

TECHNIQUES FOR IMPROVING THE EFFICIENCY OF HEURISTIC SEARCH

BY

JOHN F. DILLENBURG

B.S., University of Illinois at Chicago, 1989

M.S., University of Illinois at Chicago, 1991

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 1993

Chicago, Illinois

ACKNOWLEDGMENTS

I am indebted to my thesis advisor Professor Peter C. Nelson in many ways. Professor Nelson provided thoughtful guidance and both financial and emotional support throughout my graduate studies. I would like to thank my thesis committee members, Professors David Boyce, Jorge Lobo, Robert H. Sloan and Jeffery J.P. Tsai for their helpful comments and suggestions. I would also like to thank Christopher Lain for his help in implementing many of the algorithms tested in this thesis. I would like to thank Dr. Claude Reed for the advice and encouragement he provided me throughout both my undergraduate and graduate studies. Financial support from the Department of Electrical Engineering and Computer Science was greatly appreciated. Finally, I owe a debt of gratitude to my father and other family members for their support and encouragement.

JFD

TABLE OF CONTENTS

<u>CHAPTER</u>		<u>PAGE</u>
1.	INTRODUCTION.....	1
2.	BACKGROUND AND SIGNIFICANCE	4
3.	AGENDA DATA STRUCTURES.....	6
3.1	Introduction to Agenda Data Structures	6
3.2	Analysis of Operations Performed in A*	7
3.3	Analysis of Data Structures	8
3.4	Data Structure for MA*	13
3.5	Summary of Agenda Data Structures.....	14
4.	ISLAND SEARCH.....	16
4.1	Introduction to Island Search.....	16
4.2	Single Level Island Search	19
4.2.1	Description of Algorithm I.....	19
4.2.2	Empirical Results.....	20
4.3	Multiple Level Island Search.....	22
4.3.1	Algorithm I _n	23
4.3.2	Description of Algorithm I _n	23
4.3.3	Admissibility of Algorithm I _n	25
4.3.4	Efficiency of Algorithm I _n	26
4.3.5	Island Interference.....	26
4.3.6	Algorithm I _{np}	27
4.3.7	Admissibility of Algorithm I _{np}	29
4.3.8	Efficiency of Algorithm I _{np}	29
4.3.9	Test Results for Algorithm I _{np}	30
4.4	Island Search for Route Planning	34

TABLE OF CONTENTS (continued)

<u>CHAPTER</u>		<u>PAGE</u>
4.5	Extensions to Island Search	37
4.5.1	Iterative Depth-First Island Search	38
4.5.2	An ϵ -admissible Multiple Level Island Search	39
4.6	Summary of Island Search Results	40
5.	CYCLE CHECKING.....	41
5.1	Introduction to Cycle Checking	41
5.2	Test Results.....	43
5.2.1	The Grid Search Problem	44
5.2.2	The 15-Puzzle Problem	45
5.2.3	The Route Planning Problem	47
5.3	Summary of Cycle Checking Results.....	49
6.	PERIMETER SEARCH	52
6.1	Introduction to Perimeter Search	52
6.2	Perimeter Search Algorithms	55
6.3	Analyzing Perimeter Search.....	58
6.4	Minimizing Heuristic Evaluations	62
6.5	Test Results.....	63
6.5.1	15-Puzzle Problem	64
6.5.2	Think-A-Dot	66
6.6	Near Optimal Perimeter Search.....	68
6.7	Constant Evaluation Perimeters	72
6.8	Parallel Perimeter Search.....	74
6.9	Perimeter Search for the Grid Search Problem.....	76
6.10	Summary of Perimeter Search.....	82

TABLE OF CONTENTS (continued)

<u>CHAPTER</u>	<u>PAGE</u>
7. CONCLUSIONS	85
REFERENCES	87
VITA	91

LIST OF TABLES

<u>TABLE</u>		<u>PAGE</u>
I	OPERATIONS PERFORMED IN A* ALGORITHM	8
II	COMPARISON OF VARIOUS DATA STRUCTURES FOR USE AS AN AGENDA	9
III	SUMMARY OF TEST RESULTS FOR THE 15-PUZZLE PROBLEM.....	47
IV	SUMMARY OF TEST RESULTS FOR THE ROUTE PLANNING PROBLEM	48
V	HYPOTHESES	49
VI	RESULTS OF TESTS OF HYPOTHESES	49
VII	GUIDE TO CYCLE CHECKING TYPE TO USE	50
VIII	OPTIMUM VALUES OF ∂ FOR THE 15-PUZZLE PROBLEM AND THE CORRESPONDING SEARCH SPEEDUP	62
IX	SUMMARY OF TEST RESULTS FOR THE 15-PUZZLE PROBLEM.....	65
X	SUMMARY OF TEST RESULTS FOR THE THINK-A- DOT PROBLEM.....	68
XI	COMPARISON OF CONSTANT DEPTH AND CONSTANT EVALUATION PERIMETERS	74
XII	RESULTS OF PARALLEL PERIMETER SEARCH TESTS	75

LIST OF FIGURES

<u>FIGURE</u>		<u>PAGE</u>
1	One node in an augmented balanced binary tree data structure.....	11
2	Hash table and binary tree agenda data structure	12
3	One node in an augmented balanced binary tree data structure for MA *	14
4	Example search problem with identifiable islands.....	17
5	Sample grid search problem used to test island search algorithms.....	20
6	Node expansion comparison of Algorithm I to A* search. One random obstacle, two islands, every optimal path through an island.....	21
7	Elapsed CPU time comparison of Algorithm I to A* search. One random obstacle, two islands, every optimal path through an island.....	22
8	Search space with pronounced interference problem.....	27
9	Comparison of algorithm complexity with and without branch and bound technique	29
10	Node expansion comparison between four different search methods	31
11	Elapsed CPU time comparison between four different search methods	31
12	Node expansion comparison for multiple level island search	32
13	Elapsed CPU time comparison for multiple level island search.....	33
14	Memory usage in multiple level island search.....	34

LIST OF FIGURES (continued)

<u>FIGURE</u>		<u>PAGE</u>
15	Simulated amount of time required to plan routes using in-vehicle electronics.....	37
16	Cycle checking results for four connected grid	44
17	Cycle checking results for four connected grid	45
18	15-puzzle test results.....	46
19	Hypothetical search space showing perimeter around goal Γ	56
20	Algorithm for generating perimeter P of depth d.....	57
21	Analytical time complexity of perimeter search for the 15-puzzle problem.....	60
22	Test results for 15-puzzle.....	65
23	Think-A-Dot Problem used in tests	66
24	Test results for Think-A-Dot.....	68
25	Hypothetical search space showing path found by near-optimal perimeter search and optimal path.....	70
26	Search times for near-optimal perimeter search	71
27	Path lengths for near-optimal perimeter search	72
28	Graphical depiction of two types of perimeters.....	73
29	Grid search problem used in analysis	77
30	Nodes within these boundaries will be expanded by perimeter search	80
31	Theoretical number of nodes expanded for grid search problem	81
32	Maximum value of f versus grid search problem size	82

NOMENCLATURE

s	start node
Γ	goal node
m, n	other nodes
i, i_1, i_2, \dots, i_k	island nodes
$c(m, n)$	cost of direct path from m to n
$g^*(n)$	cost of least cost path from s to n
$h^*(n)$	cost of least cost path from n to Γ
$h^*(n/i)$	cost of least cost path from n to Γ containing i
$h^*(n, m)$	cost of least cost path from n to m
$f^*(n)$	$g^*(n) + h^*(n)$
$g(n), h(n),$ $h(n/i), h(n, m), f(n)$	estimates of $g^*(n), h^*(n), h^*(n/i), h^*(n, m),$ and $f^*(n)$
OPEN1, OPEN2	lists (agendas)
CLOSED1, CLOSED2	lists
IS	set of island nodes
E	lower bound on number of islands on optimal path to the goal
$ s $	number of elements in set s
$IS_p(n)$	set of island nodes already passed through along a path from s to node
P_{opt}	Set of nodes on an optimal path from s to Γ
$h^*(n/t)$	Least cost path from n to Γ constrained to contain all the islands in set t
$h(n/t)$	Estimate of $h^*(n/t)$
d, L	Search depth, shorthand for $f^*(s)$
P_∂	Set of nodes in a perimeter of depth ∂
$ P_\partial $	Size of a perimeter of depth ∂
∂	Distance from Γ to nodes in perimeter
t	Time per node expansion

NOMENCLATURE (continued)

f	Fraction of time spent evaluating the heuristic during a node expansion
n_{∂}	Number of node expansions with perimeter search and a perimeter of depth ∂
n_{BFS}	Number of node expansions by breadth-first search when generating perimeter
B	Brute force branching factor ($h(n) = 0$)
b	Effective branching factor ($h(n) > 0$)
η	The number of goals
$f(n) = O(g(n))$	$f(n) \leq k \cdot g(n)$ for some $k > 0$ and all $n \geq n_0$
$f(n) = \Omega(g(n))$	$f(n) \geq k \cdot g(n)$ for some $k > 0$ and all $n \geq n_0$
$f(n) = \Theta(g(n))$	$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

SUMMARY

Four techniques are presented which improve the efficiency of heuristic search algorithms: agenda data structures, island search, cycle checking and perimeter search. Both theoretical and experimental results are presented which show how these techniques improve the time complexity of heuristic search for certain problems.

The first step towards improving the efficiency of a heuristic search algorithm is to use the fastest possible data structure for storing unexplored nodes in the search space. Several candidate data structures are reviewed and a new type of data structure is presented which is both space and time efficient.

Islands are shown to be a generalization of subgoals. Two new algorithms are presented which make use of islands to improve search efficiency. Test results are presented which compare the performance of the island search algorithms to the performance of A^* . The test results cover two problem domains, a grid search problem and a route planning problem. Island search is shown to be superior for these problems when island information is available.

A technique for improving the efficiency of depth-first search on graphs is presented. This technique is referred to as cycle checking since it prevents the generation of duplicate nodes caused by cycles in the search space. Two types of cycle checking are compared and test results are presented for the grid search and route planning problems. Simple guidelines are presented showing which type of cycle checking to use on a given problem.

Perimeter search can be viewed as a new type of bidirectional search. Unlike other bidirectional search algorithms, perimeter search avoids the pitfalls involved in trying to perform two simultaneous searches. Analytical and experimental results are presented to show how the perimeter search technique improves the time complexity of A^* and IDA^* on

SUMMARY (continued)

two types of problems. Also presented are near-optimal and parallel versions of the perimeter search algorithms.

1. INTRODUCTION

The research presented here focuses on heuristic search improvement techniques. Four specific techniques are presented: agenda data structures, island search, cycle checking and perimeter search. Although mostly independent, some of these methods can be combined. For instance, agenda data structures can be used with island search. Island search and cycle checking can be combined in an iterative deepening search.

The primary motivation for the research reported in this thesis is to further the development of general problem solving paradigms. For this reason, much of the research done here concentrates on improving widely used problem solvers like A^* (Hart, 1968) and IDA^* (Korf, 1985). Chapter 2 provides a brief introduction to the field of heuristic search and a general description of heuristic search algorithms. A more detailed description of IDA^* can be found in chapter 5 and a description of A^* can be found in chapter 3.

The work on agenda data structures in chapter 3 was prompted by a general lack of literature about the proper implementation of the OPEN list in A^* and MA^* (Chakrabarti et al., 1989). There are, however, a number of references which describe the effects of heuristic accuracy on search efficiency (Pearl, 1984; Hansson, 1992; Chakrabarti, 1989; Chenoweth and Davis, 1991). This is surprising given the fact that heuristic search requires an exponential number of node expansions for most problem domains. A small reduction in the time per node expansion can result in a substantial improvement in search efficiency. In fact, it will be shown that choosing a good data structure for use with A^* or MA^* can result in reducing the exponent in the exponential time complexity from $2d$ to d , where d is the search depth. When the error in the heuristic estimate is linear, the best one can hope for is to avoid a combinatorial explosion by reducing the base of the exponent with a better

heuristic. A good heuristic will therefore permit harder problems to be solved in the same amount of time. Likewise, the choice of an efficient agenda data structure can double the size of problems which can be solved.

The island search algorithms in chapter 4 provide a general method of incorporating subgoal information into a heuristic search. In fact, the concept of a subgoal is taken a step further and a more general concept, the island, is discussed. Subgoals are states in the search space which *must* lie on a path to the goal whereas islands are states that *might* be on a path to the goal. The efficiency of a heuristic search can be improved when island information is available since the islands can be used to prune additional nodes from the search tree.

The cycle checking research presented in chapter 5 was prompted by the observation that the IDA* (Korf, 1985) algorithm performed poorly in problem domains whose state space graphs contained many cycles. Some state spaces, the 15-puzzle's for instance, do not contain many cycles and it is not necessary to check for cycles in these types of problem domains. In fact, the extra time used checking for cycles may actually decrease the search performance if there are relatively few cycles. The need to check for cycles when using depth-first type searches was previously noted (Pearl, 1984), but no guidelines were given indicating which types of problem domains cycle checking should be used on. Guidelines for the use of cycle checking will be presented in chapter 5 based on three case studies.

Chapter 6 introduces a new type of heuristic search called perimeter search. Perimeter search started out as an idea to randomly distribute seed states throughout a search space, plan optimal paths to the seed states, and then, when solving a problem, stop searching when a seed state was encountered. Unfortunately, testing with the 15-puzzle

problem revealed that even with 10,000 seed states the probability of encountering a seed state was less than 5 percent. However, the seed states can be arranged around the goal in such a manner as to force every path to the goal to go through at least one seed state. The seed states formed a perimeter around the goal, hence the name “perimeter search.” Chapter 6 will show that perimeter search is a form of bidirectional search and will compare and contrast perimeter search to other bidirectional search algorithms. Test results will show that perimeter search outperforms other bidirectional and unidirectional search algorithms. Concluding remarks and a summary of this dissertation are contained in chapter 7.

2. BACKGROUND AND SIGNIFICANCE

Some of the earliest work in artificial intelligence (AI) concentrated on such things as chess playing and theorem proving, for it was thought that these must surely embody intelligence. For this reason, early AI research dealt with higher level reasoning and problem solving (Newell, 1969). One of the most important goals of AI research is to develop a general problem solving paradigm like *heuristic search*.

Work in heuristic search can be divided up into three classes of problems: two-player games, constraint satisfying, and path finding. Chess, checkers and go are examples of two-player games. Examples of constraint satisfying problems are the eight queens and the graph coloring problems. Finally, some path finding problems include the eight puzzle, Rubik's Cube, and the traveling salesman problem. From these examples, one can see the similarities and differences between the three problem types. The work in this dissertation concentrates mainly on the path finding aspect of heuristic search.

Heuristic search is based upon the *problem space* hypothesis of Allen Newell and Herbert Simon (Newell and Simon, 1972). The problem space hypothesis basically claims that heuristic search is a completely general model of intelligence because all goal oriented symbolic activity can occur in a problem space.

A problem space has two main components: a set of *states* and a collection of *operators*. The states of the problem space represent a configuration of the world or the problem being solved. The operators are actions that map one state of the world into another state. A *problem instance* is a particular problem to be solved and can be viewed as a problem space, a start state, and either a set of goal states or a test for a goal state. A

heuristic search algorithm takes a problem instance as input and finds a sequence of operators which will transform the start state into the goal state.

One way of viewing a problem space is as a graph. *Nodes* of the graph correspond to states in the problem space and edges in the graph correspond to the operators which transform one state into another. Because of this correspondence between problem spaces and graphs, the terms node and state will be used interchangeably.

Most heuristic search algorithms work as follows. First, the start node is stored in a data structure called an *agenda*. Next, a node is selected for expansion. *Node expansion* means generating all the nodes which are successors to the node selected. The selected node is checked to see if it is a goal, and if so the algorithm terminates. Otherwise, the selected node is expanded and its successors are either stored in memory or discarded. This process repeats itself with the selection of the next node until the agenda is either empty or the goal is found.

The *efficiency* of a heuristic search algorithm is typically measured by the number of nodes it generates. One heuristic search algorithm, A, is said to be more efficient than another algorithm, B, when it is shown that A cannot generate more nodes than B. Measuring the performance of a search algorithm solely by the number of nodes generated can be misleading, however. Some algorithms have a higher node expansion overhead than others. For this reason, the CPU time will also be presented along with node generation data when comparing algorithms.

3. AGENDA DATA STRUCTURES

The data structures used to store nodes in many heuristic search algorithms can have a profound impact on the performance of the algorithm. By using the proper data structure, the complexity of a heuristic search can be reduced from $\Theta(b^{2d})$ to $\Theta(db^d \log b)$, where b is the branching factor and d is the search depth. In this chapter, several data structures will be reviewed. The advantages and disadvantages of each data structure will be analyzed and an augmented balanced binary tree data structure will be presented which achieves the $\Theta(db^d \log b)$ time complexity.

3.1 Introduction to Agenda Data Structures

Many heuristic search algorithms make use of a data structure known as an agenda for storing nodes. The choice of an efficient data structure for the agenda is crucial since it is accessed at least once every node expansion. Surprisingly, this topic has received little attention from the artificial intelligence community. There are a few notable exceptions, however. Chakrabarti et al. (1989) make several suggestions for a data structure to use with their MA* algorithm including binary trees and multiple ordered linked lists. The multiple stack branch and bound algorithm was developed specifically for the purpose of reducing the node selection overhead from the agenda (Sarkar et al., 1991). Kumar et al. (1988) suggest using a heap as the data structure for the agenda. It will be shown, however, that these data structures are not necessarily the most efficient for a given problem domain.

The use of a proper data structure is almost as important as the choice of a good heuristic for a given problem domain. Although the data structures presented here cannot

make the time complexity of an NP-complete problem polynomial, they can halve the exponent of an exponentially complex problem. To see this, note that Pearl has shown that the complexity of heuristic search is an exponential function of the error in the heuristic (Pearl, 1984). Most heuristics have linear error, resulting in an exponential time complexity. A better heuristic will reduce the base of the exponent and further delay a combinatorial explosion (Korf, 1991). This allows larger problems to be solved in the same amount of time. By choosing a good data structure, the exponent itself can be reduced, from $2d$ to d . This, in effect, doubles the size of problems which can be solved.

The operations performed on an agenda will be detailed in Section 3.2. A data structure will be presented which is asymptotically faster than the data structures suggested above in Section 3.3. Section 3.4 contains a brief discussion about a good data structure to use for storing nodes when using the MA^* algorithm.

3.2 Analysis of Operations Performed in A^*

Before a data structure can be developed, the operations which will be performed on the data structure must be clarified. The A^* algorithm will be used for this preliminary analysis (Nilsson, 1980). The analysis will be based on the time complexity needed to perform a node expansion. A node expansion is defined as the process of generating the successors of a given state, see Step 5 of A^* . Table I lists the operations performed in A^* . The quantity $c(m,n)$ is the arc cost between a state m and its successor n and $h(n,m)$ is the heuristic estimate of the cost of the least cost path from n to m .

Algorithm A^*

- Step 1.* Set $f(s) = g(s) = 0$, $p(s) = \text{nil}$. Place the start node s into OPEN. CLOSED is empty
- Step 2.* If OPEN is empty, then exit with no solution found

- Step 3.* Extract a node n from OPEN with the smallest value of $f(n)$
- Step 4.* If n is the goal, then exit. The solution path can be found by tracing parent pointers p back from n to the start node s
- Step 5.* Expand node n . For each successor n' of n do the following:
- (1) Set $g(n') = g(n) + c(n, n')$, $f(n') = g(n') + h(n', \Gamma)$, and $p(n') = n$
 - (2) If OPEN and CLOSED do not contain a node which represents the same state as n' , then add n' to OPEN
 - (3) If OPEN or CLOSED do contain a node m which represents the same state as n' and $g(n') < g(m)$, then set $g(m) = g(n')$ and $p(m) = n$. If m was on CLOSED, then remove it from CLOSED and place it into OPEN.
- Step 6.* Go to step 2

Operation	Step(s)	Description
$Insert(node\ n)$	1, 5.1, 5.3	Inserts node n into OPEN
$n = ExtractMin$	3	Extracts node n with minimum f value
$m = Find(node\ n)$	5.2, 5.3	Finds a node m which is equal to n
$Update(node\ n)$	5.3	Updates f value of a node

Table I. Operations performed in A* algorithm.

3.3 Analysis of Data Structures

Table II provides a summary comparison of some data structures which can be used for an agenda. The table shows the worst case time complexity for each operation listed in Table II and the memory required to maintain the data structure. The advantages and disadvantages of each data structure listed in the table will be discussed in turn.

Data Structure	Operation				Memory Usage
	<i>Insert</i>	<i>ExtractMin</i>	<i>Find</i>	<i>Update</i>	
Heap	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	<i>state</i>
Balanced Tree, <i>state</i> is key	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	<i>state</i> , two pointers and one bit
Balanced Tree and Hash Table	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$ (average case)	$\Theta(\log n)$	<i>state</i> , three pointers, one bit and unused hash table space
Multiple Stacks	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	<i>state</i>
Augmented Balanced Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	<i>state</i> , three pointers, and one bit

Table II. Comparison of various data structures for use as an agenda, n is the size of the OPEN list.

Using a binary heap (Williams, 1964) is advantageous if the *Find* operation can be eliminated since the *Insert* and *ExtractMin* operations can be performed in $\Theta(\log |\text{OPEN}|)$ time per node expansion, where $|\text{OPEN}|$ is the size of the OPEN list. Note that the *Update* operation is not needed when the *Find* operation is not used. Eliminating the *Find* and *Update* operations is possible when duplicate nodes are not likely to be encountered in the search domain (Pearl, 1984). This translates into an overall $\Theta(\log |\text{OPEN}|)$ time complexity per node expansion. Although a balanced binary tree data structure also exhibits a $\Theta(\log |\text{OPEN}|)$ time complexity per node expansion, a heap uses less memory and thereby eliminates the usefulness of any type of unaugmented balanced binary tree.

If the search domain is likely to contain many duplicate nodes, then the *Find* operation will be needed to keep the size of OPEN as small as possible and to limit the number of node expansions. Unfortunately, finding a node in a heap requires each element

of the heap to be examined in turn. This will increase the time complexity of performing a node expansion to $\Theta(|\text{OPEN}|)$.

Using the time complexity per node expansion, the overall time complexity when using a heap, T_{heap} , can now be estimated. First, assume b^d node expansions, where b is the branching factor and d is the search depth. Next, assume that OPEN grows by one node every node expansion. Since the complexity of performing a node expansion using a heap is $\Theta(|\text{OPEN}|)$, we can determine the overall complexity by summing $k|\text{OPEN}|$ over all node expansions:

$$T_{heap} = \sum_{|\text{OPEN}|=1}^{b^d} k|\text{OPEN}| = k \sum_{|\text{OPEN}|=1}^{b^d} |\text{OPEN}| = \frac{k}{2}(b^{2d} + b^d) = \Theta(b^{2d}),$$

where k is the constant used in the “big-Theta.”

Unlike the *Find* operation in a heap, the *Find* operation in a balanced binary tree keyed on the *state* descriptor can be performed in $\Theta(\log |\text{OPEN}|)$ time. The *Update* operation can be replaced by two tree operations: a *Delete* operation followed by an *Insert*. Since balanced binary trees support deletions in $\Theta(\log |\text{OPEN}|)$ time, the *Update* operation would also have an $\Theta(\log |\text{OPEN}|)$ time complexity. The only operation which remains is the *ExtractMin* operation. Since the *ExtractMin* operation is keyed on a node's f value, the entire binary tree would need to be searched in order to locate the node with the minimum f value. For this reason, unaugmented balanced binary trees should not be used. Since the time complexity per node expansion using a balanced binary tree is the same as a heap, the overall time complexity for a balanced binary tree will also be $\Theta(b^{2d})$.

Although balanced binary trees are not much use by themselves, a few simple modifications can be made to make balanced binary trees a good choice for an agenda data structure. All the operations listed in Table I can be performed in $\Theta(\log |\text{OPEN}|)$ time by

augmenting balanced binary trees with a minimum pointer, see Figure 1. The minimum pointer for a node t points to the node with the minimum f value in the subtree rooted at t . The minimum pointers need to be checked whenever an insertion, deletion or rotation occurs in the binary tree. Only the minimum pointers along a path from a disturbance to the root need to be checked. Hence, maintaining the minimum pointers after an insertion, deletion or rotation will take $\Theta(\log |\text{OPEN}|)$ time and the time complexity per node expansion will be $\Theta(\log |\text{OPEN}|)$.

The overall time complexity when using an augmented balanced binary tree, T_{ABBT} , can be estimated the same way T_{heap} was estimated. Using the same assumptions,

$$T_{ABBT} = \sum_{|\text{OPEN}|=1}^{b^d} k \log |\text{OPEN}| = k \log((b^d)!) = \Theta(db^d \log b),$$

where k is the constant factor used in the “big-Theta.”

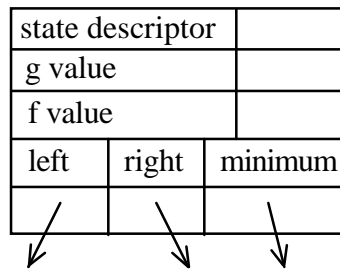


Figure 1. One node in an augmented balanced binary tree data structure.

Instead of adding a minimum pointer to balanced binary trees, a hash table (Morris, 1968) can be used in conjunction with the tree, see Figure 2. The binary tree is keyed on the f value so that the *ExtractMin* operation can be performed in $\Theta(\log |\text{OPEN}|)$ time. The hash table is keyed on the *state* description so that the *Find* operation can be performed in $\Theta(1)$ time. All other operations require $\Theta(\log |\text{OPEN}|)$ time. The only advantage of using a hash table together with a balanced binary tree over an augmented binary tree is the

constant time *Find* operation. The disadvantage is the extra memory being used by the hash table. Since the time complexity of a node expansion is still $\Theta(\log |\text{OPEN}|)$, using a hash table together with a balanced binary tree over an augmented binary tree is questionable.

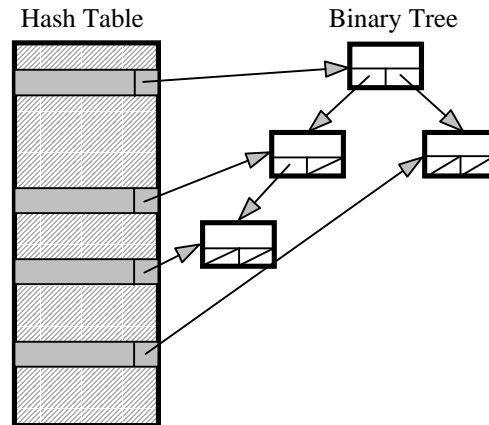


Figure 2. Hash table and binary tree agenda data structure.

One open question related to using a balanced binary tree is choosing the type of balanced binary tree from the many types available. One type of balanced binary tree is especially suited for use as an agenda: red-black trees (Bayer, 1972; Cormen et al., 1990). Red-black trees require only a constant number of rotations whenever a node is inserted or deleted from the tree. Hence, the minimum pointers can be maintained quite easily. Also, red-black trees require only one bit of state to keep track of a node's color.

The multiple stack branch and bound algorithm (MSBB) (Sarkar et al., 1991) uses a data structure quite different from any discussed so far. The MSBB algorithm uses multiple stacks to order nodes in a manner similar to a radix sort. Each node is placed into one of k stacks based on its f value. The *ExtractMin* operation of A^* is replaced by a pop from the first non-empty stack and the *Find* and *Remove* operations are not used. This means that MSBB should not be used in problem domains where node duplication is likely. Since the

node popped from the first stack is not necessarily the one with the minimum f , the first path found by MSBB is not optimal. The optimal path can be found, however, by using the length of path found as an upper bound on f , the smallest f value in the first non-empty stack as a lower bound on f , and performing another iteration until the stacks are left empty. The advantages of using MSBB are its simplicity, constant time per node expansion, and low memory requirements. The disadvantages of using MSBB are the node duplication problem and the fact that one must choose k carefully. Choosing k too large or too small will result in poorer performance relative to A^* (Sarkar et al., 1991).

3.4 Data Structure for MA^*

The MA^* algorithm is a memory bounded heuristic search algorithm (Chakrabarti et al., 1989). First, MA^* extracts a node n from OPEN with the minimum f value, where $f(n)$ is defined in MA^* as the estimate of the cost to reach the goal through *any unexplored* path originating with n . When memory runs short, unpromising nodes are pruned from OPEN to make room for newly generated nodes. A node n is considered unpromising based on its F value, where $F(n)$ is an estimate of the cost required to reach the goal through *any* path originating with n .

Because of its use of both f and F , the MA^* algorithm requires a new type of data structure for storing nodes. In addition to the operations listed in Table 1, MA^* requires an *ExtractMax* operation which extracts the node from the agenda with the maximum F value. Note that the *ExtractMin* operation extracts the node with the minimum f and that f and F are two different quantities.

The augmented red-black tree data structure can also be used for MA^* simply by adding an additional maximum pointer; see Figure 3. The maximum pointer can be

maintained using the same method described previously for maintaining the minimum pointer in A^* . The time complexity per node expansion for MA^* using this data structure will be $\Theta(\log|OPEN|)$.

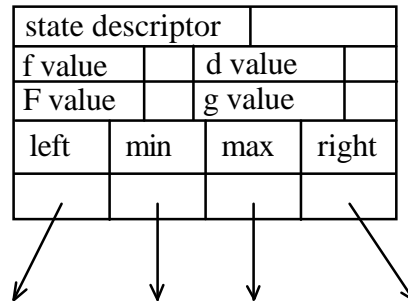


Figure 3. One node in an augmented balanced binary tree data structure for MA^* .

3.5 Summary of Agenda Data Structures

One of the most time consuming operations in heuristic search algorithms like A^* and MA^* is maintaining the agenda. The agenda must support insertions, deletions, and searches under one key, extraction of a minimum based on a different key, and, in the case of MA^* , extraction of a maximum based on yet another key. By augmenting a red-black tree, a type of balanced binary tree, these operations can all be performed in $\Theta(\log |OPEN|)$ time. The disadvantage to using red-black trees is the extra memory needed to store the left, right and minimum pointers. Even though a heap requires $\Theta(|OPEN|)$ time for search operations, there is no additional memory overhead. Additionally, if duplicate nodes are not likely to be encountered in the search space, the search operation can be eliminated and a heap will be the preferred data structure.

The use of a balanced binary tree together with a hash table was also discussed. Although the time complexity per node expansion is the same for both of these data structures, an implementation using the hash table is likely to be faster because of its

constant time *Find* operation. However, this needs to be weighed against the memory required for the hash table before choosing a data structure. When memory usage is not a concern, problem domains with large branching factors are likely to favor the use of a balanced tree and hash table since the *Find* operation is performed once for each successor.

4. ISLAND SEARCH

Certain types of heuristic search problems may contain an identifiable subgoal. This subgoal can be used to break a search into two parts, thus reducing search complexity. However, it might be the case that instead of one subgoal, there will be many possible subgoals, not all of which lie on an optimal path. For this case, the search cannot be simply broken into two parts. However, the possibility of reducing the search complexity still exists.

Previously, Chakrabarti et al. developed a search algorithm called Algorithm I which exploits islands to improve search efficiency. An island as defined by Chakrabarti et al. is a possible subgoal. Previously, this algorithm had only been analyzed theoretically. In this chapter, some experimental results comparing Algorithm I to A^* search are presented. Algorithm I has also been generalized to cases where more than one possible subgoal can appear on an optimal path. Two new heuristic island search algorithms have been created from this generalization and are shown to provide even further improvement over Algorithms I and A^* . The use of possible subgoals can make any type of search more efficient, not just A^* . Modifications are discussed which describe how to incorporate possible subgoal knowledge into IDA^* search.

4.1 Introduction to Island Search

For some heuristic search problems a set of possible subgoals can be identified. These possible subgoals really amount to secondary heuristic information which can potentially help guide the search by breaking it into multiple components. For example, given a starting node s where the optimal path from s to Γ is expected to pass through some

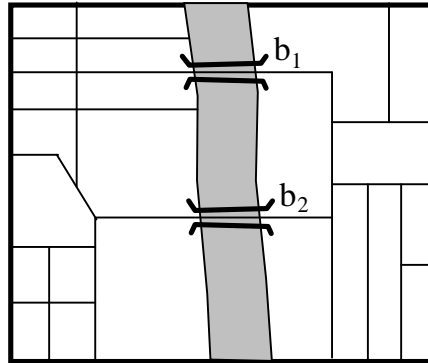


Figure 4. Example search problem with identifiable islands. Roads are represented as dark lines. The two bridges b_1 and b_2 form an island set.

subgoal i , the search can be broken into two components: s to i and i to Γ . By doing so the maximum search depth is reduced, providing a potentially significant improvement in search performance. Subtleties arise as there are usually many subgoals and furthermore only some unknown subset of the subgoals will actually lie on an optimal path.

Many search problems contain identifiable subgoals which can be used to guide the search and thereby reduce the search cost. Take, for instance, the real-life problem of navigating across a city. Say it is necessary to travel across the city and it is known that the city is divided by a river, see Figure 4. Since the river must be crossed on a bridge, this information can be used to help us plan our route. In this example, one of the two bridges b_1 or b_2 will necessarily be a subgoal in our search.

Actually, the word subgoal is a slight misnomer and the term *island* is preferable. It is important here to note the difference between islands and subgoals. The best way to understand islands is to group them together into what is called an island set. An island set is defined by the fact that at least one element of the set will be a node on an optimal path. An island is now defined as simply being one element of the island set. The example in

Figure 4 contains an island set with two elements, one for each bridge across the river. One of the two elements in the list will be on an optimal path (i.e. we must cross one of the bridges to get to the other side). Another example of an island set is the set of all nodes in the search space. It can be shown that in this extreme case the island set would not provide any efficiency improvements.

Island search generalizes heuristic search by allowing the addition of islands. Island search will be shown to be superior to normal A^* search in terms of search efficiency when islands are involved. Island search uses the same heuristic estimate that A^* search would use. The increase in efficiency is due solely to the additional information provided by knowing the location of islands.

In this chapter, it is assumed that the possible subgoal information is known a priori. In other words, island search will only improve search efficiency for problem domains where possible subgoal information is known. For problem domains where possible subgoal information is easy to obtain but not reliable (i.e. it is possible for no islands to be on an optimal path), an ϵ -admissible version of island search can be used, see Section 4.5. As an example of this, consider Figure 4 once again. By testing to see if a straight line between the start and the destination intersects any rivers, it is possible to determine which bridges are possible subgoals. This method is not guaranteed to produce the correct results every time and so the ϵ -admissible version of island search should be used.

The rest of this chapter is organized into five major sections. Section 4.2 contains an overview of the past work done on island search along with some new theoretical and experimental results. Section 4.3 presents a new concept in island search which we call multiple level island search. More theoretical and experimental results will show that multiple level island search is superior to the traditional A^* search under certain

circumstances. Section 4.4 presents a route planning problem and shows how island search can be used to improve route planning efficiency. Section 4.5 shows how the concept island search can be extended into other paradigms such as depth-first iterative deepening. Finally, Section 4.6 summarizes the results and mentions future research possibilities.

4.2 Single Level Island Search

Island search can be divided into two categories: single and multiple level island search. The former, Algorithm I, was first introduced in (Chakrabarti et al., 1986). For single level island search it is assumed that only one subgoal will appear on a path from the start to the goal. Multiple level island search generalizes this idea to cover cases where more than one island may be on a path from the start to the goal. Multiple level island search will be discussed further in Section 4.3. In this section, Algorithm I will be described and some empirical results will be shown comparing the relative merits of island based search to the traditional A^* search (Hart, Duda and Raphael, 1968; Nilsson, 1980).

4.2.1 Description of Algorithm I

Algorithm I is basically an A^* algorithm modified in two ways. First, there are two OPEN and CLOSED lists, OPEN1, OPEN2, CLOSED1, and CLOSED2. Normal A^* search uses only one OPEN and CLOSED list. The extra lists are used in Algorithm I to keep track of which nodes are from a path that contains an island. The second difference is that there are two heuristic estimates employed, $h(n)$ and $h(n/i)$. The heuristic is identical to the heuristic used in A^* . The heuristic $h(n/i)$ estimates the cost of a path from n to the goal constrained to contain an island i . Any nodes which are generated along a path up to and including an island are placed in OPEN1. After reaching an island, newly generated nodes are placed in OPEN2. Nodes which have already been expanded are placed in CLOSED1 if

The rectilinear grid search problem described here contains islands at the edges of obstacles. The reason for this is that the edge of at least one obstacle needs to be skirted in order to reach the goal. This assumes that the optimal path is blocked by at least one obstacle. The island set in this case is the set of all the obstacle edges. If the optimal path were not blocked by any obstacles, then there would not be any acceptable island set to use.

Figures 6 and 7 show the results from some test cases where an obstacle was placed directly between the start and the goal at a random angle while insuring every optimal path goes through an island. Each point on the graph in Figures 6 and 7 is an average of 10 random obstacle placements.

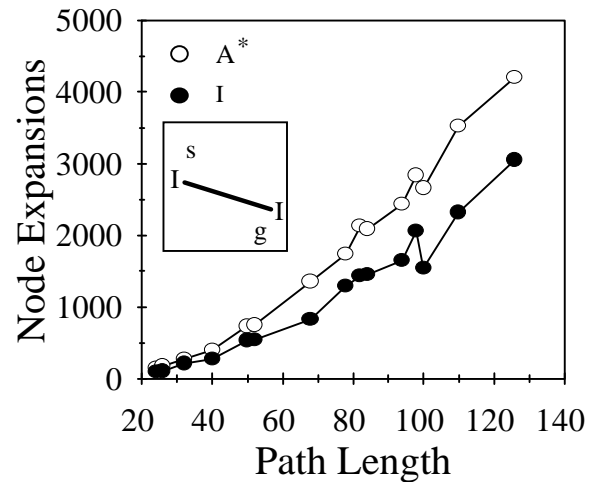


Figure 6. Node expansion comparison of Algorithm I to A* search. One random obstacle, two islands, every optimal path through an island.

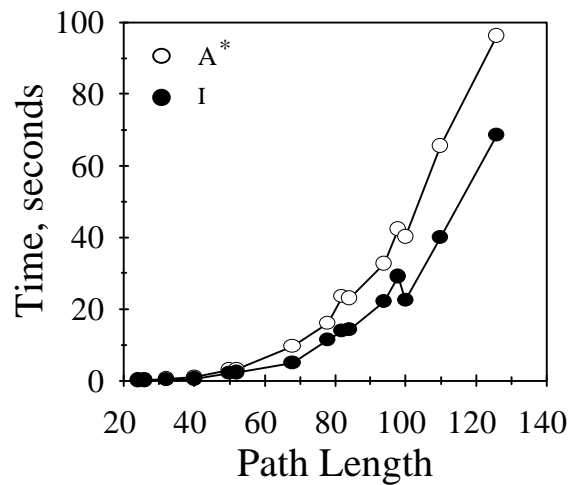


Figure 7. Elapsed CPU time comparison of Algorithm I to A* search. One random obstacle, two islands, every optimal path through an island.

4.3 Multiple Level Island Search

The test results presented in Section 4.2 show that Algorithm I does offer some improvement over A*. However, Algorithm I reverts back to A* after it passes through an island. Often an optimal path will pass through several islands on the way to the goal. Clearly this is the case for the grid search instance shown in Figure 5. Multiple level island search generalizes the idea of island search to cover cases where it is possible to encounter one or more islands on an optimal path from the start to the goal.

In the following sections, two types of multiple level island search algorithms will be presented. The first one, Algorithm I_n , is most similar to Algorithm I. Instead of searching directly for the goal after an island is found, Algorithm I_n searches for the next island. Only after a fixed number of islands have been passed does Algorithm I_n search for the goal. Algorithm I_{np} is a slightly more sophisticated version of Algorithm I_n . Algorithm I_{np} looks not for the next island, but for the next *set* of islands. This further improves search performance in certain cases.

4.3.1 Algorithm I_n

Algorithm I_n is exactly the same as Algorithm I, but with the addition of the variable IS_p(*n*) and the user input parameter *E*. The addition of these two variables allows Algorithm I_n to keep track of how many islands each newly generated node has passed through on a path from the start to that node. The variable IS_p(*n*) is used to keep track of which islands were passed by each node. The quantity *E* is a lower bound on the number of islands between the start and the goal that must be passed through on an optimal path. Algorithm I_n searches for a path to the goal through an island if |IS_p(*n*)| < *E*, otherwise it searches for a path directly to the goal. A newly generated node *n* will be placed on OPEN1 when |IS_p(*n*)| < *E*. Only when |IS_p(*n*)| ≥ *E* will *n* be placed on OPEN2. As testimony to the generality of Algorithm I_n, note that for *E* = 0 Algorithm I_n is equivalent to A*, and for *E* = 1 it is equivalent to Algorithm I.

4.3.2 Description of Algorithm I_n

Algorithm I_n

Inputs: IS Island set
 E Minimum number of islands on path
 s Start state

Output: Optimal path from *s* to Γ , if a path exists

- Step 1.* Put *s* into OPEN1. OPEN2, CLOSED1 and CLOSED2 are empty. Set $g(s) = 0, f(s) = 0, IS_p(s) = \emptyset$ and $IS = (i_1, i_2, \dots, i_k)$.
- Step 2.* If both OPEN1 and OPEN2 are empty, then exit with failure.
- Step 3.* Select one node *n* from OPEN1 or OPEN2 such that $f(n)$ is minimum, resolve ties arbitrarily but always in favor of the goal node. Put *n* in CLOSED1 if it was selected from OPEN1, otherwise put *n* into CLOSED2.
- Step 4.* If *n* is the goal, then exit successfully with the solution obtained.
- Step 5.* Expand node *n*. If there are no successors then go to Step 2. For each successor n_i compute

$$g_i = g(n) + c(n, n_i).$$

Step 6. If $n \in IS$ then set $IS_{pi} = IS_p(n) \cup \{n\}$ else set $IS_{pi} = IS_p(n)$.

Step 7. If $|IS_{pi}| \geq E$ or if n was selected from OPEN2, then do:

- (1) If an immediate successor n_i is not in any of OPEN1, OPEN2, CLOSED1, or CLOSED2, then do:

$$\begin{aligned} g(n_i) &= g_i \\ f(n_i) &= g_i + h(n_i) \\ IS_p(n_i) &= IS_{pi} \\ \text{Put } n_i &\text{ in OPEN2.} \end{aligned}$$

- (2) If an immediate successor n_i is already in one of OPEN1, OPEN2, CLOSED1, or CLOSED2, and $g(n_i) > g_i$, then do :

$$\begin{aligned} g(n_i) &= g_i \\ f(n_i) &= g(n_i) + h(n_i) \\ IS_p(n_i) &= IS_{pi} \\ \text{Put } n_i &\text{ in OPEN2.} \end{aligned}$$

Step 8. If $|IS_{pi}| < E$ and n was selected from OPEN1, then do:

- (1) If an immediate successor n_i is not in any of OPEN1, OPEN2, CLOSED1 or CLOSED2, then do:

$$\begin{aligned} g(n_i) &= g_i \\ f(n_i) &= g_i + \text{Min}_j h(n_i/i_j) && i_j \in IS - IS_{pi} \\ IS_p(n_i) &= IS_{pi} \\ \text{Put } n_i &\text{ in OPEN1.} \end{aligned}$$

- (2) If an immediate successor n_i is in either OPEN1 or CLOSED1 and $g(n_i) > g_i$, then do:

$$\begin{aligned} g(n_i) &= g_i \\ f(n_i) &= g_i + \text{Min}_j h(n_i/i_j) && i_j \in IS - IS_{pi} \\ IS_p(n_i) &= IS_{pi} \\ \text{Put } n_i &\text{ in OPEN1.} \end{aligned}$$

- (3) If an immediate successor n_i is in either OPEN2 or CLOSED2 and $g(n_i) > g_i$, then do:

$$\begin{aligned} g(n_i) &= g_i \\ f(n_i) &= g_i + h(n_i) \\ IS_p(n_i) &= IS_{pi} \\ \text{Put } n_i &\text{ in OPEN2.} \end{aligned}$$

Step 9. Go to Step 2.

4.3.3 Admissibility of Algorithm I_n

Recall that a search algorithm is *admissible* if it finds an optimal path whenever an optimal path exists. This section will present an informal proof of the admissibility of Algorithm I_n given the following assumptions:

- At least E islands lie on an optimal path from the start s to the goal Γ .
- The heuristic estimate is admissible: $0 \leq h \leq h^*$.
- The heuristic obeys the triangle inequality

$$h(n, n') \leq h(n, m) + h(m, n').$$

A necessary condition for Algorithm I_n being admissible is that it terminates when a goal is reachable. This has been proven for both finite and infinite graphs in (Dillenburg, 1991).

Theorem 1: Algorithm I_n is admissible if there exists a path from s to Γ .

Proof: The algorithm can either terminate in step 2 after both OPEN lists are empty or in step 4 after finding the goal. The algorithm cannot terminate in step 2 because at least one node exists on the OPEN lists that is on an optimal path (Dillenburg, 1991). Therefore, the algorithm must terminate in step 4 after finding the goal. The path it found to the goal must be an optimal one. To see this, assume the contrary, that the algorithm terminated without finding an optimal path so that $f(\Gamma) > f^*(s)$. There must exist a node n' on one of the OPEN lists just before termination with $f(n') \leq f^*(s)$. But this means that the algorithm would have chosen n' over Γ because of its lower f value. By contradiction we see the algorithm must terminate by finding an optimal path. ■

4.3.4 Efficiency of Algorithm I_n

After showing Algorithm I_n is admissible, it is easily proved that it expands no more nodes than Algorithm I. This means that for any search in which islands can be identified, Algorithm I_n would yield a search at least as efficient as Algorithm I, which was previously shown to be at least as efficient as A^* . We keep the same assumptions as in the previous section.

Theorem 2: Algorithm I_n expands no more nodes than Algorithm I.

Proof: Nilsson (Nilsson, 1980) showed that for two admissible algorithms, the one with the greater heuristic estimate will not expand any more nodes than its less informed counterpart. This is the case between Algorithm I_n and Algorithm I because

$$h_{I_n} = \begin{cases} h(n/i) & \text{if } |IS_p(n)| < E \text{ or} \\ h(n) & \text{otherwise and} \end{cases}$$

$$h_I = \begin{cases} h(n/i) & \text{if } |IS_p(n)| < 1 \text{ or} \\ h(n) & \text{otherwise} \end{cases}$$

so one can see that $h_{I_n} \geq h_I$ when $E \geq 1$. ■

4.3.5 Island Interference

One problem with island search becomes evident when there is a large number of islands but relatively few optimal paths through the islands. What can happen in this case is that the heuristic estimate $h(n/i)$ guides the search toward the wrong island. A number of test results revealed that in these cases island search is only marginally better than A^* . Figure 8 is an extreme example of how an island heuristic can mislead a search. The heuristic estimate through island I_2 (h_2) is smaller than the estimate through I_1 (h_1) so $\text{Min}_j h(n/i_j) = h_2$ for a large portion of the search. This will lead the search towards I_2 when it really should initially be heading towards I_1 . Since I_2 is so close to the goal, Algorithm I's

performance will not be much better than A^* . Similar situations also occur for Algorithm I_n .

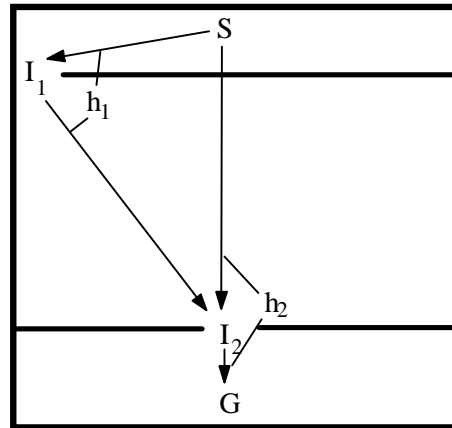


Figure 8. Search space with pronounced interference problem.

4.3.6 Algorithm I_{np}

The quantity $\text{Min}_j h(n_i/i_j)$ picks the island which yields the lowest heuristic estimate. This guarantees admissibility, but does not cut down the size of the search space for cases similar to that depicted in Figure 8. Although Algorithm I_n does generalize to multiple islands, it still suffers from the same interference problem as Algorithm I. One solution is to use $\text{Min}_j h(n/t_j)$ in place of $\text{Min}_j h(n_i/i_j)$. The function that computes $\text{Min}_j h(n/t_j)$ is called the permuted heuristic and instead of finding the shortest path through one island in $IS - IS_p(n)$, it finds the shortest path through a permutation of $E - |IS_p(n)|$ different islands in $IS - IS_p(n)$. An algorithm for computing the permuted heuristic can be defined recursively as follows:

Permuted Heuristic

```

function  $hx(n, IS_p)$ 
  if ( $|IS_p| == E$ )
     $retval = h(n, \Gamma)$ 
  else
     $retval = \infty$ 
    for every island  $i$  in  $IS$  but not in  $IS_p$  do
       $retval = Min(retval, h(n, i) + hx(i, IS_p \cup \{i\}))$ 
    end for
  end if
   $hx = retval$ 
end

```

The hx heuristic is admissible if and only if at least E islands are on an optimal path from the start to the goal. It is also assumed that a heuristic $h(n, m)$ is available to estimate the distance between any node and any island (or the goal). The hx heuristic can be added to Algorithm I_n to produce Algorithm I_{np} simply by replacing all occurrences of $Min_j h(n_i/i_j)$ with calls to $hx(n_i, IS_{pi})$. Algorithm I_{np} , like Algorithm I_n , is equivalent to A^* for $E = 0$ and Algorithm I for $E = 1$.

The permuted heuristic is, in general, intractable. This can be proved by a simple reduction to the traveling salesman problem. Note, however, that the heuristic is only intractable when the number of islands is a function of the size of the search space. Through the use of branch and bound techniques, the computational complexity can be reduced as seen in Figure 9. The branch and bound technique was applied to the permuted heuristic by keeping a global minimum and stopping a recursive call whenever hx exceeded this minimum. The use of branch and bound together with a relatively small island set can make the permuted heuristic more usable.

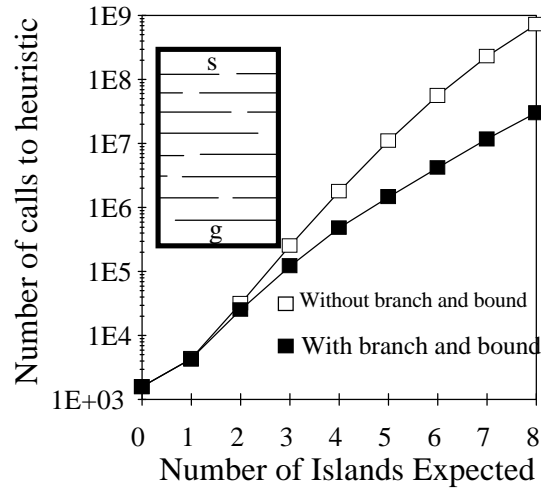


Figure 9. Comparison of algorithm complexity with and without branch and bound technique.

4.3.7 Admissibility of Algorithm I_{np}

Algorithm I_{np} must be shown to be admissible before further claims can be made about it. The theorems showing that Algorithm I_{np} is admissible are very similar to those presented for Algorithm I_n . The theorems in this section require the same assumptions made for Algorithm I_n , exceptions noted. As with Algorithm I_n , Algorithm I_{np} also terminates when a goal is reachable, see (Dillenburg, 1991).

Theorem 3: Algorithm I_{np} is admissible if there exists a path from s to Γ .

Proof: This proof is identical to the proof of Theorem 1. ■

4.3.8 Efficiency of Algorithm I_{np}

It has already been shown that Algorithm I_n is at least as efficient as Algorithm I which is at least as efficient as A^* . We can show that Algorithm I_{np} is at least as efficient as Algorithm I_n . Showing that Algorithm I_{np} is at least as efficient as Algorithm I_n will prove (by transitivity) it is at least as efficient as any of the three island algorithms. Of course, all

of this is under the assumptions made earlier about there being at least E islands from the island set IS on an optimal path.

Theorem 4: Algorithm I_{np} expands no more nodes than Algorithm I_n .

Proof: Both algorithms are admissible, so the algorithm with the lower heuristic estimate will expand at least as many nodes as its more informed counterpart.

$$h_{t_{np}} = \begin{cases} h(n|t) & \text{if } IS_p(n) < E \text{ (t is a)} \\ h(n) & \text{otherwise} \end{cases}$$

$$h_{t_n} = \begin{cases} h(n|i) & \text{if } IS_p(n) < E \text{ or} \\ h(n) & \text{otherwise} \end{cases}$$

One can see that $h_{t_{np}} \geq h_{t_n}$ because

$$h(n|t) = h(n, i_1) + h(i_1, i_2) + \dots + h(i_k, \Gamma)$$

is greater than

$$h(n|i) = h(n, i) + h(i, \Gamma)$$

when $h(n, m) \leq h(n, n') + h(n', m)$. ■

4.3.9 Test Results for Algorithm I_{np}

The rectilinear grid search problem was also used to test Algorithm I_{np} . The test involved partitioning the grid search space into $n + 1$ spaces using n obstacles, each with a small hole, as shown in the diagram in Figure 10. This can be viewed as $n + 1$ rooms placed end to end with a doorway (island) between each room in a random location. Algorithms I_n and I_{np} are admissible for this test so long as $E \leq n$. Figure 10 shows the number of node expansions versus the problem size and Figure 11 shows the elapsed CPU time versus the problem size. The improved performance of Algorithm I_{np} is due to the elimination of the interference problem.

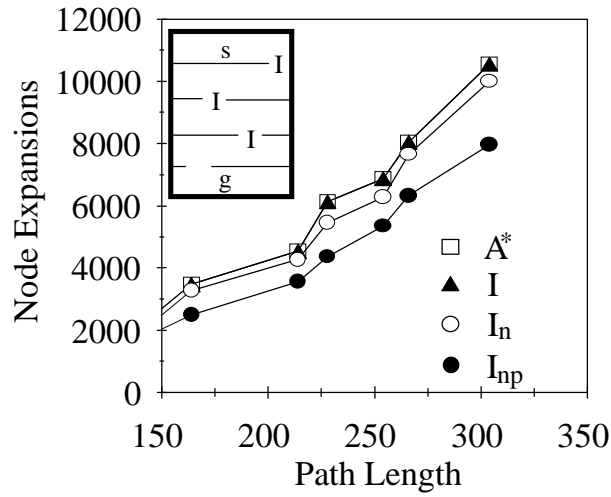


Figure 10. Node expansion comparison between four different search methods.

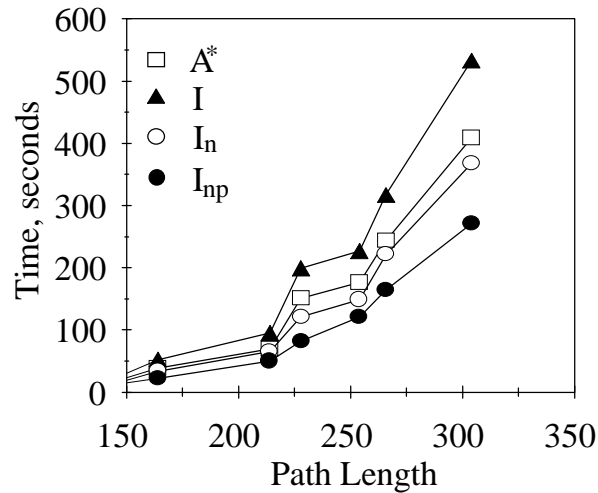


Figure 11. Elapsed CPU time comparison between four different search methods.

In addition to testing search performance as the problem size grows, it is also interesting to test the search efficiency versus the parameter E . Remembering that for $E = 0$ both algorithms are equivalent to A^* and that for $E = 1$ both algorithms are equivalent to Algorithm I, we see from Figure 12 that Algorithm I provides little or no

benefit over A^* for this case involving multiple islands. Figure 13 shows that multiple level island search needs more CPU time in some cases because the extra time needed to compute the island heuristic is not made up by fewer node expansions. The improvement using Algorithm I_n as compared to Algorithm I_{np} is slight due to the interference problem. The effect of the interference problem can be seen by comparing the two curves in Figure 12 since the permuted heuristic effectively eliminates this problem. The results show that the number of node expansions drop as the parameter E increases. Past a certain point, however, hx is no longer admissible and the number of node expansions increases dramatically. Only results from admissible heuristics are shown in Figures 12 and 13.

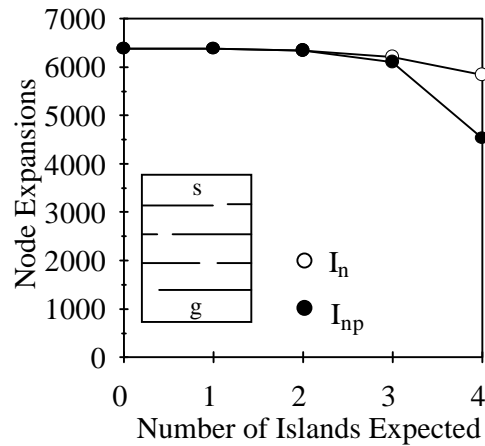


Figure 12. Node expansion comparison for multiple level island search.

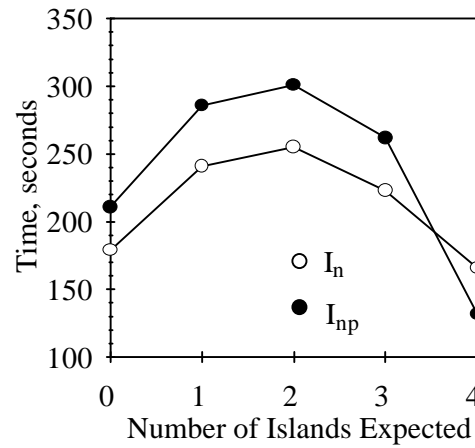


Figure 13. Elapsed CPU time comparison for multiple level island search.

Another benefit of reducing the number of node expansions is that the amount of memory needed to solve the problem is also reduced. Because fewer nodes are being expanded, fewer nodes need to be stored in the OPEN and CLOSED lists. Figure 14 shows how the amount of memory needed varies as a function of E , the lower bound on the number of islands. As E approaches the actual number of islands, the amount of memory used by Algorithm I_{np} drops substantially. Because Algorithm I_n offers only a small improvement in the number of node expansions, the amount of memory it uses does not improve for this case.

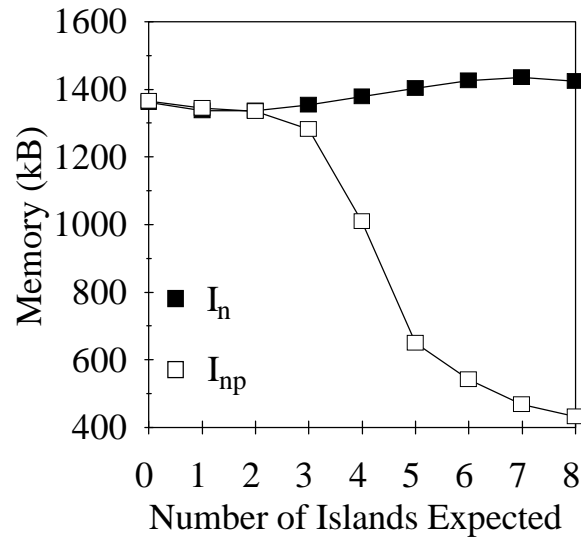


Figure 14. Memory usage in multiple level island search.

4.4 Island Search for Route Planning

The test results in the previous sections show that when island information is available, it may be used to improve search performance. It may be questioned whether there actually exist any “real” problems for which island information is available. In this section, we describe an intelligent vehicle routing problem for which island search is a viable candidate.

Each year the loss of U.S. productivity due to traffic congestion is estimated to be \$100 billion (Ben-Akiva et al., 1992). Unless significant improvements are made to our highway system, the Federal Highway Administration estimates that congestion delays will increase by 400 percent. In an attempt to solve congestion and other related problems associated with our highway system, the development of intelligent vehicle highway system (IVHS) technology is receiving high priority by both the public and private sector.

IVHS covers four broad interrelated areas, advanced traffic management systems, (ATMS), commercial vehicle operations (CVO), advanced vehicle control systems (AVCS) and advanced traveler information systems (ATIS) (Jurgen, 1991).

Currently there are several ATI system demonstrations underway including ADVANCE (Kirson et al., 1992), TRAVTEK (Rillings and Lewis, 1991), and SOCRATES (Catling and de Beek, 1991). The overriding goal of these and other ATI systems is to provide the motorist with useful travel information. With this in mind, a primary function of most ATI systems involves having the ability to plan optimal routes.

The majority of ATI systems currently under development consist of equipping vehicles with computers that are able to exchange traffic information via radio with a centralized computer. Given this type of architecture, the route planning ability of ATI systems can be facilitated using centralized computing resources, although most ATI systems currently in development use in-vehicle computational resources (Kirson et al., 1992; Rillings and Lewis, 1991; Catling and de Beek, 1991). The primary advantage of this type of approach is fault tolerance, vehicles can continue to plan routes even in the absence of a centralized computing center.

However, there are significant challenges associated with planning routes using in-vehicle electronics. These challenges are a result of trying to keep the cost of the in-vehicle electronics to a reasonable (marketable) level. Achieving this is difficult because the in-vehicle electronics must meet automotive standards (temperature, vibration, etc.) and therefore have poor price/performance ratios. Thus, the route planning algorithm must be both space and time efficient.

The island search algorithms I_n and I_{np} can be used to significantly reduce the number of node expansions required to find routes and hence are good candidates for use in

ATI systems. The reduction in node expansions is possible because most routes between two regions in a road network end up passing through a small number of intermediate nodes. These nodes correspond to an island set and can be determined through a learning process as described below.

In a practical implementation, it would be necessary to determine the set of islands associated with each pair of regions in the road network. For the purposes of testing, however, just two regions of a 60 mile radius Chicago area map were selected for study. Each region corresponded to approximately 1% of the map. The island set for the two regions was determined by planning every possible route between the two regions during rush hour traffic conditions. The rush hour travel times were calculated using a transportation network model (Hicks, 1992). Fifty nine nodes were identified which were present in 95% of the routes. Six of these nodes were selected randomly for use as islands.

The island search algorithms I_n and I_{np} were tested by planning routes between 100 randomly selected start/destination pairs located in the two regions. The location of the islands was determined using simulated rush hour conditions and the algorithms were then asked to use these islands to find routes in simulated light traffic conditions. This provided a worst case scenario for testing and was done because in practice the island set for a pair of regions would be precomputed and would be used under varying traffic conditions. Figure 15 shows the results of the tests. Each point in Figure 15 represents the average amount of time needed to plan a route over the 100 test cases. The time data is based on a simulation of a slow external storage device which requires 800 milliseconds to access each node (i.e. an automotive quality CD-ROM). Even though the island search heuristic $h(n/i)$ required more time to compute than the normal A^* heuristic $h(n)$, this time is more than

made up for by the reduction of node expansions. In fact, Figure 15 shows that Algorithm I_{np} was over 200% faster than A^* .

Figure 15 also shows an increase in simulated time for Algorithm I_n when $E=6$. This is due to a combination of the interference effect and the fact that there are actually only five islands on the optimal paths for the 100 test cases. Algorithm I_{np} was not affected as much by the errant island due to its use of the permuted heuristic hx . The modeled travel times of the routes planned by the island search algorithms were optimal in all cases except when $E=6$, in which case routes planned by Algorithm I_n were on average 1% longer and routes planned by Algorithm I_{np} were on average 6% longer than optimal.

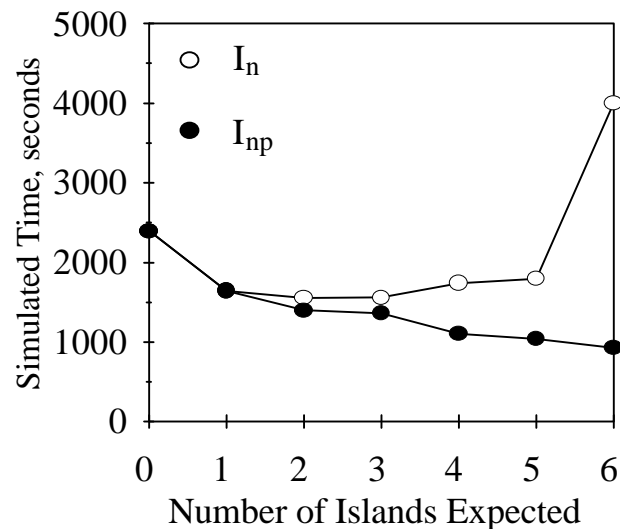


Figure 15. Simulated amount of time required to plan routes using in-vehicle electronics.

4.5 Extensions To Island Search

There are a number of extensions that can be made to island search. These extensions allow the algorithms to be used on a wider variety of problems. For instance, there are many different categories of admissibility (Pearl, 1984). These can be applied to

island search just as easily as they can be applied to A^* . Basically, these *semi-admissible* searches cannot guarantee an optimal solution, but instead a solution *close* to optimal. Another example of an extension that might be useful is the idea of iterative depth-first search. Iterative depth-first search maintains the guarantee of optimality, but has an advantage in terms of the amount of storage space used during the search when compared to regular best-first search. Iterative depth-first search can also be shown to be optimal with respect to time (Korf, 1985).

4.5.1 Iterative Depth-First Island Search

An iterative depth-first search uses only linear storage space while still being able to guarantee an optimal solution path. The algorithm for an iterative depth-first version of island search (IDIS) would be as follows:

IDIS Algorithm

- Step 1. Set $next_cutoff = h(s)$, $g(s) = f(s) = 0$. If s is in IS, then set $i(s) = \text{TRUE}$ else set $i(s) = \text{FALSE}$
- Step 2. Place s on the stack. Set $cutoff = next_cutoff$ and $next_cutoff = \infty$.
- Step 3. If the stack is empty, go to step 2
- Step 4. Pop a node n off the stack. If n is the goal, then exit successfully with the solution obtained.
- Step 5. For each immediate successor n_i of n do the following:
- (1) Set $g(n_i) = g(n) + c(n, n_i)$
 - (2) If $i(n)$ is TRUE or if n is an island, then set $i(n_i) = \text{TRUE}$ and $f(n_i) = g(n_i) + h(n_i)$, otherwise set $i(n_i) = \text{FALSE}$ and $f(n_i) = g(n_i) + \text{Min}_j h(n_i / i_j)$
 - (3) If $f(n_i) \leq cutoff$, then push n_i onto the stack, else set $next_cutoff = \text{Min}(next_cutoff, f(n_i))$
- Step 6. Go to step 3

The iterative depth-first island search algorithm is basically the same as IDA* (Korf, 1985). By adding a Boolean variable to determine when an island has been encountered, the algorithm can decide when to use the island heuristic and when to use the normal heuristic. Note that the Boolean variable can be kept on the stack along with the other state information. Since the heuristic used in IDIS follows the guidelines presented in (Korf, 1985) (admissible and monotone), the IDIS algorithm will also be optimal in terms of time and memory usage. A proof detailing the exact conditions for the admissibility of IDA* (and by extension IDIS) can be found in (Mahanti et al., 1992).

4.5.2 An ϵ -admissible Multiple Level Island Search

The type of semi-admissible search that will be discussed here is called ϵ -admissible. A search algorithm is said to be ϵ -admissible if it can find a solution with a cost of not more than $f^*(s) + \epsilon$. Compare this to the normal requirement that the solution have a cost of not more than $f^*(s)$. With an ϵ -admissible algorithm, we are only guaranteed that a solution is close to optimal. This is the price we pay for a potentially faster search. A simple modification to Algorithm I produces an ϵ -admissible algorithm. Chakrabarti et al. (Chakrabarti et al., 1986) made this extension to Algorithm I to create Algorithm I'. Algorithm I' requires an additional input d which is used to guarantee that the cost of the solution will not be more than $d + f^*(s)$.

Algorithm I_n and I_{np} can become ϵ -admissible in a manner similar to Algorithm I. The only change to the algorithms occurs in steps 8(1) and (2). Instead of always using $\text{Min}_j h(n_i/i_j)$ for the heuristic, we can use $h(n_i)$ whenever $|\text{Min}_j h(n_i/i_j) - h(n_i)| > d$. Basically, this change insures that the heuristic does not become too large. By comparing the possibly inadmissible $\text{Min}_j h(n_i/i_j)$ to the admissible $h(n_i)$ we can guarantee the search finds a path within the error bound. Note that for Algorithm I_{np}' , we use the permuted heuristic in

place of $\text{Min}_j(h(n_i/i_j))$. With this change to the algorithms, it can be shown that we now have an ϵ -admissible algorithm when ϵ is equal to d . This proof is very similar to the proof of Theorem 1.

As mentioned in Section 4.1, the ϵ -admissible versions of the island search algorithms are useful in cases where information about the island set IS or the number of islands on the optimal path E is unreliable. By using an ϵ -admissible island search algorithm, one can guarantee a bounded solution if either IS or E are in error.

4.6 Summary of Island Search Results

Experimental results have been presented showing that island search techniques can be used to reduce the number of node expansions. The single level island search Algorithm I (Chakrabarti et al., 1986) was compared with the A^* algorithm. Algorithm I was shown to offer some improvement. Search spaces which contain identifiable islands might have solution paths through multiple islands. With this in mind two multiple level island search algorithms, I_n and I_{np} , were presented which can be used in such situations. Test results show that both of these algorithms outperform Algorithm I, with Algorithm I_{np} showing the greatest improvement. Furthermore, Algorithm I_{np} has been shown to be admissible and to expand no more nodes than Algorithm I_n .

The use of island search for route planning was also mentioned. By taking advantage of the fact that many paths in a road network pass through a common set of nodes, the island search algorithms I_n and I_{np} were shown to reduce the number of node expansions without sacrificing route quality. These results could have important ramifications for advanced traveler information systems which must rely on in-vehicle route planners.

5. CYCLE CHECKING

An experimental evaluation of a technique for improving the efficiency of depth-first search on graphs is presented in this chapter. The technique, referred to as cycle checking, prevents the generation of duplicate nodes in the depth-first search of a graph by comparing each newly generated node to the nodes already on the search path. Although cycle checking can be applied to any depth-first type search, this chapter focuses on its effect with the heuristic depth-first iterative deepening algorithm IDA*. Two types of cycle checking are compared, full cycle checking and parent cycle checking. Full cycle checking compares a new node to all nodes on the search path. Parent cycle checking merely compares a new node to the parent of the node being expanded. Both types of cycle checking are shown to improve the efficiency of IDA* search in a grid search problem and a route planning problem. Simple guidelines are presented showing which type of cycle checking should be used on a given problem.

5.1 Introduction to Cycle Checking

The IDA* algorithm was introduced by Richard Korf in 1985 as an optimal heuristic search algorithm. In addition to being a relatively simple algorithm, IDA* is provably optimal in terms of memory usage and time complexity over all best-first searches on a tree (Korf, 1985; Mahanti et al., 1992). Note that the optimality of IDA* applies only to trees and not to graphs in general. There are two major shortcomings of using IDA* on graphs. First, if IDA* is used on directed acyclic graphs with an edge branching factor greater than the node branching factor, then it will waste time exploring alternative paths to

the same node. The second shortcoming deals with using IDA* on graphs in which cycles are present. Addressing this second shortcoming will be the focus of this chapter.

To see why cycle checking is important, one must first understand how the IDA* algorithm works. The IDA* algorithm is based on the concept of depth-first iterative deepening (DFID). A brute force DFID search uses successive depth-first searches to find the goal. A normal depth-first search stops only when there are no more nodes left to expand. The depth-first search used by DFID stops when a depth limit has been reached. This is necessary because it is not possible to mark every node which has been expanded when the search space is large. Successive depth-first searches are performed to greater and greater depths. After each depth-first search, the depth limit is increased by one. Initially, the depth limit is one. A DFID search terminates when the goal is encountered.

The IDA* algorithm improves upon brute force DFID by using a heuristic, $h(n)$. Successive depth-first searches are performed with the A* cost function $g(n) + h(n)$ used to determine when to stop searching along the current path. Search continues along a path to the next node n only if $g(n) + h(n) \leq cutoff$. Unlike brute force DFID, where *cutoff* is always incremented by one, the value of *cutoff* for the next iteration of IDA* is determined from the minimum of all cost values that exceeded *cutoff* in the current iteration. Initially, *cutoff* is the heuristic estimate of the cost from the start state to the goal state. Note that the path from the start state to the current state must be maintained as part of the depth-first search in order to obtain the path to the goal when the algorithm terminates.

Since IDA* is based on depth-first search, it will blindly wind its way through a cycle until its depth limit is reached. The algorithm can be modified to check for these cycles fairly easily (Pearl, 1984, p.39). Since the current path back to the start must be maintained, a test can be put in place which compares each node being generated to some

or all of the nodes on the current path. Two types of cycle checking will be considered here, *parent* and *full* cycle checking. Parent cycle checking discovers cycles of length two by simply comparing a newly generated node to the parent of node being expanded. Full cycle checking discovers cycles of any length by comparing a newly generated node to every node on the current path. Empirical evidence suggests that this $O(d)$ brute force search is faster than using a $O(1)$ hash table lookup because the overhead of adding and deleting nodes from the hash table is quite large and usually negates any benefit obtained from faster cycle detection.

Since cycle checking IDA* cuts off more branches of the search tree, it will have a lower effective branching factor than regular IDA*. A lower branching factor means less time spent doing node expansions. This is balanced against the extra time needed to check for cycles. When comparing two search spaces with different branching factors, IDA* will be asymptotically faster on the search space with the lower branching factor, even when the extra time for cycle checking is taken into account. The next section will provide experimental evidence which shows that full cycle checking provides a significant reduction in node expansions and search time for two types of search problems.

5.2 Test Results

Tests were run on a four-connected grid search problem, the 15-puzzle problem (Pearl, 1984), and a transportation route planning problem in order to determine the effectiveness of cycle checking. All tests were run on Sun SparcStation IPX workstations. It will be shown that parent cycle checking improves the search time of all problems tested. Full cycle checking is shown to improve the search time of the grid search problem and the route planning problem.

5.2.1 The Grid Search Problem

The objective of a grid search problem is to find a path from one point on a grid to another point on a grid, with obstacles blocking some grid positions. The results for the grid search problem are summarized in Figures 16 and 17. To make things interesting, the tests run in these figures had a single obstacle placed perpendicular to a line connecting the start and the goal. The distance between the start and the goal grid positions was varied and one test was run for each start/goal position. The air-distance heuristic was used in all tests for the grid search problem. Shown in Figure 16 is the number of node expansions as a function of the solution length. By fitting an exponential function to each curve, the branching factor was determined to drop from 2.15 with no cycle checking to 1.97 with parent checking and 1.92 with full checking.

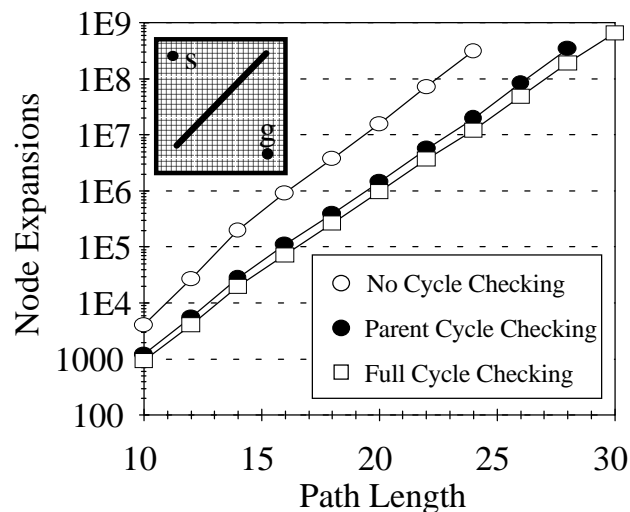


Figure 16. Cycle checking results for four connected grid.

Reducing the number of node expansions is only meaningful if there is also a corresponding decrease in the search time. Figure 17 shows the amount of CPU time needed to solve each of the grid search problems versus path length. The amount of time

needed correlates very closely with the number of node expansions. The small added cost of cycle checking is more than made up for by the reduction in node expansions. Full cycle checking IDA* is 33% faster than parent cycle checking IDA* when the path length is 28. This graph shows that for the grid search problem, it is worthwhile to use full cycle checking.

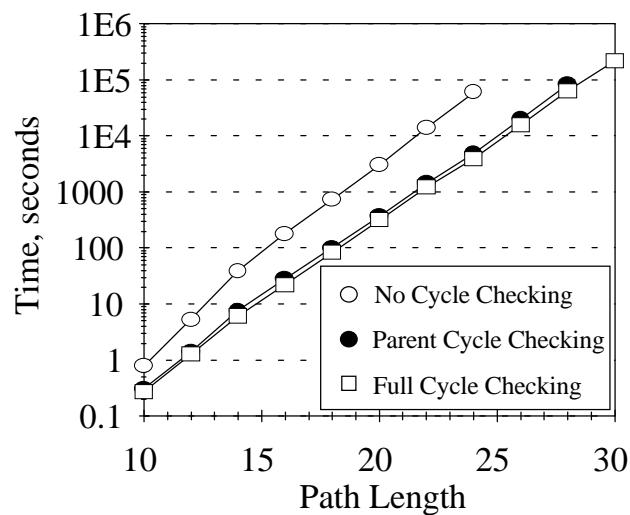


Figure 17. Cycle checking results for four connected grid.

5.2.2 The 15-Puzzle Problem

Unlike the tests for the grid search problem, the tests for the 15-puzzle problem involved solving twenty random puzzles for each path length rather than solving just one case for each path length. Each problem was solved without cycle checking, with parent cycle checking IDA*, and with full cycle checking IDA*. The Manhattan distance was used as the heuristic. The test results are summarized in Figure 18. Branching factors were computed by fitting an exponential curve through the node expansions versus path length data. This is basically equivalent to computing the mean value of the d^{th} root of the number of nodes expanded. The branching factor dropped from 1.9 without cycle checking to 1.44

with cycle checking yielding a corresponding exponential improvement in search time. The graph shows that cycle checking IDA* is about 800 times faster than non-cycle checking IDA* at path length 30. Parent cycle checking IDA* was about 13% faster than the full cycle checking IDA* at the same path length.

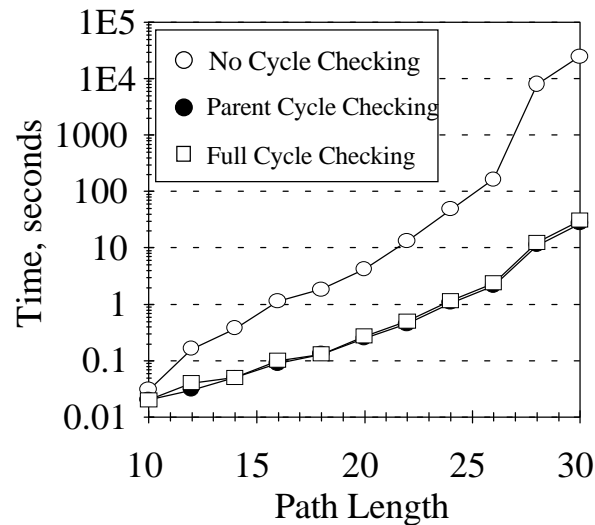


Figure 18. 15-puzzle test results.

A second test was run to see if full cycle checking would provide more of a benefit on puzzle instances with longer path lengths. The second test involved solving twenty random puzzle instances of arbitrary path lengths. Random 15-puzzle instances are intractable with non-cycle checking IDA* and hence no puzzles were solved without cycle checking. Once again, the Manhattan distance was used as the heuristic. The results are summarized in Table III. The three columns of the table provide information on six random variables: the number of node expansions for full and parent cycle checking, the elapsed time for full and parent cycle checking, and the percent improvement of parent versus full cycle checking computed using node expansions and elapsed time. The table shows that

even though full cycle checking IDA* expands less nodes, it is about 17% slower than parent cycle checking IDA*.

		Number of Tests = 20		Mean Path Length = 50.25	
		Parent Cycle Checking	Full Cycle Checking	$\frac{Parent - Full}{Parent} \times 100$	
Nodes Expansions	Mean	92,230,862	92,155,932	0.05	
	Deviation	166,398,011	166,201,336	0.06	
Elapsed Time, sec	Mean	48,686.0	60,326.2	-16.6	
	Deviation	95,248.5	116,412.2	15.8	

Table III. Summary of test results for the 15-puzzle problem.

The 15-puzzle experiments show what happens when full cycle checking is used on a simple search space with a strong heuristic. The search space for the 15-puzzle problem contains cycles of length 2 with the next larger length cycles being of length 12 and then length 16. Parent cycle checking eliminates the cycles of length 2, and the Manhattan heuristic eliminates most of the longer length cycles. For these reasons, full cycle checking IDA* should not be used on this type of problem.

5.2.3 The Route Planning Problem

The tests for the route planning problem involved finding the fastest travel-time path between two points on a Chicago area map. The map contained over 16,000 miles of roadway within a 60 mile radius of Chicago. The map did not contain any local residential streets. The map was represented as a directed graph with 8,847 nodes and 27,836 edges. The average node branching factor was 3.1. The heuristic used was the air-distance divided by 55 miles per hour. This heuristic is admissible since the maximum legal speed limit in the map area is 55 miles per hour. The cost of traversing a road segment was set equal to the

length of the segment divided by the speed limit on the segment. This assumes very light traffic conditions, a total absence of any congestion, and also ignores traffic signals.

Data was collected on a set of 100 pseudo-randomly selected (*start, destination*) pairs. The *start* and *destination* points were restricted to be within 2 to 4 miles of each other in order to reduce the variance in the test results. Parent cycle checking and full cycle checking IDA* were used to solve each of the 100 test cases. The test results are summarized in Table IV. The table shows that full cycle checking IDA* expanded 38% fewer nodes than parent cycle checking IDA* and was 31% faster. This is an example of the improvement in efficiency that can be obtained by using full cycle checking in a complex search space with a relatively weak heuristic. Tests run with longer path lengths indicated an even further improvement in the efficiency of full versus parent cycle checking.

Number of Tests = 100

		Parent Cycle Checking	Full Cycle Checking	$\frac{Parent - Full}{Parent} \times 100$
Nodes Expansions	Mean	16,263,766	3,695,788	38.8
	Deviation	48,707,689	11,589,785	33.3
Elapsed Time, sec	Mean	5,264	1,481	31.4
	Deviation	15,799	4,678	38.0

Table IV. Summary of test results for the route planning problem.

The effectiveness of full cycle checking was compared to the effectiveness of parent cycle checking by the analyzing the two hypotheses listed in Table V. The analysis was based on the unbiased, non-trivial test cases for the route planning problem domain. The significance of the hypotheses were evaluated using the difference-of-means test (Dowdy and Weardon, 1983). The results of the hypothesis tests are shown in Table VI. The quantity δ is the standard deviation and Z represents the standardized variable. The results

show that, for a two-tailed test, the difference between the average performance of parent and full cycle checking IDA* is significant at the 0.02 level for both hypotheses. Therefore, we can assert with 98% confidence that full cycle checking IDA* significantly outperforms parent cycle checking IDA* in the route planning domain.

Hyp. number	Hypothesis
1	Full cycle checking IDA* expands fewer nodes than parent cycle checking IDA* in the route planning domain
2	Full cycle checking IDA* is faster than parent cycle checking IDA* in the route planning domain

Table V. Hypotheses

Hyp. number	δ	Z	Test	Significance level	Confidence level
1	5006757	2.51	two-tailed	0.006	99.4%
2	1647.7	2.30	two-tailed	0.011	98.9%

Table VI. Results of tests of hypotheses

5.3 Summary of Cycle Checking Results

Two weaknesses of the IDA* algorithm were discussed. Both weaknesses stem from the fact that IDA* is basically a series of depth-first searches with little knowledge of previous computations. The weakness pointed out here is that each depth-first search will blindly follow a cycle if an unmodified version of the IDA* algorithm is used. A modification to the IDA* algorithm can eliminate these cycles by checking each newly generated node to determine if the node is already present on the current path. This simple modification to the IDA* algorithm was shown to significantly improve the performance of IDA* when a complex state space is searched using a weak heuristic.

Two types of cycle checking were presented. Deciding which type of cycle checking to use depends mainly on two factors: heuristic pruning power and search space complexity. The 15-puzzle problem showed that longer length cycles will be eliminated by a strong heuristic and hence obviate the necessity of doing full cycle checking. Define a simple search space as a search space with few cycles of length greater than two. The 15-puzzle search space is an example of a simple search space. Full cycle checking should not be used in a simple search space since the cost of checking for cycles of length greater than two will negate any benefit obtained by reducing the number of node expansions. Table VII provides a guide to the type of cycle checking which should be used on a given problem. The entries marked with a "?" indicate that experimentation should be used to determine which type of cycle checking works best.

	Simple Search Space	Complex Search Space
Weak Heuristic	?	Full Cycle Checking
Strong Heuristic	Parent Cycle Checking	?

Table VII. Guide to cycle checking type to use

There are other methods of reducing the complexity of IDA* that are notable. Sen and Bagchi present an algorithm called MREC which uses a fixed amount of memory to store nodes (Sen and Bagchi, 1989). This not only eliminates cycles, but it also prevents the same node from being generated more than once. When memory runs out, however, MREC reverts back into an IDA* type search and hence the cycle checking modifications presented here can also be applied to MREC. Also note that cycle checking IDA*, unlike MREC, does not require any additional memory usage beyond what IDA* would normally use. Taylor and Korf also present a novel method of eliminating cycles by using a finite

state automaton to generate moves (Taylor and Korf, 1992). The automaton is constructed by performing a breadth-first search prior to the use of IDA*. The drawback to this approach is that it is only useful for problems which are described implicitly, like the 15-puzzle problem or the Towers of Hanoi problem. The full cycle checking approach presented here, however, can potentially improve the node expansion efficiency of IDA* in any problem domain.

6. PERIMETER SEARCH

A new type of heuristic search algorithm is presented in this chapter. This admissible algorithm is referred to as perimeter search since it relies on a perimeter of nodes around the goal. The perimeter search algorithm works as follows: First, the perimeter is generated by a breadth-first search from the goal to all nodes at a given depth ∂ . The path back to the goal along with each perimeter node's state descriptor are stored in a table. The search then proceeds normally from the start state. During each node generation, however, the current node is compared to each node on the perimeter. If a match is found, the search can terminate with the path being formed with the path from the start to the perimeter node together with the previously stored path from the perimeter node to the goal. Both analytical and experimental results are presented to show that perimeter search is more efficient than IDA* and A* in terms of time complexity and number of nodes expanded for two problem domains.

6.1 Introduction to Perimeter Search

The perimeter search algorithm can be categorized as a bidirectional search algorithm. As with other bidirectional search algorithms (Champeaux and Sint, 1977; Kwa, 1989; Pohl, 1971; Politowski and Pohl, 1971), perimeter search requires a state space with reversible operators and a heuristic which can estimate the distance between any two nodes. Also like other bidirectional search algorithms, perimeter search attempts to reduce the complexity of heuristic search by splitting the search space between two separate searches; one from the start, and another from the goal. Unlike other bidirectional search algorithms, perimeter search avoids some of the pitfalls of attempting to do the two searches

simultaneously. After performing a depth-limited breadth-first search from the goal, perimeter search proceeds as a unidirectional search from the start.

Pohl's original bidirectional heuristic path algorithm (BHPA) (Pohl, 1971), attempts to reduce the complexity of heuristic search by performing two simultaneous searches, one from the start to the goal, the other from the goal to the start. The algorithm finds successively better paths as the two searches collide with each other. The algorithm terminates when no further improvements can be made to the path length. A best-case scenario for BHPA would be for the two searches to "meet in the middle" initially. This would reduce a $O(b^d)$ search into two $O(b^{d/2})$ searches. Unfortunately, the two search spaces often do not meet at a midpoint. The result of this, combined with satisfying BHPA's admissibility conditions, is that BHPA can end up expanding more nodes than a unidirectional algorithm.

De Champeaux and Sint eliminated the "meet in the middle" problem with their BHFFA algorithm by using a heuristic which guides the nodes in each wavefront toward the closest node in the opposing wavefront. This guarantees that the first node found by BHFFA is the best one, but the time complexity of the heuristic used increases proportionally with the size of the opposing wavefront. Compounding this problem is the fact that the heuristic value for a node cannot be stored and must be recomputed every time the opposing wavefront changes.

Politowski and Pohl created a bidirectional search algorithm similar to BHFFA, but with the heuristic changed so that the search was guided toward only one special node in the opposing wavefront called the d-node. The d-node is the node in the opposing wavefront furthest from the start(goal). The d-node retargeting algorithm works by performing n node expansions towards the d-node, switching directions, recomputing the

heuristic values for the new d-node, and repeating the process until the wavefronts meet. Unfortunately, the d-node retargeting algorithm is inadmissible and requires a careful choice for the value of n . Choosing too large a value for n leads to a unidirectional search. Making n too small leads to lower quality paths and more overhead for recomputing the heuristic values (Politowski and Pohl, 1971).

One of the most recent developments in bidirectional search algorithms is Kwa's BS^* algorithm. The BS^* algorithm is derived from Pohl's BHPA algorithm with refinements added to eliminate excessive node expansions. The BS^* algorithm is provably admissible and has been shown to outperform BHPA. Unfortunately, the test results given in (Kwa, 1989) and in Section 6.5 show that BS^* is inferior to A^* not only in terms of running time, but also in terms of node expansions. This is probably due to the extra overhead necessary for nipping and pruning useless nodes from the OPEN lists.

The best admissible bidirectional search algorithms considered here, BHFFA and BS^* , take two different approaches to dealing with the "meet in the middle" problem. The BHFFA algorithm forces the two wavefronts to meet in the middle, but in doing so increases the cost of the heuristic exponentially. The BS^* algorithm relies upon the ability to eliminate useless nodes once the wavefronts have met in order to reduce the number of node expansions. Perimeter search takes a third approach to dealing with this problem: generating one of the wavefronts with a breadth-first search and limiting the size of the wavefront.

Limiting the size of one of the wavefronts has many important advantages. First, the wavefront only needs to be generated once for each possible goal. This allows the cost of performing the breadth-first search to be amortized over many problem instances, if necessary. In most cases the optimum wavefront size will be so small that the time it takes

to generate it is insignificant. Next, fixing a depth limit on one side of the search avoids the wasteful necessity of retargeting toward a continually changing opposing wavefront as is done in the BHFFA and the d-node retargeting algorithms. Finally, the time complexity of the perimeter search algorithm can be analyzed and the optimum wavefront size can be calculated. Section 6.3 will show how these advantages allow perimeter search to substantially outperform other admissible heuristic search algorithms.

6.2 Perimeter Search Algorithms

The perimeter search algorithm consists of two searches performed in sequence. The first search, a depth-limited breadth-first search from the goal, generates the perimeter, see Figure 19. The second search proceeds from the start using $\text{Min}_{m \in P}(h(n,m) + h^*(m,\Gamma))$ as the heuristic, where n is the node being expanded, P is the set of perimeter nodes, and $h^*(m,\Gamma)$ is the cost of the least cost path from perimeter node m to Γ . Since the two searches are completely independent, any type of heuristic search algorithm can be used for the second search. Discussion will be limited to using A* (Nilsson, 1980) and IDA* (Korf, 1985). Perimeter search with A* will be denoted as PS* and perimeter search with IDA* will be denoted as IDPS*.

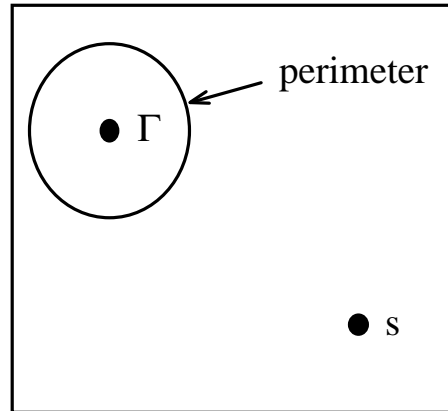


Figure 19. Hypothetical search space showing perimeter around goal Γ .

In addition to modifying the heuristic used in the second search, a test needs to be put in place which compares the node selected for expansion against the nodes in the perimeter. If a match is found, the second search can terminate with the solution path being formed from the path to the perimeter node together with the previously stored path from the perimeter node to the goal.

Both PS^* and $IDPS^*$ use a heuristic similar to the one in BHFFA to force the heuristic search towards the perimeter. That is, $\text{Min}_{m \in P}(h(n,m) + h^*(m,\Gamma))$ is used instead of $h(n)$ as the heuristic. For problem domains with unity cost arcs, $h^*(m,\Gamma)$ is equal to the perimeter depth ∂ . Since ∂ is constant throughout the search, it can be omitted leaving $\text{Min}_{m \in P}h(n, m)$ as the heuristic for unity cost problems.

Assuming $h(n,n) = 0$, it is only necessary to check for a collision with the perimeter when $\text{Min}_{m \in P}h(n,m) = 0$. Once a node in the perimeter has been selected for expansion, the search can terminate. The optimal path from s to Γ can be formed by appending the path from p to Γ onto the path from s to p , where p is the perimeter node encountered. Since

PS^* and $IDPS^*$ are based upon admissible algorithms, and since $\text{Min}_{m \in P}(h(n,m) + h^*(m,\Gamma))$ is an admissible heuristic, both PS^* and $IDPS^*$ are admissible.

The perimeter search algorithm does not require the perimeter depth ∂ to be a natural number. If the problem domain has real valued costs, then the depth-limited breadth-first search used to generate the perimeter should only add those nodes to the perimeter which have a depth greater than or equal to ∂ . See Figure 20 for an algorithm which can generate perimeters for any search domain.

```

procedure makePerimeter(State goal, Real d, var Perimeter P)
  add goal to the back of the OPEN queue, set  $g(goal) = 0$ 
  while OPEN is not empty
    Remove  $n$  from the front of the OPEN queue
    if CLOSED set does not include  $n$  then
      Add  $n$  to CLOSED set
      if  $g(n) < d$  then
        For each successor  $n_i$  of  $n$  do begin
          Add  $n_i$  to OPEN, set  $g(n_i) = g(n) + c(n, n_i)$ 
        end for
      else
        Add  $n$  to perimeter  $P$ 
      end if
    end if
  end while
end procedure makePerimeter

```

Figure 20. Algorithm for generating perimeter P of depth d

The perimeter need not be generated using a breadth-first depth limited search as shown in Figure 20. In fact, any set of nodes which encompass the goal will work as a perimeter. One such alternative method of generating the perimeter would be to use A^* . This would generate a constant evaluation perimeter because all nodes on the perimeter would have approximately the same f value. The algorithm listed in Figure 20 generates

constant depth perimeters because all the nodes on the perimeter have approximately the same g value. This chapter will focus primarily on constant depth perimeters for the following reasons: First, analyzing the number of node expansions is much easier with constant depth perimeters. Second, constant evaluation perimeters are more expensive to generate since an agenda must be maintained by A^* . Third, constant evaluation perimeters can only be used for one start/goal pair whereas constant depth perimeters can be reused as long as the goal remains the same. Finally, empirical results show that there is no significant advantage to using constant evaluation perimeters instead of constant depth perimeters, see Section 6.7.

6.3 Analyzing Perimeter Search

A simple analysis of the perimeter search algorithm can be used to help decide when perimeter search will improve the search efficiency for a particular problem domain. The same analysis can also be used to determine the optimal value of ∂ to use. In the absence of such an analysis, empirical results based on small problem instances may be used in order to determine if perimeter search is useful for a particular problem domain.

Let t be the amount of CPU time necessary per node expansion with one heuristic evaluation to a single goal. This analysis assumes that t remains constant throughout the search. Let f be the fraction of time t spent evaluating the heuristic $h(n,m)$. Let n_∂ be the number of nodes expanded by the forward search when a perimeter of depth ∂ is used. Finally, let η be the number of goals.

The time needed to evaluate h once for each goal during a single node expansion is $t/f\eta$. The time needed for the rest of the operations during a node expansion is $t(1-f)$. Since evaluating $\text{Min}_{m \in P} h(n,m)$ requires $|P_\partial|$ heuristic evaluations for each goal (in the

worst case, see Section 6.4), the total amount of time spent evaluating the heuristic over all node expansions is $|P_\partial| t f \eta n_\partial$. The total amount of time spent on non-heuristic operations is $(1 - f) t n_\partial$. Let n_{BFS} be the number of nodes expanded when creating the perimeter for one goal. Assuming the cost per node expansion is t for the backwards search, the total running time of a perimeter search algorithm is

$$R^*(t, P_\partial, f, n_\partial, d, \eta) = (|P_\partial| f \eta + 1 - f) t n_\partial + t \eta n_{BFS}. \quad (1)$$

Assuming a tree search, n_∂ can be estimated as $b^{L-\partial}$ where b is the effective branching factor and L is the cost of a least cost path from s to Γ . The perimeter size and the number of nodes expanded to generate the perimeter can be estimated with B^∂ where B is the brute force branching factor. Substituting these into equation (1) yields

$$R(t, B, f, b, \partial, L, \eta) = (B^\partial f \eta + 1 - f) t b^{L-\partial} + t \eta B^\partial. \quad (2)$$

Equation (2) is plotted in Figure 21 for $b=1.44$, $B=2.13$, $t=1$, $L=53$, and various values of f . The values for b and L correspond to empirical estimates of the branching factor and average path length for the 15-puzzle problem when using IDA* and the Manhattan distance heuristic (Pearl, 1984). The value for B was computed exactly, see (Taylor and Korf, 1992).

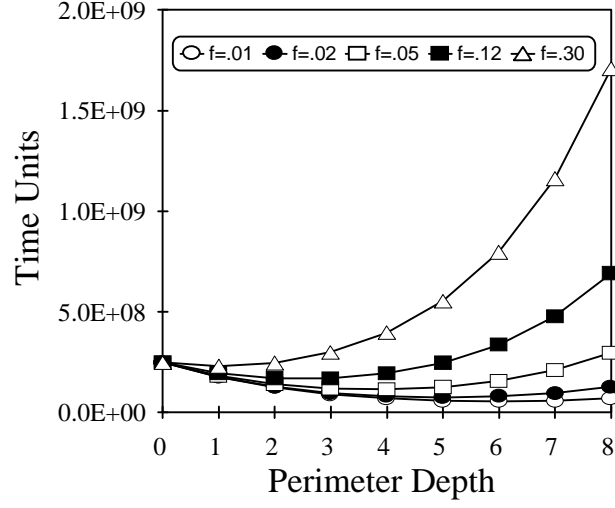


Figure 21. Analytical time complexity of perimeter search for the 15-puzzle problem

Figure 21 shows that the usefulness of perimeter search for the 15-puzzle problem depends critically on the fractional amount of time spent evaluating the heuristic. For values of f greater than about 0.3, perimeter search will no longer improve the search efficiency. The usefulness of perimeter search can be determined for other problem domains as well. Taking the derivative of equation (2) with respect to ∂ yields

$$\frac{dR}{d\partial} = b^{L-\partial} t f \log b - b^{L-\partial} t \log b + b^{L-\partial} B^\partial t f \eta \log B - b^{L-\partial} B^\partial t f \eta \log b + B^\partial t \eta \log B. \quad (3)$$

Perimeter search is no longer useful when the minimum R occurs at $\partial \leq 0$. Substituting $\partial = 0$ into equation (3) and solving for $dR/d\partial = 0$ reveals that perimeter search will be useful if

$$f \leq \frac{b^L \log b - \eta \log B}{b^L (\log b - \eta \log b + \eta \log B)}. \quad (4)$$

For the case with one goal and for sufficiently large values of L , equation (4) simplifies to

$$f \leq \frac{\log b}{\log B}. \quad (5)$$

Equation (5) provides a simple test to determine when perimeter search will improve search performance. Note that equation (5) only applies to problem domains where ∂ can be any real number. For problem domains like the 15-puzzle problem where ∂ must be an natural number, $\partial = 1$ is substituted into equation (3) and $dR/d\partial = 0$ is solved to yield

$$f_{nat} \leq \frac{b^L \log b - bB\eta \log B}{b^L (\log b - B\eta \log b + B\eta \log B)}. \quad (6)$$

For the case with one goal and for sufficiently large values of L , equation (6) simplifies to

$$f_{nat} \leq \frac{\log b}{\log b - B \log b + B \log B}. \quad (7)$$

Substituting in the appropriate values of $b = 1.44$ and $B = 2.13$ into equation (7) reveals that perimeter search will be useful for the 15-puzzle problem if f is less than or equal to 0.304.

Equations (4) through (7) are only valid under the assumption of a tree search. For problem domains where this is not a valid assumption, a different analysis must be done based either on empirical equations for n_∂ and n_{BFS} , or else exact equations for n_∂ and n_{BFS} can be determined using methods similar to those found in (Pearl, 1984). Such an analysis has been done for the grid search problem domain, see Section 6.9.

Once the utility of perimeter search has been verified, the next step is to determine the value of ∂ which yields the minimum time complexity. This can be achieved by setting equation (3) equal to zero and solving for ∂ . Unfortunately, there is no closed form solution to this equation and a numerical analysis must be used instead. Table VIII lists out some optimum ∂ values for the 15-puzzle problem with one goal. Also shown in the table is the speedup obtained by using perimeter search. Speedup is defined here as the ratio

$$Speedup = \frac{R(t, B, f, b, 0, L, 1)}{R(t, B, f, b, \partial, L, 1)}. \quad (8)$$

f	Optimum ∂	<i>Speedup</i>
0.01	6	4.6
0.02	5	3.3
0.05	4	2.2
0.1	3	1.6
0.2	2	1.2
0.3	1	1.1

Table VIII. Optimum values of ∂ for the 15-puzzle problem and the corresponding search speedup

6.4 Minimizing Heuristic Evaluations

Table VIII shows that the performance of perimeter search is dependent on the amount of time spent evaluating the heuristic. Many of the methods for improving the heuristic evaluation time are implementation dependent. However, there is a general technique which can be used with IDPS* or multiple goal IDA* to minimize the number of heuristic evaluations.

At first it may seem that using $\text{Min}_{m \in P} h(n, m)$ as a heuristic requires $|P|$ heuristic evaluations per node expansion. This is not the case, however, if the heuristic is monotonic. Recall that if a heuristic is monotonic, then

$$|h(n) - h(m)| \leq c(m, n) \quad (9)$$

holds for all nodes n and m , where $c(n, m)$ is the cost of traversing the arc between m and n . With a monotonic heuristic, the estimate of the distance from a node n to a perimeter node p can only change by at most $c(m, n)$, where m is the parent of n . The result is that a heuristic value does not need to be recomputed if its value cannot change enough to affect the minimum.

The technique for reducing the number of heuristic evaluations is now as follows. The heuristic is used once at the beginning of the search to estimate the distance from the start to each perimeter node and this estimate is then stored along with the node. Next, when a node n is generated from a node m , the heuristic value of the current minimum is recomputed using h . The arc cost $c(m,n)$ is then subtracted from the rest of the stored heuristic values. If any of the stored heuristic values falls below the minimum, it is recomputed using h also. Using this technique with IDPS* and a perimeter containing 4 nodes, the number of heuristic evaluations was reduced by 74% when solving 25 random 15-puzzles .

Note that this technique for reducing the number of heuristic evaluations requires $O(P)$ memory to store the heuristic values for every perimeter node. This is the reason the technique cannot be used with PS*, it would require $O(P)$ memory for each node in the OPEN list. IDPS* does not have this problem because the heuristic values only need to be stored with each node along the current path.

6.5 Test Results

The theoretical analysis of Section 6.3 indicate that perimeter search can improve search efficiency quite dramatically. This section will support the theoretical analysis through experimental results which show that perimeter search improves search efficiency for the 15-puzzle and Think-A-Dot problems. All tests in this section were run on Sun SparcStation IPX workstations. The elapsed time was calculated based on the number of CPU seconds required to solve a problem. The elapsed time for each PS* and IDPS* test includes the time needed to generate a perimeter.

6.5.1 15-Puzzle Problem

The 15-puzzle problem is a sliding tile problem based on a game in which fifteen numbered tiles are placed in a four by four grid. The objective of the game is to arrange the tiles in a specified pattern by moving the tiles, one at a time and without crossing over each other, into the one blank position. The IDA* and IDPS* algorithms were used to solve random instances of the 15-puzzle problem. The A*, PS*, and BS* algorithms could not be used due to their large memory requirements.

One of the best heuristics available for the 15-puzzle problem is the Manhattan distance heuristic. The Manhattan distance heuristic estimates the number of moves needed to solve a sliding tile puzzle by summing up the distance between each tile and its “home” position. All 15-puzzle problems were solved using the Manhattan distance heuristic.

Perimeter search was shown in Section 6.3 to be useful for the 15-puzzle problem only if $f \leq 0.3$. The actual value of f for the Manhattan distance heuristic was estimated to be approximately 0.02 by solving twenty “easy” problems. Easy problem instances were generated by making a limited number of random moves from the goal state. The low f value results from computing the Manhattan distance heuristic incrementally using a lookup table based on the current blank position, the new blank position, and the tile being moved. This low f value indicates that perimeter search should provide a speedup of at least 3, see Table VIII.

The test results for the 15-puzzle problem are summarized in Figure 22 and Table IX. Each point in Figure 22 represents the average elapsed CPU time required to solve 100 random 15-puzzles. Note that the point for perimeter depth 0 represents IDA* search and the remaining points are IDPS* searches with different perimeter depths. Table IX shows the mean and standard deviation of six random variables: the number of

node expansions for IDA* and IDPS*, the amount of CPU time for IDA* and IDPS*, and the speedup computed using node expansions and CPU time. Note that the last column in the table represents the mean of a ratio and not the ratio of two means. The test results show that IDPS* at depth 4 attained the highest speedup; it was 3.8 times faster than IDA*. This speedup surpasses the theoretical speedup of 3.3, see Table VIII.

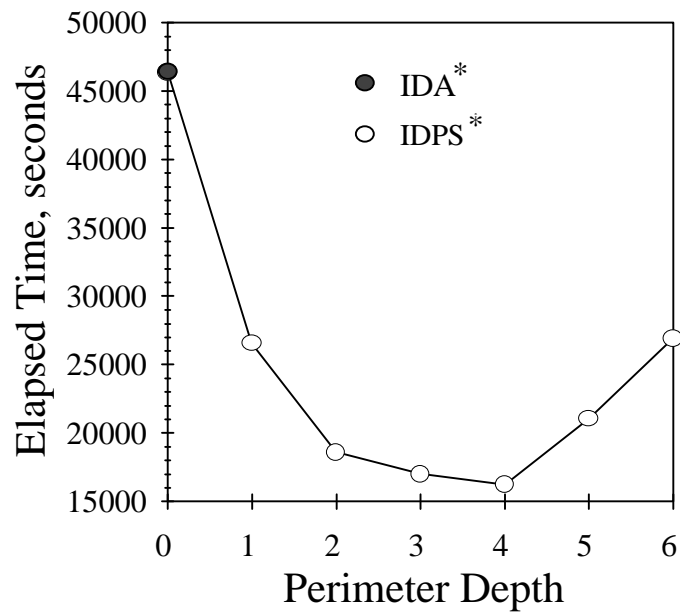


Figure 22. Test results for 15-puzzle

Number of Tests = 100

Mean Path Length = 50.6

		IDA*	IDPS*, $\partial = 4$	$\frac{IDA^*}{IDPS^*}$
Nodes Expansions	Mean	131,809,315	35,990,122	4.96
	Deviation	305,101,594	102,444,848	3.27
Elapsed Time, sec	Mean	46,416	16,225	3.85
	Deviation	115,069	44,772	2.68

Table IX. Summary of test results for the 15-puzzle problem.

6.5.2 Think-A-Dot

The Think-A-Dot problem (Mostow and Prieditis, 1989) is based on a game in which a ball drops through a series of levers, see Figure 23. Each lever causes the ball to drop to either the left or the right. After a ball passes by a lever, the lever switches directions so that the next ball to pass by will go the opposite direction. The objective of the game is to place all the levers in a given position by dropping balls, one at a time, into one of the four positions in the top layer. The number of moves needed to solve a Think-A-Dot problem can be estimated by finding the maximum number of incorrectly oriented levers in a single layer.

The utility of using the perimeter search algorithm was determined by estimating the values of B , b , and f by solving several easy problem instances. The value for L was assumed to be large enough so that equation (7) could be used. The other values, when substituted into equation (7), indicated that perimeter search could be used to improve

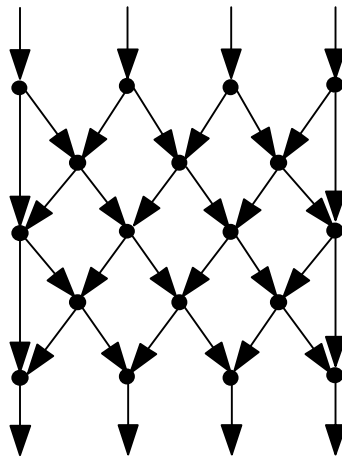


Figure 23. Think-A-Dot Problem used in tests

search times. The optimal value for ∂ was determined to be 4 by finding the minimum value of equation (2).

The A^* , BS^* , and PS^* algorithms were used to solve 100 random Think-A-Dot problems. An augmented red-black tree (Bayer, 1972) was used for maintaining the OPEN lists used by these algorithms, resulting in $O(\log n)$ complexity for all operations performed on the lists, where n is the number of nodes in the list. The test results are summarized in Figure 24 and Table X. Each point in Figure 24 represents the average elapsed CPU time required to solve 100 random Think-A-Dot problem instances. Table X contains the mean and deviation of eight random variables: the number of node expansions for A^* , BS^* , and PS^* , the amount of CPU time for A^* , BS^* and PS^* , and the speedup relative to A^* computed using node expansions and CPU time. The results for A^* are shown at perimeter depth 0 in Figure 24. The results show that PS^* at depth 4 is 1.7 times faster than A^* . The results of the BS^* tests are given in Table X. These results show that the BS^* algorithm expanded more nodes and took more time to solve the problems than either A^* or PS^* . The perimeter search algorithm PS^* had, on average, the least number of node expansions and required the least amount of running time to solve the problems.

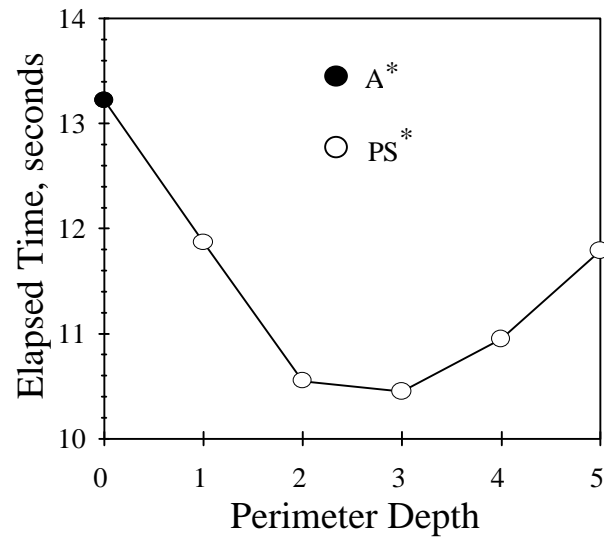


Figure 24. Test results for Think-A-Dot

Number of Tests = 100 Mean Path Length = 18.4

		A*	BS*	PS*, $\partial = 4$	$\frac{A^*}{PS^*}$
Nodes Expansions	Mean	5,369	5,479	2,673	4.11
	Deviation	4,214	4,782	2,916	5.13
Elapsed Time, sec	Mean	13.2	38.7	10.9	1.69
	Deviation	10.17	52.3	11.8	0.53

Table X. Summary of test results for the Think-A-Dot problem.

6.6 Near Optimal Perimeter Search

A near-optimal version of the perimeter search algorithm exists and may be useful for certain problems. The near-optimal perimeter search algorithm differs from the optimal algorithm in that the heuristic used is $h(n)$ and not $\text{Min}_{m \in ph(n,m)}$. Both versions of perimeter search must check to see if a node on the perimeter has been encountered. When

a perimeter node is encountered using $\text{Min}_{m \in \rho} h(n, m)$ as the heuristic, we are guaranteed to have an optimal path from s to Γ . When a perimeter node is encountered using a monotone heuristic, $h(n)$, we are only guaranteed an optimal path from s to the particular perimeter node encountered. This perimeter node may or may not be on an optimal path from s to Γ and so the algorithm is inadmissible. It can be shown, however, that the solution cost obtained using near-optimal perimeter search is at most $L + \partial$, where L is the cost of the least cost path from s to Γ and ∂ is the perimeter depth.

Theorem 5: Near-optimal perimeter search is ϵ -admissible.

Proof: Since every path to Γ contains at least one perimeter node, near optimal perimeter search must either expand one perimeter node or else the goal is unreachable. Let the perimeter node encountered be n and let the perimeter node which is on the optimal path from s to Γ be n' , see Figure 25. Since n was expanded before n' , it must be the case that $f(n) \leq f(n')$. Substituting $f = g + h$ and rearranging, we get

$$g(n) \leq g(n') + h(n') - h(n). \quad (10)$$

Equation (10) defines an upper bound on the cost of the path found by near-optimal perimeter search. Because the optimal path from any perimeter node to Γ is of length ∂ , and since the heuristic is admissible, we have $0 \leq h(n) \leq \partial$ and $0 \leq h(n') \leq \partial$. A worst case scenario has $h(n) = \partial$ and $h(n') = 0$, in which case equation (10) simplifies to

$$g(n) \leq g(n') + \partial. \quad (11)$$

Assuming a monotone heuristic (Nilsson, 1980), the first path found by near-optimal perimeter search to a node n will be the best path, so g^* may be substituted for g in equation (11) yielding

$$g^*(n) \leq g^*(n') + \partial. \quad (12)$$

Substituting equation (12) into $f^*(n) = g^*(n) + h^*(n)$ yields

$$f^*(n) \leq g^*(n') + h^*(n) + \partial. \quad (13)$$

Since $h^*(n)=h^*(n')=\partial$ and since $f^*(n')=g^*(n')+h^*(n')$ we get

$$f^*(n) \leq f^*(n') + \partial. \quad (14)$$

A heuristic search algorithm is said to be ϵ -admissible when the path found by the algorithm is not worse than a factor of $1+\epsilon$ from optimal. Since $f^*(n')$ is the optimal path length and since $f^*(n)$ is the length of the path found by near-optimal perimeter search, equation (14) shows that near-optimal perimeter search is ϵ -admissible with

$$\epsilon = \partial / f^*(n') \quad (15)$$

In fact, equation (14) shows that near-optimal perimeter search is in some respects better than other ϵ -admissible algorithms since near-optimal perimeter search has an *absolute* bound and not a *relative* bound on its worst case path length. ■

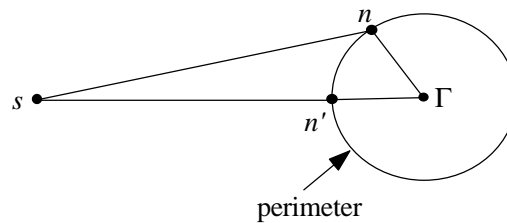


Figure 25. Hypothetical search space showing path found by near-optimal perimeter search and optimal path

Figure 26 shows the dramatic speedup possible when using near-optimal perimeter search. For each perimeter depth, twenty random 15-puzzles were solved and the average time needed to find a path to a state in the perimeter is plotted. The search time for IDA* is shown in Figure 26 at depth 0 for comparison. Near-optimal perimeter search is over 11 times faster than IDA* when a perimeter of depth 14 is used.

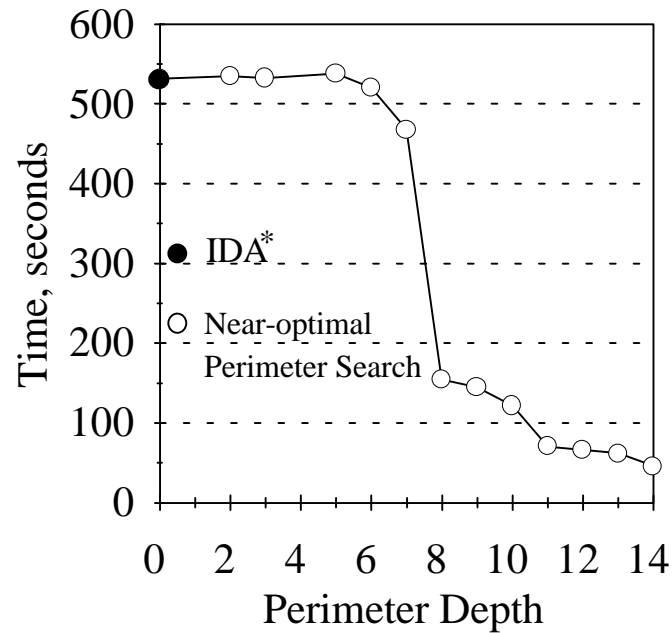


Figure 26. Search times for near-optimal perimeter search

Although the search time decreases, the path length increases because near-optimal perimeter search is not admissible. Fortunately, as shown in Figure 27, the path lengths are very close to optimal. According to Theorem 5, the path lengths could be as much as 14 moves from being optimal. In fact, the path lengths are no more than three moves from being optimal when a perimeter of depth 14 is used, a very encouraging result.

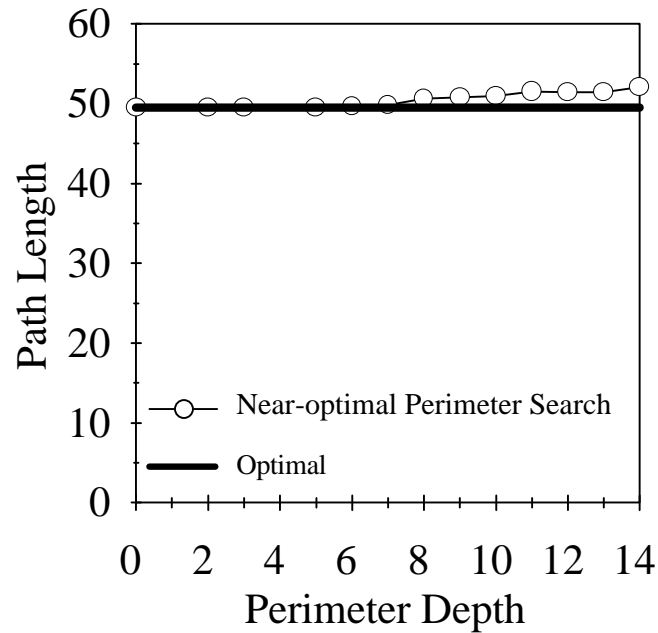


Figure 27. Path lengths for near-optimal perimeter search

6.7 Constant Evaluation Perimeters

As discussed in Section 6.2, there is the possibility of using a constant evaluation perimeter instead of a constant depth perimeter in the perimeter search algorithms. Recall that a constant depth perimeter is one in which every perimeter node n has $g(\Gamma, n) = \partial$ and a constant evaluation perimeter is one in which $f(\Gamma, n) = k$, where k is some constant. Unlike ∂ in constant depth perimeters, the constant k used in constant evaluation perimeters is not as important since the use of an imperfect heuristic makes choosing an optimal value for k difficult. Therefore, it is the size of constant evaluation perimeters (number of nodes in the perimeter) which is used to characterize the perimeter.

The procedure for generating a constant evaluation perimeter would be as follows: The search would start at Γ and search toward s using A^* until the size of the OPEN list reached some predetermined limit. The nodes in the OPEN list are then placed into the

perimeter and the paths to the goal are computed using the parent pointers maintained by the A^* algorithm. Because the node expansion rate of A^* is slower than the node expansion rate of a breadth-first search, it will take more time to generate a constant evaluation perimeter than a constant depth perimeter.

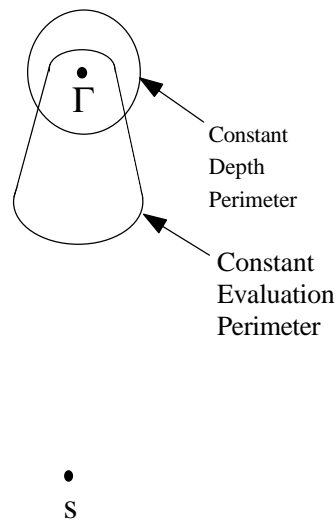


Figure 28. Graphical depiction of two types of perimeters

Even though a constant evaluation perimeter takes longer to generate, one might expect better search performance since the bulk of the perimeter nodes would tend to be closer to the start node s , see Figure 28. To test this theory, twenty random 15-puzzles were solved twice, once using each type of perimeter. The perimeters used in each case were either the same size or as close to the same size as possible. The results of the tests are summarized in Table XI. The table shows there is no significant performance improvement when using a constant evaluation perimeter over a constant depth perimeter.

Number of Tests = 20

Perimeter Size (nodes)	Constant Depth (seconds)	Constant Eval (seconds)	$\frac{\text{ConstantEval} - \text{ConstantDepth}}{\text{ConstantEval}} \times 100$
4	17,238	18,213	5.6
10	23,098	25,429	-5.0
24	34,519	41,084	9.7

Table XI. Comparison of constant depth and constant evaluation perimeters

6.8 Parallel Perimeter Search

One of the main advantages of bidirectional search algorithms is the possibility of performing the forward and backward searches in parallel. Even though the perimeter search algorithm does not perform simultaneous searches from the start and from the goal, there are still plenty of opportunities for parallelization. If a single processor is assigned to each node in the perimeter, then each processor will search a relatively disjoint portion of the search space. However, the main advantage of this type of parallel search is that almost no communication is required between the processors. This allows searches to be conducted on a large number of distributed computing systems with little or no communication overhead.

To test this theory, the parallel perimeter search algorithm was implemented using a client/server methodology. The server generates the perimeter and distributes perimeter nodes to the clients when requested. The server also keeps track of the paths found by the clients. The clients periodically compare their best estimate of the search cost ($f(n)$ for A^* and *cutoff* for IDA^{*}) to the cost of the best path found so far. If the client's best estimate is greater than or equal to the best path cost, then the client stops its search and requests another perimeter node. When all perimeter nodes have been served out and when all

clients have terminated, then the optimal path is output by the server process and the parallel perimeter search algorithm terminates.

The parallel perimeter search algorithm was used to solve 50 random 15-puzzle problems. The processors used were Sun SparcStation IPX computers. The computers were networked together with ethernet cables and communicated with each other using connected sockets. Two sets of tests were run, one with 10 processors and one with 24 processors. The results of the tests are shown in Table XII. The uniprocessor time used in the calculation of the efficiency is based on the IDA* algorithm (Korf, 1985). The table shows a disappointingly small efficiency rating.

There are a number of reasons for this low efficiency rating. First, the parallel perimeter search algorithm was implemented in C++, was meant to be as general as possible, and allowed clients to use either PS* or IDPS*. Unfortunately, overheads associated with C++ virtual functions are estimated to have lowered the efficiency rating by about 30% (the efficiency calculated using node expansions was 80% for the 10 processor case). Another reason for the low efficiency rating is the fact that a lot of work is being duplicated among the various processors. There is no mechanism for controlling which processors work on which part of the search space.

Number of Processors	Speedup	Efficiency
10	4.9	49%
24	7.3	30%

Table XII. Results of parallel perimeter search tests

An analysis of the efficiency of the parallel perimeter search algorithm reveals that the efficiency should drop as a function of the number of processors. This is due to the fact that the number of processors increases faster than expected improvement in search time. To see this, note that the number of processors is equal to the perimeter size. Assuming a tree search, there will be B^∂ processors for a constant depth perimeter of depth ∂ . The search time will be proportional to the maximum number of nodes expanded by any one processor. Assuming that the processors progress at an equal rate, the processors assigned the closest perimeter node will terminate first with $b^{L-\partial}$ node expansions. The other processors will stop searching once their best estimate of the path cost exceeds $L - \partial$ and so they too will terminate with $b^{L-\partial}$ node expansions. The efficiency of a parallel algorithm is equal to the speedup divided by the number of processors. For the parallel perimeter search algorithm,

$$Efficiency = \frac{b^L}{B^\partial b^{L-\partial}} = \left(\frac{b}{B}\right)^\partial. \quad (16)$$

Since the effective branching factor b is smaller than the brute force branching factor B , the efficiency of parallel perimeter search will decrease as more processors are added.

6.9 Perimeter Search for the Grid Search Problem

An equation for evaluating the effectiveness of perimeter search was given in Section 6.3. This equation assumes, however, that the search space of the problem domain is a tree with a uniform branching factor. This type of model is sufficient for most problem domains. Problem domains which contain many cycles or many paths to the same node are the exception. The grid search problem domain is one example of a problem domain which should not be modeled with a tree. This section will present an analysis of the performance

of the perimeter search algorithm for the grid search problem domain. This analysis for perimeter search is based upon a similar analysis for A^* found in (Pearl, 1984).

This analysis is based on the fact that the A^* algorithm expands all nodes n for which $f(n) \leq L$, where L is the cost of the least cost path from s to Γ (Pearl, 1984). This also holds true for the PS^* algorithm since PS^* is based on A^* . The number of nodes expanded by PS^* can be calculated by counting the number of nodes in the search space which have $f(n) \leq L$.

The grid search problem used for the analysis is show in Figure 29. The start s is at grid location $(0,0)$ and the goal Γ is at grid location (m,n) . There are no obstacles. The search space has been divided into four quadrants labeled I, II, III and IV. The number of nodes expanded Z_i will be calculated for each quadrant ($i = I..IV$) and then summed up to determine the total number of nodes expanded, Z .

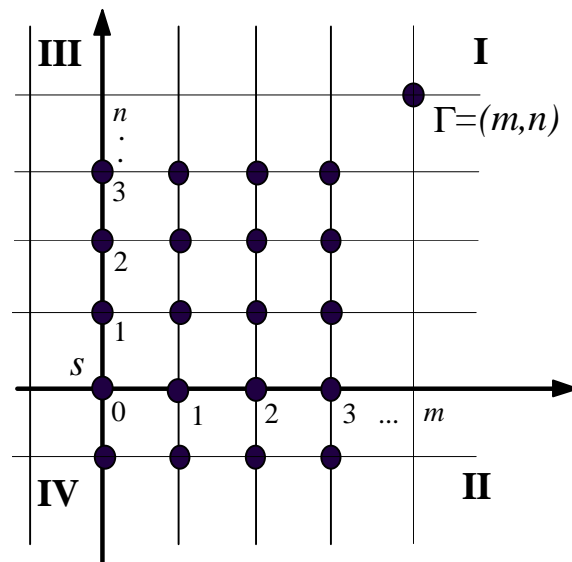


Figure 29. Grid search problem used in analysis.

For this analysis, a constant depth perimeter of depth 1 will be assumed. This means there are four perimeter nodes at the grid locations $(m-1,n)$, $(m,n-1)$, $(m+1,n)$ and $(m,n+1)$ for the four-connected grid search problem. The heuristic for this perimeter is

$$h(x, y) = \text{Min} \left(\begin{array}{l} \sqrt{(x - m + 1)^2 + (y - n)^2}, \\ \sqrt{(x - m - 1)^2 + (y - n)^2}, \\ \sqrt{(x - m)^2 + (y - n + 1)^2}, \\ \sqrt{(x - m)^2 + (y - n - 1)^2} \end{array} \right) + 1, \quad (17)$$

where (x,y) is the current grid position. The quantity $g(x,y)$ is the distance from the start state s to the current state, so

$$g(x, y) = |x| + |y|. \quad (18)$$

Note that slightly modified notations are used here for h and g . The number of nodes expanded is equal to the number of nodes for which the inequality

$$h(x, y) + g(x, y) \leq m + n \quad (19)$$

holds since the shortest path from s to Γ is of length $m + n$.

Quadrant I: $x \geq 0, y \geq 0$

Equation (19) becomes

$$h(x, y) + x + y \leq m + n. \quad (20)$$

Every grid location above $y = 0$, to the right of $x = 0$, and which satisfies equation (20) will be expanded. The boundary of the area of nodes expanded can be determined by finding roots of

$$h(x, y) + x + y = m + n. \quad (21)$$

There are six equations which define the roots to equation (21): $y=n$, $x=m-1$, $y=n-2+2/(x+m-1)$, $y=n-1$, $x=m$ and $y=n-1+2/(x+m-2)$, see Figure 30. The number of nodes within the outermost of these boundaries is

$$Z_I = nm - 1 \quad (22)$$

Quadrant II: $x \geq 0, y < 0$

Equation (19) becomes

$$h(x, y) + x - y \leq m + n. \quad (23)$$

Changing the inequality to an equality and finding the roots reveals that nodes between $y=0$ and

$$y = 1 - n - \frac{2}{2 - m - 2n + x} + \frac{4n}{2 - m - 2n + x} - \frac{2n^2}{2 - m - 2n + x} \quad (24)$$

will be expanded. Integrating equation (24) yields the number of nodes:

$$Z_{II} = - \int_{x=0}^m \left(1 - n - \frac{2}{2 - m - 2n + x} + \frac{4n}{2 - m - 2n + x} - \frac{2n^2}{2 - m - 2n + x} \right) dx$$

$$Z_{II} = (n - 1)m + 2(n - 1)^2 \log(2n - 2) - 2(n - 1)^2 \log(m + 2n - 2) \quad (25)$$

Quadrant III: $x < 0, y \geq 0$

This case is symmetric to quadrant II with m and n interchanged, so

$$Z_{III} = (m - 1)n + 2(m - 1)^2 \log(2m - 2) - 2(m - 1)^2 \log(n + 2m - 2) \quad (26)$$

Quadrant IV: $x < 0, y < 0$

Equation (19) becomes

$$h(x, y) - x - y \leq m + n. \quad (27)$$

The boundary is defined by two equations which intersect at

$$x_0 = (3 - 2m - 4n + \sqrt{9 - 16m + 8m^2 - 16n + 16mn + 8n^2}) / 2.$$

The equations which define the boundary are

$$y_A = \frac{n - nm + 2x - 2mx - nx}{x + 2n + m - 1}, \quad (28)$$

and

$$y_B = \frac{m - mn + x - 2mx - nx}{x + 2n + m - 2}. \quad (29)$$

Integrating equations (28) and (29) yields the total number of nodes expanded in quadrant IV:

$$Z_{IV} = \int_{x=(mn-n)/(2-n-2m)}^{x_0} y_A dx + \int_{x=x_0}^0 y_B dx.$$

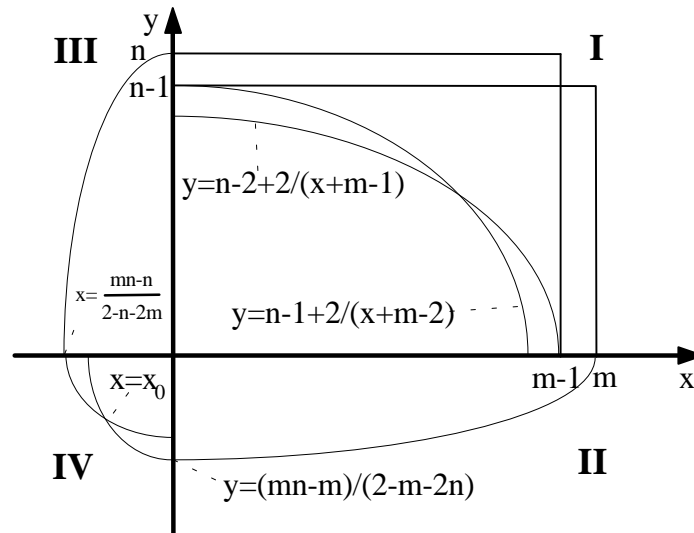


Figure 30. Nodes within these boundaries will be expanded by perimeter search

The total number of nodes expanded for PS^* with a constant depth perimeter of depth 1, $Z = Z_I + Z_{II} + Z_{III} + Z_{IV}$, is plotted in Figure 31 for the special case where $m = n$. The figure also shows the theoretical number of nodes expanded by A^* (Pearl, 1984). The figure shows that PS^* offers only a small improvement over A^* . If the number of nodes expanded when generating the perimeter is taken into account, then PS^* actually expands more nodes than A^* for small problems. For larger problems, PS^* potentially offers some improvement over A^* .

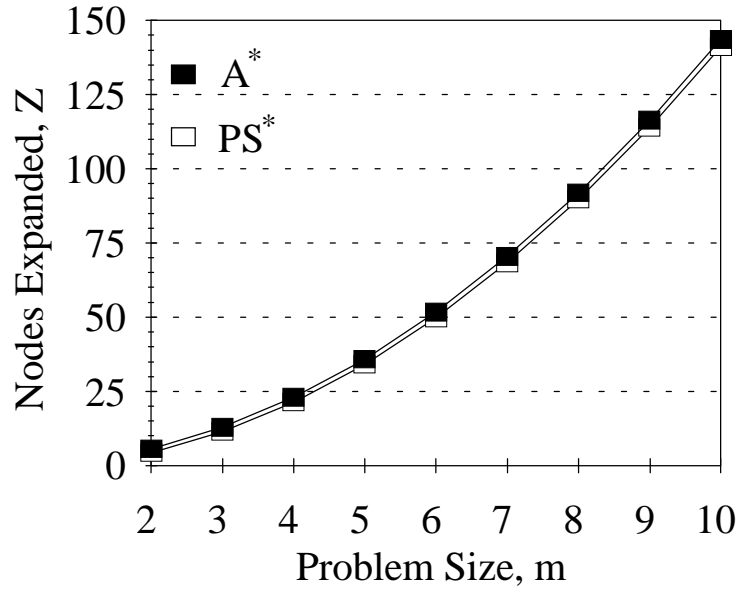


Figure 31. Theoretical number of nodes expanded for grid search problem

Although PS^* decreases the search efficiency of small grid search problems, it potentially improves search efficiency for larger problems. The usefulness of perimeter search on the grid search problem will be both a function of the problem size and of the amount of time spent evaluating the heuristic. Let $Z_{PS} = Z$ be the number of nodes expanded by PS^* (with $m = n$). Let $Z_A = 1.436m^2$ be the number of nodes expanded by A^* (Pearl, 1984). Since there are four perimeter nodes, the time complexity of perimeter search can be estimated as

$$R_{PS} = t(4fZ_{PS} + (1-f)Z_{PS} + 4), \quad (30)$$

where f is the fractional amount of time spent evaluating the heuristic and t is the amount of time per node expansion for A^* with one goal. The estimated time complexity of A^* is

$$R_A = tZ_A. \quad (31)$$

The usefulness of perimeter search is a function of m and f . The functional relationship between m and f can be determined by setting $R_A = R_{PS}$ and solving for f , see Figure 32.

The figure shows the upper bound on f for which perimeter search is at least as fast as A^* . The figure shows that if f is greater than about 0.0004, then perimeter search will not be useful regardless of the problem size. It is interesting to note that f decreases with problem size and that it peaks around $m = 40$. In summary, an impossibly small value of f is necessary to make perimeter search as fast as A^* for the grid search problem domain. Therefore, PS^* should not be used for the grid search problem.

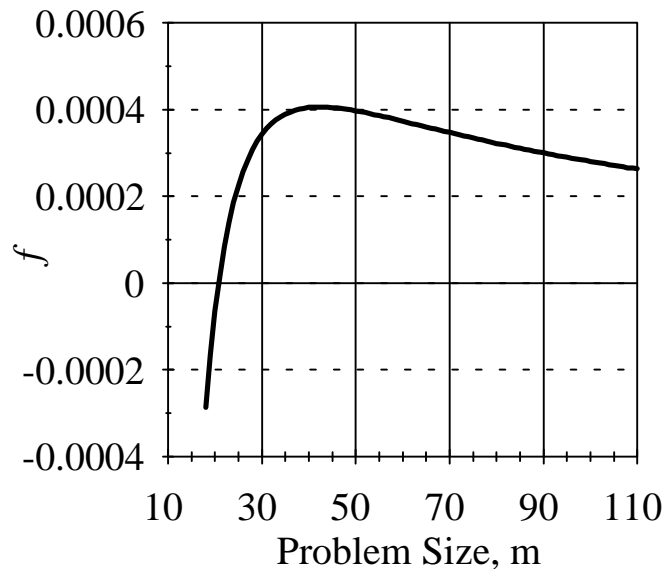


Figure 32. Maximum value of f versus grid search problem size

6.10 Summary of Perimeter Search

A new search algorithm has been presented called perimeter search. Perimeter search can be viewed as a bidirectional search algorithm. Like other bidirectional search algorithms, perimeter search attempts to reduce search complexity by splitting the search space into two separate parts. Unlike other bidirectional search algorithms, perimeter search does not attempt to perform two searches simultaneously. This allows perimeter

search to avoid the overhead involved in retargeting or pruning as in the d-node retargeting and BS* algorithms.

By assuming the search space is a tree, analytical equations were presented which provide a method to predict when perimeter search will be useful for a particular problem domain. These equations can also be used to calculate the perimeter size which yields optimum performance.

Unlike the 15-puzzle and Think-a-Dot search spaces, the grid search space cannot be modeled using a tree. A separate set of analytical equations were derived for the number of node expansions when a perimeter of depth 1 is being used on the grid search problem. It was shown that perimeter search will be an improvement over A* only if the fraction of time spent evaluating the heuristic can be kept below 0.04% of the overall node expansion time. This fraction of time was found to decrease even further as the problem size increased. These facts indicate that perimeter search is not useful for the grid search problem domain. Since the search space of a typical route planning problem is similar to a four connected grid, perimeter search will not be useful for most route planning problems.

Tests results performed on the 15-puzzle and Think-A-Dot problems show that perimeter search can decrease the search complexity quite substantially. For instance, the 15-puzzle results show that IDPS* is about 3.8 times faster than IDA*. The Think-A-Dot tests show that PS* is 1.7 times faster than A*. Also note that PS* expands only one quarter of the nodes expanded by A* for the Think-A-Dot problem resulting in a corresponding reduction in the amount of memory required.

The IDPS* algorithm can be viewed as a memory bounded search algorithm, with the memory bound equal to the perimeter size. Unlike memory bounded algorithms like MA* and MREC, IDPS* keeps a bottom-up memory rather than a top-down memory. A

top-down memory prevents the same node from being reexpanded while memory is available. A bottom-up memory does not prevent nodes from being reexpanded, but rather eliminates the last and most expensive iterations of an IDA* search.

A near-optimal version of the perimeter search algorithm exists and may be useful. The near-optimal perimeter search algorithm finds paths whose costs are guaranteed to be at most ∂ from optimal, where ∂ is the perimeter depth. Tests results for the 15-puzzle indicate that near-optimal perimeter search is much faster than IDA* while still finding high quality paths.

A perimeter is defined as any set of nodes such that at least one of the nodes must appear on an optimal path. This is basically the same definition as an *island set*, the difference being that a perimeter is generated by the search algorithm and an island set is given as part of the problem description. Two types of perimeters were discussed, constant depth and constant evaluation. Although one would intuitively expect constant evaluation perimeters to improve search efficiency, test results indicate this is not the case.

A parallel version of the perimeter search algorithm was discussed. Test results indicate a 30% efficiency rating for 24 processors and a 50% efficiency rating for 10 processors. Analytical results show that the efficiency of parallel perimeter search is a function of the heuristic pruning power and the size of the perimeter. The weaker the heuristic, the better the efficiency of parallel perimeter search. The efficiency decreases as the number of processors increases.

7. CONCLUSIONS

The research presented in this dissertation has concentrated on ways of improving the efficiency of heuristic search algorithms. Chapter 3 presented efficient data structures which can be used for the agenda in A^* , Algorithm I, and MA^* . These data structures allows node expansions to be performed in $\Theta(\log |\text{OPEN}|)$ time, where $|\text{OPEN}|$ is the number of nodes in the agenda. This is a substantial improvement over other suggested data structures whose node expansion time complexity is $\Theta(|\text{OPEN}|)$.

Chapter 4 presented various island search algorithms. Test results indicate that island search improves search performance for two problem types, the grid search problem with obstacles and the route planning problem. Unfortunately, the island search algorithms require an island set as part of their input. For many problem instances, obtaining an island set is not a trivial task. This is somewhat alleviated by the ϵ -admissible version of the island search algorithm which relaxes the requirement that at least one node must be on an optimal path. Given the positive results obtained when using the multiple level island search algorithms for the route planning problem, further research should be conducted to study the feasibility of a practical implementation of an in-vehicle route planner based on island search.

Chapter 5 presented a study of the effectiveness of eliminating cycles when using depth-first type searches. Problem domains were shown for which full cycle checking improved search efficiency. One problem domain was shown for which full cycle checking actually lowered search efficiency. A methodology was given for choosing whether or not to use full cycle checking on a given problem instance.

Finally, chapter 6 presented a new bidirectional search algorithm called perimeter search. Perimeter search was shown to be more efficient than other unidirectional and bidirectional search algorithms. A detailed methodology was given for deciding when to use perimeter search based on the properties of the given problem domain. Both optimal and near-optimal versions of the perimeter search algorithm were presented. The near-optimal perimeter search algorithm was proved to be ϵ -admissible. Two types of perimeters were presented and it was shown that there was relatively no difference in their performance. A parallel perimeter search algorithm was also discussed. Results of a uni-processor simulation of a distributed computing environment indicate that perimeter search has great potential as a parallel/distributed search algorithm. The distributed perimeter search algorithm was implemented using a client/server methodology and the resulting algorithm achieved 50% efficiency when running on 10 Unix workstations. Further research should be conducted to study the possibility of increasing the efficiency of the parallel perimeter search algorithm through a better implementation and by assigning more than one perimeter node per processor.

REFERENCES

- Bayer, R. Symmetric binary B-trees: Data Structures and Maintenance Algorithm. Acta Informatica 1: 290-306, 1972.
- Ben-Akiva, M., Bernstein, D., Hotz, A., Koutsopoulos, H., Sussman, J.: The Case for Smart Highways. Technology Review, July 1992: 39-47.
- Catling, I. and de Beek, F.O.: SOCRATES: System of Cellular RAdio for Traffic Efficiency and Safety. Vehicle Navigation & Information Systems Conference Proceedings, 1991: 147-150.
- Chakrabarti, P.P., Ghose, S., and Desarkar, S.C.: Heuristic Search Through Islands. Artificial Intelligence 29:339-348, 1986.
- Chakrabarti, P.P., Ghose, S., Acharaya, A., and De Sarkar, S.C.: Heuristic Search in Restricted Memory. Artificial Intelligence 41:197-221, 1989.
- Cormen, T.H., Leiserson, C.E., and Rivest, R.L.: Introduction to Algorithms. Cambridge, The MIT Press, 1990.
- De Champeaux, D. and Sint, L.: An Improved Bidirectional Heuristic Search Algorithm. Journal of the Association for Computing Machinery 24:177-191, 1977.
- Dillenburg, J.F.: Theoretical and Experimental Results of Heuristic Island Search Studies. Master's Thesis, University of Illinois at Chicago, 1991.
- Dowdy, S. and Wearden, S.: Statistics for Research. New York, NY, John Wiley & Sons, 1983.
- Hart, P., Duda, R. and Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on SSC 4:100-107, 1968.

- Hicks, J.E.: Static Network Equilibrium Models and Analyses for the Design of Dynamic Route Guidance Systems. Ph.D. Dissertation, Dept. of Civil Engineering, University of Illinois at Chicago, 1992.
- Jurgen, R.K.: Smart Cars and Highways Go Global. IEEE Spectrum, May 1991, 26-36.
- Kirson, A., Smith, B., Boyce, D., Nelson, P., Hicks, J., Sen, A., Schofer, J., Koppelman, K., Bhat, C.: The Evolution of ADVANCE. Third International Conference on Vehicle Navigation and Information Systems, September 1992: 516-522.
- König, D.: Theorie der Endlichen und Unendlichen Graphen. Akademische Verlagsgesellschaft, Leipzig, 1936.
- Korf, R.E.: Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. Artificial Intelligence 27:97-109, 1985.
- Korf, R.E.: Search: A Survey of Recent Results. Exploring Artificial Intelligence, ed. H.E. Shrobe, pp. 197-237. San Mateo, CA, Morgan Kaufmann, 1986.
- Kumar, V., Ramesh, K., and Nageshwara, V.N.: Parallel Best-First Search of State-Space Graphs: A Summary of Results. Proceedings of the Sixth National Conference of Artificial Intelligence (AAAI-88): 122-127, 1988.
- Kwa, J.B.: BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm. Artificial Intelligence 38: 95-109, 1989.
- Mahanti, A., Ghosh, S., Nau, D.S., Pal, A.K. and Kanal, L.: Performance of IDA* on Tress and Graphs, Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92): 539-544, 1992.
- Morris, R., Scatter Storage Techniques, Communications of the ACM 11: 35-44, 1968.

- Mostow, J. and Prieditis, A.E.: Discovering Admissible Heuristics by Abstracting and Optimizing: A transformational Approach. Proceedings of the International Journal of Artificial Intelligence 1 :701-707, 1989.
- Nelson, Peter C.: Parallel Heuristic Search using Islands. Proceedings Hypercubes, Concurrent Computers and Applications 4, 1989.
- Nilsson, Nils J.: Principles of Artificial Intelligence. Los Altos, Morgan and Kaufmann, 1980.
- Pearl, J.: Heuristics. Reading, Addison-Wesley, 1984.
- Pohl, I.: Bi-directional Search, in: B. Meltzer and D. Michie, Eds., Machine Intelligence 6: Edinburgh University Press, Edinburgh, 127-140, 1971.
- Politowski, G. and Pohl, I.: D-Node Retargeting in Bidirectional Heuristic Search. Proceedings of the Second National Conference of Artificial Intelligence (AAAI-84), Austin, TX, 274-277, 1984.
- Rillings, J.H. and Lewis, J.W.: TravTek. Vehicle Navigation & Information Systems Conference Proceedings 1991: 729-738.
- Sarkar, U.K., Chakrabarti, P.P., Ghose, S., and De Sarkar, S.C.: Multiple Stack Branch and Bound. Information Processing Letters 37: 43-48, 1991.
- Sen, A. and Bagchi, A.: Fast Recursive Formulations for Best-First Search that Allow Controlled use of Memory, Proceedings of the 11th International Joint Conference of Artificial Intelligence (IJCAI-89): 297-302, 1989.
- Taylor, L.A. and Korf, R.E.: Pruning Duplicate Nodes in Depth-First Search. UCLA Computer Science Technical Report, University of California at Los Angeles, 1992.

Williams, J.W.J.: Algorithm 232: Heapsort, Communications of the ACM 7 347-348, 1964.

VITA

NAME: John F. Dillenburg

EDUCATION: B.S.E., Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, Illinois, 1989

M.S., Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, Illinois, 1991

EXPERIENCE: Research Assistant, Intelligent Vehicle Highway Systems Laboratory, Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, Illinois, 1989 and 1991.

Teaching Assistant, Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, Illinois, 1990.

Systems Programmer, Engineering Physics Division, Argonne National Laboratory, Argonne, Illinois, 1986-1990.

HONORS: EECS Senior Scholar Award, University of Illinois, Chicago, Illinois, 1989.

Second Place, EECS Senior Design Exposition, University of Illinois, Chicago, Illinois, 1989.

MEMBERSHIP: Association for Computing Machinery

Golden Key National Honor Society

Tau Beta Pi

PUBLICATIONS:

- Nelson, P.C. and Dillenburg, J.F. "Multiple Level Island Search," *Proceedings of the Fifth Rocky Mountain Conference on Artificial Intelligence* 1990: 231-238.
- Nelson, P.C., Dillenburg, J.F., and Dubinsky, Lana "HSAS: A Heuristic Development Tool," *Tools for Artificial Intelligence* 1990: 478-484.
- Dillenburg, J.F. and Nelson, P.C. "Improving the Efficiency of Depth-First Search through Cycle Checking," *Information Processing Letters* 45: 5-10, 1993.
- Nelson, P.C., Dillenburg, J.F., and Lain, C. "Vehicle-Based Route Planning in Advanced Traveler Information Systems," *1993 IEEE Intelligent Vehicle Symposium*, Tokyo, Japan, July 1993: 152-156.
- Dillenburg, J.F. and Nelson, P.C. "Improving Search Efficiency using Possible Subgoals," submitted to *Applied Intelligence*.
- Dillenburg, J.F. and Nelson, P.C. "Perimeter Search," submitted to *Artificial Intelligence*.
- Dillenburg, J.F. and Nelson, P.C. "Efficient Data Structures for Heuristic Search," submitted to *Information Processing Letters*.