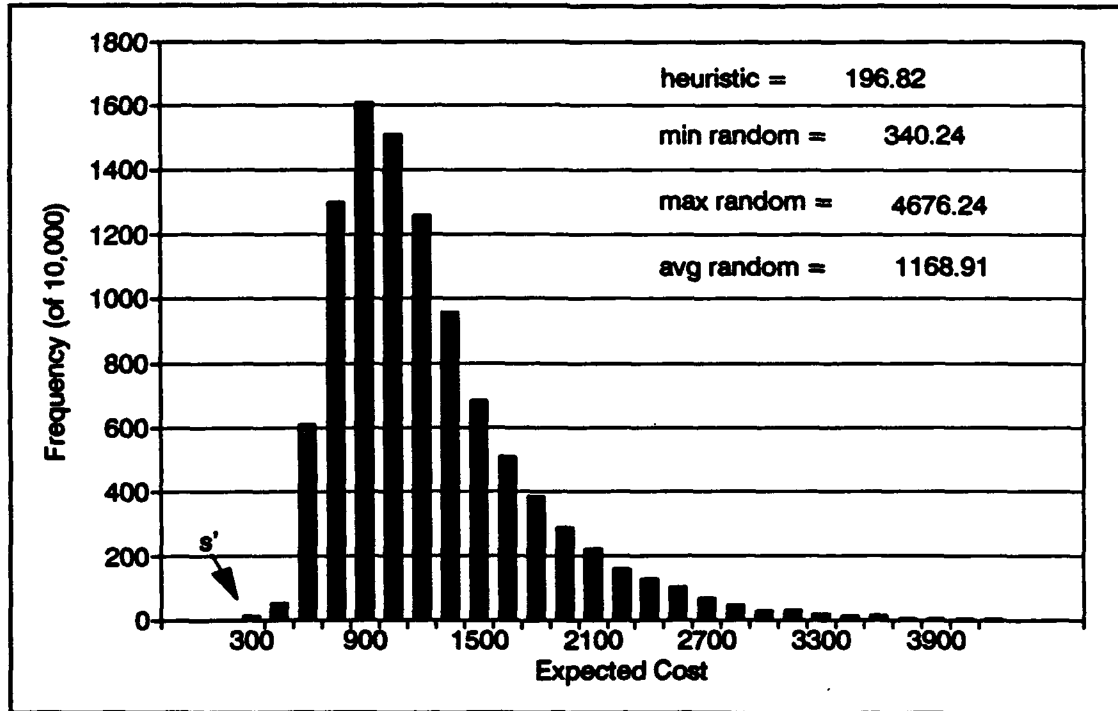


Figure 5.12. Frequency diagram for a problem with 10 cells, search costs 1-10, and switch costs 90-100.

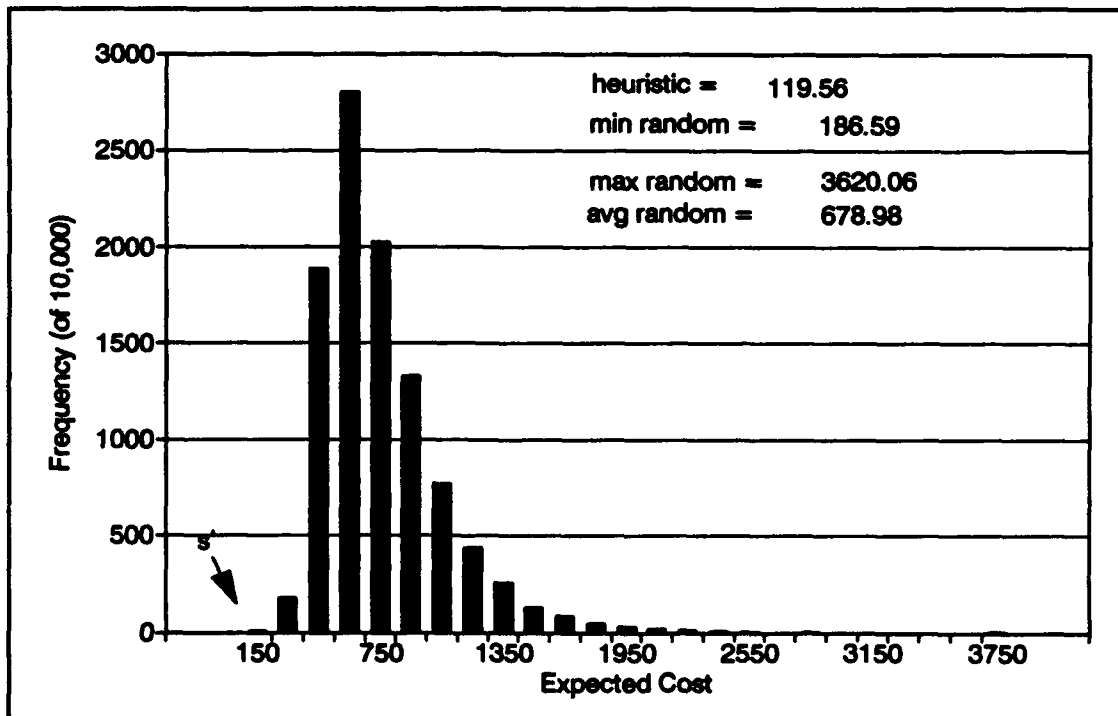


Figure 5.13. Frequency diagram for a problem with 10 cells, search costs 1-10, and switch costs 1-100.

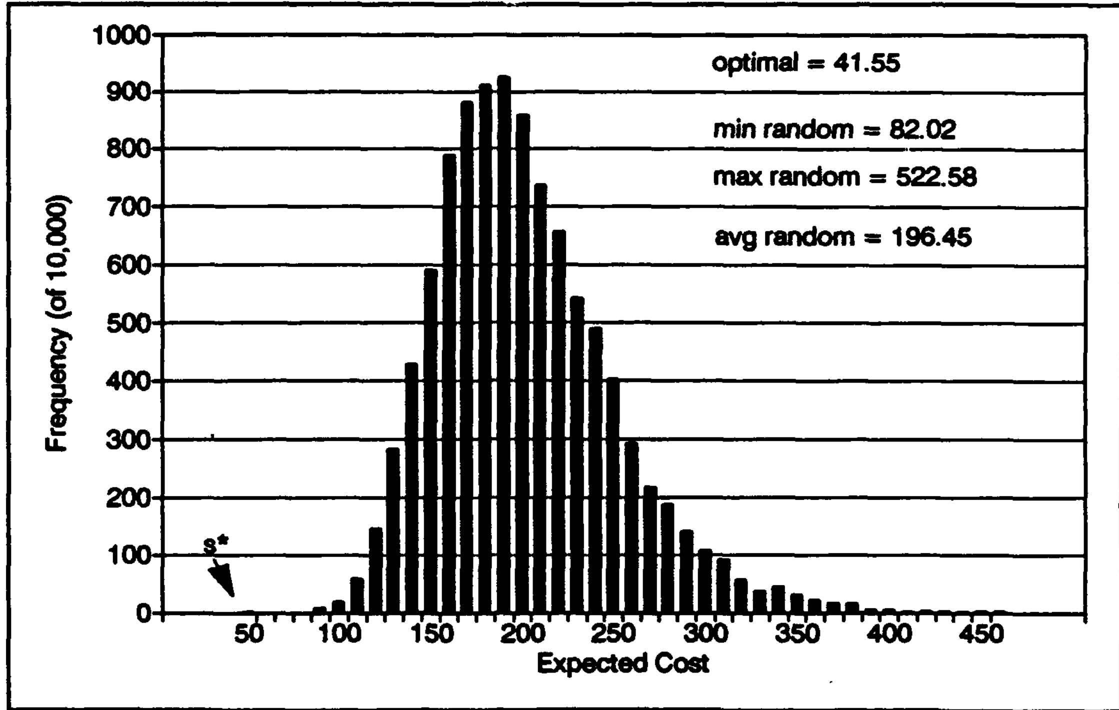


Figure 5.14. Frequency diagram for a problem with 25 cells, search costs 1-10, and no switch costs.

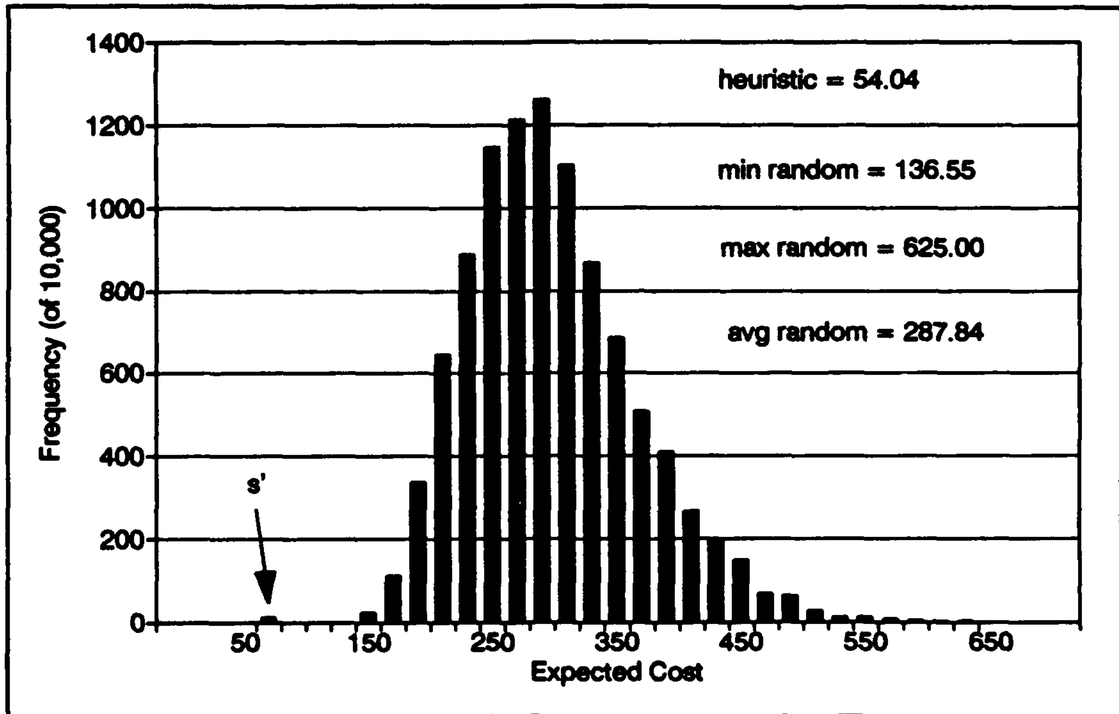


Figure 5.15. Frequency diagram for a problem with 25 cells, search costs 1-10, and switch costs 1-4.

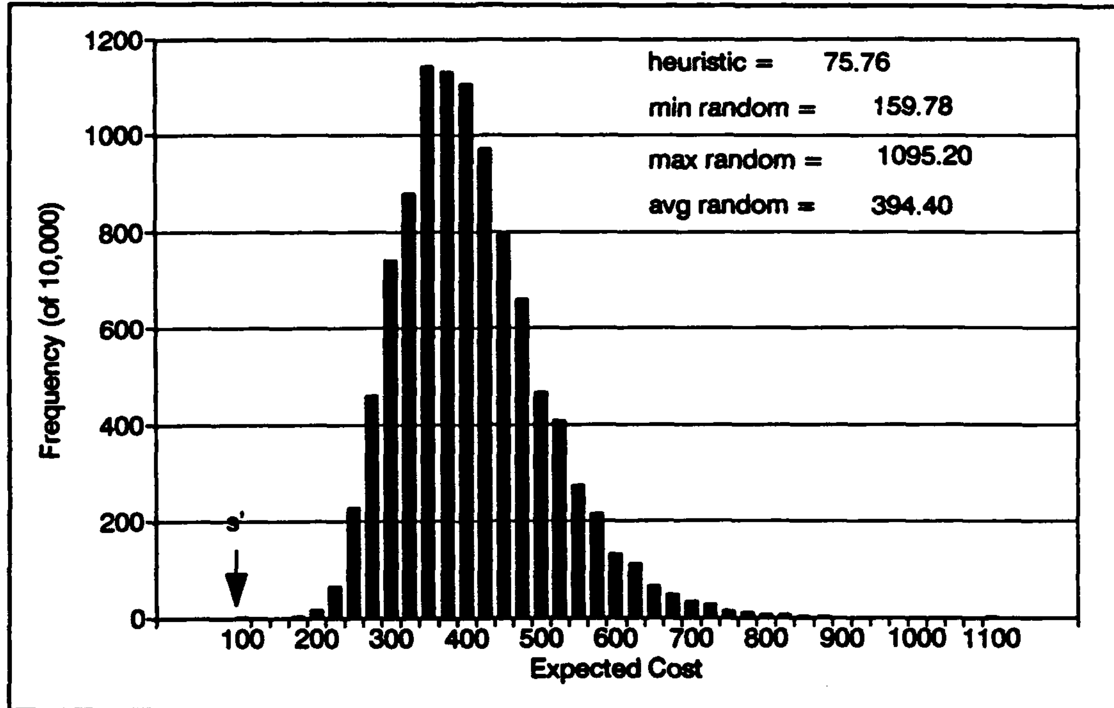


Figure 5.16. Frequency diagram for a problem with 25 cells, search costs 1-10, and switch costs 1-10.

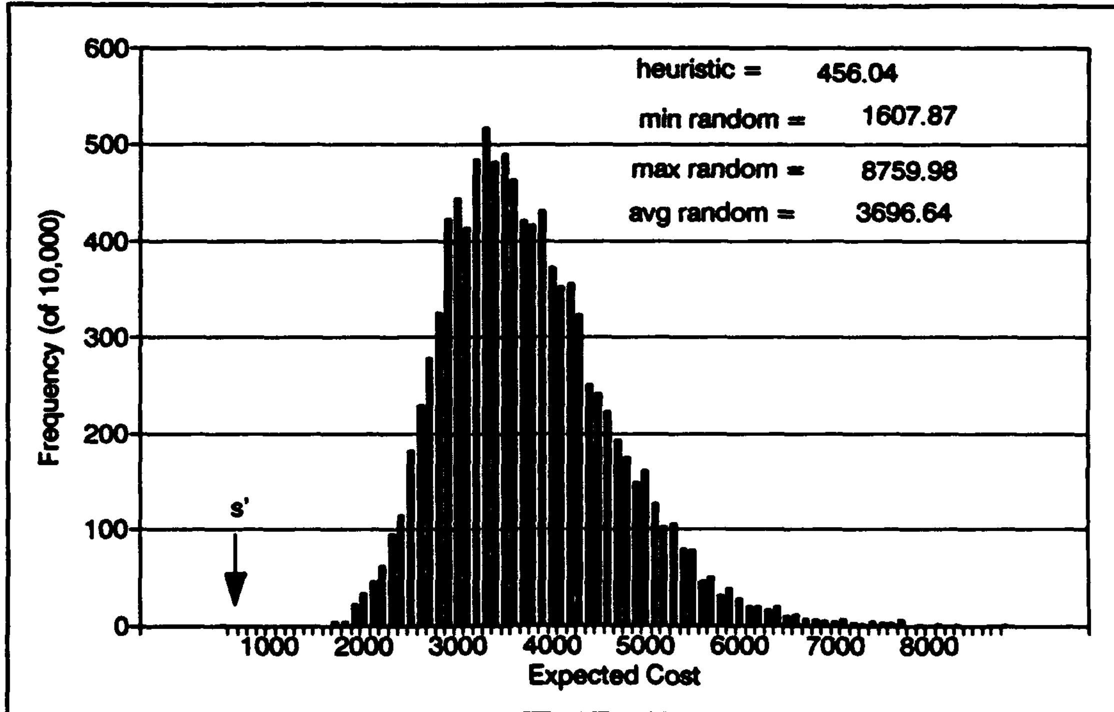


Figure 5.17. Frequency diagram for a problem with 25 cells, search costs 1-10, and switch costs 90-100.

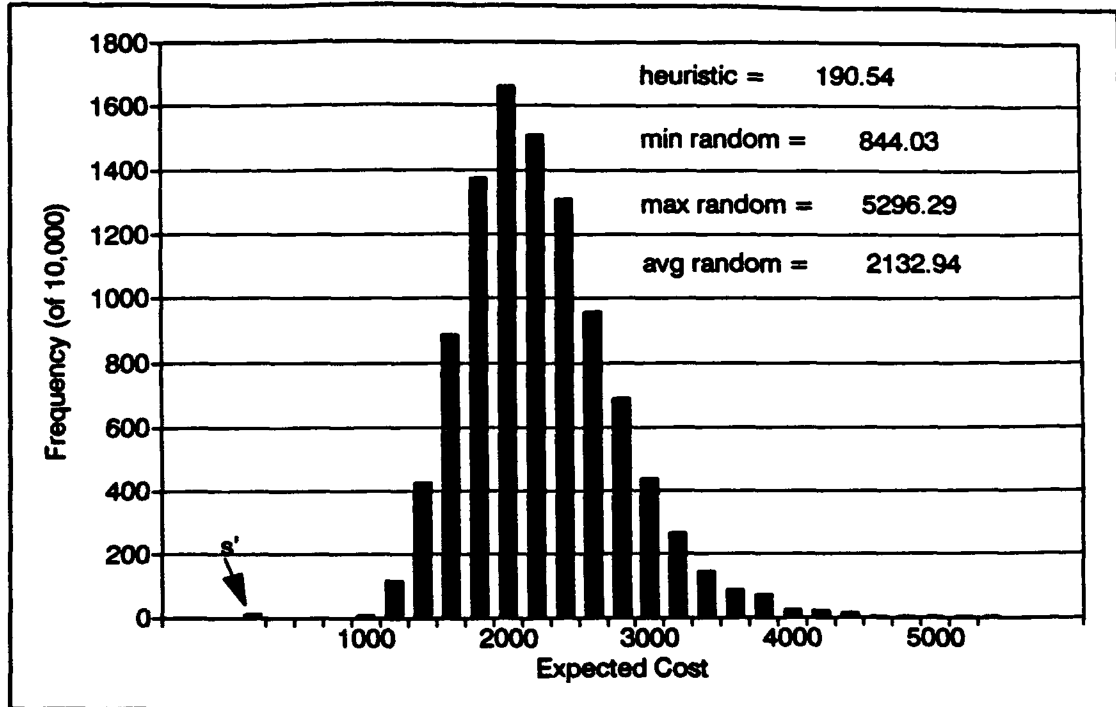


Figure 5.18. Frequency diagram for a problem with 25 cells, search costs 1-10, and switch costs 1-100.

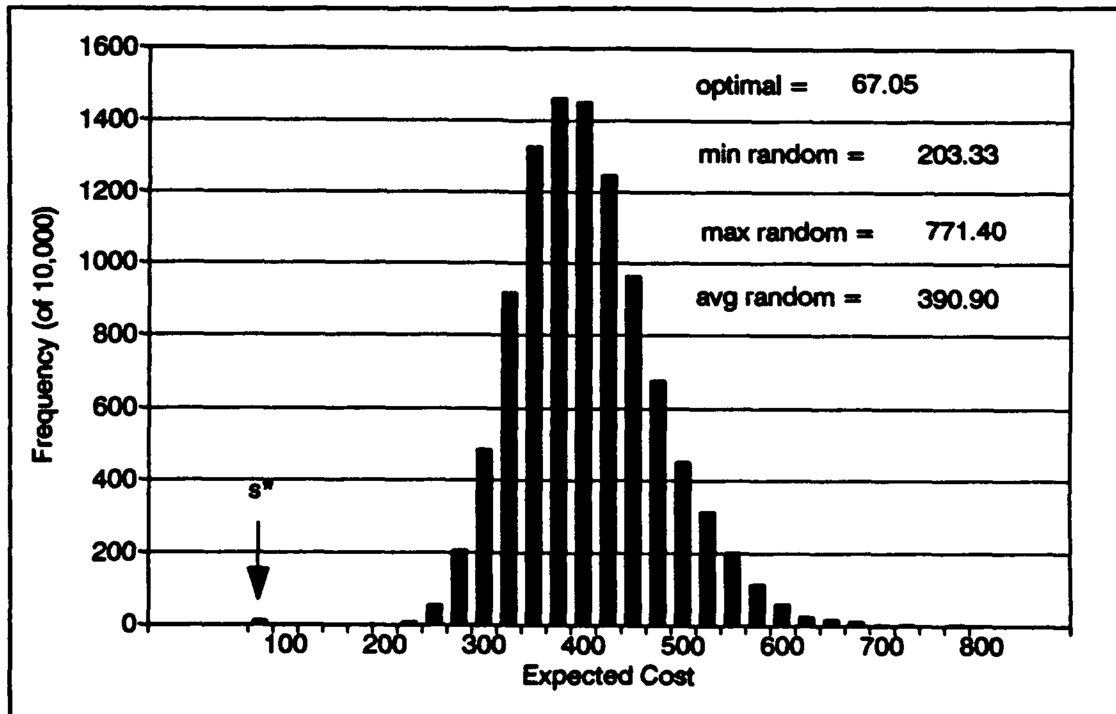


Figure 5.19. Frequency diagram for a problem with 50 cells, search costs 1-10, and no switch costs.

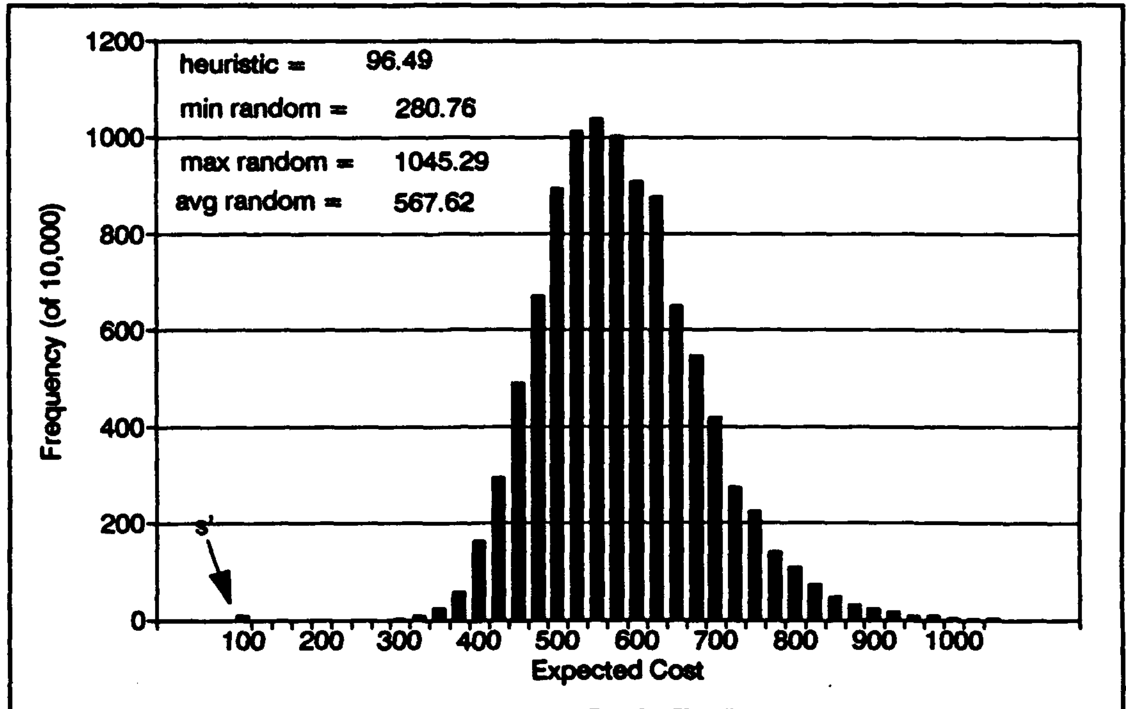


Figure 5.20. Frequency diagram for a problem with 50 cells, search costs 1-10, and switch costs 1-4.

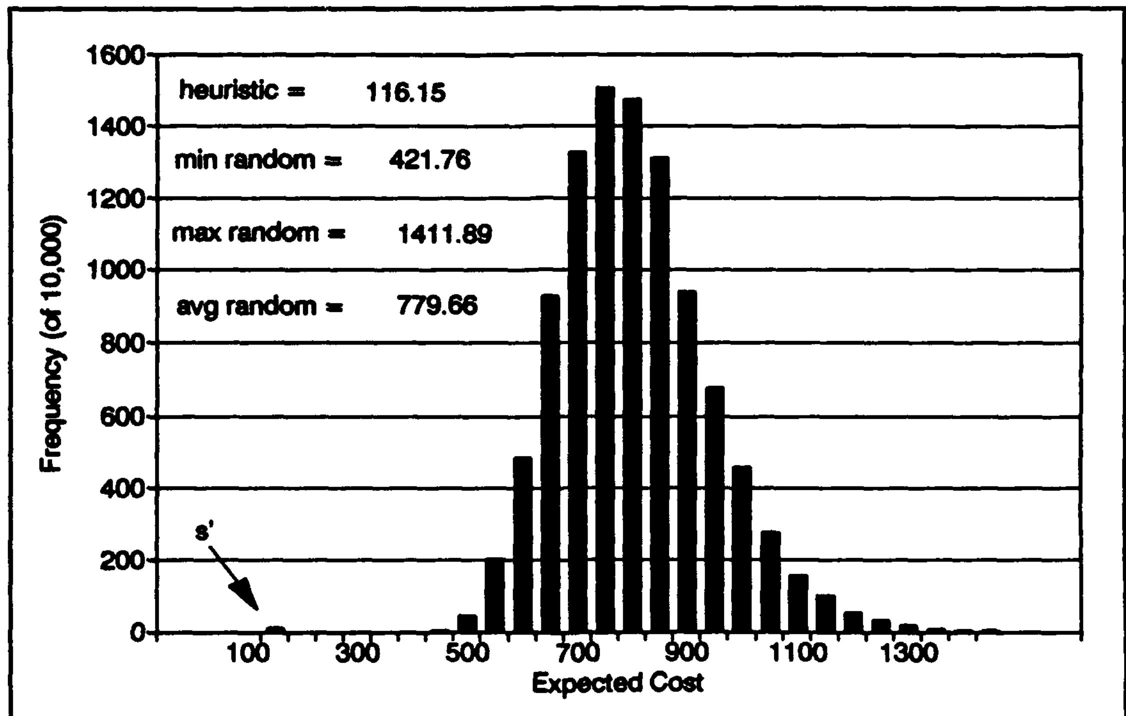


Figure 5.21. Frequency diagram for a problem with 50 cells, search costs 1-10, and switch costs 1-10.

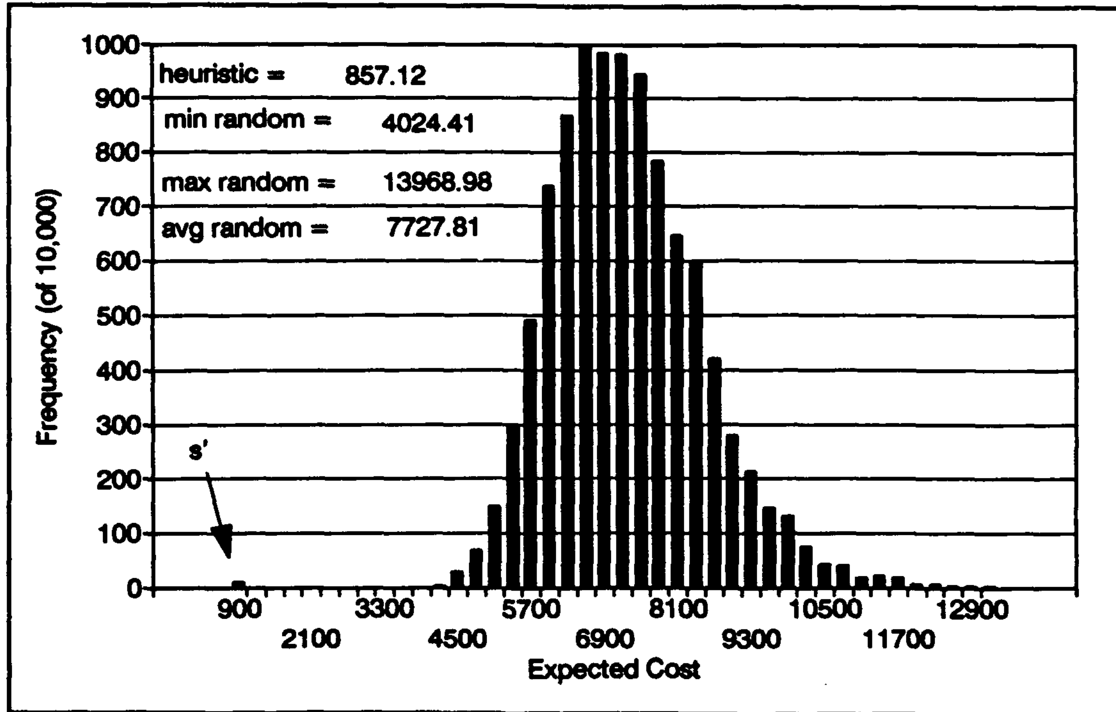


Figure 5.22. Frequency diagram for a problem with 50 cells, search costs 1-10, and switch costs 90-100.

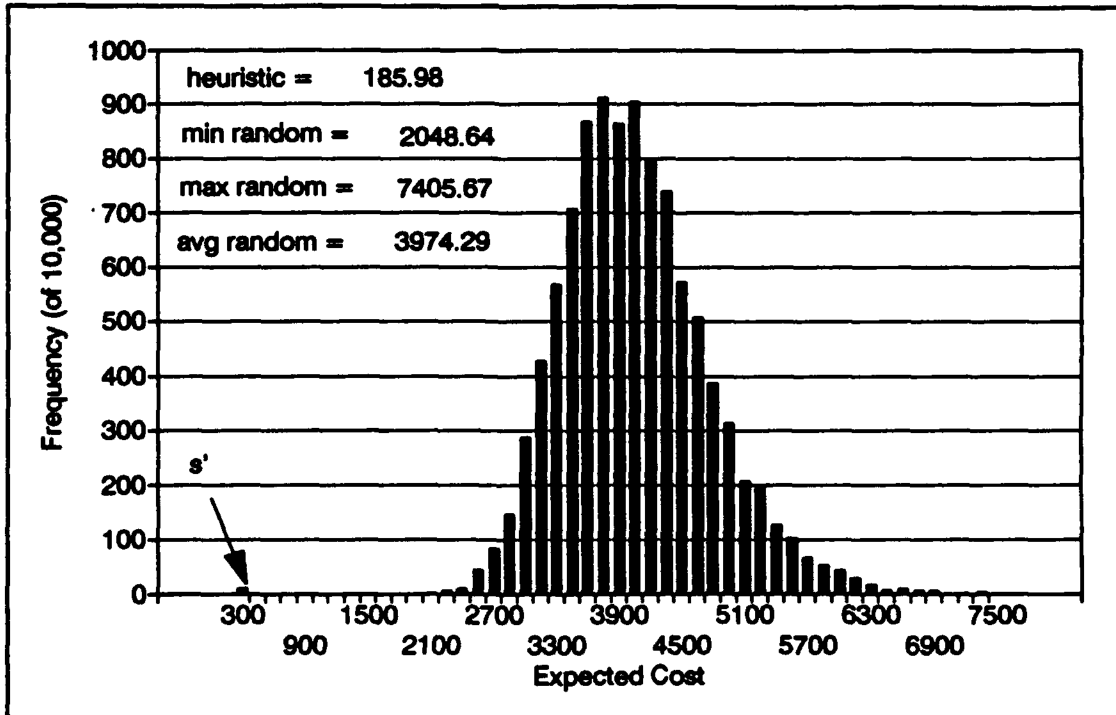


Figure 5.23. Frequency diagram for a problem with 50 cells, search costs 1-10, and switch costs 1-100.

5.3 Determining a Measure of the Performance of the Heuristic.

From the frequency diagrams we see that the heuristic is good. However, the complexity of the problem, as described earlier, cautions against assuming that we are very near the optimal. The natural question then is: "How close is the heuristic to the optimal?" Without knowing the actual optimal value, this question cannot be answered exactly. However, there is one case in which we know the optimal: the two cell problem in Lossner and Wegener [1982]. Although this is a small problem, it does give some insight into the performance question.

In Figure 5.24, s^*, s', and r' are shown together with 1000 random policies for the Lossner and Wegener (L/W) problem. All three values differ by no more than .0013. The value of r' is less than of s', a situation not uncommon for small problems. As discussed in the previous section, there are not as many possible combinations of policies for two cell problems.

Figure 5.24 suggests two methods of measuring the performance of the heuristic. The first is to use the ratio of the difference between the optimal and the heuristic over the optimal value: $\frac{s^* - s'}{s^*}$. For L/W, this value is -.0012669, or, the heuristic's value is about .13% higher than that of the optimal policy, s^*. For problems in which

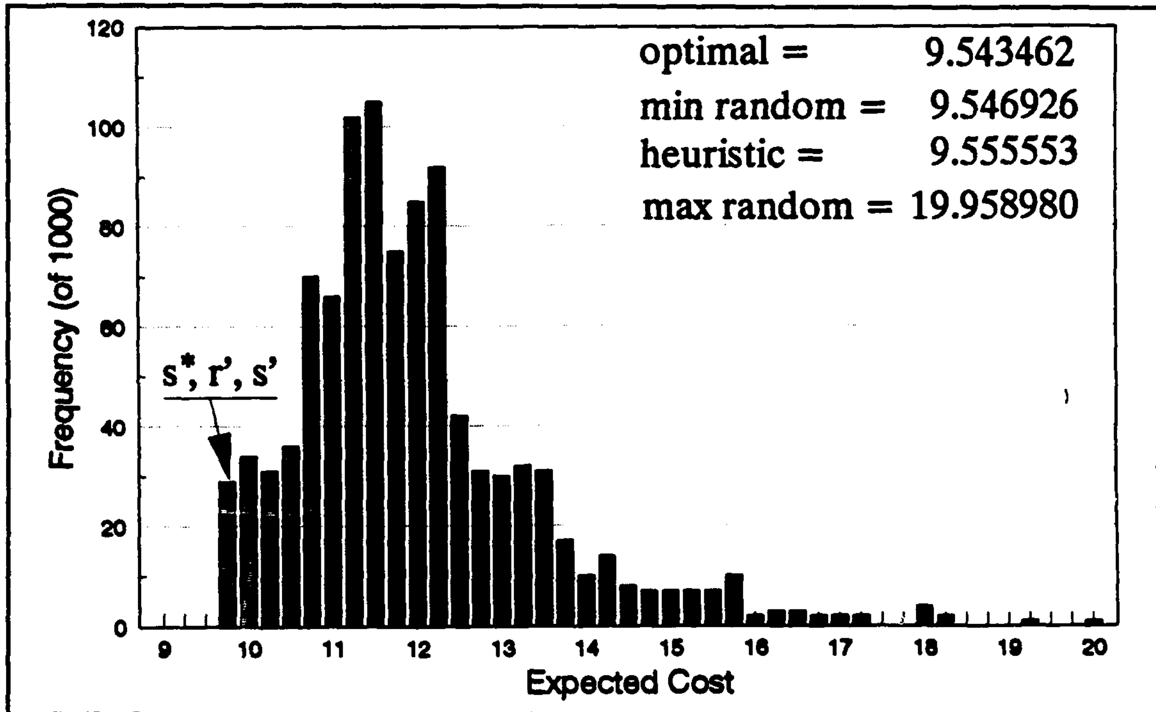


Figure 5.24. The example problem from Lossner and Wegener [1982].

the optimal is unknown, r' is substituted for s^* and the denominator is divided by min(r', s'): $\frac{r' - s'}{\min(r', s')}$. For the L/W problem, this would be -.000577, about .06% higher than s^*. A positive value with this method indicates that the heuristic is better than the best random policy and the value itself indicates by what percentage (of s'). A negative value indicates $r' < s'$ and the value itself indicates by what percentage (of r'). The advantage of this method is that it is simple. The disadvantage is that there is no estimate made of s^*.

For the second method, Law and Kelton [1991, p.401], present an estimate of the location parameter, γ, from a set

of data (see Figure 5.25).

The method of Law and Kelton was presented to make estimating parameters for shifted or truncated distributions easier. In most of the examples in the text, the location parameter is not the parameter of interest, but only a necessary computation. In our case, however, the true location parameter, γ, represents s^*, and is of prime concern. An unfortunate aspect of this formula is that it sometimes gives negative values, an impossibility in our case.

After estimating s^* (with $\hat{\gamma}$), the performance of the heuristic can be expressed as: $(\hat{\gamma}-s')/\hat{\gamma}$. In this case the heuristic value will always be larger than the estimate of s^*, and the resulting measure of performance will always be

$$\hat{\gamma} = \frac{X_{(1)}X_{(n)} - X_{(k)}^2}{X_{(1)} + X_{(n)} - 2X_{(k)}}$$

where: $x_{(1)}, x_{(2)}, x_{(3)}, \ldots$ are an ordered sample
$x_{(1)}$ = smallest value
$x_{(k)}$ = second smallest value
$x_{(n)}$ = largest value

Figure 5.25. A method of estimating the location parameter, gamma. From Law and Kelton [1991].

negative. This is reasonable because, by observation, $E[s^*] < E[s']$. (The observation is that for small n, $r' < s'$ is not uncommon; there is no reason to expect the heuristic to behave better for larger n; and, in all cases, $s^* \leq r'$). For the L/W problem, the estimate of s^* is 9.5500393 and the performance of the heuristic would be reported as -.0005773, again about .06% higher than optimal. As did the earlier estimate, this reports the heuristic's performance about twice as good (.06% vs .13%) as it actually is.

The advantages with the method of Law and Kelton are that there is an estimate of s^*, and that the range of the distribution is considered in that estimation. The disadvantage is that there is no measure of the accuracy of the estimate of s^*, and that in some cases the formula gives impossible (negative) values, and therefore, cannot be used. This happens when $r' - s'$ is large.

Considering the advantages and disadvantages of the two methods, the first is better and will be used here. The main reason is that in estimating s^* with Law and Kelton's formula, more chance for error is introduced into the problem. That the formula's results are unusable in some cases also limits its use as well as calls into question those results that are usable. Thus, $\frac{r' - s'}{\min(r', s')}$ will be used as the measure of performance.

5.4 Behavior of the Problem as a function of the number of cells.

Two values of interest are dependent on n: the expected cost of search policies, and the length of search policies. Both are positively correlated with n, and the reasons they are such are similar. Graphs of these relationships are in Figures 5.26 and 5.27. One way to look at these graphs is that Figure 5.26 is the result of Figure 5.27 and that Figure 5.27 is the result of the average p_i. When n is large, more searches are required to get the probability of not finding the target below tolerance. Because more looks are required to get below tolerance, the expected cost increases. Each look in a large n problem removes less

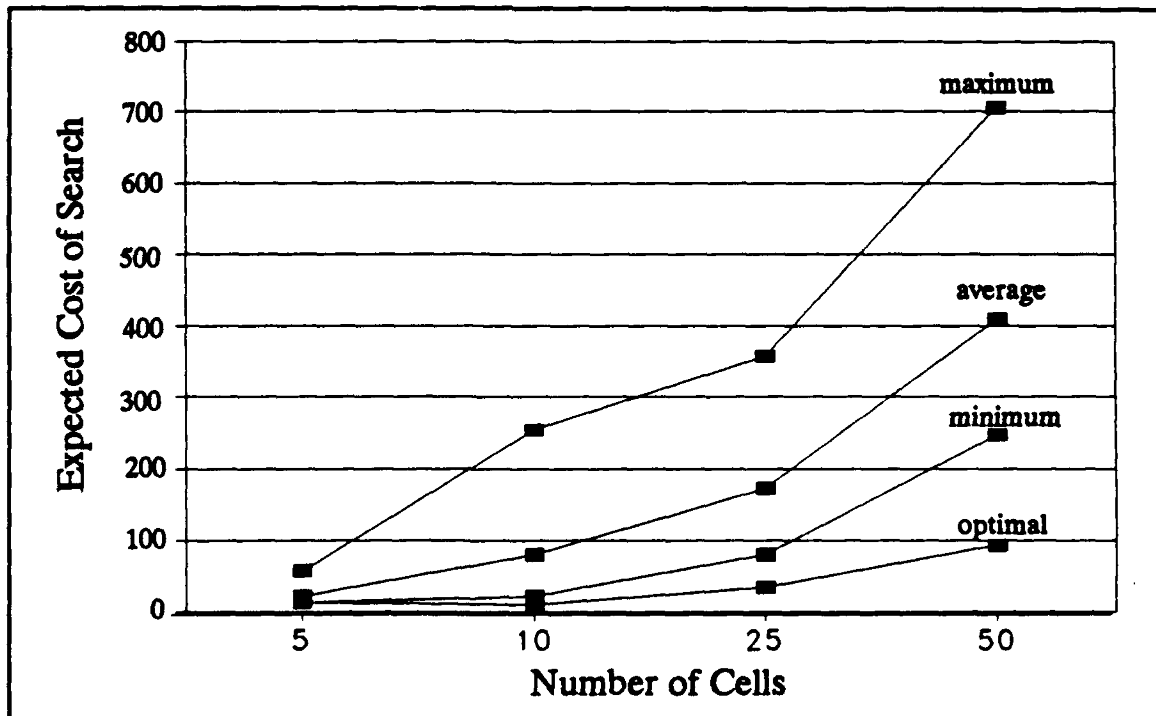


Figure 5.26. Expected Cost as a function of n. Results are for 1000 random policies for a problem with no switch costs.

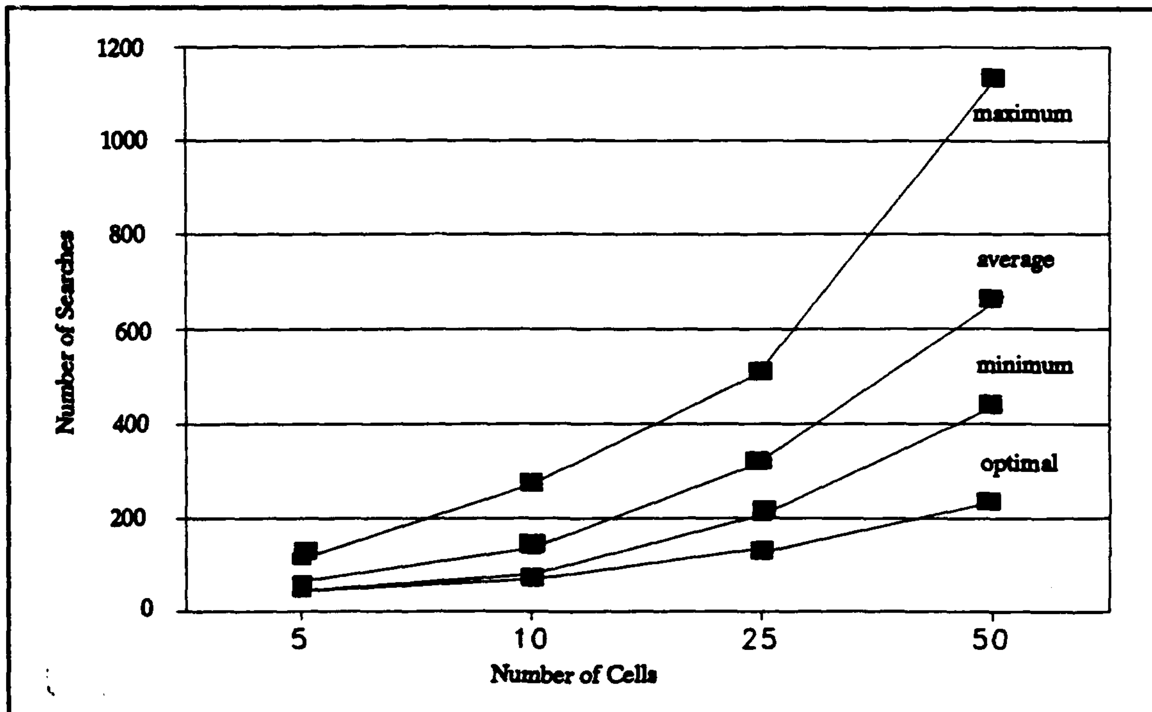


Figure 5.27. Number of Searches as a function of *n*. Results are from 1000 random policies for a problem with no switch costs.

probability of not finding the target than in a small *n* problem.

Figure 5.27 is important in appreciating the complexity of the search problem. For example, consider the 50 cell problem. The optimal policy took 225 searches to get below tolerance. If this number is used as an estimate for the number of searches required for all 50 cell problems (including those with switch costs), then an estimate of the number of policies to evaluate by exhaustive enumeration can be computed. The number of possible policies to consider is 50^{225} (more than $1\text{x}10^{375}$), clearly much too numerous to consider using exhaustive enumeration to find the optimal.

The number of searches required and the large number of possible policies of that length are important because we can see that the 10,000 or 1,000,000 random policies that we are using in our experiments is relatively small. It is virtually impossible to evaluate a significant portion of the population. (As a note of interest, a test of 1,000,000 random 50 cell policies on an IBM compatible 486 33mz takes about ten hours to run.)

For very small *n*, the opposite effect is true. The optimal policy took 43 searches to get below tolerance for the 5 cell problem. In section 5.6 we examine 2 cell problems which required only 8-12 searches. In a small problem each look removes a larger amount of probability that the target remains undetected than in a large problem.

The purpose of examining the complexity of the problem and trying to estimate the size of the population of possible policies is to caution against thinking that the best of 1,000,000 random policies is "very" close to the optimal. The gap between r' and s' seen in Figure 5.3 shows this in practice.

5.5 Behavior of the problem as a Function of the Switch Costs.

Our prediction was that the heuristic would perform better for small switch costs than for large switch costs. The reason for this prediction is that these problems are

'closer' to problems without switch costs, for which the heuristic is optimal. The results of the experiment do not support this conjecture.

Figure 5.30 shows that performance seems more dependent on the number of cells than on the range of switch costs. Figure 5.32 shows a close-up of the performance for eight or less cells. The problems with smaller m_{ji}, 1-4 and 1-10, have the lowest performances for 2, 3, 4 and 8 cells. If our original conjecture were true then problems with lower switch costs would generally have higher performance because these problems are 'closer' to problems for which the heuristic is optimal.

For large n the heuristic performed best when $m_{ji} \in (1,100)$. For these problems, r' is more than six times s' (50 cells, Figure 5.23). The reason for this is the increased variation in the switch costs. When the range of m_{ji} is small (1-4, 1-10, or 90-100), the variation among switch costs contributes less to the total variance of the random policies. When the range of m_{ji} is large (1-100), the variance in the random policies will include the increased variance of the switch costs.

For all problems the heuristic tends to choose the lower values of m_{ji}. The heuristic always chooses, as the next cell to visit, the cell for which the value $\frac{p_i b_i a_i^{N_i}}{c_i + m_{ji}}$ is a maximum. When $m_{ji} \in (1, 100)$, and cells are chosen at

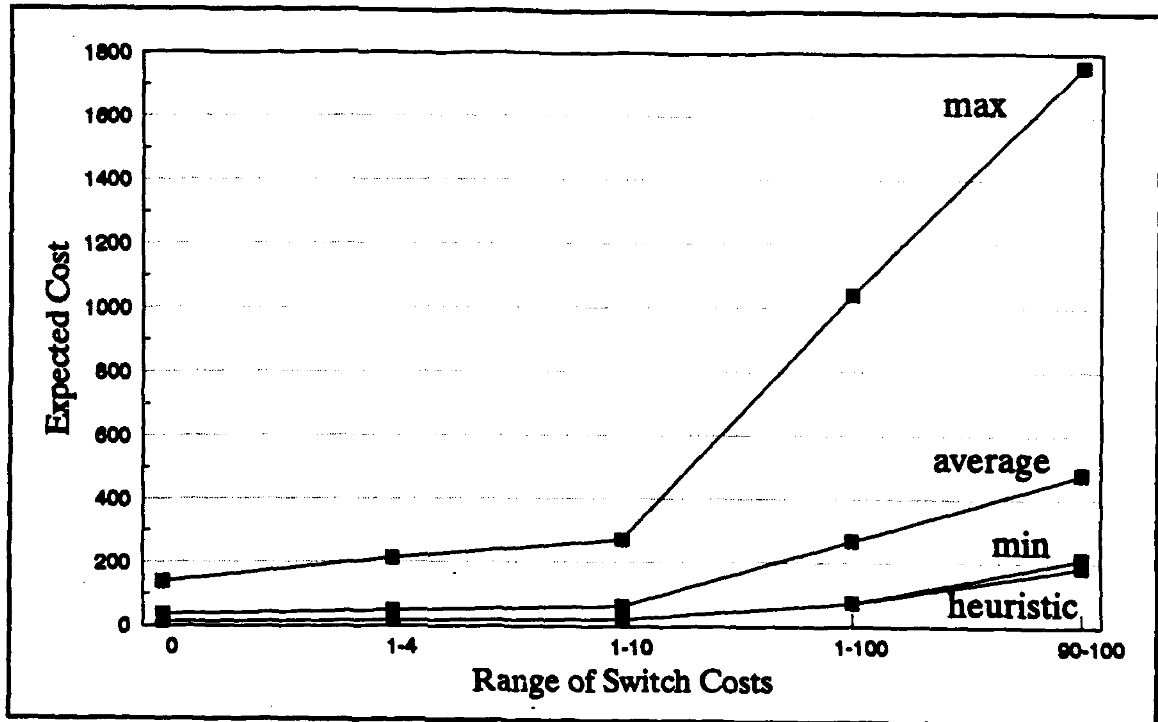


Figure 5.28. Expected cost as a function of the switch cost range.

random, then $E[m_{ji}] = 50.5$. When cells are chosen using the heuristic, we expect $E[m_{ji}] < 50.5$. Similarly, when $m_{ji} \epsilon$ (1,4) and cells are chosen at random, $E[m_{ji}] = 2.5$. When cells are chosen using the heuristic, we expect $E[m_{ji}] < 2.5$. In both cases the heuristic is expected to, on average, choose lower values of m_{ji}.

Since our measure of performance, $\frac{r' - s'}{\min(r', s')}$, is relative, we might expect the performance for the different switch cost ranges to be about the same. However, $E[m_{ji}$ (random)$] - E[m_{ji}$ (heuristic)$]$ is greater when $m_{ji} \epsilon$ (1, 100) than when $m_{ji} \epsilon$ (1,4). Because of this greater difference, and the greater variance in a larger range, the

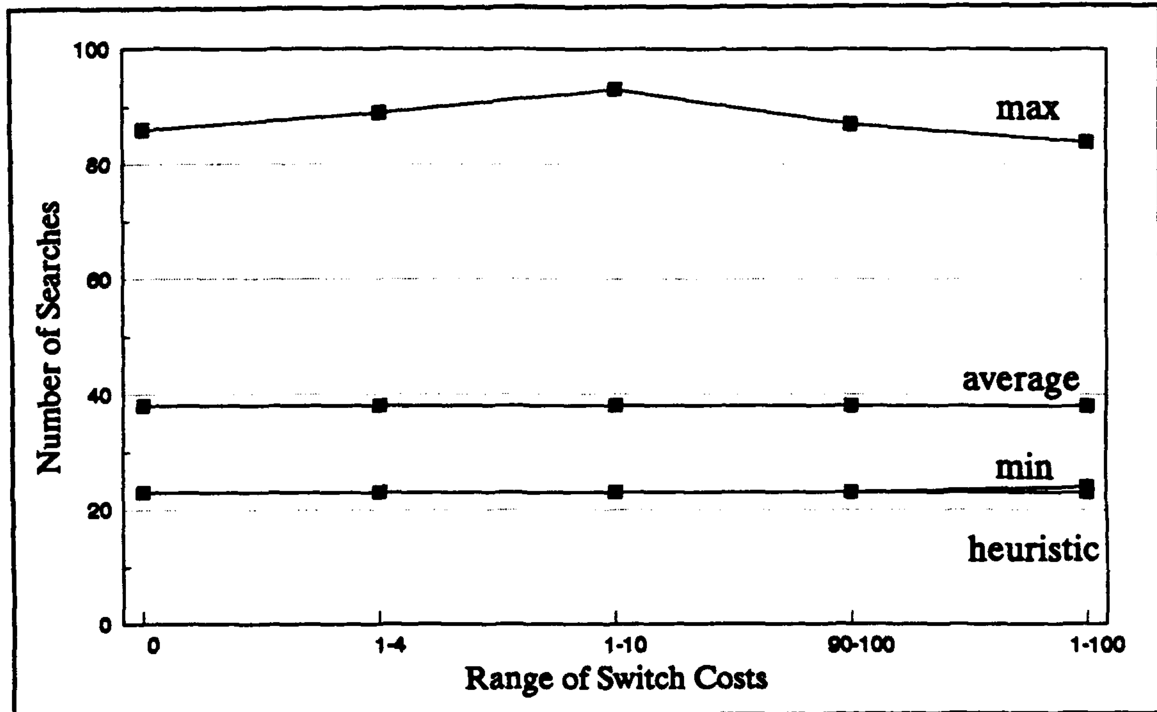


Figure 5.29. Number of searches as a function of switch cost ranges.

heuristic performs better against problems with greater ranges of switch costs.

As a note for completeness, we do not examine further the number of required searches as a function of the range of switch costs (Figure 5.29). This is because the switch costs, as well as the search costs, do not influence the number of searches. At each stage of the search, some amount of positive probability that the target remains undetected is removed. This amount is $p_i b_i a_i^{M_i}$, and does not involve costs. When the sum of these amounts is greater than 1 - tolerance, the evaluation of that policy is finished.

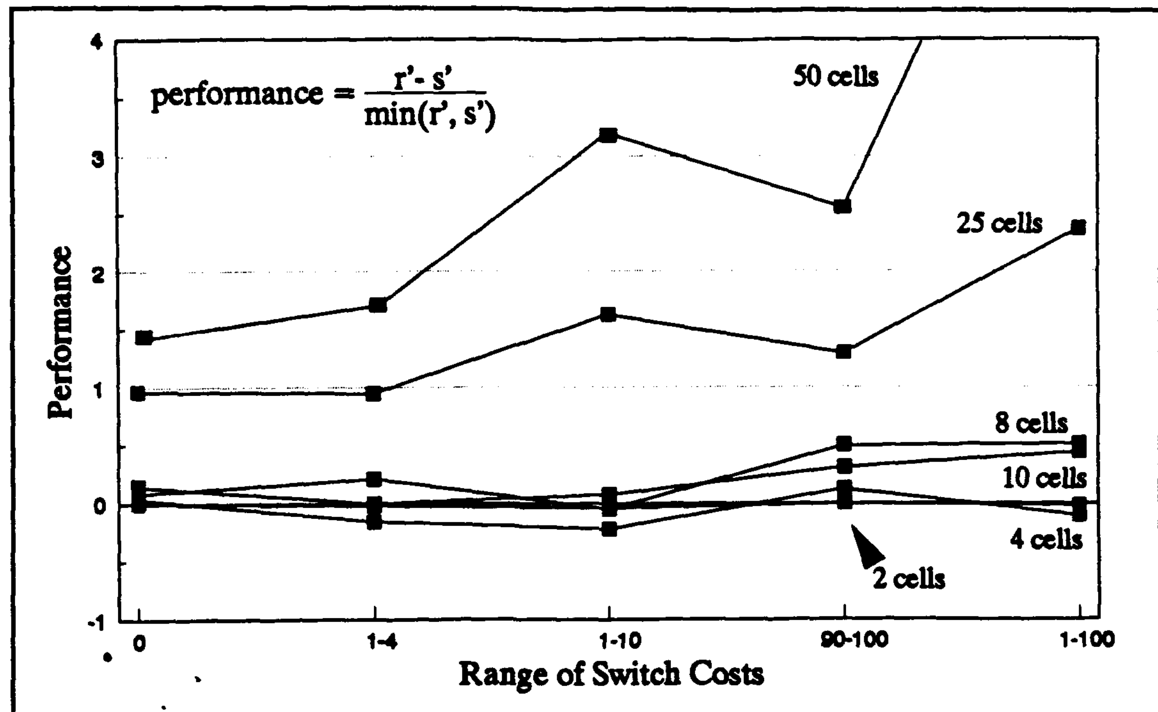


Figure 5.30. Performance of the heuristic as a function of switch cost range.

5.6 The Performance of the Heuristic.

Figures 5.31 and 5.32 show the heuristic's performance across the various factors we have been examining. As with the other measures, there is a difference in the measure of performance between the different levels of n. As expected, and seen earlier, $r' < s'$ for most of the small n problems, while, for the large n problems, $r' > s'$.

The point at which $s' = r'$ (ie, performance = 0), is about 5 cells. Interestingly, the performance of 2 cell problems is generally higher than that of 3 and 4 cell problems. The average performance of the 2-5 cell problems is -.09, that is, $s' > r'$ by about 9%.

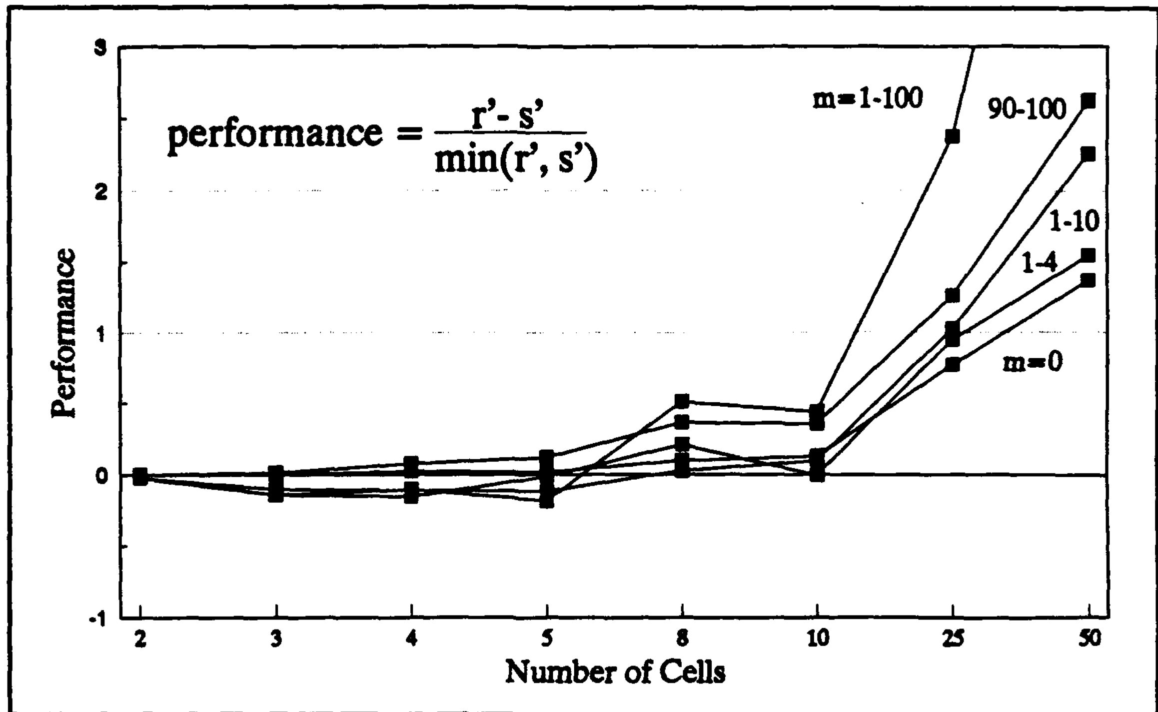


Figure 5.31. Performance of the Heuristic as a function of *n*.

For larger problems the performance grows rapidly (Figure 5.31). As discussed earlier, the complexity of the problem has a lot to do with this. The control, $m_{ji} = 0$, gives us an idea of how we should expect the performance to behave. The control compares the optimal value to the best random value. We see that as *n* gets large, the best random value gets farther away from the optimal value.

For small problems, we observe that the heuristic is generally between 10% under and 20% over the expected cost of the best random policy. We cannot extrapolate this above 5 cells, however, there is no reason to expect the effectiveness of the heuristic to either improve or deteriorate as *n* increases.

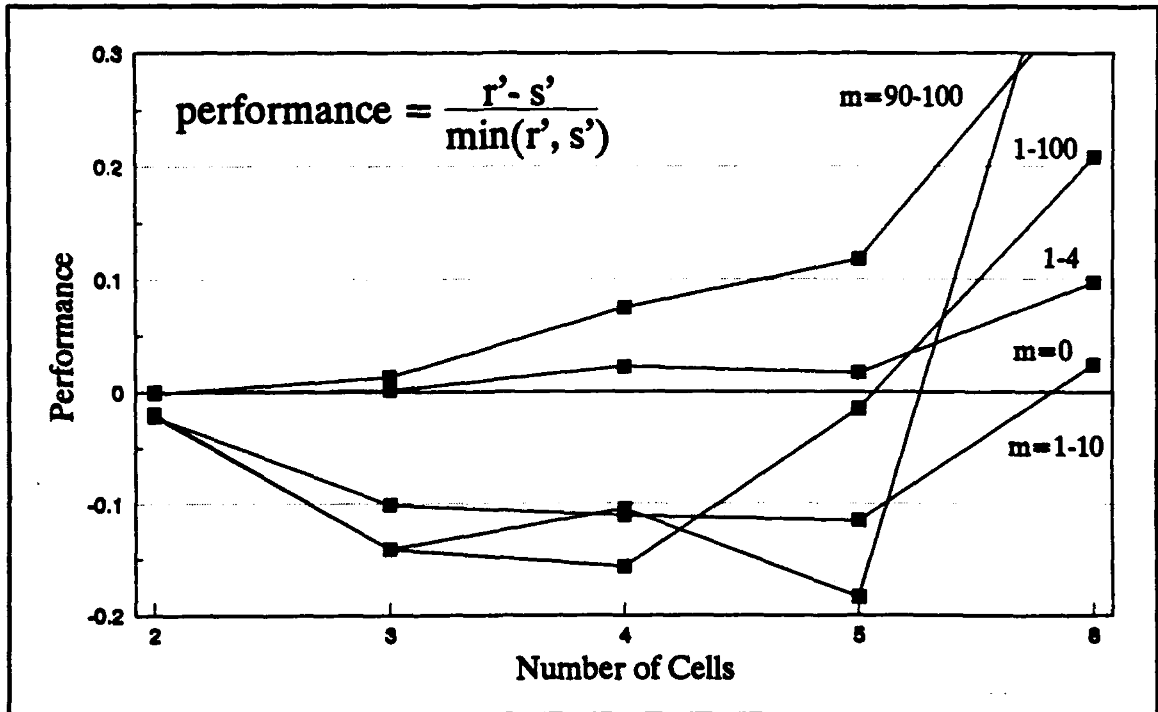


Figure 5.32. Performance of the heuristic for 2 through 8 cells.

From the experiments, we conclude that the heuristic is good, but we cannot say exactly how good. The heuristic is also fast: the computations required are a function of n^2.

VI

CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions.

The main conclusions of this study are:

1. The distribution of the expected costs of search policies for a given problem is mound shaped, truncated and skewed to the right. When n is small, the approach of the left tail to the x-axis is more abrupt than for large n. In all cases the right tail appears asymptotical.

2. For small n, the population of possible policies, given a tolerance, is small enough that a sample of, say, 1,000,000 random policies returns a good approximation of s^*. (In many of the 2 and 3 cell problems the best random policy was identical to the optimal policy.) For large n, the population is immense, and r' rarely comes within 200-300% of the heuristic.

3. The heuristic performs well as compared with r'. An exact measure of performance is not available. For small n, $s' < 1.2\,r'$, generally. Beyond 5 cells, we cannot generalize.

6.2 Recommendations for Future Research.

There are three main directions to take to further this study: find a better estimate of s^*; mathematically establish a lower bound for s^*; and improve on the performance of the heuristic.

In estimating s^*, there are, again, two approaches: use statistical methods to estimate gamma (the location parameter or point of truncation of the distribution of expected costs); and improve random policy generation with the goal of getting closer to s^*.

When looking at the search problem from a statistical viewpoint, it is tempting to try to fit the data to a distribution. If this were possible, then we could evaluate the heuristic using its p-value for that distribution. However, goodness of fit tests almost always reject an estimated distribution when the sample size is very large [Law and Kelton, 1991]. There is also a dilemma concerning the point of truncation. Since $P(X < \gamma) = 0$, using either s' or r' to estimate γ causes problems. Some variation of the formula presented by Law and Kelton, [1991, p. 401], may yield better results.

There are some promising possibilities for attempting to generate a random policy closer in value to s^*. Since the heuristic is easy to construct, use it as a starting point. Then, randomly generate changes to this policy in the attempt to get one with a lower value. We observed that very often the difference between s' and r' (or between s^* and r') is one or two changes (for small n).

A characteristic of problems without switching costs is that, for a given number of searches, the order in which the cells are searched does not impact the expected cost. This makes for a more efficient enumeration of possible combinations. (Because the policies (1, 2) and (2, 1) are equivalent if there are no switching costs and one is only concerned with the ultimate result and not the incremental results.) If this could be extended in some manner to the problem with switching costs, then a "smart" enumeration algorithm might be designed.

Mathematically establishing a lower bound for s^* would be helpful in estimating s^*. It seems reasonable that a good starting point is to remove the movement costs. That is, the expected cost of the same problem without movement costs should be less than with movement costs.

In improving on the heuristic, one can use the existing heuristic as a standard to improve upon, and not be as concerned with estimating s^*. Since switch costs are what

make the problem difficult, they are where the effort should go when seeking improvement. A look ahead feature can be provided at each stage of the policy generation process. For example, if m'_i were the average movement cost away from a cell, then a policy creation rule which includes m'_i in the denominator should tend to choose cells with lower movement costs. Similarly, one could compute m''_i, m'''_i, etc., for more than one look ahead. Each look ahead increases the complexity of the policy construction rule by a factor of n.

Another possibility for improving the heuristic is to look for or try to introduce periodicity into the policy. As seen earlier [Matula, 1964 and Lossner and Wegener, 1982] when the overlook probabilities, a_i, meet certain restrictions, optimal policies are ultimately periodic. If we restrict the a_i's then we can look for or introduce periodicity in the resulting heuristic policy in the hopes of lowering the expected cost.

The restriction on a_i to give periodic results is that $\frac{\log a_i}{\log a_j}$ be rational numbers for all $i,j \in I$. This should not cause problems, because we can find a_i's which meet this restriction which are arbitrary close to any values we choose [Matula, 1964].

As stated in Ahlswede and Wegener [1987], (regarding discrete search problems with switching costs): "...almost all interesting questions are, however, still open."

VIII

REFERENCES AND BIBLIOGRAPHY

Ahlswede, Rudolph and Wegener, Ingo. 1987. *Search Problems.* John Wiley & Sons, New York.

Aigner, Martin. 1988. *Combinatorial Search.* John Wiley and Sons, Chichester.

Arkin, V. I. 1964. A Problem of Optimum Distribution of Search Effort. *Theory of Probability and Its Applications* **9**, 159-160.

Beightler, Charles S.; Phillips, Don T.; and Wilde, Douglas J. 1979. *Foundations of Optimization.* Prentice Hall, Inc., New Jersey.

Bellman, Richard. 1957. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey.

Bellman, Richard. 1963. Problem 63-9, An Optimal Search. *SIAM Review* **5**, 274. (Solution given in Franck, 1965.)

Berry, Donald A. and Mensch, Roy F. 1986. Discrete Search with Directional Information. *Operations Research* **34**, 470-477.

Blachman, Nelson M. 1959. Prolegomena to Optimum Discrete Search Procedures. *Naval Research Logistics Quarterly* **6**, 273-281.

Blachman, Nelson and Proschan, Frank. 1959. Optimum Search for Objects Having Unknown Arrival Times. *Operations Research* **7**, 625-638.

Black, William L. 1965. Discrete Sequential Search. *Information and Control* **8**, 159-162.

Brown, Scott S. 1980. Optimal Search for a Moving Target in Discrete Time and Space. *Operations Research* **28**, 1275-1289.

Charnes, A. and Cooper W. W. 1958. The Theory of Search: Optimum distribution of Search Effort. *Management Science* **5**, 44-50.

Chew, Milton C. Jr. 1967. A Sequential Search Procedure. *The Annals of Mathematical Statistics* **38**, 494-502.

Chudnovsky, David V. and Chudnovsky, Gregory V. 1989. *Search Theory (Some Recent Developments).* Marcel Dekker, Inc., New York.

Danskin, John M. 1966. A Helicopter Versus Submarine Search Game. *Operations Research* **16**, 509-517.

de Guenin, Jacques. 1961. Optimum Distribution of Effort: an Extension of the Koopman Basic Theory. *Operations Research* **9**, 1-7.

Discenza, Joseph H. and Stone, Lawrence D. 1981. Optimal Survivor Search with Multiple States. *Operations Research* **29**, 309-323.

Dobbie, James M. 1963. Search Theory: A Sequential Approach. *Naval Research Logistics Quarterly* **10**, 323-334.

Dobbie, James M. 1968. A Survey of Search Theory. *Operations Research* **16**, 525-537.

Dobbie, James M. 1975. Search for an Avoiding Target. *SIAM Journal of Applied Mathematics* **28**, 72-86.

Dreyfus, Stuart E. and Law, Averill M. 1977. *The Art and Theory of Dynamic Programming*. Academic Press, New York.

Eagle, James N. 1984. The Optimal Search for a Moving Target When the Search Path is Constrained. *Operations Research* **32**, 1107-1115.

Eagle, James N. and Yee, James R. 1990. An Optimal Branch-and-Bound Procedure for the Constrained Path, Moving Target Search Problem. *Operations Research* **38**, 110-114.

Enslow, Philip H., Jr. 1966. A Bibliography of Search Theory and Reconnaissance Theory Literature. *Naval Research Logistics Quarterly* **13**, 177-202.

Franck, Wallace. 1965. An Optimal Search Problem. *SIAM Review* **7**, 503-512.

Gal, Shmuel. 1980. *Search Games.* Academic Press, Inc., New York.

Gilbert, E. N. 1959. Optimal Search Strategies. *Journal of the Society for Industrial and Applied Mathematics* **7**, 413-424.

Gilbert, E. N. 1962. Games of Identification or Convergence. *SIAM Review* **4**, 16-24.

Gluss, Brian. 1961a. An Alternative Solution to the "Lost at Sea" Problem. *Naval Research Logistics Quarterly* **8**, 117-122.

Gluss, Brian. 1961b. The Minimax Path in a Search for a Circle in a Plane. *Naval Research Logistics Quarterly* **8**, 357-360.

Gumacos, Constantine. 1963. Analysis of an Optimum Sync Search Procedure. *IEEE Transactions on Communications Systems* **CS-11**, 89-99.

Hassin, Refael. 1984. A dichotomous Search for a Geometric Random Variable. *Operations Research* **32**, 423-439.

Isbel, J.R. 1957. An Optimal Search Pattern. *Naval Research Logistics Quarterly* **4**, 357-359.

Kadane, Joseph B. 1968. Discrete Search and the Neyman-Pearson Lemma. *Journal of Mathematical Analysis and Applications* **22**, 156-171.

Kadine, Joseph B. and Simon, Herbert A. 1977. Optimal Strategies for a Class of Constrained Sequential Problems. *The Annals of Statistics* **5**, 237-255.

Kisi, Takasi. 1966. On an Optimal Searching Schedule. *Journal of the Operations Research Society of Japan* **8**, 53-65.

Klein, Morton. 1968. A Note on Sequential Search. *Naval Research Logistics Quarterly* **15**, 469-475.

Koopman, Bernard O. 1952. New Mathematical Methods in Operations Research. *Journal of the Operations Research Society of America* **1**, 3-9.

Koopman, Bernard O. 1953. The Optimum Distribution of Effort. *Journal of the Operations Research Society of America* **1**, 52-63.

Koopman, Bernard O. 1956a. The theory of Search I. Kinematic Bases. *Operations Research* **4**, 324-346.

Koopman, Bernard O. 1956b. The theory of Search II. Target Detection. *Operations Research* **4**, 503-531.

Koopman, Bernard O. 1957. The theory of Search III. The Optimum Distribution of Searching Effort. *Operations Research* **5**, 613-626.

Koopman, Bernard O. 1980. *Search and Screening.* Pergamon Press, New York.

Law, Averill M. and Kelton, David W. 1991. *Simulation Modeling and Analysis*. McGraw-Hill, Inc., New York.

Land, A. H. and Doig, A. G. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica* **28**, 497-520.

Lossner, Udo and Wegener, Ingo. Discrete Sequential Search with Positive Switch Cost. *Mathematics of Operations Research* **7**, 426-440.

Matula, David. 1964. A Periodic Optimal Search. *American Mathematical Monthly* **71**, 15-21.

Moore, Michael L. 1970. *A Review of Search and Reconnaissance Theory Literature*. Management Information Services, Detroit, Michigan.

Moorse, Phillip M. 1948. Mathematical Problems in Operations Research. *Bulletin of the American Mathematical Society* **54**, 602-621.

Moorse, Phillip M. 1953. Trends in Operations Research. *Journal of the Operations Research Society of America* **1**, 159-165.

Moorse, Philip M. 1982. Bernard Osgood Koopman, 1900-1981. *Operations Research* **30**, 417-427.

McDonald, A. M. C., Fergusson, J.G., and Elliott, R.W. 1962. Theory of Search in *Some Techniques of Operational Research*. The English Universities Press, London.

Onaga, Kenji. 1971. Optimal Search for Detecting a Hidden Object. *SIAM Journal of Applied Mathematics* **20**, 298-318.

Pollock, Stephen M. 1970. A Simple Model of Search for a Moving Target. *Operations Research* **18**, 883-903.

Posner, Edward C. 1963. Optimal Search Procedures. *IEEE Transactions on Information Theory* **IT-9**, 157-160.

Richardson, Henry R. and Stone, Lawrence D. 1971. Operations Analysis During the Underwater Search for Scorpion. *Naval Research Logistics Quarterly* **18**, 141-157.

Ruckle, W H. 1983. *Geometric Games and their Applications.* Pitman Advanced Publishing Program, Boston.

Smith, Mark W. 1971. Optimum Sequential Search with Discrete Locations and Random Acceptance Errors. *Naval Research Logistics Quarterly* **18**, 159-167.

Staroverov, O. V. 1963. On a Searching Problem. *Theory of Probability and Its Applications* **8**, 184-187.

Stone, Lawrence D. 1975. *Theory of Optimal Search.* Academic Press, New York.

Stone, Lawrence D. and Kadane, Joseph B. 1981. Optimal Whereabouts Search for A Moving Target. *Operations Research* **29**, 1154-1166.

Stone, Lawrence D. 1983. The Process of Search Planning: Current Approaches and Continuing Problems. *Operations Research* **31**, 207-233.

Stone, Lawrence D. 1989. What's Happened in Search Theory Since the 1975 Lanchester Prize? *Operations Research* **37**, 501-506.

Thomas, Lyn C. and Washburn, Alan R. 1991. Dynamic Search Games. *Operations Research* **39**, 415-422.

Trummel, K.E. and Weisinger, J.R. 1986. The Complexity of the Optimal Searcher Path Problem. *Operations Research* **34**, 324-327.

Washburn, Alan R. 1980a. On a Search for a Moving Target. *Naval Research Logistics Quarterly* **27**, 315-322.

Washburn, Alan R. 1980b. Search-Evasion Game in a Fixed Region. *Operations Research* **28**, 1290-1298.

Washburn, Alan R. 1981. An Upper Bound Useful in Searching for a Moving Target. *Operations Research* **29**, 1227-1230.

Washburn, Alan R. 1983. Search for a Moving Target: the FAB Algorithm. *Operations Research* **31**, 739-751.

Wegener, Ingo. 1980. The Discrete Search Problem with Nonrandom Cost and Overlook Probabilities. *Mathematics of Operations Research* **5**, 373-380.

Wegener, Ingo. 1982. The Discrete Search Problem and the Construction of Optimum Allocations. *Naval Research Logistics Quarterly* **29**, 203-212.

Whitmore, William F. 1953. Edison and Operations Research. *Journal of the Operations Research Society of America* **1**, 83-85.

Wong, Eugene. 1964. A Linear Search Problem. *SIAM Review* **6**, 168-174.

Zahl, Samuel. 1963. An Allocation Problem with Applications to Operations Research and Statistics. *Operations Research* **11**, 426-441.

Zimmerman, Seth. 1959. An Optimal Search Procedure. *American Mathematical Monthly* **66**, 690-693.

APPENDIX A

PROGRAM CODE

The program code is given in three sections: SEARCH_1.PAS; SEARCH_2.PAS; and MENU.PAS. All programs are written using Turbo Pascal 6.0.

The first program, SEARCH_1.PAS, is interactive and lets the user specify ranges for the parameters of the problem. The program then generates a problem randomly, calculates the heuristic policy, and generates a number of random policies.

The second program, SEARCH_2.PAS performs the same calculations, but is not interactive. It is this program which generated the main data for this thesis.

The third program. MENU.PAS, is a unit which holds the code for the menu and some other frequently used routines.

```
{ SEARCH_1.PAS

     This interactive program compares the heuristic policy
to a user defined number of random policies.  Random
problems are generated using input parameters provided by
the user.

}

{================== program identification ================}

program search_1;

uses menu, crt;

{================== types and declarations ================}

type
   coef = array[1..100] of single;                     {a,b,c,p}
   coef = array[0..100] of coef;            { movement costs }
   count_array = array[1..100] of word;
                                 { times_searched per cell }
   search_array = array[0..150] of integer;
                                         { search policies }
   parameter_rec = record       { parameters for a problem }
      number_of_cells    : integer;
      min_movement_cost  : integer;
      max_movement_cost  : integer;
      min_search_cost    : integer;
      max_search_cost    : integer;
      number_of_random_policies : integer;
      number_of_tests    : integer;
      file_option        : integer;
      data_name          : string;
      pause_between_tests: boolean;
   end;

var
   a, b, c, p : coef;    { a[i] = overlook prob for cell i }
                            { b[i] is the compliment of a }
                         { c[i] is the cost of searching i }
                        { p[i] is the a priori prob dist.. }

   m : m_coef;           { m[i][j] = move cost from i to j }
   data, try_data        : text;    { output file variable }
   data_name             : string;      { output file name }
   times_wrong           : integer;
   tolerance             : real;
   problem               : byte;
   big_m                 : real;
   policy_was_successful : boolean;
   done                  : boolean;
```

```
{================= initial procedures ==================}

procedure set_defaults;
   begin
      with parameters do
      begin
         number_of_cells      := 5;        { target locations }
         min_movement_cost    := 0;       { switch cost range }
         max_movement_cost    := 0;
         min_search_cost      := 1;       { search cost range }
         max_search_cost      := 10;
         number_of_random_policies := 20;
                     { no. of random policies per problem }
         number_of_tests      := 1;
                  { different tests with the same parameters }
         file_option          := 0;    { 0 --> no output file }
         data_name            := 'no output file';
                                         { default file name }
         pause_between_tests:= false;
      end;
      tolerance := 0.001;     { 1 - tolerance = P(success) }
      big_m := 99999999.9;    { an initial comparing value }
   end;  { set_defaults }

procedure open_file;                 { opens a file with a file }
   var                             { name based on the parameters }
      cells : string[2];
      move  : string[3];
      search: string[3];
   begin
      if parameters.file_option = 4 then
         begin
         str(parameters.number_of_cells, cells);
         str(parameters.max_movement_cost, move);
         str(parameters.max_search_cost, search);
         parameters.data_name := cells + '_' + search + '_'
            + move +'.TXT';
         end;
      assign(data, parameters.data_name);
      if parameters.file_option = 2 then
         append(data)
      else
         rewrite(data);
   end;  { open_file }


procedure write_defaults_to_file;
   begin
      writeln(data);
      writeln(data, ' Number of cells = ',
         parameters.number_of_cells:3);
```

```
      writeln(data, ' Max Move Cost =    ',
         parameters.max_movement_cost:3);
      writeln(data, ' Max Search Cost = ',
         parameters.max_search_cost:3);
      writeln(data, ' Tolerance =        ', tolerance:10:8);
      writeln(data);
   end;  { write_defaults_to_file }

procedure initialize;                  { initial procedures }
   begin
      randomize;
      textcolor(3);
      if parameters.file_option > 0 then
         begin
         open_file;
         write_defaults_to_file;
         end;
   end;  { initialize }

procedure finalize;                 { close the output file }
   begin
      if parameters.file_option > 0 then close(data);
   end;

procedure write_problem_data_to_disk;
   var
      i, j : integer;
   begin
      writeln(data, ' Problem ', problem);
      writeln(data);
      for i := 1 to parameters.number_of_cells do
         begin
         if i = 1 then
            writeln(data, ' i   a[i]    b[i]   c[i]   p[i]',
                          m[i-1][i] matrix ');
         write(data, i:3, '  ', a[i]:4:4, '  ',b[i]:4:4,
               '  ', c[i]:4:1, '  ',p[i]:6:6, '   ');
         if parameters.max_movement_cost > 0 then
            begin
            write(data, ' ', m[0][i]:4:1);
            for j := 1 to parameters.number_of_cells do
               write(data, '  ', m[i][j]:4:1);
            writeln(data);
            end
         else
            if i = 2 then
               writeln(data, '  no movement costs')
            else
               writeln(data);
         end;
   end;  { write_problem_data_to_disk }
```

```
procedure write_problem_to_screen;
   var
      i, j : integer;
   begin
   for i :=1 to parameters.number_of_cells do
      begin
      if i = 1 then
         begin
         textcolor(14);
         gotoxy(1, 4);
         writeln(' i    a[i]    b[i]   c[i]   p[i]',
            '            m[i-1][i] matrix ');
         textcolor(3);
         end;
      if i <= 10 then
         begin
         write(i:3, '  ', a[i]:4:4, '  ',b[i]:4:4, '  ',
               c[i]:4:1, '  ',p[i]:6:6, '   ');
         if parameters.max_movement_cost > 0 then
            begin
            write(' ', m[0][i]:4:1);
            for j := 1 to min(parameters.number_of_cells, 5)
               do
               write('  ', m[i][j]:4:1);
               if parameters.number_of_cells > 5 then
                  writeln(' ... ')
               else
                  writeln;
            end
         else
            if i = 2 then
               writeln('  no movement costs')
            else
               writeln;
         end;
      if i = 11 then
         writeln(' Remaining cell parameters not shown. ');
      end;
   end;  { write_problem_to_screen }


{==================== main procedures ====================}

procedure generate_random_search_problem;
   var                        { sets up a random test problem }
      percentage_left     : real;
      multiplier          : real;
      index               : integer;
      i, j                : integer;
      movement_cost_range : integer;
      search_cost_range   : integer;
```

```
begin
   search_cost_range := parameters.max_search_cost -
                        parameters.min_search_cost;
   movement_cost_range := parameters.max_movement_cost -
                          parameters.min_movement_cost;
   percentage_left := 1.00;
   for j := 1 to parameters.number_of_cells do
                                   { initial movement cost }
      m[0][j] := random(movement_cost_range) +
                 parameters.min_movement_cost;
   for i := 1 to parameters.number_of_cells do
                                        { fill a, b, c, p  }
      begin
      a[i] := random * 0.5;       { overlook probability }
      b[i] := 1 - a[i];        { prob find | it is there }
      c[i] := random(search_cost_range) +
         parameters.min_search_cost;
      case parameters.number_of_cells of
         1..5    : multiplier := 1.0;
         6..10   : multiplier := 0.8;
         11..20  : multiplier := 0.6;
         21..40  : multiplier := 0.4;
         41..60  : multiplier := 0.3;
         else multiplier := 0.2
      end;
      if i = parameters.number_of_cells then
                              { assign remaining percent }
         p[i] := percentage_left
      else
         begin
            p[i] := random * multiplier;
                            { assign a random value to p }
            if p[i] > percentage_left then
                              { if it is too much then.. }
               p[i] := percentage_left/2;
                            {  take half of what is left }
            percentage_left := percentage_left - p[i];
         end;
      if parameters.max_movement_cost > 0 then
         begin
            for j := 1 to parameters.number_of_cells do
               m[i][j] := random(movement_cost_range) +
                          parameters.min_movement_cost;
               m[i][i] := 0.00;
         end;
      if parameters.max_movement_cost = 0 then
         for j := 0 to parameters.number_of_cells do
            m[j][i] := 0.00;
      end;
   write_problem_to_screen;
   if parameters.file_option > 0 then
      write_problem_data_to_disk;
```

```
   end;  { generate_random_search_problem }

procedure output_heuristic_to_screen
                              (heuristic_policy : search_array;
                                 heuristic_cost : real;
                                          steps : integer);
   var
      i : integer;
   begin
      gotoxy(2, 18);
         write('Heuristic cost = ');
         textcolor(14);
         writeln(heuristic_cost:12:6);
         textcolor(3);
         writeln(' with policy: ');
      for i := 1 to min(10, steps) do
         write('  ', heuristic_policy[i]);
      writeln('...  ');
      write(' which took ');
      textcolor(14);
      write(steps);
      textcolor(3);
      write(' searches.  ');
   end;  { output_heuristic_to_screen }

procedure output_random_to_screen(policy : search_array;
                                  cost   : real;
                                  k      : integer;
                                  steps  : integer);
   var
      i : integer;
   begin
      gotoxy(38, 18);
         write('Best random cost = ');
         textcolor(14);
         write(cost:12:6);
         textcolor(3);
      gotoxy(38, 19);
         write('with policy number ', k, ':  ');
      gotoxy(38, 20);
      for i := 1 to min(10, steps) do
         write('  ', policy[i]);
      writeln('...   ');
      gotoxy(38, 21);
      write('which took ');
      textcolor(14);
      write(steps);
      textcolor(3);
      write(' searches. ');
   end;  { output_random_to_screen }

procedure output_heuristic_to_disk
```

```
                              (heuristic_policy : search_array;
                                heuristic_cost : real;
                                         steps : integer);
   var
      i : integer;
   begin
      writeln(data);
      writeln(data, ' Report for problem ', problem,'.');
      writeln(data);
      writeln(data, ' Heuristic Policy');
      writeln(data);
      writeln(data, '    searches   cost       first 30
         steps of policy');
      write(data, '  ', steps:5, '  ',
         heuristic_cost:14:6,'  ');
      for i := 1 to min(30, steps) do
         write(data, ' ', heuristic_policy[i]:2);
      writeln(data);
      writeln(data);
      writeln(data, ' Random Policies');
      writeln(data, ' #  searches   cost       first 30
         steps of policy');
      writeln(data);
   end;  { output_heuristic_to_disk }

procedure output_random_to_disk(k  : integer;
                     random_policy : search_array;
                              cost : real;
                             steps : integer);
   var
      i : integer;
   begin
      if k < 10 then write(data, ' ');
      write(data, k, '  ', steps:5, '  ', cost:14:6,'  ');
      for i := 1 to min(30, steps) do
         write(data, ' ', random_policy[i]:2);
      writeln(data);
   end;

procedure output_best_to_disk( random_policy : search_array;
                                        cost : real;
                                       steps : integer);
   var
      i : integer;
   begin
      writeln(data);
      writeln(data, ' Best Random Policy ');
      writeln(data);
      write(data,'    ', steps:5, '  ', cost:14:6,'  ');
      for i := 1 to min(30, steps) do
         v~ite(data, ' ', random_policy[i]:2);
      writ. n(data);
```

```
   end;  { output_best_to_disk }

procedure conduct_search;
   var
      sum_of_costs      : real;
      expected_cost     : real;
      heuristic_cost    : real;
      best_cost         : real;
      prob_left         : real;
      a_to_m            : real;
      tail_probability  : real;
      times_searched    : count_array;
      policy            : search_array;
      best_policy       : search_array;
      i, j              : integer;
      last_cell         : integer;
      next_cell         : integer;
      best_length       : integer;

   procedure reset_evaluate;
      var
         k : integer;
      begin
         prob_left := 1.00;
         sum_of_costs := 0.00;
         expected_cost := 0.00;
         for k := 1 to parameters.number_of_cells do
            times_searched[k] := 0;
         i := 1;
      end;

   procedure evaluate_next_cell(last, next : integer);
      begin
         sum_of_costs := sum_of_costs + m[last][next] +
            c[next];
         a_to_m := power(a[next], times_searched[next]);
         tail_probability := p[next] * b[next] *  a_to_m;
         prob_left := prob_left - tail_probability;
         expected_cost := expected_cost +
            (tail_probability * sum_of_costs);
         inc(times_searched[next]);
      end; { evaluate_next_cell }

   function get_next_heuristic_cell(last : integer) :
         integer;     var
         a_to_m, ratio, best_ratio : real;
         k : integer;

      begin
         best_ratio := 0;
         for k := 1 to parameters.number_of_cells do
            begin
```

```
                        { find the best cell to search next }
            a_to_m := power(a[k], times_searched[k]);
            ratio :=  (a_to_m * b[k] * p[k])/
                      (c[k] + m[last][k]);
            if (ratio > best_ratio) then
                              { the larger ratio is better }
               begin
               get_next_heuristic_cell := k;
               best_ratio := ratio;
               end;
            end;
      end; { get_next_heuristic_cell }

   procedure heuristic_search;
      begin;
      gotoxy(14, 2);
      reset_evaluate;
      policy[0] := 0;
      next_cell := 0;
      while (prob_left > tolerance) do
         begin
         last_cell := next_cell;
         next_cell := get_next_heuristic_cell(last_cell);
         evaluate_next_cell(last_cell, next_cell);
         policy[i] := next_cell;
         inc(i);
         if 500 mod i = 0 then write(' .');
         end;
      gotoxy(14, 2);
      write('                                                   ');
      if parameters.file_option > 0 then
         output_heuristic_to_disk
            (policy, expected_cost, i-1);
      output_heuristic_to_screen
            (policy, expected_cost, i-1);
      end;  { heuristic_search }

   procedure random_search;
      begin
         reset_evaluate;
         policy[0] := 0;
         next_cell := 0;
         while (prob_left > tolerance) do
            begin
            last_cell := next_cell;
            next_cell := random
               (parameters.number_of_cells) + 1;
            if i < 100 then policy[i] := next_cell;
            evaluate_next_cell(last_cell, next_cell);
            inc(i);
            end;
         gotoxy(38, 16);
```

```
            write('Random Policy ', j, ' cost = ',
               expected_cost:10:6);
            if parameters.file_option > 0 then
               output_random_to_disk
                  (j, policy, expected_cost, i-1);
         end;  { random_search }

      begin  { conduct_search }
         policy[0] := 0;
         heuristic_search;
         heuristic_cost := expected_cost;
         best_cost := big_m;
         for j := 1 to parameters.number_of_random_policies do
            begin
            random_search;
            if expected_cost < heuristic_cost then
               inc(times_wrong);
            if expected_cost < best_cost then
               begin
               best_policy := policy;
               best_cost := expected_cost;
               best_length := i-1;
               output_random_to_screen
                  (policy, expected_cost, j, best_length);
               end;
            end;
         if parameters.file_option > 0 then
            output_best_to_disk
               (best_policy, best_cost, best_length);
      end;  { conduct_search }

   {==================== final procedures ===================}

   procedure ask_to_continue;
      var
         answer : char;

      begin
         clrscr;
         gotoxy(2, 2);
            textcolor(3);
            write('More searches? ');
            textcolor(14);
            write('(y/n) ');
         answer := readkey;
         if upcase(answer) = 'N' then
               done := true;
         if done then
            begin
               clrscr;
               gotoxy(37, 12);
                  textcolor(14);
```

```
            write('Bye!');
         hold;
         clrscr;
      end;
end;


{==================== program main =====================}

begin {main}
   clrscr;
   done := false;
   set_defaults;
   while not done do
   begin
      search_menu;
      initialize;
      times_wrong := 0;
      for problem := 1 to parameters.number_of_tests do
      begin
         gotoxy(2, 2);
         write(' Problem ', problem);
         generate_random_search_problem;
         conduct_search;
         if parameters.pause_between_tests then
            begin
               gotoxy(2, 24);
               write('Press a key...');
               hold;
               gotoxy(2, 24);
               write('                ');
            end;
      end;
      finalize;
      gotoxy(2, 24);
      write('Total times wrong for ',
            parameters.number_of_tests, ' problems = ',
            times_wrong, '.   ');
      hold;
      ask_to_continue;
   end;
end.  { main }

{=================== end of SEARCH_1.PAS ================}
```

```
{ SEARCH_2.PAS

   This program is used to generate the main data.

}

{================== program identification ================}

program search_2;

uses menu, crt;

{================== types and declarations ================}

type
   coef = array[1..100] of single;                { a, c, pb }
   m_coef = array[0..100] of coef;           { movement costs }
   count_array = array[1..100] of word;
                                          { times_searched per cell }
   search_array = array[0..150] of integer;
                                                  { search policies }
   parameter_rec = record          { parameters for a problem }
      number_of_cells     : integer;
      min_movement_cost   : integer;
      max_movement_cost   : integer;
      min_search_cost     : integer;
      max_search_cost     : integer;
      number_of_random_policies : longint;
      number_of_tests     : integer;
      file_option         : integer;
      data_name           : string;
      pause_between_tests: boolean;
   end;

var
   a, c, pb : coef;        { a[i] = overlook prob for cell i }
                                   { b is the compliment of a }
                           { c[i] is the cost of searching i }
                             { p is the a priori prob dist.. }
                       { .. ie. prob the target is in cell i }

   m : m_coef;             { m[i][j] = move cost from i to j }

   data, try_data         : text;    { output file variable }
   data_name              : string;      { output file name }
   tolerance              : real;
   problem                : byte;
   big_m                  : real;
   policy_was_successful  : boolean;
   done                   : boolean;
   cell_steps             : integer;
   cell_data              : array[1..10] of integer;
```

```
   search_steps          : integer;
   search_data           : array[1..2] of array[1..10] of
                           integer;
   move_steps            : integer;
   move_data             : array[1..2] of array[1..10] of
                           integer;
   hh, ii, jj            : integer;
   parameters            : parameter_rec;
   counter               : integer;

{================= initial procedures ===================}

procedure set_defaults;
   begin
      textcolor(3);
      with parameters do
         begin
         number_of_random_policies := 1000000
                               { no. of policies per problem }
         number_of_tests            := 1;
                            { tests with the same parameters }
         file_option                := 1;
                                { option = 1 --> output file }
         counter                    := 1;
         end;

      cell_steps := 4;
               { number of changes in the number of cells }
         cell_data[1] := 5;
         cell_data[2] := 10;
         cell_data[3] := 25;
         cell_data[4] := 50;

      search_steps := 1;
                      { number of changes in search costs }
         search_data[1][1] := 1;  { min & max search costs }
         search_data[2][1] := 10;

      move_steps := 5;
                    { number of changes in movement costs }

         move_data[1][1] := 0;  { min & max movement costs }
         move_data[2][1] := 0;

         move_data[1][2] := 1;
         move_data[2][2] := 4;

         move_data[1][3] := 1;
         move_data[2][3] := 10;

         move_data[1][4] := 90;
         move_data[2][4] := 100;
```

```
         move_data[1][5] := 1;
         move_data[2][5] := 100;

      tolerance := 0.0001;    { 1 - tolerance = P(success) }
      big_m := 99999999.9;    { an initial comparing value }
   end;  { set_defaults }

procedure open_file;       { opens a file with default name }
   var
      cells  : string[3];
      move   : string[1];
      prob   : string[2];
   begin
      str(parameters.number_of_cells, cells);
      str(jj, move);
      str(problem, prob);
      parameters.data_name := prob + '_' + cells + '_' +
         move +'.TXT';
      assign(data, parameters.data_name);
      rewrite(data);
   end;  { open_file }

procedure write_defaults_to_file;
   begin
      writeln(data);
      writeln(data, ' Parameters for all problems in this
          file:');
      writeln(data);
      writeln(data, ' Number of cells = ',
         parameters.number_of_cells:3);
      writeln(data, ' Min Search Cost = ',
         parameters.min_search_cost:3);
      writeln(data, ' Max Search Cost = ',
         parameters.max_search_cost:3);
      writeln(data, ' Min Move Cost =   ',
         parameters.min_movement_cost:3);
      writeln(data, ' Max Move Cost =   ',
         parameters.max_movement_cost:3);
      writeln(data, ' Tolerance =      ', tolerance:5:3);
   end;  { write_defaults_to_file }

procedure initialize;
   begin
      if parameters.file_option > 0 then
         begin
         open_file;
         write_defaults_to_file;
         end;
   end;  { initialize }

procedure finalize;
   begin
```

```
      if parameters.file_option > 0 then close(data);
   end;

procedure write_problem_data_to_disk;
   var
      prob      : string[2];
      cells     : string[3];
      move      : string[1];
      prob_data : text;
      i,j       : integer;
   begin
      str(problem, prob);
      str(parameters.number_of_cells, cells);
      str(jj, move);
      data_name := prob + '_' + cells + '_' + move + '.DAT';
      assign(prob_data, data_name);
      rewrite(prob_data);
      writeln(prob_data);
      writeln(prob_data, ' Parameters for this file:');
      writeln(prob_data);
      writeln(prob_data, ' Number of cells = ',
         parameters.number_of_cells:3);
      writeln(prob_data, ' Min Search Cost = ',
         parameters.min_search_cost:3);
      writeln(prob_data, ' Max Search Cost = ',
         parameters.max_search_cost:3);
      writeln(prob_data, ' Min Move Cost =   ',
         parameters.min_movement_cost:3);
      writeln(prob_data, ' Max Move Cost =   ',
         parameters.max_movement_cost:3);
      writeln(prob_data, ' Tolerance =     ',
         tolerance:10:8);
      writeln(prob_data);

      for i := 1 to parameters.number_of_cells do
         begin
         if i = 1 then
            writeln(prob_data, ' i   a[i]    b[i]      p[i]
     c[i]',
                   '          m[h][i] matrix');
         if i < 10 then
            write(prob_data, ' ');
         write(prob_data, i, '  ', a[i]:4:4, '  ',
            (1-a[i]):4:4, '    ',pb[i]/(1-a[i]):14,
            '    ', m[i][i]:4:1, '    ');
         if parameters.max_movement_cost > 0 then
            begin
            write(prob_data, ' ', (m[0][i]-m[i][i]):5:1);
            for j := 1 to parameters.number_of_cells do
               write(prob_data, ' ', (m[i][j]-m[i][i]):5:1);
            writeln(prob_data);
            end
```

```
        else
           if i = 2 then
              writeln(prob_data, '  no movement costs')
           else
              writeln(prob_data);
        end;
        close(prob_data);
  end;  { write_problem_data_to_disk }

{==================== main procedures ====================}

procedure generate_random_search_problem;
                              { sets up a random test problems }
  var
     percentage_left     : real;
     multiplier          : real;
     b, p                : real;
     index               : integer;
     i, j                : integer;
     movement_cost_range : integer;
     search_cost_range   : integer;

  begin
     randomize;
     search_cost_range := parameters.max_search_cost -
                          parameters.min_search_cost + 1;
     percentage_left := 1.00;
     for i := 1 to parameters.number_of_cells do
                                                  { fill a, c, pb }
        begin
        a[i] := random * 0.5;     { overlook probability }
        b    := 1 - a[i];       { prob find | it is there }
        c[i] := random(search_cost_range) +
                parameters.min_search_cost;
        case parameters.number_of_cells of
           1..5    : multiplier := 1.0;
           6..10   : multiplier := 0.8;
           11..20  : multiplier := 0.5;
           21..40  : multiplier := 0.2;
           41..60  : multiplier := 0.1;
           else multiplier := 0.05
        end;
        if i = parameters.number_of_cells then
                                  { assign remaining percent }
           p := percentage_left
        else
           begin
              p := random * multiplier;
                               { assign a random value to p }
              if p > percentage_left then
                                  { if it is too much then.. }
                 p := percentage_left/2;
```

```
                                { take half of what is left }
                    percentage_left := percentage_left - p;
                 end;
              pb[i] := p*b;
              end;
        end; { generate_random_search_problem }


     procedure generate_random_move_costs;
        var
           i, j                  : integer;
           movement_cost_range : integer;
2        begin
           movement_cost_range := parameters.max_movement_cost -
                                  parameters.min_movement_cost + 1;
           if parameters.max_movement_cost > 0 then
              for i := 1 to parameters.number_of_cells do
                 begin
                 for j := 1 to parameters.number_of_cells do
                    m[i][j] := random(movement_cost_range) +
                        parameters.min_movement_cost + c[i];
                 m[i][i] := c[i];         { thus m[i][i] = 0.00 }
                 m[0][i] := random(movement_cost_range) +
                              parameters.min_movement_cost + c[i];
                 end
           else
              for i := 1 to parameters.number_of_cells do
                 for j := 0 to parameters.number_of_cells do
                    m[j][i] := c[i];
           if parameters.file_option > 0 then
              write_problem_data_to_disk;
        end;

     procedure output_heuristic_to_disk
                                  (heuristic_policy : search_array;
                                     heuristic_cost : real;
                                              steps : integer);
        var
           i : integer;
        begin
           writeln(data);
           writeln(data, ' Report for problems ', problem,'.');
           writeln(data);
           writeln(data, 'searches      cost       first 30 steps
              of policy');
           writeln(data);
           writeln(data, ' Heuristic Policy');
           writeln(data);
           write(data, ' ', steps:4, '  ',
              heuristic_cost:14:6,'  ');
           for i := 1 to min(steps, 30) do
              write(data, ' ', heuristic_policy[i]:2);
```

```
      writeln(data);
   end;  { output_heuristic_to_disk }

procedure write_header_to_disk;
   begin
      writeln(data);
      writeln(data, ' Problem ', problem);
      writeln(data);
      writeln(data, ' Random Policies');
      writeln(data, ' #  searches   cost');
      writeln(data);
   end;  { output_heuristic_to_disk }

procedure output_random_to_disk( k  : integer;
                                cost : real;
                               steps : integer);
   var
      i : integer;
   begin
      writeln(data, steps:6, '    ', cost:14:6);
   end;

procedure output_best_to_disk( random_policy : search_array;
                                        cost : real;
                                       steps : integer);
   var
      i : integer;
   begin
      writeln(data);
      writeln(data, ' Best Random Policy ');
      writeln(data);
      write(data, ' ', steps:4, '   ', cost:14:6,'  ');
      for i := 1 to min(30, steps) do
         write(data, ' ', random_policy[i]:2);
      writeln(data);
      writeln(data);
   end;  { output_best_to_disk }

procedure conduct_search;
   var
      sum_of_costs      : real;
      expected_cost     : real;
      heuristic_cost    : real;
      best_cost         : real;
      prob_left         : real;
      a_to_m            : real;
      tail_probability  : real;
      times_searched    : count_array;
      policy            : search_array;
      best_policy       : search_array;
      heuristic_policy  : search_array;
      i, j              : longint;
```

```
   last_cell         : integer;
   next_cell         : integer;
   best_length       : integer;
   heuristic_length  : integer;

procedure reset_evaluate;
   var
      k : integer;
   begin
      prob_left := 1.00;
      sum_of_costs := 0.00;
      expected_cost := 0.00;
      for k := 1 to parameters.number_of_cells do
         times_searched[k] := 0;
      i := 1;
   end;

procedure evaluate_next_cell(last, next : integer);
   begin
      sum_of_costs := sum_of_costs +
                      m[last][next] + c[next];
      a_to_m := power(a[next], times_searched[next]);
      tail_probability := pb[next] *  a_to_m;
      prob_left := prob_left - tail_probability;
      expected_cost := expected_cost +
         (tail_probability * sum_of_costs);
      inc(times_searched[next]);
   end; { evaluate_next_cell }

function get_next_heuristic_cell
                        (last : integer) : integer;
   var
      a_to_m, ratio, best_ratio : real;
      k : integer;

   begin
      best_ratio := 0;
      for k := 1 to parameters.number_of_cells do
         begin      { find the best cell to search next }
         a_to_m := power(a[k], times_searched[k]);
         ratio :=  (a_to_m * pb[k])/m[last][k] {+ c[k]};
         if (ratio > best_ratio) then
                           { the larger ratio is better }
            begin
            get_next_heuristic_cell := k;
            best_ratio := ratio;
            end;
         end;
   end; { get_next_heuristic_cell }

procedure heuristic_search;
   begin;
```

```
   reset_evaluate;
   policy[0] := 0;
   next_cell := 0;                        { start at cell "0" }
   while (prob_left > tolerance) do
      begin
      last_cell := next_cell;
      next_cell := get_next_heuristic_cell(last_cell);
      evaluate_next_cell(last_cell, next_cell);
      if i < 50 then
         policy[i] := next_cell;
      inc(i);
      end;
   end;  { heuristic_search }

procedure random_search;
   begin
      reset_evaluate;
      policy[0] := 0;
      next_cell := 0;
      while (prob_left > tolerance) do
         begin
         last_cell := next_cell;
         next_cell :=
                  random(parameters.number_of_cells) + 1;
         if i < 30 then policy[i] := next_cell;
         evaluate_next_cell(last_cell, next_cell);
         inc(i);
         end;
   end;  { random_search }

begin  { conduct_search }
   policy[0] := 0;
   heuristic_search;
   heuristic_policy := policy;
   heuristic_length := i-1;
   heuristic_cost := expected_cost;
   best_cost := big_m;
   write_header_to_disk;
   for j := 1 to parameters.number_of_random_policies do
      begin
      random_search;
      output_random_to_disk(j, expected_cost, i-1);
      if expected_cost < best_cost then
         begin
         best_policy := policy;
         best_cost := expected_cost;
         best_length := i-1;
         end;
      end;
   output_heuristic_to_disk
      (heuristic_policy,heuristic_cost,heuristic_length);
   output_best_to_disk
```

```
          (best_policy, best_cost, best_length);
   end;   { conduct_search }

{==================== screen output =====================}

procedure write_parameters_to_screen;
   var
      l : integer;
   begin
      gotoxy(2, 3);
      write('Parameters for this run:');
      gotoxy(2, 6);
         write('Numbers of Cells:');
         for l := 1 to cell_steps do
            begin
            gotoxy(4 + l*8, 7);
            write(cell_data[l]:3);
            end;
      gotoxy(2, 10);
         write('Search Costs:    ');
         for l := 1 to search_steps do
            begin
            gotoxy(4 + l*8, 11);
            write(search_data[1][l]:2, '-',
                  search_data[2][l]:2);
            end;
      gotoxy(2, 14);
         write('Movement Costs:  ');
         for l := 1 to move_steps do
            begin
            gotoxy(4 + l*8, 15);
            write(move_data[1][l]:2, '-',
                  move_data[2][l]:2);
            end;
      gotoxy(2, 18);
         write('Problem');
      textcolor(14);
   end;

procedure update_screen(par_type : char; step : integer);
   var
      baseline : integer;
      max      : integer;
   begin
      case par_type of
         'c' : begin
               baseline := 7;
               max := cell_steps;
               end;
         's' : begin
               baseline := 11;
               max := search_steps;
```

```
                end;
          'm' : begin
                baseline := 15;
                max := move_steps;
                end;
       end; {case}
       if step > 1 then
          begin
          gotoxy((step-1)*8 + 6, baseline + 1);
          write('  ');
          end
       else
          begin
          gotoxy(max*8 + 6, baseline + 1);
          write('  ');
          end;
       gotoxy(step*8 + 6, baseline + 1);
          write('¦¦');
    end;

{==================== program main =====================}

begin {main}
   clrscr;
   set_defaults;
   write_parameters_to_screen;
   for hh := 1 to cell_steps do
      begin {h}
      update_screen('c', hh);
      parameters.number_of_cells := cell_data[hh];
      for ii := 1 to search_steps do
         begin {i}
         update_screen('s', ii);
         parameters.min_search_cost := search_data[1][ii];
         parameters.max_search_cost := search_data[2][ii];
         for problem := 1 to parameters.number_of_tests do
            begin
            generate_random_search_problem;
            for jj := 1 to move_steps do
               begin {j}
               update_screen('m',jj);
               parameters.min_movement_cost :=
                  move_data[1][jj];
               parameters.max_movement_cost :=
                  move_data[2][jj];
               generate_random_move_costs;
               inc(counter);
               initialize;
               gotoxy(11,18);
               write(problem, '  ');
               conduct_search;
               finalize;
```

```
              end;  {j}
           end;  {problem}
        end;  {i}
     end;  {h}
end.  { main }

{================== end of SEARCH_2.PAS ===================}
```

```
{  MENU.PAS

   This unit is the interactive menu used with SEARCH_1.PAS.

}

{==================== identification ====================}


unit menu;

{=====================} interface {=====================}

uses dos, crt;

type
   parameter_rec = record
      number_of_cells      : integer;
      min_movement_cost    : integer;
      max_movement_cost    : integer;
      min_search_cost      : integer;
      max_search_cost      : integer;
      number_of_random_policies : integer;
      number_of_tests      : integer;
      file_option          : integer;
      data_name            : string;
      pause_between_tests: boolean;
   end;


var
   start_time, start_hr                  : word;
   start_min, start_sec, start_100 : word;
   end_time, end_hr                      : word;
   end_min, end_sec, end_100             : word;
   elapsed_time, average_time            : word;
   parameters                            : parameter_rec;

procedure hold;
procedure pause;
procedure compute_elapsed_time;
procedure start_the_clock;
procedure stop_the_clock;
function power(base, exponent: real): real;
function min(first, second : integer): integer;
procedure search_menu;

{===================} implementation {===================}

procedure hold;       { pauses a program without a linefeed }
   var
      proceed : char;
```

```
   begin
      repeat until keypressed;
      proceed := readkey;
   end;

procedure pause; { pauses program and requests a key press }
   var
      proceed : char;

   begin
      writeln;
      writeln(' Press any key to continue... ');
      repeat until keypressed;
         proceed := readkey;
      writeln;
   end;

function power(base, exponent:real): real;
                                       { evaluates exponents }
    begin
       if base = 0 then pause;
       power := exp(exponent * ln(base));
    end;

function min(first, second : integer): integer;
                                  { gives min of two integers }
   begin
      if first < second then
         min := first
      else
         min := second;
   end;

procedure search_menu;
   var
      answer : boolean;

   procedure show_defaults;
                          { display current default settings }
      begin
        gotoxy(2, 2);
           write('Welcome to ');
           textcolor(14);
           write('Search!');
           textcolor(3);
           write(' -- the program that evaluates a search
                 heuristic.');
        gotoxy(2, 4);
           write('Following are the default parameters ',
                 'for the search problem:');
        gotoxy(2, 6);
           write('Number of different random ');
```

```
   textcolor(14);
   write('t');                    { key letter 't' }
   textcolor(3);
   write('ests to be run = ');
   textcolor(14);
   write(parameters.number_of_tests);
   textcolor(3);
gotoxy(2, 8);
   write('Number of ');
   textcolor(14);
   write('r');                    { key letter 'r' }
   textcolor(3);
   write('andom policies per test =        ');
   textcolor(14);
   write(parameters.number_of_random_policies);
   textcolor(3);
gotoxy(2, 10);
   write('Number of ');
   textcolor(14);
   write('c');                    { key letter 'c' }
   textcolor(3);
   write('ells (locations) in each test =   ');
   textcolor(14);
   write(parameters.number_of_cells);
   textcolor(3);
gotoxy(2, 12);
   textcolor(14);
   write('S');                    { key letter 's' }
   textcolor(3);
   write('earch cost per cell =                    ');
   textcolor(14);
   write(parameters.min_search_cost, ' - ',
         parameters.max_search_cost);
   textcolor(3);
gotoxy(2, 14);
   textcolor(14);
   write('M');                    { key letter 'm' }
   textcolor(3);
   write('ovement cost between cells =           ');
   textcolor(14);
   write(parameters.min_movement_cost, ' - ',
         parameters.max_movement_cost);
   textcolor(3);
gotoxy(2, 16);
   write('Save test data to ');
   textcolor(14);
   write('f');                    { key letter 'f' }
   textcolor(3);
   write('ile =                     ');
   textcolor(14);
   if parameters.file_option = 0 then
      write('no output file                 ');
```

```
        if parameters.file_option = 1 then
           write('overwrite ',
              parameters.data_name, '      ');
        if parameters.file_option = 2 then
           write('append ',
              parameters.data_name, '         ');
   end;  { show_defaults }

procedure change_parameters;
                            { update problem parameters }
   var
      done : boolean;
      response : char;
      new_data, test : integer;
      new_name : string;

   procedure update_number_of_tests;
      begin
         gotoxy(2, 20);
            textcolor(3);
            write(' Enter new number of tests. ');
         gotoxy(47, 6);
            write('     ');
         gotoxy(47, 6);
            textcolor(4);
            readln(new_data);
         if (new_data > 0) and (new_data < 10001) then
            parameters.number_of_tests := new_data
         else
            begin
              gotoxy(2, 20);
                 textcolor(4);
                 write('Input data out of range
                       (1 - 10000). ');
              gotoxy(2, 21);
                 textcolor(3);
                 pause;
              gotoxy(2, 22);
                 write('                               ');
            end;
         gotoxy(47, 6);
            textcolor(14);
            write(parameters.number_of_tests, '       ');
      end;  { update_number_of_tests }

   procedure update_number_of_random_policies;
      begin
         gotoxy(2, 20);
            textcolor(3);
            write(' Enter new number of random
                  policies ');
         gotoxy(47, 8);
```

```
            write('    ');
         gotoxy(47, 8);
            textcolor(4);
            readln(new_data);
            if (new_data > 0) and (new_data < 1001) then
            parameters.number_of_random_policies :=
               new_data
         else
            begin
              gotoxy(2, 20);
                 textcolor(4);
                 write('Input data out of range
                       (1 - 1000).                        ');
              gotoxy(2, 21);
                 textcolor(3);
                 pause;
              gotoxy(2, 22);
                 write('                                  ');
            end;
         gotoxy(47, 8);
            textcolor(14);
            write(parameters.number_of_random_policies,
                  '     ');
      end;  { update_number_of_policies }

   procedure update_number_of_cells;
      begin
         gotoxy(2, 20);
            textcolor(3);
            write(' Enter new number of cells. ');
         gotoxy(47, 10);
            write('    ');
         gotoxy(47, 10);
            textcolor(4);
            readln(new_data);
         if (new_data > 0) and (new_data < 101) then
            parameters.number_of_cells := new_data
         else
            begin
              gotoxy(2, 20);
                 textcolor(4);
                 write('Input data out of range
                       (1 - 100). ');
              gotoxy(2, 21);
                 textcolor(3);
                 pause;
              gotoxy(2, 22);
                 write('                                   ');
            end;
         gotoxy(47, 10);
            textcolor(14);
            write(parameters.number_of_cells, '        ');
```

```
   end;  { update_number_of_cells }

procedure update_max_search_cost;
   begin
      gotoxy(2, 20);
         textcolor(3);
         write(' Enter new minimum search cost.    ');
      gotoxy(47, 12);
         write('    ');
      gotoxy(47, 12);
         textcolor(4);
         readln(new_data);
      if (new_data > 0) and (new_data < 101) then
         parameters.min_search_cost := new_data
      else
         begin
           gotoxy(2, 20);
              textcolor(4);
              write('Input data out of range
                    (1 - 100).                      ');
           gotoxy(2, 21);
              textcolor(3);
              pause;
           gotoxy(2, 22);
              write('                               ');
         end;
      gotoxy(47, 12);
         textcolor(14);
         write(parameters.min_search_cost, '    ');
      gotoxy(2, 20);
         textcolor(3);
        write(' Enter new maximum search cost.     ');
      gotoxy(52, 12);
         write('    ');
      gotoxy(52, 12);
         textcolor(4);
         readln(new_data);
      if (new_data >= parameters.min_search_cost) and
         (new_data < 101) then
         parameters.max_search_cost := new_data
      else
         begin
           parameters.max_search_cost :=
              parameters.min_search_cost;
           gotoxy(2, 20);
              textcolor(4);
              write('Input data out of range (',
                parameters.min_search_cost,
                ' - 100).');
           gotoxy(2, 21);
              textcolor(3);
              pause;
```

```
            gotoxy(2, 22);
               write('                              ');
         end;
      gotoxy(47, 12);
         textcolor(14);
         write(parameters.min_search_cost, ' - ',
               parameters.max_search_cost, '  ');
   end;  { update_max_search_cost }

procedure update_max_movement_cost;
   begin
      gotoxy(2, 20);
         textcolor(3);
         write(' Enter new minimum movement cost.  ');
      gotoxy(47, 14);
         write('   ');
      gotoxy(47, 14);
         textcolor(4);
         readln(new_data);
      if (new_data >= 0) and (new_data < 101) then
         parameters.min_movement_cost := new_data
      else
         begin
            gotoxy(2, 20);
               textcolor(4);
               write('Input data out of range
                     (0 - 100).                 ');
            gotoxy(2, 21);
               textcolor(3);
               pause;
            gotoxy(2, 22);
               write('                              ');
         end;
      gotoxy(47, 14);
         textcolor(14);
         write(parameters.min_movement_cost);
      gotoxy(2, 20);
         textcolor(3);
         write(' Enter new maximum movement cost.  ');
      gotoxy(51, 14);
         write('    ');
      gotoxy(51, 14);
         textcolor(4);
         readln(new_data);
      if (new_data >= parameters.min_movement_cost)
            and (new_data < 101) then
         parameters.max_movement_cost := new_data
      else
         begin
         parameters.max_movement_cost :=
            parameters.min_movement_cost;
         gotoxy(2, 20);
```

```
            textcolor(4);
            write('Input data out of range (',
                  parameters.min_movement_cost,
                  ' - 100).');
         gotoxy(2, 21);
            textcolor(3);
            pause;
         gotoxy(2, 22);
            write('                                            ');
         end;
      gotoxy(47, 14);
         textcolor(14);
         write(parameters.min_movement_cost, ' - ',
               parameters.max_movement_cost, '  ');
   end;  { update_max_movement_cost }

procedure update_file_options;
   var
      file_exists : integer;
              { used to see if a file already exists }
      temp_option : integer;
                    { holds the original file option }
      finished     : boolean;
                   { equals true when done with menu }
      try_data     : text;
                 { used in looking for existing file }
      temp_name    : string;
                   { holds original output file name }
      cells        : string[3];
                            { used in constructing.. }
      move         : string[3];
                            { ..default file name    }
      search       : string[3];

   begin
      temp_name := parameters.data_name;
      temp_option := parameters.file_option;
      gotoxy(2, 20);
         textcolor(3);
         write('Enter new name (without ext) or
               type ');
         textcolor(14);
         write('N');
         textcolor(3);
         write(' for no file or ');
         textcolor(14);
         write('D');
         textcolor(3);
         write(' for default name.');
      gotoxy(47, 16);
         write('                           ');
      gotoxy(47, 16);
```

```
   textcolor(4);
   readln(new_name);
if (length(new_name) >= 1)
      and (length(new_name) <= 8) then
   if (new_name = 'N') or (new_name = 'n') then
      begin
         parameters.file_option := 0;
         parameters.data_name :=
            'no output file';
      end
   else
      begin  {0}
      if (new_name = 'D')
         or (new_name = 'd') then
         begin
         parameters.file_option := 4;
         str(parameters.number_of_cells, cells);
         str(parameters.max_movement_cost,
             move);
         str(parameters.max_search_cost,
             search);
         parameters.data_name := cells +
             '_' + search + '_' + move +'.TXT';
         end
      else
         parameters.data_name :=
             new_name + '.TXT';
      {$I-}          { turns off input checking }
      assign(try_data, parameters.data_name);
      reset(try_data);   { tries to open file }
      {$I+}           { turns on input checking }
      file_exists := IORESULT;{ file opened?? }
      if file_exists = 0 then
                     { if  = 0 then file exists }
         begin   { 1 }
         gotoxy(2, 20);
            textcolor(4);
            write('File exists.');
            textcolor(14);
            write('   A');
            textcolor(3);
            write('ppend current file, ');
            textcolor(14);
            write('o');
            textcolor(3);
            write('verwrite file, or ');
            textcolor(14);
            write('r');
            textcolor(3);
            write('eturn without change?     ');
         finished := false;
         while not finished do
```

```
                    begin   { 2 }
                    gotoxy(55, 20);
                    response := readkey;
                    case response of
                      'a', 'A' : begin
                           parameters.file_option := 2;
                           finished := true
                           end;
                      'o', 'O' : begin
                           parameters.file_option := 3;
                           finished := true
                           end;
                        'r', 'R' : begin
                           parameters.data_name :=
                                     temp_name;
                           parameters.file_option :=
                                     temp_option;
                           finished := true;
                                   end;
                    end;   { case }
                    end   { begin 2 }
                 end  { begin 1 }
              else
                    parameters.file_option := 1
              end   { begin 0 }
        else
           begin
              gotoxy(2, 20);
                 textcolor(4);
                 write('Incorrect file name
                      (1 - 8 characters).',        ');
              gotoxy(2, 21);
                 textcolor(3);
                 pause;
              gotoxy(2, 22);
                 write('                               ');
        end;  { also endif }
        gotoxy(47, 16);
           textcolor(14);
           case parameters.file_option of
              0, 1, 4 : write(parameters.data_name,' ');
              2 : write('append ', parameters.data_name,
                        '           ');
              3 : write('overwrite ',
                  parameters.data_name, '        ');
           end;
     end;  { update_file_options }

  procedure ask_to_quit;
     begin
        gotoxy(2, 22);
        write('Quit! Are you sure? (y/n) ');
```

```
      response := readkey;
      if upcase(response) = 'Y' then
         begin
         clrscr;
         halt;
         end;
   end;

procedure list_options;
   begin
   gotoxy(2, 20);
      textcolor(3);
      write('Press the ');
      textcolor(14);
      write('highlited');
      textcolor(3);
      write(' letter to change a value or ');
      textcolor(14);
      write('B');
      textcolor(3);
      writeln(' to begin searching.                  ');
      write(' Press ');
      textcolor(14);
      write('Q ');
      textcolor(3);
      writeln('to quit.                   ');
      if parameters.pause_between_tests = false then
         begin
         write(' Press ');
         textcolor(14);
         write('P ');
         textcolor(3);
         write('to pause between tests.    ');
         end
      else
         begin
         write(' Press ');
         textcolor(14);
         write('P ');
         textcolor(3);
         write('to not pause between tests.');
         end;
   end;  { list options }


begin  { change_parameters }
   done := false;
   while not done do
      begin;
      list_options;
      response := readkey;
```

```
            case response of
               'b', 'B' : done := true;
               't', 'T' : update_number_of_tests;
               'r', 'R' : update_number_of_random_policies;
               'c', 'C' : update_number_of_cells;
               's', 'S' : update_max_search_cost;
               'm', 'M' : update_max_movement_cost;
               'f', 'F' : update_file_options;
               'q', 'Q' : ask_to_quit;
               'p', 'P' : parameters.pause_between_tests :=
                             not(parameters.pause_between_tests);
            end;  { case }
         end;
      end;  { change_paramaters }

   begin  { search_menu }
      clrscr;
      textcolor(3);
      show_defaults;
      change_parameters;
      clrscr;
   end;  { search_menu }


begin
end.

{=================== end of menu.pas =====================}
```

A HEURISTIC FOR DISCRETE SEARCH PROBLEMS WITH POSITIVE SWITCH COSTS

by

STEPHEN ROBERT RIESE

B.Arch., University of Notre Dame, 1982

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Industrial Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1992

ABSTRACT

An object is hidden in one of n cells according to a known probability distribution $p_1,...p_n$. A search policy s is a sequence of cells to be visited and searched in an attempt to find the target. The probability of overlooking the target if we search cell i and if the target is in cell i is a_i. The cost of searching cell i is c_i. The cost of moving, or switching, from cell j to cell i is m_{ji}. An optimal policy, s^*, is one for which the expected cost of finding the target is a minimum. When all $m_{ji} = 0$ the problem has a well known solution. The problem with positive m_{ji} is NP-hard and there is not an easy solution. We provide a heuristic to construct a search policy, s', which is good, but is not guaranteed to be optimal. The heuristic is an extension of the optimal solution for the problem with zero switching costs.