SLAC-104
UC-32
(MISC)

BI-DIRECTIONAL AND HEURISTIC SEARCH IN PATH PROBLEMS

IRA POHL

STANFORD LINEAR ACCELERATOR CENTER
Stanford University
Stanford, California 94305

PREPARED FOR THE U.S. ATOMIC ENERGY
COMMISSION UNDER CONTRACT NO. AT(04-3)-515

May 1969

ABSTRACT

Path finding is a key process in many areas of computation. Optimization problems and heuristic search problems are two notable examples. The first part of this dissertation presents a class of algorithms, denoted VGA, for solving the two point shortest path problem in directed graphs with non-negative edge weights. This class is a bi-directional extension of the most efficient known uni-directional shortest path algorithms. While it has long been realized that bi-directional algorithms often provide computational savings, a theory of this has not been forthcoming until now. This theory shows how a bi-directional method using the proposed cardinality comparison strategy is a priori the best shortest path algorithm within the class of algorithms VGA.

These theoretical results are verified by extensive tests of VGA. A computer program was written where several standard uni-directional and bi-directional strategies were compared with cardinality comparison. The program randomly generated a number of large directed graphs and each strategy in turn was tried on numerous path problems within these graphs.

In heuristic search for artificial intelligence problems, algorithms similar to the two point shortest path problem are used. The spaces searched are enormous, often infinite, and in consequence the constraint on finding the shortest path is abandoned. The concern is for finding any solution path with minimum effort. The second part of the dissertation presents a theory of these problems and some experiments with the fifteen puzzle in using the methods suggested by this theory of heuristic search.

The evaluation function directing the search is the sum of the distance from the starting node and an estimate of the distance to the goal. This second component is the heuristic term, and if accurate, allows efficient path finding in

large spaces. Some results on the effect of error in the heuristic term are presented. Especially interesting is theorem 7.9, showing that the distance from the start should be incorporated in the evaluation function. This particular result runs counter to the reliance strictly on the heuristic term, a practice which is widespread.

Bi-directional heuristic search is also proposed. VGHA, a bi-directional class of algorithms, is an extension of the Hart, Nilsson, and Raphael uni-directional heuristic search algorithms. Their results are extended to this more general class.

These methods are used in solving fifteen puzzle problems and comparing the number of nodes explored. It is a continuance of the empirical work started by Doran and Michie with the Graph Traverser. The most interesting results show the importance of appropriately weighting the heuristic term in the evaluation function. For example, overrelaxation seems to be an important principle, which means weighting the heuristic term on an average slightly more than the cost-to-date term. This data is a successful prediction from the theory.

Some further results include the extension of bi-directional methods to the network flow problem; the description of a new efficient algorithm for finding bridges in directed graphs, which are structurally interesting as they can be used for finding partitions; and applications of hashing techniques to remove problems of intersection in bi-directional search.

PREFACE

The range of topics and ideas in computer science is so extensive that it is often criticized as an amorphous mass rather than a discipline. Where is the thread that runs from numerical analysis to systems programming and on to artificial intelligence? As a student of this wilderness several points of unity impressed me. The aim of all people in the discipline is to solve problems algorithmically, and the tool used inherently discretizes the problems to be solved. Graph Theory is one "language" that provides a description for many areas of computer science. It is therefore important to be able to manipulate these structures computationally. One of the basic problems in this representation is finding paths between two nodes. It is a core problem in such disparate fields as operations research, circuit theory, and artificial intelligence, hence the importance of efficient algorithms for this problem. This then is the chief concern of our work.

0.1 Organization

This dissertation is divided into two major parts. First, Chapters 1 through 5 are concerned with the classical two node shortest path problem. Secondly, Chapters 6 through 9 are concerned with heuristic search. The connection between the two problems is that our model of heuristic search is a path problem in a directed graph. In each case we are interested in a computationally efficient solution. The insights from the better understood shortest path problem provide the tools for formalizing and solving the heuristic path problem.

Chapter 1 is an introduction to the shortest path problem, exhibiting some of the principal methods to solve the problem. Chapter 2 contains our general bidirectional shortest path algorithm which subsumes the current best algorithms as special cases. This allows us to prove that this class of algorithms is correct and to ask which is the best member of this class. We then propose the cardinality

comparison algorithm and in Chapter 3 discuss a model of efficiency for this class of algorithms. Chapter 4 proves that a priori expected work performed by cardinality comparison is a minimum for our class of algorithms. Finally in Chapter 5 we show some results of extensive experimental tests confirming the theoretical conclusions of the previous two chapters.

Chapter 6 introduces a directed graph model of artificial intelligence problem as path problems. It discusses how the search for a solution path is expedited by appropriate use of heuristic functions. Chapter 7 presents a theory of unidirectional heuristic search. It presents a formal characterization of the effect of error in the heuristic function on search efficiency. Chapter 8 presents some results of using the ideas developed in formally characterizing heuristic search in solving fifteen puzzle problems in the mode of Doran and Michie.[18] Chapter 9 is the extension of the Hart, Nilsson and Raphael Theory[30] to bi-directional heuristic search. This unifies some of the work on the shortest path problem with the heuristic search problem, as the solution paths obtained must be shortest. Chapter 10 presents some further observations on computational graph theory and its use as a model of artificial intelligence. It notes some unsolved problems and presents conclusions on the problem area and our particular approach.

0.2 Contributions

The contributions to the field have been:

a. A formulation and proof of correctness of a class of algorithms for efficiently solving the two node shortest path problem.

b. A theory of efficiency in solving these problems (shortest path space).

c. The discovery and proof of the cardinality comparison strategy as the most efficient a priori bi-directional shortest path

method. This is demonstrated by theorem 4.5, which is an interesting probabilistic result on this type of decision problem.

d. Empirical verification of this theory and the gain in efficiency by cardinality comparison over other standard methods.

e. A theory of efficient heuristic search and associated worst case analysis of heuristic functions.

f. Extensions of the Hart, Nilsson and Raphael results to bi-directional heuristic search.

g. The use of associative search (hashing techniques) to solve the redundancy problem and the tree intersection problem.

h. An efficient bridge ("narrows") finder in graph spaces.

We feel that these explicit contributions are the result of a computational approach to these problem areas. One keeps in mind algorithmic efficiency without failing to justify rigorously the method. Theory and practice naturally develop in step, each bringing insight to the other.

ACKNOWLEDGMENTS

I am profoundly grateful to my thesis advisor, Professor William F. Miller, for his constant encouragement and guidance throughout the course of this research. I wish to thank Professors David Gries and Nils Nilsson for their many astute remarks and their reading of the dissertation. In addition many friends and colleagues provided stimulating discussion and encouragement. In this regard Susan Graham, Alan Shaw, Charles Zahn and James George were particularly helpful.

During my work, I have also had the cooperation and assistance of Linda Lorenzetti, Kathleen Maddern and Carla West.

TABLE OF CONTENTS

Chapter		Page
1.	The Shortest Path Problem	1
	1.1 Introduction	1
	1.2 Problem Statement	2
	1.3 E. W. Dijkstra's Method	4
	1.4 Dantzig's Bi-directional Method	8
	1.5 Nicholson's Bi-directional Method	12
	1.6 Stopping Conditions	13
2.	Generalization of the Shortest Path Algorithm	15
	2.1 Intuitive Description	15
	2.2 Formal Description of the Very General Algorithm	17
	2.3 Proof of the Correctness of VGA	19
	2.4 Extensions of VGA	24
3.	Shortest Path Space	26
	3.1 Graph Density	27
	3.2 A Locally Optimal Decision Rule	31
4.	On the Optimality of Our Decision Strategy	33
	4.1 Probabilistic Analysis	33
	4.2 Pathological Possibilities	41
5.	Empirical Results	44
	5.1 Data	44
	5.2 Evaluation	44
	5.3 Results	45
6.	Heuristic Search as a Path Problem	50
	6.1 Introduction	50
	6.2 Problem Spaces and Heuristic Search	52

Chapter		Page
7.	Theory of Uni-directional Heuristic Search	60
	7.1 Some Theorems on Searching	60
	7.2 Heuristic Error	62
	7.3 How Error Affects Heuristic Search	69
8.	Some Uni-directional Experiments with the Fifteen Puzzle	78
	8.1 Heuristic Functions	79
	8.2 Data	81
	8.3 Experiment	81
	8.4 Results	83
	8.5 Remarks	90
9.	Bi-directional Heuristic Search	91
	9.1 Extension to Bi-directional Heuristic Search	92
	9.2 Correct Extension — The Very General Heuristic Algorithm.	94
	9.3 Correctness of VGHA	98
	9.4 Strategies in Bi-directional Search	100
	9.5 Associative Search as a Solution to Redundancy and Intersection	104
10.	Further Observations, Open Problems, and Conclusions	107
	10.1 Network Flow Algorithm	107
	10.2 Bi-directional Intersection	108
	10.3 Learning	111
	10.4 Structural Features — Bridges	111
	10.5 Some Concluding Remarks	115
Appendix I	ALGOL W Implementation of VGA	116
Appendix II	Comparative Results Using Different Strategies in VGA	126

Chapter Page

Appendix III ALGOL W Implementation of VGHA for the Fifteen Puzzle 138

References 152

LIST OF TABLES

		Page
5.1	Distribution functions, Example A: 200 node graph	46
5.2	Cumulative results on 500 node graphs	49
7.1	Commonly used evaluators	60
8.1	Results for $f_1 = g + \omega \cdot P$	84
8.2	Results for $f_2 = g + \omega(P + 20 \cdot R)$	85
8.3	Results for $f_3 = g + \omega \cdot S$	86
8.4	Results for $f_4 = S + \omega \cdot (S + 20 \cdot R)$	87
8.5	Initial values of the heuristic functions and the shortest solution paths found	88
8.6	Density vs. path length for function f_4, problem A5	90
II.1	Graph size 500, average degree 3	127
II.2	Graph size 500, average degree 6	128
II.3	Graph size 500, average degree 9	129
II.4	Graph size 500, average degree 12	130
II.5	Graph size 500, average degree 15	131
II.6	Forward crosstable	136
II.7	Pohl crosstable	137

LIST OF FIGURES

		Page
1.1	Example: Nicholson's graph	6
1.2	Nicholson's graph after "conceptual replacement" of edges	11
2.1	A shortest path from s to t of length 21 is shown in the above graph. Each edge is undirected and may be thought of as representing two directed edges pointing in opposite directions . .	16
3.1	Mapping Nicholson's graph into shortest path space	29
3.2	Graph with edge lengths all unity	31
4.1	A pathological example	42
6.1	15 puzzle	53
6.2	Some puzzle configurations and descriptions	55
6.3	An example of a puzzle solution. How HPA with h=P and $\omega = \infty$ solves a particular 15 puzzle	57
7.1	Regular infinite trees	64
7.2	Nodes numbered in order visited by a depth first search to level 3	68
7.3	The goal node is marked by an x. Other nodes are labeled by order of search (inside) and f value outside. Five nodes are searched when x is found	70
7.4	Comparison between two possible ω's for h*: (a) $\omega=\infty$ (b) $\omega=1$	72
8.1	Tile 5 is easier to move in (i) because it is next to the blank	80
8.2	Initial positions and code numbers used in experiments	82
8.3	Performance of each evaluator with respect to ω	89
9.1	Euclidean counterexample	93

		Page
9.2	All shortest paths are subpaths of a Hamilton circuit	102
10.1	Bi-directional search	108
10.2	Intermediate board conjecture	110
10.3	Edge (c,d) is a bridge	112
II.1	Results — 500 node graphs	134
II.2	Results — 150 node graphs	135

CHAPTER 1

THE SHORTEST PATH PROBLEM

1.1 Introduction

The problem of finding the shortest path between two points pervades many fields of science. It is of fundamental importance to operations research,[7,8,11,32] and has wide application in computer science,[18,30,61] especially in modeling various artificial intelligence problems involving searches of large but effectively defined spaces. The discrete nature of the problem and its simplicity argue for an elegant computer algorithm. Over the years, many computer scientists,[17,23] operations researchers,[6,7,15] and applied scientists[40,63] have attempted to solve the problem in an efficient manner. While one school, the operations researchers, have characterized the problem in linear programming terms[15] and dynamic programming terms,[6] the pragmatic computer scientists unencumbered by tradition, have attempted to intuit a naive but efficient computational method and have been remarkably successful.[17,20,23] Historically the work derives its impetus from this computer science tradition and this work is an attempt to generalize this approach and generate a theory of efficient solutions to the shortest path problem and analogous discrete algorithms which benefit from these ideas.

Shortest path work as noted above covers many disciplines and the literature is widespread and difficult to survey. Often a particular method or discovery is attributed to several authors regardless of temporal primacy because of the different disciplines using these results. Dreyfus[20] has gone to great lengths to appropriately credit the originators of various of the shortest path algorithms. In this regard my work owes its allegience to Dijkstra's method which in operations research is attributed to Dantzig. I would emphasize that Dantzig's work

in both a computer science vein and an operations research vein has been important. Nevertheless, my work was initiated and pursued in computer science terms. This may be characterized as an eclectic pragmatic approach with efficiency the paramount goal.

In generalizing the Dijkstra approach, an attempt is made to provide a unified theory for efficiency in this class of algorithms. This unification has led to a more efficient cardinality comparison strategy and extensions of this approach to heuristic search.[30,57]

1.2 Problem Statement

Consider a directed graph G(X, U) where X is a collection of nodes and U is a collection of edges. Each member of U may be considered an ordered pair of nodes of X. Let the nodes of X be mapped into the integers in increasing order starting from 1; then some member of $\alpha \in U$ may be written as e_{ij} where i is the initial node of α and j is the terminal node of α. A person standing on the ith node of G could walk along street α and reach the jth node of G, where α is a one-way street.

The cardinality of a set will be denoted by "| |". The cardinality of G will be the cardinality of its vertex* set X.

A path μ from node s to node t is a sequence of edges

$$\mu = \left(e_{X_0X_1}, e_{X_1X_2}, \ldots, e_{X_{K-1}X_K}\right)$$

where $X_0 = s$ and $X_K = t$. Alternatively we write

$$\mu = (X_0, X_1, \ldots, X_K) ,$$

or

$$\mu = (U_1, U_2, \ldots, U_K) \text{ where } U_i = e_{X_{i-1}X_i} .$$

*Node and vertex will be used interchangeably as will arc and edge.

The graphs we will consider have their edge set mapped into the non-negative reals. These values will be called the lengths of the edges and will be represented as

$$\ell(e_{ij}) \geq 0 \quad .$$

The length of a path μ will also be written as $\ell(\mu)$ where

$$\mu = (U_1, \ldots, U_K) \text{ and } \ell(\mu) = \sum_i \ell(U_i) \quad .$$

The shortest path problem considered in this paper is:

given $s, t \in X$ find some μ^*, a path from s to t such that

$\ell(\mu^*)$ is a minimum over all paths from s to t.

There are many important variants of this problem.[20] One such is the specialization of length to take on only a value of unity. Then the shortest path between two nodes is the one which traverses the fewest edges. This is the cardinality or Manhattan distance of a path. This problem can be solved by the methods described here; some other variants like the kth-shortest path cannot. However, the two node problem is the basic shortest path problem with many areas of applicability.

The problem is clearly solvable, a most primitive solution being a search over all possible paths.[24,26] An improvement over this exhaustive enumeration is to recognize that if $\mu = (s, x_1, x_2, \ldots, x_k, t)$ is optimal then all its subpaths are optimal. This satisfies the fundamental dynamic programming principle and therefore one can use Bellman's method (see Ref. 6, p. 230).

1. $g_i^{(0)} = \ell\left(e_{x_i t}\right)$, $i = 1, 2, \ldots n$, $\qquad g_t^{(0)} = 0$

2. $g_i^{(k)} = \min_{j \neq i} \left[\ell\left(e_{x_i x_j}\right) + g_j^{(k-1)}\right]$, $\qquad i = 1, 2, \ldots n$, $\; g_t^{(k)} = 0$

Step 2 is iterated up to n-1 times (stop when two successive iterations are the same) and gives the shortest paths from all nodes to t. This requires $0(n^3)$ (order n-cubed) operations — a manageable amount of work. Nevertheless, better methods of $0(n^2)$ have subsequently been developed. These are acknowledged as the current best methods[20] and are unlikely to be superceded by better methods on ordinary Von Neumann machines. The $0(n^2)$ methods constitute the precursors of this work.

In order to place this work in perspective, we must describe the antecedents of the general algorithm and theory. This will give the reader insight into the development of this work as a generalization of shortest path algorithms. In doing this, we will use an example of Nicholson's[44] to demonstrate the mechanics of each method.

1.3 E. W. Dijkstra's Method[17]

This algorithm was independently discovered by G. Dantzig[15] and others.[20, 6] Dijkstra defines three sets:

A The nodes having their minimum path from s (the initial or starting node) known.

B The direct successors of the above set which are not in it.

C The remaining nodes.

The computation proceeds in two stages.

1. A node in set B with current minimum distance to s is transferred to set A. If this node is t (terminal node) the computation halts.
2. The successors of the node just placed in A by step 1 are calculated. Of these, the nodes that are in C are transferred to B. The value of the distance from s, of the

nodes already in B, are changed if the new distance calculated is smaller than their current value.

So, with each node, let us associate its d value and its wf value.

d(n) = current best distance from s

wf(n) = immediate predecessor of n along path from s, for which d(n) was calculated.

Initially all nodes have

$$d(x_i) = \infty \quad \text{and} \quad wf(x_i) = \text{undefined}.$$

We always begin by placing s in set A with d(s) = 0.

Let us look at this method applied to Nicholson's graph and find the shortest path from node 1 to node 9 (see Fig. 1.1).

Step 1

Set A	$d(x_i)$	$wf(x_i)$	Set B	$d(x_i)$	$wf(x_i)$	Set C	$d(x_i)$	$wf(x_i)$
1	0	--	2	3	1	5	∞	--
			3	6	1	6	∞	--
			4	7	1	7	∞	--
						8	∞	--
						9	∞	--

We will no longer show set C, since it can be found from complementing A∪B.

Step 2 node 2 has minimum distance in B.

Set A	$d(x_i)$	$wf(x_i)$	Set B	$d(x_i)$	$wf(x_i)$
1	0	--	3	4	2
2	3	1	4	7	1
			5	7	2

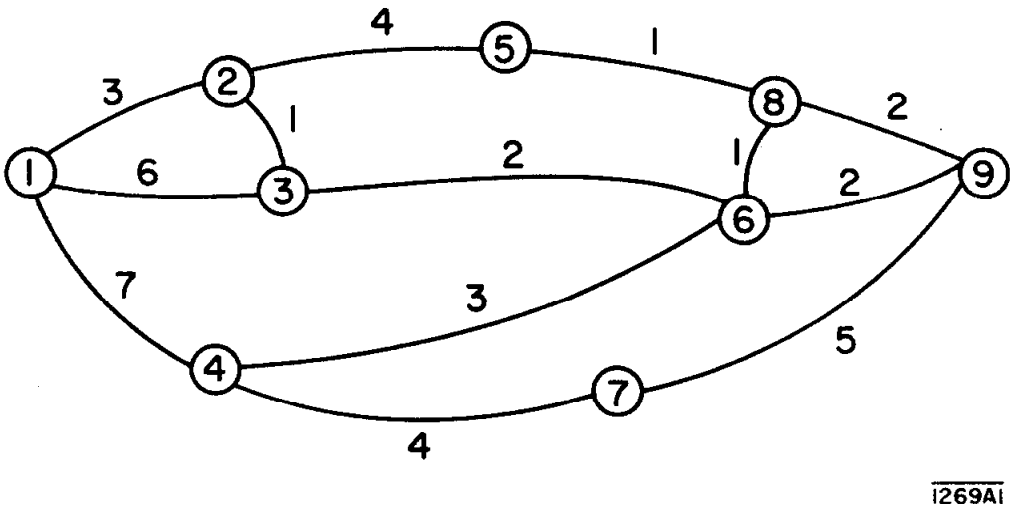


FIG. 1.1--Example: Nicholson's graph.

Step 3

Set A	$d(x_i)$	$wf(x_i)$	Set B	$d(x_i)$	$wf(x_i)$
1	0	--	4	7	1
2	3	1	5	7	2
3	4	2	6	6	3

Step 4

Set A	$d(x_i)$	$wf(x_i)$	Set B	$d(x_i)$	$wf(x_i)$
1	0	--	4	7	1
2	3	1	5	7	1
3	4	2	8	7	6
6	6	3	9	8	6

Step 5

Set A	$d(x_i)$	$wf(x_i)$	Set B	$d(x_i)$	$wf(x_i)$
1	0	--			
2	3	1	5	7	2
3	4	2	7	11	4
4	7	1	8	7	6
6	6	3	9	8	6

Steps 6 and 7

Set A	$d(x_i)$	$wf(x_i)$	Set B	$d(x_i)$	$wf(x_i)$
1	0	--	7	11	4
2	3	1	9	8	6
3	4	2			
4	7	1			
5	7	2			
6	6	3			
8	7	6			

Step 8

Set A	$d(x_i)$	$wf(x_i)$	Set B	$d(x_i)$	$wf(x_i)$
1	0	--	7	11	4
2	3	1			
3	4	2			
4	7	1			
5	7	2			
6	6	3			
8	7	6			
9	8	6			

The computation halts with $d(9) = 8$. By tracing back through wf we have

$$9, wf(9), wf(wf(9)) \ldots, 1$$

or that the shortest path is

$$(1, 2, 3, 6, 9) \ .$$

Dantzig in proposing a similar algorithm, recommended that the edge lists be ordered by length. This makes for fewer comparisons and additions when augmenting sets A and B. However, as Dreyfus[20] points out, any savings are outweighed by the computation required to order the edges.

1.4 Dantzig's Bi-directional Method

The earliest widely published mention of a bi-directional algorithm occurs in Dantzig.[15] The description of this algorithm is ambiguous and vague, leading to subsequent misinterpretation by Dreyfus.[20] In fact, a computer scientist would label the description as violating the principle of effectiveness.[33] For this reason much of the credit for a correct bi-directional algorithm has accrued to Nicholson. However, when asked by G. Dantzig[16] to investigate this issue, I discovered an interpretation that leads to a correct algorithm.

Dantzig's[15] description was as follows: (p. 365)

> "If the problem is to determine the shortest path from a given origin to a given terminal, the number of comparisons can often be reduced in practice by fanning out from both the origin and the terminal, as if they were two separate independent problems.
>
> "However, once the shortest path between a node and the origin or the terminal is found in one problem, the path is conceptually replaced by a single arc in the other problem. The algorithm terminates whenever the fan of one of the problems reaches its terminal in the other."

In terminology analogous to Dijkstra's, the algorithm can be described as alternating between sets A, B, and C in a forward manner, and sets D, E, and F in a backward manner. Where

D The set of nodes having their minimum path to t known.

E The direct predecessors of set D, which are not in D.

F The remaining nodes.

In the Dijkstra algorithm, the set A starts initially with node s, and with each iteration grows until it includes t. The nodes $x_i \in A$ form a rooted tree with s as the root, where

$$T = \{(X, E)\}$$

$$X = \{x_i: x_i \in A\}$$

$$E = \{(s, x_i): x_i \in A\}$$

with

$$\ell(s, x_i) = d(x_i)$$

In Dantzig's algorithm a similar tree exists for set D which is rooted from node t. We interpret "conceptually replaced" as modifying the original graph by rooted trees grown in the fashion described above. In this algorithm the sets A and D are expanded alternately; the first of these sets to include both s and t contains the shortest path.

"conceptually replaced":

Let some iteration place n in A, then all edges connecting n with any $x_i \in A$ are deleted from the graph and replaced by edge (s,n) with length d(n). Correspondingly, if n is placed in set D the edge (n,t) is included in the graph.

Again we use Nicholson's example, where in addition to quantities in Dijkstra's algorithm we have:

$d_t(n)$ = current best distance to t

wt(n) = immediate successor of n along path to t, for which $d_t(n)$ was calculated.

Initially all nodes have $d(x_i) = d_t(x_i) = \infty$ and $wf(x_i) = wt(x_i)$ = undefined. We begin by placing s in set A with d(s) = 0 and t in set D with $d_t(t) = 0$.

Step 1 We expand set A

A	$d(x_i)$	$wf(x_i)$	B	$d(x_i)$	$wf(x_i)$
1	0	--	3	4	2
2	3	1	4	7	1
			5	7	2

Step 2 We expand set D

D	$d_t(x_i)$	$wt(x_i)$	E	$d_t(x_i)$	$wt(x_i)$
6	2	9	3	4	6
9	0	--	4	5	6
			7	5	9
			8	2	9

Step 3 We expand set A

A	$d(x_i)$	$wf(x_i)$	B	$d(x_i)$	$wf(x_i)$
1	0	--	6	2	9
2	3	1	8	2	9
3	4	2	9	0	--

Step 4 We expand set D

D	$d_t(x_i)$	$wt(x_i)$	E	$d_t(x_i)$	$wt(x_i)$
6	2	9	3	4	6
8	2	9	4	5	6
9	0	--	5	3	8
			7	5	9

At the end of step 4 the revised graph looks like Fig. 1.2. Note that (2, 3) and (6, 8) are no longer of interest and have been

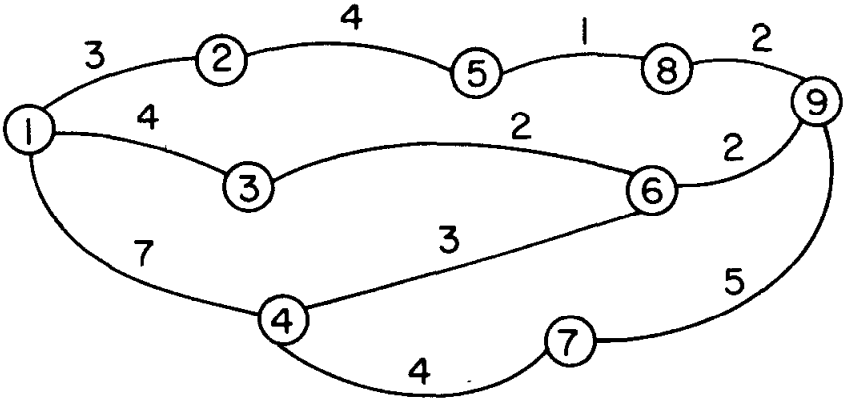


FIG. 1.2--Nicholson's graph after "conceptual replacement" of edges.

"conceptually replaced." The above alternating expansion of sets A and D continues until step 11.

Step 11 We expand set A

A	$d(x_i)$	$wf(x_i)$	B	$d(x_i)$	$wf(x_i)$
1	0	--	7	11	4
2	3	1			
3	4	2			
4	7	1			
5	7	2			
6	6	3			
9	8	6			

The algorithm halts with node 9 being placed in set A. Only six expansions of set A were required, as opposed to seven expansions with the Dijkstra algorithm. Node 8 was not included in set A because of conceptual replacement of its edges. However, in this example more work was done by the Dantzig bi-direction algorithm than by the Dijkstra algorithm. The Dantzig method is to use two separate shortest path methods where the savings are made as each reduces the complexity of its end of the graph. However, as will be seen later, the termination condition is needlessly bad, and several refinements will be discussed below. In light of the above elaboration of Dantzig's method, he must be credited as an early (earliest?) correct innovator in bi-directional methods.

1.5 Nicholson's Bi-directional Method

Nicholson's algorithm[44] differs from Dantzig's in two important ways. Firstly, Nicholson's is not strictly alternating between forward and backward sets. Secondly, his termination condition is more complex, but allows the algorithm to terminate much sooner, in general, than Dantzig's.

Once again we have the sets A, B, C, D, E, and F as described above. However, instead of alternating, the set expanded depends on

$$x_i \text{ such that } x_i \in B \cup E \text{ and}$$

$$d(x_i) \quad \text{or} \quad d_t(x_i) \text{ is a minimum.}$$

As long as nodes are closer to s than to t, set B is used and the algorithm augments the forward set A, otherwise set E is used and set D is augmented. All nodes tied at the minimum distance are simultaneously expanded, and no conceptual replacement occurs. The algorithm terminates when

$$\min_{x_i \in A \cap D} d(x_i) + d_t(x_i) \leq \min_{x_i \in B} d(x_i) + \min_{x_i \in E} d_t(x_i)$$

Nicholson proves this condition is correct, and works out the example in Fig. 1.1.

1.6 Stopping Conditions

Dreyfus[20] suggests an alternate terminating condition to that of Nicholson.

Terminate when there is some node

$n \in A \cap D$ and look at

$n \cup Y$ $\quad Y = \left\{ x_i : x_i \in A \cap E \right\}$

The shortest path will be found by

$$\min \left(\forall_{x_i \in Y} \; d(x_i) + d_t(x_i) \; , \; d(n) + d_t(n) \right) .$$

This condition is simpler to check and ordinarily saves computation. Nicholson's condition requires recomputation every iteration, whereas the Dreyfus criterion is a test on set inclusion, and can be done by means of simple Boolean flags (see Appendix I).

A further refinement of the Dreyfus condition is possible. Nicholson requires that all ties be treated in the same iteration. This is unnecessary. Assume, that

only one at a time is handled, ties being broken in order of node number. Then consider that a given node n occurs in the intersection of A and D stopping the computation. If n was last placed in set A, and if there are some rules in set B with $d(x_i) = d(n)$, $x_i \in B$ then place these nodes in A. The termination step need only look at $x_i \in A \cap E$ and n. However, set E in this case may not include some nodes on its perimeter which would have been simultaneously included in the Dreyfus-Nicholson method. Therefore the terminating condition treats a smaller set. A formalization of this refinement appears later, along with a proof of correctness.

The importance of these stopping procedures is that they allow an essentially analogue process to be treated digitally. Early proposals of bi-directional methods were in error because of an incorrect terminating condition. The typical mistake was to assume $n \in A \cap D$ was always on the shortest path.[8,20] Unfortunately, notation and complex terminating conditions obscure a basically simple and naive algorithm. Hopefully, the following amoeba model will elucidate the ideas described above.

CHAPTER 2

GENERALIZATION OF THE SHORTEST PATH ALGORITHM

2.1 Intuitive Description

Picture two amoeba, one dyed red and the other dyed blue. The red one is placed on the starting node s, and the blue one is placed on the terminating node t. Only the behavior of the red amoeba will be described in detail as the blue amoeba behaves analogously. The red amoeba moves at a velocity V_r. If the red amoeba reaches a node, it splits into the number of outgoing edges (edges where the initial node is the node where the amoeba is), with one progeny traveling each edge. The red amoeba and its progeny all travel at the same speed V_r. The blue amoeba and progeny have speed V_b and are performing in the same fashion with respect to ingoing edges. The first two amoeba of different lineage to meet have traveled the shortest path from s to t. Let dr(t) = the distance covered by red amoeba in time t and let db(t) = the distance covered by a blue amoeba in time t. At any time t these functions represent the distance covered by all amoeba of the corresponding color. If t* is the time at which two amoeba of different color meet, then the distance traveled would be dr(t*) + db(t*). Since dr and db are both monotonic functions with respect to time, any pair of amoeba meeting at $t^* + \epsilon$, $\epsilon > 0$ would have traveled more than the pair that met first. Therefore the above procedure is correct.

The major complication in implementing the above algorithm by a discrete process is to have a correct stopping criterion.†

This intuitive description covers all the standard shortest path methods and one type not previously proposed.

a. $V_r \neq 0$, $V_b = 0$ forward uni-directional algorithm[17]

b. $V_r = 0$, $V_b \neq 0$ backward uni-directional algorithm

†Reference 8, p. 174 gives a bi-directional algorithm with an incorrect termination criterion (also see Ref. 7).

c. $V_r = V_b \neq 0$ unbiased bi-directional algorithm[44]

d. $V_r \neq V_b$, $V_r \neq 0$, $V_b \neq 0$ weighted bi-directional algorithm (proposed here)

If properly trained amoeba could be found, the above represents an effective procedure for finding the shortest path, providing a path of finite length exists. However, there is no guarantee that the amoebas traveling the shortest path will meet at a node. A digital simulation of the above algorithm must have a complicated stopping criterion.

Before going on to a formal description of my method the reader should try to apply variants a, b, and c described above to Fig. 2.1 with results as follows:

Each finds the shortest path $\mu = (s, c, f, d, t)$, $(\mu) = 21$

a. Red amoebas visit all nodes of the graph.

b. Blue amoebas visit all but node e.

c. Red amoebas visit s, e, c, g

Blue amoebas visit t, d, b, f.

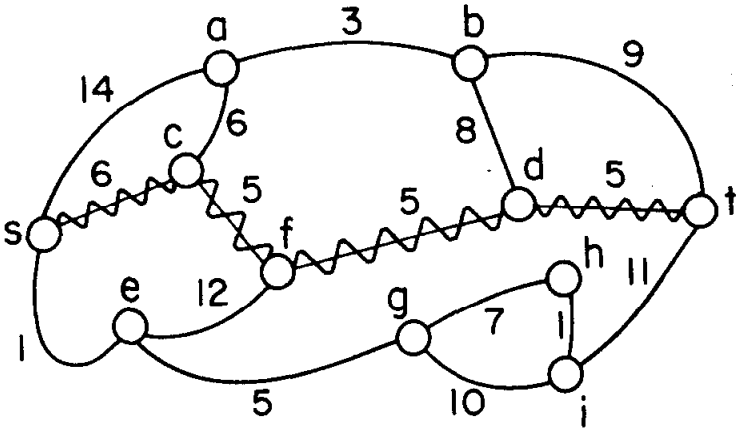


FIG. 2.1--A shortest path from s to t of length 21 is shown in the above graph. Each edge is undirected and may be thought of as representing two directed edges pointing in opposite directions.

2.2 Formal Description of the Very General Algorithm

Notation

s = starting node

t = terminal node

S = set of nodes reached from s

T = set of nodes reached from t

$\tilde{S}$ = set of nodes reachable along one edge from S but not in S

$\tilde{T}$ = set of nodes reachable along one edge from T but not in T

$g_s(x)$ = current distance from s to x

$g_t(x)$ = current distance from x to t

$wf(x)$ = immediate predecessor node from which x was reached

$wt(x)$ = immediate successor node from which x was reached

S, T, $\tilde{S}$, $\tilde{T}$, $g_s(x)$, $g_t(x)$, $wf(x)$, $wt(x)$ will change throughout the computation; they are functions of the iteration step in the computation.

Algorithm

1. Place s in S and calculate all successors, placing them in $\tilde{S}$.

   For each successor x_i, calculate

   $$g_s(x_i) := \ell\left(e_{sx_i}\right), \quad wf(x_i) := s \ .$$

   Similarly place t in T and calculate all predecessors of t, placing them in $\tilde{T}$. Each predecessor of t has

   $$g_t(x_i) := \ell\left(e_{x_i t}\right), \quad wt(x_i) := t \ .$$

2. Decide to look at either $\tilde{S}$ or $\tilde{T}$.

3. If $\tilde{S}$ was selected in step 2, then select a node $x \in \tilde{S}$ which has the smallest value of g_s. If $\tilde{T}$ was selected then this step and the following would be carried out with respect to $\tilde{T}$ and associated functions. If more than one node minimizes g_s, then select all of them $x_1, \ldots, x_k$.

4. Remove $x_1, \ldots, x_k$ from set $\tilde{S}$ and place in set S. If any of these nodes satisfy $x_i \in S \cap T$, then go to the terminal step 6.

5. For each x_i, calculate its successors (predeeessors in case of $\tilde{T}$) and their values of g_s. If these nodes already have a value of g_s and the new calculation is greater or equal to the old value, then leave alone. If $g_s(x)$ is calculated where y is its predecessor, then $g_s(x) = g_s(y) + \ell(e_{yx})$. However, if the new value is less or has not previously been calculated, then place the node in set $\tilde{S}$ and make x_i the value of wf. Return to step 2.

6. The minimum distance is for

   $\forall x_i$ such that $x_i \in S \cap (T \cup \tilde{T})$ pick x_i such that $g_s(x_i) + g_t(x_i)$ is a minimum. The path can be found by tracing through wf and wt. (Note $x_i \in T \cap (S \cup \tilde{S})$ would work equally well.)

In the above algorithm, step 2 is not an effective computational rule. The point here is that any decision rule can be used as will be proved below. If rule 2 is always to choose $\tilde{S}$, we would have the forward uni-directional method. The unbiased bi-directional method selects $\tilde{S}$, if

$$g_{\tilde{S}}^{\min} \leq g_{\tilde{T}}^{\min}$$

otherwise it selects $\tilde{T}$ where

$$g_{\tilde{S}}^{\min} = \min_{x \in \tilde{S}} (g_s(x))$$

$$g_{\tilde{T}}^{\min} = \min_{x \in \tilde{T}} (g_t(x)) \quad .$$

2.3 Proof of the Correctness of VGA

We wish to prove that regardless of what decision rule is used in step 2 of VGA, the algorithm will correctly find a shortest path. The graphs of interest will be finite connected graphs with positive edge weights.

Notation

Let the successive sets created by iterations of VGA be S^0, S^1, S^2, ... Similarly $\tilde{S}^0$, $\tilde{S}^1$, ... Let $\bar{g}_S(S^i)$ be the maximum of the current values $g_S(x_j)$ for $x_j \in S^i$, and $\underline{g}_S(\tilde{S}^i)$ be the current minimum for $x_j \in \tilde{S}^i \cdot S^0$ is $\{s\}$ with $\bar{g}_S(S^0) = 0$ always. The corresponding notation will be used for T and $\tilde{T}$. It is obvious from VGA that the order of creation of sets S^i and T^i by different decision rules in step 2 of VGA do not affect their composition.

Lemma 2.1

A node placed in set S is never returned to set $\tilde{S}$. The corresponding result is true for nodes in set T.

Proof

Consider step 3 of VGA. This states that nodes x_i with $g_S(x_i) = \underline{g}_S(\tilde{S}^j)$ are selected on the jth use of set $\tilde{S}$. Since edge lengths are positive, any successor of these nodes will have a larger distance value.

$$\therefore \underline{g}_S(\tilde{S}^j) < \underline{g}_S(\tilde{S}^{j+1})$$

so once a node is placed in set S, any future value must be larger.

■

By this lemma VGA need never recompute a distance for any node placed in S on a previous iteration. This is not true of nodes in $\tilde{S}$, which can have better values calculated in later iterations.

Lemma 2.2

All nodes x with a path of length less than or equal to $\bar{g}_s(S^i)$ from s are in S^i. The corresponding result is true for nodes in set T.

Proof

Consider the first set, S^n, for which the lemma is false. There is some optimal path

$$\mu = (s, x_1, x_2, \ldots, x_k, y)$$

with the fewest number of nodes for which it fails.

$$\ell(\mu) \leq \bar{g}_s(S^n),\ y \notin S^n$$

Now $x_k \in S^n$ since it is along a path of fewer nodes.

If $x_k \in S^{n-1}$ then $y \in \tilde{S}^{n-1}$ and step 3 would have placed y in S^n since $g_s(y) < \bar{g}_s(S^n)$.

So $x_k \in S^n$ and $x_k \notin S^{n-1}$

$$\therefore\ \bar{g}_s(S^n) = g_s(x_k)$$

but $g_s(x_k) < \ell(\mu) \leq \bar{g}_s(S^n)$ Contradiction. ■

The lemma is true for $S^0 = \{s\}$; so there can be no first S^n for which it is false.

This shows that sets S and T are shortest path trees grown from s and t respectively.

Theorem 2.1: VGA terminates, always finding the shortest path from s to t, with any decision rule used in step 2.

Proof

a. The algorithm terminates.

|S| + |T| is monotonically increasing with each iteration of VGA. Step 4 always adds at least one node. When |S| + |T| > |G|, there is some node n such that $n \in S \cap T$. (Note that even if set $\tilde{S}$ is always

selected in step 2, eventually by the above argument, and the fact that $t \in T$, VGA would halt with $t \in S \cap T$.)

b. Upon termination a path from s to t is found.

At termination there exists some $n \in S \cap T$. Therefore as noted in lemma 2.2 there is some path from s to n, and some path from n to t. So, here is at least one path through n, which goes from s to t.

c. The path found in step 6 of VGA is the shortest path from s to t.

Node n was found in step 4, where $n \in S \cap T$, and step 4 placed n in S during the final iteration of VGA. The argument is symmetric if n was last placed in set T.

If n was placed in S on the kth use of $\tilde{S}$, and in T on the jth use of $\tilde{T}$, then the path through n has length $= \bar{g}_s(S^k) + \bar{g}_t(T^j)$ or abbreviated to $\bar{g}_s + \bar{g}_t$.

Now assume that this is not the shortest path, but that it is μ^*.

$$\mu^* = (y_1, y_2, \ldots, y_j), \quad y_1 = s, \quad y_j = t$$

For path μ^* there is some y_i such that

$$g_s(y_i) \leqq \bar{g}_s \quad \text{and} \quad g_s(y_{i+1}) > \bar{g}_s \ .$$

$$g_s(y_1) = \bar{g}_s(S^0) = 0 \leqq \bar{g}_s$$

or if $g_s(y_{i+1}) \not> \bar{g}_s$, then $y_i = t$ and consequently by lemma 2.2, $n = t$ and the shortest path would be found.

Let $g_s^*(x_i)$ and $g_t^*(x_i)$ denote optimal distances from s and t respectively. Since μ^* is the shortest path

$$g_s^*(y_{i+1}) + g_t^*(y_{i+1}) < \bar{g}_s + \bar{g}_t$$

$\bar{g}_s < g_s(y_{i+1})$ from above and thus $g_t^*(y_{i+1}) < \bar{g}_t$.

By lemma 2.2, y_{i+1} must be in T. Therefore $y_i \in \tilde{T}$ from the execution of step 5 of VGA. So $y_i \in S \cap \tilde{T}$, and the shortest path would have been found by step 6. ■

So we have proved that any decision rule inserted in step 2 leaves VGA correct. Some examples of previously used decision rules are:

1. Always use set $\tilde{S}$

   Dijkstra's procedure — what we call the forward uni-directional method.

2. Always use set $\tilde{T}$

   The backward uni-directional method.

3. Alternate between $\tilde{S}$ and $\tilde{T}$

   Dantzig's procedure.

4. Let $g_{s_{min}}$ be the current minimum for $\tilde{S}$ and $g_{t_{min}}$ be the current minimum for $\tilde{T}$. If $g_{s_{min}} \leq g_{t_{min}}$ use $\tilde{S}$ otherwise use $\tilde{T}$.

   Nicholson's procedure.

VGA as described uses Dreyfus' terminating condition. In section 1.6 we proposed a further refinement to this terminating condition which we now prove to be equivalent to the Dreyfus condition.

Change step 3 to read: "If more than one node minimizes g_s, then select any one of them." Change step 6 to read: "If n found by step 4 is last placed in set S, then include all nodes x_i with $g_s(n) = g_s(x_i)$ in set S" (their successors need not be computed). Correspondingly, if n is last placed in set T, perform the symmetric calculation, and look at $x_i \in T \cap (S \cup \tilde{S})$.

Theorem 2.2: VGA as redefined above is still correct.

Proof

The proof will consider the case where n is placed in set S last. The argument is symmetric for set T.

In the regular VGA let S, $\tilde{S}$, T, $\tilde{T}$ be the sets upon termination and in the redefined VGA (called VGAR) call these sets S_-, $\tilde{S}_-$, T_-, $\tilde{T}_-$.

a. $S \equiv S_-$

b. $T \supset T_-$

In step 6 of the VGAR set S_- is completed with respect to ties. Now in VGA the terminating step considers

$$S \cap (T \cup \tilde{T})$$

and in VGAR the terminating step considers

$$S_- \cap (T_- \cup \tilde{T}_-)$$

$$T_- \cup \tilde{T}_- \subset T \cup \tilde{T}$$

This is true from b and the fact that the successors of the subset must be either in the set T or in the successor set $\tilde{T}$.

Let k be the last node included in T_- by VGAR at a distance $g_t(k)$. If there were no ties at this distance $T_- \equiv T$ and $\tilde{T}_- \equiv \tilde{T}$ and the shortest path would be found by VGAR.

Call the optimum path

$$\mu = (s, x_1, x_2, \ldots x_k, t)$$

with $x_i \in S$, $x_{i+1} \notin S$. If $x_{i+1} \in T$ then $x_{i+1} \in T_- \cup \tilde{T}_-$. This is because the only nodes in T that are not in T_- must be nodes at $g_t(k)$, but all these nodes must be in $\tilde{T}_-$, since they are predecessors of nodes x_i with $g_t(x_i) < g_t(k)$. So if $x_{i+1} \in T$ then the path would be found. Assume $x_{i+1} \in \tilde{T}$ but not in $\tilde{T}_-$. Then $x_{i+2} \in T$ and so $g_t(x_{i+2}) \leq g_t(k)$, in fact $g_t(x_{i+2}) = g_t(k)$. If $g_t(x_{i+2}) < g_t(k)$ then $x_{i+2} \in T_-$ and $x_{i+1} \in \tilde{T}_-$.

Now $\ell(\mu) \leq g_s(n) + g_t(n)$ and $g_t(n) \leq g_t(k)$ since $g_t(k)$ is the last value used for set T.

$$g_s(x_{i+1}) + g_t(x_{i+2}) \leq g_s(x_{i+2}) + g_t(x_{i+2}) \leq g_s(n) + g_t(k)$$

$$\therefore g_s(x_{i+1}) \leq g_s(n) \quad \text{and} \quad x_{i+1} \in S \text{ contradiction.} \qquad \blacksquare$$

The refinement allows a smaller set of nodes to be calculated, in that nodes which are tied on the periphery of set T need not be computed. The refinement is of practical interest for graphs with identical integral length edges.

2.4 Extensions of VGA

The two point shortest path problem is the basic problem in the area. There are numerous variants of this problem[20,23] and some of these can be computed by VGA with minimal modification.

Multiple Endpoints

Instead of a path from s to t, we may be interested in the shortest path in G from any node in subset A to any node in subset B, where A and B are subsets of X.

Given $s_i \in A$, $t_j \in B$ find some μ^*, a path from s_i to t_j such that $\ell(\mu^*)$ is a minimum over all paths for $\forall_{ij}$ from s_i to t_j.

Initialize set S in VGA to set A and set T to set B and proceed as usual, and we have an algorithm for this problem.

Disjoint Components

We cannot always be sure the graphs of interest are connected, thus there may not be a path from s to t. If we always include an edge from s to t of length inf where

$$\text{inf} > |G| \cdot \max_{e_{ij} \in U} \left(\ell(e_{ij})\right);$$

then if the algorithm terminates with a path of length inf we know that no path exists.

All Shortest Paths

Ordinarily we want any shortest path from s to t, but a secondary criterion may be of interest (most scenic shortest path). So, first we want to find all shortest paths. This is done by modifying step 6 of VGA to return all paths of minimum length. Along with this, $wf(x_i)$ must be extended to a multiple entry table in order that ties be stored as the algorithm places nodes in $\tilde{S}$. Then step 5 is modified to allow predecessors (successors in expansion from t) which are along equal length sub-paths to all be stored in the wf(wt) table.

CHAPTER 3

SHORTEST PATH SPACE

The correctness of VGA regardless of what decision rule is used in step 2 brings up the question of what rule to use. If one wanted, set $\tilde{S}$ or $\tilde{T}$ could be chosen by using a coin toss or a pseudorandom number generator. This seems an unintelligent way to make the decision. Similarly, there seems no apparent point to always picking set $\tilde{S}$ (or set $\tilde{T}$), which is the uni-directional approach. To determine a good rule, we must have the appropriate criterion.

VGA has an inner loop of steps 2 through 5 and the work the algorithm does is related directly to number of nodes placed in set S and set T. This places an upper bound on the number of iterations, $|G|$. Each node involves calculating the g value of its successors and can have up to $|G|$ successors. This places a bound on the computation of $O(|G|^2)$ operations. Ordinarily, the search does not include all the nodes of G. The cardinality of S and T, $|S| + |T|$ provides a natural measure of computational efficiency. For a particular problem where μ is the shortest path of interest:

$$\mu = (x_1, \ldots, x_k)$$

$$s = x_1, \quad t = x_k$$

$$k = |\mu| \leq |S| + |T| \leq |G| \quad .$$

3.1 Graph Density

To explore more exactly the meaning of efficiency in shortest path computation, some concepts on density and distribution of nodes in a graph will be developed.

Consider $\bar{\lambda}$ = the shortest distance from s to t in graph G.

Let $d_f^{\lambda}(n)$* = the number of nodes reachable in $\leq$ distance λ from node n.

Let $d_b^{\lambda}(n)$ = the number of nodes that can reach node n in $\leq$ distance λ .

In Fig. 2.1

$$d_f^6(s) = 4 \quad \text{nodes s, e, g, c}$$
$$d_b^{10}(t) = 4 \quad \text{nodes t, d, b, f}$$
$$d_f^{21}(t) = 11$$

The number of nodes placed in set S by a forward uni-directional method is $d_f^{\bar{\lambda}}(s)$. The corresponding number for the backward uni-directional method is $d_b^{\bar{\lambda}}(t)$. The unbiased bi-directional method looks at approximately

$$d_f^{\bar{\lambda}/2}(s) + d_b^{\bar{\lambda}/2}(t) \quad .$$

this is not exact because there may not be a node at distance $\bar{\lambda}/2$ from each endpoint. However, we will assume that for sufficiently large problems the above expression is accurate.

A weighted graph does not necessarily have a Euclidean representation when the edges are considered straight lines. The triangle inequality is often violated. We may induce a useful planar representation of the nodes in a graph for a

---

* $\Gamma^i(n)$ is the normal notation for the nodes i edges from node n. This is the sense of our superscript notation.

particular shortest path problem. Place node s at the origin and note t at distance $\bar{\lambda}$ on the x-axis. Each node x_i in the graph under consideration will have two coordinates, r_1 and r_2. The first coordinate is the shortest distance from node s to x_i and the second coordinate is the shortest distance from node x_i to t. The node x_i will be placed in the upper half plane with distance r_1 from the origin and r_2 from node t. If a node in our graph is not connected to s or t we will not be interested in it. In Fig. 3.1 this mapping is shown for Nicholson's graph (see Fig. 1.1).

Theorem 3.1: For all $x_i \in G$

$$r_1 + r_2 \geq \bar{\lambda} \tag{3.1}$$

$$r_1 + \bar{\lambda} \geq r_2, \quad r_2 + \bar{\lambda} \geq r_1 \tag{3.2}$$

Proof

1. if for some $n \in G, r_1 + r_2 < \bar{\lambda}$ then there is a path μ_n through n such that $\ell(\mu_n) < \bar{\lambda}$ which contradicts the fact that $\bar{\lambda}$ is a minimum distance.
2. Suppose $r_1 + \bar{\lambda} < r_2$

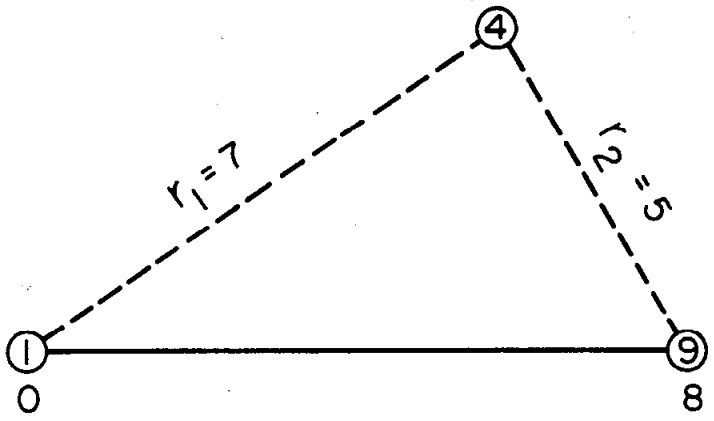
Then there is a path from n to s and from s to t of length $r_1 + \bar{\lambda}$ and therefore r_2 is not the shortest distance from n to t. ■

VGA expands a la Huygens' wavefronts from both s and t. The space these wavefronts are propagating in is the one just described. The most efficient decision rule for VGA is the one where r is found such that

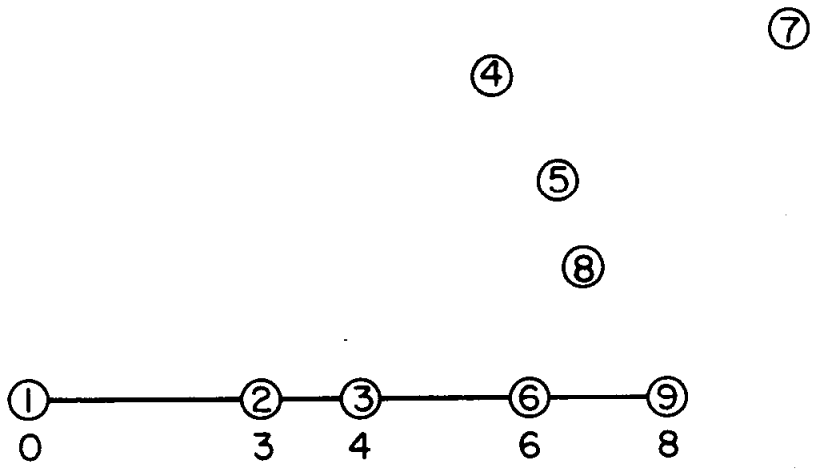
$$\left[d_f^r(s) + d_b^{\bar{\lambda}-r}(t)\right] \tag{3.3}$$

is a minimum over $0 \leq r \leq \bar{\lambda}$. We wish to approximate the above description by a continuous one which is easier to discuss analytically.

$$\rho_s(\lambda) \approx \frac{d_f^{\lambda+\Delta\lambda}(s) - d_f^{\lambda}(s)}{\Delta\lambda}$$



(a) PLACING NODE 4 IN SHORTEST PATH SPACE



(b) THE FULL GRAPH

1269A12

FIG. 3.1--Mapping Nicholson's graph into shortest path space.

and

$$\rho_t(\lambda) \approx \frac{d_b^{\lambda+\Delta\lambda}(t) - d_b^{\lambda}(t)}{\Delta\lambda} .$$

These functions may be considered as density functions for a given shortest path problem. The most efficient algorithm is now the one which goes distance r from s and distance $\bar{\lambda}$-r from t such that

$$\left[\int_0^r \rho_s(\lambda)\, d\lambda + \int_0^{\bar{\lambda}-r} \rho_t(\lambda)\, d\lambda\right] . \qquad (3.3')$$

is a minimum over $0 \le r \le \bar{\lambda}$.

To find r requires a priori knowledge of ρ_s and ρ_t, where in most cases these can only be determined a posteriori. Therefore unless the shortest path space is in some way characterized previous to solving problems in it, there is no assured way of having an optimum decision rule (namely one which says pick set $\tilde{S}$ as long as $g_{\tilde{S}}^{\min} \le r$; otherwise pick set $\tilde{T}$). For example if someone told you the problems of interest are in lattices in E_n* where most of the possible connections exist, then

$$\rho_s(\lambda),\ \rho_t(\lambda) \propto c\lambda^{n-1}$$

$$\int_0^r c\lambda^{n-1}\, d_\lambda + \int_0^{\bar{\lambda}-r} c\lambda^{n-1}\, d_\lambda = c\,\frac{r^n}{n} + \frac{c(\bar{\lambda}-r)^n}{n} = F(r)$$

$$\frac{dF}{dr} = c\, r^{n-1} - c(\bar{\lambda}-r)^{n-1}$$

$$\frac{dF}{dr} = 0 \quad \therefore \text{ minimum at } r = \frac{\bar{\lambda}}{2}$$

(3.3') evaluates to

$$\frac{2c}{n}\left(\frac{\bar{\lambda}}{2}\right)^n$$

where a uni-directional method would give

$$\int_0^{\bar{\lambda}} c\lambda^{n-1}\, d\lambda = \frac{c}{n}\bar{\lambda}^n$$

a factor of $(1/2)^{n-1}$ is gained.

*These are n-tuples $(x_1, x_2, \ldots, x_n)$ where all x_i are integers.

The above argument underlies the obvious choice of unbiased bi-directional methods, and is why some experts have been moved to dismiss weighted methods.[12] It seems that they implicitly assume a symmetric distribution of nodes.

3.2 A Locally Optimal Decision Rule

In using VGA, on each iteration the cardinality of $\tilde{S}$ and the cardinality of $\tilde{T}$ would be known. These numbers would reflect the local density of the regions adjoining S and T. This information on cardinalities requires no additional computation and is a simple a priori estimate of density. Then one reasonable decision rule for step 2 of VGA would be if $|\tilde{S}| \leq |\tilde{T}|$ use set $\tilde{S}$ or else use set $\tilde{T}$ (cardinality comparison strategy). VGA with this rule will be called a weighted bi-directional (WBIDI) method. (UBIDI will mean the unbiased bi-directional method.)

Examine Fig. 3.2 where each edge is of length one. The most efficient algorithm is the backward uni-directional method. It would visit only those nodes on

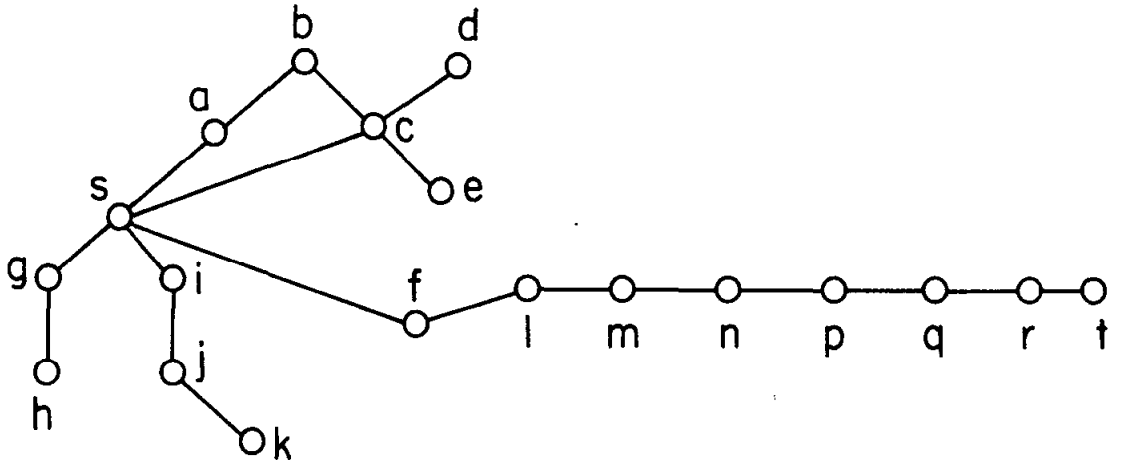


FIG. 3.2--Graph with edge lengths all unity.

the shortest path from s to t. The forward uni-directional method and UBIDI would visit all nodes in the graph. The WBIDI method, described above, would also visit only those nodes on the shortest path. If s and t were interchanged, the best method would now be the forward uni-directional method and WBIDI. UBIDI in no way accounts for the difference in densities. In each case distance $\bar{\lambda}$ needs to be covered by the respective versions of VGA. A local approximation to density in regions $\tilde{S}$ and $\tilde{T}$ are known and without other information, progress in the sparser region requires less work. To add substance to this argument we take two complementary approaches to verifying this optimality. One approach is to generate large numbers of graphs and compare the efficiency of the different strategies. Additionally one can analyze a posteriori how close each strategy came to minimize r_{opt}. The other approach is to calculate the expected number of nodes placed in S and T over a given class of path problems and compare the strategies on this basis. In both instances our geometric intuition outlined above is vindicated.

CHAPTER 4

ON THE OPTIMALITY OF OUR DECISION STRATEGY

4.1 Probabilistic Analysis

We wish to analyse the efficiency of different decision strategies, and to demonstrate the efficacy of cardinality comparison. In order to do this, we will take a probabilistic model of our algorithm as applied to some path problem, and attempt to calculate the expected number of steps needed by a given decision strategy. VGA for a given problem finds a path

$$\mu = (s, x_1, x_2, \ldots, x_k, t) \ .$$

At each iteration of the algorithm, sets S, $\tilde{S}$, T, and $\tilde{T}$ are changed, with $|S|$ and $|T|$ increasing. For simplicity, we will use VGAR, where $|S^{i+1}| = |S^i| + 1$, i.e., only one node is added at a time. The sequences $|S^0|$, $|S^1|$, ..., and $|T^0|$, T, ... are monotonically increasing. Also, in general, $|\tilde{S}^i|$ and $|\tilde{T}^i|$ are monotonically increasing, since the graphs of interest are connected with average degree greater than one. Monotonicity of these tilde sets, $\tilde{S}$ and $\tilde{T}$, is our hypothesis and our experiments show this.*

Our algorithm must find and place in set S or in set T the nodes x_1, x_2, ... x_k. On any given iteration some node $n \epsilon \tilde{S}^i$ (or $n \epsilon \tilde{T}^j$) is selected. At this point in the computation some $x_\ell \epsilon \mu$ is in set $\tilde{S}^i$ and likewise some $x_m \epsilon \mu$ is in set $\tilde{T}^j$ - to see this go back to the proofs of the correctness of VGA and VGAR. When a node along μ is placed in set S its successor and the associated minimal distance function is placed in set $\tilde{S}$. Thus, if we have a unique shortest path μ, then at any time in the computation we are interested in finding the next successor

*A violation of this condition is called pathological, and is discussed later.

node from s currently in $\tilde{S}$ and the next predecessor node from t in $\tilde{T}$. Since, we have no additional information, only the fact that some node on the shortest path is in $\tilde{S}^i$ or $\tilde{T}^j$, the probability on a given iteration of finding a node along the shortest path is $1/|\tilde{S}^i|$ or $1/|\tilde{T}^j|$ depending on which set is selected by VGAR step two. A decision rule in VGAR corresponds to some sequence of choosing either $\tilde{S}$ or $\tilde{T}$, and is expressible as D = SSTST...T for a given problem. Each strategy has an expected number of steps until success which may be calculated and compared with other strategies for a given path problem. The k nodes along the shortest path μ must be found one at a time from either end, and we will show that cardinality comparison has the smallest number of expected steps over all strategies for doing this, under the monotonicity hypothesis for the tilde sets. To prove this rigorously, we now derive some fundamental results from probability theory about expected values of decision problems of this type.

Theorem 4.1:

Let $a_1, a_2, \ldots, a_k$ be a monotonic increasing sequence, and $b_1, b_2, \ldots, b_k$ be some other sequence. Consider

$E(\tau) = \sum_{i=1}^{k} a_i b_{\tau}(i)$ where τ is a permutation over the index set, $[1, 2, \ldots k]$. Then

$E_{min} = \min_{\forall \tau} [E(\tau)]$ is given by a permutation where

$b_{\tau(1)}, b_{\tau(2)}, \ldots, b_{\tau(k)}$ is a monotonic decreasing sequence.

Proof

Consider $E(I) = \sum_{i=1}^{k} a_i b_i$ (The identity permutation)

Take the first

*) b_i such that $b_i < b_{i+1}$, and interchange b_i with b_{i+1}.

$$a_i b_i + a_{i+1} b_{i+1} > a_i b_{i+1} + a_{i+1} b_i$$

$$(a_{i+1} - a_i)\, b_{i+1} > (a_{i+1} - a_i)\, b_i$$

$$b_{i+1} > b_i$$

The sequence formed by the interchange has a smaller value of E than the original sequence. For any ordering $b_{\tau(i)}$, if an interchange of the form (*) is possible, then the new value of $E(\tau)$ is smaller. The only sequence where this interchange is not possible is $b_{\tau(i)} > b_{\tau(i+1)}$ for all i and this is a monotonic decreasing system. ■

Corollary 4.1:

$$\text{For } E_{max} = \max_{\forall \tau} [E(\tau)]$$

we have $b_{\tau(1)}, b_{\tau(2)}, \ldots, b_{\tau(k)}$ a monotonic increasing sequence.

Proof

The argument is completely symmetric with the above. ■

We call the above proof a proof by bubble sort. This result is also found in Ref. 29.

We wish to calculate the expected number of steps until one success occurs for some permutation of probabilities $p_1, p_2, \ldots, p_k$.

$$E = \sum i \cdot (\text{probability of success on the ith step but not before})$$

$$= \sum_{i=1}^{k+1} i \cdot p_i \cdot \prod_{j=1}^{i-1} q_j \qquad (4.1)$$

where $q_j = 1 - p_j$ and $p_{k+1} = 1$. This last probability means that the sequence always terminates on the k+1st turn if success has not been previously achieved.

Theorem 4.2:

The value of the expectation (4.1) is a minimum if the probabilities are in monotonically decreasing order.

Proof

Let us assume that the sequence $p_1, p_2, \ldots, p_k$ is in monotonic decreasing order. Then $q_1, q_2, q_3, \ldots q_k$ will be in monotonic increasing order. By theorem 4.1 $\sum i \cdot p_i$ is a minimum, since $a_1 = 1$, $a_2 = 2$, $\ldots$ $a_k = k$ is a monotonic increasing sequence.

If p_1 is the largest value then q_1 is the smallest value. Now the product terms are

$$q_1, \; q_1q_2, \; q_1q_2q_3, \; \ldots, \qquad \prod_{i=1}^{k} q_i ,$$

and these terms, corresponding to the ordering of the p's, are the smallest possible. Any other ordering of p's gives larger product terms, since in the original case we use a smallest first criterion. So

$\sum i \cdot p_i$ is a minimum and $\sum i \cdot p_i \cdot \prod q_i$ must also be a minimum since the $\prod q_i$ are the smallest possible factors. The k+1st term of the expectation

$$(k+1)\, p_{k+1} \prod_{i=1}^{k} q_i \qquad \text{with } p_{k+1} = 1$$

is the same for all possible orderings and as a constant does not affect the arguments from monotonicity. ■

So if our game consists of continuing to play until one success occurs, then playing it with the probabilities of success in monotonic decreasing order is our best strategy for finishing fastest. Now, we may want to play until two successes have occurred and are interested in how best to proceed.

If $p_1, p_2, p_3, \ldots, p_k$ is the order of probabilities of single success then

$$r_2 = p_1 p_2$$

$$r_3 = p_3(p_1 q_2 + q_1 p_2)$$

$$\vdots$$

$$r_i = p_i \left(p_1 \prod_{j=2}^{i-1} q_j + p_2 \prod_{\substack{j=1 \\ j \neq 2}}^{i-1} q_j + \ldots + p_{i-1} \prod_{j=1}^{i-2} q_j \right)$$

$$\vdots$$

where the r's represent the probability of success on a given step but not before. For any ordering $p_{\tau(i)}$ there is a corresponding set of r's.

Theorem 4.3:

The expected number of steps in the two success game will be minimized for p_i monotonic decreasing.

Proof

$$E = \sum_{i=2}^{k+1} i \cdot r_i$$

$r_1 = 0$ since the game cannot end in one step

$$r_{k+1} = \left(\prod_{j=1}^{k} q_j + p_1 \prod_{j=2}^{k} q_j + \ldots + p_k \prod_{j=1}^{k-1} q_j \right)$$

r_{k+1} is a constant and is independent of the original order of the p's.

We shall proceed akin to the method of theorem 4.1. Consider the first p_i, such that $p_i < p_{i+1}$. In this regard we always make $p_1 > p_2$, since the order of p_1 and p_2 has no effect on E. Let us calculate how the interchange of these two

terms will change E.

$r_2, \ldots, r_{i-1}$ will remain the same

$r_{i+2}, \ldots, r_{k+1}$ will remain the same

$$r_i = p_i \left(p_1 \prod_{j=2}^{i-1} q_j + \ldots + p_{i-1} \prod_{j=1}^{i-2} q_j \right)$$

Let r* represent the redefined sequence.

$$r_i^* = p_{i+1} \left(p_1 \prod_{j=2}^{i-1} q_j + \ldots + p_{i-1} \prod_{j=1}^{i-2} q_j \right)$$

$$r_{i+1}^* = p_i q_{i+1} \left(p_1 \prod_{j=2}^{i-1} q_j + \ldots + p_{i-1} \prod_{\substack{j=1 \\ j=i-1}}^{i-1} q_j + \frac{p_{i+1}}{q_{j+1}} \cdot \prod_{j=1}^{i-1} q_j \right)$$

1) $r_i + r_{i+1} = r_i^* + r_{i+1}^*$

$$r_i + r_{i+1} = p_i \alpha + p_{i+1} q_i \left(\alpha + \frac{p_i}{q_i} \cdot \prod_{j=1}^{i-1} q_j \right)$$

where

$$\alpha = \left(p_1 \prod_{j=2}^{i-1} q_j + \ldots + p_{i-1} \prod_{j=1}^{i-2} q_j \right)$$

$$r_i^* + r_{i+1}^* = p_{i+1} \alpha + p_i q_{i+1} \left(\alpha + \frac{p_{i+1}}{q_{i+1}} \prod_{j=1}^{i-1} q_j \right)$$

Let $\beta = \prod_{j=1}^{i-1} q_j$

$$p_i \alpha + p_{i+1} q_i \left(\alpha + \frac{p_i}{q_i} \beta \right) = p_{i+1} \alpha + p_i q_{i+1} \left(\alpha + \frac{p_{i+1}}{q_{i+1}} \beta \right)$$

$$p_i \alpha + p_{i+1} q_i \alpha + p_i p_{i+1} \beta = p_{i+1} \alpha + p_i q_{i+1} \alpha + p_i p_{i+1} \beta$$

$$\alpha \left(p_i + (1 - p_i) p_{i+1} \right) + p_i p_{i+1} \beta = \alpha \left(p_{i+1} + p_i (1 - p_{i+1}) \right) + p_i p_{i+1} \beta$$

$$\alpha \left(p_i + p_{i+1} - p_i p_{i+1} \right) + p_i p_{i+1} \beta = \alpha \left(p_i + p_{i+1} - p_i p_{i+1} \right) + p_i p_{i+1} \beta$$

1) is true.

2) $i \cdot r_i + (i+1)\, r_{i+1} > i \cdot r_i^* + (i+1)\, r_{i+1}^*$

$i(r_i + r_{i+1}) + r_{i+1} > i(r_i^* + r_{i+1}^*) + r_{i+1}^*$

by (1) this reduces to showing

$r_{i+1} > r_{i+1}^*$

$p_{i+1}q_i \left(\alpha = \frac{p_i}{q_i}\, \beta\right) > p_i q_{i+1} \left(\alpha + \frac{p_{i+1}}{q_{i+1}}\, \beta\right)$

$\alpha p_{i+1}q_i + p_i p_{i+1}\beta > \alpha p_i q_{i+1} + p_i p_{i+1}\beta$

$\alpha p_{i+1}q_i > \alpha p_i q_{i+1}$

Since

$p_{i+1} > p_i$ then $1 - p_i > 1 - p_{i+1}$ so $p_{i+1} > p_i$ and $q_i > q_{i+1}$

$$\therefore\ p_{i+1}q_i > p_i q_{i+1}$$

Therefore the interchange reduces the value of E and by the argument in theorem 4.1 (the bubble sort device), the $p_{\tau(i)}$ which is monotonic decreasing is the best ordering for the two success game.

We now generalize theorem 4.3 to cover the k success game. The proof is identical and parallel to theorem 4.3 exactly.

Theorem 4.4:

The expected number of steps in the k success game will be minimized for p_i monotonic decreasing.

Proof

$$E = \sum_{i=k}^{n+1} i \cdot r_i$$

Consider the first p_i, such that $i \geq k$ and $p_i < p_{i+1}$. The order of the p's before this is irrelevant. Let us calculate how the interchange of these terms will change E.

Then using the same notation as in theorem 4.3 we have:

$r_i = p_i \cdot$ (all combinations of k-1 p's with the rest q's)

$r_{i+1} = p_{i+1} q_i \cdot \Big($all combinations of k-1 p's with the rest q's $+ \dfrac{p_i}{q_i}$

(all combinations of k-2 p's with the rest q's)$\Big)$.

Let α = all combinations of k-1 p's with the rest q's

Let β = all combinations of k-2 p's with the rest q's

$$r_i = p_i\alpha, \quad r_{i+1} = p_{i+1}q_i \cdot \left(\alpha + \frac{p_i}{q_i}\beta\right)$$

$$r_i^* = p_{i+1}\alpha, \quad r_{i+1}^* = p_i q_{i+1} \left(\alpha + \frac{p_{i+1}}{q_{i+1}}\beta\right)$$

But these are the same as in theorem 4.3 and the same proof holds. ■

The introductory discussion of the probabilistic analysis of VGAR showed it to be a k success game where the probabilities of success at a given point are $1/|\tilde{S}^i|$ or $1/|\tilde{T}^j|$ depending on the set chosen. This leads us naturally to the following theorem as a consequence of theorem 4.4.

Theorem 4.5:

An optimal strategy in the sense of the a priori expected work, is to choose the set with the currently fewest nodes, i.e., use set $\tilde{S}^i$ if $|\tilde{S}^i| < |\tilde{T}^j|$ otherwise use set $\tilde{T}^j$.

Proof

The value of a given strategy is the expected number of steps it will take to find the k nodes along the shortest path. For a given problem a strategy corresponds to some sequence of choosing $\tilde{S}^i$ and $\tilde{T}^j$ where the probability of success at any given point is the inverse of their cardinality.

Let us call D* the decision sequence for cardinality comparison, and let the sequence of probabilities corresponding to this strategy be $p_1, p_2, \ldots, p_\ell$. This sequence is monotonic decreasing since using D* together with the monotonicity hypothesis assure this. Therefore according to theorem 4.4 any reordering corresponding to some other bi-directional strategy must be worse.

However a uni-directional method could be used. This would mean that some p_i's will not occur. The sequence that does arise is a subset of the p_i's which are again monotonic. If we call the probabilities arising from a uni-directional strategy q_i then we have $p_i \geq q_i$ and therefore D* must be at least as good as any uni-directional strategy. ■

4.2 Pathological Possibilities

It has been shown that given no a priori information about the structure of shortest path graph, the optimal strategy is cardinality comparison. However, it is possible to produce examples where cardinality comparison is significantly worse than other strategies. The understanding of these examples confirms the validity of the analysis.

Consider Fig. 4.1, where the shortest path of interest is between s and t and is a path of length 7. The forward uni-directional algorithm would visit nodes 1 through 9 (including s and t, of course). The backward method would visit 4 through 19. The cardinality comparison method would find that $|\tilde{S}|$ is 4 and $|\tilde{T}|$ is 3 initially. The set $\tilde{T}$ would be used and its cardinality as is evident from the construction would remain 3. In effect the cardinality comparison method would duplicate the behavior of the backward method.

If some observer were looking down on the graph and watching the behavior of the cardinality comparison algorithm, he could say that two steps away $\tilde{S}$ dies out and therefore one should in fact explore this path in the forward direction.

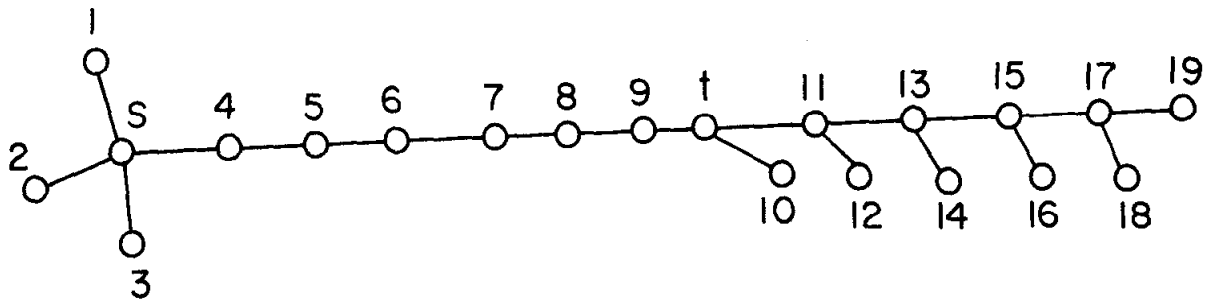


FIG. 4.1--A pathological example.

This behavior could be built into a two step look-ahead, but this increases the cost of each individual decision. The anomolous behavior exhibited in this problem could then be replicated over a graph which was constructed to be dense two steps away from s. If the particular class of problems is noted for highly irregular behavior then a look-ahead feature could be useful as a smoothing out device which is similar to measures needed by hill climbers in avoiding local maxima.

In terms of shortest path space, we have three distinct regions appearing in this graph. A small region around s of high density 4, a large constant region near t of density 3 and a region in between of density 1. So ρ_s, near s is 4 but changes to 1 while ρ_t stays 3. In a graph where $\rho_s = \rho_t = C$, any method works with the same efficiency. Ordinarily a graph with average degree >1 will have a monotonic increasing set of nearest neighbors. Then if one direction is favored for many steps the cardinality of the tilde set on that side grows to exceed the cardinality of the other tilde set. Thus a dynamic balance is maintained which is not appreciably influenced by a local distortion. Another point is that local

distortions of the space are not a priori detectable by one step methods and strategies and in practice rarely occur.

One further anomaly is seen in very dense graphs where a node is just a few nodes away from every other node along some shortest path. In this case $\tilde{S}$ and $\tilde{T}$ will soon exhaust the graph and further iterations will only deplete these sets. In this case the algorithm using cardinality comparison will continue with only one set, where a more symmetric procedure would be efficient. This case, like the previous, does not frequently occur. In standard shortest path problems the graphs are large and sparse. The cases of interest are rarely ones where all shortest path are only 2 or 3 edges long. However in generating high symmetric dense graphs of reasonably uniform weight, these graphs were produced.

CHAPTER 5

EMPIRICAL RESULTS

As a practical test, VGA was programmed in ALGOL W[5, 62] (see Appendix I) and tried on a large number of shortest path problems. Corresponding to step 2 of VGA, was a logical procedure written as a case expression which included decision rules for Dijkstra's forward method, the analogous backward unidirectional method, Nicholson's equi-distance method, and our cardinality comparison rule. The results were gratifying, in that VGA with our rule was the most efficient algorithm.

5.1 Data

In order to obtain a meaningful result, a large number of graphs and path problems using them were needed. A 200 node graph involves 40,000 bits of information, a rather large amount of input. We therefore used a random graph generator.[48] It provided randomly generated graphs of appropriate size and density, weighted or unweighted according to the substituted parameters. For each edge a random number generator produced values over (0, 1), which were compared to the density; if less the edge was included with a length generated randomly over 1, 2, ..., weight (if unweighted then length was uniformly 1). The data was represented in edge list fashion to enable the program to generate very large sparse graphs up to 1000 nodes, 4000 edges.

5.2 Evaluation

The basic measure of efficiency was the number of nodes $|S| + |T|$ at the end of the computation, each method being run for exactly the same data. In addition, a system of a posteriori analysis routines were incorporated into the program to

measure the distribution of nodes in the shortest path spaces used. These routines printed out $d_f^{\lambda}(s)$ and $d_b^{\lambda}(t)$, allowing r_{opt} to be found by inspection (see Eq. (3.3)). In addition the radii of S and T were printed for each method to compare with r_{opt}.

5.3 Results

The results clearly show the advantage of the bi-directional methods over the uni-directional. In all cases investigated, the uni-directional methods visited at least twice the number of nodes as the bi-directional. The Nicholson method and the cardinality comparison method are the same order of magnitude, but invariably the latter is more efficient. The closeness of these methods is because the graphs used were in general symmetric, where $r_{opt} \doteq 0.5*\lambda$ most of the time. However, the Nicholson method is badly out-performed in those examples which a posteriori analysis shows are highly unsymmetrical. A general measure of optimality is the absolute value of the difference between r_{opt} and the r found by the given method. This measure along with the overall comparison of nodes visited favors the cardinality comparison algorithm. The models used in analyzing these algorithms formally are borne out in the empirical test. One note in this regard is the verification of the monotone increasing nature of the sets $\tilde{S}$ and $\tilde{T}$ throughout a computation for even sparse graphs of average degree 2 or 3.

Let us pursue in detail one example in terms of our shortest path space model. We will make the assumption that the space of interest is E^2, because the average degree of our example is approximately the same as for a lattice in E^2. The graph is symmetric and unweighted, of size 200 and average degree 4. The full distribution table is given in the tables marked Example A, table 5.1.

Table 5.1

Distribution Functions, Example A: 200 Node Graph

200 Nodes Maximum Length is 20 Symmetric with Average Degree 4			
Distribution from S	$d_f(s)$	$d_b(t)$	$d_f + d_b$
0	1	85	86
1	2	71	73
2	3	63	65
3	5	55	60
4	6	49	55
5	8	40	48
6	10	37	47
7	12	33	45
8	14	26	40
9	16	24	40
10	19	22	41
11	22	19	41
12	26	17	43
13	32	11	43
14	39	8	47
15	44	6	50
16	48	5	53
17	54	4	58

Table 5.1 (cont.)

Distribution from S	$d_f(s)$	$d_b(t)$	$d_f + d_b$
18	63	3	66
19	68	3	71
20	77	3	80
21	90	3	93
22	101	2	103
23	111	2	113
24	124	2	126
25	129	2	131
26	136	2	138
27	141	2	143
28	149	1	150

Nodes Visited

Forward method	149 nodes
Backward method	85 nodes
Nicholson's method	44 + 11 = 55 nodes
Pohl's method	22 + 22 = 44 nodes

Distance Traveled

Nicholson's method	15 forward	15 backward
Pohl's method	11 forward	18 backward
Uni-directional methods		28

In E we have

δ_f = density of nodes per unit volume in the forward direction

$\pi/2\left(\delta_f r^2 + \delta_b(2 - r^2)\right)$ = number of nodes visited

This is a minimum at

$$\frac{d}{dr}\left(\pi/2\left[\delta_f r^2 + \delta_b(\lambda - r^2)\right]\right) = 0$$

$$\delta_f r + \delta_b(r - \lambda) = 0$$

$$r = \frac{\delta_b \cdot \lambda}{\delta_f + \delta_b}$$

If we assume that δ_f and δ_b are proportional to the results of the unidirectional forward and backward methods, then

$$r_{opt} = \frac{|T_b| \cdot \lambda}{|S_f| + |T_b|}$$

where S_f is the set of nodes visited by the forward method and T_b is the set visited by the backward method. Then for Example A (table 5.1) we have $r_{opt} = 10.1$, $\delta_f = 0.12$, $\delta_b = 0.07$. Using these parameters and the observed radii of sets S and T, the cardinality comparison method would investigate 22.8 + 35.6 nodes — a total of 58.4 nodes, while Nicholson's method would visit 42.3 + 24.7 nodes — a total of 67 nodes. In our case, we have $r_s = 11$ and $r_t = 18$ while in Nicholson's case it is $r_s = 15$ and $r_t = 15$. A posteriori one sees $r_{opt} = 9$, which is very close to what the cardinality comparison method found. The equi-distance approach, while achieving a symmetric search, was less efficient.

The results are in favorable agreement with the model, and reflect the cardinality comparison rule's attempt a priori to minimize Eq. (3.3). While the

discrete nature of the spaces, and the random nature of generation do not make for an exact correspondence of the model to the test cases, the closeness of r_s to r_{opt} in virtually all examples demonstrates the correctness of this approach.

Table 5.2 summarizes the results of tests using 500 node graphs with average degree 3, 6, 9, 12, and 15. Our method requires 1/4 the work of the uni-directionally methods. Nicholson's method visits over 1/3 more nodes than ours, but it compares favorably to uni-directionally methods. These results and more data are presented in greater detail in Appendix II. Our model and the optimality of our cardinality comparison strategy are validated by these experiments.

Table 5.2

Cumulative Results on 500 Node Graphs

Degree	Forward	Backward	Pohl	Nicholson
3	2583	1991	563	662
6	3406	3336	707	1151
9	2724	2924	510	828
12	2627	2434	581	742
15	2521	2596	619	681
Average per case				
	277	266	60	81
Ratio to Pohl's method				
	4.6	4.4	1	1.4

CHAPTER 6

HEURISTIC SEARCH AS A PATH PROBLEM

6.1 Introduction

In many areas of artificial intelligence, improvement has not been evident over the early paradigms.[39] The GPS model[42] has not been superceded and the ideas in what heuristic search is and how to do it have remained the same over the past decade. The situation reminds me of the state of mechanical translation of natural language in the early 1960's. It was at this point that the criticism of Bar-Hillel[4] was becoming convincing. The original ideas of a simple syntactic model and dictionary look-up were seen to not be able to bear the weight of the problem. It was clear that mathematical linguistics had to be better understood. I think in heuristic search the same situation exists. The formal tools need to be developed to better understand and increase the power of heuristic programming. The precise characterizing of these ideas allows not only a quantitative improvement in computational performance, but through a deeper understanding can lead to a qualitative improvement from generalizing and extending these methods.* It is with this spirit and intent that this work is carried out. The formal model of heuristic search presented here has led to a new understanding of solving problems with occasionally unexpected results.

One of the important general models of artificial intelligence is the directed graph† model.[1,2,10,18,19,30,34,37,41,45,55,57] In this model a node

---

*One noteworthy example of this is the current sophisticated use of α-β by Samuels' checker program[54] and Greenblatt's chess program.[27] Early researchers in game playing[21,43] had used the idea without considering it significant enough to explore its ramifications or write on its usefulness.

†For graph theory terminology, see Berge[8] or Ore.[46,47]

contains a description of a possible problem state. If it is possible to get from some state x to state y in a single move (rule of inference, operator, etc.) then there is a directed edge from x to y. More generally, we wish to know if a path exists between two nodes. We distinguish one as the initial node and look for a path to the other, designated the goal node. Such a path is called the solution to our problem. Sometimes we are interested in the shortest path between two nodes, but normally any solution path will be acceptable.

The work of Amarel,[1,2] Michie and Doran[18,19,37] and Hart, Nilsson, and Raphael[30,45] contributed to different aspects of formulating problems in this model. Amarel worked principally on the representation of different problems in this model. Michie and Doran have developed a general problem-solving program, called the Graph Traverser, for finding paths using heuristic functions to control the search for the goal node. Hart, Nilsson and Raphael have given sufficient conditions on heuristic functions to guarantee that a class of path finding algorithms will find the shortest solution path in the space. Algorithms which fall in this class are called admissible.

This work will consider how problems represented in the directed graph model can be solved efficiently. There are two basic extensions over the efforts outlined above. First, the pure heuristic uni-directional search of the Graph Traverser is examined mathematically. This leads to results on the efficient use of the heuristic function and a first theory of the effect of error in the heuristic function. Secondly, the notion of admissible heuristic functions (algorithms)[30] is extended to bi-directional search. Along with admissibility, the question of pragmatically implementing a bi-directional procedure is examined. Associative search implemented by hashing schemes is shown to be a very powerful technique for the redundancy and tree intersection problems. These

questions are not only dealt with theoretically, but our approach is tested using the 15 puzzle as an experimental environment. In all these areas the spirit of the shortest path work is found. We feel, that it is just this viewpoint that has allowed us to discover many new results, some mildly surprising, and to understand more deeply the mechanics of heuristic search. Artificial intelligence is in many ways a branch of discrete mathematics and a science of effective and intelligent enumeration in spirit close to enumerative combinatories. It is not surprising to see that Polya,this century's outstanding combinatorialist,also wrote extensively on how to attack and solve problems.[52]

6.2 Problem Spaces and Heuristic Search

A directed graph G is a set of nodes X and a mapping Γ from the nodes into themselves.

$$G: \quad X = \{x_1, x_2, \ldots, x_n\}$$

$$\Gamma: \quad X \rightarrow X$$

$$E = \{(x_i, x_j) \mid x_i \in X \wedge x_j \in \Gamma(x_i)\}$$

The size or cardinality of the graph is denoted by |G| and can be unbounded. When using directed graphs to characterize problem domains we attach to each x_i a data structure which contains the complete description of the problem. For example, in the case of the 15 puzzle (see Fig. 6.1) a data structure describing it would be the vector (9, 5,1, 3, 13, 7, 2, 8, 14, 6, 4, 11, 10, 15, 12, 0) where 0 denoted the blank position. The mapping Γ would represent possible single moves from one problem state to another. In this domain we are at some initial node (or set of nodes) and wish to reach some goal node (or goal set). We must produce a path from the initial node to the goal node. Purely exhaustive methods are impractical in complicated spaces with |G| and |E| large or possibly infinite. In most instances we have

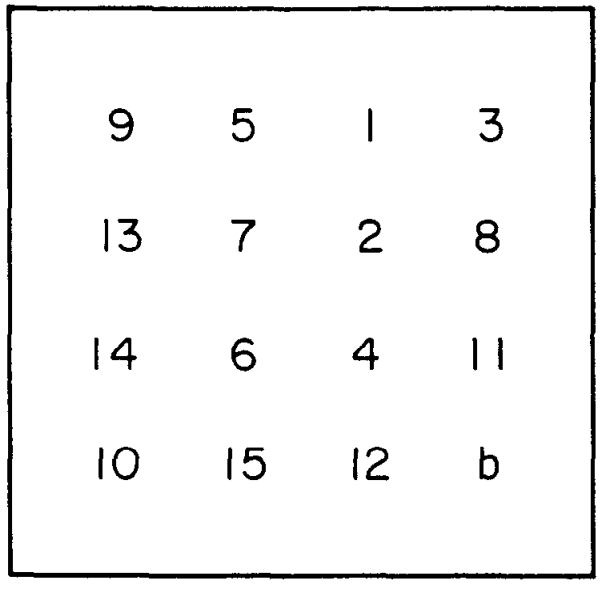


FIG. 6.1--15 puzzle.

heuristics which aid in narrowing the search. For our discussion, heuristic information is a function over state vectors $\vec{v}_i$, attached to the nodes, into the non-negative reals.

An Algorithm for Heuristic Search

When solving most artificial intelligence problems we are not ordinarily interested in the most 'elegant' or shortest path, but in how to obtain any path cheaply. A search method visits a number of nodes in G to find a path. We want this number to be as few as possible, so that it may be computationally feasible to find solution paths which are inherently long; i.e., the shortest path is long.

HPA — Heuristic Path Algorithm

s = initial node

t = goal node

$g(x)$ = the number of edges from s to x,

as found in our search

$h(x)$ = an estimate of the number of edges

from x to t, our heuristic function

$f(x) = g(x) + \omega \cdot h(x)$ $\quad 0 \leq \omega \leq \infty$

By convention if $\omega = \infty$ then $f(x) = h(x)$

S = the nodes that have been visited

$\tilde{S}$ = the nodes which can be reached from S

along an edge, but are not in S

$\Gamma(x)$ = the set of successors of node x

1. Place s in S and calculate $\Gamma(s)$, placing them in $\tilde{S}$.

   If $x \in \Gamma(s)$ then $g(x) = 1$ and $f(x) = 1 + \omega \cdot h(x)$.

2. Select $n \in \tilde{S}$ such that f(n) is a minimum.

3. Place n in S and $\Gamma(n)$ in $\tilde{S}$ (if not already in $\tilde{S}$) and calculate f for the successors of n.

   If $x \in \Gamma(n)$ then $g(x) = 1 + g(n)$ and

   $f(x) = g(x) + \omega \cdot h(x)$.

4. If n is the goal state then halt, otherwise go to step 2.

Note: HPA builds a tree; as each node is reached a pointer to its predecessor is maintained. Upon termination the solution path is traced back from the goal node through each predecessor.

An Example of the Use of HPA

The 15 puzzle is a simple, but combinatorially large problem space. Each space* contains 16!/2 configurations, too large to be searched exhaustively. The average degree (number of moves) of a node is 3, allowing exhaustive search to find solutions of about 10 steps. On the other hand, the space is simple enough to study as an heuristic search problem and heuristic functions are easy to formulate.

*A particular 15 puzzle configuration may be in one of two spaces. A configuration in one space cannot be manipulated by any sequence of moves into a configuration of the other space.

The standard problem is — given some initial configuration, * how can we push the tiles around to reach the standard goal configuration? In Fig. 6.2, we see a possible initial configuration and its state description as given by a 16-tuple.

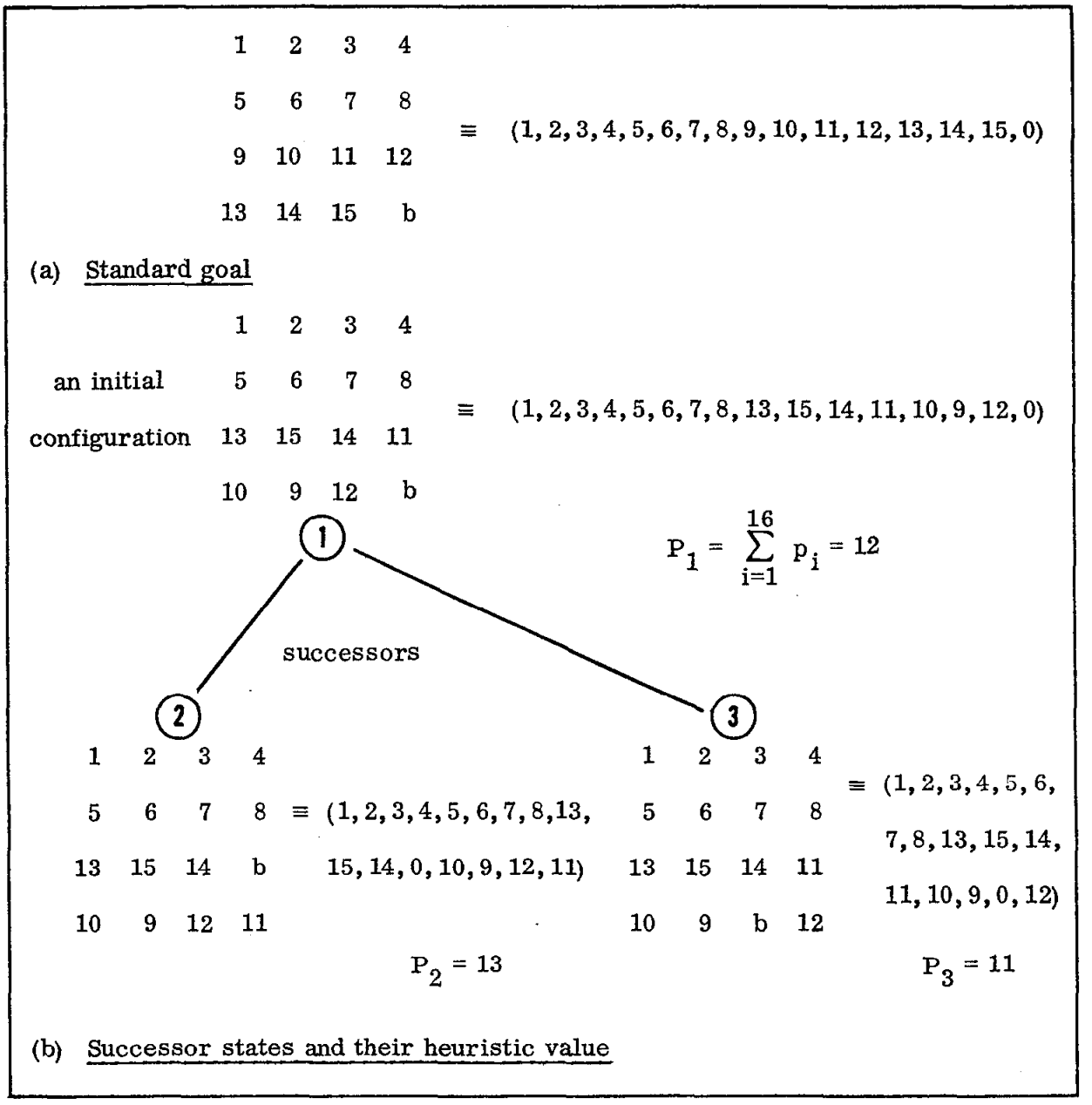


FIG. 6.2--Some puzzle configurations and descriptions.

*In the appropriate parity space.

From the initial configuration, there are two successor states possible. These correspond to switching any tile adjacent to the blank into the blank's position. The position of the blank in the center of the board allows four possible moves, on the sides three possible moves and in the corners two possible moves.

In applying HPA to our problem, we ordinarily attempt to find a good heuristic function h. If for example we chose h=0, then we have an exhaustive search which will ordinarily require too much time and space. Now one simple heuristic measure[18] is a position count. We have a 16-tuple of tile values which are out of order. Any particular tile is so many squares away from its position in the goal configuration.

We can say, as in the Graph Traverser[18] work, that

p_i = the Manhattan distance of the tile in position i from its goal position

$$P = \sum_{i=1}^{16} p_i \quad .$$

Note, if P = 0 then we are finished; also P represents a lower bound on how many moves to the goal. In Fig. 6.2(b) we show the initial configuration with its two successor states and their position count. The first step of HPA would place nodes 2 and 3 (number in circles) in set $\tilde{S}$ and then node 3 would be placed in set S because it has the smaller value. This process continues until the goal state is reached or the computational resources allotted to the problem are exhausted.

In Fig. 3, we show the entire search tree that HPA would visit in solving this problem. While this particular instance is simple enough so that HPA is never misled, it still presents the flavor of heuristic search as thought of in our model.

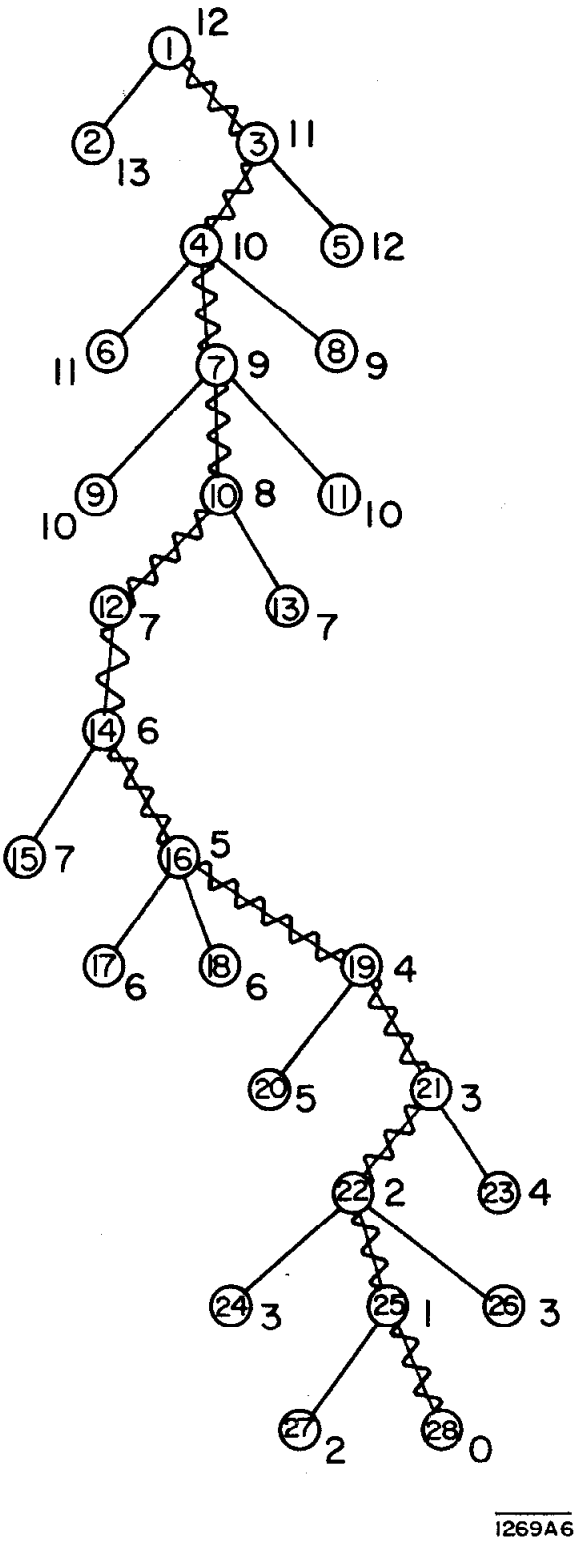


FIG. 6.3--An example of a puzzle solution. How HPA with h=P and $\omega = \infty$ solves a particular 15 puzzle.

FIG. 6.3 (cont.)

Node	State	P
1	(1,2,3,4,5,6,7,8,13,15,14,11,10,9,12,0)	12
2	(1,2,3,4,5,6,7,8,13,15,14,0,10,9,12,11)	13
3	(1,2,3,4,5,6,7,8,13,15,14,11,10,9,0,12)	11
4	(1,2,3,4,5,6,7,8,13,15,0,11,10,9,14,12)	10
5	(1,2,3,4,5,6,7,8,13,15,14,11,10,0,9,12)	12
6	(1,2,3,4,5,6,0,8,13,15,7,11,10,9,14,12)	11
7	(1,2,3,4,5,6,7,8,13,0,15,11,10,9,14,12)	9
8	(1,2,3,4,5,6,7,8,13,15,11,0,10,9,14,12)	9
9	(1,2,3,4,5,0,7,8,13,6,15,11,10,9,14,12)	10
10	(1,2,3,4,5,6,7,8,13,9,15,11,10,0,14,12)	8
11	(1,2,3,4,5,6,7,8,0,13,15,11,10,9,14,12)	10
12	(1,2,3,4,5,6,7,8,13,9,15,11,0,10,14,12)	7
13	(1,2,3,4,5,6,7,8,13,9,15,11,10,14,0,12)	7
14	(1,2,3,4,5,6,7,8,0,9,15,11,13,10,14,12)	6
15	(1,2,3,4,0,6,7,8,5,9,15,11,13,10,14,12)	7
16	(1,2,3,4,5,6,7,8,9,0,15,11,13,10,14,12)	5
17	(1,2,3,4,5,0,7,8,9,6,15,11,13,10,14,12)	6
18	(1,2,3,4,5,6,7,8,9,15,0,11,13,10,14,12)	6
19	(1,2,3,4,5,6,7,8,9,10,15,11,13,0,14,12)	4
20	(1,2,3,4,5,6,7,8,9,10,15,11,0,13,14,12)	5
21	(1,2,3,4,5,6,7,8,9,10,15,11,13,14,0,12)	3
22	(1,2,3,4,5,6,7,8,9,10,0,11,13,14,15,12)	2
23	(1,2,3,4,5,6,7,8,9,10,15,11,13,14,12,0)	4
24	(1,2,3,4,5,6,0,8,9,10,7,11,13,14,15,12)	3
25	(1,2,3,4,5,6,7,8,9,10,11,0,13,14,15,12)	1
26	(1,2,3,4,5,6,7,8,9,0,10,11,13,14,15,12)	3
27	(1,2,3,4,5,6,7,0,9,10,11,8,13,14,15,12)	2
28	(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0)	0

The search was conducted using $f(x) = h(x)$ - pure heuristic search. However, if $f(x) = g(x) + h(x)$ was used, the exact same search would have occurred. The nature of efficient heuristic search is clearly visible in this type of problem environment, and leads to questions of how to appropriately use the heuristic information.

CHAPTER 7

THEORY OF UNI-DIRECTIONAL HEURISTIC SEARCH

HPA is a typical path finding algorithm and is similar to the algorithms used in the work of Michie and Doran, and Hart, Nilsson and Raphael. It will find a path if one exists and the graph is finite, and can fail if the graph is infinite. In the Graph Traverser[18] only h is used, by our convention $\omega = \infty$. The intuitive reason for this weighting is that prior distance in reaching a node is so much "water over the dam." Indeed, if h is an accurate estimator of distance from the goal, it will indicate the node nearest the goal. This remaining distance is what determines the fewest nodes to visit. This argument is plausible, but relies on the accuracy of the heuristic function. Any space for which an accurate estimator exists is a solved problem domain. Only domains with inaccurate estimators are interesting, and it is these cases for which the efficient use of heuristic information is necessary. In table 7.1 we list some common weights and the type of search produced.

Table 7.1

Commonly Used Evaluators

$\omega = 0$,	$f(n) = g(n)$	exhaustive parallel or breadth first search
$\omega = \infty$,	$f(n) = h(n)$	simple or pure heuristic search — Graph Traverser
$\omega = 1$,	$f(n) = g(n) + h(n)$	compound heuristic search

7.1 Some Theorems on Searching

In examining formally the claims of the above argument two extremes are easily dealt with. First, we could have a heuristic function which always returned the exact distance to the goal, a function having this property we call perfect.

Secondly, we could have a heuristic function which is completely in error; this would be the inverse of the perfect function.

Theorem 7.1:

If h is perfect (exact, correct) then for $\omega \geq 1$, the search by HPA is optimal, i.e., visits the fewest nodes possible.

Proof

Case 1. $\omega = \infty$.

Let the shortest path be k steps long. $\mu = (s, x_1, \ldots, x_k)$, $x_k = t$. Since h is perfect then $h(x_i) = k - i$. Now when s is expanded $x_1 \in \Gamma(s)$ and since other nodes are off the shortest path they must have an h value greater than k - 1. So x_1 is picked on the next iteration, and is expanded in turn. At each iteration the node along the shortest path and currently in $\tilde{S}$ is placed in S. Therefore only nodes on the shortest path are expanded, and so our method is optimal.

Case 2. $\omega = 1$.

The argument is the same as above, except that along μ, $f(x_i) = g(x_i) + h(x_i)$ $= i + k - i = k$. So along the shortest path, all nodes evaluate to k, and other nodes evaluate to greater than k.

Case 3. $1 < \omega < \infty$.

Consider $f^*(n) = \frac{f(n)}{\omega} = \frac{g(n)}{\omega} + h(n)$. Each step from s adds $1/\omega$ from the first term, and along the shortest path the second term is reduced by 1. Now $1/\omega < 1$, so along shortest path f^* decreases with $f^*(x_j) = k - j + j/\omega$. Each node along μ decreases in value by $1 - 1/\omega$ while nodes off μ increase by at least $1/\omega$. Thus f^* will expand only the nodes on μ, and so is optimal. Now f^* determines the same search order as f, so f is optimal. ■

We see that for h perfect and for $\omega \geq 1$ HPA only expands nodes on the shortest path. If $\omega < 1$, then additional nodes may be expanded, with $\omega = 0$ the worst case.

This case is the exhaustive parallel search (see table 7.1). However, the key point of this result is that using g(x) in our evaluator does not decrease the efficiency of search, when appropriately weighted. This is already in some measure refutes the "common sense" arguments of the pure heuristic searchers.

Theorem 7.2:

If h* is the inverse of the perfect heuristic function h, then the search by HPA using pure heuristic search ($\omega = \infty$) will always visit the goal node last. If the space is infinite, the goal will never be found. Therefore $\omega = 0$ gives the best search under these conditions.

Proof

Since we are using the reciprocal of the perfect function, the further from the goal node the smaller h*. So HPA ($\omega = \infty$) will be led away from the goal, and only if it exhausts the rest of the space will it reach the goal. It is obvious that the larger ω, the more misleading the evaluator

$$f = g + \omega \cdot h^* .$$

$\therefore$ using $\omega = 0$, HPA visits the fewest nodes. ■

So in the case where the heuristic function is counterproductive, the less we rely on it the better. It now remains to investigate cases where h is somewhere between these extremes in its accuracy.

7.2 Heuristic Error

To do this rigorously will require easily analyzable spaces. However, as will become clear, this should shed much light on the use of heuristic search, where previously only heated "intuitively" justifiable arguments were used. The spaces used will be regular infinite rooted trees with unique goal nodes. The root is the only node without predecessors, and a regular tree is one in which each node has the same number of descendants.

The simplest such space is the unary tree (Fig. 7.1(a)). Over this space all functions, representing any heuristic function and weighting, are equivalent. The search always proceeds from node 1 to node 2, ... until the goal node is encountered. This case is without interest and we move on to the binary tree space (Fig. 7.1(b)). This is already non-trivial and complex enough to represent reasonable problem domains such as Lisp programs.[36]

Theorem 7.1 applies regardless of the specific directed graph structure, thus the use of a perfect h in our evaluator f is optimal for $1 \leq \omega < \infty$. Perhaps no heuristic information exists for our domain, and we therefore have h identically zero throughout the binary tree space. The evaluators we could use are then:

(a) $f = g + \omega \cdot h = g$ $\quad (h = 0) \quad 0 < \omega < \infty$

(b) $f = h = 0$ $\quad \omega = \infty$

The use of g constitutes a parallel search, while the use of 0 is a search where all the nodes in $\tilde{S}$ (open nodes) will be tied. At each iteration, step 2 of HPA will randomly choose from the nodes tied, and therefore (b) produces a random search. If instead our tie-break rule was first-in/first-out (FIFO) we would have parallel search. Last-in/first-out (LIFO) would be a depth first rule.

Theorem 7.3:

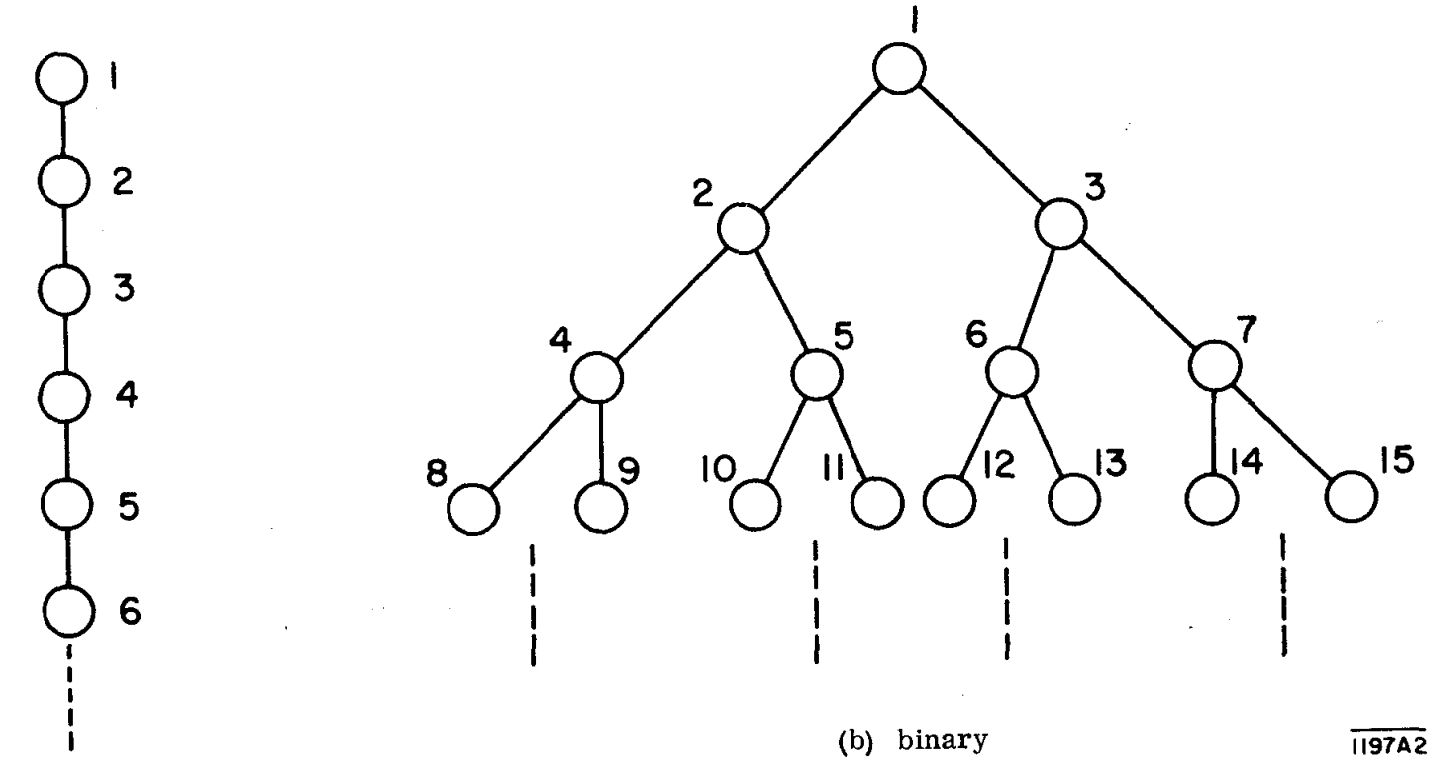
Over an infinite binary tree, a parallel search on the average requires $2^k + 2^{k-1} - 1/2$ nodes expanded to find a node k steps from the root.

Proof

The number of nodes in a binary tree of diameter k (maximum length from the root) is $B_k = 2^{k+1} - 1$. Since a node may be anywhere along the kth level with equal probability, we must search $B_k + 1$ to B_{k+1} nodes with the average being

$$\frac{1}{2}\left(B_k + 1 + B_{k+1}\right) = \frac{1}{2}\left(2^k + 2^{k+1} - 1\right) = 2^k + 2^{k-1} - \frac{1}{2}$$

Lemma: A binary tree of diameter k has $2^{k+1} - 1$ nodes.



(a) unary

(b) binary

FIG. 7.1 --Regular infinite trees.

Proof: By induction.

Case k = 0: B_0 is just the root node. $B_0 = 2^{0+1} - 1 = 2 - 1 = 1$.

Case k = 1: B_1 is just the root node and two successors. $B_1 = 3 = 2^{1+1} - 1$.

Inductive step: Assume $B_k = 2^{k+1} - 1$, to show $B_{k+1} = 2^{k+2} - 1$.

Each level has 2^k nodes, where k is the distance from the root.

$$B_{k+1} = B_k + (k + 1 \text{ level})$$

$$= B_k + 2^{k+1} = 2^{k+1} + 2^{k+1} - 1 = 2^{k+2} - 1. \quad \blacksquare$$

So in a parallel search of a binary tree, we have the above formula determining, on average, how many nodes must be visited. It is exponentially varying with the distance from the root; typical behavior in complex problem spaces.

In contrast, let us examine the expected number of nodes visited by a random search in finding a goal node k steps from the root. In the simplest non-trivial case k equals 1 (k = 0 is trivial).

Theorem 7.4:

Over an infinite binary tree, a random search expects to visit an unbounded number of nodes to find a goal node 1 step from the root.

Note: HPA is not told that the node is only 1 away and consequently does not restrict its search to this level.

Proof

E = Expected number of nodes visited

r_i = Probability of finding the goal node in exactly i steps

p_i = Probability of finding the goal node on the ith step, having reached this step

ℓ_i = Probability of not finding the goal node on any step before the ith step

$$E = 1 + \sum_{i=1}^{\infty} i \cdot r_i .$$

The 1 is for the root node

$$r_i = p_i \cdot \ell_i .$$

We show

$$p_i = \frac{1}{i+1} .$$

With each step of HPA over a binary tree one node is removed from $\tilde{S}$ and placed in S, but two nodes are placed in $\tilde{S}$. This means at step i there are i + 1 nodes in $\tilde{S}$. In the case of f = 0 and random tie-breaking, they all are equally likely to be picked. Furthermore since the goal node is 1 away from the root and it is always in set $\tilde{S}$ until found by HPA so $p_i = \frac{1}{i+1}$.

$\ell_i = \frac{1}{i}$, we show this by induction

$$\ell_i = 1 - \sum_{j=1}^{i-1} r_j , \quad \ell_1 = 1$$

$$\ell_2 = 1 - r_1 = 1 - p_1 \ell_1 = 1 - \frac{1}{2} \cdot 1 = \frac{1}{2} .$$

Assume $\ell_k = 1/k$, we must show $\ell_{k+1} = 1/k+1$

$$\ell_{k+1} = 1 - \sum_{j=1}^{k} r_j = 1 - \sum_{j=1}^{k-1} r_j - r_k$$

$$= \ell_k - r_k = \ell_k (1-p_k) = \frac{1}{k} \left(1 - \frac{1}{k+1}\right) = \frac{1}{k+1} .$$

Therefore

$$E = 1 + \sum_{i=1}^{\infty} \left(i \cdot \frac{1}{i+1} \cdot \frac{1}{i} \right) = 1 + \sum_{i=2}^{\infty} \frac{1}{i}$$

$$= \sum_{i=1}^{\infty} \frac{1}{i} \text{ the harmonic series which does not converge.} \quad \blacksquare$$

This result is similar to gambler's ruin problems.[22,50] Essentially the space grows too fast, and when not lucky enough to initially find the goal node, we soon find it disappearing in the growth of $\tilde{S}$.

Normally, a search is restricted by time or space limitations. This is akin to limiting our infinite space to some maximum depth. If we are interested in finding a goal node in a binary tree of diameter k, then a maximum of $2^{k+1} - 1$ nodes need be searched. If each of these nodes is with equal probability the goal node, then any exhaustive non-repeating search would yield the same expected value for nodes visited $\frac{1}{2}(B_k + B_0)$, or 2^k. Each method would get better performance for different groups of nodes. The parallel search visits the closer nodes soonest, and for goal nodes near the root this method has a better expected value than random or depth first (LIFO) search.

In our theorem, HPA was unaware that the goal was at level 1, and so with finite probability it searched portions of the space which were beyond the solution. A modification on this would be to tell our procedure that the goal was on level k. When this is known, the depth first (see Fig. 7.2) or backtrack track method[24,26] is optimal. The algorithm should go down to a depth of k and check to see if this is the goal node. If not it backs up one level and goes down to the next node at level k. It continues backing up and going forward to the next node on level k until it finds the goal. Since this search pattern looks at nodes on the kth level

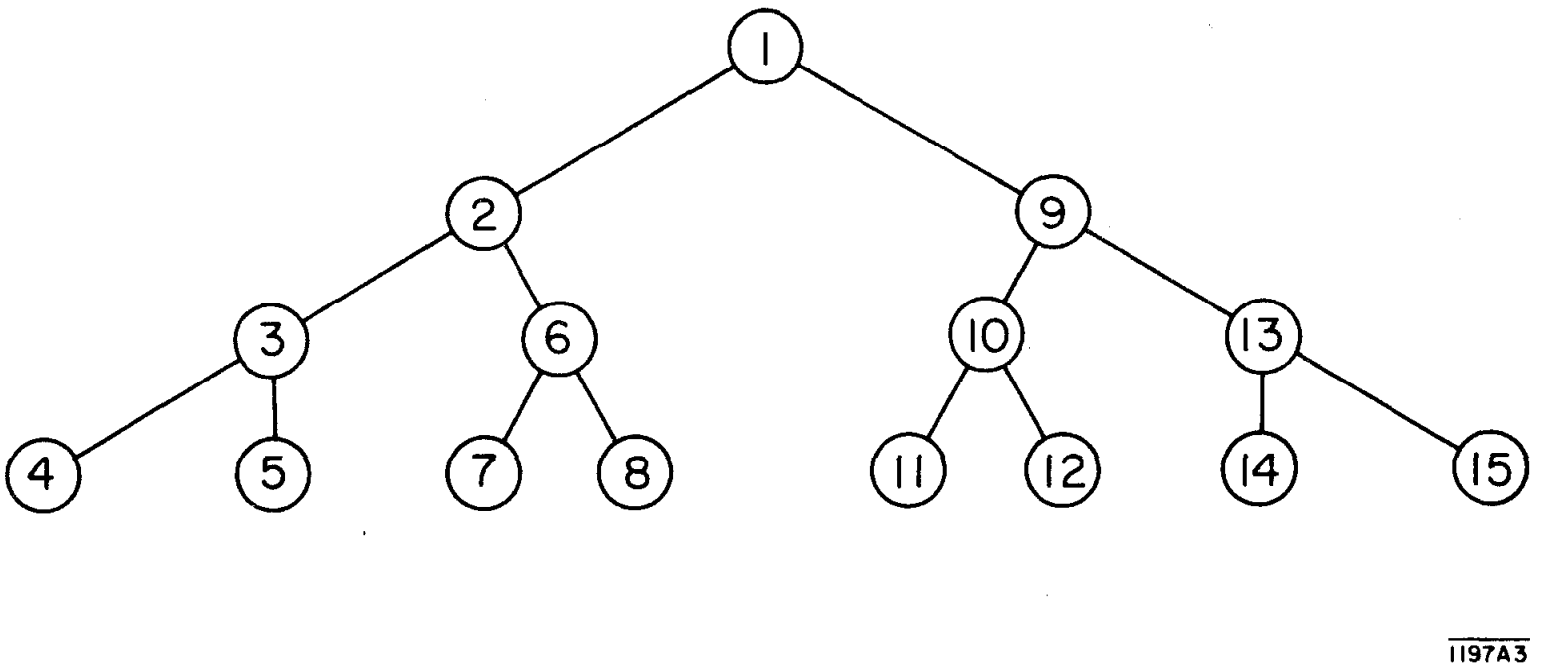


FIG. 7.2 --Nodes numbered in order visited by a depth first search to level 3.

as soon as possible, it must be best in the sense of the expected number of nodes visited. It would be the worst search pattern if the goal node was actually at level 1. Here it either finds the goal on the first try (like any other method) or must look at half the tree before returning to the goal node. A parallel search is a conservative strategy, you are guaranteed not to penetrate below the part of the tree containing the goal, while you pay by always investigating the whole subtree up to that level.

7.3 How Error Affects Heuristic Search

In general we have neither a complete lack of information nor perfectly accurate information, but instead we have a heuristic function which has error. We wish to resolve for this more typical instance how best to use a heuristic function. To investigate this question, we stay in our binary tree space using HPA. We will do a worst case analysis in the spirit of error analysis in numerical problems.

Consider

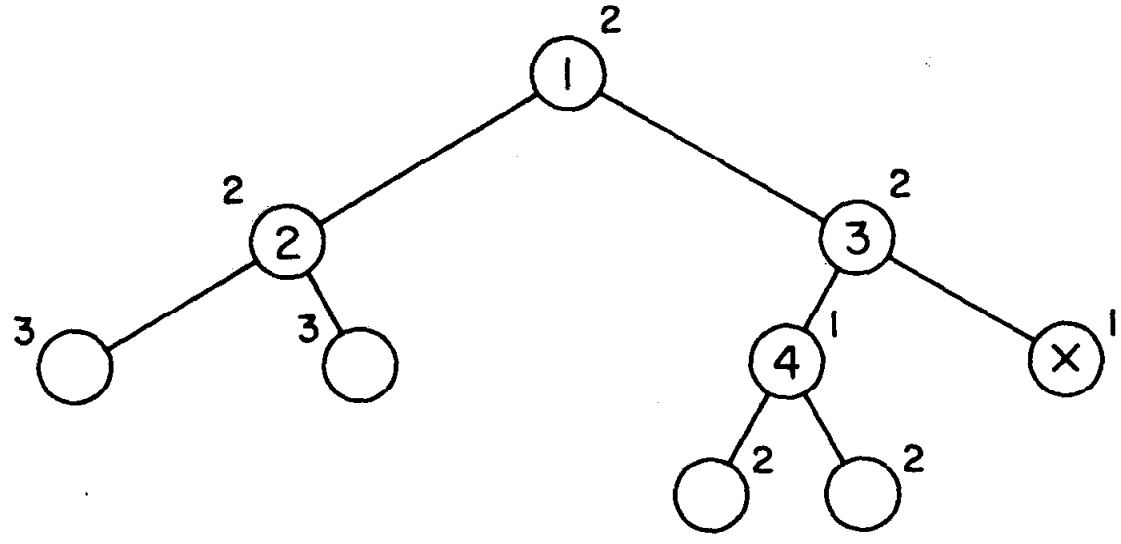
$$h = \text{perfect estimator}$$

$$\epsilon = \text{a bound on the error} \quad 0, 1, 2, 3, \ldots$$

$$h^* = \text{actual heuristic function}$$

$$h - \epsilon \leq h^* \leq h + \epsilon$$

We will choose values of h^* conforming to the above limits, but in such a manner as to mislead HPA to the greatest extent. In doing this, we assume that HPA will always choose the worst nodes in case of ties, i.e., nodes off the solution path. An example of this analysis is Fig. 7.3, where HPA just uses the h^* function as the evaluator. The order of search is according to the numbers inside the nodes with x, the goal being reached in 5 steps. To make h^* as bad as possible ($\epsilon = 1$), we add ϵ to each node on the shortest path, and



$\epsilon = 1$

$f = h^*$

1197A4

FIG. 7.3--The goal node is marked by an x. Other nodes are labeled by order of search (inside) and f value outside. Five nodes are searched when x is found.

we subtract ϵ from each node off the shortest path. If h itself was used HPA would only visit the 3 nodes on the shortest path which is a consequence of theorem 7.1.

One of the principal questions is the comparison between h^* and $g + h^*$ as evaluators. Both to get more of a flavor of our error analysis and some inklings as to this comparison, we work through the example of Fig. 7.4. Let us examine HPA using $f = h^* + g$, as in Fig. 7.4(b). At the goal node x,

$$h(x) = 0, \qquad g(x) = 1$$
$$h^*(x) = h(x) + \epsilon = 0 + 2 = 2$$
$$f(x) = 3;$$

while at node 2 we have

$$h(2) = 2, \qquad g(2) = 1$$
$$h^*(2) = h(2) - \epsilon = 2 - 2 = 0$$
$$f(2) = 1.$$

Node 3 has

$$h(3) = 3, \qquad g(3) = 2$$

and so both have increased by 1 from the values of its predecessor node 2.

Thus

$$h^*(3) = 3 - 2 = 1$$
$$f(3) = 3$$

an increase of 2 from its predecessor. In contrast when using only h^* (Fig. 7.4(a)), f increases by 1. This allows the search in Fig. 7.4(b) to cut-off sooner along an incorrect path. The results of Fig. 7.4 are:

distance to goal = 1

maximum error $\epsilon = 2$

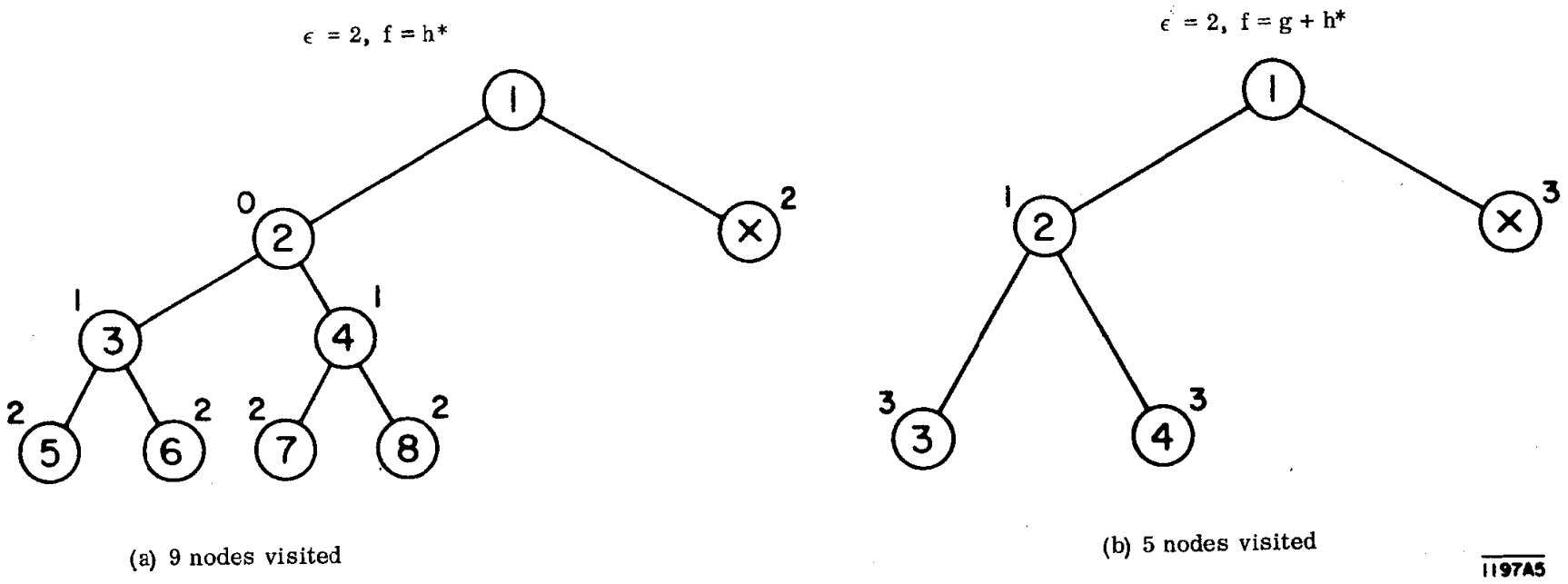


FIG. 7.4--Comparison between two possible ω's for h*:
(a) $\omega = \infty$ (b) $\omega = 1$

nodes visited $f = h^*$ are 9

nodes visited $f = h^* + g$ are 5

We can generalize this result and find a formula giving the number of nodes visited for different errors and path lengths in our binary tree space.

In analyzing the worst case behavior, we must show that $h^* = h + \epsilon$ on the solution path and $h^* = h - \epsilon$ off the shortest path leads to the poorest searches.

Theorem 7.5:

If $h_1 = h_2$ except on the solution path where

$$h_1(x) \geq h_2(x), \qquad x \text{ on solution path}$$

Then the search by HPA using h_1 always visits all the nodes visited when using h_2.

Proof

Let $\mu = (x_1, x_2, \ldots, x_k)$ be the solution path.

$S(x_i)$ will be the tree explored by HPA, when x_i is included in set S. So $S(x_k)$ is the set of nodes searched when HPA finds the solution path.

Let $S_1(x_i)$ be the trees searched by HPA using h_1 and $S_2(x_i)$ will be the corresponding trees for h_2. We show by induction that

$$S_1(x_i) \supseteq S_2(x_i)$$

a) $S_1(x_1) \supseteq S_2(x_1)$

Since $h_1(x_1) \geq h_2(x_1)$ and nodes off the solution path have the same values.

b) Assume $S_1(x_i) \supseteq S_2(x_i)$, then $S_1(x_{i+1}) \supseteq S_2(x_{i+1})$.

This is obvious from the same argument as in case (a).

Therefore increasing the value of the heuristic function along the solution path can only increase the number of nodes searched. ■

Theorem 7.6:

If $h_1 = h_2$ on the solution path, but everywhere else

$$h_1(x) \leq h_2(x), \qquad x \text{ off the solution path}$$

Then the search by HPA using h_1 always visits all the nodes visited when using h_2.

Proof

The argument is similar to theorem 7.5. Now the reason more nodes may be visited using h_1 is that the values off the solution path are lower using h_1 and hence sooner included in set S. ■

Taken together the above theorems show that a worst case search occurs when $h+\epsilon$ is used on the solution path and $h-\epsilon$ is used off the solution path. These results extend to using any particular value of ω in making the corresponding f_1 and f_2

Theorem 7.7:

Let k be the distance from the root node to the goal node and $f = g + h^*$ be the function used by HPA, then the maximum number of nodes visited in our binary tree space is

$$2^{\epsilon} \cdot k + 1 \ .$$

Proof

If the goal node is distance k from the root, then if h^* is perfect HPA visits $k + 1$ nodes (theorem 7.1). In the worst case with an error of ϵ, all nodes ϵ off the shortest path will be visited, excluding the nodes succeeding the goal node. This is shown in our discussion of Fig. 7.4.

Case $\epsilon = 0$.

This is as stated above $k + 1$.

Case $\epsilon = 1$.

These are the nodes on the shortest path plus those one off the shortest path. There are k nodes one off the shortest path so we have

$$k + k + 1 = 2k + 1 .$$

Case $\epsilon \geq 2$.

At this point each unexpanded node (leaf) has two not yet explored successors. The number of leaves in a binary tree grows as $2^{\epsilon-1} \cdot k$. So the tree for error $\epsilon \geq 2$ is

$$\underbrace{2k + 1}_{\epsilon=1} + 2 \cdot k + 2^2 k + \ldots + 2^{\epsilon-1} k$$

$$= 1 + 2k + k \sum_{i=1}^{\epsilon-1} 2^i$$

$$= 1 + 2^{\epsilon} \cdot k$$

Similarly we prove

Theorem 7.8:

Let k be the distance from the root node to the goal node and $f = h^*$ be the function used by HPA, then the maximum number of nodes visited in our binary tree space is

$$\frac{4^{\epsilon}}{2} \cdot k + 1 \qquad \epsilon \geq 1$$

$$k + 1 \qquad \epsilon = 0$$

Proof

Case $\epsilon = 0$.

Again by theorem 7.1 a path to a goal node distance k from the root is found by HPA visiting k + 1 nodes, if h^* is perfect.

Case $\epsilon = 1$.

Here as in the previous theorem HPA visits nodes 1 off the solution path and we have

$$2k + 1 = \frac{4^1}{2} k + 1 \qquad \text{nodes visited.}$$

Case $\epsilon \geq 2$.

After the first level each increment in the error allows a maximum of two additional levels to be visited. This is because along the solution path we use $h + \epsilon$ and off the solution path we use $h - \epsilon$ giving a 2ϵ leeway. The trees with the maximum number of explored nodes are

$$\underbrace{2k + 1}_{\epsilon=1} + \underbrace{2k + 2^2 k}_{\epsilon=2} + \underbrace{2^3 k + 2^4 k}_{\epsilon=3} + \ldots$$

$$+ \underbrace{2^{2\epsilon-3} k + 2^{2\epsilon-2} k}_{\epsilon}$$

$$= 1 + 2k + k \sum_{i=1}^{2\epsilon-2} 2^i = 1 + \frac{4^\epsilon}{2} \cdot k$$

These results suggest two plausible conclusions for general heuristic search.

1. The more accurate h^*, the fewer nodes visited by HPA.
2. It is better to include g in the evaluator.

Furthermore the results are extendable to any tree structured problem space with a unique goal node of interest. Namely

Theorem 7.9:

If HPA is searching any tree structured space for some goal node then

a) $f = h^*$ will visit at least as many nodes $f = g + h^*$ in the sense of the above worst case analysis.

(b) If $h - \epsilon_1 \leq h_1^* \leq h + \epsilon_1$ and $h - \epsilon_2 \leq h_2^* \leq h + \epsilon_2$, $\epsilon_2 > \epsilon_1$. Then the number of nodes visited by HPA using h_2^* will be at least as many as when using h_1^* in the sense of the worst case and with ω being the same for both evaluators.

Proof

Part (b) is obvious.

Part (a) follows from theorems 4 and 5. ■

The major defect of the above analysis and consequently the generality of the results is that problem spaces are ordinarily not trees. In a tree each node has a unique path back to the root. Problem domains have circuits normally and many alternate solution paths. Also the above analysis is for the worst case and while these results are attainable, in practice they are unlikely. It is important to monitor the behavior of HPA in actual problems.

CHAPTER 8

SOME UNI-DIRECTIONAL EXPERIMENTS WITH THE FIFTEEN PUZZLE

Each problem space and each heuristic function for this space presents a problem in selecting an appropriate ω. Exclusive use of g guarantees finding the shortest solution path, however, a price is paid in the breadth of the search. On the other hand, using only h is possibly unstable; HPA then runs down the search tree to great depths before changing its search to another part of the space. This behavior is analogous to the situation described by theorem 7.4. It is appropriate to look for a middle ground between the pitfalls of these extremes.

Let us consider a specific heuristic function, h*, used by HPA to solve problems in some problem space. We can characterize its effectiveness for a given problem by the number of nodes N it visits in finding a solution path of length K. A further number describing its search is its branch rate.† The branch rate ρ is the number of successors a node has in a regular tree of diameter K and size N. So given a particular heuristic function and a weighted evaluator, we solve a number of sample problems in our space, obtaining for each solution

N_ω = nodes searched for this weighting

K_ω = solution path length.

We can then determine

ρ_ω = branch rate for this weighting

by solving for ρ_ω in

$$N_\omega = \frac{\rho_\omega}{\rho_\omega - 1}\left(\rho_\omega^{K_\omega} - 1\right)$$

†This is a suggestion of Nils Nilsson.

We then use this information to select the most successful value of ω for our evaluation function.

The fifteen puzzle represents a general problem domain in having many possible alternate solution paths. It is therefore reasonable to study the behavior of HPA using different heuristic functions with respect to some sample of problems.

8.1 Heuristic Functions

In the previous chapter, we stepped through an example of a typical fifteen puzzle problem using a function P referred to as position count. This function has the lower bound property [30] with respect to the actual metric on the space, and therefore

$$\text{a)} \qquad f = g + \omega P \qquad\qquad 0 \leq \omega \leq 1$$

is admissible.

In the Graph Traverser experiments, Doran and Michie developed a more sophisticated function which had two separate terms.

$$\text{b)} \qquad S = \sum_{i=1}^{16} h_i^{\alpha} \, p_i^{\beta}$$

p_i is the position value as in P; h_i is the Manhattan distance from the blank square to tile i. The experiments with the Graph Traverser found $\alpha = 0.5$, $\beta = 2$ to be best. The addition of h_i into our evaluation ($\alpha = 0$, $\beta = 1$ makes S = P) adds the fact that a tile may not be moved unless it is adjacent to the blank position. Consequently rearrangement is harder the more distance between tile and blank (see Fig. 8.1). S still was found to be impractical by Doran and Michie and they included the 'ad hoc' (their terminology) reversal term. The reversal term is not precisely defined in Ref. 18. The interpretation given it in our experiments

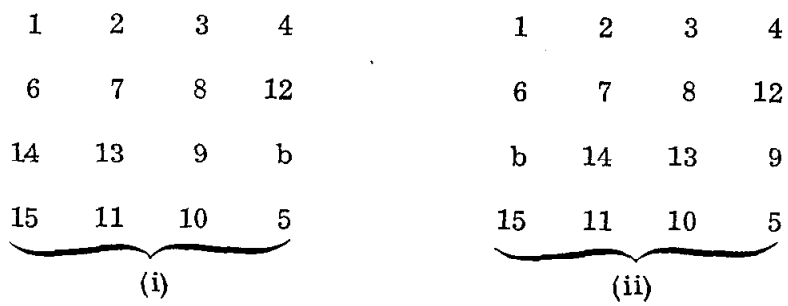


FIG. 8.1--Tile 5 is easier to move in (i) because it is next to the blank.

is described by the following Algol segment.

```
R := 0 ;
for i := 0 step 4 until 12 do
    for j := 1 step 1 until 3 do
        if board [i + j] = board [i + j + 1] + 1
            and board [i + j] = i + j + 1 then R := R + 1;
for i := 1 step 1 until 4 do
    for j := 0 step 4 until 12 do
        if board [i + j] = board [i + j + 1] + 4
            and board [i + j] = i + j + 4 then R := R + 1;
```

The reversal count R as defined above is incremented whenever two adjacent tiles are interchanged from the goal position. The reversal term used by the Graph Traverser* differs from ours in relaxing the requirement that the tiles be in their goal positions. Instead, they may be in any adjacent positions within their goal column or row. Since our purpose is to test the effect of ω weightings on heuristic search it is unnecessary to have the same heuristic functions as the Graph Traverser. In fact the function developed using the Graph Traverser is too good to be interesting. A function so reliable behaves well over a wide range of ω values as expected from theorem 7.1. This situation is unrealistic in

*Private communication with Jim Doran.

difficult problem domains, and heuristic functions with significant error but of positive benefit are more interesting.

The above terms were combined into four different functions.

1. $f_1 = g + \omega \cdot P$

2. $f_2 = g + \omega \cdot (P + 20 \cdot R)$

3. $f_3 = g + \omega \cdot S$

4. $f_4 = g + \omega \cdot (S + 20 \cdot R)$

The reversal term was weighted by 20 because this is approximately the number of moves times reversals it takes to solve a position whose only defect is a pair of reversals. The functions f_1 and f_2 constitute one pair of related functions and f_3 and f_4 are another pair. The basic term of the first pair P is a naive heuristic in comparison to S, the basic term in the second pair.

8.2 Data

Figure 8.2 shows the ten positions used in our experiment. They cover a wide range of difficulty and special features. A1 and A6 have their top two rows already in order. However A6 has many reversals. A9 is almost in reverse order, so the individual tiles are quite far from their goal squares. A8 is the configuration appearing in Ref. 18 and A10 is a puzzle used by Ref. 56. The rest of the positions are randomly selected.

8.3 Experiment

An Algol W program (Appendix III) incorporating the HPA procedure was run with positions A1 - A10. Functions f_1 and f_2 were run with $\omega = 1, 2, 3, 4, 8, 16, \infty$. Functions f_3 and f_4 were run with $\omega = 0.5, 0.75, 1, 1.5, 2, 3, 4, 16, \infty$. For each case the solution path length and the number of nodes expanded were recorded. If the number of nodes expanded reached 1000 the search was terminated without a solution. This then was an arbitrary limit selected as being the highest price we would pay for a solution.

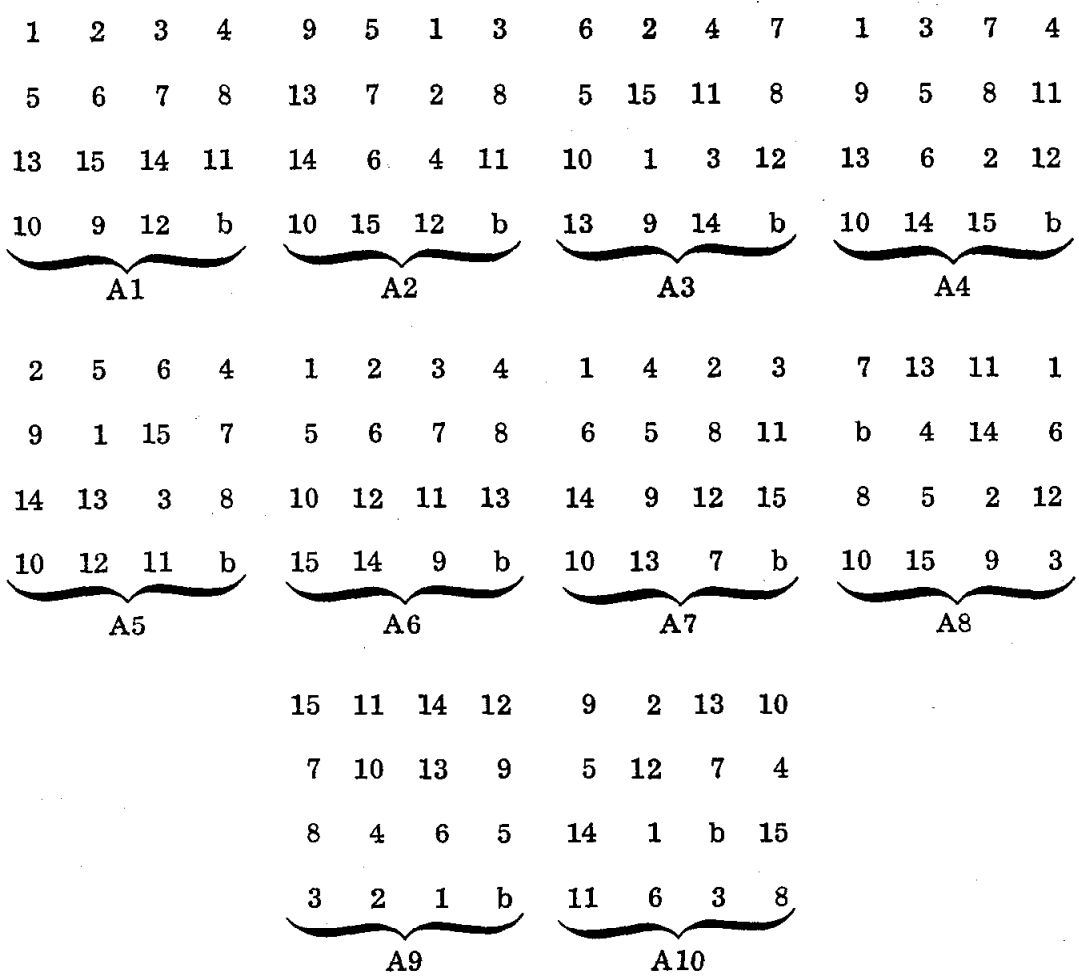


FIG. 8.2--Initial positions and code numbers used in experiments.

The different values of ω tested for f_1 and f_2 versus f_3 and f_4 come from normalizing h with respect to g. P normally underestimates the actual distance to the goal, while S generally overestimates the actual distance to the goal. In making these calculations for position A1 we have P=12 and S=47 where the actual distance is 12. To have equal importance in the evaluator, the h term must be normalized or scaled to the g values. Since S is an overestimator, we use smaller values of ω in our tests with it.

8.4 Results

The performance of the different heuristic functions varied significantly with ω. Tables 8.1 - 8.4 show the results for the four functions and ten positions. All of these functions were of positive benefit in pruning the search space. In conducting a parallel search (f=g) with position A1, the search was terminated when 1000 nodes were expanded. The tree depth was 9 which is 3 away from the solution. This, being the simplest puzzle, shows that exhaustive search could not within the 1000 node limitation solve any of our problems.

Function f_4 was the most powerful evaluator, followed in order by f_2, f_1 and f_3. f_4 solved problems most consistently and with the fewest nodes expanded. The P functions f_1 and f_2 were much nearer in performance than the S functions f_3 and f_4. The reversal term was much more significant in improving the S functions than the P function.

Function f_2 was better than function f_1 in 37 cases, while f_1 was better than f_2 7 times (this is out of a total of 70 cases). The scorecard for f_4 versus f_3 was 60 to 1 out of 90 cases. Case A1 was not significant in these comparisons and other ties normally occurred because 1000 nodes were reached by both functions. Function f_4 with $\omega = 1.5, 2, 16, \infty$ solved all the problems. Function f_2 with $\omega = 4, 16 \infty$ solved all but one problem. The best performance for f_3 was $\omega = 0.75$ which solved 6 problems; while f_1 with $\omega = 3, 4, 8, \infty$ solved 6 problems. Except for $\omega = 0.5$, f_4 solves at least 9 out of 10 problems for each ω value. It was also best in the sense of visiting fewer nodes, on the average, than f_2 the next best function. For example, comparing f_2 with f_4 with $\omega = \infty$, f_4 was better in 5 problems than f_2 and looks at 2928 nodes; while f_2 is better in 4 problems and looks at 3459 nodes. Comparing the fewest total nodes visited, f_1 was best with $\omega = 3$, f_2 with $\omega = 4$, f_3 with $\omega = 1.5$ and f_4 with $\omega = 1.5$.

Table 8.1

Results for $f_1 = g + \omega \cdot P$

ω	A1		A2		A3		A4		A5		A6		A7		A8		A9		A10		Total N
	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	
1	28	12	129	26	1000	-	120	20	1000	-	1000	-	1000	-	1000	-	1000	-	1000	-	7277
2	12	12	66	28	585	36	69	20	1000	-	1000	-	1000	-	1000	-	801	86	1000	-	6533
3	12	12	35	28	930	48	122	20	1000	-	393	36	317	38	1000	-	1000	-	1000	-	5809
4	12	12	36	28	1000	-	228	20	368	44	447	36	900	42	1000	-	1000	-	1000	-	5991
8	12	12	36	28	1000	-	326	32	516	44	834	40	431	62	1000	-	1000	-	1000	-	6156
16	12	12	36	28	1000	-	418	32	1000	-	1000	-	279	62	1000	-	1000	-	1000	-	6745
∞	12	12	996	86	853	148	119	21	1000	-	1000	-	514	80	1000	-	360	152	1000	-	6854

N = nodes visited

K = solution path length

Table 8.2

Results for $f_2 = g + \omega(P + 20 \cdot R)$

ω	A1		A2		A3		A4		A5		A6		A7		A8		A9		A10		Total N
	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	
1	12	12	66	26	1000	-	106	20	1000	-	1000	-	1000	-	1000	-	1000	-	1000	-	7184
2	12	12	35	28	1000	-	60	20	1000	-	443	32	81	36	1000	-	1000	-	1000	-	5631
3	12	12	36	28	1000	-	84	20	653	46	663	40	97	36	1000	-	313	92	655	64	4513
4	12	12	36	28	705	62	124	20	206	46	317	36	118	48	1000	-	155	92	270	64	2943
8	12	12	36	28	598	66	320	38	189	46	248	48	124	48	240	85	1000	-	1000	-	3762
16	12	12	36	28	1000	-	209	34	323	66	129	58	610	66	245	85	691	120	865	90	4120
∞	12	12	433	104	121	58	119	20	123	46	133	56	138	58	742	203	648	196	1000	-	3469

N = nodes visited

K = solution path length

Table 8.3

Results for $f_3 = g + \omega \cdot S$

ω	A1		A2		A3		A4		A5		A6		A7		A8		A9		A10		Total N
	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	
0.5	13	12	48	26	1000	-	31	20	1000	-	1000	-	1000	-	1000	-	1000	-	1000	-	7092
0.75	12	12	1000	-	286	42	26	20	783	38	401	32	1000	-	1000	-	1000	-	737	74	6245
1	12	12	1000	-	488	40	26	20	1000	-	754	38	1000	-	1000	-	1000	-	511	78	6791
1.5	12	12	1000	-	394	48	26	20	108	46	269	44	1000	-	1000	-	1000	-	1000	-	5809
2	12	12	1000	-	203	48	26	20	1000	-	441	44	1000	-	1000	-	1000	-	1000	-	6682
3	12	12	1000	-	1000	-	24	20	1000	-	250	40	1000	-	1000	-	619	196	1000	-	6905
4	12	12	1000	-	1000	-	24	20	1000	-	438	40	1000	-	1000	-	1000	-	1000	-	7474
16	12	12	1000	-	1000	-	24	20	1000	-	1000	-	1000	-	1000	-	1000	-	1000	-	8036
∞	12	12	1000	-	178	80	24	20	377	106	1000	-	1000	-	1000	-	1000	-	1000	-	6591

N = nodes visited

K = solution path length

Table 8.4

Results for $f_4 = S + \omega \cdot (S + 20 \cdot R)$

	A1		A2		A3		A4		A5		A6		A7		A8		A9		A10		Total N
	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	N	K	
0.5	13	12	48	26	1000	-	31	20	868	38	1000	-	1000	-	1000	-	1000	-	1000	-	6960
0.75	12	12	222	40	78	42	26	20	258	38	282	38	933	38	675	95	1000	-	349	72	3835
1	12	12	396	40	97	42	26	20	203	46	162	38	344	48	882	107	1000	-	288	82	3410
1.5	12	12	230	60	207	52	26	20	98	46	267	46	283	48	552	123	803	140	249	88	2720
2	12	12	215	62	154	72	26	20	202	76	224	48	317	68	613	133	611	140	648	108	3022
3	12	12	188	62	350	72	24	20	163	76	166	48	407	68	992	149	405	176	1000	-	3707
4	12	12	184	62	380	72	24	20	157	76	162	48	493	68	1000	-	649	164	999	140	4055
16	12	12	173	62	415	102	24	20	147	76	162	48	257	102	783	185	552	208	882	122	3407
∞	12	12	161	64	370	102	24	20	155	78	157	50	300	60	332	155	558	226	859	126	2928

N = nodes visited

K = solution path length

Overrelaxation

The initial ρ values of the ten problems (table 8.5) show that P is roughly an underestimate of about 1/2, the shortest solution path. S is an overestimate of about twice the shortest solution path. The results for both the P functions and the S functions are better for ω values that cause an overestimate of the distance. This is similar to theorem 7.1 which said that for $\omega \geq 1$ the perfect heuristic estimator was optimal. In underestimating we suffer from broadening the search; in overestimating this does not occur (see Fig. 8.3). The behavior of search with respect to ω is akin to relaxation in elliptic differential equation methods.

Table 8.5

Initial values of the heuristic functions and the shortest solution paths found.

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	Total
P	12	24	18	14	24	12	20	39	52	32	247
S	32.2	82.2	68.4	39.3	80.2	40.9	51.8	183.6	344	139.8	1062.4
k_{min}	12	26	36	20	38	32	36	85	86	64	435

Dead-ends

The significant effects of reversals on the more depth first heuristic, S, is also noteworthy. In backtrack search, the main efficiency is a result of the immediate abandonment of a dead-end, as soon as it is detected.[26] In a heuristic search application to checkmates, Huberman's worse functions[31] provide examples of dead-end detectors. Similarly to place a large value on reversals produces dead-end behavior in the fifteen puzzle. The more depth first the search, the more important to detect dead-ends. It is the difference between the worst evaluator f_3 and the best evaluator f_4. Dead-end detection serves an analogous purpose to using compound heuristic search as opposed to pure heuristic search.

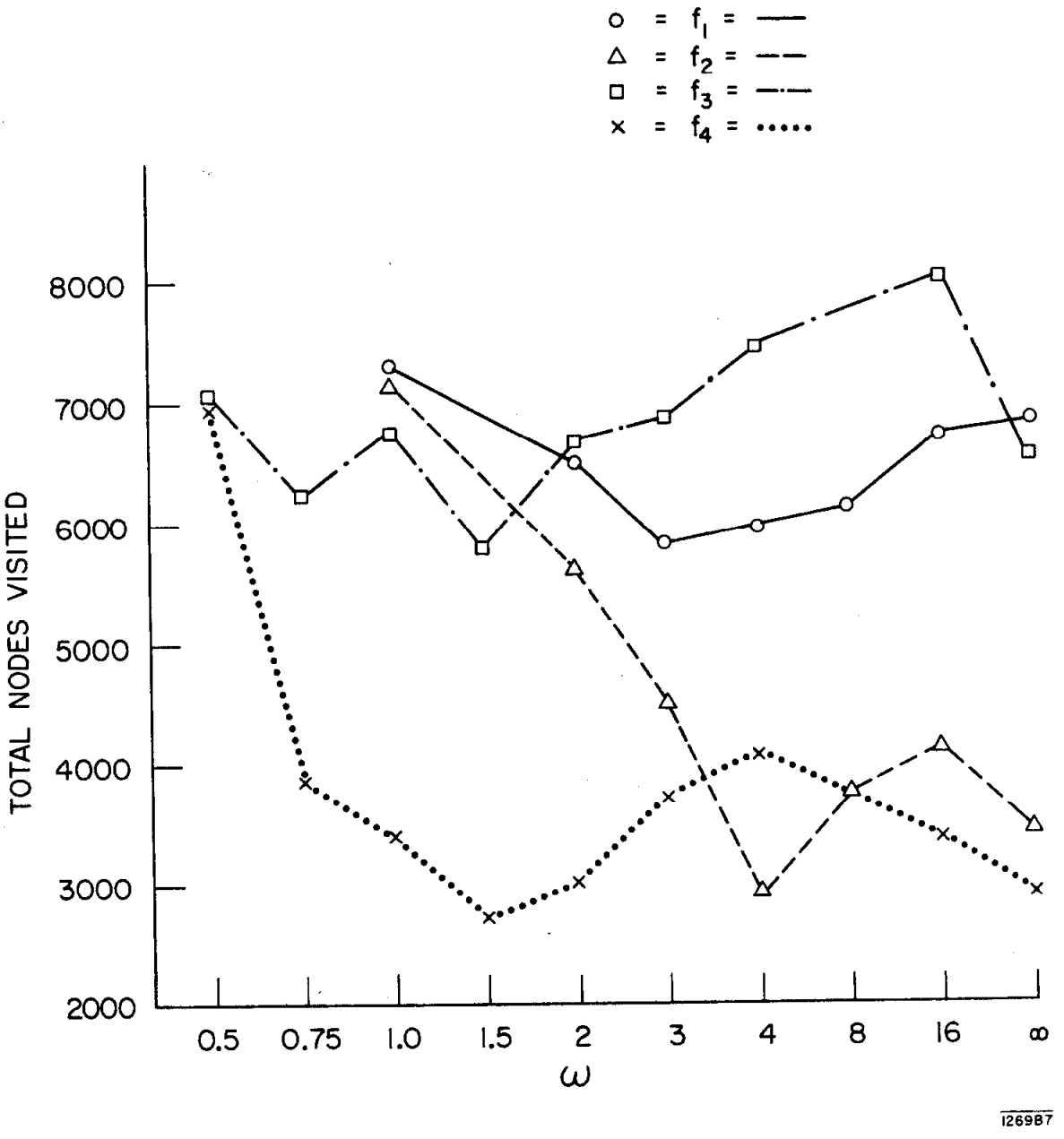


FIG. 8.3--Performance of each evaluator with respect to ω.

In compound search a particular path is no longer searched when its progress does not justify initial expectations. Dead-end detection is a more cathartic form of recognizing that progress is not keeping up to expectations.

8.5 Remarks

In general as ω increased, path length k_ω increased and branching rate ρ_ω decreased (see table 8.6). These results, along with the observations on dead-end detection and overrelaxation are more qualitative than quantitative. It is indicative of the importance of developing this theory in general problem solving domains. Minimally one could say that compound heuristic search is a more general and efficient procedure than pure heuristic search.

Table 8.6

Density vs. Path Length for Function f_4, Problem A5

ω	k_ω	ρ_ω	N
0.5	38	1.129	868
0.75	38	1.083	258
1	46	1.054	203
1.5	46	1.029	98
2	76	1.023	202
3	76	1.018	163
4	76	1.017	157
16	76	1.016	147
∞	78	1.016	155

CHAPTER 9

BI-DIRECTIONAL HEURISTIC SEARCH

After seeing the sizable gains in computational efficiency made by bi-directional methods in the shortest path problem, it is desirable to extend these benefits to the heuristic case. Many problems have a known goal or goal set, and the additional power of the bi-directional technique is a welcome aid in these cases. At first, as in Ref. 30, we want the extension of VGA to the heuristic case preserving admissibility.

Before going on to the extension, we would like to deal with two common objections to the use of bi-directional techniques.

1. They are only useful when both the goal node and the initial node are specified and the graph is symmetric. (This remark is inferred from Ref. 18, p. 257.)

As we have already seen with VGA:

a) We may have a goal set and an initial set rather than single nodes;

b) The graphs may be directed (unsymmetric);

c) If a property specifies the goal, it normally determines a set, which as we note in section 2.4 can be handled.

2. The extra bookkeeping for bi-directional methods, especially the test for a node lying in the intersection of forward and backward trees makes the method cumbersome.

This was not the case in VGA and in the description (to follow) of the program a general method based on hashing, simulates associative search and efficiently finds nodes in the tree intersection along with providing a simple test for redundancy.

9.1 Extension to Bi-directional Heuristic Search

The extension of the class of uni-directional admissible algorithms A*[30] to bi-directional algorithms at first appears simple. Naively, one would in VGA replace g_s by $f_s = g_s + h_s$. This seems reasonable and worked in the uni-directional case where Dijkstra's algorithm was so extended to the class A*.

However, a modification of our canonical counterexample, to include Euclidean information (see Fig. 9.1) as the heuristic information, refutes the above extension of VGA. In the figure the Euclidean distances are marked above the dotted lines, with the example drawn accurately to show it is realizable in the plane. We step through VGA modified as described above using for step 2 the alternating rule: on odd iterations use the forward direction and on even iterations use the backward direction.

Iteration 1:

s is placed in S $\quad f_s(s) = 6$

t is placed in T $\quad f_t(t) = 6$

nodes a and b are placed in $\tilde{S}$

$$f_s(a) = 7, \qquad f_s(b) = 7\frac{5}{8}$$

nodes a and c are placed in $\tilde{T}$

$$f_t(a) = 7, \qquad f_t(c) = 7\frac{5}{8}$$

node a is min in $\tilde{S}$ and is placed in S, t is placed in $\tilde{S}$, $f_s(t) = 8$.

Iteration 2:

node a is min in $\tilde{T}$ and is placed in T a $\epsilon S \cap T$ and therefore we go to step 6 of VGA to terminate.

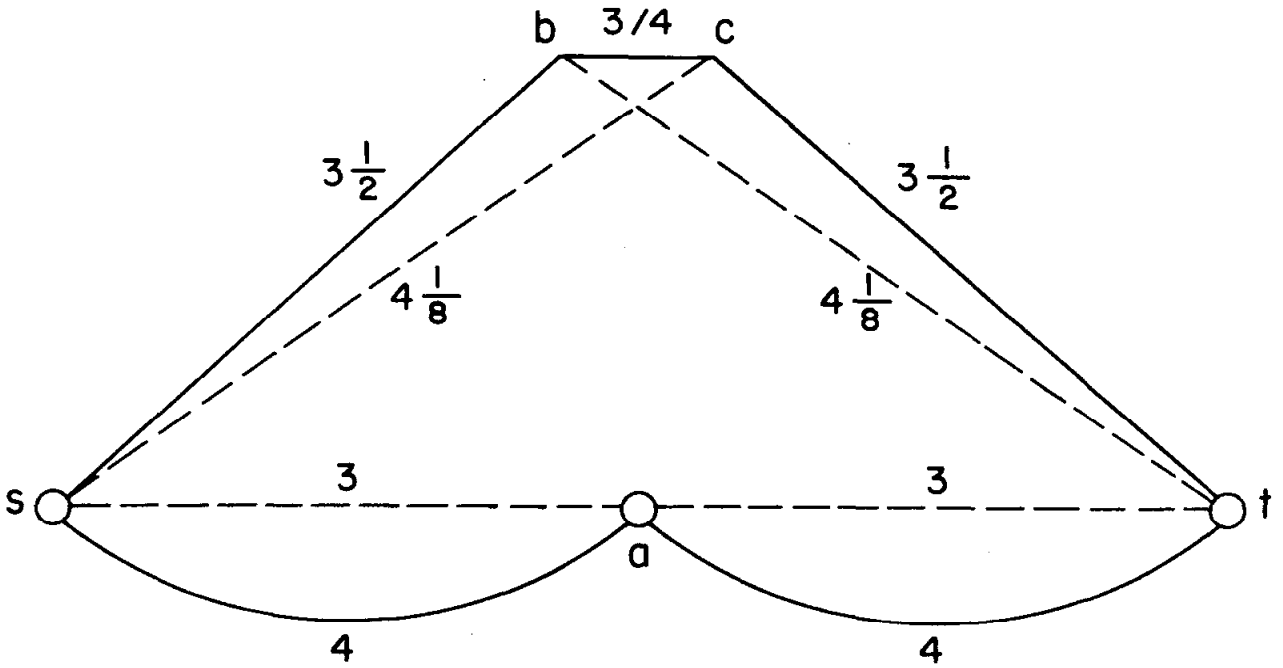


FIG. 9.1--Euclidean counterexample.

1269A8

The only path found is (s, a, t) with length 8. However this is the wrong result, the path (s, b, c, t) has length $7\frac{3}{4}$. Thus the algorithm is not admissible even though the Euclidean metric satisfies the lower bound criterion and the 'consistency' (Ref. 30, p. 12) condition for heuristic functions. The problem is more complex than the uni-directional extension, and possibly the more complete use of the heuristic information must go into the selection of the next node to be expanded

An attractive criterion for expansion is the minimum over

$$\forall x \in S \; \forall y \in T \left(g_s(x) + h(x, y) + g_t(y) \right)$$

where h(x, y) is a heuristic estimate of the distance from x to y. We want h to behave nicely and we use an h which satisfies consistency so

$$h(x, t) \leq h(x, y) + g_t(y)$$

This means that we need only look at $t \in T$ (and it must always be in T from the initialization step). Then we have come back to the original algorithm which was shown to fail. So even this calculation, almost exhaustive with respect to both the forward and backward sets adds nothing. In fact if we had not deduced its equivalence to our previous method, we would have to calculate $O(|\tilde{S}| \cdot |\tilde{T}|)$ computations per iteration which is not computationally feasible in problems of interest.

9.2 Correct Extension — The Very General Heuristic Algorithm

The above attempts are of interest in displaying the subtlety of the problem in bi-directional approaches. When dealing with the shortest path problem, Berge and others had trouble because of the terminating condition. We have been hesitant in our extension to the heuristic case of challenging one previously well oiled terminating condition. Once again, it is just this point that is the stumbling block.

In reconsidering the problem, an elegant device will be used, which hopefully clarifies and lays to rest the termination confusion. In place of distance we will substitute the concept of excess or waste. Waste will be the amount a solution takes over some optimum which is estimated initially. Equivalent to finding the least wasteful path is finding the shortest path. We define r* to denote our original estimate of optional distance.

$$r^* = h_s(s) = h_t(t) \ .$$

the waste in the forward direction is

$$w_s(n) = f_s(n) - r^* = g_s(n) + h_s(n) - r^*$$

and in the backward direction is

$$w_t(n) = f_t(n) - r^* = g_t(n) + h_t(n) - r^* \ .$$

We can now describe VGHA — the very general heuristic algorithm, where the changes from VGA will be that w_s and w_t are used in place of g_s and g_t. Also the termination condition is changed and the h used in computing f satisfies both lower boundedness and consistency. The consistency assumption (Ref. 30, p. 12) is a form of triangle inequality for the distance estimators used in these domains.

VGHA

1. Place s in S and calculate a w_s for all successors of s placing them in $\tilde{S}$.

$$w_s(x_i) := \ell\left(e_{sx_i}\right) + h_s(x_i) - r^*$$

$$wf(x_i) := s$$

similarly calculate all predecessors of t, placing them in $\tilde{T}$ and t in T.

$$\text{Set } a_{min} := \infty$$

2. Decide to look at either $\tilde{S}$ or $\tilde{T}$.

3. If $\tilde{S}$ was selected in step 2, then select node $x \epsilon \tilde{S}$ which has the smallest $w_s(x_i)$.

4. Place x from step 3 in S and check if $x \epsilon S \cap T$. If yes then

$$a_{min} := \min \left(a_{min}, g_s(x) + g_t(x) - r^* \right)$$

5. For each successor (predecessor) of x calculate $w_s(w_t)$ and see if they are in $\tilde{S}$ ($\tilde{T}$) yet. If not place in $\tilde{S}$ ($\tilde{T}$) where

$$w_s(x) := g_s(x) + \ell \left(e_x(x) \right) + h_s(x) - r^*$$

$$wf(x) := x$$

If the successors (predecessors) are already in $\tilde{S}$ ($\tilde{T}$), but the new value is lower, update the values.

6. If

$$a_{min} \leq \max \left(\min_{x \epsilon \tilde{S}} \left(w_s(x) \right), \min_{x \epsilon \tilde{T}} \left(w_t(x) \right) \right)$$

then terminate with the path that gave this a_{min}. Otherwise go to step 2.

We will show that this algorithm is indeed correct for any decision procedure used by step 2. Before going on, note that r^* is a constant throughout the computation and may be dropped without changing the order in which nodes are found or the path found. The notion of waste is a didactic one and makes the algorithm more persuasive in its correctness.

Also at this point it is useful to step through the algorithm using our previous counterexample. We again employ an alternating strategy. The reader is welcome to employ any strategy he can think up. For example a forward uni-directional strategy clearly yields the admissible algorithm A*.

Iteration 1:

s is placed in S $\quad \begin{cases} r^* = 6 \\ w_s(s) = 0 \end{cases}$

t is placed in T $\quad w_t(t) = 0$

b, a are placed in $\tilde{S}$

$$w_s(b) = 3\frac{1}{2} + 4\frac{1}{8} - 6 = 1\frac{5}{8}$$

$$w_s(a) = 4 + 3 - 6 = 1$$

c, a are placed in $\tilde{T}$

$$w_t(c) = 1\frac{5}{8} \qquad w_t(a) = 1$$

node a is min in $\tilde{S}$ and is placed in S

t is placed in $\tilde{S}$ $\quad w_s(t) = 8 - 6 = 2$.

Iteration 2:

node a is min in $\tilde{T}$ and is placed in T

s is placed in $\tilde{T}$ $\quad w_t(s) = 8 - 6 = 2$

$$a \in S \cap T \qquad \text{and} \qquad a_{min} = 2$$

This is not a minimum over either $\tilde{S}$ or $\tilde{T}$ and so the algorithm continues, where before we stopped (incorrectly) at this point.

Iteration 3:

node b is min in $\tilde{S}$ and is placed in S

c is placed in $\tilde{S}$ $\quad w_s(c) = 1\frac{3}{4}$

a_{min} is still too large.

Iteration 4:

node c is min in $\tilde{T}$ and is placed in T

b is placed in $\tilde{T}$ $\quad w_s(b) = 1\frac{3}{4}$

a_{min} is still too large.

Iteration 5:

node c is min in $\tilde{S}$ and is placed in S

$$c \in S \cap T, \qquad a_{min} = 1\frac{3}{4}$$

which is a minimum for either set $\tilde{S}$ and $\tilde{T}$ and therefore we halt with path (s, b, c, t) the shortest path from s to t.

9.3 Correctness of VGHA

The argument is even simpler than the one for VGA. In essence, the algorithm only terminates when the actual waste of the current best path is less than any estimated waste of the set of possible paths. Since h_s and h_t are always a lower bound, actual waste must be at least as much as expected waste, and therefore we have the shortest path.

Theorem 9.1:

Upon termination, the path from s to t found by VGHA is the shortest path from s to t. We are assuming positive edge lengths and the existence of some path.

Pf.

Consider the algorithm terminated with path $\mu = (x_1, x_2, \ldots, x_k)$, but that this is not the shortest path. The shortest path is instead $\mu^* = (y_1, y_2, \ldots, y_\ell)$

$$x_1 = y_1 = s, \qquad x_k = y_\ell = t \ .$$

By lemma 2 of Ref. 30, any node in S or T has a shortest path already found, leading back to respectively the initial or terminal node. Now there is some y_i and y_j, such that $i < j$ and $y_i \in S$, $y_{i+1} \notin S$ and $y_j \in T$, $y_{j-1} \notin T$. If this were not so, μ^* would have been found before termination. We claim:

$$w_s(y_{i+1}) < a_{min}, \qquad y_{i+1} \in \tilde{S}$$

$$w_t(y_{j-1}) < a_{min}, \qquad y_{j-1} \in \tilde{T}$$

This is because h_s is a lower bound and therefore w_s must be a lower bound on waste for nodes with their least $g_s(n)$.

Since $\ell(\mu^*) < \ell(\mu)$, its waste is smaller and so is any lower bound on this value. Thus, the algorithm could not have been terminated, because a_{min} is not less than or equal to the minimum waste over either $\tilde{S}$ or $\tilde{T}$.

The algorithm must terminate since all nodes are eventually placed in either S or T, including those on the shortest path (Ref. 30, p. 10). ■

Corollary 9.1:

If h_s and h_t satisfy the lower bound condition, but not necessarily consistency, then VGHA still terminates with the correct solution, if step 5 is modified. Step 5 must add the following: If a smaller value of w_s (w_t) is found for a node already in S (T), then the node is removed from there and placed back in $\tilde{S}$ ($\tilde{T}$).

Pf.

The above proof of theorem 9.1 is used here again as the central argument. However lemma 1 of Ref. 30 must now be used to show there is a node x in $\tilde{S}$, $x \in \mu^*$ with $w_s(x) < a_{min}$. A similar node exists in $\tilde{T}$.

The heuristic functions h_s and h_t need not be defined in similar fashion, so long as each satisfies the lower bound condition. However one ordinarily makes use of the best possible heuristic function available. In so far as one regards growing the shortest path tree S (or T), the results of Ref. 30 hold with respect to one heuristic function dominating another. If h_{s1} is always a greater lower bound than h_{s2}, then the tree h_{s1} grows will be at least as sparse as the tree grown using h_{s2} (theorem 3 in Ref. 30) out to any given node along the shortest path.

9.4 Strategies in Bi-directional Search

After selecting appropriate heuristic functions, we are again confronted as was the case with VGA, with what decision rule to use in step 2 of VGHA. In proposing an optimal strategy for VGA, we made two hypotheses

1. monotonicity of $\widetilde{S}$ and $\widetilde{T}$
2. equi-probability of any node in these open sets being the next node on the shortest path.

The monotonicity hypothesis is usually satisfied in large graphs with an average degree of greater than one, which is also a characteristic of complex problem domains. The fifteen puzzle has a cardinality of 16!/2 with an average degree of three. The equi-probable hypothesis coupled with the monotonicity hypothesis means that as the algorithm iteratively augments S and T, the sets grow larger and the probability at a given iteration of finding the next node of the shortest path decreases proportionately to the size of the set. Under these conditions, the cardinality comparison strategy produces the best selection of either the forward or backward direction in the sense that the expected number of nodes visited is a minimum over all strategies. A further justification is that if the decision on a particular iteration is independent, then it is clearly better to choose from the smaller set.

Our hypotheses are reasonable and VGA following the cardinality comparison decision rule outperforms other strategies. However, as in the case of the monotonicity hypothesis discussed previously, the equi-probable hypothesis is not exact. In point of fact the edges of a graph are not equi-probable in the number of occurrences along the shortest paths, but in some complicated fashion these probabilities are inversely proportional to the edge lengths suitably normalized with respect to the smallest edge length. The longest edge in the graph has a

smaller chance of lying on some shortest path than the shortest edge; providing they are not of equal length.

Consider a directed graph of size n with edge lengths distributed from 1 to n. An edge of unit length must appear in at least one shortest path — namely the path it constitutes. However, an edge of length n (or even of length 2) may appear in none. For example, the graph in Fig. 9.2 where there is a Hamilton circuit[49] with each edge of unit length. Edges off the circuit are of length seven. The longest shortest path in this graph is of length three ($\mu = (1, 2, 3, 4)$) and only edges of unit length are included in any shortest path. Theoretically, a probability should be assigned to a node in an open set depending on its distance from the root node in comparison to the other members of the set.

A decision strategy to be optimal must select the open node that has the highest probability of being on the shortest path. Each set $\tilde{S}$ or $\tilde{T}$ must be considered separately. Each will have a current best candidate whose probability of being on the shortest path is related to the values of the distance computed for each member of the set.

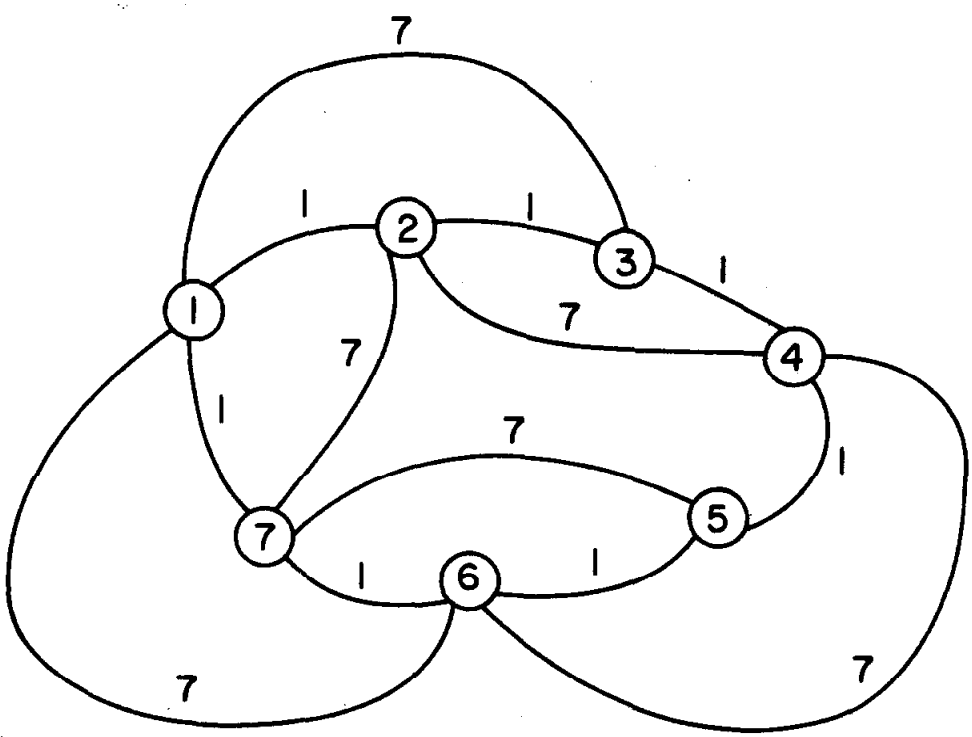
Set $\tilde{S}$ has $|\tilde{S}|$ nodes each having some distance (or waste) value. If p_1, p_2, ..., $p_{|S|}$ are the probabilities that a given node is the next node along the shortest path, then we must have

$$\sum_{i=1}^{|\tilde{S}|} p_i = 1$$

where we select the node with the highest probability. We also do this for $\tilde{T}$ and then compare, taking that direction and node with the current maximum probability.

If for example, there is only one node in set $\tilde{S}$ then no matter what its value (of waste or distance), it must lie on the shortest path.

$$|\tilde{S}| = 1$$



I269A9

FIG. 9.2--All shortest paths are subpaths of a Hamilton circuit.

and

$$\sum_{i=1}^{1} p_i = 1 .$$

Consider some computation reaching the following situation:

$$\tilde{S} = (x_1, x_2, x_3)$$

$$\tilde{T} = (y_1, y_2)$$

$$f_s(x_1) = 5, \qquad f_s(x_2) = 5$$

$$f_s(x_3) = 200$$

$$f_t(y_1) = 100$$

$$f_t(y_2) = 100$$

Knowing nothing else about these nodes we have

$$p_{y_1} = p_{y_2} = 1/2$$

$$p_{x_1} = p_{x_2} = 1/2 - \epsilon \qquad p_{x_3} = 2\epsilon$$

So we would select the backward direction choosing either y_1 or y_2 even though $f_t(y_1) \gg f_s(x_1)$. Since a graph is a complex structure, a distribution of edge lengths and the consequent probabilities induced with respect to a given shortest path problem is exceedingly difficult to calculate. The equi-probable assumption is a workable approximation which requires no additional computation.

The theoretical results (theorem 4.4) still hold for VGHA, but the discussion of probabilities must now include the heuristic estimator. A node whose expected path length is small is better than a node with a larger expected path length. We also have to worry about the accuracy of our heuristic estimate. Thus VGHA provides a more complicated decision problem then VGA, and cardinality comparison is again a pragmatic solution. However, some other problems arise in

bi-directional heuristic search strategies, and in our discussion of open questions we will make further comments on this topic.

9.5 Associative Search as a Solution to Redundancy and Intersection

Previously we had mentioned the antipathy to bi-directional search because of the extra mechanism involved — especially with regard to the termination problem. Step 4 of VGHA asks to check if $x \in S \cap T$. If x was just placed in S, we must search all the nodes of T for that same state. In the VGA case we normally have an explicit representation of a finite graph of n nodes. We can then keep logical vectors for each set.

```
Boolean array    SVEC, TVEC [1:n];
        comment  SVEC[i] = true if node i is in set S — similarly for TVEC.
        When node i is placed in set S its SVEC [i] is set true and the check
        for intersection is:
            if   TVEC [i] then go to intersect
                 else go to nointersect .
```

When a node i represents some state $\vec{v}_i$ and the nodes are being generated by a successor (predecessor) routine, we do not have a predetermined node-state labeling. The check for whether some state $\vec{v}_i$ is in a given set requires a search of the set node by node with a comparison of the associated states to $\vec{v}_i$. This comparison is necessary not only in determining $x \in S \cap T$, but is also needed for finding out if a new node is redundant.

In generating search trees where the space contains cycles, the same state can be reached along many alternate paths. These redundant nodes can be ignored and left in the search tree, or for each node generated a redundancy check can be used. This requires checking each new node against all others generated in the

same search direction. The work involved in making a simple comparison search would be $0(n^2)$ where n is the number of nodes in the final search tree. For spaces where there is likely to be a small amount of redundancy, the extra effort in weeding it out is greater than generating a small number of nodes redundantly.

The redundancy problem and the intersection problem are both the same problem. There is a minor difference in that the redundancy problem requires the check to be in the same search tree as the state of interest, while the intersection problem requires the check to be in the opposite tree as the state of interest. Conceptually what is needed is an associative search. A state is a vector $\vec{v}_i = \left(v_i^{(1)}, v_i^{(2)}, \ldots, v_i^{(k)}\right)$ and a simple hash function over this vector would allow us to simulate associative search.

One hash function possible, which we use in our implementation is

$$\text{hash}_i = \sum_{j=1}^{k} j \cdot v_i^{(j)}$$

where each node with this hash value is chained together. Then a check for redundancy or intersection consists of computing the hash value and doing a chained search of all nodes with this hash value. The search reduction possible for a well-behaved hashing scheme is on the order of the number of equivalence classes produced by the hash. Experimentally, a function of the above form was used with the fifteen puzzle — dividing the 16!/2 positions into 680 equivalence classes. This produced two orders of magnitude increase in running time for 1000 node searches. The hash chains were between 0-30 long for a tree of size 1000, which is a considerable improvement over searching the whole tree.

The importance of this computational idea should not be underestimated. This idea recurs throughout combinatoric and enumerative programming. In some sense the hash provides a semi-canonical form. It is even conceivable

that the hash be the truncated evaluation value of the node. These uses of a hash places the bi-directional search inner loop almost on a par with the uni-directional search inner loop, which does not need to check for intersection but just if x is the goal.

CHAPTER 10

FURTHER OBSERVATIONS, OPEN PROBLEMS, AND CONCLUSIONS

It has been productive to use a computational approach to the shortest path problem. This computational — combinatoric — heuristic blend has also been useful in the Hamilton path problem,[49] the traveling salesman problem,[35] and the graph isomorphism problem.[13,59] This area is pregnant with untried possibilities for usefully handling difficult problems. A noteworthy achievement will be the unification of these methods into a package[48,53] with interactive capability.[3,38] These will aid in solving applied problems such as optimization of resource allocation, and will also help graph theorists generate and test examples and conjectures.

The work on efficient graph algorithms is just at a beginning stage. The idea of using local properties,[35,49,51,58,59] which may be easily computed, to aid in some global calculation normally requiring an exponential amount of work is gaining wider attention. Large graph problems, like constrained optimal path problems, certainly require the special intuition the computer scientist has in regard to efficient computation.

10.1 Network Flow Algorithm

One immediate extension of our work on bi-directional shortest path methods is to the network flow algorithm of Ford and Fulkerson.[25] The heart of their algorithm is the flow augmenting procedure. This is just a path finding procedure which can be handled by our more efficient bi-directional methods. This gains a factor of efficiency over the ordinary uni-directional method, which is comparable to the improvement in the shortest path algorithm. It is probable that bi-directional techniques are feasible for other search problems with similar computational savings.

In using the path problem as a vehicle for studying artificial intelligence in a rigorous manner, many open problems and extensions of our work and the recent work of others[19,37,34,55] remain.

10.2 Bi-directional Intersection

In the shortest path problem we have the bi-directional search expanding like two wave fronts in shortest path space. In the heuristic path problem we have the search expanding like two cones (see Fig. 10.1). Effective use of bi-directional

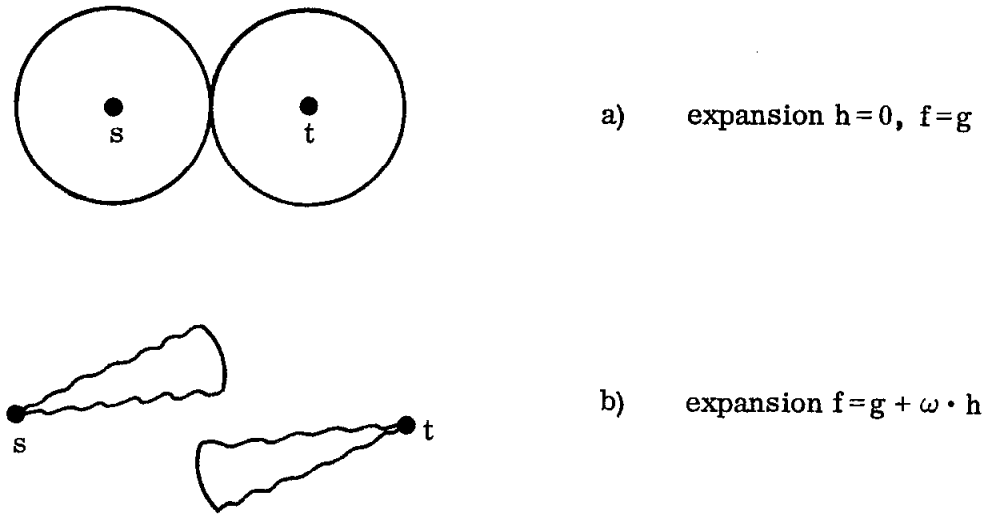


FIG. 10.1--Bi-directional search.

heuristic search requires that the cones meet each other near the middle of their separation. Otherwise, if they intersect near the endpoints it is twice the work of uni-directional heuristic search. When bi-directional search works, it provides a means of finding solutions of path length 2k with only twice the computation needed for the uni-directional search of a solution of path length k. The uni-directional search would need $0(\rho^k)$ extra work. The payoff in making bi-directional heuristic search work is therefore quite large, leaving the solution to the intersection problem an important open question.

At first, we anticipated no problem and because of the symmetry of the 15 puzzle,ran a simple alternating bi-directional search. The typical result was that both the forward and backward tree had grown almost complete but separate solution paths. Intersection would occur near one or the other endpoint rather than the middle of the space. It then becomes apparent that each heuristic search tree is a tiny sliver in the search space and very unlikely to intersect each other even when moving approximately toward each other. It is as if two missiles were independently aimed at each others base in the hope that they would collide. Two attempts were made to guide the trees toward each other, but neither have yet proved fruitful.

Intermediate Board Conjecture

Suppose we have an initial state $\vec{v}_s = \left(v_s^{(1)}, \ldots, v_s^{(k)}\right)$ and a terminal state $\vec{v}_t = \left(v_t^{(1)}, \ldots, v_t^{(k)}\right)$ and we can conjecture some intermediate state $\vec{v}_i = \left(v_i^{(1)}, \ldots, v_i^{(k)}\right)$. We then take our heuristic function which in the forward direction is measuring the distance from node x to node t and add a term corresponding to the distance from node x to node i and similarly for the backward evaluation. We should then have a search which would steer toward its respective endpoint by way of the intermediate position.

The method does not specify that the intermediate node must be visited — only that it is used as part of the heuristic evaluation. We could have specifically subdivided the original problem into two sub-problems of finding a path from s to i and from i to t. This idea which is close to the notion of "lemma" in theorem proving is certainly an important one, and an interesting topic for further research. The "lemma" conjecturing problem and planning in general fit naturally into our model.

In using the intermediate board conjecture for weighting our search we hope to improve our intersection likelihood. For example two possible intermediate classes of positions in the 15 puzzle could be the ones in Fig. 10.2. We could

1	2	3	4
5	6	7	8
x	x	x	x
x	x	x	x

a) halving

1	2	3	4
5	x	x	x
9	x	x	x
13	x	x	x

b) 8-puzzle reduction

FIG. 10.2--Intermediate board conjecture.

calculate P for a board position and add the indicated positions twice. This was tried and did not work. It seems that the interaction between aiming at a goal position and simultaneously placing emphasis on certain vector components is complex and must be studied in more detail.

Shaping

Another possibility is the continuous updating of the heuristic function to measure the distance to the front of the opposite search tree. This is using our function to continuously re-aim our missiles at the point that the opposing missile just reached. In experimenting with bi-directional methods using this type of search tree 'shaping,' we also had no success. The searches produced in this instance were much worse than without shaping. Possibly the fact that the heuristic functions are inaccurate creates a severe instability when continually re-aiming the search.

Both shaping and weighting intermediate positions have been examined. The effective use of bi-directional heuristic search is important enough to warrant

further investigation into pragmatic and theoretical devices for forcing search tree intersection.

10.3 Learning

In Chapter 8 we explored empirically the relation of search efficiency to a parameter ω. The adjustment of this weight could be "learned" as HPA solved problems in a particular domain. In general we may have several heuristic functions h_1, h_2, ..., h_k and g. These are functions over the state space which we would like to use in directing our search. In a simple instance we may attempt to find some linear combination

$$f = g + \omega_1 h_1 + \omega_2 h_2 + \ldots + \omega_k h_k$$

where the ω_i's are adjusted as in Ref. 54.

More interesting is to attempt to find automatically a useful heuristic function. Possibly these are observed by noting interesting structural characteristics of the space.

10.4 Structural Features — Bridges

In Ref. 2, Amarel notes the importance of 'narrows' in these graph spaces. These are in more traditional terms proper cut-sets with bridges being especially important.[9,14,28]

A cut-set of a graph is a set of edges which, when removed from the graph, leaves the graph unconnected. A proper cut-set is a cut-set which has no proper subset which in turn is a cut-set. A bridge is a cut-set of one edge, and is therefore identically a proper cut-set. A graph is called h edge-connected*, when h is the cardinality of its smallest (proper) cut-set (see Fig. 10.3).

*Berge[7] calls this h-coherent, but we will from now on refer to graphs as h-connected, meaning edge-connected.

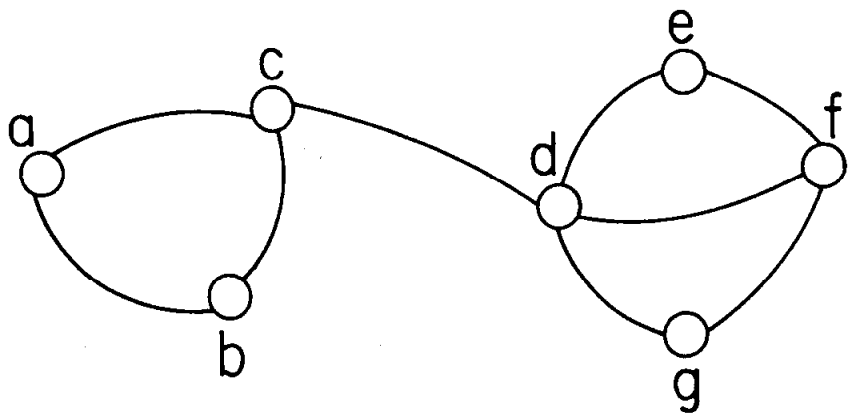


FIG. 10.3--Edge (c,d) is a bridge.

Find all the bridges in a graph. One can do this simply by removing each edge in turn and checking the remaining graph for connectedness. There are up to n(n-1)/2 edges in a loop-free undirected graph and this approach is obviously too tedious.

At this point let us note a simple theorem (Ref. 11, p. 18):

> "Every spanning tree has at least one edge in common with every cut-set of a graph."

In particular, we note that any spanning tree must contain all bridges of the graph. Generating a spanning tree is a simple computation, and is on the average only twice the work of generating a path. Now in a dense graph there are many spanning trees possible, and by suitably generating successive spanning trees and intersecting their edge sets, one should be left with only a smaller number of edges (< n) to check as bridges. This then is the method we outline below in detail.

Spanning Tree Algorithm

1. Mark all nodes as unreached and unused.
2. Choose some node $i \in G$ as the root node and mark it reached.
3. Select any node n that is reached but unused and mark it used.
4. Mark all nodes n_k, which are connected by an edge to n and not previously reached as reached. Include the edges (n, n_k) in the spanning tree.
5. If all the nodes in G are reached then halt, else go to step 3.

By selecting different root nodes and by suitably varying the order in which nodes are examined in step 3, a reasonably different sampling of spanning trees will be constructed, if possible. One simple possibility is to use reached nodes in ascending value and varying this by next choosing them in descending value. Also this algorithm is a test for connectedness, for if no reached but unused nodes exist at some stage before the computation halts, the graph must be unconnected.

Bridge Finding Algorithm

1. Compute two spanning trees in different (as possible) ways.
2. Find the set of edges in the intersection of these two trees — set I.
3. If I is empty halt.
4. Take the first edge in I and delete it from the graph and from I.
5. Generate a new spanning tree (again try to make it different from the previous ones).
6. If the tree does not have all the nodes of the graph, then list the removed edge as a bridge. Otherwise, intersect the new tree with I to obtain the new I. Return to step 3.

Remark: At most n-1 spanning trees will be constructed, where this limit is attainable.

Pf.

A spanning tree of a graph of size n has n-1 edges. Therefore set I can have at most n-1 edges initially. If the graph is just a simple circuit:

$$X = \{1, 2, 3, \ldots, n\}$$

$$E = \{(1,2), (2,3), \ldots, (n,1)\} ,$$

then the maximum number of intersections will be achieved.

If the graph of interest is dense, then there will be many possible different spanning trees. The intersection of two of these will leave but few candidate edges. Outside of an iteration required for each bridge found, the algorithm will normally need only a few intersections before all extraneous edges are discarded. In implementing the algorithm, the number of intersections stayed between three and five over a wide range of graph sizes and densities.

Generalization

The more general problem of finding the minimum proper cut-sets of a graph is a great deal more difficult. Methods based on the repeated use of the Ford-Fulkerson network flow algorithm,[25] with edge capacities identically one, can be used. The fundamental result is that the maximum flow is equal to the minimum cut capacity and the Ford-Fulkerson algorithm may be programmed to find the cut-set. In the case of bridges, obviously the tree intersection algorithm requires substantially less work. While the Ford-Fulkerson algorithm is efficient, it is more complex than the simple tree spanning algorithm, and each iteration of it is about the same work as a complete spanning tree computation.

It is possible to generalize our method to cut-sets of higher order. Consider a cut-set of cardinality 2; name it C_2. By our theorem, each spanning tree must

include one or the other edge of C_2. Therefore if k spanning trees are generated, some member of C_2 will appear more than k/2 times. If edges are investigated in order of number of occurrences (given that they appear $\geq$ k/2 times) the case of finding the other edge in C_2 is reduced to finding a bridge. This scheme seems more reasonable, especially in very dense graphs, than the more complex flow algorithm. The method, of course, is iteratively applicable to C_n with a criterion of k/n appearances. However, it is most reasonable for n small.

Efficient algorithms for the recognition of important structural characteristics of problem spaces is but one of many fruitful approaches to general problem solving methods in our model.

Other areas of interest are statistical error analysis instead of worst case. If closed form solutions are unavailable then Monte Carlo simulations should aid in understanding these problems. Parallel computer organization should also prove important in extending the class of problems which can be solved. These are but a few of the possibilities for extension of the approach we have tried to use throughout this work.

10.5 Some Concluding Remarks

One is always questioned on the significance of the work. In general this leads to some exaggeration, especially when one has some perspective on the range of human thought. A simple answer is to say — here is the best shortest path method or a first theory of heuristic search. However, we would rather stress a notion of computational insight coupled to some combinatoric rigor and experimental investigation. In a sense this work is using the computer in the Von Neumann sense[60] of heuristic — to gain a feel or intuition into a difficult problem domain, and it is hoped some small contribution has been demonstrated.

APPENDIX I

ALGOL W IMPLEMENTATION OF VGA

This appendix contains a commented ALGOL W implementation of VGA. We will explain the data representation used and the purpose of each procedure.

In my experiments two different data representations were used:

a) adjacency-matrix

b) edge list

Representation (a) is a n×n matrix A, where n = |G|. An element of A, a_{ij} is the length of the edge (i, j). If there is no such edge then a large positive value representing ∞ is entered. Representation (b) is for very large sparse graphs. It consists of two n element vectors, in-index and out-index and four n maxind matrices, in-edge, out-edge, in-length and out-length. Maxind is the maximum degree of any node in the graph. For each node i, in-index (i) is the number of predecessors it has, and out-index (i) is the number of successors it has. Then out-edge (i, 1::out-index (i)) is a list of successors of node i; the lengths represented by these edges are stored in out-length (i, 1::out-index (i)). The corresponding arrays and matrices represent the predecessor. This representation requires n × (4 × maxind + 2) words and can save considerable storage space over (a) for large sparse graphs. Consider that we have 42,000 words of store, then representation (a) could store a complete 200 node graph. Representation (b) could store a complete 100 node graph, but if the maximum degree is 10 it could store a 1000 node graph.

Below we show both representations for Nicholson's graph (Fig. 1).

(a)

∞	3	6	7	∞	∞	∞	∞	∞
3	∞	1	∞	4	∞	∞	∞	∞
6	1	∞	∞	∞	2	∞	∞	∞
7	∞	∞	∞	∞	3	4	∞	∞
∞	4	∞	∞	∞	∞	∞	1	∞
∞	∞	2	3	∞	∞	∞	1	2
∞	∞	∞	4	∞	∞	∞	∞	5
∞	∞	∞	∞	1	1	∞	∞	2
∞	∞	∞	∞	∞	2	5	2	∞

$\alpha \equiv$ (1::in-edge (node)), $\beta \equiv$ (1::out-edge (node))

(b)

node	in-index	in-edge$_\alpha$	in-length$_\alpha$	out-index	out-edge$_\beta$	out-length$_\beta$
1	3	2,3,4	3,6,7	3	2,3,4	3,6,7
2	3	1,3,5	3,1,4	3	1,3,5	3,1,4
3	3	1,2,6	6,1,2	3	1,2,6	6,1,2
4	3	1,6,7	7,3,4	3	1,6,7	7,3,4
5	2	2,8	4,1	2	2,8	4,1
6	4	3,4,8,9	2,3,1,2	4	2,3,1,2	2,3,1,2
7	2	4,9	4,5	2	4,9	4,5
8	3	5,6,9	1,1,2	3	5,6,9	1,1,2
9	3	6,7,8	2,5,2	3	6,7,8	2,5,2

The graph is undirected and this symmetry is found in each representation by noting

a) $A = A^T$

b) in-values = out-values

The version listed here works with (b) and has generated and used 1000 node graphs.

Graph generating procedures:

1. RANDOM - a pseudo-random number generator with range (0, 1).

2. GENEDGE - uses random to generate edge-list representations of undirected graphs. A node is not allowed to have more than MAXIND edges. The lengths are randomly generated integers with range [1, WT].

Shortest path algorithm — VGA:

1. WBIED - this is the ALGOL W incarnation of VGA. It consists of the following local procedures.

2. DECIDE - a logical function procedure representing step 2 of VGA. It contains various strategies of interest in a case expression, e.g., number 1 is cardinality comparison.

3. INITIAL - this does the initialization step, step 1 of VGA.

4. MIDDLE - this is steps two through five of VGA and is the basic iterative loop. Both this procedure and INITIAL are distinct because of the OS/360 segmentation problem.

Analysis procedures:

1. SORT - this bubble sorts the shortest distances found to the nodes in set S (set T).

2. DLAMBA - counts the number of nodes in S (T) with distance less than R from initial (terminal) node.

Graphs of different size and density and edge length distribution were generated by GENEDGE. Node pairs were selected from these graphs and for each pair different decision rules in DECIDE were compared for efficiency.

```
%ALGOL

0001          BEGIN  COMMENT  IRA POHL  SLAC (1968)
0002
0003              ALGOLW IMPLEMENTATION OF VGA - VERY GENERAL ALGORITHM
0004           FOR FINDING SHORTEST PATHS IN DI-GRAPHS. THIS VERSION USES THE
0005           EDGE LIST REPRESENTATION FOR ECONOMICAL STORING OF LARGE SPARSE
0006           GRAPHS.
0007              EXPLANATIONS OF PARAMETERS OCCUR THROUGHOUT THIS PROGRAM NEAR
0008              THEIR ACTUAL USE.                                    ;
0009           INTEGER SCOUNT,TCOUNT,CNTNODE;
0010
0011           INTEGER N,MAXIND,START,TERMINUS,COUNT,INF;
0012           INTEGER RANDOMX; LONG REAL RANDOMC;
0013           INTEGER DENSITY, DF, DB, MIND ;
0014           INTEGER I,J,LAST;
0015
0016           COMMENT     ***** DEFINITION OF GLOBAL VARIABLES  *********
0017
0018              MANY OF THE VARIABLES ARE SYMMETRIC WITH REGARD TO THE FORWARD
0019           AND BACKWARD DIRECTION. NORMALLY FORWARD VARIABLES ARE BEGUN WITH
0020           S AND BACKWARD ONES WITH T. THE DEFINITIONS WILL BE WITH RESPECT
0021           TO THE FORWARD VARIABLES WITH THE CORRESPONDING BACKWARD ONES IN
0022           PARENTHESES.
0023
0024           SCOUNT= CARDINALITY OF SET S  (TCOUNT)
0025           DF= FORWARD DISTRIBUTION VALUE FOUND BY DLAMBDA (DB)
0026           DENSITY= EDGE DENSITY OF THE DI-GRAPH. 10^ PER-CENT REPRESENTS
0027           THE COMPLETE GRAPH.
0028           N= CARDINALITY OF GRAPH, START= INITIAL NODE, TERMINUS=FINAL NODE,
0029           INF= REPRESENTS INFINITY,
0030           MIND=MINIMUM DISTANCE FOUND- IS INF IF NO PATH EXISTS.
0031           CNTNODE= THE CENTRAL NODE ON THE SHORTEST PATH, I.E. THE NODE FOUND
0032           BY BOTH THE FORWARD AND BACKWARD SEARCH.
0033
0034           OTHERS ARE TEMPORARIES OR ARE UNIMPORTANT OR ARE EXPLAINED LATER ON;
0035
0036
0037           COMMENT RANDOM GENERATES RANDOM NUMBERS 0<RANDOM<1   ;
0038
0039           LONG REAL PROCEDURE RANDOM;
0040           BEGIN  RANDOMX:=1220703125*RANDOMX;
0041              NUMBER(BITSTRING(RANDOMX) AND #7FFFFFFF)/RANDOMC    END;
0042
0043           COMMENT THE FOLLOWING VALUES MUST BE INITIALIZED  :
0044
0045            INTFIELDSIZE  I/O PARAMETER , INF  INFINITE EDGE LENGTH
0046            MAXIND  THE MAX LOCAL DEGREE ALLOWED IN THE LIST REPRESENTATION ;
0047
0048
0049              RANDOMX:=1;  RANDOMC:=2**31;
0050              INTOVFL:= NULL;
0051              INTFIELDSIZE:=6;
0052              INF:=999999;
0053
0054            COMMENT  MAXIND=MAX DEGREE ALLOWED   K=GRAPH SIZE ;
0055
0056           FOR MAXIND:=7 DO
0057           FOR K:=1000 DO
0058           BEGIN
```

```
0059
0060            COMMENT  ***EDGE LIST REPRESENTATION OF DI-GRAPH ***
0061
0062               THE MAXIMUM DEGREE OF A NODE IS MAXIND DEFINED IN THE OUTER
0063            BLOCK ALONG WITH K THE NUMBER OF NODES. THE ACTUAL DEGREE
0064            OF EACH NODE IS FOUND IN THE ARRAYS ININDEX AND OUTINDEX. THESE
0065            ARE IN AND OUT EDGE DEGREES RESPECTIVELY. EACH NODE HAS A LIST
0066            OF ITS SUCCESSORS AND PREDECESSORS STORED IN INEDGE AND
0067            OUTEDGE. CORRESPONDING TO THESE LISTS ARE THE LENGTHS OF
0068            THESE EDGES FOUND IN INLENGTH AND OUTLENGTH
0069
0070               THE REMAINING ARRAYS ARE USED BY THE SHORTEST PATH ALGORITHM
0071            WHEN BUILDING MINIMUM PATH TREES.  WF IS THE WHERE FROM POINTER
0072            FOR THE FORWARD TREE AND SDIST IS THE CURRENT BEST DISTANCE.
0073            WT AND TDIST PLAY THE SYMMETRIC ROLE IN THE BACKWARD CASE. DIST
0074            IS A TEMPORARY NEEDED IN DOING A POSTERIORI ANALYSIS.  ;
0075
0076            INTEGER ARRAY WT,SDIST,TDIST(1::K+1);
0077            INTEGER ARRAY ININDEX,OUTINDEX,WF,DIST(1::K+1);
0078            INTEGER ARRAY INEDGE,OUTEDGE,INLENGTH,OUTLENGTH(1::K,1::MAXIND);
0079
0080         COMMENT GENERATE SYMMETRIC WEIGHTED GRAPHS AS EDGE LISTS **** ;
0081
0082         PROCEDURE  GENEDGE(INTEGER VALUE N,WT; REAL VALUE DENSITY);
0083         COMMENT  N=GRAPH SIZE, WT=MAXIMUM EDGE LENGTH, DENSITY=EDGE DENSITY;
0084         BEGIN
0085         FOR I:=1 STEP 1 UNTIL N DO
0086            ININDEX(I):=OUTINDEX(I):=0;
0087         FOR I:= 1 STEP 1 UNTIL N DO
0088         BEGIN
0089            FOR J:=I+1 STEP 1 UNTIL N DO
0090         IF OUTINDEX(I)=MAXIND THEN GO TO EXED ELSE
0091            IF (ININDEX(J) ¬=MAXIND) AND(RANDOM<DENSITY) THEN
0092            BEGIN
0093            OUTINDEX(I):=ININDEX(I):=OUTINDEX(I) +1;
0094            OUTINDEX(J):=ININDEX(J):=OUTINDEX(J) +1;
0095            OUTEDGE(I,OUTINDEX(I)):=INEDGE(I,ININDEX(I)):=J;
0096            OUTEDGE(J,OUTINDEX(J)):=INEDGE(J,ININDEX(J)):=I;
0097               OUTLENGTH(I,OUTINDEX (I)):=INLENGTH(J,ININDEX(J)):=
0098               OUTLENGTH(J,OUTINDEX (J)):=INLENGTH(I,ININDEX(I)):=
0099                  ENTIER(1+RANDOM*WT);
0100            END;
0101      EXED: ; END;
0102         END GENEDGE;
0103
0104         COMMENT   **** A POSTERIORI ANALYSIS ROUTINES   ****
0105
0106               IN ORDER TO FIND R-OPT, THE OPTIMUM FORWARD RADIUS, WE SOLVE
0107            BY BOTH THE FORWARD AND BACKWARD UNI-DIRECTIONAL SHORTEST PATH
0108            ALGORITHMS. THEN EACH TREE GENERATED IS SORTED BY DISTANCE FROM
0109            THE RESPECTIVE INITIAL NODE.  DLAMBDA THEN COUNTS THE NUMBER OF
0110            NODES THAT A METHOD GOING OUT TO A GIVEN R WOULD SEARCH. A SCAN
0111            OF THIS DISTRIBUTION PRODUCES THE MINIMUM  GIVING R-OPT. THIS A
0112            POSTERIORI ANALYSIS THEN CAN BE USED TO CHECK THE OPTIMALITY
0113            OF A GIVEN STRATEGY.       ;
0114
0115         COMMENT  DLAMBDA  FINDS HOW MANY NODES ARE WITHIN DISTANCE R FROM
0116            THE INITIAL NODE (USE SDIST) OR TERMINAL NODE(USE TDIST). ;
0117
0118         INTEGER PROCEDURE DLAMBDA(INTEGER VALUE R; INTEGER ARRAY SDIST(*));
```

```
0119          BEGIN  INTEGER I;      I:=1;
0120             WHILE R >= SDIST(I) DO I:=I+1;  I
0121          END DLAMBDA;
0122
0123
0124          COMMENT  SORT BUBBLE SORTS THE DISTANCES FOUND  ;
0125
0126          PROCEDURE SORT(INTEGER VALUE N; INTEGER ARRAY DIST,SDIST(*));
0127          BEGIN COMMENT SORTS DIST INTO SDIST IN INCREASING VALUE;
0128             INTEGER T;  LOGICAL FLG;
0129             FLG:=TRUE;
0130             FOR I:=1 STEP 1 UNTIL N DO SDIST(I):=DIST(I);
0131             FOR I:=1 STEP 1 UNTIL N DO
0132                BEGIN  FLG:=FALSE;
0133                FOR J:=1 STEP 1 UNTIL N-I DO
0134                   IF SDIST(J) > SDIST(J+1) THEN
0135                   BEGIN T:=SDIST(J); SDIST(J):=SDIST(J+1); SDIST(J+1):=T;
0136                      FLG:=TRUE   END;
0137                   IF ¬FLG THEN GO TO EXIT;
0138                END;
0139          EXIT:
0140          END SORT;
0141
0142
0143          COMMENT  ****V-ERY G-ENERAL A-LGORITHM   ******
0144
0145                VGA OR WBIED (WEIGHED BI-DIRECTIONAL ALGORITHM) SOLVES
0146             THE TWO POINT SHORTEST PATH PROBLEM. THE DECISION STRATEGY,
0147             STEP TWO OF VGA, IS SELECTED BY DNM. THIS IS THE DECISION
0148             NUMBER FOR DECIDE . DECIDE IS A CASE STATEMENT OF THE CURRENT
0149             STRATEGIES IN USE.      ;
0150
0151          PROCEDURE  WBIED (INTEGER VALUE N,INF,START,TERMINUS,DNM;
0152                INTEGER SCOUNT,TCOUNT,MIND,CNTNODE;
0153                INTEGER ARRAY WF,WT,SDIST,TDIST (*) );
0154          BEGIN
0155
0156             COMMENT   ***GLOBAL PARAMETERS CORRESPOND TO FORMAL PARAMETERS.
0157             THE LOCAL PARAMETERS ARE DESCRIBED BELOW.     ;
0158
0159             INTEGER SMIND,TMIND,TT1,T3,T4,T5;  LOGICAL FLG;
0160             INTEGER TT,NUMSTWD,NUMTTWD,T1C,T2C;  INTEGER ARRAY T1,T2(1::N);
0161             LOGICAL ARRAY SVEC,TVEC,STWDVEC,TTWDVEC(1::N);
0162
0163             COMMENT  **** LOCAL VARIABLES  ******
0164
0165
0166             SMIND=CURRENT MINIMUM DISTANCE IN SET S-TILDA (TMIND)
0167             FLG=LOGICAL FLAG SET TO TRUE IF A NODE APPEARS IN THE
0168                INTERSECTION OF S AND T
0169             NUMSTWD=NUMBER OF NODES IN SET S-TILDA (NUMTTWD)
0170             SVEC=SET MEMBERSHIP FLAG FOR SET S. IF SVEC(I) IS TRUE THEN NODE
0171                I IS IN SET S (TVEC)
0172             STWDVEC=CORRESPONDING LOGICAL ARRAY FOR S-TILDA (TTWDVEC)
0173             ***NOTE*** SVEC AND STWDVEC MAY BOTH BE FALSE FOR A GIVEN NODE,
0174                BUT THEY MAY NOT BOTH BE TRUE .
0175             T1=LIST OF NODES TIED AT MINIMUM DISTANCE IN S-TILDA  (T2)
0176
0177             THE REMAINING VARIABLES ARE TEMPORARIES.    ;
0178
```

```
0179
0180
0181        COMMENT  DECIDE IS THE DECISION STRATEGY FOR VGA. IT IS A CASE
0182           STATEMENT WHERE A SPECIFIC STRATEGY IS SET BY DNM.   ;
0183
0184        LOGICAL PROCEDURE DECIDE;
0185        BEGIN COMMENT DNM SELECTS APPROPRIATE RULE FOR STEP 2 OF VGA;
0186           CASE (DNM) OF
0187           ( NUMSTWD<=NUMTTWD,
0188              SMIND<=TMIND,
0189              TRUE,
0190              FALSE,
0191              NUMTTWD <= NUMSTWD
0192            )
0193
0194           COMMENT      DNM                    RULE
0195                        1              CARDINALITY COMPARISON (POHL)
0196                        2              EQUIDISTANCE (NICHOLSON)
0197                        3              UNI-DIRECTIONAL FORWARD (DIJKSTRA)
0198                        4              UNI-DIRECTIONAL BACKWARD
0199                        5              REVERSE OF 1 DEGENERATES TO 3  ;
0200
0201        END;
0202
0203           COMMENT   BECAUSE OF SEGMENT OVERFLOW PROBLEM THE ALGORITHM IS
0204        DIVIDED INTO INITIAL AND MIDDLE AND FINAL PARTS.  INITIAL IS STEP 1
0205        OF VGA. MIDDLE IS STEPS 2 THROUGH 5, AND FINAL IS STEP 6.
0206           FINAL IS THE LAST PIECE OF CODE IN WBIED AND IS NOT A SEPARATE
0207        PROCEDURE.    ;
0208
0209        PROCEDURE INITIAL;
0210        BEGIN
0211           SMIND:=TMIND:=0;
0212
0213           COMMENT  NODES INITIALLY IN NO SETS  ;
0214
0215           FOR I:=1 STEP 1 UNTIL N DO
0216           BEGIN  SVEC(I):=TVEC(I):=STWDVEC(I):=TTWDVEC(I):=FALSE;
0217              SDIST(I):=TDIST(I):=INF;  WF(I):=WT(I):=-1
0218           END;
0219           TT:=1;  SCOUNT:=TCOUNT:=0;  NUMSTWD:=NUMTTWD:=1;
0220           WF(START):=0;   WT(TERMINUS):=0;
0221
0222           COMMENT INITIALIZE VALUES FOR ENDPOINTS   ;
0223
0224           SDIST(START):=0;   TDIST(TERMINUS):=0;
0225           SVEC(START):=TVEC(TERMINUS):=TRUE;
0226           STWDVEC(START):=TTWDVEC(TERMINUS):=TRUE;
0227           FLG:=FALSE; MIND:=0;
0228
0229           COMMENT    AN IMAGINARY EDGE OF INF LENGTH IS ADDED TO OUR
0230              GRAPH. IF NO OTHER PATH IS FOUND THIS WILL BE PATH OF
0231              MINIMUM DISTANCE.          ;
0232
0233        IF OUTINDEX(START)< MAXIND THEN
0234           BEGIN
0235           OUTINDEX(START):= OUTINDEX(START)+1; OUTEDGE(START,
0236                 OUTINDEX(START)):=TERMINUS;
0237                 OUTLENGTH(START,OUTINDEX(START)):=INF;
0238           END ELSE
```

```
0239            BEGIN  OUTEDGE(START,MAXIND):=TERMINUS; OUTLENGTH(START,MAXIND)
0240                 :=INF;
0241            END;
0242         IF  ININDEX(TERMINUS)< MAXIND THEN
0243            BEGIN
0244             ININDEX(TERMINUS):=ININDEX(TERMINUS)+1;INEDGE(TERMINUS,
0245                  ININDEX(TERMINUS)):=START;
0246                  INLENGTH(TERMINUS,ININDEX(TERMINUS)):=INF;
0247            END ELSE
0248            BEGIN   INEDGE(TERMINUS,MAXIND):=START;  INLENGTH(TERMINUS,MAXIND
0249                 ):=INF;
0250            END;
0251         END INITIAL;
0252
0253            PROCEDURE MIDDLE;  COMMENT  SEGMENT OVERFLOW BYPASS;
0254            WHILE (¬FLG) AND (¬(MIND=INF)) DO
0255            BEGIN
0256               IF DECIDE THEN  COMMENT  DECIDE UPON DIRECTION;
0257               BEGIN
0258                  MIND:=INF;
0259
0260                  COMMENT    THE CURRENT MINIMUM DISTANCE OVER NODES IN S-
0261                     TILDA IS FOUND. TIES ARE STORED IN ARRAY T1 WITH T1C
0262                     THE NUMBER OF TIES.     ;
0263
0264                  FOR I:=1 STEP 1 UNTIL N DO
0265                  IF STWDVEC(I) THEN
0266                  BEGIN
0267                     IF SDIST(I) < MIND THEN
0268                     BEGIN MIND:=SDIST(I); T1C:=1; T1(T1C):=I END
0269                   ELSE IF SDIST(I) = MIND THEN
0270                     BEGIN T1C:=T1C+1; T1(T1C):=I END   ;
0271                  END;
0272
0273                  COMMENT  T1C REPRESENTS THE NUMBER OF NODES TRANSFERRED
0274                     FROM SET S TO SET S-TILDA. THE APPROPRIATE COUNTERS ARE
0275                     CHANGED.       ;
0276
0277                  NUMSTWD:=NUMSTWD-T1C;     SCOUNT:=SCOUNT+T1C;
0278                  SMIND:=MIND;
0279
0280                  COMMENT  APPROPRIATE SET MEMBERSHIP FLAGS ARE SET. EACH
0281                     NODE BEING CHECKED FOR BEING IN S INTERSECTION T.    ;
0282
0283                  FOR I:=1 STEP 1 UNTIL T1C DO
0284                  BEGIN  TT:=T1(I);
0285                  IF ¬FLG THEN BEGIN FLG:=TVEC(TT);TT1:=TT END;
0286                  SVEC(TT):=TRUE;  STWDVEC(TT):=FALSE;
0287                  FOR J:=1 STEP 1 UNTIL OUTINDEX(TT) DO
0288                  BEGIN  T3:=OUTEDGE(TT,J);  T4:=SDIST(T3);
0289                     T5:=OUTLENGTH(TT,J);
0290                     IF T4 > MIND +T5 THEN
0291                     BEGIN  SDIST(T3):=MIND+T5;  WF(T3):=TT;
0292                        IF ¬ STWDVEC(T3) THEN
0293                        BEGIN NUMSTWD:=NUMSTWD+1; STWDVEC(T3):=TRUE END
0294                     END
0295                  END
0296                  END
0297               END
0298               ELSE
```

```
0299            BEGIN
0300
0301               COMMENT  SYMMETRIC TO THE ABOVE LOOP WITH RESPECT TO THE
0302                  BACKWARD DIRECTION.
0303
0304               MIND:=INF;
0305               FOR I:=1 STEP 1 UNTIL N DO
0306               IF TTWDVEC(I) THEN
0307               BEGIN
0308                  IF TDIST(I) < MIND THEN
0309                  BEGIN MIND:=TDIST(I); T2C:=1; T2(T2C):=I END
0310            ELSE IF TDIST(I) = MIND THEN
0311                  BEGIN T2C:=T2C+1; T2(T2C):=I END ;
0312               END;
0313               NUMTTWD:=NUMTTWD-T2C;     TCOUNT:=TCOUNT+T2C;
0314               TMIND:=MIND;
0315               FOR I:=1 STEP 1 UNTIL T2C DO
0316               BEGIN  TT:=T2(I);
0317               IF ¬FLG THEN BEGIN FLG:=SVEC(TT);TT1:=TT END;
0318               TVEC(TT):=TRUE;  TTWDVEC(TT):=FALSE;
0319               FOR J:=1 STEP 1 UNTIL  ININDEX(TT) DO
0320               BEGIN  T3:= INEDGE(TT,J);  T4:=TDIST(T3);
0321                  T5:= INLENGTH(TT,J);
0322                  IF T4 > MIND +T5 THEN
0323                  BEGIN  TDIST(T3):=MIND+T5;  WT(T3):=TT;
0324                     IF ¬ TTWDVEC(T3) THEN
0325                     BEGIN NUMTTWD:=NUMTTWD+1; TTWDVEC(T3):=TRUE END
0326                  END
0327               END
0328               END
0329            END
0330         END;
0331
0332
0333         COMMENT * * * * * * * *   MAIN ROUTINE * * * * * * * *  ;
0334         INITIAL;  MIDDLE;
0335         WRITE("VGA TERMINATED BY NODE ",TT1,"SD=",SDIST(TT1)," TD=",
0336            TDIST(TT1));
0337         WRITE("SMIND=",SMIND,"  TMIND=",TMIND);
0338
0339         COMMENT   S INTERSECTION T-TILDA IS CHECKED AND THE DISTANCE
0340            OF ANY SUCH PATHS ARE COMPUTED AND COMPARED TO MINIMUM.  ;
0341
0342         MIND := SDIST(TT1) + TDIST(TT1);
0343         T1C:=TT1;
0344         FOR I:=1 STEP 1 UNTIL N DO
0345            IF SVEC(I) AND TTWDVEC(I) THEN
0346            BEGIN
0347               T2C:= SDIST(I) + TDIST(I);
0348               IF T2C< MIND THEN  BEGIN MIND:=T2C; T1C:=I END;
0349            END;
0350         CNTNODE:=T1C;
0351      END WBIED;
0352
0353         FOR DENS:=2 DO
0354         BEGIN   COMMENT  LOOP OVER GRAPH DENSITY IN 1/500;
0355            WRITE(" ");
0356            WRITE("MAXIND=",MAXIND,"  SYMMETRIC WITH DENSITY=",DENS/500);
0357            WRITE("TIME TO GENERATE ",TIME(1),"SIZE ",K);
0358            GENEDGE(K,2C,   DENS/500);
```

```
0359                WRITE("TIME TO GENERATE ",TIME(1));
0360                FOR CC:=1 STEP 1 UNTIL 5  DO
0361                BEGIN
0362                COMMENT  *********
0363                   I AND J ARE INITIAL AND TERMINAL NODES SELECTED AT RANDOM;
0364
0365                   I:=ENTIER(K*RANDOM +1);
0366                   J:=ENTIER(K*RANDOM +1);
0367                   WRITE("   ");  WRITE("I,J",I,J); WRITE("   ");
0368                   WBIED(K,INF,I,J,3,SCOUNT,TCOUNT,MIND,CNTNODE,WF,WT,SDIST,
0369                      TDIST);
0370                   WRITE("FORWARD METHOD SCOUNT",SCOUNT,"  MIND=",MIND);
0371                   SORT(K,SDIST,DIST);
0372                   WBIED(K,INF,I,J,4,SCOUNT,TCOUNT,MIND,CNTNODE,WF,WT,SDIST,
0373                      TDIST);
0374                   WRITE("BACKWARD METHOD TCOUNT",TCOUNT,"  MIND=",MIND);
0375                   SORT(K, TDIST, SDIST);
0376                   IF MIND < INF THEN
0377                   BEGIN  WRITE("TABLE OF DISTRIBUTION FUNCTIONS");
0378                      FOR R:=0 STEP 1 UNTIL MIND DO
0379                      BEGIN
0380                         DIST(K+1) := INF + 1;
0381                         SDIST(K+1) := INF + 1;
0382                         DF:=DLAMBDA(R,DIST); DB:=DLAMBDA(MIND-R,SDIST);
0383                         WRITE(R, DF, DB, DB + DF);
0384                      END R;
0385                   END;
0386                   FOR D:= 1, 2  DO
0387                   BEGIN
0388                      WRITE("TIME ENTERED WBIED NO", D,"  TIME ",TIME(1));
0389                      WBIED(K,INF,I,J,D,SCOUNT,TCOUNT,MIND,CNTNODE,WF,WT,
0390                         SDIST,TDIST);
0391                      WRITE("TIME EXITED  WBIED ",TIME(1));
0392                      WRITE("CNTNODE IS ", CNTNODE,"   SCOUNT ",SCOUNT,
0393                         "  TCOUNT ", TCOUNT);
0394                      WRITE("SHORTEST PATH LENGTH IS ",MIND); WRITE("   ");
0395                   END;
0396                END
0397             END DENSLOOP;
0398          END  LOOPK;
0399       END.

ELAPSED TIME IS 00:00:49
```

APPENDIX II

COMPARATIVE RESULTS USING DIFFERENT STRATEGIES IN VGA

In this appendix we list some results of actual computer runs (see Appendix I for the program). Our cardinality comparison strategy is compared to the forward and backward uni-directional strategies and Nicholson's equi-distance bi-directional strategy. The measure of efficiency is the number of nodes at the end of a computation, that have been visited by VGA, i.e., $|S| + |T|$. The data provides a verification of the efficiency of our method and the veracity of our model.

The figures and tables in this appendix present a portion of the computational experience of the author. A graph of given size and edge density was generated randomly, as described above, and two nodes were randomly selected. The shortest path problem between these nodes was solved using VGA for each strategy and the following data collected.

1. length - the shortest path length, which of course is the same for each method
2. $|S|$, $|T|$ - the number of nodes in these sets
3. r_{opt} - the radius the forward method should reach for optimal efficiency
4. r_f, r_b - the forward and backward radii for a bi-directional method.

For each graph 10 different shortest path problems were solved. A graph is characterized by its size, the average degree of its nodes and the maximum length of its edges. In table 5.2, we have already summarized the results of the raw data displayed in tables II.1 through II.5. The headings not previously

GRAPH SIZE = 500
AVERAGE DEGREE = 3
MAXIMUM EDGE LENGTH = 2^

Table II.1

CASE	LENGTH	FORWARD	BACKWARD	POHL	NICHOLSON	POHL F	POHL B	NICHOLSON F	NICHOLSON B	OPT	R (P) F	R (P) B	R (N) F	R (N) B
1	38	142	116	38	38	21	17	21	17	18	19	19	19	19
2	17	25	17	24	27	12	12	18	9	6	10	14	13	12
3	46	279	274	81	68	38	43	25	43	24	26	23	23	23
4	41	230	125	51	59	24	27	38	21	18	18	23	21	20
5	75	437	301	82	171	43	39	140	23	25	26	48	42	42
6	72	363	326	63	54	37	26	37	17	34	37	35	37	36
7	53	281	269	73	65	43	30	43	22	26	25	30	29	28
8	59	373	232	72	77	37	35	59	18	28	27	34	32	31
9	36	92	22	28	29	12	16	24	5	12	16	31	21	22
10	87	441	318	51	74	24	27	62	12	39	34	54	47	46
SUMS														
	524	2583	1991	563	662	291	272	475	187	230	244	315	284	279

$$\sum_{i=1}^{10} \left| \text{length}/2 - r_{opt} \right|_i = 34$$

$$\sum_{i=1}^{10} \left| r_f(P) - r_{opt} \right|_i = 26$$

$$\sum_{i=1}^{10} \left| r_f(N) - r_{opt} \right|_i = 56$$

Table II.2

GRAPH SIZE = 500
AVERAGE DEGREE = 6
MAXIMUM EDGE LENGTH = 2

CASE	LENGTH FORWARD	LENGTH BACKWARD	PERL NICHOLSON PERL F	PERL NICHOLSON PERL B	PERL NICHOLSON NICHOLSON F	PERL NICHOLSON NICHOLSON B	R OPT	R (P) F	R (P) B	R (N) F	R (N) B
1	41	389	457	64	64	30	34	18	46	23	23	18	21	20
2	42	387	460	97	101	48	49	27	74	23	24	18	21	20
3	38	329	203	76	78	39	37	59	19	14	20	25	23	22
4	24	122	165	43	47	24	19	24	23	13	14	12	14	13
5	27	53	114	24	20	11	13	7	13	15	17	14	15	14
6	58	494	164	48	397	23	25	393	4	13	14	46	34	35
7	44	448	414	101	126	49	52	98	30	19	21	27	25	24
8	38	388	412	74	77	42	32	30	47	20	21	18	20	20
9	37	311	431	93	121	47	46	36	85	21	23	18	22	21
10	65	485	491	87	118	37	50	27	91	36	36	30	34	34
SUMS														
	416	3436	3336	707	1151	350	357	719	432	197	213	225	229	223

$$\sum_{i=1}^{10} \left| \text{length}/2 - r_{opt} \right|_i = 38$$

$$\sum_{i=1}^{10} \left| r_f(P) - r_{opt} \right|_i = 16$$

$$\sum_{i=1}^{10} \left| r_f(N) - r_{opt} \right|_i = 44$$

Table II.3

GRAPH SIZE = 500
AVERAGE DEGREE = 6
MAXIMUM ECCE LENGTH = 20

CASE	LENGTH	FORWARD	BACKWARD	POHL	NICHOLSON	POHL F	POHL B	NICHOLSON F	NICHOLSON B	F CPT	R (P) F	R (P) B	R (N) F	R (N) B
1	26	292	488	83	123	38	45	3	120	18	19	10	13	13
2	33	457	454	53	123	24	29	108	15	13	14	19	18	17
3	30	370	496	79	224	36	43	7	217	21	21	10	16	16
4	19	122	429	43	103	19	24	10	93	14	13	6	11	10
5	18	275	247	63	73	32	31	42	31	7	7	8	8	8
6	18	225	100	30	28	11	19	18	10	6	6	10	7	8
7	18	196	211	40	44	30	19	18	26	8	9	7	8	8
8	9	54	5	5	19	2	3	16	3	1	1	8	6	8
9	25	441	487	57	82	48	49	15	67	17	18	13	15	14
10	14	252	7	8	9	3	5	7	2	2	1	13	3	11
SUMS	205	2724	2924	510	828	243	267	244	584	107	109	104	105	113

$$\sum_{i=1}^{10} \left| \text{length}/2 - r_{\text{opt}} \right|_i = 32\tfrac{1}{2}$$

$$\sum_{i=1}^{10} \left| r_F(P) - r_{\text{opt}} \right|_i = 6$$

$$\sum_{i=1}^{10} \left| r_F(N) - r_{\text{opt}} \right|_i = 28$$

Table II.4

GRAPH SIZE = 500
AVERAGE DEGREE = 12
MAXIMUM EDGE LENGTH = 25

CASE	LENGTH	FORWARD	BACKWARD	POHL	NICHOLSON	POHL F	POHL B	NICHOLSON F	NICHOLSON B	R OPT	R (P) F	R (P) B	R (N) F	R (N) B
1	13	311	259	58	51	34	24	34	17	6	7	7	7	6
2	12	191	92	81	56	36	45	68	28	5	8	10	9	9
3	12	268	91	19	31	8	11	26	5	4	4	8	6	6
4	12	276	343	67	66	44	23	26	40	7	8	5	7	6
5	10	246	64	22	40	10	12	34	6	3	3	7	5	5
6	18	469	422	140	129	61	79	95	34	8	8	11	9	9
7	13	370	100	22	73	10	12	66	7	4	4	9	7	8
8	13	52	246	46	51	21	25	9	42	5	11	7	9	8
9	18	355	475	79	147	32	47	29	127	12	11	7	10	9
10	15	159	351	47	58	20	27	12	46	9	10	8	9	9
SUMS														
	136	2627	2434	581	742	276	335	390	352	67	74	79	78	75

$$\sum_{i=1}^{10} \left| \text{length}/2 - r_{opt} \right|_i = 17$$

$$\sum_{i=1}^{10} \left| r_f(P) - r_{opt} \right|_i = 9$$

$$\sum_{i=1}^{10} \left| r_f(N) - r_{opt} \right|_i = 15$$

Table II.5

GRAPH SIZE = 500
AVERAGE DEGREE = 15
MAXIMUM EDGE LENGTH = 20

CASE	LENGTH FORWARD	LENGTH BACKWARD	POHL	NICHOLSON	POHL F	NICHOLSON B	POHL F	NICHOLSON B	R OPT	R (P) F	R (P) B	R (N) F	R (N) B
1	9	34	44	19	13	4	6	7	6	4	4	5	5	5
2	11	366	236	71	55	41	30	65	30	5	5	6	6	6
3	14	352	415	122	122	55	67	55	67	7	8	8	8	8
4	20	488	483	92	91	36	56	57	34	9	9	11	10	10
5	16	445	438	92	93	38	54	56	37	8	8	9	9	8
6	11	132	273	36	62	24	12	24	38	6	7	4	7	6
7	11	371	144	63	78	23	40	66	12	4	4	8	6	5
8	10	125	152	45	33	23	22	11	22	5	6	5	5	5
9	7	45	82	24	24	10	14	10	14	5	4	3	4	3
10	10	123	289	64	70	37	27	26	44	7	7	4	6	5
SUMS	119	2521	2596	619	661	291	328	377	304	60	62	63	66	61

$$\sum_{i=1}^{10} \left| \text{length}/2 - r_{opt} \right|_i = 7\tfrac{1}{2}$$

$$\sum_{i=1}^{10} \left| r_f(P) - r_{opt} \right|_i = 4$$

$$\sum_{i=1}^{10} \left| r_f(N) - r_{opt} \right|_i = 10$$

explained are:

POHLF, POHLB - |S| and |T| respectively for the cardinality comparison strategy.

NICHOLSONF, NICHOLSONB - as above for the equi-distance strategy.

$r_f(P)$, $r_f(N)$ - the forward radii of the POHL and NICHOLSON strategies.

The results are summed for the ten cases in each graph and additionally

(i) $$\sum_{i=1}^{10} \left| \text{length}/2 - r_{opt} \right|_i$$

(ii) $$\sum_{i=1}^{10} \left| r_f(P) - r_{opt} \right|_i$$

(iii) $$\sum_{i=1}^{10} \left| r_f(N) - r_{opt} \right|_i$$

are computed. In the case of the bi-directional methods, these are indications of how well the methods a priori solved Eq. (3.3). The value of (i) reflects the asymmetry of the problems used. Since the graphs generated were undirected and uniformly produced, it is to be expected that the consequent densities are reasonably symmetric. The results for a given problem where this is not so is favorable to cardinality comparison. For example, table II.1, case 5 has

$$\frac{r_{opt}}{\text{length}} = \frac{25}{75} = 1/3$$

where

$$r_f(P) = 28, \qquad r_b(P) = 48$$

and

$$r_f(N) = r_b(N) = 42 \quad .$$

The symmetry assumptions in the equi-distance strategy, inflexibly lead to more work. In all but two cases, (table II.3, case 8 and 10)

$$\left| r_f(N) - r_b(N) \right| \leq 1 .$$

The cardinality comparison strategy tends to equalize the cardinalities of S and T, which is appropriate from considerations of efficiency, as seen in our model.

In Fig. II.1, we plot the comparative data for each method in the 500 node experiments. Similarly in Fig. II.2, we plot results from 150 node experiments with average degree 2 through 16. Table II.6 presents the 500 node data for the forward uni-directional strategy indexed by path length. Table II.7 is the same presentation for the cardinality comparison data. These tables show how number of nodes visited is directly proportional to path length and density, as expected from our model of shortest path space. Overall the results in this appendix confirm our theoretical insights.

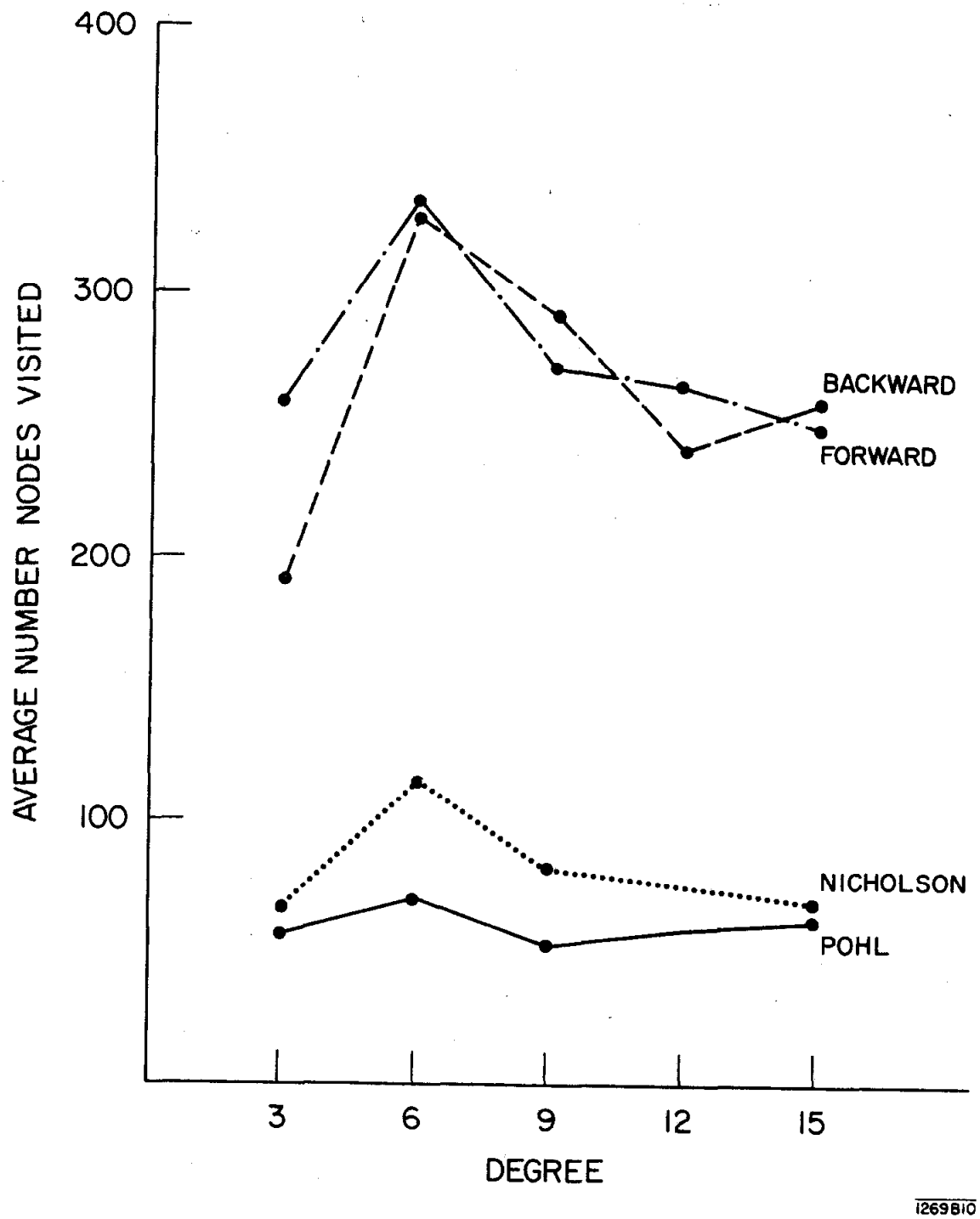


FIG. II.1--Results — 500 node graphs.

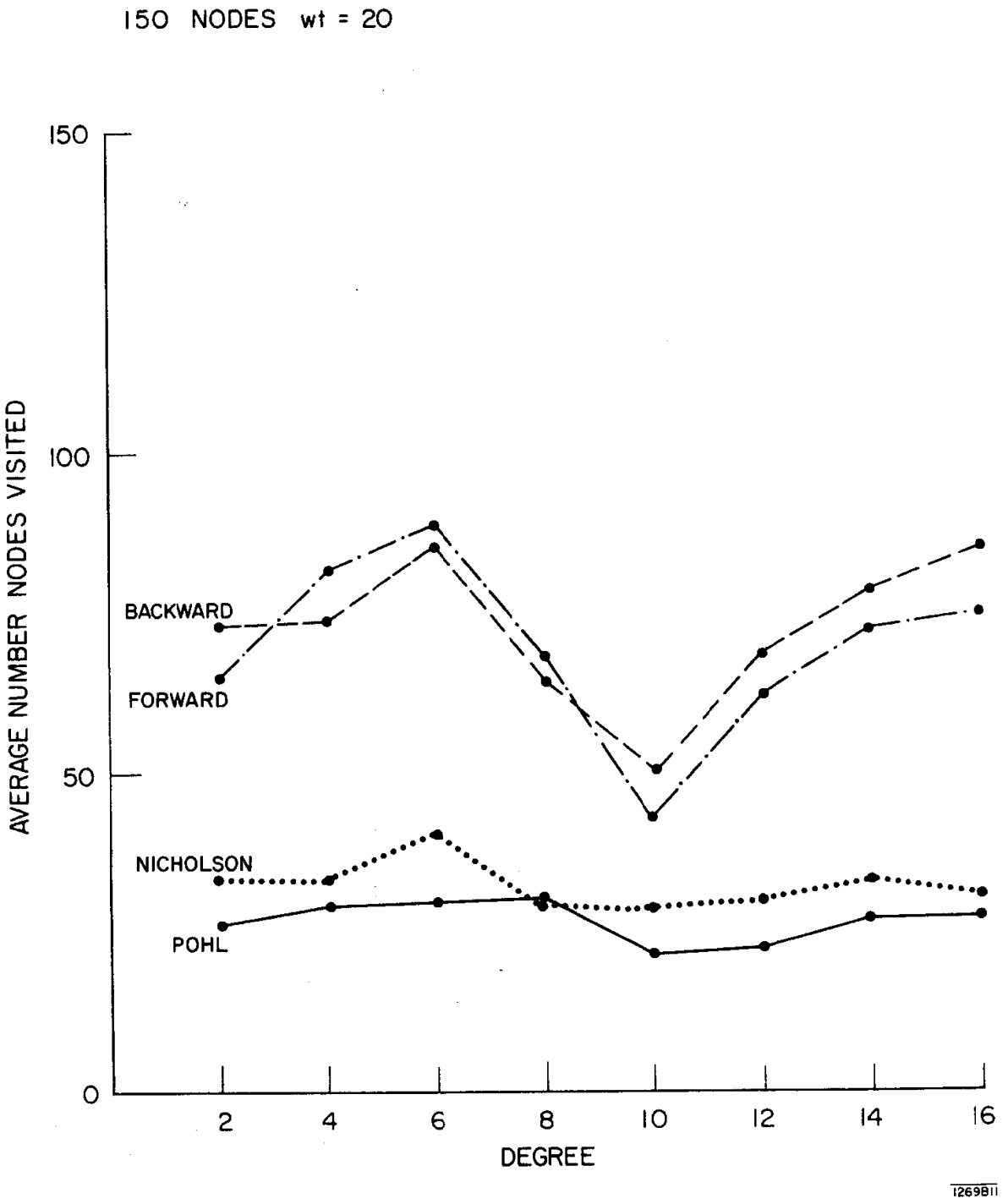


FIG. II.2--Results — 150 node graphs.

Table II.6

FORWARD CROSSTABLE

LENGTH	DEGREE 3	6	9	12	15
0			54		34
					45
10	25		122	311	366
			275	191	392
			225	268	445
			196	206	132
			252	246	371
				469	125
				370	123
				52	
				355	
				159	
20		122	292		488
		53	441		
30	142	329	497		
	92	388	370		
		311			
40	209	389			
	230	387			
		448			
50	281	494			
	373				
60		485			
70	437				
	363				
	431				

Table II.7

POHL CROSSTABLE

LENGTH	DEGREE 3	6	9	12	15
0			5		10 24
10	24		43 63 30 49 8	58 81 19 67 22 140 22 46 79 47	71 122 92 36 63 45 64
20		43 24	83 97		92
30	38 28	76 74 93	53 79		
40	81 51	64 97 101			
50	73 72	48			
60		87			
70	82 63 51				

APPENDIX III

ALGOL W IMPLEMENTATION OF VGHA FOR THE FIFTEEN PUZZLE

This appendix describes the ALGOL W implementation of VGHA as used in experiments with the fifteen puzzle. The basic data representation was a list structure created from records and references in ALGOL W.

```
RECORD    NODE (STRING (17) ENCOD; INTEGER PZ,WF, DIST;
                REAL VALU; REFERENCE (NODE) SPILL)
```

FIELDS:

ENCOD is a string containing the state description. Here it has the values of the sixteen positions.

PZ is the position of the blank tile. It is useful in efficiently generating adjacent positions.

WF is the index to the predecessor node.

DIST is the cardinality distance back to the initial node.

VALU is the value of the evaluation function for this node.

SPILL is the pointer for the hash equivalence class. It is NULL if this is the last node within a given class or else it points to the next member of the class.

The state encoding information could of course be for any other problem domain. This coupled with the successor procedure and the evaluation procedure would be tailored to a specific problem domain.

A short description of the procedures constituting VGHA follows below. In conjunction with the documented listing, this appendix allows a detailed understanding of VGHA.

Debugging and I/O procedures

1. WRTBRD1 - this produces a square array printout of a fifteen puzzle configuration.

2. WRTBRD2 - this produces a string printout and hash value of a fifteen puzzle configuration.

3. DMP - this uses a case statement to printout various parts of the search trees. A variable CND selects the particular dump wanted.

4. TRACE - this prints the solution path found by VGHA.

5. HDISTR - this prints the number of nodes found in each hash equivalence class, i.e. the hash distribution.

Auxiliary routines

1. ENCODE - this takes an array representation BOARD and maps it into a string representation VAR for a given fifteen puzzle configuration.

2. DECOD - this is the inverse of ENCODE.

3. HASH - this uses the array representation BOARD to compute the hash value of a configuration.

4. PTABINIT - this is the PTABLE initializer. The PTABLE is a table lookup for the position value (see Chapters 6 and 8).

5. INITIALIZE - this routine initializes all the necessary flags and arrays to their appropriate value. It also designates the initial node INIT and the terminal node GOAL.

Principal routines

1. SUCCESSOR - this takes a given configuration and generates the neighboring configurations in NXNODES.

2. EVALUATE - this evaluates the board configuration provided by SUCCESSOR. This is a case statement incorporating the various evaluation functions to be tested.

3. NONREDUNDANT - this uses the hash equivalence class of a node to conduct a linked list search for redundancy.

4. DECIDE - this is the step that decides which direction the search should take, either forward (DECIDE:=TRUE) or backward (DECIDE:=FALSE).

5. TERMINATE - this uses the hash classes to see if a given node is in both the forward and backward search trees.

The basic iteration step is coded symmetrically for the forward and backward search. The backward search corresponding to DECIDE=FALSE has its variable identifiers prefixed by B.

```
%ALGOL    3:55,9000

0001          BEGIN  COMMENT  I.POHL  OCT 1968 (SLAC)
0002
0003                   ALGOL W IMPLEMENTATION OF VGHA-VERY GENERAL HEURISTIC
0004                   ALGORITHM. THIS VERSION SOLVES THE FIFTEEN PUZZLE.
0005
0006               PARAMETERS : **************;
0007
0008          INTEGER MAXITER,NUMNODE,BNUMNODE,CURNOD,DEG,T1,T2,BCURNOD;
0009          INTEGER  FUSE, BUSE, DECN, EVALN, DMPN, CSN, SDIST, TDIST;
0010          INTEGER PEX,PEX1,J, T5, TIN;
0011          LOGICAL    FLG,TFLAG;    REAL WT,VAL,MIN;
0012          LOGICAL ARRAY TP(0::15);    REAL ARRAY WW(1::10);
0013          LOGICAL ARRAY UNDEVELOPED,BUNDEVELOPED (1::3000);
0014          INTEGER ARRAY MAP,MAP1,BCARD,BCARD1(0::15);  INTEGER ARRAY NZ(1::4);
0015          STRING (16) GOAL,INIT,ST1,ST2;
0016          STRING (16) ARRAY NXNODES(1::4);  INTEGER ARRAY PTABLE(0::15,0::15);
0017          RECORD  NODE(STRING (16) ENCOD; INTEGER PZ,WF,DIST; REAL VALU;
0018                       REFERENCE (NODE) SPILL);
0019          REFERENCE (NODE) ARRAY PT(1::3000);
0020          REFERENCE (NODE) ARRAY BPT(1::3000);
0021          REFERENCE (NODE) ARRAY HSH(0::800);
0022          REFERENCE (NODE) ARRAY BHSH(0::800);
0023          REFERENCE (NODE) P1, P2;
0024
0025          COMMENT    *****  DEFINITION OF IMPORTANT VARIABLES  *****
0026
0027             VARIABLES PREFIXED BY B ARE BACKWARD DIRECTION VARIABLES.
0028
0029             NODE = RECORD REPRESENTATING ONE NODE AND ASSOCIATED
0030                STATE INFORMATION.
0031                   ENCOD = STRING FIELD FOR 15 PUZZLE CONFIGURATION.
0032                   PZ = INTEGER FIELD NOTING POSITION OF THE BLANK.
0033                   WF = INDEX TO PREDECESSOR NODE.
0034                   DIST = CARDINALITY DISTANCE TO ENDPOINT.
0035                   VALU = THE VALUE OF THE EVALUATION FUNCTION.
0036                   SPILL = POINTER TO NEXT NODE IN HASH EQUIVALENCE CLASS.
0037             PT (BPT) = REFERENCE ARRAYS POINTING TO FORWARD(BACKWARD)
0038                SEARCH TREES.
0039             HSH (BHSH) = POINTERS TO INITIAL NODE IN EACH HASH CLASS.
0040             CURNOD (BCURNOD) = INDEX INTO REFERENCE ARRAY PT (BPT).
0041                PT(CURNOD) POINTS TO THE CURRENT NODE BEING EXPANDED.
0042             INIT = STRING ENCODING OF STARTING 15 PUZZLE CONFIGURATION.
0043             GOAL = TERMINATING OR GOAL CONFIGURATION.
0044             NXNODES = ARRAY OF SUCCESSOR COFIGURATIONS OF CURNOD(BCURNOD).
0045             MAXITER = MAXIMUM NUMBER OF NODES EXPANDED BEFORE SEARCH IS ENDED
0046             NUMNODE(BNUMNODE) = NUMBER OF FORWARD (BACKWARD) NODES SEARCHED.
0047             DECN = PICKS DECISION STRATEGY IN DECIDE.
0048             EVALN = PICKS EVALUATOR IN EVALUATE.
0049             DMPN = PICKS DUMP ROUTINE IN DMP.
0050             WT = WEIGHT OF HEURISTIC FUNCTION VERSUS DIST IN EVALUATOR.
0051             UNDEVELOPED(BUNDEVELOPED) = IS TRUE IF NODE IS NOT YET EXPANDED.
0052
0053             OTHER VARIABLES ARE TEMPORARIES OR HAVE SPECIAL FUNCTIONS;
0054
0055
0056
0057
0058
```

```
0059            COMMENT  ENCODE TAKES INTEGER ARRAY REPRESENTATION OF A 15
0060              PUZZLE CONFIGURATION -BOARD AND ENCODES IN STRING FORM-VAR;
0061
0062          PROCEDURE ENCODE(INTEGER ARRAY BOARD(*); STRING (16) VAR);
0063             COMMENT USES ALGOLW IMPLICIT PROCEDURE  CODE;
0064             FOR I:=0 STEP 1 UNTIL 15 DO   VAR(I|1):= CODE(BOARD(I));
0065
0066             COMMENT  INVERSE PROCEDURE FOR ENCODE;
0067
0068          PROCEDURE DECOD (STRING (16) VAR; INTEGER ARRAY BOARD(*));
0069             COMMENT  USES INVERSE OF CODE -DECODE;
0070             FOR I:=0 STEP 1 UNTIL 15 DO   BOARD(I):= DECODE(VAR(I|1));
0071
0072
0073
0074
0075             COMMENT  NONREDUNDANT CHECKS TO SEE IF THE 15 PUZZLE CONFIGURA-
0076               TION VAR HAS ALREADY BEEN FOUND.  ;
0077
0078          LOGICAL PROCEDURE  NONREDUNDANT( STRING (16) VAR; INTEGER H1;
0079                                           LOGICAL FLG);
0080             BEGIN COMMENT
0081                      VAR= STATE BEING CHECKED
0082                      H1=HASH INDEX OF VAR
0083                      FLG = WHICH TREE IS SEARCHED FOR REDUNDANT NODE.
0084                        IF TRUE THEN FORWARD TREE ELSE BACKWARD TREE. ;
0085               LOGICAL T ;
0086               DECOD (VAR,BOARD); H1:=HASH(BOARD);
0087               P1:= (IF FLG THEN HSH(H1) ELSE BHSH(H1));  T :=TRUE;
0088               COMMENT CHAINED SEARCH ;
0089               WHILE P1 ¬= NULL DO
0090                  IF ENCOD(P1)=VAR THEN
0091                   COMMENT NODE ALREADY EXISTS IN HASH CLASS;
0092                     BEGIN  T:=FALSE; GOTO OUT END
0093                  ELSE        P1:=SPILL(P1);
0094          OUT:    T
0095             END NONREDUNDANT;
0096
0097
0098
0099
0100
0101
0102             COMMENT SUCCESSOR FINDS NODES ADJACENT TO VAR. THE NUMBER
0103               IT FINDS IS DEG ,AND THEY ARE STORED IN NXNODES WITH THE ZERO
0104               POSITIONS RECORDED IN NZ;
0105
0106          PROCEDURE SUCCESSOR( STRING (16) VAR; INTEGER DEG,PZ;
0107             INTEGER ARRAY NZ(*); STRING (16) ARRAY NXNODES(*));
0108             BEGIN
0109                LOGICAL L,R,U,D;  INTEGER  A1;
0110
0111             COMMENT L,R,U,D  ARE LEFT,RIGHT,UP,DOWN FLAGS RESPECTIVELY.THEY
0112                TELL NXT WHICH OF FOUR POSSIBLE MOVES TO GENERATE. ;
0113
0114                PROCEDURE NXT;
0115                   BEGIN  A1:=0;
0116                      IF R THEN
0117                        BEGIN   A1:=1;
0118                           NXNODES(1):=VAR; NXNODES(1)(PZ|1):=VAR(PZ+1|1);
```

```
0119                        NXNODES(1)(PZ+1|1):=CODE(0);  NZ(1):=PZ+1;
0120                      END RIGHTMOVE;
0121                    IF L THEN
0122                      BEGIN  A1:=A1+1;
0123                        NXNODES(A1):=VAR; NXNODES(A1)(PZ|1):=VAR(PZ-1|1);
0124                        NXNODES(A1)(PZ-1|1):=CODE(0);  NZ(A1):=PZ-1;
0125                      END LEFTMOVE;
0126                    IF U THEN
0127                      BEGIN  A1:=A1+1;
0128                        NXNODES(A1):=VAR; NXNODES(A1)(PZ|1):=VAR(PZ-4|1);
0129                        NXNODES(A1)(PZ-4|1):=CODE(0);  NZ(A1):=PZ-4;
0130                      END UPMOVE;
0131                    IF D THEN
0132                      BEGIN  A1:=A1+1;
0133                        NXNODES(A1):=VAR; NXNODES(A1)(PZ|1):=VAR(PZ+4|1);
0134                        NXNODES(A1)(PZ+4|1):=CODE(0); NZ(A1):=PZ+4;
0135                      END DOWNMOVE;
0136                END NXT;
0137
0138              COMMENT SUCCESSOR USES THE PZ=POSITION OF THE ZERO IN VAR TO FIND
0139                 NEW BOARD POSITIONS;
0140                 CASE PZ+1 OF
0141                 BEGIN
0142                    BEGIN  R:=D:=TRUE; L:=U:=FALSE; DEG:=2; NXT END;
0143                    BEGIN  R:=L:=D:=TRUE; U:=FALSE; DEG:=3; NXT END;
0144                    BEGIN  R:=L:=D:=TRUE; U:=FALSE; DEG:=3; NXT END;
0145                    BEGIN  L:=D:=TRUE; R:=U:=FALSE; DEG:=2; NXT END;
0146                    BEGIN  R:=U:=D:=TRUE; L:=FALSE; DEG:=3; NXT END;
0147                    BEGIN  R:=L:=U:=D:=TRUE;        DEG:=4; NXT END;
0148                    BEGIN  R:=L:=U:=D:=TRUE;        DEG:=4; NXT END;
0149                    BEGIN  L:=U:=D:=TRUE; R:=FALSE; DEG:=3; NXT END;
0150                    BEGIN  R:=U:=D:=TRUE; L:=FALSE; DEG:=3; NXT END;
0151                    BEGIN  R:=L:=U:=D:=TRUE;        DEG:=4; NXT END;
0152                    BEGIN  R:=L:=U:=D:=TRUE;        DEG:=4; NXT END;
0153                    BEGIN  L:=U:=D:=TRUE; R:=FALSE; DEG:=3; NXT END;
0154                    BEGIN  R:=U:=TRUE; L:=D:=FALSE; DEG:=2; NXT END;
0155                    BEGIN  L:=R:=U:=TRUE; D:=FALSE; DEG:=3; NXT END;
0156                    BEGIN  L:=R:=U:=TRUE; D:=FALSE; DEG:=3; NXT END;
0157                    BEGIN  L:=U:=TRUE; R:=D:=FALSE; DEG:=2; NXT END;
0158                 END MOVES;
0159              END SUCCESSOR;
0160
0161
0162
0163
0164           INTEGER PROCEDURE HASH(INTEGER ARRAY BOARD(*));
0165              BEGIN  COMMENT HASHES A BOARD POSITION  INTO  560  TO  1240;
0166                 INTEGER T; T:=0;
0167                 FOR I:=0 STEP 1 UNTIL 15 DO  T:=T+BOARD(I)*I;
0168                 (T-560)
0169              END HASH;
0170
0171
0172
0173
0174
0175
0176
0177
0178
```

```
0179
0180            COMMENT EVALUATE POSITION VAR RESULT IS VAL. EVALN SELECTS
0181               PARTICULAR EVALUATION FUNCTION AND FLG IS USED WHEN DIRECTION-
0182               AL INFORMATION IS WANTED.     ;
0183
0184         PROCEDURE  EVALUATE(STRING (16) VAR; INTEGER PZ; REAL VAL,WT;
0185                    INTEGER EVALN; LOGICAL FLG);
0186            BEGIN
0187               INTEGER P,T,PEX,PEX1,PEX2,R;  REAL S;
0188               INTEGER ARRAY  SEQ,PP,BT,HB (0::15);
0189
0190            COMMENT R IS THE REVERSAL COUNT;
0191            INTEGER PROCEDURE REVERSALS;
0192            BEGIN
0193            R:=0;
0194            FOR J:=0 STEP 4 UNTIL 12 DO
0195               FOR I:=0 STEP 1 UNTIL 2 DO
0196                  IF BOARD(I+J)= I + J +2 THEN IF BOARD(I+J+1)=I+J+1 THEN
0197                     R:=R+1;
0198
0199            FOR I:=0 STEP 1 UNTIL 3 DO
0200               FOR J:=0 STEP 4 UNTIL 8 DO
0201                  IF BOARD(I+J)=I+J+5 THEN IF BOARD(I+J+4)=I+J+1 THEN
0202                     R:=R+1;
0203            ( R )
0204            END REVERSALS;
0205
0206            COMMENT DORAN-MICHIE EVALUATOR WITHOUT REVERSAL TERM;
0207            REAL PROCEDURE MDWR;
0208            BEGIN
0209               FOR I:=0 STEP 1 UNTIL 14 DO     HB(I):=PTABLE(PZ,I+1);
0210               HB(15):=PTABLE(PZ,15)+(IF (PZ+1) REM 4=0 THEN -1 ELSE 1);
0211               S:=0;
0212               FOR I:=0 STEP 1 UNTIL 15 DO
0213                  S:=S + SQRT(HB(I))*PP(I)*PP(I);
0214            ( S )
0215            END MDWR;
0216
0217            COMMENT MDWR WITH MY REVERSAL TERM ;
0218            REAL PROCEDURE MD;       (MDWR+20*REVERSALS) ;
0219
0220            COMMENT IF BACKWARD DIRECTION THEN POSITION IS MAPPED INTO
0221               THE ANTI-SYMMETRIC CONFIGURATION ALLOWING EVALUATE TO
0222               TREAT IT NORMALLY. ;
0223               IF ¬FLG THEN
0224                  BEGIN   DECOD (VAR,BOARD1);
0225                     FOR I:=0 STEP 1 UNTIL 15 DO  BCARD(I):=MAP(BOARD1(I));
0226                  END
0227               ELSE
0228                  BEGIN   CECOD (VAR,BOARD1);
0229                     FOR I:=0 STEP 1 UNTIL 15 DO  BCARD(I):=MAP1(BOARD1(I));
0230                  END;
0231
0232               PEX:=PEX1:=P:=0;
0233
0234               FOR I:=0 STEP 1 UNTIL 15 DO
0235               BEGIN  T:=BOARD(I);
0236                  BT(T):=PP(I):=PTABLE(I,T);
0237                  P:=P+PP(I);
0238               END;
```

```
0239
0240          COMMENT  THE FOLLOWING VARIABLES ARE NOT CURRENTLY BEING USED.
0241             PEX AND PEX1 REPRESENT PARTIAL P EVALUATION AND ARE USED IN
0242             SOME BI-DIRECTIONAL INTERSECTION EXPERIMENTS. THEY ARE TURNED
0243             OFF FOR PURPOSES OF EFFICIENCY. ;
0244           COMMENT TURN OFF
0245                FOR I:=0 STEP 1 UNTIL 15 DO
0246                BEGIN
0247                   IF FLG THEN BEGIN IF TP[I] THEN PEX1:=PEX1+BT[I] END
0248                   ELSE BEGIN IF ¬TP[I] THEN PEX1:=PEX1+BT[I] END
0249                END;
0250
0251                COMMENT PEX SCORE  REORDER 1/2 OF BOARD;
0252          COMMENT  TURN OFF
0253                BEGIN FOR I:=0 STEP 1 UNTIL 7 DO PEX:=PEX+BT[I] END;
0254
0255             CASE EVALN OF
0256             BEGIN
0257                VAL:=(IF FLG THEN DIST(PT(CURNOD)) ELSE DIST(BPT(BCURNOD)))+
0258                   WT*P+1;
0259                VAL:=(IF FLG THEN DIST(PT(CURNOD)) ELSE DIST(BPT(BCURNOD)))+
0260                   WT*P+1+20*REVERSALS;
0261                VAL:=(IF FLG THEN DIST(PT(CURNOD)) ELSE DIST(BPT(BCURNOD)))
0262                   +WT*MDWR;
0263                VAL:=(IF FLG THEN DIST(PT(CURNOD)) ELSE DIST(BPT(BCURNOD)))
0264                   +WT*MD;
0265                VAL:=P;
0266                VAL:=P+20*REVERSALS;
0267                VAL:=MDWR;
0268                VAL:=MD;
0269                VAL:=(IF FLG THEN DIST(PT(CURNOD)) ELSE DIST(BPT(BCURNOD)))+1;
0270                VAL:=(IF FLG THEN DIST(PT(CURNOD)) ELSE DIST(BPT(BCURNOD)))+P
0271                   +PEX +1;
0272                VAL:=(IF FLG THEN DIST(PT(CURNOD)) ELSE DIST(BPT(BCURNOD)))+P
0273                   +P+2*PEX1+1;
0274                VAL:=(IF FLG THEN DIST(PT(CURNOD)) ELSE DIST(BPT(BCURNOD)))+P
0275                   +PEX+PEX1+1;
0276                VAL:=0;
0277             END;
0278             END EVALUATE;
0279
0280
0281          PROCEDURE WRTBRD2( STRING (16) VAR);  COMMENT LINEAR REPRESENTATION;
0282          BEGIN
0283             INTFIELDSIZE:= 3;
0284             WRITE("  ");  DECOD(VAR, BOARD);
0285             FOR I:=0 UNTIL 15 DO WRITEON(BOARD[I]);
0286             WRITE["HASH VALUE  ", HASH(BCARD)];
0287             INTFIELDSIZE:=12;
0288          END WRTBRD2;
0289
0290
0291          PROCEDURE WRTBRD1( STRING (16) VAR);
0292             COMMENT WRITES OUT 4 BY 4 ARRAY REPRESENTATION OF FIFTEEN PUZZLE;
0293          BEGIN
0294             INTFIELDSIZE:= 3;        WRITE("  ");  DECOD(VAR, BCARD);
0295             FOR I:=0 UNTIL 3 DO WRITEON(BOARD[I]);  WRITE("  ");
0296             FOR I:=4 UNTIL 7 DO WRITEON(BOARD[I]);  WRITE("  ");
0297             FOR I:=12 UNTIL 15 DO WRITEON(BOARD[I]);   INTFIELDSIZE:=12;
0298          END WRTBRD1;
```

```
0299
0300
0301
0302            COMMENT HDISTR COLLECTS AND PRINTS HASH CLASS DISTRIBUTION.;
0303
0304          PROCEDURE HDISTR;
0305          BEGIN
0306             INTEGER ARRAY NH(0::680);
0307             WRITE("HASH TABLE NUMBER OF OCCURRENCES");
0308             FOR I:=0 STEP 1 UNTIL 680 DO NH(I):=0;
0309             FOR I:=1 UNTIL NUMNODE DO
0310             BEGIN
0311                DECOD(ENCOD(PT(I)), BOARD);  T1:=HASH(BOARD);
0312                NH(T1):= NH(T1) + 1;
0313             END;
0314             INTFIELDSIZE:=3;
0315             FOR J:=0 STEP 10 UNTIL 670 DO
0316             BEGIN   WRITE(J);
0317                FOR I:=0 UNTIL 9 DO WRITEON( NH(I+J) );
0318             END;
0319          END HDISTR;
0320
0321
0322             COMMENT DMP DUMPS VARIOUS NODES OF THE SEARCH TREES;
0323
0324          PROCEDURE DMP(INTEGER CND);
0325             CASE CND OF
0326             BEGIN
0327             COMMENT  CND = 1 :FULL FORWARD AND BACKWARD TREES  ;
0328               BEGIN
0329                FOR I:=1 STEP 1 UNTIL NUMNODE DO
0330                BEGIN
0331                   WRTBRD2(ENCOD(PT(I))  );
0332                   WRITE( WF(PT(I)), VALU(PT(I)),DIST(PT(I)) );
0333                END;
0334                FOR I:=1 STEP 1 UNTIL BNUMNODE DO
0335                BEGIN
0336                   WRTBRD2(ENCOD(BPT(I))  );
0337                   WRITE( WF(BPT(I)), VALU(BPT(I)),DIST(BPT(I)) );
0338                END;
0339               END;
0340
0341             COMMENT  CND =2: FULL FORWARD TREE;
0342                FOR I:=1 STEP 1 UNTIL NUMNODE DO
0343                BEGIN
0344                   WRTBRD2(ENCOD(PT(I))  );
0345                   WRITE( WF(PT(I)), VALU(PT(I)),DIST(PT(I)) );
0346                END;
0347
0348             COMMENT CND = 3: FULL BACKWARD TREE ;
0349                FOR I:=1 STEP 1 UNTIL BNUMNODE DO
0350                BEGIN
0351                   WRTBRD2(ENCOD(BPT(I))  );
0352                   WRITE( WF(BPT(I)), VALU(BPT(I)),DIST(BPT(I)) );
0353                END;
0354
0355             COMMENT CND = 4: EVERY TENTH NODE IN FORWARD TREE;
0356                FOR I:=1 STEP 10 UNTIL NUMNODE DO
0357                BEGIN
0358                   WRTBRD2(ENCOD(PT(I))  );
```

```
0359                WRITE( WF(PT(I)), VALU(PT(I)),DIST(PT(I)) );
0360              END;
0361          ; COMMENT CASE 5 IS NULL;
0362
0363            COMMENT CND = 6: LAST TEN NODES IN BOTH FORWARD AND BACKWARD
0364               TREES;
0365              BEGIN
0366               INTEGER TF,TB;
0367               IF  NUMNODE<11 THEN TF:=1 ELSE TF:=NUMNODE-10;
0368               IF BNUMNODE<11 THEN TB:=1 ELSE TB:=BNUMNODE-10;
0369               FOR I:=NUMNODE STEP -1 UNTIL TF DO
0370               BEGIN
0371                  WRTBRD2(ENCOD(PT(I))  );
0372                  WRITE( WF(PT(I)), VALU(PT(I)),DIST(PT(I)) );
0373               END;
0374               FOR I:=BNUMNODE STEP -1 UNTIL TB DO
0375               BEGIN
0376                  WRTBRD2(ENCOD(BPT(I))  );
0377                  WRITE( WF(BPT(I)), VALU(BPT(I)),DIST(BPT(I)) );
0378               END;
0379              END;
0380            END;
0381
0382
0383            COMMENT TRACE PRINTS OUT THE SOLUTION PATH;
0384
0385          PROCEDURE TRACE;  COMMENT TRACES PATH USES T1,T2 AS EXTERNALLY
0386             SUPPLIED STARTING POINTS FROM PROCEDURE TERMINATE;
0387             BEGIN
0388                INTEGER COUNT;
0389                COUNT:=0;
0390             COMMENT FORWARD DIRECTION STARTS WITH NODE T1. USES WF TO CHAIN
0391                THROUGH PATH UNTIL 0 IS ENCOUNTERED AT NODE INIT;
0392                WHILE T1¬=0 DO
0393                   BEGIN
0394                      P1:=PT(T1);
0395                      WRITE("NODE   ", "  TREE POS  ", T1," VAL  ",VALU(P1));
0396                      WRTBRD1(ENCOD(P1));
0397                      T1:=WF(P1);
0398                      COUNT:=COUNT+1;
0399                   END;
0400                WRITE("FORWARD TREE NODES ",NUMNODE,"   PATH IS ",COUNT);
0401                COUNT:=0;
0402             COMMENT BACKWARD DIRECTION STARTS WITH NODE T2. USES WF TO CHAIN
0403                THROUGH PATH UNTIL 0 IS ENCOUNTERED AT NODE GOAL;
0404                WHILE T2¬=0 DO
0405                   BEGIN
0406                      P2:=BPT(T2);
0407                      WRITE("NODE   ", "  TREE POS  ", T2," VAL  ",VALU(P2));
0408                      WRTBRD1(ENCOD(P2));
0409                      T2:=WF(P2);
0410                      COUNT:=COUNT+1;
0411                   END;
0412                WRITE("BACKWARD TREE NODES ",BNUMNODE,"   PATH IS ",COUNT);
0413             END TRACE;
0414
0415
0416
0417
0418
```

```
0419
0420          COMMENT     ***** PTABLE INITIALIZATION *****
0421            PTABLE(POSITION, TILE VALUE) EQUALS MANHATTAN DISTANCE FROM
0422            TILE TO ITS GOAL SQUARE     ;
0423
0424          PROCEDURE PTABINIT;
0425          BEGIN
0426             FOR I:=0 UNTIL 14 DO
0427                  PTABLE(I,0):=PTABLE(I,I+1):=0;
0428             PTABLE(15,0):=0;
0429             PTABLE(0,2):=PTABLE(0,5):=PTABLE(1,3):=PTABLE(1,6):=PTABLE(2,4):=
0430             PTABLE(2,7):=PTABLE(3,8):=PTABLE(4,6):=PTABLE(4,9):=PTABLE(5,7)
0431                :=1;
0432             PTABLE(0,3):=PTABLE(0,6):=PTABLE(0,9):=PTABLE(1,4):=PTABLE(1,5):=
0433             PTABLE(1,7):=PTABLE(1,10):=PTABLE(2,6):=PTABLE(2,8):=2;
0434             PTABLE(0,4):=PTABLE(0,7):=PTABLE(0,10):=PTABLE(0,13):=
0435             PTABLE(1,8):=PTABLE(1,11):=PTABLE(1,14):=PTABLE(2,5):=3;
0436             PTABLE(0,8):=PTABLE(0,11):=PTABLE(0,14):=PTABLE(1,12):=
0437             PTABLE(1,13):=PTABLE(1,15):=PTABLE(2,9):=PTABLE(2,14):=
0438             PTABLE(3,5):=4;
0439             PTABLE(0,12):=PTABLE(0,15):=PTABLE(2,13):=PTABLE(3,9):=
0440             PTABLE(3,14):=5;
0441             PTABLE(5,10):=PTABLE(6,8):=PTABLE(6,11):=PTABLE(7,12):=
0442             PTABLE(8,13):=PTABLE( 9,11):=PTABLE( 9,14):=PTABLE(10,12):=
0443             PTABLE(8,10):=1;
0444             PTABLE(2,11):=PTABLE(3,7):=PTABLE(3,12):=PTABLE(4,7):=
0445             PTABLE(4,13):=PTABLE(5,8):=PTABLE(5,9):=PTABLE(5,11):=
0446             PTABLE(4,10):=2;
0447             PTABLE(2,10):=PTABLE(2,12):=PTABLE(2,15):=PTABLE(3,6):=
0448             PTABLE(4,8):=PTABLE(4,11):=PTABLE(4,14):=PTABLE(5,12):=
0449             PTABLE(3,11):=3;
0450             PTABLE(3,10):=PTABLE(3,15):=PTABLE(4,12):=PTABLE(4,15):=
0451             PTABLE(7,9):=PTABLE(7,14):=PTABLE(6,13):=4;
0452             PTABLE(3,13):=6;
0453             PTABLE(5,14):=PTABLE(6,10):=PTABLE(6,12):=PTABLE(6,15):=
0454             PTABLE(8,11):=PTABLE(8,14):=PTABLE( 9,12):=PTABLE( 9,13):=
0455             PTABLE(7,11):=2;
0456             PTABLE(5,13):=PTABLE(5,15):=PTABLE(6,9):=PTABLE(6,14):=
0457             PTABLE(7,15):=PTABLE(8,12):=PTABLE(8,15):=PTABLE(7,10):=3;
0458             PTABLE( 9,15):=PTABLE(10,14):=PTABLE(11,15):=PTABLE(12,15):=2;
0459             PTABLE(10,15):=PTABLE(12,14):=PTABLE(13,15):=1;
0460             PTABLE(1,9):=PTABLE(10,13):=PTABLE(11,14):=3;
0461             PTABLE(11,13):=4;  PTABLE(7,13):=5;
0462             FOR I:=0 UNTIL 14 DO
0463                FOR J:=I+2 UNTIL 15 DO PTABLE(J-1,I+1):=PTABLE(I,J);
0464             PTABLE(15,1):=6; PTABLE(15,2):=PTABLE(15,5):=5;PTABLE(15,15):=1;
0465             PTABLE(15,3):=PTABLE(15,6):=PTABLE(15,9):=4;  PTABLE(15,12):=1;
0466             PTABLE(15,4):=PTABLE(15,7):=PTABLE(15,10):=PTABLE(15,13):=3;
0467             PTABLE(15,8):=PTABLE(15,11):=PTABLE(15,14):=2;
0468          END PTABINIT;
0469
0470          PROCEDURE INITIALIZE;    COMMENT   SEGMENT OVERFLOW  ;
0471          BEGIN
0472          COMMENT    * INITIALIZATION OF PARAMETERS   *          ;
0473             UNDEVELOPED(1):=BUNDEVELOPED(1):= TRUE;
0474              FOR I:=0 STEP 1 UNTIL 800  DO HSH(I):=BHSH(I):=NULL;
0475             NUMNODE:=BNUMNODE:=1;  FUSE:=BUSE:=0; SDIST:=TDIST:=0;
0476             FOR I:=0 STEP 1 UNTIL 14 DO  BOARD(I):=I+1; BOARD(15):=0;
0477             ENCODE(BOARD,GOAL);
0478             FOR I:=0 UNTIL 14 DO MAP1(BOARD(I)):=I+1;  MAP1(BOARD(15)):=0;
```

```
0479            DECOD(INIT,BOARD);
0480            FOR I:=0 UNTIL 14 DO      MAP(BOARD(I)):=I+1;   MAP(BOARD(15)):=0;
0481            COMMENT     INITIAL EVAL IS MEANINGLESS AND IS SET TO 0;
0482            VAL:=0;  PT(1):=NODE(INIT,T5,0,0,VAL,NULL);
0483            BPT(1):=NODE(GOAL,15,0,0,VAL,NULL);
0484            DECOD (INIT,BOARD);
0485            HSH(HASH(BOARD)):=PT(1);   COMMENT INITIAL HASH CLASS;
0486            DECOD (GOAL,BOARD);
0487            BHSH(HASH(BOARD)):=BPT(1);
0488          END INITIALIZE;
0489
0490
0491          PTABINIT;
0492
0493          COMMENT  WW IS AN ARRAY OF WEIGHTS TO BE USED WITH EACH FUNCTION;
0494          WW(1):=0.5;  WW(2):=0.75;  WW(3):=1;  WW(4):=1.5;  WW(5):=2;
0495          WW(6):=3;  WW(7):=4;  WW(8):=16;
0496          FOR I:=0 STEP 1 UNTIL 15 DO TP(I):=FALSE;
0497          FOR I:=0,1,2,3,4,8,12 DO TP(I):=TRUE;
0498
0499        OVER:
0500          COMMENT  READ IN PARAMETERS  ;
0501
0502          READ(MAXITER,DECN,EVALN,DPPN,CSN,WT);
0503          FOR I:=0 STEP 1 UNTIL 15 DO
0504           BEGIN  READCN(BOARD(I));  IF BOARD(I)=0 THEN T5:=I  END;
0505          ENCODE(BOARD,INIT);
0506          IOCONTROL(3);
0507          WRITE("   BI-DIRECTIONAL GRAPH TRAVERSER WITH GRAPHS IN RECORDS");
0508          WRITE("CASE ",CSN,"  DECN ",DECN,"  EVALN  ",EVALN);
0509          WRITE("PARAMETERS ","  WT=  ",WT,"   MAXITER=  ",MAXITER);
0510          WRITE("STANDARD GOAL ", "  INITIALLY ");
0511          WRTBRD1(INIT);
0512          TIN:=ENTIER(WT);
0513          FOR CC:=TIN    UNTIL 8 DO
0514          BEGIN WT:=WW(CC);
0515             INITIALIZE;
0516
0517             COMMENT   ****  MAIN PROGRAM LOOP  *******;
0518
0519             J:=0;
0520             WHILE J < MAXITER DO
0521             BEGIN
0522
0523
0524          LOGICAL PROCEDURE DECIDE (INTEGER VALUE CN);
0525          CASE CN OF   (
0526             TRUE, COMMENT - FORWARD SEARCH;
0527             FALSE,  COMMENT - BACKWARD SEARCH;
0528             ((J REM 2) = 0),  COMMENT - ALTERNATING BI-DIRECTIONAL SEARCH;
0529             ((J REM 3) = 0),  COMMENT -ALTERNATING  1 FORW 2 BACK;
0530             (SDIST < TDIST), COMMENT - BI-DIRECTIONAL EQUIDISTANT SEARCH;
0531
0532             COMMENT  A FORM OF PENETRANCE RULE;
0533             (  (SDIST+1)/(NUMNODE-FUSE) > (TDIST+1)/(BNUMNODE-BUSE) ),
0534
0535             COMMENT  DECISION BASED ON BRANCHING FOR TREE OF LENGTH *DIST;
0536             ( LN(NUMNODE-FUSE)*(TDIST+1) < LN(BNUMNODE-BUSE)*(SDIST+1) )
0537                );
0538
```

```
0539
0540          PROCEDURE TERMINATE(STRING (16) VAR;  LOGICAL FLG);
0541             BEGIN  COMMENT  CHECKS WHETHER CRNOD IS IN BOTH TREES     ;
0542                INTEGER T;   DECOD (VAR,BOARD); T:=HASH(BOARD);
0543                IF FLG THEN
0544                   BEGIN P2:=BHSH(T);
0545                      WHILE P2 ¬=NULL DO
0546                         IF VAR=ENCOD(P2) THEN
0547                         BEGIN
0548                            T1:=NUMNODE;     T2:=WF(P2);
0549                            P1:=PT(NUMNODE);  GOTO TRACEPATH
0550                         END
0551                         ELSE BEGIN T2:=WF(P2); P2:=SPILL(P2) END
0552                   END
0553                ELSE
0554                   BEGIN P1:=HSH(T);
0555                      WHILE P1 ¬= NULL DO
0556                         IF VAR=ENCOD(P1) THEN
0557                         BEGIN
0558                            T2:=BNUMNODE;     T1:=WF(P1);
0559                            P2:=BPT(BNUMNODE);  GOTO TRACEPATH;
0560                         END
0561                         ELSE BEGIN T1:=WF(P1); P1:=SPILL(P1) END
0562                   END
0563             END TERMINATE;
0564
0565
0566
0567                TFLAG:=TRUE;
0568          IF DECIDE( DECN) THEN
0569             BEGIN
0570             COMMENT     FORWARD  DIRECTION    ;
0571                MIN:=100000;  FLG:=TRUE;
0572
0573                COMMENT  SEARCH FOR UNDEVELOPED NODE WITH MINIMUM VALUE;
0574                FOR I:=NUMNODE STEP -1 UNTIL 1 DO
0575                   IF  UNDEVELOPED(I) THEN
0576                      BEGIN
0577                         IF MIN>VALU(PT(I)) THEN
0578                            BEGIN  MIN:=VALU(PT(I)); CURNOD:=I END;
0579                      END UNDEV;
0580
0581                COMMENT INCREMENT NUMBER OF NODES EXPANDED;
0582                FUSE:=FUSE+1 ;
0583                UNDEVELOPED(CURNOD):= FALSE;
0584                SDIST:=DIST( PT( CURNOD));
0585
0586                COMMENT  GENERATE ADJACENT NODES AND CHECK FOR REDUNDANCY;
0587                SUCCESSOR(ENCOD(PT(CURNOD)),DEG,PZ(PT(CURNOD)),NZ,NXNODES);
0588                FOR I:=1 STEP 1 UNTIL DEG DO
0589                IF NONREDUNDANT(NXNODES(I),T2,FLG) THEN
0590                BEGIN
0591
0592                   COMMENT  ADD NEW NODE TO FORWARD SEARCH TREE;
0593                   EVALUATE(NXNODES(I),NZ(I),VAL,WT,EVALN,FLG);
0594                   NUMNODE:=NUMNODE+1;
0595                   ST1:=NXNODES(I);
0596                   PT(NUMNODE):=NODE(ST1,NZ(I),CURNOD,DIST(PT(CURNOD))+1,
0597                      VAL,NULL);
0598                      UNDEVELOPED(NUMNODE):= TRUE;
```

```
0599
0600                    COMMENT   CALCULATE SPILL POINTER FOR HASH CLASS;
0601                    IF HSH(T2)=NULL THEN HSH(T2):=PT(NUMNODE)
0602                    ELSE
0603                       BEGIN
0604                            P1:=HSH(T2);
0605                            WHILE P1 ¬= NULL DO
0606                            BEGIN  P2:=P1; P1:=SPILL(P1) END;
0607                            SPILL(P2) := PT(NUMNODE)
0608                       END;
0609
0610                 COMMENT  CHECK IF NODE IS CONTAINED IN BOTH SEARCH TREES;
0611                 TERMINATE(NXNODES(I),FLG)
0612              END LOOPI;
0613           END
0614           ELSE
0615           BEGIN  COMMENT  BACKWARD DIRECTION ;
0616              MIN:=100000; FLG:=FALSE;
0617              FOR I:=BNUMNODE STEP -1 UNTIL 1 DO
0618                 IF  BUNDEVELOPED(I) THEN
0619                    BEGIN
0620                       IF  MIN >  VALU(BPT(I)) THEN
0621                          BEGIN  MIN:= VALU(BPT(I)); BCURNOD:=I END;
0622                    END ;
0623              BUSE:=BUSE+1 ;
0624              BUNDEVELOPED(BCURNOD):=FALSE;
0625              TDIST:=DIST(BPT(BCURNOD));
0626              SUCCESSOR(ENCOD(BPT(BCURNOD)),DEG,PZ(BPT(BCURNOD)),NZ,NXNODES)
0627              ;
0628              FOR I:=1 STEP 1 UNTIL DEG DO
0629              IF NONREDUNDANT(NXNODES(I),T2,FLG) THEN
0630              BEGIN
0631                 EVALUATE(NXNODES(I),NZ(I),VAL,WT,EVALN,FLG);
0632                 BNUMNODE:=BNUMNODE+1;
0633                 ST1:=NXNODES(I);
0634                 BPT(BNUMNODE):=NODE(ST1, NZ(I), BCURNOD,
0635                    DIST(BPT(BCURNOD))+1,VAL,NULL);
0636                 BUNDEVELOPED(BNUMNODE):= TRUE;
0637                 IF  BHSH(T2)=NULL THEN BHSH(T2):=BPT(BNUMNODE)
0638                 ELSE
0639                    BEGIN
0640                       P1:=BHSH(T2);
0641                       WHILE P1 ¬= NULL DO
0642                          BEGIN  P2:=P1;  P1:=SPILL(P1) END;
0643                       SPILL(P2):=BPT(BNUMNODE)
0644                    END;
0645                 TERMINATE(NXNODES(I),FLG)
0646              END LOOPI;
0647           END ELSECLAUSE;
0648
0649           J:=J+1;
0650        END JLOOP;
0651     TRACEPATH: IF J<MAXITER THEN TRACE ELSE WRITE("MAXIMUM ITERATIONS");
0652        WRITE(" ");  WRITE("F DEV NODES ",FUSE," B DEV NODES ",BUSE);
0653        HDISTR;
0654        IF  FUSE+BUSE = MAXITER THEN IF DMPN=5 THEN DMPN:=6;
0655        DMP(DMPN);
0656        END CC;
0657       GOTO OVER;
0658       END.
```

REFERENCES

1. Amarel, S., "An approach to heuristic problem solving and theorem proving in the propositional calculus," Carnegie Institute of Technology, Pittsburgh, Pennsylvania (June 1966).
2. Amarel, S., "On machine representations of problems of reasoning about actions — The missionaries and cannibals problem," Carnegie Institute of Technology, Pittsburgh, Pennsylvania (June, 1966).
3. Baecker, R., "Planar representations of complex graphs," Report No. TN-ESD-TR-67-61, Lincoln Laboratory, MIT, Cambridge, Massachusetts, (1967).
4. Bar-Hillel, Y., (Editor), Language and Information, (Addison-Wesley, Palo Alto, California, 1964), especially "Nonfeasibility of FAHQT," pp. 174-179.
5. Bauer, H., S. Becker, and S. Graham, "ALGOL W implementation," Report No. TR-CS-98, Stanford Computer Science Department, Stanford University, Stanford, California (May 1968).
6. Bellman, R. and S. Dreyfus, Applied Dynamic Programming, (Princeton University Press, Princeton, New Jersey, 1962).
7. Berge, C., The Theory of Graphs and Its Applications, (Metheum Co. Ltd., London, England, 1962).
8. Berge, C., and A. Ghouila-Houri, Programming, Games, and Transportation Networks, (Metheun Co. Ltd., London, England, 1965).
9. Berztiss, A., "A note on segmentation of computer programs," Information and Control, 12, 21-22 (January 1968).
10. Burstall, R., "Writing search algorithms in functional form," Machine Intelligence 3, D. Michie, Editor (Oliver & Boyd, Edinburgh, England, 1968); pp. 373-385.

11. Busacker, R. and T. Saaty, Finite Graphs and Networks, (McGraw Hill Company, New York, New York, 1965).

12. Chartres, B., "Letter to the Editor," Computer Journal, 10, 118-119, (May 1967).

13. Corneil, D., "Graph isomorphism," Ph.D. Thesis, University of Toronto, Toronto, Canada, (1968).

14. Corneil, D., and C. Gotlieb, "Algorithm for finding a fundamental set of cycles for an undirected graph," Communications of the ACM, 10, 780-783, (December 1967).

15. Dantzig, G., Linear Programming and Extensions, (Princeton University Press, Princeton, New Jersey, 1963).

16. Dantzig, G., (personal communications, 1968).

17. Dijkstra, E., "A note on two problems in connection with graphs," Numerische Mathematik, 1, 269-271, (1959).

18. Doran, J. and D. Michie, "Experiments with the graph traverser program," Proceedings of the Royal Society (A), 294, 235-259, (1966).

19. Doran, J., "An approach to automatic problem-solving," Machine Intelligence 1, N. Collins and D. Michie, Editors, (Oliver and Boyd, Edinburg, England, 1967); pp. 105-123.

20. Dreyfus, D., "An appraisal of some shortest path algorithms," Report No. RM-5433, Rand Corporation, Santa Monica, California, (August 1967).

21. Feigenbaum, E., "Artificial intelligence: Themes in the second decade," Memo-67, A.I. project, Stanford University, Stanford, California, (August 1968).

22. Feller, W., An Introduction to Probability Theory and Its Applications 1, (John Wiley & Sons, London, England, 1950).

23. Floyd, R., "Algorithm 97, shortest path," Communications of the ACM, 5, 345, (1962).

24. Floyd, R., "Nondeterministic algorithms," Journal of the ACM, 14, 636-644, (October 1967).

25. Ford, L., and D. Fulkerson, Flows in Networks, (Princeton University Press, Princeton, New Jersey, 1962).

26. Golomb, S., and L. Baumert, "Backtrack programming," Journal of the ACM, 12, 516-524, (October 1965).

27. Greenblatt, R. E., D. E. Eastlake, and S. Crocker, "The Greenblatt chess program," Proceedings Fall Joint Computer Conference, (1967); pp. 801-810.

28. Harary, F., R. Normal, and D. Cartwright, Structural Models: An Introduction to the Theory of Directed Graphs, (John Wiley and Sons, Inc., New York, New York, 1965).

29. Hardy, G., J. Littlewood, and G. Polya, Inequalities, (Cambridge University Press, Cambridge, Massachusetts, 1964).

30. Hart, P., N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," Stanford Research Institute report, (June 1967); and IEEE Trans. on Sys. Sci. and Cybernetics, (July 1968).

31. Huberman, B., "A program to play chess end games," Report No. CS-106, Computer Science Department, Stanford University, Stanford, California, (1968).

32. Kaufmann, A., Graphs, Dynamic Programming and Finite Games, (Academic Press, New York, New York, 1967).

33. Knuth, D., "Fundamental algorithms," The Art of Computer Programming 1, (Addison-Wesley, Reading, Massachusetts, 1968).

34. Kozdrowicki, E., "An adaptive tree pruning system: A language for programming heuristic tree searches," Proceedings of the ACM, (1968); pp. 725-735.

35. Lin, Shen, "Computer solution of the traveling salesman problem," Bell System Technical Journal, (December 1965); pp. 2245-2269.

36. McCarthy, J., LISP 1.5 Programmer's Manual, (MIT Press, Cambridge, Massachusetts, 1964).

37. Michie, D., "Strategy building with the graph traverser," Machine Intelligence 1, (Oliver and Boyd, Edinburg, England, 1967); pp. 135-152.

38. Michie, D., J. G. Fleming, and J. V. Oldfield, "A comparison of heuristic, interactive, and unaided methods of solving a shortest-route problem," Machine Intelligence 3, (Oliver and Boyd, Edinburg, England, 1968); pp. 245-255.

39. Minsky, M., "Steps toward artificial intelligence," Computers and Thought, E. Feigenbaum, and J. Feldman, Editors, (McGraw Hill Company, New York, New York, 1963); pp. 406-450.

40. Moore, E., "The shortest path through a maze," Proceedings of an International Symposium on the theory of switching, Part II, April 1957, (Harvard University Press, Cambridge, Massachusetts, 1959); pp. 285-292.

41. Newell, A., and G. Ernst, "The search for generality," Proceeding of IFIP Congress (1965); pp. 17-22.

42. Newell, A., and H. Simon, "GPS, a program that simulates human thought," Computers and Thought, E. Feigenbaum, and J. Feldman, Editors, (McGraw Hill Company, New York, New York, 1963); pp. 279-293.

43. Newell, A., J. C. Shaw and H. Simon, "Chess-playing programs and the problem of complexity," Computers and Thought, E. Feigenbaum, and J. Feldman, Editors, (McGraw Hill Company, New York, New York, 1963); pp. 39-70.

44. Nicholson, T., "Finding the shortest route between two points in a network," Computer Journal, 9, 275-280 (November 1966).

45. Nilsson, N., "Searching problem — solving and game playing trees for minimal cost solutions," IFIPS Congress preprints (1968); pp. H125-H130.

46. Ore, O., Graphs and Their Uses, (Random House, New York, New York, 1963).

47. Ore, O., Theory of Graphs, (AMS Colloquium Publications, Providence, Rhode Island, 1962); p. 38.

48. Pohl, I., "Graph package," GSG Memo-43, Stanford Linear Accelerator Center, Stanford University, Stanford, California (June 1967).

49. Pohl, I., "A method for finding Hamilton paths and Knight's tours," Communications of the ACM, 10, 446-449, (July 1967).

50. Pohl, I., "Phrase-structure productions in PL/I," Letter to Communications of the ACM, 10, 757, (December 1967).

51. Pohl, I., "A generalized extension to shortest path methods," GSG Memo-56, Stanford Linear Accelerator Center, Stanford University, Stanford, California, (March 1968).

52. Polya, G., How to Solve It, (Doubleday, Garden City, New York, 1957); 2nd Ed.

53. Ramamoorthy, C., "Analysis of graphs by connectivity considerations," Journal ACM, 13, 211-222, (April 1966).

54. Samuel, A., "Some studies in machine learning using the game of checkers," Computers and Thought, E. Feigenbaum, and J. Feldman, Editors, (McGraw Hill Company, New York, New York, 1963); pp. 71-105.

55. Sandewall, E., "A planning problem solver based on look-ahead in Stochastic game trees," Report No. NR-13, Department of Computer Science, Uppsala University, Uppsala, Sweden, (1968).

56. Shaw, A., "The fifteen puzzle," (private communication, 1965).

57. Slagle, J., and P. Bursky, "Experiments with a multipurpose, theorem-proving heuristic program," Journal of the ACM, 15, 85-99, (January 1968).

58. Tonge, F., "Summary of a heuristic line balancing procedure," Computer and Thought, E. Feigenbaum, and J. Feldman, Editors, (McGraw Hill Company, New York, New York, 1963); pp. 168-190.

59. Unger, S., "G.I.T. — A heuristic program for testing pairs of directed line graphs for isomorphism," Communications of the ACM, 7, 26-34, (January 1964).

60. Von Neumann, J., Theory of Self-Reproducing Automata, Edited and completed by Arthur W. Burks, (University of Illinois Press, Urbana, Illinois, 1966).

61. Warshall, S., "A theorem on Boolean matrices," Journal of the ACM, 9, 11-12, (January 1962).

62. Wirth, N., and C. Hoare, "A contribution to the development of ALGOL," Communications of the ACM, 9, 413-431, (June 1966).

63. Witzgall, C., "On labeling algorithms for determining shortest paths in networks," Report No. 9840, U. S. National Bureau of Standards, Washington, D.C., (May 1968).