An Analysis of Iterative-Deepening-A*

Brian Glen Patrick
School of Computer Science
McGill University
Montreal, Canada

November 1991

A Thesis
Submitted to the Faculty of Graduate Studies and Research
In partial fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

Iterative-deepening-A* (IDA*) is an admissible heuristic search algorithm which is optimal with respect to space complexity and the cost of solution found over the class of admissible best-first tree search algorithms. However, the optimality of IDA* with respect to time complexity is subject to a number of conditions. It is the focus of this dissertation to identify the conditions that give rise to the worst case performance of IDA* and to delineate a time complexity spectrum between its optimal and worst case performance. In addition, the expected case performance of IDA* is derived with respect to a probabilistic model of computation that assumes the differential edge costs are independently and identically distributed as random non-negative integers. Under this assumption, IDA* exhibits asymptotic optimal performance for any integer probability distribution that satisfies a couple of weak conditions. Finally, to redress the worst case phenomenon of expanding only a few additional nodes over several iterations, a new admissible search algorithm, called Binary IDA* (BIDA*), is presented and compared against the performance of IDA* on the Euclidean traveling salesperson problem. It is shown in a small empirical study that BIDA* is a significant improvement over IDA* as both the tour size and the precision of the edge costs increase.

Résumé

Iterative-deepening-A* (IDA*) est un algorithme heuristique admissible de recherche qui, parmi la classe des algorithmes admissibles de recherche d'arborescence meilleur d'abord, est optimal en ce qui a trait à la complexité en taille et au coût de calcul de la solution. Cependant, l'optimalité de la complexité en temps de IDA* est sujette à de nombreuses contraintes. Le but de la présente thèse est de préciser sous quelles conditions le pire des cas est atteint et de spécifier quelle est la plage de variation de la complexité en temps de l'algorithme entre le meilleur et le pire des cas. De plus, la complexité en moyenne de IDA* est obtenue dans le cas où est faite l'hypothèse que les coûts différentiels sont donnés par des entiers positifs indépendants aléatoirement et identiquement distribués. Sous cette hypothèse, IDA* est asymptotiquement optimal, et ce pour toute distribution de probabilité soumises en plus à quelques contraintes supplémentaires relativement faibles. Finalement, dans le but de corriger dans le pire des cas le problème de l'expansion de seulement un petit nombre de sommets sur plusieurs itérations, un nouvel algorithme de recherche, baptisé Binary IDA* (BIDA*), est introduit. L'efficacité des deux algorithmes (IDA* et BIDA*) est comparée en les testant successivement sur le problème du voyageur de commerce Euclidien. En augmentant le nombre de villes et la précision de calcul des distances, on trouve empiriquement que BIDA* est un algorithme plus efficace que IDA*.

translated by Dr. Jocelyn Desbiens

Statement of Originality

Although all work herein that is not otherwise credited represents an original and distinct contribution to the study of heuristic search, the following individuals nonetheless deserve special recognition:

- The worst case analysis of Chapter 5 is based in large part on the paper entitled "An upper bound on the time complexity of iterative-deepening-A*" [35] which was co-authored with the aid of Mohammed Almulla and Monroe M. Newborn.

- The analytical development of Chapter 6 was suggested by Dr. Luc Devroye of McGill University who demonstrated the use of generating functions in the derivation of the asymptotic expected case analysis of IDA*.

Notwithstanding the contributions of those individuals mentioned above, any remaining oversights and errors are the sole responsibility of the author.

Acknowledgements

The pursuit of research requires the continued support of many people and organizations over a number of years. I gratefully acknowledge this support and extend my deep appreciation to:

- The Natural Sciences and Engineering Research Council of Canada which granted me the financial means to undertake graduate studies,
- The Collège Militaire Royal de St. Jean which generously provided the computing resources for the preparation of my dissertation, and
- My supervisor, Dr. Monroe M. Newborn, who is owed a sincere thank-you for his guidance and concern.

Yet, any success that I enjoy has always been a testament to a wonderful family who weathered the storms and the doldrums of research with patience and understanding. It is to my father and mother and my two brothers, Kevin and Sean, that I dedicate this work with all my love.

Contents

List of Figures

List of Tables

Chapter 1

Heuristic Search

1.1 Introduction

A wide variety of difficult and often intractable problems in artificial intelligence, operations research and combinatorics are represented in terms of the solution space model and involve a search of that space to find an optimal or near-optimal solution [13, 32]. Consequently, considerable research has focussed and continues to focus on questions of how to incorporate and manipulate *knowledge* in the searching process [33, 38]

Without the availability of any knowledge, a search reduces to a systematic enumeration and comparison of all possible solutions that comprise the *solution space* of a problem. For most combinatorial applications of interest, the exploration of the entire solution space is infeasible in terms of computational time. On the other hand and with the availability of perfect knowledge, a search is propelled directly toward an optimal solution. Unfortunately, a search in such cases often suggests a polynomial time solution to problems which are provably intractable. Hence, the development of so-called "intelligent" search strategies

rests on a mutual dependency. The presence of incomplete knowledge necessitates the implementation of a search strategy to utilize this knowledge in an efficient and productive way. And conversely, the size of most solution spaces necessitates the use of knowledge, however incomplete, to limit the search toward those areas of the solution space that offer the most promise of success. The combination of knowledge and search is called *heuristic search.*

1.2 Optimal Path Problem

The solution space of a problem is abstractly defined as a weighted, directed graph $G = (V, E, C)$ where:

1. V is a set of nodes $\{n_1, n_2, n_3, \ldots\}$,
2. E is set of directed edges $\{(n_i, n_j) \in V \times V\}$, and
3. C is a cost function, defined on E, that assigns a real positive cost to each edge.

A single node is distinguished as the start node s and represents the initial configuration of the problem. A subset of nodes is distinguished as the goal nodes and represents those solutions that satisfy the stated objectives of the problem definition. Each other node n in V represents a distinct partial solution along any path rooted at the start node s. A directed edge (n_i, n_j) represents the transformation from one partial solution n_i to another n_j. The cost of this transformation from n_i to n_j is represented by the cost of the directed edge (n_i, n_j) and is therefore equal to $C(n_i, n_j)$.

Any path from the start node s to a goal node is defined as a *solution path.* Hence, many problems are formulated in terms of finding not only a

solution path but also a path that satisfies some criteria of optimality [41]. These problems are often stated in terms of the *optimal path problem.*

Definition 1.1 *Given a solution space graph G, the optimal path problem is to find the minimum cost path from a start node to a goal node*

In general, the cost of a path P from the start node s to any other node n is an arbitrary function of the costs assigned to the nodes and edges along that path.

1.3 Best-First Search

The representation of the solution space $G = (V, E, C)$ is often too large to be stated explicitly. Therefore the solution space graph is constructed algorithmically given an implicit definition of G. Thus,

$$G = (s, \Upsilon)$$

where s is the start node and Υ is an *expansion operator* [33]. The application of Υ to a selected node n, called the *expansion of* n, generates all successors of n and the costs of the associated edges from n to each of its successors. It is assumed that the solution space graph is locally-finite whereby each node has a finite number of successors. The systematic application of Υ to the start node s, its successors and so on explicates the solution space graph $G = (s, \Upsilon)$ and defines in effect the process of search.

1.3.1 Description

One of the most common search algorithms for the optimal path problem is the *informed, best-first strategy* [33, 38, 51]. Each best-first search algorithm

employs an *evaluation function* f [26] that assigns a non-negative cost to each node n. This cost, called the *f-value*, is an estimate of the latent merit or promise of a partial solution. Therefore, the cost assigned to node n is generally a composite of information that is gathered along the path from the start node s to n and that is inferred about the remaining cost from n to a goal node. The estimate of the remaining cost is gleaned from the underlying problem domain and is formulated in terms of a *heuristic function* h. The heuristic function h is defined on each node of G and is incorporated into the definition of the solution space graph. Thus,

$$G = (s, \Upsilon, h).$$

At the beginning of a best-first search, an OPEN list of expandable nodes contains only the start node s and its associated cost $f(s)$. On each step of the search, the node with the minimum f-value is selected from OPEN and *expanded* by the application of Υ. Once a node has been expanded, it is removed from OPEN and added to a CLOSED list. Each successor n that is generated by the expansion operator Υ is assigned an f-value. If node n has not been generated before, it is added to OPEN. If node n has been generated before and its new f-value is greater than its current f-value on OPEN or CLOSED then node n is discarded; otherwise, node n is added to OPEN and the previous instance of n is removed from either OPEN or CLOSED. Using these f-values as a guide, best-first search continues the process of selection and expansion until either a goal node is selected for expansion or no node is available for expansion, in which case a failure is proclaimed[1]. One best-first search algorithm called BF* is outlined in Figure 1.1 [8].

[1]It is assumed throughout the dissertation that at least one solution path does exist from the start node s to a goal node

```
Comments:

     OPEN   is a list of expandable nodes.
     CLOSED is a list of expanded nodes.
     f(n)   is the f-value of the most recent generation of n.
     F(n)   is the minimum f-value of node n on OPEN or CLOSED.

Step 1: Put the start node s on OPEN and calculate f(s).
        Set F(s) equal to f(s).
Step 2: If OPEN is empty then
           Exit with failure (No solution exists).
Step 3: Remove from OPEN a node p whose f-value is minimum.
        (Break ties arbitrarily but always in favour of a goal
         node.)
        Place p on CLOSED.
Step 4: If p is a goal node then
           Exit successfully with solution obtained by tracing
           back the path along the pointers from p to s.
        (Pointers are assigned in Steps 5 and 6).
Step 5: Expand node p, generating all successors of p.
        Direct pointers back from each successor n to p.
Step 6: For each successor n of p
        a) Calculate f(n).
        b) If n is neither on OPEN nor on CLOSED then
              Add n to OPEN
              Set F(n) equal to f(n).
        c) Else
              {n already resides on OPEN or CLOSED}
              If f(n) < F(n) then
                 Add n to OPEN
                 Set F(n) equal to f(n)
                 Remove previous n from OPEN or CLOSED
              Else
                 Remove current n.
Step 7: Go to Step 2.
```

Figure 1.1: The BF* Algorithm

If the search is guaranteed to terminate with an optimal solution path then the search is called *admissible* [33]. Admissibility is easily ensured if the evaluation function f assigns to each node n a lower bound on the cost of the optimal solution path from the start node s, through n, to a goal node [18].

1.3.2 Disadvantages

Each best-first search algorithm maintains all feasible paths that are rooted at the start node s. For most combinatorial problems, the prioritized list of expandable nodes and the number of feasible paths grows exponentially. Available memory is therefore rapidly consumed. In order to limit memory requirements and permit the solution of larger and possibly more complex problems, alternate search strategies must be considered.

One alternative is called the *partially-informed, best-first strategy* [38]. Each path is extended from the most recently-generated rather than the entire set of expandable nodes. Such a strategy is based on a backtracking scheme and consumes only a linear amount of memory with respect to the length of the current path. The simplicity of this scheme and its economical use of space suggests the development of hybrid search strategies that combine the storage economy of backtracking with the efficient utilization of heuristic knowledge [22]. One such algorithm and the analytical focal point of this dissertation is *iterative-deepening-A**.

1.4 Iterative-Deepening-A*

The iterative-deepening-A* algorithm (IDA*) is a hybrid of the best-first search algorithm called A* and the iterative backtracking algorithm called

depth-first iterative-deepening [23]. The A* algorithm [18] is by far the most studied version of best-first search. It is one of the few heuristic search techniques that is supported by a substantial theoretical body of knowledge A fuller discussion on the optimality and performance of the A* algorithm is deferred until Chapter 2.

Depth-first iterative-deepening (DFID), on the other hand, is a brute-force method of search. Over the class of brute-force search algorithms, DFID is optimal along the dimensions of path length, node expansions and memory space among exponential tree searches [23, Theorem 4.2]. The DFID algorithm achieves each objective by performing successive depth-first searches that probe to deeper and deeper levels of the solution space G. Each probe or *iteration* that does not return a goal node is repeated to a deeper level of the solution space. If the explored solution space grows exponentially from one iteration to the next, it has been shown that the time complexity of DFID is asymptotically optimal over the class of brute-force search algorithms [23]. In essence, DFID is a cross between depth-first and breadth-first search: It maintains the space optimality of depth-first search, but bounds each search such that an optimal path length is found.

Depth-first iterative-deepening has been successfully merged with other heuristic search techniques. DFID was originally applied to alpha-beta search [48] where it became and remains the method of search found at the heart of the best chess-playing programs [25]. Its space economy permits search to greater depths of the alpha-beta tree and hence allows a more assured estimate of the best possible move at each stage of play. DFID has also been applied to the problem of automated theorem proving [49] When proving a theorem, all possible proofs in one step are attempted first, then two steps and so on until a satisfactory resolution is reached [31].

The optimality of the IDA* algorithm with respect to space complexity and the cost of solution found is respectively ensured by the nature of depth-first search and by the admissibility of the evaluation function f [23, 24]. However, the time optimality of IDA* is contingent upon a number of interrelated factors which depend on both the structure of the solution space and the behaviour of the evaluation function f. It is the aim of this dissertation to provide a clearer understanding of how the performance of IDA* is influenced by these factors and under what conditions the three parameters of optimality are realized.

1.5 Thesis Outline

The remainder of the dissertation is partitioned as follows:

Chapter 2: A full description on the performance measurements of the A* algorithm is presented with respect to the class of admissible best-first search algorithms.

Chapter 3: A full description of the IDA* algorithm is presented and a common measure of time complexity is established between the A* and IDA* algorithms.

Chapter 4: A strict upper bound on the time complexity of IDA* is established. As well, the conditions that give rise to the worst case performance of IDA* are identified.

Chapter 5: A time complexity spectrum of IDA* is delineated by its asymptotic optimal and worst case performance.

Chapter 6: An expected case analysis of IDA* is derived with respect to a probabilistic model of computation. The asymptotic expected case

analysis assumes that the values of the differential edge costs are independently and identically distributed as random non-negative integers.

Chapter 7: A variation of IDA*, called Binary IDA*, is introduced and compared against the performance of IDA* on the Euclidean traveling salesperson problem.

Chapter 8: Final remarks and future avenues of research are offered.

Chapter 2

The A* Algorithm

2.1 Description of A*

The A* algorithm is a heuristic, best-first search algorithm which was developed specifically for additive cost measures [18]. These measures define the cost of a path as the sum of its edge costs. Therefore, given a path $P = n_1, n_2, n_3, \ldots, n_p$ between nodes n_1 and n_p, the cost of path P is equal to

$$\sum_{i=1}^{p-1} C(n_i, n_{i+1})$$

where each $C(n_i, n_j)$ is greater than some positive constant δ.

2.1.1 Evaluation Function

The A* algorithm employs an evaluation function f that assigns a cost to each node n that is the sum of two components, $g(n)$ and $h(n)$, where:

1. $g(n)$ is the cost of the current path from the start node s to n, and
2. $h(n)$ is a non-negative heuristic estimate of $h^*(n)$ where $h^*(n)$ is the cost of the optimal path from n to a goal node. If n is a goal node then $h(n)$

is equal to 0.

For completeness, $g^*(n)$ and $f^*(n)$ are defined as follows:

1. $g^*(n)$ is the cost of the optimal path from s to n, and

2. $f^*(n)$ is the cost of the optimal solution path from s to a goal node that is constrained to pass through n, that is, $f^*(n) = g^*(n) + h^*(n)$. The cost of the optimal solution path from s to any goal node is equal to $f^*(s)$ and is denoted C^*.

The evaluation function of A* therefore has the form $f = g + h$ which is a compromise of two earlier search techniques:

1. The uniform cost algorithm [10], and

2. The graph traverser algorithm [11].

Uniform cost search employs an evaluation function $f = g$ and guides the search toward those areas of the solution space based strictly on perfect knowledge. The graph traverser algorithm, on the other hand, employs an evaluation function $f = h$ and hence, guides the search toward those areas of solution space based strictly on heuristic knowledge. The A* algorithm places an equal emphasis on perfect and heuristic knowledge, although this ratio can be generalized to

$$f(n) = (1 - \omega)g(n) + \omega h(n)$$

where $0 \leq \omega \leq 1$ [40, 43]. The values of $\omega = 0$, 0.5 and 1 respectively define the evaluation functions for the uniform cost, the A* and the graph traverser algorithms.

2.1.2 Admissibility

Theorem 2.1 ([18]) *Given a solution space graph $G = (s, \Upsilon, h|h \leq h^*)$, A* is admissible.*

Proof Outline. Since $h \leq h^*$, the f-value of each node along an optimal solution path is less than or equal to C^*. At any time before A* terminates, there exists on OPEN a node n that is on an optimal solution path with $f(n) \leq C^*$. Assume that A* terminates at some goal node t without finding an optimal solution path, that is, $f(t) = g(t) > C^*$. But node n with $f(n) < f(t)$ is on OPEN. Thus, A* would have selected node n for expansion rather than node t which contradicts the assumption that A* terminated. □

Clearly, any heuristic function h that is a lower bound on h^* is an *admissible heuristic* and any evaluation function $f = g + h$ that uses an admissible heuristic is an *admissible function*. Unless otherwise stated, it is assumed that both h and f are admissible throughout the dissertation.

Nilsson [34] regards the requirement that $h \leq h^*$ as part of the definition of A*. However, in accordance with Nilsson's original definition in [33] and more recent literature [2, 5, 8, 38], the term A* is assigned to any BF* algorithm that uses the additive combination $f = g + h$, placing no restriction on h. Therefore, A* is identified by how it processes input information rather than by the type of information that it may encounter. Two special cases of the solution space graph G are henceforth distinguished based on a qualification of the heuristic function h:

1. Admissible solution space graph where $G = (s, \Upsilon, h|h \leq h^*)$.

2. Uninformed solution space graph where $G = (s, \Upsilon, h|h = 0)$.

Definition 2.1 *Any search algorithm that is applied to an uninformed solution space graph G is called a brute-force search.*

2.1.3 Conditions of Expansion

Given a solution space graph $G = (s, \Upsilon, h | h \leq h^*)$, the sufficient and necessary conditions of expansion are stated for any node n:

Sufficient Condition: There exists a path from the start node s to n along which each node has an f-value that is strictly less than C^* [38, Theorem 5, p. 80]

Necessary Condition: There exists a path from the start node s to n along which each node has an f-value that is less than or equal to C^* [38, Theorem 6, p. 81].

Therefore, a certain subset of nodes that may otherwise satisfy the necessary condition of expansion may or may not be selected for expansion subject to the tie-breaking rule that is employed by A*. To provide a measure of performance that is independent of the choice of tie-breaking rules, the notion of a *surely-expanded* node is introduced

Definition 2.2 *A node n is surely-expanded by the A* algorithm if and only if it satisfies the sufficient condition of expansion.*

Any node n may be reached by more than one path from the start node s to n. Therefore, a node n that is generated more than once is re-expanded for each assigned f-value that is less than the cost of the optimal solution path The possibility that a node n will be re-expanded several times, depending on the succession of f-values that are assigned to it, potentially explodes the overall execution time of the search [28].

Definition 2.3 ([43]) *Given a solution space graph $G = (V, E, C)$, a heuristic function h satisfies the property of monotonicity on G if and only if*

$$h(m) \leq h(n) + C(m, n)$$

for all n generated by m.

Given the property of monotonicity, three important results hold for an A* search:

1. Every monotonic heuristic function h is admissible [38, Theorem 9, p. 83].

2. No node is re-selected for expansion. Once a node n is initially selected for expansion, the optimal path from s to n is found, that is, $g(n) = g^*(n)$ [38, Theorem 10, p. 83]

3. The f-values of the sequence of nodes that are expanded by A* are non-decreasing [38, Theorem 11, p. 84].

Clearly, if h is monotonic, the sequence of values assigned by an evaluation function f along any path rooted at s is also monotonic, that is,

$$f(m) \leq f(n)$$

for all n generated by m. Furthermore, the sufficient and necessary conditions of expansion of node n are simplified for monotonic h [38, Theorem 12, p. 84]:

Sufficient condition: $g^*(n) + h(n) < C^*$.

Necessary condition: $g^*(n) + h(n) \leq C^*$.

Again, a node n is surely-expanded by the A* algorithm if it satisfies the sufficient condition of expansion.

2.2 Measures of Performance

The performance of the A* algorithm is analyzed with respect to the following three measures of complexity:

Node complexity: The number of *distinct* nodes that are expanded during an A* search

Time complexity: The *total* number of nodes that are expanded during an A* search.

Space complexity: The amount of memory space consumed during an A* search.

2.2.1 Node Complexity

The optimality of A*, with respect to node complexity, was established in [5, 29, 33] for the additive rule $g + h$ but to the exclusion of alternate policies of expansion. It was correctly argued in [8, 9, 16] that the optimality of A* must be judged against a wider class of equally-informed best-first search algorithms. To that end, the evaluation function was generalized to assign an f-value to each node n based on any combination of the edge costs and the heuristic values along the path from the start node s to n. The following result is stated in [8].

Theorem 2.2 *Given a solution space graph $G = (s, \Upsilon, h | h \leq h^*)$, the A* algorithm is optimal, in terms of the number of distinct nodes that are surely-expanded, over the class of admissible best-first search algorithms.*

A subtle variation of Theorem 2.2 stems from Result 2 in Section 2.1.3.

Corollary 2.1 *Given a solution space graph $G = (s, \Upsilon, h | h \leq h^*)$ where h is monotonic, the A* algorithm is optimal, in terms of the number of nodes that are surely-expanded, over the class of admissible best-first search algorithms.*

Interestingly, this result confirms the earlier conjectures of [5, 29, 33]. Furthermore, it suggests that any combination of g and h where $h \leq h^*$ will expand every node that is surely-expanded by A*. In other words, the additive combination $g + h$ is an optimal aggregation of g and h for additive cost measures.

2.2.2 Time Complexity

The time complexity of A*, unlike the node complexity, is measured in terms of the *total* number of node expansions. Under the property of monotonicity which states that no node is re-selected for expansion, the time complexity of A* is equal to the number of nodes that are surely-expanded plus a subset of nodes, including the optimal goal node, whose f-values are equal to C^*.

When the monotonic assumption is relaxed, the f-values of the sequence of nodes that are expanded by A* are not necessarily non-decreasing. In the worst case, the A* algorithm performs $O(2^M)$ expansions where M is the number of distinct nodes that are expanded by A* on G [28]. The poor performance of A* for non-monotonic h encouraged the development of several variants of A*, including Martelli's B algorithm [28], Mérõ's B′ algorithm [29], Bagchi and Mahanti's C algorithm [2] and most recently, Mahanti and Ray's D algorithm [27]. Each variant attempts to minimize the effect of non-monotonicity by either:

1. Ignoring the heuristic estimate of an expandable node n when it fails to increase the f-value of n above the current maximal f-value of expanded

nodes (Algorithms B and C), or

2. Modifying (increasing) the heuristic estimate(s) of a parent node or its children upon expansion to reflect information that either the parent or its children had originally overlooked (Algorithms B′ and D).

The worst case time complexity of each new algorithm, except the B′ algorithm, is $O(M^2)$ if the monotonic assumption is waived[1]. However, if the evaluation function satisfies the property of monotonicity, each variant of A* above behaves exactly as the A* algorithm itself.

The time complexity analysis thus far has foregone any analysis of how the accuracy of the heuristic estimate impacts on the number of distinct nodes that are expanded by A*. It was originally shown in [18] that as the heuristic function h better approximates h^*, fewer distinct nodes are expanded by A*. Even when the monotonic assumption is relaxed, if a heuristic h_1 provides a better estimate of h^* than another heuristic h_2 over all nodes of G, that is, $h_2 < h_1 \leq h^*$, then A* is guaranteed to expand at least as many distinct nodes with h_2 as with h_1 [19, 30]. Indeed, if A* employs a perfectly-informed heuristic ($h = h^*$) then the search is propelled directly toward a goal node without expanding any node off an optimal solution path. If the goal node is unique and located at depth d from the start node s then A* performs exactly d expansions. At the other extreme, if A* employs no heuristic information at all ($h = 0$) then the search is exhaustive and, in most cases, exponential in d.

The results presented within this section assume that the solution space $G = (s, \Upsilon, h)$ is modeled as an undirected tree with a uniform branching factor b. The solution space tree is rooted at the start node s and may be

[1]It was later demonstrated that, on the weakness of its tie-breaking rule, Mérõ's B′ algorithm requires in the worst case $O(2^M)$ expansions [27]

either infinite or finite with a minimum depth of d. Each edge is assigned a symmetric unary cost and there exists an unique goal node situated at depth d from the start node s. The earliest work that attempted to quantify the trade-off between the complexity of the A* algorithm and the precision of the heuristic h used to guide the search delineated the spectrum of the precision-complexity trade-off between two points [15, 40, 43, 50]. At one end of the spectrum, if the absolute heuristic errors, denoted $h^* - h$, are bounded by a fixed quantity then the time complexity of A* is linear in d. At the other end of the spectrum, if these errors grow linearly with h^* then the time complexity of A* is exponential in d.

A probabilistic extension of these results was presented in [21]. The errors produced by the heuristic estimates were treated as independent random variables with distributions that could vary over the nodes of the uniform tree model. It was shown that if the relative errors are bounded away from zero with a probability greater than $1/b$ then the expected time complexity of A* is exponential in d. The probabilistic model was generalized in [37] to account for all points along the spectrum, revealing the exponential character of the precision-complexity exchange. If the typical heuristic error grows like $\phi(h^*)$ where ϕ is a sublinear function then the expected time complexity of A* grows approximately like

$$\Theta(d \cdot \exp(c\phi(d)))$$

where c is a positive constant. Therefore, to ensure a polynomial time complexity, the precision of the heuristic function h employed by A* must be logarithmic, for example, $\phi(d) = (\log(d))^k$.

The above analyses were extended by [4, Theorems 3.3 and 3.4] in the presence of multiple goal nodes. Each goal node is located at a distance no

greater than $d + c\phi(d)$ where c is a constant greater than or equal to 0. Given that the probability distribution satisfies some very weak conditions, it was shown that the expected time complexity of A* is exponential in d except when both ϕ is logarithmic and the number of goal nodes is polynomial in d. Therefore, given a heuristic function that exhibits logarithmic precision, the expected time complexity of A* is polynomial if and only if the number of solution paths is also polynomial in d. However, if the number of solution paths is exponential in d then the expected time complexity of A* remains exponential regardless of the accuracy of the heuristic function employed.

2.2.3 Space Complexity and Inadmissibility

The performance of the A* algorithm and its variants has been analyzed at length under the condition of inadmissibility [2, 3, 8, 17, 27, 36, 39, 42]. The analyses concentrate on two measures of performance:

1. The optimality or sub-optimality of the solution found, and
2. The time and space complexity of the resulting search.

The arguments in favour of non-admissible heuristics are motivated by three points:

1. The choice of a heuristic function is no longer constrained to admissible h.
2. Admissible search strategies often spend a disproportionate amount of time discriminating among equally meritorious alternate solutions.
3. Although the inherent strength of the A* algorithm, in particular, lies in its ability to convert heuristic knowledge into appreciable time savings, much of this ability is diluted in the presence of multiple goal nodes [4].

For most combinatorial problems of interest, the above arguments are predicated on a common objective: To reduce the scope of search and thereby to reduce the vast amounts of memory that are consumed by the A* and other best-first search algorithms.

2.3 Concluding Remarks

The A* algorithm is an important member of the class of best-first search algorithms, primarily on the strength of the following three results. Given a solution space graph $G = (s, \Upsilon, h | h \leq h^*)$:

1. A* is admissible.

2. A* is optimal, in terms of the number of distinct nodes that are surely-expanded, over the class of admissible best-first search algorithms.

3. If the heuristic function h is also monotonic then A* is optimal, in terms of the number of nodes that are surely expanded, over the class of admissible best-first search algorithms.

However, because the optimal path problem is a central problem of many artificial intelligence applications, the question of memory consumption remains a critical consideration notwithstanding the above results.

Chapter 3

Iterative-Deepening-A*

3.1 Description of IDA*

Iterative-deepening-A* (IDA*) is a heuristic search algorithm that combines the efficient utilization of heuristic knowledge with an efficient utilization of memory space [23]. In order to meet these objectives, the IDA* algorithm:

1. Assigns a cost to each node that is determined by the evaluation function $f = g + h$ [1], and

2. Performs successive depth-first searches that are bounded by increasing values of f.

The cost bound of the initial iteration of IDA* is denoted C_1 and is set to the cost of the start node s, that is, $C_1 = f(s)$. At the beginning of each iteration, a STACK of expandable nodes contains only the start node s and its associated cost $f(s)$. On each step of the search, the most recently-generated node (i.e, the top node of STACK) is selected and expanded by the partial

[1] Therefore, as with A*, an optimal discriminant for additive cost measures is employed.

application of Υ. The partial expansion of a node generates a single successor at a time. Once all successors of a node have been generated, the node is considered *fully-expanded*; otherwise, it is considered only *partially-expanded* [2]. If the selected node is fully-expanded then it is removed from the STACK and discarded. Each successor n that is generated by the expansion operator Υ is assigned an f-value equal to $g(n) + h(n)$. If the f-value of node n is less than or equal to the cost bound of the iteration during which it is selected then n is added to the top of STACK. The depth-first search continues the process of selection and expansion until either one of two conditions is met:

1. A goal node is selected for expansion, or
2. The f-values of all expandable nodes is greater than the cost bound of iteration i, denoted C_i.

On each iteration, except the final iteration, IDA* performs an exhaustive search of all paths along which each node has an f-value that is less than or equal to the cost bound of the iteration. Once a goal node is selected for expansion, the IDA* algorithm terminates; otherwise, a depth-first search of the solution space G is repeated with a greater cost bound C_{i+1}. The new cost bound is set to the *minimum* f-value among all nodes that were generated on iteration i and that exceeded the cost bound C_i. The IDA* algorithm is outlined in Figure 3.1.

An example of an IDA* search is presented in Figure 3.2. Each node A through M is labeled with its f-value. Nodes A and K are designated as the start and goal nodes respectively. Only those generated nodes whose f-values

[2]Partial expansion ensures that for any (in)finite branching factor the linear memory constraints of depth-first search are satisfied and that each currently explored path is maintained in STACK

```
Comments:

     C    is the cost bound of the current iteration.
     M    is the cost bound of the subsequent iteration,
          that is, the minimum f-value among generated nodes
          that exceeds C.
     f(n) is the f-value of the most recent generation of n.
     MAX  is an arbitrarily large number.

Step 0: Set M equal to f(s), the cost of the start node s.
Step 1: Push s onto a STACK of expandable nodes.
        Set C equal to M.  Reset M equal to MAX.
Step 2: If STACK is empty then
           If M remains equal to MAX then
              Exit with failure; no solution exists
           Else
              Go to Step 1 and proceed with the next iteration.
Step 3: Assign the top node of STACK to p.
Step 4: If p is a goal node then
           Exit successfully;
           Solution maintained by the STACK of nodes.
Step 5: Generate the next successor n of p.
Step 6: If node p has no further successors then
           Pop p from STACK
        Else
           a) Calculate f(n).
           b) If f(n) <= C then
                 Push n onto STACK
                 Assign the newly-computed f(n) to n.
           c) If C < f(n) < M then
                 Lower M to f(n).
Step 7: Go to Step 2.
```

Figure 3.1: The IDA* Algorithm

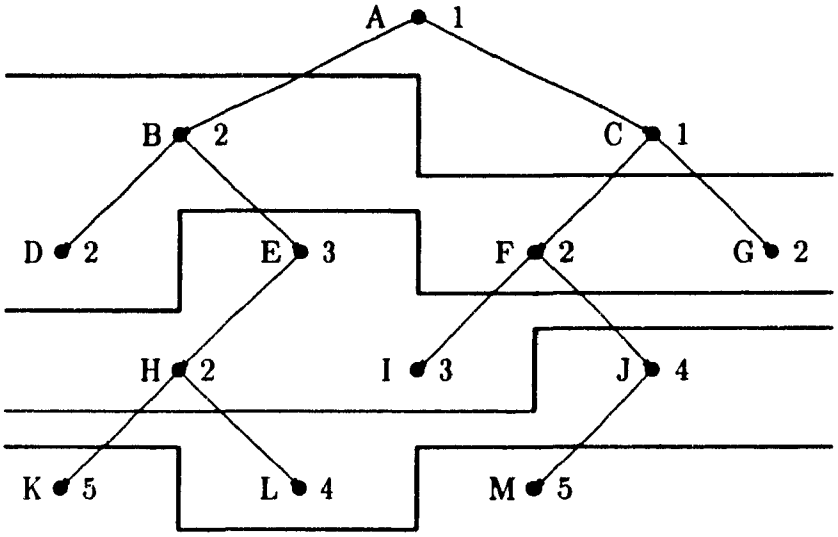

Figure 3.2: Example of an IDA* Search

are less than or equal to the f-value of node K are shown. The cost bound of the initial iteration is set to the f-value of the start node A. Therefore, C_1 is equal to 1. Nodes A and C are selected for expansion on the initial iteration. The cost bound of the second iteration is set to the minimum f-value among expandable nodes that exceeded the cost bound of the initial iteration. Therefore, C_2 is equal to 2. Nodes B, D, F and G in addition to nodes A and C are selected for expansion on the second iteration. Similarly, the cost bound of the third iteration C_3 is equal to 3. Nodes E, H and I are selected for expansion on the third iteration in addition to all nodes that were selected for expansion on iteration 2. To note, because the f-value of node H is non-monotonic, it is initially selected for expansion only on iteration 3. On the fifth and final iteration, IDA* terminates once a goal node is reached and

hence, only nodes A, B, D, E, H and K are selected for expansion

3.1.1 Admissibility

By extending the search from one iteration to the next along contours of minimum increments, three preliminary results are stated.

Theorem 3.1 ([24]) *Given a solution space graph* $G = (s, \Upsilon, h | h \leq h^*)$, *IDA* is admissible.*

Proof Outline: The cost bound of the initial iteration C_1 is equal to the f-value of the start node s. Since $f(s) \leq f^*(s) = C^*$, the cost bound C_1 is less than or equal to the cost of the optimal solution path. Furthermore, since the cost bound of each succeeding iteration is the minimum f-value among expandable nodes that exceeded the previous cost bound and since the f-value of each node along an optimal solution path does not exceed C^*, the cost bound of some iteration must eventually equal the cost of an optimal solution path. When the cost bound is equal to C^*, the optimal goal node is selected and no other goal node with an f-value greater than C^* is expanded. Therefore, the cost of the first goal node selected for expansion is equal to the cost of an optimal solution path.□

Lemma 3.1 *Given a solution space graph* $G = (s, \Upsilon, h | h \leq h^*)$, *the cost bound of the final iteration of IDA* is equal to the cost of the optimal solution path* C^*.

Lemma 3.2 *Given a solution space graph* $G = (s, \Upsilon, h | h \leq h^*)$, *IDA* expands at least one additional node on iteration* i *where the* f*-value of that node is equal to the cost bound* C_i.

Proof: The cost bound C_1 of the initial iteration is equal to the f-value of the start node s. Since the start node is selected on each iteration, at least one additional node whose f-value is equal to C_1, namely the start node s, is selected for expansion on the initial iteration. The cost bound of iteration i is set to the minimum f-value among all expandable nodes that exceeded the cost bound C_{i-1}. Hence, there are no paths from the start node s along which the maximum f-value is greater than C_{i-1} and less than C_i. Therefore, at least one additional node whose f-value is equal to C_i must be selected for expansion on iteration i.□

3.1.2 Conditions of Expansion

Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$, the sufficient and necessary conditions of expansion are stated for any node n on iteration i:

Sufficient Condition: There exists a path from the start node s to n along which each node has an f-value that is less than or equal to the cost bound C_i where C_i is also strictly less than C^*.

Necessary Condition: There exists a path from the start node s to n along which each node has an f-value that is less than or equal to the cost bound C_i where C_i is also less than or equal to C^*.

Like the A* algorithm, the sufficient and necessary conditions for the expansion of node n are simplified for monotonic h:

Sufficient Condition: $g^*(n) + h(n) \leq C_i$ and $C_i < C^*$.

Necessary Condition: $g^*(n) + h(n) \leq C_i$ and $C_i \leq C^*$.

It is important to note that the sufficient and necessary conditions of expansion are identical on each iteration leading up to and including the penultimate iteration but differ on the the final iteration.

If the evaluation function satisfies the property of monotonicity, the f-values of the sequence of nodes expanded by IDA* are *not* necessarily non-decreasing. Monotonicity ensures only that the f-values along any path rooted at the start node s are non-decreasing. Therefore, on the final iteration when an optimal goal node is selected for expansion, a subset of nodes that were expanded on the penultimate iteration may not be expanded before an optimal goal node is selected. Each node that is expanded on the final iteration satisfies only the necessary condition of expansion, that is, $f(n) \leq C^*$. Thus, only the start node s and the optimal goal node may be selected for expansion on the final iteration as shown in Figure 3.3. If nodes A and B are designated as the start and goal nodes respectively then only nodes A and B are selected for expansion on the final iteration: The solution space tree rooted at node C is unexplored.

Although the necessary condition holds over all iterations, the sufficient condition of expansion holds only on each iteration leading up to and including the penultimate iteration.

Definition 3.1 *A node n is surely-expanded by IDA* on iteration i if and only if it satisfies the sufficient condition of expansion.*

Definition 3.2 *A node n is said to be C_i-surely-expanded if it is surely-expanded on iteration i.*

Any node that is surely-expanded on, say, four successive iterations is said to be surely-expanded four times. Although the same node is selected for

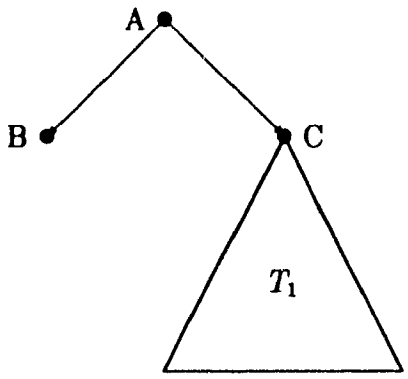

Figure 3.3: Expansion on the Final Iteration of IDA*

expansion on each of the four iterations, it is said as well that four nodes are surely-expanded. Finally, no nodes are surely-expanded on the final iteration of IDA*.

Lemma 3.3 *Given a solution space graph* $G = (s, \Upsilon, h | h \leq h^*)$, *the set of distinct nodes that are surely-expanded by A* is equal to the set of distinct nodes that are surely-expanded by IDA* on its penultimate iteration.*

Proof: From Lemma 3.1, the minimum f-value that exceeded the cost bound of the penultimate iteration is equal to the cost of the optimal solution path C^*. Therefore, on the penultimate iteration, IDA* expands all nodes n for which there exists a path from the start node s to n and for which each node along that path has an f-value that is strictly less than C^*. Thus, Definition 2.2 is satisfied.□

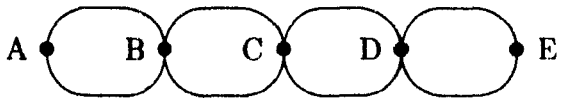

Figure 3.4: Non-Acyclic Solution Space Graph

3.1.3 Property of Acyclicy

The property of monotonicity cannot guarantee, as it does with A*, that no node is re-selected for expansion during each iteration of an IDA* search. On each iteration, IDA* performs a bounded depth-first search which by definition, maintains only the current path from the start node s to the most recently-generated node n. Therefore, if a node n is reached by more than one path from the start node s then node n is re-expanded for each assigned f-value that is less than or equal to the cost bound of the current iteration.

For example, the explicit graph G in Figure 3.4 is considered. Nodes A and E are designated as the start and goal nodes respectively. Each edge is directed away from the start node A and has unit cost. If no heuristic information is employed ($h = 0$) then the property of monotonicity is ensured. The complete expansion of either node A, B, C or D generates two instances of node B, C, D or E respectively. Each instance is assigned a cost which is equal to its distance from the start node A. According to algorithm BF* (Figure 1.1), only the first instance of each generated node is added to OPEN (Step 6b): The second instance is removed (Step 6c).

The IDA* algorithm, on the other hand, carries out successive depth-first

searches that are each rooted at node A and that are bounded by depths of 0 to 4 respectively. Each partial expansion of A, B, C or D (Step 5, Figure 3.1) generates a single instance of B, C, D or E at a time. Therefore, on the first iteration, only node A is expanded. On the second iteration, node A is again expanded but node B is selected twice for expansion, once for each instance that is generated. Similarly, on the third and fourth iterations, nodes C and D are expanded 4 and 8 times respectively. Because all paths from A to E are optimal, only 5 nodes are selected for expansion before IDA* is terminated on the fifth and final iteration. Hence, the performance of IDA* degrades exponentially over A* as the number of distinct paths to each node grows exponentially with depth (cost).

In order to ensure that no node is re-selected for expansion, the property of monotonicity is supplanted by the stronger requirement of acyclicy.

Definition 3.3 *The property of acyclicy states that no node in G is reachable by more than one path from the start node s.*

The property of acyclicy is easily guaranteed if the solution space graph is a tree rooted at the start node s. Hence, there exists at most one path to and therefore, one possible expansion of each node in G on any iteration.

3.2 Measures of Performance

3.2.1 Node Complexity

In light of Lemma 3.3, the arguments that support the node optimality of A* are equally valid in establishing Theorem 3.2.

Theorem 3.2 *Given a solution space graph $G = (s, \Upsilon, h|h \leq h^*)$, IDA* is optimal, in terms of the number of distinct nodes that are surely-expanded, over the class of admissible best-first search algorithms.*

The property of acyclicy is not a prerequisite of node optimality, that is, the total number of node expansions does not affect the number of distinct nodes that are selected for expansion. However, assuming that the property of acyclicy is satisfied on G, the number of distinct nodes and the total number of nodes that are surely-expanded by the A* algorithm are equal. The criteria of A* optimality is restated without the condition of monotonicity.

Lemma 3.4 *Given a solution space tree $G = (s, \Upsilon, h|h \leq h^*)$, the A* algorithm is optimal, in terms of the total number of nodes that are surely-expanded, over the class of admissible best-first tree search algorithms.*

Unfortunately, the interchange of node and time complexity is not so obviously extended in the case of the IDA* algorithm as shown in the following section.

3.2.2 Time Complexity

Under the property of acyclicy, a subtle variation of Lemma 3.3 is stated.

Lemma 3.5 *Given a solution space tree $G = (s, \Upsilon, h|h \leq h^*)$, the number of nodes that are surely-expanded by A* is equal to the number of nodes that are surely-expanded by IDA* on its penultimate iteration.*

Lemma 3.5 establishes a common measure of time complexity between the A* and IDA* algorithms. Because no nodes are surely-expanded by IDA* on the final iteration, the time complexity of IDA* is restricted to the total number of nodes that are surely-expanded on each iteration leading up to and including the penultimate iteration.

Definition 3.4 *The penultimate iteration is said to be iteration k, $k \geq 0$. If $k = 0$ then the optimal solution path is found on the initial iteration and no node is surely-expanded.*

Definition 3.5 *The number of nodes that are C_i-surely-expanded, $0 \leq i \leq k$, is denoted $M(i)$. From Definition 3.4, $M(0)$ is equal to 0.*

Definition 3.6 *The number of additional nodes that are surely-expanded on iteration i, $1 \leq i \leq k$, is denoted $N(i)$ and is equal to the number of nodes that are C_i-surely-expanded less the number of nodes that are C_{i-1}-surely-expanded. Hence,*

$$N(i) = M(i) - M(i-1).$$

From Definition 3.6, it follows that

$$M(i) = \sum_{j=1}^{i} N(j).$$

Definition 3.7 *The total number of nodes that are surely-expanded by IDA* over k iterations is denoted $T(k)$ and is equal to*

$$\sum_{i=1}^{k} M(i) \tag{3.1}$$

or equivalently

$$\sum_{i=1}^{k} \sum_{j=1}^{i} N(j). \tag{3.2}$$

It is also useful to note that each additional node that is surely-expanded on iteration i is re-expanded on each subsequent iteration j, $i+1 \leq j \leq k$, for a total of $(k-i+1)$ expansions. The additional nodes that are surely-expanded on the first iteration are re-expanded on each subsequent iteration for a total of $kN(1)$ expansions. The additional nodes that are surely-expanded on the

second iteration are re-expanded on each subsequent iteration for a total of $(k-1)N(2)$ expansions. In general, the additional nodes that are surely-expanded on iteration i are re-expanded on each subsequent iteration for a total of $(k-i+1)N(i)$ expansions. Hence, the time complexity $T(k)$ of IDA* is also equal to

$$\sum_{i=1}^{k}(k-i+1)N(i). \tag{3.3}$$

Equations 3.1, 3.2 and 3.3 above represent equivalent general formulae of the time complexity of iterative-deepening-A*.

3.2.3 Time Optimality

The number of nodes that are surely-expanded by A* on an admissible solution space tree G is defined in terms of the number of nodes that are surely-expanded by IDA* on G during its penultimate iteration. Therefore, the time complexity of A* is equal to $M(k)$. This equivalence allows a convenient and reciprocal measure of comparison between the A* and IDA* algorithms. Because the A* algorithm is optimal, in terms of the number of surely-expanded nodes, over the class of admissible best-first tree search algorithms, the IDA* algorithm is also asymptotically optimal if $T(k)$ is $O(M(k))$.

The time optimality of IDA*, however, depends on what fraction of the overall computational effort is spent on those iterations i, $1 \leq i \leq k-1$, leading up to penultimate iteration k.

Definition 3.8 *The overhead of the IDA* algorithm, denoted $R(k)$, is equal to the total number of nodes that are surely-expanded on all iterations leading up to but not including the penultimate iteration.*

To express $T(k)$ in terms of $M(k)$ and $R(k)$, Equations 3.1 and 3.2 are respec-

tively rewritten as

$$\begin{aligned} T(k) &= M(k) + \sum_{i=1}^{k-1} M(i) \\ &= M(k) + R(k) \end{aligned} \tag{3.4}$$

and

$$\begin{aligned} T(k) &= M(k) + \sum_{i=1}^{k-1} \sum_{j=1}^{i} N(j) \\ &= M(k) + R(k). \end{aligned} \tag{3.5}$$

Therefore, to ensure the asymptotic time optimality of IDA*, the overhead $R(k)$ must be $O(M(k))$ as well.

3.3 Direct Derivation of Time Complexity

Given a sequence A of p nodes $n_1, n_2, \ldots, n_p$ that are surely-expanded by the A* algorithm on an admissible solution space tree G, the time complexity of IDA* is derived directly from sequence A. First, an ascending subsequence of A, denoted A', is constructed as follows:

1. The first element of A' is n_{j_1} where $n_{j_1} = n_1$.
2. The i^{th} element of A', $i \geq 2$, is the next element of A from $n_{j_{i-1}}$ onward whose f-value is strictly greater than $f(n_{j_{i-1}})$.

The construction continues until the entire sequence of A has been scanned for ascending f-values. The resulting subsequence A' is $n_{j_1}, n_{j_2}, \ldots, n_{j_k}$ where $n_{j_1} = n_1$.

Property 3.1 *The f-values of all nodes between n_{j_i} and $n_{j_{i+1}}$ in sequence A are less than or equal to $f(n_{j_i})$.*

Lemma 3.6 *Given the sequence A of nodes that are surely-expanded by A* on a solution space tree* $G = (s, \Upsilon, h | h \leq h^*)$, *the successive cost bounds* $C_1, C_2, \ldots, C_k$ *of IDA* on G are respectively equal to the f-values of the ascending subsequence A', that is,* $C_1 = f(n_{j_1}), C_2 = f(n_{j_2}), \ldots, C_k = f(n_{j_k})$.

Proof by Induction: The cost bound of the initial iteration C_1 is equal to the f-value of the start node n_1 which is equal to $f(n_{j_1})$ by definition. Assume that the cost bound C_i of iteration i is equal to $f(n_{j_i})$. Clearly, all nodes n_1 through $n_{j_{i+1}-1}$ are C_i-surely-expanded [Property 3.1]. The cost bound of the subsequent iteration is equal to the minimum f-value that exceeds C_i. Since the A* algorithm selects at each step the node with the minimum f-value among all expandable nodes then $n_{j_{i+1}}$ is a node with the minimum f-value that exceeds the cost bound of iteration i. Therefore, $C_{i+1} = f(n_{j_{i+1}})$.□

Lemma 3.7 *Given the sequence A of nodes that are surely-expanded by A* on a solution space tree* $G = (s, \Upsilon, h | h \leq h^*)$, *there exists a partition of A into mutually-exclusive subsequences* $A_1, A_2, \ldots, A_k$ *such that each element of* A_i *is* C_i*-surely-expanded but not* C_{i-1}*-surely-expanded,* $1 \leq i \leq k$

Proof: Let the sequence A be partitioned into mutually-exclusive and non-empty subsequences $A_1, A_2, \ldots, A_k$ where the first element of each subsequence A_i is the i^{th} element of A', that is, the f-value of the first element of A_i is equal to the cost bound of iteration i. Hence, subsequence A_i is equal to $n_{j_i}, n_{j_i+1}, \ldots, n_{j_{i+1}-1}$, $1 \leq i \leq k$.

Assume that a node m of subsequence A_i is not C_i-surely-expanded. Therefore, there exists a node t along the path from s to m such that $f(t) > C_i$. From Property 3.1, the f-values of all nodes in subsequence A_i are less than or equal to $C_i = f(n_{j_i})$ which, in turn, is greater than the f-values of all preceding nodes n_1 through n_{j_i-1} of sequence A. Hence, node t cannot exist.

Assume now that node m of subsequence A_i is also C_{i-1}-surely-expanded. Therefore, $f(m) < C_i = f(n_{j_i})$ which implies that node m is distinct from node n_{j_i}. If node m is C_{i-1}-surely-expanded then the f-value of each node along the path P from s to m is less than or equal to C_{i-1}. Therefore, A* would select node m for expansion before node n_{j_i} which has a greater f-value. But node m was selected for expansion *after* node n_{j_i}. Therefore, path P cannot exist.□

Lemma 3.8 *Given the sequence A of nodes that are surely-expanded by A* on a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$, the number of additional nodes $N(i)$ that are expanded by IDA* on iteration i is equal to the length of the subsequence A_i.*

Once the sequence A has been partitioned into mutually exclusive and non-empty subsequences, it is straightforward to substitute the values of $N(i)$ into either Equation 3.2 or Equation 3.3 in order to determine the time complexity of IDA*. A simple example illustrates the above results.

Example: Let $A = n_1, n_2, \ldots, n_9$ and let the respective f-values of A be 3,2,2,4,5,5,3,7,6. It follows that $A' = n_1, n_4, n_5, n_8$ and that $A_1 = n_1, n_2, n_3$, $A_2 = n_4$, $A_3 = n_5, n_6, n_7$ and $A_4 = n_8, n_9$. Therefore, IDA* performs 4 iterations where $N(1) = 3$, $N(2) = 1$, $N(3) = 3$ and $N(4) = 2$ [Lemma 3.8]. The time complexity of IDA* is evaluated by Equation 3.3 and is equal to

$$T(4) = \sum_{i=1}^{4} (4 - i + 1) N(i) = 12 + 3 + 6 + 2 = 23 \text{ nodes.}$$

3.4 Comparison of IDA* with A*

The A* and IDA* algorithms represent the two ends of the space complexity spectrum. On one hand, the A* algorithm maintains all feasible paths that

are rooted at the start node s. Hence, minimal pruning is performed. On the other hand, the IDA* algorithm prunes all paths, except the current path, immediately after a node expansion. Two algorithmic "bridges" between the IDA* and A* algorithms called MREC and MA* were recently proposed in [47] and [6] respectively. In both cases, the heuristic search is admissible and performed within a memory constraint that is treated as a parameter of the search. Essentially, MREC carries out a best-first search until the maximum memory constraint is reached and then regenerates nodes in a fashion similar to depth-first iterative-deepening until an optimal solution path is found. On the other hand, MA* carries out a best-first search until the the maximum memory constraint is reached and then selectively prunes its leaf nodes from the OPEN list [3]. To help retain the cost information of descendent nodes, MA* uses bottom-up cost revision to update the f-values of ancestral nodes.

Lemma 3.5 provides a common measure of comparison between the A* and IDA* algorithms subject to one consideration. The implementations of the A* and IDA* algorithms employ different data structures to maintain the list of expandable nodes. On one hand, the A* algorithm employs a priority queue. The priority queue, implemented as a heap, requires on average $O(\log M)$ time to insert and delete each node where M is the number of elements in the heap [1]. On the other hand, the IDA* algorithm employs a stack which requires at most $O(1)$ or constant time to insert and delete each node. Therefore, a logarithmic proportionality factor is introduced. If the criteria of comparison is measured only with respect to the number of node expansions then an interesting phenomenon can arise. The real execution time of IDA* may be *less* than the execution time of A* although IDA* will surely-expand at least

[3]The parent of a pruned leaf node may also be removed from CLOSED, put in OPEN and subsequently pruned itself

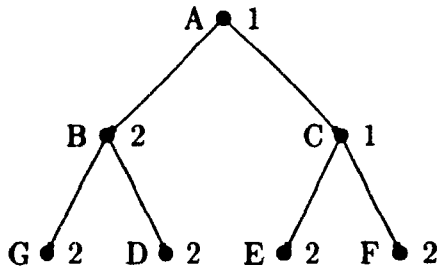

Figure 3.5: Example of the Final Iteration of IDA*

the number of nodes that are surely-expanded by A*. This phenomenon was first noted in [23] and provides an additional incentive to incorporate the simpler selection criteria of a backtracking versus a best-first strategy.

If the IDA* algorithm expands $M(k)$ nodes during its penultimate iteration on an admissible solution space tree G then A* surely-expands the same number of nodes on G. Unfortunately, there exists no common measure of comparison that considers the number of nodes that are expanded on the final iteration. The difficulty in establishing a common measure of comparison is due to the difference between the tie-breaking rules of A* and IDA*. The IDA* algorithm selects each node according to a depth-first criteria, that is, the most-recently generated node whose f-value is less than or equal to the cost bound of the current iteration. Even if the tie-breaking rule of A* favours a last-in, first-out policy as suggested in [23, Theorem 6.4], it is no guarantee that the same set of additional nodes will be expanded by IDA* on its final iteration as shown in Figure 3.5

Each node A through G is labeled with its f-value. Nodes A and G are

designated as the start and goal nodes respectively. Because the A* algorithm selects node C for expansion after the selection of node A, nodes E and F are more "recently-generated" than node B. Assuming that A* employs a last-in, first-out tie-breaking policy, nodes E and F are selected for expansion before node B. However, on the second and final iteration of IDA*, only nodes A, B and G are selected for expansion, thus avoiding the unnecessary expansions of nodes E and F.

3.5 Concluding Remarks

Chapter 3 has established five important properties of an IDA* search on a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$:

1. IDA* is admissible.

2. No fewer than one additional node is expanded by IDA* on each iteration.

3. The property of acyclicy ensures that no node is re-selected for expansion during an A* search and that no node is re-selected for expansion on each iteration of IDA*.

4. The number of nodes that are surely-expanded by A* on G is equal to the number of nodes that are surely-expanded by IDA* on its penultimate iteration.

5. IDA* is optimal, in terms of the number of distinct nodes that are surely-expanded, over the class of admissible best-first tree search algorithms.

Chapter 4

Worst Case Analysis

4.1 Derivation

The worst case analysis establishes a strict upper bound on the time complexity of IDA*. The upper bound is stated in terms of the number of nodes that are surely-expanded by A* on an admissible solution space tree. To begin, two preliminary lemmas are proved.

Lemma 4.1 *Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$, IDA* performs at most $M(k)$ iterations where $M(k)$ is the number of nodes that are surely-expanded on the penultimate iteration.*

Proof: Follows directly from Lemma 3.2 if exactly one additional node is surely-expanded on each iteration.□

Lemma 4.2 *Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$, IDA* surely-expands in the worst case exactly one additional node per iteration.*

Proof: On the penultimate iteration k, IDA* surely-expands $M(k)$ nodes. On iteration $k-1$, IDA* surely-expands at most $M(k)-1$ nodes since at least one

additional node is expanded on iteration k. Similarly, on iteration $k-2$, IDA* surely-expands at most $M(k)-2$ nodes since again at least one additional node is expanded on both iteration $k-1$ and iteration k. In general, IDA* surely-expands at most $M(k)-i$ nodes on iteration $k-i$ where $0 \leq i \leq M(k)-1$ [Lemma 4.1]. Since the time complexity of IDA* is maximized on each iteration i, $1 \leq i \leq M(k)$, over a maximum number of iterations, it follows immediately from the principle of optimality [12] that the overall time complexity of IDA* is maximized as well. Hence, in the worst case, exactly one additional node is expanded on each iteration leading up to and including the penultimate iteration.□

Theorem 4.1 *Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$, IDA* surely-expands in the worst case $(M^2(k) + M(k))/2$ nodes where $M(k)$ is the number of nodes that are surely-expanded by A* on G.*

Proof: Lemmas 4.1 and 4.2 imply that in the worst case $M(i) = i$. From Equation 3.1,

$$\begin{aligned} T(k) &= \sum_{i=1}^{k} M(i) \\ &= \sum_{i=1}^{k} i \\ &= \frac{k(k+1)}{2}. \end{aligned}$$

Since $k = M(k)$,

$$T(k) = \frac{M^2(k) + M(k)}{2}.\square$$

4.2 Worst Case Conditions

Theorem 4.2 *The worst case behaviour of IDA* is realized on a solution space tree $G = (s, \Upsilon, h|h \leq h^*)$ if and only if the evaluation function used by IDA* (and A*) on G assigns a cost to each node such that the following two conditions are met:*

1. *Condition 1 (Uniqueness): The f-values of all surely-expanded nodes are unique.*
2. *Condition 2 (Monotonicity): The f-values along each path from the start node s to a surely-expanded node are strictly increasing.*

Proof: Two preliminary observations are noted:

1. During any tree search, each node is generated and expanded at most once. Therefore, each node is assigned a single distinct value.
2. There exists a unique path from the start node s to each other node in G.

Let p be the number of nodes that are surely-expanded by A* on G. As well, let $A = n_1, n_2, \ldots, n_p$ be the sequence of nodes that are surely-expanded by A* where n_1 is designated as the start node. The f-values are implicitly defined.

In the worst case, exactly one additional node is surely-expanded by IDA* on each iteration. Therefore, the sequence A is equivalent to its ascending subsequence A' as constructed in Section 3.3. Because A' is equivalent to A, it immediately follows that all f-values must be unique. Furthermore, the f-values along each path from the start node to a surely-expanded node must be strictly increasing; otherwise, the f-values of sequence A are non-increasing.

Given an admissible solution space tree G that satisfies Conditions 1 and 2, the worst case performance of IDA* is realized in the following manner.

By definition, the cost bound of the first iteration, denoted C_1, is set to the cost of the start node n_1, that is, $C_1 = f(n_1)$. A depth-first search is carried out until either a goal node is selected for expansion or the f-values of all expandable nodes are greater than C_1. If an optimal goal node is not selected on the first iteration then $f(n_1)$ is unique and less than all f-values along any path rooted at the start node. Hence, only the start node n_1 is selected for expansion on the first iteration. The cost bound of the second iteration C_2 is set to the minimum f-value, say $f(n_2)$, that exceeded $f(n_1)$ on the first iteration. If an optimal goal node is not selected on the second iteration then $f(n_2)$ is unique. Hence, under the condition of monotonicity, only nodes n_1 and n_2 are selected for expansion on the second iteration. Without loss of generality, on any iteration i that does not select an optimal goal node, at least one node n_{i+1} has an f-value that is equal to the minimum f-value that exceeded C_i. If an optimal goal node is not selected on iteration $i+1$ then $f(n_{i+1})$ is unique and node n_{i+1} along with all nodes n_j, $1 \leq j \leq i$, are selected for expansion on iteration $i+1$. Therefore, exactly one additional node is expanded on each iteration that does not select a goal node. It also follows that p iterations are required to encompass and expand the set of all nodes that are surely-expanded by A*.□

Theorem 4.1 is based on the conditions of uniqueness and monotonicity. Theorem 4.3 below establishes the non-vacuity of Conditions 1 and 2 and more importantly, reveals that the worst case behaviour of IDA* may be realized on any solution space tree.

Theorem 4.3 *For any solution space tree $G = (s, \Upsilon)$, there exists an admissible heuristic function h, $0 \leq h \leq h^*$, such that Conditions 1 and 2 are satisfied.*

Proof: If $h = h^*$ then no node is surely-expanded by IDA* and the theorem is trivially proved. Assume instead that an admissible monotone heuristic function $h' < h^*$ is applied to each non-goal node. The number of additional nodes that are expanded by IDA* on each iteration i, excluding the final iteration, is equal to the number of nodes whose f-values are equal to the cost bound C_i. Let N_i represent the number of additional nodes that are surely-expanded by IDA* on iteration i. If node $n_{i,j}$ represents the j^{th} additional node that is expanded on iteration i then define an heuristic function h that assigns a value to $n_{i,j}$ such that

$$h(n_{i,j}) = h'(n_{i,j}) + (j-1) \cdot \frac{C_{i+1} - C_i}{N_i}$$

where $1 \leq j \leq N_i$. The evaluation function $f = g + h$ remains admissible since $f(n_{k,j})$ where k is the penultimate iteration remains less than the cost of the optimal solution path $C_{k+1} = C^*$. Because $C_i < C_{i+1}$, the f-value of each additional node that was originally expanded on iteration i is now unique and falls within the interval $[C_i, C_{i+1})$. As well, the sequence of f-values of those additional nodes that were originally expanded on iteration i is monotonically increasing. Therefore, the f-values along any path from the start node to a surely-expanded node are also monotonically increasing. Hence, the worst case conditions of uniqueness and monotonicity are satisfied.□

4.3 Worst Case Examples

In the following two subsections, worst case examples are shown for solution space trees with uniform and non-uniform branching factors. In addition to Conditions 1 and 2 outlined in Theorem 4.2, Condition 3 given below is also assumed:

Condition 3: The f-value of the optimal goal node(s) is greater than the f-values of all non-goal nodes.

Under Condition 3, both A* and IDA* must ultimately expand and therefore surely-expand *all* non-goal nodes before selecting the optimal goal node for expansion.

4.3.1 Uniform Branching Factor

Lemma 4.3 *There exists a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$ with an uniform branching factor $b \geq 1$ and unit edge costs such that Conditions 1, 2 and 3 are satisfied.*

Proof: Let G be a complete, directed binary tree of depth d, that is, $b = 2$. Each edge is directed away from the start node s and has unit cost. Every node at depth d is a goal node and each node is labeled level-by-level, top-to-bottom as $n_1, n_2, \ldots, n_p$ where $p = 2^{d+1} - 1$. Figure 4.1 shows the solution space G when $d = 3$.

Define an heuristic function h as follows:

$$h(n_i) = \begin{cases} 0 & \text{if } n_i \text{ is a goal node} \\ h^*(n_i) - 1 + \sum_{k=1}^{i} 2^{-k} & \text{otherwise.} \end{cases}$$

Condition 1 (Uniqueness): Let n_i and n_j be any two distinct non-goal nodes. The f-values of n_i and n_j are respectively equal to $d - 1 + \sum_{k=1}^{i} 2^{-k}$ and

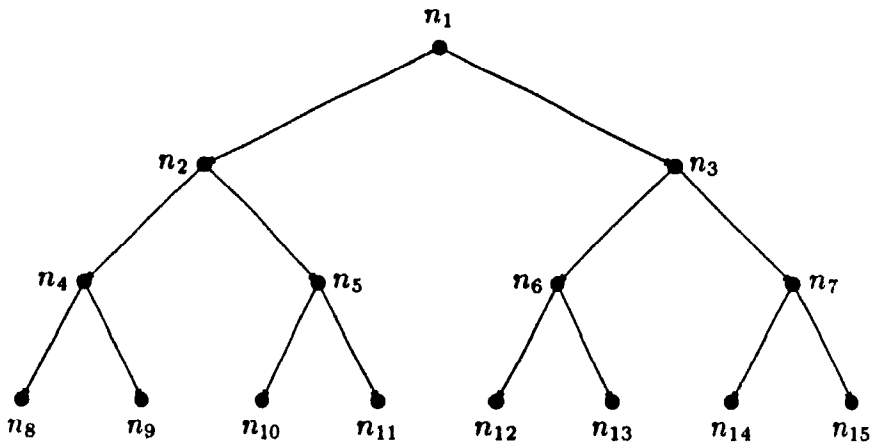

Figure 4.1: Labeled Binary Tree of Depth 3

$d - 1 + \sum_{k=1}^{j} 2^{-k}$. Since $i \neq j$, it follows that the f-values of all non-goal nodes are unique.

Condition 2 (Monotonicity): Assume that node n_j is generated from the expansion of node n_i. Therefore,

$$
\begin{aligned}
f(n_j) &= g(n_j) + h(n_j) \\
&= g(n_i) + 1 + \begin{cases} 0 & \text{if } n_j \text{ is a goal node} \\ h^*(n_j) - 1 + \sum_{k=1}^{j} 2^{-k} & \text{otherwise} \end{cases} \\
&= \begin{cases} g(n_i) + h^*(n_i) & \text{if } n_j \text{ is a goal node} \\ g(n_i) + h^*(n_i) - 1 + \sum_{k=1}^{j} 2^{-k} & \text{otherwise} \end{cases} \\
&> g(n_i) + h^*(n_i) - 1 + \sum_{k=1}^{i} 2^{-k} \qquad \text{since } i < j \\
&= f(n_i).
\end{aligned}
$$

Condition 3: Since the f-value of each goal node is equal to d, all non-goal nodes are expanded before an optimal goal node is selected for expansion.□

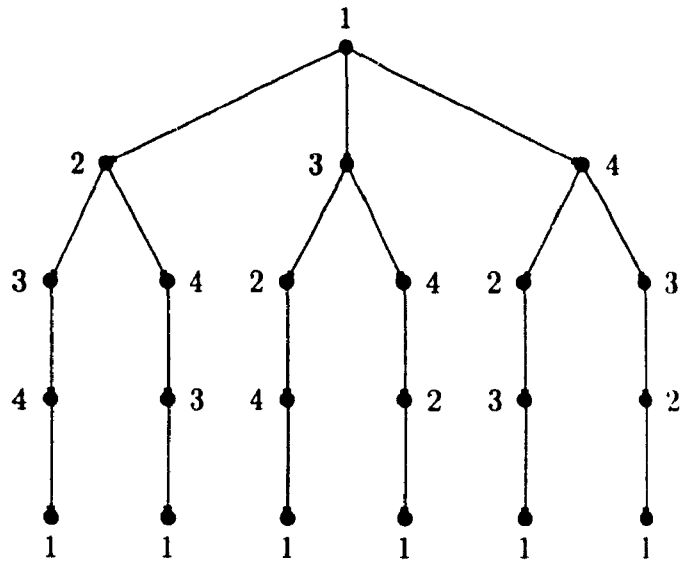

Figure 4.2: Solution Space of a 4-City ATSP

4.3.2 Non-Uniform Branching Factor

An example of an exponential search with a non-uniform branching factor is represented by an instance of the asymmetric traveling salesperson problem (ATSP).

Definition 4.1 *Given a positive adjacency matrix* (c_{ij}) *where each element* c_{ij} *represents the cost of traveling from city* i *to city* j*, the asymmetric traveling salesperson problem is to find the minimum cost tour that begins at an arbitrary city, visits each other city exactly once and returns to the starting city*

The solution space for a 4-city ATSP is shown in Figure 4.2. Each path from the start node to a goal or leaf node represents one of $(4-1)!$ alternate tours

that begin and end at city 1.

Lemma 4.4 *There exists an instance G_m of the m-city ATSP such that $\forall\ m > 1$, Conditions 1, 2 and 3 are satisfied on G_m.*

Proof: Assume, without loss of generality, that the tour begins and ends at city 1. Define an instance of the ATSP by a positive adjacency matrix c_{ij}, $1 \leq i, j \leq m$ with the following properties:

1. For $i = j$, $c_{ii} = \infty$.
2. For $i \neq j$, all c_{ij} are unique and assigned values from the following sets:
   (a) For $j \neq 1$, $c_{ij} \epsilon \{2^1, 2^3, \ldots, 2^{2(m-1)^2-1}\}$,
   (b) For $j = 1$, $c_{ij} \epsilon \{2^{2(m-1)^2+1}, 2^{2(m-1)^2+3}, \ldots, 2^{2m(m-1)-1}\}$.

Condition 1 (Uniqueness): Define a binary word $b_{2m(m-1)-1} \ldots b_2 b_1 b_0$ of length $2m(m-1)$ for each path rooted at the start node s as follows. If c_{ij} with cost 2^k is found along a path from s to, say, node n then set b_k of the binary word representing this path to 1; otherwise, set b_k to 0. Since all paths rooted at the start node s differ by at least one edge (cost), the value of the binary word representing the path from s to n is unique and equal to $g(n)$.

The path from node n to a goal node is stored likewise in the same binary word that represents the path from the start node s to n. Clearly, if the heuristic function $h = 0$ then the f-value of each node remains unique and equal to g. Assume, instead, that $h = \frac{1}{2}h^*$. If c_{ij} with cost 2^k is found along the path from n to a goal node then set b_{k-1} of the binary word representing this path to 1; otherwise, set b_{k-1} to 0. Since a right-shift of one bit position does not affect the bit positions representing the path from the start node s to n, the uniqueness of the cost of each path, and hence its f-value, is preserved.

Condition 2 (Monotonicity): In general, if $h = k \cdot h^*$ where $0 \leq k < 1$ then the following result can be stated When any node n_i is expanded, all newly-generated nodes n_j satisfy

$$h(n_i) = h(n_j) + k \cdot c_{ij}$$

which implies

$$f(n_i) = g(n_i) + h(n_i) < g(n_i) + h(n_j) + c_{ij} = f(n_j).$$

Therefore, for the cases $k = 0$ and $k = \frac{1}{2}$ that satisfy Condition 1 on the problem instance defined above, the f-values along any path rooted at the start node s are also strictly increasing.

Condition 3: Since the return costs from each of the $m - 1$ cities to city 1 are assigned the $2(m-1)$ high order bits of each binary word, all non-goal nodes are expanded before an optimal goal node is selected for expansion □

When $h = 0$ the A* algorithm implements brute-force uniform cost search [33, 38]. An important corollary stems from the above example.

Corollary 4.1 *Given a solution space tree $G = (s, \Upsilon, h|h = 0)$ with non-unit edge costs, IDA* is not asymptotically optimal, with respect to the number of surely-expanded nodes, over the class of admissible brute-force tree search algorithms.*

This result is contrasted with the claim in [23, Theorem 4.2] that given a solution space tree $G = (s, \Upsilon, h|h = 0)$ with unit costs and an uniform branching factor, IDA* is asymptotically optimal over the class of admissible brute-force tree search algorithms.

4.4 Concluding Remarks

For any solution space tree G, it is shown that there exists an admissible heuristic function such that the worst case conditions of uniqueness and monotonicity are satisfied on G. Under these conditions, IDA* surely-expands $(M^2(k) + M(k))/2$ nodes over $M(k)$ iterations. Therefore, in the worst case, the time complexity of IDA* is quadratic with respect to the time complexity of A*. In one interesting instance when $h = 0$, it is shown that IDA* is not asymptotically optimal, in terms of the number of surely-expanded nodes, over the class of admissible brute-force tree search algorithms with non-unit edge costs.

In [24] and the next chapter, it is argued that the asymptotic time optimality of IDA* is ensured if the number of additional nodes that are expanded on each iteration grows exponentially over the number of iterations. A corollary of Theorem 4.3 follows immediately.

Theorem 4.4 *For any solution space tree $G = (s, \Upsilon, h | h \leq h^*)$, there exists an admissible monotone heuristic function h, $0 \leq h \leq h^*$, such that IDA* is asymptotically optimal, in terms of the number of surely-expanded nodes, over the class of admissible best-first tree search algorithms.*

In practice, however, maintaining an exponential growth rate in the number of additional nodes per iteration is computationally risky. One method, suggested in [24] and implemented in [46], pushes out each depth-first search beyond the original, minimum cost bounds of each iteration. In other words, several iterations are amalgamated into one. Unfortunately, a performance measurement, based on the number of surely-expanded nodes, is inappropriate in light of the potentially large number of inadmissible nodes that may be expanded on the final iteration [38, Chapter 6].

Chapter 5

Time Complexity Spectrum of IDA*

5.1 Branching Factors

The time complexity spectrum of IDA* is delineated by its asymptotic optimal and worst case performance. To characterize the efficiency of IDA* search over its time complexity spectrum, two branching factors are introduced

Definition 5.1 *The effective branching factor, denoted $b_e(i)$, is the ratio of the total number of nodes that are expanded on iteration i over the total number of nodes that are expanded on iteration $i-1$, that is, for $2 \leq i \leq k$,*

$$b_e(i) = \frac{M(i)}{M(i-1)}.$$

Definition 5.2 *The heuristic branching factor, denoted $b_h(i)$, is the ratio of the number of additional nodes that are expanded on iteration i over the number of additional nodes that are expanded on iteration $i-1$, that is, for $2 \leq i \leq k$,*

$$b_h(i) = \frac{N(i)}{N(i-1)}.$$

In this chapter, the asymptotic time complexity of IDA* is derived when the effective or heuristic branching is[1]:

Constant:	$b(i) = b > 1,$	$2 \leq i \leq k.$
Decreasing:	$b(i) = (i/(i-1))^r,$	$2 \leq i \leq k$ and $r \geq 1.$
Unit:	$b(i) = b = 1,$	$2 \leq i \leq k.$

In each case, the time complexity of IDA* is stated as an asymptotic function of the time complexity of A*. It is shown that the constant, decreasing and unit branching factors define a relatively smooth degradation from the optimal to the worst case performance of IDA* over its time complexity spectrum.

5.2 Constant Branching Factor

Theorem 5.1 *Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$ with a constant effective or heuristic branching factor greater than one, IDA* is asymptotically optimal, in terms of the number of surely-expanded nodes, over the class of admissible best-first tree search algorithms.*

Proof: *Case I: Constant effective branching factor:* Let $M(1)$ be initialized to 1. Therefore, the total number of nodes $M(i)$ that are surely-expanded on iteration i, $2 \leq i \leq k$, is equal to b_e^{i-1}. The time complexity of IDA* follows immediately from Equation 3.4:

$$\begin{aligned} T(k) &= M(k) + \sum_{i=1}^{k-1} M(i) \\ &= M(k) + \sum_{i=1}^{k-1} b_e^{i-1} \end{aligned}$$

[1]For the unit branching factor, the time complexity of IDA* is evaluated only for b_h since b_e must be strictly greater than one for all iterations leading up to and including the penultimate iteration

$$= M(k) + \frac{b_e^{k-1} - 1}{b_e - 1}.$$

Since $M(k) = b_e^{k-1}$,

$$T(k) < M(k) + \frac{M(k)}{b_e - 1}$$

where $b_e > 1$.

Case II: Constant heuristic branching factor: Let $N(1)$ be initialized to 1. Therefore, the number of additional nodes $N(i)$ that are surely-expanded on iteration i, $2 \leq i \leq k$, is equal to b_h^{i-1}. Furthermore, the total number of nodes that are surely-expanded by A* on G is

$$\begin{aligned} M(k) &= \sum_{j=1}^{k} N(j) \\ &= \sum_{j=1}^{k} b_h^{j-1} \\ &= \frac{b_h^k - 1}{b_h - 1}. \end{aligned} \tag{5.1}$$

The time complexity of IDA* follows immediately from Equation 3.5:

$$\begin{aligned} T(k) &= M(k) + \sum_{i=1}^{k-1} \sum_{j=1}^{i} N(j) \\ &= M(k) + \sum_{i=1}^{k-1} \sum_{j=1}^{i} b_h^{j-1} \\ &= M(k) + \sum_{i=1}^{k-1} \frac{b_h^i - 1}{b_h - 1} \\ &= M(k) + \frac{1}{b_h - 1} \left[(b_h - 1) + (b_h^2 - 1) + (b_h^3 - 1) + \cdots + (b_h^{k-1} - 1) \right] \\ &= M(k) + \frac{1}{b_h - 1} \left[\frac{b_h^k - 1}{b_h - 1} - k \right]. \end{aligned}$$

From Equation 5.1,

$$T(k) < M(k) + \frac{M(k)}{b_h - 1}$$

where $b_h > 1$.

In either case, IDA* performs less than $M(k)/(b-1)$ node expansions during all iterations leading up to but not including the penultimate iteration. Since b is constant and greater than one, $R(k)$ is $\Theta(M(k))$. Therefore,

$$T(k) \in \Theta(M(k)).\square$$

5.3 Decreasing Branching Factor

The analyses of the previous section assume that either the effective or heuristic branching factor b remains constant over all iterations. If, on the other hand, the branching factor is monotonically decreasing with i, that is,

$$b(i) = \left(\frac{i}{i-1}\right)^r$$

where $r \geq 1$ then the asymptotic time complexity of IDA* is no longer optimal over the class of admissible best-first tree search algorithms.

Theorem 5.2 *Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$ with a decreasing effective branching factor, the asymptotic time complexity of IDA* on G, in terms of the number of surely-expanded nodes, is $\Theta(M^{\frac{r+1}{r}}(k))$.*

Proof: Let $M(1)$ be initialized to 1. Given $b(i) = (i/(i-1))^r$, the total number of nodes $M(i)$ that are surely-expanded on iteration i, $2 \leq i \leq k$, is equal to i^r. The time complexity of IDA* follows immediately from Equation 3.5:

$$\begin{aligned} T(k) &= M(k) + \sum_{i=1}^{k-1} M(i) \\ &= M(k) + \sum_{i=1}^{k-1} i^r. \end{aligned}$$

Since

$$\int_0^{k-1} \imath^r \, d\imath \leq \sum_{\imath=1}^{k-1} i^r \leq \int_0^{k-1} (\imath+1)^r \, d\imath, \tag{5.2}$$

the summation falls within the interval

$$\left[\frac{(k-1)^{r+1}}{r+1}, \frac{k^{r+1}}{r+1}\right].$$

Therefore, the overhead $R(k)$ is $\Theta(k^{r+1})$. Because $M(k)$ is equal to k^r,

$$T(k) \in \Theta(M^{\frac{r+1}{r}}(k)).\square$$

Theorem 5.3 *Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$ with a decreasing heuristic branching factor, the asymptotic time complexity of IDA* on G, in terms of the number of surely-expanded nodes, is $\Theta(M^{\frac{r+2}{r+1}}(k))$.*

Proof: Let $N(1)$ be initialized to 1. Given $b(\imath) = (\imath/(\imath-1))^r$, the number of additional nodes $N(i)$ that are surely-expanded on iteration $\imath$, $2 \leq \imath \leq k$, is equal to $\imath^r$. Furthermore, the total number of nodes that are surely-expanded by A* on G is

$$\begin{aligned} M(k) &= \sum_{j=1}^{k} N(j) \\ &= \sum_{j=1}^{k} j^r \\ &\in \Theta(k^{r+1}). \end{aligned} \tag{5.3}$$

The time complexity of IDA* follows immediately from Equation 3.5:

$$\begin{aligned} T(k) &= M(k) + \sum_{\imath=1}^{k-1} \sum_{j=1}^{\imath} N(j) \\ &= M(k) + \sum_{\imath=1}^{k-1} \sum_{j=1}^{\imath} j^r. \end{aligned}$$

Using the bounds established in Equation 5.2, the inner summation falls within the interval

$$\left[\frac{i^{r+1}}{r+1}, \frac{(i+1)^{r+1}}{r+1}\right]$$

and, likewise, the outer summation falls within the interval

$$\left[\frac{(k-1)^{r+2}}{(r+1)(r+2)}, \frac{(k+1)^{r+2}}{(r+1)(r+2)}\right].$$

Therefore, the overhead $R(k)$ is $\Theta(k^{r+2})$. Because $M(k)$ is $\Theta(k^{r+1})$ from Equation 5.3,

$$T(k) \in \Theta(M^{\frac{r+2}{r+1}}(k)).\square$$

5.4 Unit Branching Factor

Theorem 5.4 *Given a solution space tree $G = (s, \Upsilon, h|h \leq h^*)$ with a heuristic branching factor equal to one, the asymptotic time complexity of IDA* on G, in terms of the number of surely-expanded nodes, is $\Theta(M^2(k))$.*

Proof: If $b_h = 1$ then a fixed number of additional nodes are expanded on each iteration. Let $N(1)$ be initialized to c, $c \geq 1$. Therefore, the total number of nodes $M(i)$ that are surely-expanded on iteration i, $1 \leq i \leq k$, is equal to $c \cdot i$. The time complexity of IDA* follows immediately from Equation 3.5:

$$\begin{aligned} T(k) &= M(k) + \sum_{i=1}^{k-1} \sum_{j=1}^{i} N(j) \\ &= M(k) + c \sum_{i=1}^{k-1} \sum_{j=1}^{i} 1 \\ &= M(k) + c \cdot \frac{k(k-1)}{2}. \end{aligned}$$

Since $M(k) = c \cdot k$,

$$T(k) = M(k) + \frac{M(k)[M(k) - c]}{2c}.$$

Therefore, the overhead $R(k)$ is $\Theta(M^2(k))$ and

$$T(k) \in \Theta(M^2(k)).\square$$

The worst case analysis of Chapter 4 is a special case of Theorem 5.4 when $c = 1$.

5.5 Concluding Remarks

The idea of a heuristic branching factor was originally defined in [24] as follows.

Definition 5.3 *The heuristic branching factor b_h is the average ratio of the number of nodes at a given cost to the number of nodes at the next smaller cost, that is,*

$$b_h = \frac{\sum_{i=2}^{k} N(i)/N(i-1)}{k-1}.$$

Using this definition of b_h, the following conjecture is stated in [24]

> Given a non-overestimating (admissible) heuristic function with heuristic branching factor greater than one then IDA* is asymptotically optimal in time over the class of best-first search algorithms that find optimal solutions on a tree.

Although the above conjecture is nearly identical to Theorem 5.1, it fails to exclude the non-optimal case of a decreasing branching factor where

$$b_h = \frac{\sum_{i=2}^{k} (i/(i-1))^r}{k-1} > 1 \quad \forall k \geq 2.$$

The three cases of a constant, decreasing and unit branching factors respectively correspond to the exponential, polynomial and constant growth of

the explored solution space from one iteration to the next. The optimal time complexity of IDA* is ensured, that is,

$$T(k) \in \Theta(M(k))$$

if the effective or heuristic branching factor is constant and greater than one. On the other hand, the worst case time complexity of IDA* is achieved, that is,

$$T(k) \in \Theta(M^2(k))$$

if the heuristic branching factor is exactly equal to one. Finally, if the branching factor is decreasing then the time complexity of IDA* falls between the two extremes above. For a decreasing effective branching factor where $b_e(i) = (i/(i-1))^r$,

$$T(k) \in \Theta(M^{\frac{r+1}{r}}(k))$$

and for a decreasing heuristic branching factor where $b_h(i) = (i/(i-1))^r$,

$$T(k) \in \Theta(M^{\frac{r+2}{r+1}}(k)).$$

The three cases together define a time complexity spectrum of IDA* from its optimal to its worst case performance.

Chapter 6

Expected Case Analysis

6.1 Model of Computation

The expected case analysis of IDA* assumes a probabilistic model of computation. Rooted at the start node s, the solution space graph $G = (V, E, C)$ is modeled as an infinite tree with a constant branching factor $b > 1$. Each edge is directed away from the start node s and is assigned an independent and identically distributed (i.i.d.) random integer. The integer value assigned to each edge $(n_i, n_j) \in E$ represents not the cost of (n_i, n_j) as defined in Chapter 1 but the *differential cost* of (n_i, n_j) as defined below.

Definition 6.1 *Given a solution space graph $G = (V, E, C)$, the differential cost of $(n_i, n_j) \in E$, denoted $\Delta f(n_i, n_j)$, is equal to the difference between the f-values assigned to each of its end nodes, that is,*

$$\Delta f(n_i, n_j) = f(n_j) - f(n_i)$$

Lemma 6.1 *Given a solution space graph $G = (V, E, C)$, if $\Delta f(n_i, n_j) \geq 0$ for all $(n_i, n_j) \in E$ then the evaluation function f satisfies the property of monotonicity.*

Proof: Since $\Delta f \geq 0$, it follows that $f(n_j) \geq f(n_i)$ for all $(n_i, n_j) \in E$.□

Definition 6.2 *Given a solution space graph $G = (V, E, C)$, the differential cost of a path $n_1, n_2, \ldots, n_p$ where $s = n_1$ is equal to the sum of the differential edge costs from the start node s to node n_p.*

Therefore, the differential cost of a path $n_1, n_2, \ldots, n_p$ is equal to

$$\begin{aligned} & \sum_{i=1}^{p-1} \Delta f(n_i, n_{i+1}) \\ = & \sum_{i=1}^{p-1} [f(n_{i+1}) - f(n_i)] \\ = & f(n_p) - f(s). \end{aligned}$$

Unlike the expected case analyses of [4, 37], the heuristic value defined at each node n is coalesced into the differential edge costs along the path from s to n. If no heuristic information is available ($h = 0$) then $\Delta f(n_i, n_j)$ is equal to $C(n_i, n_j)$ for each edge $(n_i, n_j) \in E$. On the other hand, if heuristic information is perfect ($h = h^*$) then $\Delta f(n_i, n_j)$ is equal to 0 for each edge (n_i, n_j) along an optimal solution path. Figure 6.1 shows Figure 3.2 as labeled with its differential edge costs.

6.2 Basic Formulations

Without loss of generality, the cost of the start node s is initialized to 0. Hence, the f-value of each node n is equal to the differential cost of the path from s to n. Assuming the property of monotonicity, the number of additional nodes $N(i)$ that are surely-expanded by IDA* on iteration i is defined as the number of paths with an differential cost of i [1]. Thus, the cost bound C_i of iteration i

[1] It is important to note that the number of paths with a differential cost of i may not be greater than zero which violates the requisite that at least one additional node is expanded

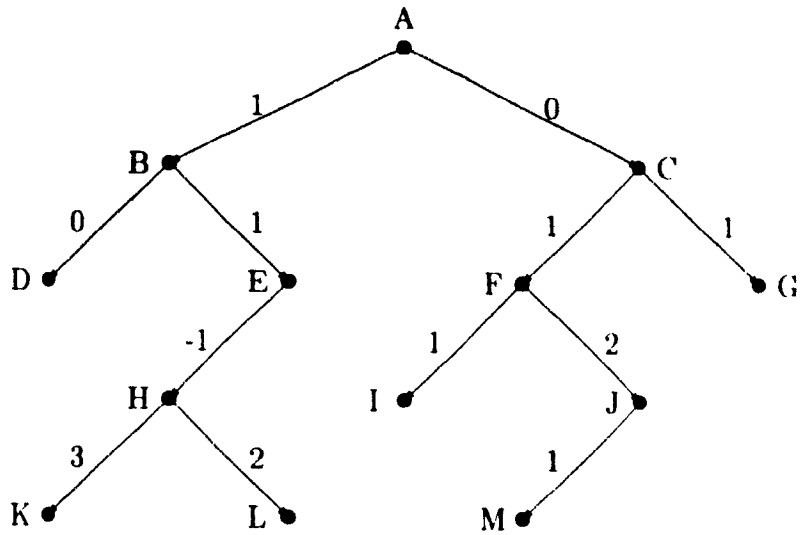

Figure 6 1: Differential Edge Costs of Figure 3.2

is also equal to i. The total number of nodes that are expanded on iteration $k \geq 0$ and the overall time complexity of IDA* are respectively

$$M(k) = \sum_{i=0}^{k} N(i)$$

and

$$T(k) = \sum_{i=0}^{k} M(i)$$

The expected values of $N(k)$, $M(k)$ and $T(k)$, denoted $EN(k)$, $EM(k)$

on each iteration However, in light of the exponential growth of $T(k)$ derived in Section 6 3, this flaw is factored out in the asymptotic analysis

and $ET(k)$, are defined in terms of the generating functions ν, μ and τ. Thus,

$$\begin{aligned} \nu(z) &= \sum_{k=0}^{\infty} EN(k)z^k, \\ \mu(z) &= \sum_{k=0}^{\infty} EM(k)z^k \end{aligned}$$

and

$$\tau(z) = \sum_{k=0}^{\infty} ET(k)z^k.$$

By convention,

$$\begin{aligned} EN(k) &= [z^k]\nu(z), \\ EM(k) &= [z^k]\mu(z) \end{aligned}$$

and

$$ET(k) = [z^k]\tau(z)$$

are the coefficients of z^k in the expansions of ν, μ and τ respectively.

The differential edge costs are treated as random non-negative integers and are characterized by an integer probability distribution defined in terms of the generating function ϕ where

$$\phi(z) = \sum_{j=0}^{\infty} P(X = j)z^j.$$

In addition, the definition of ϕ satisfies the following two properties.

Property 6.1 $\phi(0) = P(X = 0) < 1$.

Property 6.2 $\gcd\{j | P(X = j) > 0\} = 1$.

Properties 6.1 and 6.2 are motivated by the following two lemmas.

Lemma 6.2 *If $\phi(0) < 1$ then ϕ is monotonically increasing in the interval* $[0, 1]$.

Lemma 6.3 *If* $\gcd\{j | P(X = j) > 0\} = 1$ *then there exists a constant* $k_0 \geq 0$ *such that* $\sum_{j=0}^{\infty} P(X_1 + \cdots + X_j = k) > 0$, $\forall\, k \geq k_0$.

Lemma 6.3 ensures that asymptotically, the expected number of additional nodes that are surely-expanded on iteration i is greater than zero.

Given a probability distribution ϕ that satisfies Properties 6.1 and 6.2, the expected number of paths in the solution space tree that have an differential cost of k is stated directly. Thus,

$$EN(k) = [z^k]\nu(z) = \sum_{j=0}^{\infty} b^j P(X_1 + \cdots + X_j = k)$$

where $b > 1$. Therefore, given that all X_i's are independent and identically distributed,

$$\begin{aligned}
\nu(z) &= \sum_{k=0}^{\infty} z^k \sum_{j=0}^{\infty} b^j P(X_1 + \cdots + X_j = k) \\
&= \sum_{j=0}^{\infty} b^j \sum_{k=0}^{\infty} z^k P(X_1 + \cdots + X_j = k) \\
&= \sum_{j=0}^{\infty} b^j E(z^{X_1 + \cdots + X_j}) \\
&= \sum_{j=0}^{\infty} b^j E(z^{X_1})^j \\
&= \sum_{j=0}^{\infty} (b\phi(z))^j \\
&= \begin{cases} (1 - b\phi(z))^{-1} & \text{if } b\phi(z) < 1 \\ \infty & \text{if } b\phi(z) \geq 1. \end{cases}
\end{aligned}$$

The functions μ and τ are also expressible in terms of ν as shown in Lemma 6.4

Lemma 6.4 *Given $z < 1$ and $b\phi(z) < 1$,*

1. $\mu(z) = \dfrac{\nu(z)}{1-z}$
2. $\tau(z) = \dfrac{\nu(z)}{(1-z)^2}$.

Proof: Since $EM(k) = \sum_{j=0}^{k} EN(j)$,

$$\begin{aligned}
(1-z)\mu(z) &= (1-z)\sum_{k=0}^{\infty} EM(k)z^k \\
&= (1-z)\sum_{k=0}^{\infty}\left(\sum_{j=0}^{k} EN(j)\right)z^k \\
&= \sum_{k=0}^{\infty}\left(\sum_{j=0}^{k} EN(j)\right)z^k - \sum_{k=0}^{\infty}\left(\sum_{j=0}^{k} EN(j)\right)z^{k+1} \\
&= EN(0) + \sum_{k=1}^{\infty}\left(\sum_{j=0}^{k} EN(j)\right)z^k - \sum_{k=1}^{\infty}\left(\sum_{j=0}^{k-1} EN(j)\right)z^k \\
&= EN(0) + \sum_{k=1}^{\infty}\left(\sum_{j=0}^{k} EN(j) - \sum_{j=0}^{k-1} EN(j)\right)z^k \\
&= EN(0) + \sum_{k=1}^{\infty} EN(k)z^k \\
&= \nu(z).
\end{aligned}$$

Therefore, $\mu(z) = \nu(z)/(1-z)$.

Similarly, since $ET(k) = \sum_{j=0}^{k} EM(j)$, it follows that $\tau(z) = \mu(z)/(1-z)$. Given $\mu(z) = \nu(z)/(1-z)$ from above, $\tau(z) = \nu(z)/(1-z)^2$.□

6.3 Expected Case Analysis

For completeness, the following definitions of complex analysis are included.

Definition 6.3 *A complex function $\psi(z)$ of a complex variable z is said to be analytic at the point $z = \theta$ if it is differentiable at θ.*

Definition 6.4 *If a complex function $\psi(z)$ is analytic in an open disk centred at θ but is not analytic at θ itself then ψ is said to have an isolated singularity at θ.*

Definition 6.5 *A function $\psi(z)$ is said to be analytic in the open region R if and only if it is analytic at every point in R.*

Definition 6.6 *If there exists an isolated singularity at the point θ but there also exists an integer $m > 0$ such that $(z-\theta)^m\psi(z)$ is analytic at θ then $\psi(z)$ is said to have a pole at θ. The smallest m that makes $(z-\theta)^m$ analytic at θ is called the order of the pole.*

Definition 6.7 ([20]) *A function $\psi(z)$ is called meromorphic if it is analytic in a bounded region R except for a finite number of poles.*

The function ν and by extension, the functions μ and τ have isolated singularities at the points which satisfy

$$b\phi(z) = 1 \tag{6.1}$$

It is within the proximity of these points that the asymptotic behaviour of the coefficients of ν, μ and τ follows directly from the lemmas below

Lemma 6.5 *There exists a unique real solution, denoted z_0 where $0 < z_0 < 1$, of Equation 6.1 if and only if $1 < b < 1/\phi(0)$.*

Proof. Follows immediately from Lemma 6.2.□

A solution of Equation 6.1 at $z_0 = 0$ implies that there exists an infinite number of paths with a differential cost of 0. Since $b\phi(0) = 1$,

$$EN(0) = \sum_{j=0}^{\infty} b^j \phi(0)^j = \infty.$$

Therefore, Property 6.1 is superseded by Property 6.3 below.

Property 6.3 $\phi(0) = P(X = 0) < 1/b$.

Lemma 6.6 *If* $1 < b < 1/\phi(0)$ *then any solution of Equation 6.1 where* $z \neq z_0$ *has* $|z| > z_0$.

Proof: It is shown first that $|z| \geq z_0$. Indeed, for any solution z of Equation 6.1 where $z \neq z_0$,

$$\begin{aligned} b\phi(z_0) &= |b\phi(z)| \\ &= |b\sum_{j=0}^{\infty} P(X = j)z^j| \\ &\leq b\sum_{j=0}^{\infty} P(X = j)|z^j| \\ &= b\sum_{j=0}^{\infty} P(X = j)|z|^j \\ &= b\phi(|z|). \end{aligned}$$

Because ϕ is monotonically increasing, $|z| \geq z_0$. To show that $|z| > z_0$, assume now that $z = z_0 e^{i\theta}$. Then in order that

$$\begin{aligned} &\sum_{j=0}^{\infty} P(X = j)z_0^j e^{ij\theta} \\ = &\sum_{j=0}^{\infty} P(X = j)z^j \\ = &\sum_{j=0}^{\infty} P(X = j)z_0^j, \end{aligned}$$

equality must be established for the real parts. Since $b > 1$ and $P(X = j) \geq 0$, $j\theta$ must be a multiple of 2π for all j with $P(X = j) > 0$. Thus, θ is a multiple of $2\pi/j$ for all j with $P(X = j) > 0$. Hence, θ is a multiple of $2\pi/gcd(j)$ where $gcd(j)$ is taken over all j with $P(X = j) > 0$. Therefore, θ is a multiple of 2π.□

Lemma 6.7 *If* $1 < b < 1/\phi(0)$ *then* ν *has a pole of order one at* $z = z_0$

Proof: It is sufficient to show that

$$\lim_{z \to z_0} (z - z_0)\nu(z) < \infty.$$

By L'Hôpital's Rule,

$$\lim_{z \to z_0} \frac{z - z_0}{1 - b\phi(z)} = -\frac{1}{b\phi'(z_0)}.$$

Since $\phi'(z_0) > 0$ from Lemma 6.2, the limit is bounded.□

Lemmas 6.5, 6.6 and 6.7 establish that the function ν is meromorphic for $|z| < R$ and analytic on $|z| = R$ where $z_0 < R < \inf\{|z| \quad b\phi(z) = 1, z \neq z_0\}$ In the region bounded by R, ν has a single pole of order one at $z = z_0$ [Lemma 6.7]. Therefore from [20, p. 81], the asymptotic behaviour of the coefficients of ν is given by

$$EN(k) = [z^k]\nu(z) = -\frac{Res(\nu; z_0)}{z_0^{k+1}} + O(R^{-k}) \tag{6.2}$$

where

$$\begin{aligned} Res(\nu; z_0) &= \lim_{z \to z_0} (z - z_0)\nu(z) \\ &= -\frac{1}{b\phi'(z_0)}. \end{aligned} \tag{6.3}$$

Substituting Equation 6.3 into Equation 6.2 yields

$$EN(k) = \frac{1}{b\phi'(z_0)z_0^{k+1}} + O(R^{-k})$$

Because $0 < z_0 < 1$ and $z_0 < R$,

$$EN(k) \in \Theta(c^{k+1})$$

where c is a constant greater than one. Similarly, it also follows that

$$EM(k) = \frac{1}{b\phi'(z_0)z_0^{k+1}(1 - z_0)} + O(R^{-k})$$

and

$$ET(k) = \frac{1}{b\phi'(z_0)z_0^{k+1}(1-z_0)^2} + O(R^{-k}).$$

The ratio of the expected number of nodes that are surely-expanded by A* to the expected number of nodes that are surely-expanded by IDA*, that is,

$$\frac{EM(k)}{ET(k)}$$

is approximately $1 - z_0$. Therefore, a constant fraction of the overall time complexity of IDA* is spent on all iterations leading up to the penultimate iteration. Assuming an integer probability distribution ϕ that satisfies Properties 6.2 and 6.3, Theorem 6.1 is stated directly.

Theorem 6.1 *Given a solution space tree with an uniform branching factor $b > 1$ and non-negative differential edge costs that are i.i.d. from an integer probability distribution where $b < 1/P(X = 0)$ then the expected number of nodes that are surely-expanded by IDA*, denoted $ET(k)$, is $\Theta(c^{k+1})$ where c is a constant greater than one*

Corollary 6.1 *Given a solution space tree with an uniform branching factor $b > 1$ and non-negative differential edge costs that are i.i.d from an integer probability distribution where $b < 1/P(X = 0)$ then IDA* is asymptotically optimal on average, in terms of the number of surely-expanded nodes, over the class of admissible best-first tree search algorithms.*

6.4 Expected Case Examples

In the following two subsections, average case examples are shown for the uniform probability distribution $U(1,2)$ and the geometric probability distribution with parameter p, $0 < p < 1$.

6.4.1 Uniform Probability Distribution

The generating function $\phi(z)$ for the uniform probability distribution $U(1,2)$ is

$$\sum_{j=1}^{2} \frac{z^j}{2}$$

which expands to

$$\frac{z+z^2}{2}.$$

Therefore,

$$\phi'(z) = \frac{1}{2} + z.$$

The real solution z_0 of $b\phi(z) = 1$ is equal to

$$\frac{-1+\sqrt{1+(8/b)}}{2}.$$

Therefore,

$$\phi'(z_0) = \frac{1}{2} + z_0.$$

Ignoring the lower order terms,

$$EN(k) = \frac{1}{b(\frac{1}{2}+z_0)z_0^{k+1}},$$

$$EM(k) = \frac{EN(k)}{1-z_0},$$

$$ET(k) = \frac{EN(k)}{(1-z_0)^2}$$

where $1 < b < \infty$.

6.4.2 Geometric Probability Distribution

The generating function $\phi(z)$ for the geometric probability distribution with parameter p where $0 < p < 1$ is

$$\sum_{j=0}^{\infty} p((1-p)z)^j.$$

Because $(1-p)z < 1$, $\phi(z)$ reduces to

$$\frac{p}{1-(1-p)z}.$$

Therefore,

$$\phi'(z) = \frac{p(1-p)}{(1-(1-p)z)^2}.$$

The real solution z_0 of $bg(z) = 1$ is equal to

$$\frac{1-pb}{1-p}.$$

Therefore,

$$\phi'(z_0) = \frac{(1-p)}{pb^2}.$$

Again, ignoring the lower order terms,

$$\begin{aligned} EN(k) &= \left(\frac{pb}{1-pb}\right)\left(\frac{1-p}{1-pb}\right)^k, \\ EM(k) &= \frac{EN(k)}{1-z_0} \\ &= EN(k)\left(\frac{1-p}{p(b-1)}\right), \\ ET(k) &= \frac{EN(k)}{(1-z_0)^2} \\ &= EN(k)\left(\frac{1-p}{p(b-1)}\right)^2 \end{aligned}$$

where $1 < b < \frac{1}{p}$.

6.5 Concluding Remarks

Since the number of iterations k performed by IDA* and the differential cost of the optimal solution path are equivalent, Theorem 6.2 is stated.

Theorem 6.2 *Given a solution space tree with an uniform branching factor $b > 1$ and non-negative differential edge costs that are i i d from an integer probability distribution where $b < 1/P(X = 0)$ then the expected number of nodes that are surely-expanded by A*, denoted $EM(k)$, is $\Theta(c^{k+1})$ where c is a constant greater than one and k is the differential cost of the optimal solution path.*

The expected case analysis of IDA* rests on the assumption that the differential edge costs are modeled as independent and identically distributed random integers. This assumption is supported by the following observations.

1. If the differential edge costs are real numbers with a maximum precision of 10^{-t}, $t \geq 1$, then each differential edge cost $\Delta f(n_i, n_j)$ is easily converted to the integral value

$$\Delta f(n_i, n_j)10^t$$

without loss of information Therefore, the model of computation is only limiting in the case of infinite precision.

2. Heuristic search is most often applied to problems for which there exists an exponential and, in some cases, an infinite number of possible solution paths. For instance, the traveling salesperson and other combinatorial optimization problems select an optimal solution among an exponential number of possibilities. Given an exponential number of goal nodes located at a distance $d + c \log d$, $c \geq 0$, from the start node s, it was

shown in [4] that the expected time complexity of A* is exponential in d regardless of the accuracy of the heuristic function employed. In the analyses presented here, an analogous result is derived. If a probability distribution satisfies Properties 6.2 and 6.3 then both A* and IDA* surely-expand on average an exponential number of nodes regardless of the probability distribution that models the differential edge costs.

3. The differential cost of an optimal solution path is equal to the difference between the cost of the optimal solution path and the cost of the start node s, that is, $f^*(s) - f(s)$. As the accuracy of heuristic function improves, the differential cost of the optimal solution path decreases. Hence as in [4], the expected number of nodes that are surely-expanded by A* and IDA* decreases exponentially.

Chapter 7

Binary Iterative-Deepening-A*

7.1 Description of BIDA*

Binary iterative-deepening-A* (BIDA*) is an admissible generalization of the IDA* algorithm. Designed specifically to redress the worst case phenomenon of expanding a few additional nodes over several iterations, the objectives of BIDA* are twofold.

1. To increase the number of additional but admissible nodes that are expanded on each iteration, and

2. To reduce the total number of iterations

In order to meet these objectives, the cost bound of each iteration of BIDA* is chosen as a point between:

1. A *lower bound* which is non-decreasing from one iteration to the next but remains less than or equal to the cost of the optimal solution path C^*, and

2. An *upper bound* which is non-increasing from one iteration to the next but remains greater than or equal to the cost of the optimal solution path C^*.

On each iteration, BIDA* performs a bounded depth-first search of the solution space $G = (s, \Upsilon, h)$. The lower bound of the initial iteration, denoted L_1, is set to the cost of the start node s, that is, $L_1 = f(s)$. The upper bound of the initial iteration, denoted U_1, is set to the cost of any solution path P from the start node s to a goal node q, that is, $U_1 = f(q)$ If the lower bound L_1 is equal the upper bound U_1 then the BIDA* algorithm terminates with the solution path P. Otherwise, the cost bound of the initial iteration is equal to

$$(1 - \omega)L_1 + \omega U_1$$

where $0 < \omega < 1$. For the initial iteration and each successive iteration $i \geq 1$, a depth-first search is performed until either one of two conditions is met:

1. A goal node is selected for expansion, or

2. The f-values of all expandable nodes is greater than the cost bound of iteration i, denoted C_i

If a goal node q is selected for expansion then the upper bound of iteration $i + 1$, denoted U_{i+1}, is set to cost of the solution path P from the start node s to q; otherwise, U_{i+1} remains equal to U_i. If, on the other hand, a goal node is not selected for expansion then the lower bound of iteration $i + 1$, denoted L_{i+1}, is set to the minimum f-value among all nodes that were generated on iteration i and that exceeded the cost bound C_i; otherwise, L_{i+1} remains equal to L_i. If the upper bound U_{i+1} is equal to the lower bound L_{i+1}, the BIDA* algorithm terminates with the most recent solution path P. If the lower bound

remains less than the upper bound then a depth-first search of the solution space is repeated with a cost bound C_{i+1} equal to

$$(1 - \omega)L_{i+1} + \omega U_{i+1}$$

where $0 < \omega < 1$. The BIDA* algorithm is outlined in Figure 7.1.

An example of a BIDA* search is presented in Figure 7.2. The solution space is a complete, directed binary tree of depth 3. Each node A through P is labeled with its f-value and the nodes H, I and P are designated as the goal nodes. It is assumed that ω is 0.5 and therefore, the cost bound C_i of each iteration i is equal to

$$\lfloor 0.5(L_i + U_i) \rfloor$$

The lower bound of the initial iteration set to the cost of the start node s, that is, $L_1 = 1$. Without loss of generality, the upper bound of the initial iteration is arbitrarily set to the cost of the leftmost solution path. Therefore, U_1 is equal to 5 and the solution path $(ABDH)$ is assigned to P. The cost bound C_1 of the initial iteration is equal to

$$\lfloor 0.5(L_1 + U_1) \rfloor = \lfloor 0.5(1 + 5) \rfloor = 3$$

Nodes A through G are expanded on the initial iteration. Since a goal node is not selected for expansion, the lower bound of the second iteration is raised to the cost of the minimum f-value among expandable nodes that exceeded the cost bound of the initial iteration. Therefore, L_2 is equal to 4. Because the upper bound remains unchanged, the cost bound C_2 of the second iteration is equal to

$$\lfloor 0.5(L_2 + U_2) \rfloor = \lfloor 0.5(4 + 5) \rfloor = 4.$$

Only nodes A, B and D are expanded on the second iteration before the goal node I is selected for expansion. Since a goal node is selected for expansion,

```
Comments:

     L  is the lower bound of the current iteration.
     U  is the upper bound of the current iteration.
     C  is the cost bound of the current iteration.
     M  is the minimum f-value among generated nodes that
        exceeds C.

Step 0: Set L equal to f(s), the cost of the start node s.
        Set U equal to the cost of any solution path P.
Step 1: If L = U then
           Exit successfully with the solution path P
        Else
           Push s onto an empty STACK of expandable nodes.
           Set C equal to (1-w)L + wU where 0<w<1.
           Set M equal to MAX.
Step 2: If STACK is empty then
           If M remains equal to MAX then
              Exit with failure; no solution exists
           Else
              Set L equal to M.
              Go to Step 1 and proceed with the next iteration.
Step 3: Assign the top node of STACK to p.
Step 4: If p is a goal node then
           Set U equal to f(p).
           Set P equal to the solution path that is found.
           Go to Step 1 and proceed with the next iteration.
Step 5: Generate the next successor n of p.
Step 6: If node p has no further successors then
           Pop p from the STACK
        Else
           a) Calculate f(n).
           b) If f(n) <= C then
                 Push n onto STACK
                 Assign the newly-computed f(n) to n.
           c) If C < f(n) < M then
                 Lower M to f(n).
Step 7: Go to Step 2.
```

Figure 7.1: The Binary IDA* Algorithm

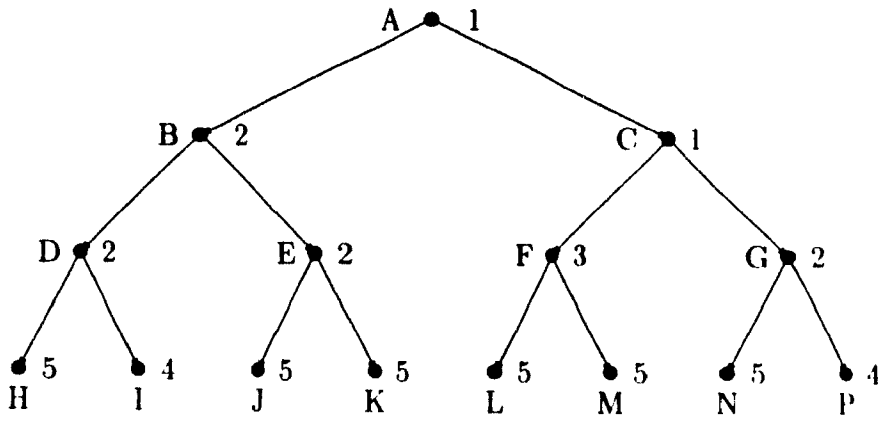

Figure 7.2: Example of a BIDA* Search

the upper bound of the third iteration is lowered to 4 and the solution path $(ABDI)$ is assigned to P. Because the lower bound remains unchanged, the lower and upper bounds of the third iteration are equal[1]. Therefore, the BIDA* algorithm terminates with solution path P.

7.1.1 Admissibility

Theorem 7.1 *Given a solution space tree* $G = (s, \Upsilon, h | h \leq h^*)$, *BIDA** *is admissible.*

Proof: It is sufficient to show that the lower bound of BIDA* will eventually equal but never exceed the cost of the optimal solution path C^* and that the upper bound will eventually equal but never fall below C^*. Therefore, the lower bound can only equal the upper bound at C^* which implies admissibility.

[1]On the third iteration, no depth-first search is actually performed

The lower bound of the initial iteration is equal to the cost of the start node s. Since $f(s) \leq f^*(s) = C^*$, the lower bound is less than or equal to C^*. The upper bound of the initial iteration is equal to the cost of any solution path. Since the cost of any solution path is greater than or equal to the cost of the optimal solution path, the upper bound is greater than or equal to C^*. If the initial lower and upper bounds are equal then an optimal solution path is immediately found; otherwise, the cost bound of the initial iteration, denoted C_1, is chosen between the initial lower and upper bounds If an optimal solution path is not found on iteration $i \geq 1$ then a bounded depth-first search is performed until either a goal node is selected for expansion or the f-values of all expandable nodes is greater than C_i. If a goal node is selected for expansion then the upper bound is set to the cost of the solution path that is found Clearly, the upper bound remains greater than or equal to C^*. Since the cost bound of the subsequent iteration is less than the new upper bound, each solution path is found at most once Hence, the cost of each solution path is less than the cost of the previous solution path that is found and the upper bound is less than the previous upper bound If a goal node is not selected for expansion then the lower bound of the subsequent iteration is set to the minimum f-value among expandable nodes that exceeded the cost bound of iteration i. Hence, the lower bound remains less than or equal to C^* but greater than the previous lower bound. In either case, the interval between the lower and upper bound is reduced from one iteration to the next Since the cost of each directed edge in G is by definition greater than some positive constant δ, there exists a finite number of nodes whose f-values fall within the interval between the initial lower and upper bounds. Because each new lower or upper bound is equal to the cost of a node whose f-value falls within the initial interval and because the interval between the lower and upper bound

strictly decreases from one iteration to the next, the number of iterations required for either the lower bound or upper bound to reach C^* is finite

If the lower bound equals C^* before the upper bound then the cost bound of each subsequent iteration remains greater than C^* until an optimal solution path is found. At this point, the upper bound is set to the cost of the optimal solution path and the search terminates. If the upper bound equals C^* before the lower bound then the cost bound of each subsequent iteration remains less than C^* until the lower bound is equal to C^*. At this point, the search terminates and returns the most recent solution path that is found Since the cost of the most recent solution path is equal to the current upper bound, that is C^*, then an optimal solution path is found □

7.1.2 Time Complexity

Lemma 7.1 *Given a solution space tree* $G = (s, \Upsilon, h|h \leq h^*)$ *and* $0 < \omega < 1$, *BIDA* performs at most* $\lceil \log_{\frac{1}{1-\varpi}}((U_1 - L_1)10^t + 1) \rceil$ *iterations where* ϖ *is equal to* $\min(\omega, 1-\omega)$ *and* 10^{-t} *is equal to the maximum precision of the edge costs*

Proof: By multiplying the edge costs by 10^t, the lower, upper and cost bounds of each iteration are treated as integral values without loss of information By definition, the cost bound C_i of iteration i is equal to

$$(1-\omega)L_i + \omega U_i$$

Since either $L_{i+1} > C_i$ or $U_{i+1} \leq C_i$, the interval between L_i and U_i is reduced by at least a factor of $\varpi = \min(\omega, 1-\omega)$. By reducing the interval from one iteration to the next by at least a factor of ϖ until L_i is equal to U_i, the maximum number of iterations performed by BIDA* given an initial interval

$[L_1, U_1]$ is equal to

$$\lceil \log_{\frac{1}{1-\omega}}((U_1 - L_1)10^t + 1) \rceil . \square$$

Corollary 7.1 *Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$, the maximum number of iterations performed by BIDA* is minimized at $\omega = 0.5$.*

Corollary 7.2 *Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$, IDA* performs at most $(C^* - C_1 + 1)10^t$ iterations where C_1 and C^* are the cost bounds of the initial and final iterations, and 10^{-t} is equal to the maximum precision of the edge costs*

Corollary 7.3 *If the initial upper bound of an admissible solution space tree is at most a polynomial function of the cost of the optimal solution path then BIDA* performs asymptotically fewer iterations than IDA* in the worst case.*

The lure of fewer iterations, however, does not immediately imply that the time complexity of BIDA* is also less than the time complexity of IDA*. Because the cost bound of each iteration of BIDA* is selected as an arbitrary point between the lower and upper bounds of that iteration, the cost bound is not constrained to be less than or equal to the cost of the optimal solution path. With the potential of expanding several inadmissible nodes on those iterations whose cost bounds are greater than C^*, BIDA* may expand a far greater number of nodes than IDA* notwithstanding the reduction in the number of iterations. However, this computational risk is mitigated in part by one factor: If the cost bound of an iteration is greater than C^* then BIDA* performs a bounded depth-first search *only* until a solution path is found. Therefore, BIDA* does not necessarily perform an exhaustive search of all paths along which each node has an f-value that is less than or equal to the cost bound of the iteration.

Iteration	Cost Bound	Number of Nodes Expanded
1	1	2
2	2	6
3	3	7
4	4	4*
* Includes the goal node I		

(a) IDA* Algorithm

Iteration	Lower Bound	Upper Bound	Cost Bound	Number of Nodes Expanded
1	1	5	3	7
2	4	5	4	4*
3	4	4	-	-
* Includes the goal node I				

(b) BIDA* Algorithm

Table 7.1: Performance of IDA* and BIDA* on Figure 7.2

Given the solution space tree in Figure 7.2, the total number of nodes that are expanded by IDA* and BIDA* on each iteration is summarized in Tables 7.1 (a) and (b) respectively. The IDA* algorithm expands a total of 19 nodes over 4 iterations and BIDA* expands a total of only 11 nodes over 2 iterations. However, if the cost of the leftmost path is increased to 9 and node *I* is no longer designated as a goal node then the total number of nodes that are expanded by IDA* and BIDA* is summarized in Tables 7.2 (a) and (b) respectively. In this case, IDA* expands a total of 24 nodes over 4 iterations but BIDA* expands a total of 28 nodes over 3 iterations It is therefore difficult to state conclusively that on average IDA* is superior to BIDA* (or vice versa) without drawing on empirical evidence.

Iteration	Cost Bound	Number of Nodes Expanded
1	1	2
2	2	6
3	3	7
4	4	9*
* Includes the goal node P		

(a) IDA* Algorithm

Iteration	Lower Bound	Upper Bound	Cost Bound	Number of Nodes Expanded
1	1	9	5	15*
2	1	4	2	6
3	3	4	3	7
4	4	4	-	-
* Includes the goal node P				

(b) BIDA* Algorithm

Table 7.2: Performance of IDA* and BIDA* on Figure 7.2 (modified)

7.2 Empirical Results

Empirical tests that compare the average case performance between the IDA* and BIDA* algorithms are carried out with respect to the traveling salesperson problem (TSP) on the basis of the following observations:

1. In Chapter 5, a worst case example of IDA* has been shown on an instance of the asymmetric traveling salesperson problem (ATSP). Similar results noted in [24] also cite the non-optimal performance of IDA* on instances of the TSP.

2. Unlike other common applications such as the 15-Puzzle and the vertex cover problems, the edge costs of the TSP are not necessarily equal to

one. Hence, the edge costs may be modeled with arbitrary precision and magnitude.

3. The solution space G_m of an m-city TSP satisfies the following two properties.

   Property 7.1 *Every terminal node in G_m is located at depth m from the start node s.*

   Property 7.2 *Every terminal node is a goal node. Therefore, every path in G_m leads to a goal node.*

   Properties 7.1 and 7.2 ensure that the depth of search is bounded and that a goal node is returned whenever the maximum depth is reached. If the cost bound of an iteration is greater than C^* then BIDA* explores a single path at a time until a solution path P is found. Hence, only those nodes on or before the solution path P are selected for expansion. Since the length of each path is at most m and every path in G_m leads to a goal node, the above properties above help to mitigate the computational risk of potentially expanding several inadmissible nodes.

7.2.1 Euclidean Traveling Salesperson Problem

In light of the above observations, the Euclidean traveling salesperson problem (ETSP) defined below is chosen as a representative problem from the class of traveling salesperson problems.

Definition 7.1 *Given a positive adjacency matrix (c_{ij}) where each element c_{ij} represents the Euclidean distance from city i to city j, the Euclidean traveling salesperson problem (ETSP) is to find the shortest tour that begins at an*

arbitrary city, visits each other city exactly once and returns to the starting city.

Each instance of an m-city ETSP is generated by randomly selecting m points in the unit square $[0,1]^2$. Each point $(x_i, y_i) \in [0,1]^2$ represents the position of city i. The Euclidean distance c_{ij} between city i and city j is calculated straightforwardly as

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

for each pair (i,j). Therefore, an instance of the ETSP is characterized by two parameters (m,t) where,

1. m represents the number of cities in the tour, and
2. t represents the maximum precision, evaluated as 10^{-t}, of the Euclidean distances between cities.

For example, the parameters (9,5) define a 9-city ETSP where 10^{-5} is the maximum precision of the edge costs.

7.2.2 Testing and Analysis

Forty random instances of the m-city ETSP are generated for each m, $5 \leq m \leq 10$. Each instance is solved for a maximum precision of 10^{-t}, $1 \leq t \leq 6$, using the A*, IDA* and BIDA* ($\omega = 0.5$) algorithms. The time complexities of both IDA* and BIDA* are measured in terms of the total number of nodes that are selected for expansion on each iteration leading up to and including the penultimate iteration. Since the time complexity of A* is equal to the number of nodes that are surely-expanded by IDA* on its penultimate iteration and since IDA* is nearly-equivalent[2] to BIDA* for $\omega = 0$, the three algorithms are

[2]The differences are reconciled in Section A.1.

Tour Size	Maximum Precision, t					
m	1	2	3	4	5	6
5	1.89	3 71	4.06	4.06	4 06	4.06
6	2.20	7.44	9.20	9.48	9.50	9.50
7	2.06	9.66	16.85	18.69	18.88	18.90
8	2.01	12.84	37.90	46.51	47.77	47.92
9	2.02	14.42	69.53	109.18	114 49	115 29
10	1.98	15.12	116 13	607.52	1126.52	1209.80

Table 7.3: Average Time Complexities of IDA* to A*

implemented as a single standard Pascal program (Appendix A) The average time complexities of A*, IDA* and BIDA* are calculated based on the forty random instances and the following ratios are recorded for each parameter (m, t):

Table 7.3: The ratio of the average time complexity of IDA* to the average time complexity of A*.

Table 7.4 (a): The ratio of the average number of iterations performed by BIDA* to the average number of iterations performed by IDA*

Table 7.4 (b): The ratio of the average time complexity of BIDA* to the average time complexity of IDA*.

Table 7.5: The ratio of the average time complexity of BIDA* to the average time complexity of A*.

Each entry in the above tables represents a ratio of an average performance measure between, say, Algorithm A and Algorithm B. As the tour size and the precision of the edge costs increase, three scenarios are noted

Tour Size	Maximum Precision, t					
m	1	2	3	4	5	6
5	0.827	0.595	0.576	0.580	0.580	0.580
6	0.810	0.424	0.364	0.357	0.357	0.357
7	0.840	0.353	0.273	0.261	0.260	0.261
8	0.841	0.240	0.151	0.137	0.134	0.134
9	0.888	0.217	0.106	0 084	0.082	0.082
10	0.861	0.199	0.075	0.046	0.037	0 035

(a) Average Number of Iterations of BIDA* to IDA*

Tour Size	Maximum Precision, t					
m	1	2	3	4	5	6
5	1.079	0.673	0.652	0.658	0 658	0.658
6	1.022	0.454	0.369	0 358	0.357	0.357
7	1.063	0.359	0.226	0.203	0.201	0.201
8	1 088	0.275	0.131	0.115	0.110	0.109
9	1.162	0.234	0.080	0.050	0 019	0.019
10	0.959	0.218	0.049	0.013	0.008	0 007

(b) Average Time Complexities of BIDA* to IDA*

Table 7.4: Average Performance Ratios of BIDA* to IDA*

1. If the ratio remains constant then the performance of A is optimal with respect to the performance of B.

2. If the ratio increases then the performance of A is non-optimal with respect to the performance of B.

3. If the ratio decreases then the performance of B is non-optimal with respect to the performance of A.

In Table 7.3, the non-optimal performance of IDA* on the ETSP is clear. As both the tour size and the precision of the edge costs increase, the ratio

Tour Size	Maximum Precision, t					
m	1	2	3	4	5	6
5	2.04	2.50	2.65	2.67	2.67	2.67
6	2.25	3.38	3.39	3.40	3.40	3.40
7	2.18	3.47	3.81	3.79	3.79	3.80
8	2.19	3.53	4.98	5.33	5.24	5.24
9	2.34	3.37	5.53	5.45	5.61	5.61
10	1.90	3.29	5.72	8.16	8.84	8.77

Table 7.5: Average Time Complexities of BIDA* to A*

of the average time complexity of IDA* to the average time complexity of A* departs quite dramatically from an optimal constant ratio. As expected and as shown in Table 7.4 (a), BIDA* performs on average fewer iterations than IDA* in every instance. Furthermore, the ratio of the average number of iterations performed by BIDA* to those performed by IDA* is decreasing as both the tour size and the precision of the edge costs increase. Hence, the performance of IDA* is non-optimal with respect to BIDA* in terms of the average number of iterations. The reduction in the number of iterations also yields an almost proportional decrease in the average time complexity of BIDA* as shown in Table 7.4 (b). This suggests that the number of inadmissible nodes that are expanded by BIDA* does not significantly impede its performance. Therefore, the average time complexity of IDA* is again non-optimal with respect to the average time complexity of BIDA*. Although the average time complexity ratio between BIDA* and A* continues to increase as both the tour size and the precision of the edge costs increase, the ratio increase in Table 7.5 is comparatively slight. It is therefore encouraging that the reduction in both the number of iterations and the time complexity of BIDA* over IDA* yields a near-optimal performance by BIDA* with respect to A*.

7.3 Comparison of BIDA* with IDA*_CR

Recently, Sarkar et al. developed an admissible version of IDA*, called IDA*_CR, that also redresses the worst case phenomenon of an IDA* search [46]. The IDA*_CR algorithm differs from BIDA* in two key respects:

1. IDA*_CR performs a depth-first branch and bound search on each iteration as opposed to a strictly depth-first search.

2. The cost bound of each iteration of IDA*_CR is chosen such that the number of additional nodes grows exponentially from one iteration to the next, that is, the heuristic branching factor b_h is constant and greater than one.

To guarantee that at least b_h^i additional nodes are expanded between the i and $(i+1)^{st}$ iterations, IDA*_CR uses a set of buckets indexed $1, 2, \ldots, p$ to group the f-values which exceeded the cost bound C_i of the current iteration. Each bucket, denoted B_j, is associated with a mutually-exclusive range of values $[r_j, r_{j+1}]$ where $r_j < r_{j+1}$ for all j, $1 \leq j \leq p-1$. For each node n whose f-value exceeds C_i, the index of bucket B_j is increased by one where

$$r_j < f(n) \leq r_{j+1}.$$

Therefore, the cost bound of iteration $i+1$ is set to the minimum r_{j+1} where the sum of the indices of buckets B_i through B_j exceeds b_h^i. At this point, two important observations are made.

1. An appropriate balance among a) the heuristic branching factor b_h, b) the number of buckets and c) the range of values associated with each bucket must be established. For an inappropriate choice, only the f-values a few generated nodes may fall within the range of values associated with the

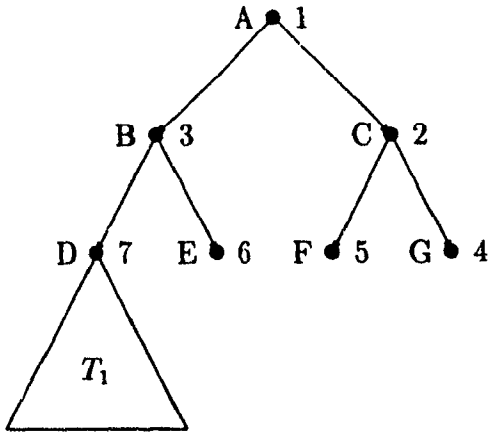

Figure 7.3: Example of an IDA*_CR Search

buckets. In this case, IDA*_CR may not be able to sustain the exponential growth rate from one iteration to the next. Even the choice of an inappropriate b_h alone may lead to a poor performance as shown in Figure 7.3.

Each node A through G is labeled with its f-value. Nodes A and G are designated as the start and goal nodes respectively. If a heuristic branching factor b_h is chosen as 2 then at least 2^0, 2^1, 2^2, ...additional nodes must be expanded on iterations 1, 2, 3, ... Therefore, the cost bounds of iterations 1, 2, and 3 must equal 1, 3 and 7 respectively. On the third and final iteration, IDA*_CR performs a depth-first branch and bound search until the optimal solution path from node A to node G is found. However, because the cost bound of the third iteration is equal to 7, IDA*_CR may expand an arbitrarily large number of inadmissible nodes in the subtree T_1 rooted at node D.

2. The cost bound of the final iteration of IDA*_CR may exceed the cost of the optimal solution path. Therefore, like the BIDA* algorithm, IDA*_CR potentially expands several inadmissible nodes before an optimal solution path is found. Since both A* and IDA* do not expand any inadmissible nodes for $h \leq h^*$, there is again no common measure of time complexity among A*, IDA* and IDA*_CR that includes the number of nodes that are expanded on the final iteration. Hence, the following claim in [46, p 213] is somewhat misleading:

> IDA*_CR expands $O(N)$ nodes where N is the number of nodes that are expanded by A*.

However, because IDA*_CR does not expand any inadmissible nodes on each iteration leading up to and including the penultimate iteration, Theorem 7.2 is stated directly.

Theorem 7.2 *Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$ with a constant heuristic branching factor greater than one, IDA*_CR is asymptotically optimal, in terms of the number of nodes that are surely-expanded by A*, over the class of admissible best-first tree search algorithms.*

Unfortunately, a similar claim cannot be made for BIDA*.

7.4 Concluding Remarks

Throughout the development and testing of BIDA*, the initial upper bound was arbitrarily set to the cost of any solution path. However, the initial upper bound is often better established using an approximation algorithm [14] rather than a random solution path. This approach was incorporated into the

MSBB algorithm of [45]. For example, Christofides' TSP algorithm [7] finds an approximate solution that is at most 1.5 times the cost of the optimal solution. Setting U_1 equal to the cost of the approximate solution, the initial lower bound L_1 may also be set to

$$\max(U_1/1.5, f(s))$$

where $f(s)$ is the cost of the start node s.

Chapter 7 has established two important properties of a BIDA* search on a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$:

1 BIDA* is admissible.

2. If the initial upper bound of an admissible solution space tree is at most a polynomial function of the cost of the optimal solution path then BIDA* performs asymptotically fewer iterations than IDA* in the worst case.

Unfortunately, the reduction in the number of iterations comes at the expense of potentially expanding a large number of inadmissible nodes. Because, in part, the TSP satisfies Properties 7.1 and 7.2 stated earlier, BIDA* is shown to be a significant improvement over IDA* as both the tour size and the precision of the edge costs increase for the ETSP. However, it remains:

1. To expand the empirical scope to other combinatorial problems and to larger instances of the traveling salesperson problem.

2. To support the empirical work with theoretical justification. For instance, the expected case behaviour of BIDA* may be derived with respect to a probabilistic model of computation that distributes the costs (depths) of the goal nodes over a solution space tree. Such an analysis would help answer an important question: What conditions must the

distribution function satisfy in order to ensure the optimal performance of BIDA* with respect to IDA* and better still to A*?

3. To generalize the above analyses for all ω, $0 < \omega < 1$.

Chapter 8

Final Remarks

Iterative-deepening-A* is an admissible heuristic search algorithm which is optimal with respect to space complexity and the cost of solution found over the class of admissible best-first tree search algorithms. However, the optimality of IDA*, as measured against the optimal time complexity of the A* algorithm, is subject to a number of conditions. These basic results and conditions are summarized below. Given a solution space tree $G = (s, \Upsilon, h | h \leq h^*)$ where A* surely-expands $M(k)$ nodes on G:

1. IDA* is asymptotically optimal over the class of admissible best-first tree search algorithms, that is, $T(k) \in O(M(k))$ if the effective or heuristic branching factor is constant and greater than one.

2. In the worst case, IDA* surely-expands $(M^2(k) + M(k))/2$ nodes over $M(k)$ iterations. The worst case performance of IDA* is a special case of the unit heuristic branching factor when IDA* expands exactly one additional node per iteration.

3. The worst case performance of IDA* is realized if and only if the evaluation function assigns an f-value to each non-goal node such that the

conditions of uniqueness and monotonicity are satisfied

4. In the expected case, IDA* is asymptotically optimal over the class of admissible best first tree search algorithms if the differential edge costs are independently and identically distributed from any integer probability distribution that satisfies Properties 6.2 and 6.3

The time complexity spectrum of IDA* is defined between its asymptotic optimal and worst case performance. Because IDA*, like the A* algorithm, is applied to a wide variety of problems, the computational efficiency of IDA* on a given problem may fall anywhere within this time complexity spectrum. However, the efficiency of search depends on a number of interrelated factors, most notably:

1. The structure of the solution space graph,

2. The behaviour of the evaluation function with respect to the conditions of uniqueness and monotonicity, and

3. The precision of the f-values assigned to each node.

If the solution space graph satisfies the property of acyclicy, no node is reselected for expansion by either the IDA* or A* algorithm. In Chapter 3, it was shown by example that the time complexity of IDA* grows exponentially over A* when the number of paths from the start node s to any other node grows exponentially with depth. Although the property of acyclicy supplants the weaker requirement of monotonicity, this result is analogous to the exponential growth in the time complexity of A* whenever the property of monotonicity is relaxed.

As the precision of the evaluation function increases, the number of possible f-values that are assignable to each node increases as well. Therefore,

the likelihood that two or more nodes will share the same f-value decreases Consequently.

1. The number of additional nodes that are expanded on each iteration is reduced, and

2. The performance of IDA* is skewed toward the worst case end of the time complexity spectrum.

This phenomenon was reflected by empirical results on the Euclidean traveling salesperson problem (ETSP) in Chapter 7. By randomly generating points in the unit square and modeling the precision of the edge costs, the performance of IDA* degraded significantly as the precision of the edge costs increased for a given number of cities. To redress the problem of expanding only a few additional nodes over several iterations, the BIDA* algorithm was developed and tested with respect to the ETSP. It was shown on a small test bed that BIDA* was much less sensitive to an increase in both the tour size and the precision of the edge costs.

The IDA* algorithm is an important search technique. It offers the attractive possibility of asymptotic optimality along the dimensions of time complexity, space complexity and the cost of solution found. Unfortunately, the sheer size of the solution spaces of most combinatorial problems often overwhelms even the best heuristic knowledge. It therefore remains to understand how the performance of IDA* is improved (or degraded) with respect to time com plexity and the cost of solution found for inadmissible heuristics. Secondly, empirical and theoretical work must continue into the development and testing of parallel versions of IDA* such as [44]. The design and analysis of parallel search algorithms introduces many difficult problems concerned with task division and interprocessor communication. However, a better understanding of

the performance of the IDA* algorithm itself is a vital prerequisite toward a better understanding of its inadmissible and parallel derivatives

Bibliography

[1] Aho, A.V., Hopcroft, J.E. and Ullman, J.D. (1974) *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA

[2] Bagchi, A. and Mahanti, A. (1983) Search algorithms under different kinds of heuristics - A comparative study, *Journal of the ACM 30(1)*, pp. 1-21.

[3] Bagchi, A. and Mahanti, A (1985) Three approaches to heuristic search in networks, *Journal of the ACM 32(1)*, pp. 1-27.

[4] Bagchi, A and Sen, A.K. (1988) Average-case analysis of heuristic search in tree-like networks, in *Search in Artificial Intelligence* (L Kanal and V. Kumar, editors), Springer Verlag, New York, N Y., pp. 131-165.

[5] Barr, A. and Feigenbaum, E.A., editors (1981) *Handbook of Artificial Intelligence*, Morgan Kaufmann, Los Altos, CA.

[6] Chakrabarti, P.P., Ghose, S., Acharya, A. and DeSarkar, S.C. (1989) Heuristic search in restricted memory, *Artificial Intelligence 41(2)*, pp. 197-221.

[7] Christofides, N. (1976) Worst case analysis of a new heuristic for the traveling salesman problem, Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA.

[8] Dechter, R. and Pearl, J. (1985) Generalized best-first search strategies and the optimality of A*, *Journal of the ACM 32(3)*, pp. 505-536.

[9] Dechter, R. and Pearl, J. (1988) The optimality of A*, in *Search in Artificial Intelligence* (L. Kanal and V. Kumar, editors), Springer Verlag, New York, N.Y., pp 166-199.

[10] Dijkstra, E.W. (1959) A note on two problems in connection with graphs, *Numerische Mathematik 1*, pp. 269-271.

[11] Doran, J. and Michie, D. (1966) Experiments with the graph traverser program, *Proceedings of the Royal Society of London 294(A)*, pp. 235 259.

[12] Dreyfus, S.E. and Law, A M. (1977) *The Art and Theory of Dynamic Programming*, Academic Press, New York, N.Y

[13] Ernst, G.W. and Newell, A (1969) *GPS A Case Study in Generality and Problem Solving*, Academic Press, New York, N Y.

[14] Garey, M.R and Johnson, D.S (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H Freeman and Company, New York, N.Y

[15] Gaschnig, J. (1979) Performance measurement and analysis of certain search algorithms, Ph.D. Dissertation, Technical Report CMU-CS-79-124, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.

[16] Gelperin, D (1977) On the optimality of A*, *Artificial Intelligence 8(1)*, pp. 69–76.

[17] Harris, L.R. (1974) The heuristic search under conditions of error, *Artificial Intelligence 5(3)*, pp. 217-234.

[18] Hart, P E., Nilsson, N.J. and Raphael, B. (1968) A formal basis for the heuristic determination of minimum cost paths, *IEEE Trans on Systems Science and Cybernetics 4*, pp. 100-107.

[19] Hart, P.E., Nilsson, N.J. and Raphael, B. (1972) Correction to "A formal basis for the heuristic determination of minimum cost paths", *SIGART Newsletter 37*, pp. 28–29.

[20] Hofri, M. (1987) *Probabilistic Analysis of Algorithms*, Springer Verlag, New York, N.Y.

[21] Huyn, N., Dechter, R. and Pearl, J. (1980) Probabilistic analysis of the complexity of A*, *Artificial Intelligence 15(3)*, pp. 241-254.

[22] Ibaraki, T. (1978) m-depth search in branch-and-bound algorithms, *International Journal of Computer Information Sciences 7(4)*, pp. 315-343.

[23] Korf, R.E. (1985) Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence 27(1)*, pp. 97-109.

[24] Korf, R.E. (1988) Optimal path-finding algorithms, in *Search in Artificial Intelligence* (L. Kanal and V. Kumar, editors), Springer Verlag, New York, N.Y., pp. 223–267.

[25] Levy, D. and Newborn M. (1991) *How Computers Play Chess,* Computer Science Press, New York, N.Y.

[26] Lin, S. (1965) Computer solutions of the traveling salesman problem, *Bell Systems Tech. Journal 44(10)*, pp. 2245–2269.

[27] Mahanti, A. and Ray, K. (1988) Network search algorithms with modifiable heuristics, in *Search in Artificial Intelligence* (L. Kanal and V. Kumar, editors), Springer Verlag, New York, N.Y., pp. 200–222.

[28] Martelli, A. (1977) On the complexity of admissible search algorithms, *Artificial Intelligence 8(1)*, pp. 1–13.

[29] Mérô, L. (1984) A heuristic search algorithm with modifiable estimate, *Artificial Intelligence 23(1)*, pp. 13–27.

[30] Newborn, M.M. (1976) Reconsideration of a theorem on admissible ordered search algorithms, *Proceedings of the Annual Conference of the ACM*, pp. 535–538.

[31] Newborn, M. (1989) *The Great Theorem Prover*, Newborn Software, Westmount, Québec, Canada.

[32] Newell, A. and Simon, H.A. (1972) *Human Problem Solving*, Prentice Hall, Englewood, N.J.

[33] Nilsson, N.J. (1971) *Problem Solving Methods in Artificial Intelligence*, McGraw Hill, New York, N.Y.

[34] Nilsson, N.J. (1980) *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA.

[35] Patrick, B.G., Almulla, M. and Newborn, M.M. (1992) An upper bound on the time complexity of iterative-deepening-A*, *Annals of Mathematics and Artificial Intelligence 5*, J.C. Baltzer, Basel, Switzerland (to appear)

[36] Pearl, J. and Kim, J.H. (1982) Studies in semi-admissible heuristics, *IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-4(4)*, pp. 392–399.

[37] Pearl, J. (1983) Knowledge versus search: A quantitative analysis using A*, *Artificial Intelligence 20(1)*, pp. 1–13.

[38] Pearl, J. (1984a) *Intelligent Search Strategies for Computer Problem Solving*, Addison Wesley, Menlo Park, CA.

[39] Pearl, J. (1984b) Some recent results in heuristic search theory, *IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-6(1)*, pp. 1–12.

[40] Pohl, I. (1970a) First results on the effect of error in heuristic search, in *Machine Intelligence 5* (B. Meltzer and D. Michie, editors), American Elsevier, New York, N.Y., pp. 219–236.

[41] Pohl, I. (1970b) Heuristic search viewed as path finding in a graph, *Artificial Intelligence 1(3)*, pp. 193–204.

[42] Pohl, I. (1973) The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving, *Proceedings of the IJCAI 3*, Stanford, CA, pp 20-23.

[43] Pohl, I. (1977) Practical and theoretical considerations in heuristic search algorithms, in *Machine Intelligence 8* (E.W. Elcock and D Michie, editors), Wiley, New York, N.Y., pp. 55–72.

[44] Rao, V.N, Kumar, V. and Ramesh, K. (1987) A parallel implementation of iterative-deepening-A*, *Proceedings of the National Conference on Artificial Intelligence (AAAI 1987)*, Seattle, Washington, pp. 178-182.

[45] Sarkar, U.K., Chakrabarti, P.P., Ghose, S. and De Sarkar, S.C. (1991) Multiple stack branch and bound, *Information Processing Letters 37(1)*, pp. 43–48.

[46] Sarkar, U.K., Chakrabarti, P.P., Ghose, S. and De Sarkar, S.C. (1991) Reducing reexpansions in iterative-deepening search by controlling cutoff bounds, *Artificial Intelligence 50(2)*, pp. 207–221.

[47] Sen, A.K. and Bagchi, A. (1989) Fast recursive formulations for best-first search that allow controlled use of memory, *Proceedings of the IJCAI 11*, Detroit, Michigan, pp. 297–302

[48] Slate, D.J. and Atkin, L.R. (1977) CHESS 4.5 - The Northwestern University Chess Program, in *Chess Skill in Man and Machine* (P. Frey, editor), Springer Verlag, New York, N.Y., pp. 82–118.

[49] Stickel, M.E. and Tyson, W.M. (1985) An analysis of consecutively bounded depth-first search with applications in automated deduction, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Los Angeles, CA, pp. 1073–1075.

[50] Vanderbrug, G. (1976) Problem representations and formal properties of heuristic search, *Information Sciences 11(4)*, pp. 279–307.

[51] Winston, P.H. (1984) *Artificial Intelligence*, Addison Wesley, Reading, MA.

Appendix A

The ETSP Program

A.1 Introductory Comments

The ETSP program uses a similar but not identical version of the BIDA* algorithm given in Figure 7.1. It differs in two key regards:

1. At least one iteration is performed by the ETSP program, and
2. The lower bound is updated *after* the test for equality with the upper bound.

Therefore, the ETSP program performs an additional iteration when the lower bound is raised and made equal to the upper bound, that is, when the lower bound is equal to the cost of the optimal solution. Since the cost bound of the subsequent iteration is also equal to the cost of the optimal solution, the same set of nodes are expanded on the final iteration for all ω, $0 \leq \omega < 1$. Importantly, the number of nodes that are surely-expanded by IDA* on the ETSP is equal to the number of nodes that are selected for expansion by the ETSP program when $\omega = 0$.

On the other hand, the ETSP program terminates immediately when the upper bound is lowered and made equal to the lower bound, that is, when the upper bound is equal to the cost of the optimal solution. Therefore, the cost bound of the final iteration is *not* equal to C^*. In this case, a different set of nodes may be expanded on the final iteration for each ω, $0 \leq \omega < 1$. It therefore remains as difficult to provide a common measure of comparison on the final iteration between BIDA* and IDA* as to provide a common measure of comparison between A* and the final iteration of IDA*.

A.2 Pascal Implementation

```
(*****************************************************************
*                                                                *
*  Title:      Euclidean Traveling Salesperson Problem           *
*                                                                *
*  Author :    Brian G. Patrick                                  *
*                                                                *
*  Purpose :   To solve a random instance of the m-city ETSP     *
*              using the BIDA* algorithm where 0 <= w < 1.       *
*              For w = 0, BIDA* reduces to the IDA* algorithm.   *
*                                                                *
*****************************************************************)

program EuclideanTSP (input,output);

const
     maxcity       = 25;

type
     index       = 1..maxcity;

     (* Node parameters for any node n *)

     nodetype    = record

         visited : array[index] of boolean;
                   (* Cities visited on current tour (TRUE) *)
                   (* Cities to be visited (FALSE)          *)
         city,     (* Current city                          *)
         tourlength,(* Number of cities on path from s to n *)
         fvalue,   (* Current f_value of node n             *)
         tourcost  (* Cost of current path from s to n      *)
                    : integer
     end;
     nodes       = array[index] of nodetype;
     stack       = record
                     path : nodes;
                     top  : 0..maxcity;
                   end;
```

```
    coordinate  = record
                    xcoord,ycoord : real
                  end;
    coordinates = array[index] of coordinate;
    list        = array[index] of 0..maxcity;
    matrix      = array[index,index] of integer;

var
    S               : stack;
    P               : coordinates;
    C               : matrix;          (* Adjacency matrix *)
    next            : list;

    (* Global Statistics *)

    nodesexpanded,
    totalexpanded,

    (* Parameters of problem instance *)

    toursize,        (*  Number of cities in the tour,    m *)
    sigdigs,         (*  Maximum precision of edge costs, t *)

    seed            : integer;
    found           : boolean;
    lastcity        : 0..maxcity;
    divfactor       : real;
```

```
(*****************************************************************
*                                                                *
*  STACK OPERATORS                                               *
*                                                                *
*  Purpose : To control the depth-first search.                  *
*                                                                *
*****************************************************************)

procedure initializestack (var S:stack);
begin
     S.top := 0
end;  {initializestack}

function stackempty (S:stack): boolean;
begin
     stackempty := S.top = 0
end;   {stackempty}

procedure popstack (var S:stack);
begin
     S.top := S.top - 1
end;  {popstack}

procedure pushstack (node:nodetype; var S:stack);
begin
     S.top := S.top + 1;
     S.path[S.top] := node
end;  {pushstack}

procedure topstack (var node:nodetype; S:stack);
begin
     node := S.path[S.top]
end;  {topstack}
```

```
(*****************************************************************
*                                                                *
*  function UNIFORM                                              *
*                                                                *
*  Purpose : Returns a uniformly distributed random variable     *
*            u where 0 <= u < 1.                                 *
*                                                                *
*****************************************************************)

function uniform (var seed:integer): real;
var
     r : integer;
begin
     r := seed div 53668;
     seed := 40014*(seed mod 53668) - (r*12211);
     if seed < 0 then
        seed := seed + 2147483563;
     uniform := seed*4.656613E-10
end;  { uniform }

(*****************************************************************
*                                                                *
*  procedure INITIALIZECOORDINATES                               *
*                                                                *
*  Purpose : Generates m random points in the unit square        *
*            and stores them in vector P.                        *
*                                                                *
*  Uses    : function UNIFORM                                    *
*                                                                *
*****************************************************************)

procedure initializecoordinates(var P:coordinates);
var
     i:index;
begin
     for i:=1 to toursize do
         with P[i] do begin
              xcoord := uniform(seed);
              ycoord := uniform(seed);
              writeln(xcoord,ycoord);
```

```
        end
end;  {initializecoordinates}

(*****************************************************************
*                                                                *
*  procedure INITIALIZEMATRIX                                    *
*                                                                *
*  Purpose : Calculates the Euclidean distances between m        *
*            points that are randomly distributed in the         *
*            unit square.                                        *
*                                                                *
*****************************************************************)

procedure initializematrix (P       : coordinates;
                            var C   : matrix;
                            sigdigs : integer     );
var
     i,j                    : index;
     deltax,deltay,distance : real;
     factor                 : integer;
begin
     for i:=1 to toursize do
         C[1,i] := maxint;

     (* The edge costs are converted to integral values that
        preserve t significant digits after the decimal.    *)

     factor := round(exp(sigdigs*ln(10)));
     for i:=1 to toursize-1 do
         for j:=i+1 to toursize do begin
             deltax  := P[i].xcoord - P[j].xcoord;
             deltay  := P[i].ycoord - P[j].ycoord;
             distance := sqrt(sqr(deltax) + sqr(deltay));
             C[1,j]   := round(factor*distance);
             C[j,i]   := C[1,j]
         end
end;  {initializematrix}
```

```
(*****************************************************************
*                                                                *
*  function SELECTTOUR                                           *
*                                                                *
*  Purpose : Returns the cost of the leftmost solution path,     *
*            that is, returns the cost of the tour from          *
*            city 1 to city 2, city 2 to city 3, ..., city m     *
*            to city 1.                                          *
*                                                                *
*****************************************************************)

function selecttour (C:matrix): integer;
var
   i     : index;
   total : integer;
begin
     total := 0;
     for i:=1 to toursize-1 do
         total := total + C[i,i+1];
     selecttour := total + C[toursize,1]
end;  { selecttour }
```

```
(*****************************************************************
*                                                                *
*  function BOUND                                                *
*                                                                *
*  Purpose : Returns the f-value of node n where:                *
*                                                                *
*            1) g(n) is the cost of the current path from        *
*               the start node s to n.                           *
*                                                                *
*            2) h(n) is equal to (r(n) + t(n))/2 where:         *
*                                                                *
*               a) r(n) is equal to the sum of the two           *
*                  minimum cost edges out of each node NOT on    *
*                  the current path from s to n.  Each edge      *
*                  may not be connected to a node in the         *
*                  middle of the current path from s to n.       *
*                  As well, only one edge may connect to         *
*                  either node s or node n.                      *
*                                                                *
*               b) t(n) is equal to the sum of the minimum       *
*                  cost edges out of the start node s and        *
*                  node n to a node NOT on the current path      *
*                  from s to n.                                  *
*                                                                *
*****************************************************************)

function bound (node     : nodetype;
                C        : matrix   ) : integer;
var
     h,min1,min2 : integer;
     i,j         : index;
begin
     with node do
          if tourlength = toursize-1 then begin

          (* Only one city remains to be visited *)

             lastcity := 1;
             while visited[lastcity] = true do
                   lastcity := lastcity + 1;
```

```
   bound := tourcost +
            C[city,lastcity] + C[lastcity,1];
end
else
   begin
        h := 0;

        (* Evaluate r(n) *)

        for i:=1 to toursize do
            if visited[i] = false then begin
               if C[i,1] < C[i,city] then
                  min1 := C[i,1]
               else
                  min1 := C[i,city];
               min2 := maxint;
               for j:=1 to toursize do
                   if visited[j] = false then
                      if C[i,j] < min1 then begin
                         min2 := min1;
                         min1 := C[i,j]
                      end
                   else
                      if C[i,j] < min2 then
                         min2 := C[i,j];
               h := h + min1 + min2
            end;

        (* Evaluate t(n) *)

        min1 := maxint;
        for i:=1 to toursize do
            if (visited[i] = false) and
               (C[1,i] < min1) then
               min1 := C[1,i];
        h    := h + min1;
        min2 := maxint;
        for i:=1 to toursize do
            if (visited[i] = false) and
               (C[city,i] < min2) then
```

```
                    min2 := C[city,1];
                h   := h + min2;

                (* Evaluate f(n) *)

                bound := tourcost + round(h/2)
            end
end;  { bound }
```

```
(*****************************************************************
*                                                                *
*  procedure CREATEROOT                                          *
*                                                                *
*  Purpose : Initializes the parameters of the start node s.     *
*                                                                *
*****************************************************************)

procedure createroot (var root:nodetype; C:matrix);
var
     i : index;
begin
     with root do begin
          city := 1;
          tourlength := 1;
          tourcost := 0;
          for i:=1 to toursize do
              visited[i] := false;
          visited[1] := true;
          fvalue := bound(root,C)
     end
end;  { createroot }

(*****************************************************************
*                                                                *
*  procedure CREATECHILD                                         *
*                                                                *
*  Purpose : Initializes the parameters of node n.               *
*                                                                *
*****************************************************************)

procedure createchild (var child : nodetype;
                       parent    : nodetype;
                       C         : matrix   );
var
     i : index;
begin
     with child do begin
          city := next[parent.city];
          tourlength := parent.tourlength + 1;
```

```
        tourcost := parent.tourcost +
                    C[parent.city,child.city];
        for i:=1 to toursize do
            visited[i] := parent.visited[i];
        visited[city] := true;
        fvalue := bound(child,C);
        next[city] := 0
    end
end;  { createchild }

(*****************************************************************
*                                                                *
*  function NEXTCITY                                             *
*                                                                *
*  Purpose : Determines the next city, if any, to be expanded.*
*                                                                *
*****************************************************************)

function nextcity (node:nodetype; var next:list): integer;
var
    city : integer;
begin
    city := next[node.city];
    repeat
          city := city + 1
    until (city>toursize) or (node.visited[city] = false);

    (* Note: If city > toursize then node is fully-expanded *)

    next[node.city] := city;
    nextcity := city
end;  { nextcity }
```

```
(*****************************************************************
*                                                                *
*  procedure DEPTHFIRSTSEARCH                                    *
*                                                                *
*  Purpose : Performs a bounded, depth-first search of the       *
*            solution space of the ETSP until either:            *
*                                                                *
*            1) A solution path is found or                      *
*                                                                *
*            2) The f-values of all expandable nodes is          *
*               greater than the iteration bound.                *
*                                                                *
*  Uses    : STACK OPERATORS, CREATECHILD, NEXTCITY              *
*                                                                *
*****************************************************************)

procedure depthfirstsearch (C                : matrix;
                            iterationbound   : integer;
                            var nextbound,
                                solutionvalue : integer );
var
     parent,child : nodetype;
     expanded     : array[index] of boolean;
     i            : index;
begin
     for i:=1 to toursize do
         expanded[i] := false;
     while not (stackempty(S) or found) do begin
           topstack(parent,S);
           if (parent.tourlength <= toursize-2)   and
              (nextcity(parent,next) <= toursize) then begin

              (* Generate next successsor of parent node *)

              createchild(child,parent,C);
              if not expanded[parent.city] then begin
                 nodesexpanded := nodesexpanded + 1;
                 expanded[parent.city] := true
              end;
              if child.fvalue <= iterationbound then
```

```
                pushstack(child,S)
             else
                if (child.fvalue < nextbound) then
                   nextbound := child.fvalue
          end
          else
             if parent.tourlength = toursize-1 then begin
                solutionvalue := parent.fvalue;
                found := true
             end
             else (* Node fully expanded *)
                begin
                     expanded[parent.city] := false;
                     popstack(S)
                end;
     end;
end;  { depthfirstsearch }
```

```
(*****************************************************************
*                                                                *
*  procedure ITERATIVEDEEPENING                                  *
*                                                                *
*  Purpose : Performs successive depth-first searches until      *
*            the lower bound is equal to the upper bound.        *
*            The cost bound of each iteration is equal to:       *
*                                                                *
*                (1-w)L + wU  where  0 <= w < 1.                 *
*                                                                *
*  Uses    : STACK OPERATORS, DEPTHFIRSTSEARCH                   *
*                                                                *
*****************************************************************)

procedure iterativedeepening(C          : matrix;
                             divfactor  : real   );
var
   node,rootnode  : nodetype;
   lowerbound,
   upperbound,
   nextbound,
   solutionvalue,
   iterationbound : integer;
begin
     createroot(rootnode,C);
     nextbound     := rootnode.fvalue;
     upperbound    := selecttour(C);
     found         := false;
     totalexpanded := 0;
     repeat
           initializestack(S);
           pushstack(rootnode,S);
           next[rootnode.city] := 0;
           nodesexpanded       := 0;

           (* Update lower bound *)

           if not found then begin
              lowerbound := nextbound;
              nextbound  := maxint;
```

```
            end;
            found          := false;

            (* Evaluate cost bound *)

            iterationbound := lowerbound +
                              trunc((upperbound - lowerbound)
                                          * divfactor);
            depthfirstsearch(C,iterationbound,
                             nextbound,solutionvalue);

            (* Update upper bound *)

            if found then
               upperbound     := solutionvalue;
            if lowerbound <> upperbound then
               totalexpanded := totalexpanded + nodesexpanded;
         until lowerbound = upperbound;

         (* Output optimal tour cost and path *)

         writeln('Tour Cost : ',upperbound:0);
         writeln('Nodes Expanded : ',totalexpanded:0);
         write('Optimal Tour: 1 ',lastcity:0);
         while not stackempty(S) do begin
               topstack(node,S);
               write(' ',node.city:0);
               popstack(S);
         end;
         writeln;
    end;  { iterativedeepening }
```

```
(*****************************************************************
*                                                               *
*                        MAIN PROGRAM                           *
*                                                               *
*****************************************************************)

begin  { main program }
     seed := 20000;

     (* Enter parameters *)

     writeln('Enter m, t and w:');
     readln(toursize,sigdigs,divfactor);
     initializecoordinates(P);
     initializematrix(P,C,sigdigs);
     iterativedeepening(C,divfactor);
end.
```