University of Alberta

SINGLE AGENT SEARCH WITH ADMISSIBLE INCONSISTENT HEURISTICS

by

Zhifu Zhang 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2008

Your file Votre référence
ISBN: 978-0-494-47455-6
Our file Notre référence
ISBN: 978-0-494-47455-6

University of Alberta

Library Release Form

Name of Author: Zhifu Zhang

Title of Thesis: Single Agent Search With Admissible Inconsistent Heuristics

Degree: Master of Science

Year this Degree Granted: 2008

Zhifu Zhang
3A 9015, 112 Street
Edmonton, Alberta
Canada, T6G 2C5

Date: _____

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Single Agent Search With Admissible Inconsistent Heuristics** submitted by Zhifu Zhang in partial fulfillment of the requirements for the degree of **Master of Science**.

_____

Jonathan Schaeffer (Supervisor)

_____

Robert Holte (Supervisor)

_____

Martin Mueller

_____

Michael Carbonaro (External)

_____

Nathan Sturtevant

Date: _____

Abstract

Searching for a path that connects a starting position to a goal position is a common task in artificial intelligence. There is a kind of search called admissible search that guarantees finding the minimum-cost solution path. Heuristics are popularly used to speed up the search, where a heuristic is a funtion, usually in the form of a lookup table, that estimates the cost from a given position to the goal. A* is a popular heuristic search algorithm that guarantees finding the minimum-cost path when using a class of heuristics called admissible heuristics. Admissible heuristics can be further divided into consistent and inconsistent heuristics. A* using an inconsistent heuristic can have a worst-case runtime complexity which is exponential in the problem size. However, inconsistent heuristics are valuable since state-of-the-art heuristic generating techniques produce high quality, possibly inconsistent heuristics. This thesis focuses on A*-like search using inconsistent heuristics.

In this thesis, we first made some clarifications and corrections to the reserach literature on inconsistent heuristics. We then implemented the historical algorithms B, B′ and C, which are variants of A* designed for inconsistent heuristics, where B and B′ guarantees a quadratic worst-case bound. We adopted the recently invented BPMX technique to A*, and implemented the recently invented variant of A* called Delay. We introduce a new variant of A* called DualProp (DP). We perform experiments in two domains comparing the performance of the above algorithms. In the theoretical aspect, we proved a quadratic worst-case bound of A* with an additional assumption.

Acknowledgements

There are many people who I am indebted to. First and foremost are my supervisor Professor Jonathan Schaeffer and co-supervisor Professor Robert Holte, who provided guidance, encouragement and a wonderful research environment. They have been instrumental during my initial explorations of this topic and my thesis revisions. Their expertise in heuristic search, insightful suggestions, and constructive feedbacks played an essential role in this work.

I convey special thanks to Dr. Nathan Sturtevant, who provided insights to my research and thesis revisions, as well as providing the search framework, HOG, to enable my experimental study. He also provided testing data for the pathfinding domain and spent time in adapting the TopSpin code from another group into the HOG environment, which enabled my TopSpin experiments. The weekly meetings with Professor Schaeffer, Professor Holte and Dr. Sturtevant have always been enlightening. It was a privilege to work with them.

I would like to thank Neil Burch for providing the TopSpin code, and other contributors to the HOG environment. Their efforts made my experiment setup much easier.

I am grateful to my friends in Canada and China. Their support and timely distraction have kept me productive during my graduate study.

My parents, brother and sister have been supportive at every step. They made my endeavor worthwhile.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this thesis, search is the task of finding a path from the start node to a goal node in a graph. Applications of search can be easily found in daily life. Consider the map of some cities in Romania in Figure 1.1, which is adapted and modified from [39]. The circles represent cities and the name of a city is shown next to it; the line segments represent roads connecting cities and the distance between the cities is shown next to the road. Supposing you want to go from Zerind to Bucharest, you can easily find a path that connects the two cities. However, if you want to find the shortest path, it is not so intuitive. In fact, the shortest path is $Zerind \rightarrow Arad \rightarrow Sibiu \rightarrow RimnicuVilcea \rightarrow Pitesti \rightarrow Bucharest$ with a distance of 493. When maps become larger and more complicated, the need for a systematic computational approach becomes more apparent. Single agent search is the subject that studies such computational approaches.

Search problems can be formulated as searching for a solution path that connects a start state to a goal state in a (finite or infinite) state space. A *state* is a configuration of the problem, and the set of configurations reachable by applying a finite sequence of actions from the initial configuration form a *state space* [35, 39]. Famous graph search algorithms developed during the early years of computer science research include breadth-first-search (BFS) [34], depth-first-search (DFS) [18] and Dijkstra's [9] algorithm. When a transition in the state space is associated with a cost, it is usually desirable to find a minimum-cost solution path. A search algorithm is *admissible* if it guarantees finding a minimum-cost solution path. BFS (when edge costs are uniform) and Dijkstra's algorithm are examples of admissible search algorithms.

Heuristics are functions that estimate the distance from any given state to a goal state. A heuristic function that never overestimates is called an *admissible* heuristic function. Single agent admissible heuristic search uses admissible heuristics to guide the search. Using heuristics usually significantly reduces the search effort needed to solve a problem, since the portion of the state space that needs to be explored is smaller than searching without heuristics. A* [16] and Iterative-Deepening A* (IDA*) [22] are examples of single agent admissible heuristic search algorithms.

Example problems of single agent admissible search include robot navigation, combinatoric puzzles, scheduling, sequence alignment, protein design, and planning. There are other search problems

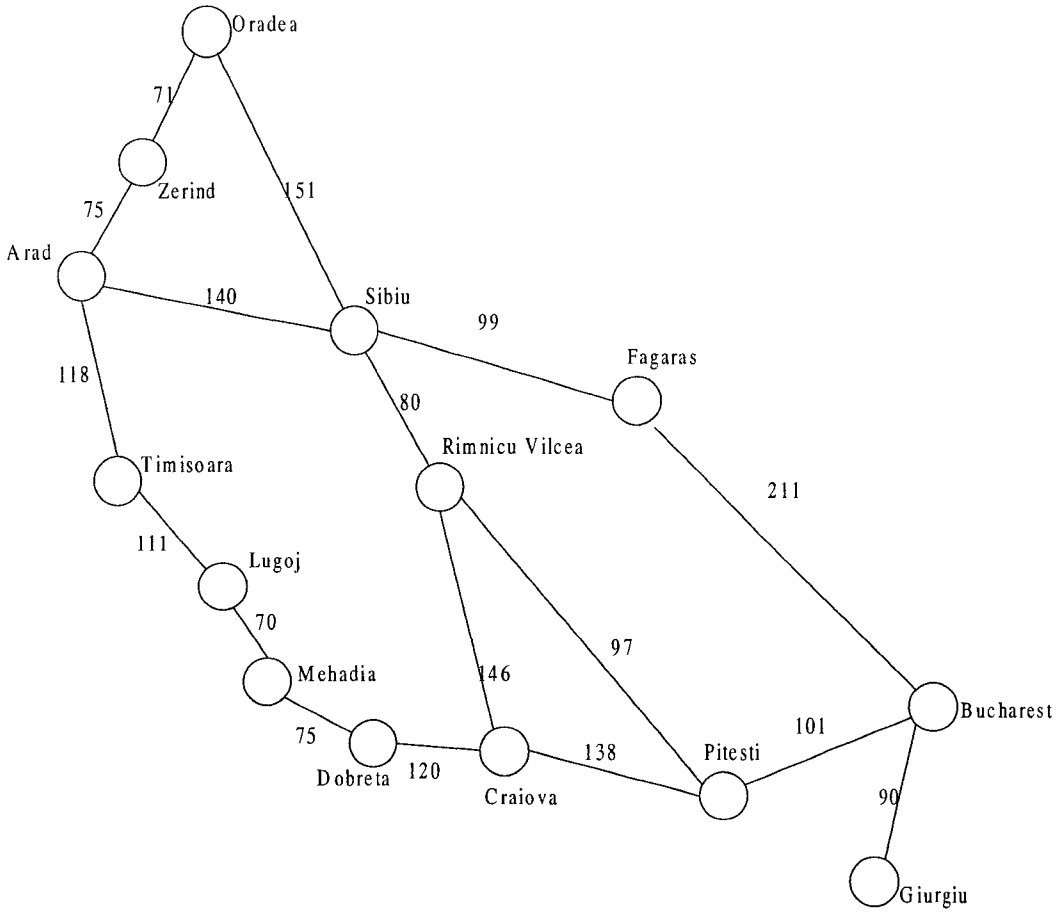

Figure 1.1: Cities in Romania.

that do not fall into the single agent search category, such as adversarial search (chess), problems with chance (backgammon), problems with hidden information (poker), problems with continuous state spaces (robot arm control), and multiple-agent search.

Single agent admissible search algorithms normally maintain a path-cost estimate for each node. When processing a node, its successors are generated, and the path-cost estimates of successors are reduced if possible. Such processing on the selected node is called a *node expansion*. When new expansions and re-expansions have comparable costs, the runtime is proportional to the number of node expansions, thus node expansions is a good measure of performance when comparing these algorithms.

For a node n, we use $h(n)$ to denote its heuristic value, $g(n)$ to denote the current estimate of the minimum cost from start to n, and $f(n) = g(n) + h(n)$ is called the f value of n and is used by many heuristic search algorithms.

1.1 Problems With Inconsistent Heuristics

A* [16] is a standard and popular algorithm for single agent heuristic search. It is guaranteed to find an optimal path if the heuristic is admissible and an optimal path exists.

An admissible heuristic is called *consistent* if, for any two connected nodes n_i and n_j, $h(n_i) - h(n_j) \leq dist(n_i, n_j)$, where $dist(n_i, n_j)$ is the minimum distance between the two nodes. Consistency guarantees admissibility, but admissibility does not guarantee consistency [36].

If an admissible heuristic is also consistent, then the total number of expansions in A* is the same as the number of unique nodes expanded. That is, each expanded node is never re-expanded. If the heuristic is admissible but inconsistent, the number of expansions could be much larger. Researchers have shown examples where the number of node expansions of A* can be an exponential function of the number of unique nodes expanded [32]. Later, variant algorithms of A* appeared, namely B [32], B′ [33] and C [1], which guarantee better worst-case bounds on the number of node expansions. These algorithms have $O(N^2)$ worst-case node expansions when there are N distinct nodes expanded. This thesis will concentrate on admissible but inconsistent heuristics. For brevity, we call them inconsistent heuristics from now on.

Given two admissible heuristic functions $h1$ and $h2$, we say $h1$ *dominates* $h2$ if $h1$ never returns a smaller value for a node than $h2$. It is believed that most admissible heuristics are naturally consistent [26]. Thus, historically the use of inconsistent heuristics was generally avoided or ignored and interest in the research of inconsistent heuristics was neglected. However, for a given problem, if there is an inconsistent heuristic that dominates all available consistent heuristics, then the number of distinct nodes expanded by A* using the inconsistent heuristic can be much smaller than A* using other heuristics. Despite possible re-expansions, the total expansions can still be smaller. This will be illustrated by our experiments in a later chapter.

1.2 Use of Inconsistent Heuristics

Research in heuristic search mainly has two themes: improving the search algorithms and improving the quality of the heuristics. Recently, pattern databases (PDBs) have attracted a lot of interest in the generation of high quality heuristics. A PDB [8] is a lookup table pre-computed by completely solving an abstraction of the original problem,[1] and it can be reused to solve different instances of the original problem [8]. State-of-the-art results have been reported in using PDBs to solve the Rubik's Cube problem [25], the 4-peg Towers of Hanoi problem [10, 28] and the sliding tile puzzles [27]. However, PDBs normally require significant pre-processing time to build pattern databases and requires lots of memory when used. Thus smart and compact ways of using PDBs are a necessity for many applications.

Researchers recently summarized several ways where inconsistent heuristics may arise when

[1] An abstraction of a problem is a reformulation that maps multiple states into one.

using pattern databases [43]:

- Random selection of heuristics. When we have several pattern databases at hand for a problem, we may want to use as many of them as possible, as long as they can fit into memory. For some domains, extra heuristics come from symmetries so they do not require extra memory. Consulting multiple heuristics and taking the maximum value can overcome the weaknesses of individual heuristics in different areas of the search space. However, each heuristic consultation increases the time to compute the final heuristic, and additional consultation provides diminishing return. Thus, it may be a good idea to randomly consult just one of the heuristics. When there is no correlation between the multiple heuristics, random selection results in an inconsistent heuristic.

- Dual heuristics. In permutation spaces, where operators are reversible and costs symmetric, each state s has a dual state s^d [12]. The dual heuristic is computed by mapping to the dual state and then looking up the PDB, which often results in an inconsistent heuristic [12]. Taking the maximum of the regular PDB lookup and the dual lookup produces a dominating but inconsistent heuristic with no extra memory cost.

- Compressed pattern databases. When a PDB is too large to fit into memory, it must be compressed in some way. Some compression techniques are lossless [2], but others are lossy and can introduce inconsistency [11, 40].

Inconsistent PDBs have been successfully used to improve the search in combinatorial domains with IDA* [43] [2] and in LRTA* search [5].[3] As memories on modern computers grow larger and heuristics get better, A* becomes more capable of attacking larger problems. Also, techniques for handling inconsistent heuristics in A* can be adapted to related algorithms in other fields such as LRTA* and its variants, and the memory-bounded algorithm SMA*.[4]

The objectives of this dissertation are:

- understanding why inconsistent heuristics have poor performance in A*,

- evaluating the performance of A*-like algorithms when using inconsistent heuristics, and

- designing new A*-like algorithms that are more suitable for use with inconsistent heuristics.

[2] Described in detail in Chapter 2

[3] Real-time heuristic search is an area of research where the agent can perceive its environment to a limited extent and must plan the next move within a limited time quota. In other words, the per move time quota is not related to the problem size. Algorithms for this field find suboptimal paths but can improve the solution quality over multiple trials. Famous algorithms for this field include Learning Real-Time A* (LRTA*) [23] and its variants [4, 20, 21, 37]. The initial heuristic values for all nodes could be zero, and these algorithms use A* search locally and improve the heuristic values during the search. Improving the heuristic values during the search is critical for ensuring that the goal is found. Many algorithms in this domain introduce inconsistency while improving the heuristic values.

[4] SMA* [38] is a memory-bounded version of A* that, when memory reaches a preset limit selectively kicks out old nodes to make room for new nodes. SMA* is guaranteed to find an optimal solution when the memory is sufficient to store an optimal path [38]. It was reported to significantly outperform IDA* in 3x7-puzzles even when using a memory size equal to twice that of the solution path [38]. A MxN-puzzle is a sliding tile puzzle that has tiles numbered 1 to $M * N - 1$ in a MxN rectangular frame, leaving one empty slot. The player is allowed to slide one adjacent tile into the empty slot at each step, and the goal is to sort all the tiles into increasing order, with the empty slot at the first place.

1.3 Contributions

The following are the contributions of this dissertation.

Clarifications and Corrections to the research literature. In 1977, Martelli showed a graph family and inconsistent heuristic configuration where A* will perform $O(2^N)$ expansions in terms of the graph size N [32]. As a solution, he proposed the algorithm B, which was proved to perform $O(N^2)$ expansions in the worst case [32]. In 1984, Mero proposed an algorithm based on B, which he called B′, that also performs $O(N^2)$ expansions in the worst case [33]. A* and B will not modify heuristic values during a search, while algorithm B′ features two rules to propagate the heuristic values, known as the *pathmax rules*. We clarify that the second pathmax rule is not stated in the correct form by Mero, and then give the correct form according to our understanding of the paper. Also, Mero proved that algorithm B′ will never perform more node expansions than algorithm B, which we will disprove by a counterexample.

First implementation of B and B′ algorithms. Algorithms B and B′ were proved to have better worst-case bounds than A* when using inconsistent heuristics, but there has been no implementations of them to our knowledge. We implemented these algorithms and tested their performance in real applications. The results show no performance advantage of B and B′ over A* in the pathfinding domain and the TopSpin domain. B and B′ perform asymptotically better than A* on Martelli's worst-case graph family. Their numbers of node expansions are shown in Table 1.1.

Graph Size	A*	B	B′
6	17	6	9
11	513	11	19
16	16385	16	29
21	524289	21	39

Table 1.1: Node Expansions by Algorithms on the Graph Family in Martelli's Paper.

Empirical evaluation of BPMX propagation in A*. In 2005, researchers [12] introduced the BPMX heuristic update rules as an enhanced version of the pathmax rules on undirected graphs. It was shown to significantly improve the search speed of IDA* when inconsistent heuristics are used. BPMX has not been applied to A*, and its effect in A* when using inconsistent heuristics has not been examined. We look into this empirically. Our experiments show great effectiveness of BPMX in the tested application domains.

The DP algorithm. We designed a new propagation method called DualProp (DP) as an enhancement to A* and its variants. Empirical results show that it improves performance when used with A*, but is less effective than BPMX propagation.

First empirical comparison of A*, B, B′, C, BPMX, Delay and DP algorithms using inconsistent heuristics. The Delay algorithm is another approach to speed up A* search when using inconsistent heuristics [41]. It was developed by Sturtevant. We performed experiments on a large

set of test cases in the pathfinding and TopSpin domains, and compared the performance of A*, B, B′, C, BPMX, Delay and DP algorithms, and got some interesting insights.

Improved theoretical bounds of A* under some common conditions. The worst-case examples of A* when using inconsistent heuristics have exponential runtime [32]. However, the examples are carefully constructed and do not seem to be likely to occur in real-world problems. We discovered that if we impose some additional assumptions on the heuristic values, or the edge costs, then the runtime complexity of A* can be quadratically bounded. The additional assumptions we impose are quite realistic for real-world problems. This discovery reduces the risk of using A* with inconsistent heuristics. Under these assumptions, there is no asymptotic advantage of B and B′ over A* in terms of worst-case complexity, and since there are implementation overheads of B and B′, we prefer using A* over B and B′ in such cases.

This thesis shows that the asymptotic worst-case complexity of A* when using inconsistent heuristics in many domains is much better than previously thought. There is a rich body of techniques, including B, B′, BPMX, Delay and DP, to speedup the search of A* when using inconsistent heuristics. So if there is an inconsistent heuristic that dominates all the available consistent heuristics by a large amount, then it is probably preferable. This then demonstrates that contrary to the popular perception in the literature, A*-like search with inconsistent heuristics is practical and effective.

Algorithm 1 Breadth First Search

```
1: procedure BFS(s,GOAL)
2:     g(s) ← 0
3:     CLOSED ← ∅
4:     OPEN ← {s}
5:     while OPEN ≠ ∅ do
6:         u ← EXTRACT-OLDEST(OPEN)                    ▷ First in first out
7:         CLOSED ← CLOSED ∪ {u}
8:         for each node v ∈ Neighbors[u] do
9:             if v ∉ CLOSED ∪ OPEN then
10:                g(v) ← g(u) + 1
11:                parent(v) ← u
12:                OPEN ← OPEN ∪ {v}
13:                if v ∈ GOAL then
14:                    return BUILD-PATH(v)
15:                end if
16:            end if
17:        end for
18:    end while
19:    return NIL
20: end procedure
```

Chapter 2

Background

One of the core tasks for solving many AI problems is searching for a sequence of steps from a start state to a goal state. This task can be described as a graph search problem as discussed below.

2.1 Goal-Directed Graph Search Problem

One of the simplest graph search problems is searching for a path from a start node to a goal node. Formally, given a graph $G = (V, E)$ where V is the set of nodes and E is the set of edges, a start node s, and a non-empty set of goal nodes $GOAL$, our task is to find a path from s to a goal node. Well known algorithms for solving this problem include breadth-first search (BFS) [34] and depth-first search (DFS) [18].

The pseudocode of BFS is shown in Algorithm 1. BFS maintains two sets of nodes, CLOSED

Algorithm 2 Depth First Search

```
1: procedure DFS(s,GOAL)
2:     return DFS-VISIT(MAKE-QUEUE(s),GOAL)
3: end procedure

4: procedure DFS-VISIT(path,GOAL)                    ▷ A recursive function
5:     u ← PEEK-TAIL(path)
6:     if u ∈ GOAL then
7:         return path
8:     end if
9:     for each node v ∈ Neighbors[u] do
10:        if v ∉ path then                           ▷ Check cycles
11:            g(v) ← g(u) + 1
12:            newpath ← DFS-VISIT(APPEND(path,v),GOAL)
13:            if newpath ≠ NIL then
14:                return newpath
15:            end if
16:        end if
17:    end for
18:    return NIL
19: end procedure
```

and OPEN. Initially CLOSED is empty, and OPEN contains the start node. Then it iteratively extracts a node from OPEN in first-in-first-out order, adds its unseen neighbors to OPEN, records their g values, and sets their parents to be the extracted node. The extracted node is then put into CLOSED. When a goal node is generated, the algorithm builds the path by looking up the child-parent relationship from the goal back to the start node in the search tree and then terminates. BFS is guaranteed to find a solution path if one exists, and the path has fewest edges among all possible paths [7]. BFS works on finite graphs and on infinite graphs if a solution exists.

The pseudocode of DFS is shown in Algorithm 2. DFS explores down a single path as deep as possible, avoiding cycles, until the path hits a dead-end or a goal is found. If a goal is found, the path is returned. If the search cannot proceed further and no goal is found, DFS backtracks to the previous level and tries an alternative sibling node that hasn't been tried. DFS may not find a solution on infinite graphs, even if a solution exists, and the path found is not guaranteed to have the fewest edges [7]. One advantage of DFS search is that its space complexity is linear in the length of the search path. During search, DFS only stores the nodes on the currently maintained path and their siblings. DFS only works on finite graphs.

If we are searching on a graph where each node has b successors (uniform branching factor) and the longest path it maintains has d_1 nodes, then the memory requirement of DFS is only $O(bd_1)$. In contrast, BFS remembers all nodes expanded. If the uniform branching factor is b and the shortest solution path has d_2 nodes, then BFS requires $O(b^{d_2})$ space (when $b > 1$) [39]. In the extreme case when $b = 1$, the search graph is a line and both algorithms require $O(d)$ space (d is the length of the only solution path). Nevertheless, as DFS does not guarantee finding the path with fewest edges,

it may explore all other reachable nodes before finding the goal. Under the same setting, the time complexity of BFS is $O(b^{d_2})$ (when $b > 1$), and the worst-case time complexity of DFS varies and depends on luck.

Algorithm 3 Iterative Deepening Search (IDS)

```
1: procedure IDS(s,GOAL)
2:     depthlimit ← 0
3:     g(s) ← 0
4:     repeat
5:         P ← DEPTH-LIMITED-SEARCH(MAKE-QUEUE(s),GOAL,depthlimit)
6:         depthlimit ← depthlimit + 1
7:     until P ≠ NIL or no new node found
8:     return P
9: end procedure

10: procedure DEPTH-LIMITED-SEARCH(path,GOAL,depthlimit)        ▷ A recursive function
11:     u ← PEEK-TAIL(path)
12:     if u ∈ GOAL then
13:         return path
14:     end if
15:     for each node v ∈ Neighbors[u] do
16:         g(v) ← g(u) + 1
17:         if g(v) ≤ depthlimit then                            ▷ Check depth limit
18:             newpath ← DEPTH-LIMITED-SEARCH(APPEND(path,v),GOAL,depthlimit)
19:             if newpath ≠ NIL then
20:                 return newpath
21:             end if
22:         end if
23:     end for
24:     return NIL
25: end procedure
```

A method called iterative-deepening search (IDS) combines some benefits of BFS and DFS [39]. The pseudocode is shown in Algorithm 3. It iteratively tries an increasing depth bound. For each depth bound, a depth-limited version of DFS is used to search for the goal. If no solution is found within the depth bound, the depth bound is increased by 1, and so on. IDS is guaranteed to find a fewest-edge solution (if one exists) as BFS does, but has a linear space complexity $O(bd)$ [39], where d is the length of the shortest solution path. IDS is a typical strategy to trade time for space. Compared to BFS, it introduces two kinds of redundancies: (1) in each iteration the nodes expanded in the previous iteration will be re-expanded; (2) if the graph has transpositions, a node may be expanded multiple times via different paths. On a tree with uniform branching factor, the asymptotic time complexity of IDS remains the same as BFS [39]. IDS works on finite graphs and on infinite graphs if a solution exists.

In the extreme case, the number of expansions by IDS could be an exponential function of the number of expansions by BFS. This is illustrated in Figure 2.1, where there are 2^{i-1} paths to the i^{th} node starting from the left and all edges have cost 1. Suppose this chain contains d nodes, BFS will

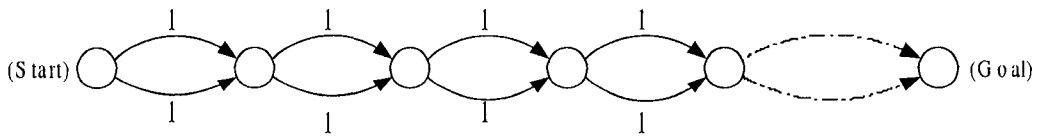

Figure 2.1: Extreme Case for IDS, from [39].

only perform d expansions, but IDS will perform $O(2^{d+1})$ expansions such that all possible paths are explored.

2.2 Single Agent Admissible Search

When each edge is associated with a cost, it is often desirable to find the path to the goal that has the minimum path cost, which is the sum of edge costs in the path. When edge costs are uniform, BFS and IDS are applicable for finding optimal paths, since an optimal path also has the fewest edges.

Single agent admissible search can be defined by a 4-tuple $(G, e, s, GOAL)$. Given a graph $G = (V, E)$, a start node s, a non-empty set of goal nodes $GOAL$, an edge cost function e, single agent admissible search aims to find the minimum-cost path from the start node to a goal node. The search terminates once the minimum-cost path to a goal is found.

For a node n, we use $g(n)$ (the g value of n) to denote the current minimum known cost from start to n, $g^*(n)$ to denote the minimum cost from start to n and $h^*(n)$ to denote the minimum cost from n to a goal. For any two adjacent nodes n_i and n_j, we use $e(n_i, n_j)$ to denote the edge cost from n_i to n_j. For any two nodes n_i and n_j, we use $dist(n_i, n_j)$ to denote the minimum cost from n_i to n_j.

In this dissertation, we will only be concerned with single agent admissible search problems with non-negative edge costs.

2.3 Single Source Shortest Path Problem

Given a graph $G = (V, E)$, a start node s, and an edge cost function e, the single source shortest path (SSSP) problem is to find minimum-cost paths to all nodes that are reachable from the start node. This problem is well defined if and only if there are no negative-cost cycles reachable from the start. The solution can be compactly represented as a tree, whose root is the start node, and a node's parent denotes its parent on a shortest path from the start. Algorithms for this problem include Bellman-Ford [3, 13] and Dijkstra [9].

Algorithm 4 Dijkstra's Algorithm (SSSP)

```
1:  procedure DIJKSTRA(e,s)                                  ▷ Adapted from [7]
2:      g(s) ← 0
3:      CLOSED ← ∅
4:      OPEN ← {s}
5:      while OPEN ≠ ∅ do
6:          u ← EXTRACT-MIN-G(OPEN)                          ▷ Get min-g node
7:          CLOSED ← CLOSED ∪ {u}
8:          for each node v ∈ Neighbors[u] do
9:              if v ∉ CLOSED ∪ OPEN then
10:                 g(v) ← g(u) + e(u, v)
11:                 parent(v) ← u
12:                 OPEN ← OPEN ∪ {v}
13:             else if g(v) > g(u) + e(u, v) then
14:                 g(v) ← g(u) + e(u, v)
15:                 parent(v) ← u
16:             end if
17:         end for
18:     end while
19: end procedure
```

2.4 Dijkstra's Algorithm

The original version of Dijkstra's algorithm [9] solves the single source shortest path problem on a graph with non-negative edges. Its pseudocode is shown in Algorithm 4.

This algorithm maintains two sets of nodes: CLOSED and OPEN. Initially CLOSED is empty and OPEN contains the start node. The the algorithm iteratively extracts the node with minimum g value from OPEN, adds its unseen neighbors to OPEN, and sets their g values and parents accordingly. It then updates the g values and parents of those neighbors whose g values can be decreased, and moves the extracted node into CLOSED. The time complexity of Dijkstra's algorithm is $O(V^2)$ and its space complexity is $O(V + E)$ [7]. This original SSSP version of Dijkstra's algorithm only works on finite graphs.

Dijkstra's algorithm can be easily adapted to single agent admissible search problems with non-negative costs using the additional condition that the search also terminates when the goal is expanded.

A modified version of Dijkstra's algorithm [19] can work on graphs with negative costs, only requiring that there is no negative-cost cycle. The modified Dijkstra's algorithm for single agent admissible search is shown in Algorithm 5. This version is almost the same as the original version except that the loop also terminates when the goal is expanded (lines 8-10), and that when a node in CLOSED has its g value reduced, it is moved back to OPEN (lines 19-21). An expanded (closed) node being put back into OPEN is called being *reopened.*

Dijkstra's algorithm guarantees finding the optimal path to a goal if one exists, thus it is an admissible algorithm. If there are no negative edges, the algorithm expands each node at most once;

Algorithm 5 Dijkstra's Algorithm (Modified)

```
1:  procedure DIJKSTRA(e,s,GOAL)
2:      g(s) ← 0
3:      CLOSED ← ∅
4:      OPEN ← {s}
5:      while OPEN ≠ ∅ do
6:          u ← EXTRACT-MIN-G(OPEN)                    ▷ Get min-g node
7:          CLOSED ← CLOSED ∪ {u}
8:          if u ∈ GOAL then
9:              return BUILD-PATH(u)
10:         end if
11:         for each node v ∈ Neighbors[u] do
12:             if v ∉ CLOSED ∪ OPEN then
13:                 g(v) ← g(u) + e(u, v)
14:                 parent(v) ← u
15:                 OPEN ← OPEN ∪ {v}
16:             else if g(v) > g(u) + e(u, v) then
17:                 g(v) ← g(u) + e(u, v)
18:                 parent(v) ← u
19:                 if v ∈ CLOSED then                 ▷ Reopen a closed node
20:                     CLOSED ← CLOSED − {v}
21:                     OPEN ← OPEN ∪ {v}
22:                 end if
23:             end if
24:         end for
25:     end while
26:     return NIL
27: end procedure
```

otherwise, the number of expansions can be $O(2^N)$ when N is the number of unique nodes expanded [19]. The time and space complexity for this version of Dijkstra's algorithm is hard to characterize using graph parameters V, E or b when the edge costs are not uniform. This is generally true for single agent admissible search algorithms with non-uniform edge costs. This version of Dijkstra's algorithm works on finite graphs and on infinite graphs if a solution exists.

2.5 Single Agent Heuristic Search

Single agent search without any additional information is also known as *blind search*. There is a kind of *informed search* called *heuristic search*. A *heuristic*, in our context, is a function for estimating the cost from any given state to a goal state. Thus, heuristic search can be defined by a 5-tuple $(G, e, h, s, GOAL)$, where h is the heuristic function.

Single agent search algorithms can be categorized according to space and time. In terms of the space requirement, there are memory-unbounded algorithms, memory-bounded algorithms, and linear space algorithms. Memory-unbounded algorithms include BFS, Dijkstra, A*, B and B′, and their memory requirement grows as the problem size grows. For memory-bounded algorithms, there is a preset limit of the number of nodes that can be stored, and once the limit is reached, the algorithm will selectively kick out some nodes from memory to make room for other new nodes. Example algorithms in this category include MA* [6] and SMA* [38]. Linear space algorithms have the smallest asymptotic space requirement, but not fully utilizing the memory will cause redundant search effort. Example algorithms in this category include IDS, IDA*, DFBnB [23], IE [38] and RBFS [24]. In terms of the time requirement, there are offline algorithms, real-time algorithms, and anytime algorithms. An offline algorithm must find an optimal solution or assert no solution exists before completion. Example algorithms in this category include A*, B and B′. A real-time algorithm has a tight budget in time and must return a move whether it finds a solution path or not. Example algorithms in this category include LRTA* [23], Koenig's LRTA* [20], LRTS [4], RTAA* [21] and P-LRTA* [37]. An anytime algorithm can terminate at any time after it finds the first solution, and if the solution is suboptimal and there is time remaining it will gradually improve the solution. Example algorithms in this category include Anytime A* [15] and ARA* [30]. Real-time and anytime search algorithms are useful in path planning in video games and robot planning. These algorithms are applicable in dynamic environments.

2.6 Admissible and Consistent Heuristics

An *admissible heuristic* never overestimates the path cost of any node to the goal, in other words $h(n) \leq h^*(n)$ for any node n [16]. Figure 2.2 shows an example of an inadmissible heuristic. Notations in the figure (and in subsequent figures) are as follows: the name of a node is beside the node, the cost of an edge is on the edge, and the heuristic value of a node is inside the node. In this

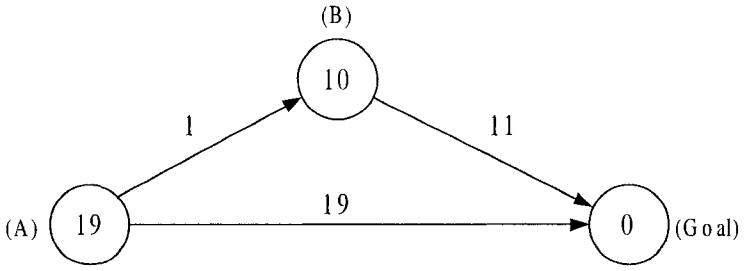

Figure 2.2: Inadmissible Heuristic.

figure, $h^*(A) = 12$, but $h(A) = 19 > h^*(A)$, so the heuristic function is not admissible.

A *consistent heuristic* is an admissible heuristic that additionally has the property that for any two nodes n_i and n_j, if there is a path from n_i to n_j then $h(n_i) - h(n_j) \leq dist(n_i, n_j)$. Figure 2.3 shows an admissible but inconsistent heuristic. As we can see, $h(A) - h(B) = 12 - 10 \not\leq 1 = dist(A, B)$. Because $h(A) = 12$, $h(B) = 10$, $dist(A, B) = 1$, and $h(A)$ and $h(B)$ are lower bounds of actual distances, we can infer that $h(A) \leq h^*(A) \leq dist(A, B) + h^*(B)$, which implies $h^*(B) \geq h(A) - dist(A, B) = 11$. Thus we can update $h(B)$ to 11 by such reasoning. In fact, an intuitive interpretation of consistent heuristics is that we cannot update the heuristic values of two non-goal nodes by only comparing their heuristic values and the minimum distance between them.

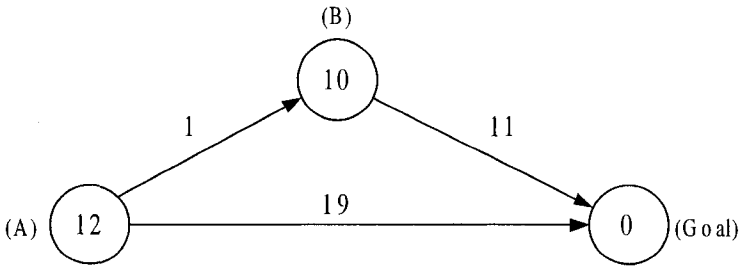

Figure 2.3: Inconsistent Heuristic.

A *locally consistent heuristic* is an admissible heuristic that has the following property: for any two nodes n_i and n_j connected by an edge (n_i, n_j), the inequality $h(n_i) - h(n_j) \leq e(n_i, n_j)$ holds. It has been proved that consistency is equivalent to local consistency [36].

2.7 A* Algorithm

The A* algorithm is an admissible algorithm for single agent search problems (with admissible heuristics). The pseudocode for A* is shown in Algorithm 6. We can see that this pseudocode is almost identical to that of the modified Dijkstra's algorithm. The differences include that every time A* chooses the next node for expansion, it chooses the node in OPEN with minimum f value (line 6) (breaks ties arbitrarily but favors goal nodes), while Dijkstra's (modified) algorithm chooses the

Algorithm 6 A*

```
1: procedure A*(e,h,s,GOAL)
2:     g(s) ← 0
3:     CLOSED ← ∅
4:     OPEN ← {s}
5:     while OPEN ≠ ∅ do
6:         u ← EXTRACT-MIN-F(OPEN)                          ▷ Get min-f node
7:         CLOSED ← CLOSED ∪ {u}
8:         if u ∈ GOAL then
9:             return BUILD-PATH(u)
10:        end if
11:        for each node v ∈ Neighbors[u] do
12:            if v ∉ CLOSED ∪ OPEN then
13:                g(v) ← g(u) + e(u, v)
14:                f(v) ← g(v) + h(v)
15:                parent(v) ← u
16:                OPEN ← OPEN ∪ {v}
17:            else if g(v) > g(u) + e(u, v) then
18:                g(v) ← g(u) + e(u, v)
19:                f(v) ← g(v) + h(v)
20:                parent(v) ← u
21:                if v ∈ CLOSED then                       ▷ Reopen a closed node
22:                    CLOSED ← CLOSED − {v}
23:                    OPEN ← OPEN ∪ {v}
24:                end if
25:            end if
26:        end for
27:    end while
28:    return NIL
29: end procedure
```

node with minimum g value from OPEN. A* also needs to keep track of the f values of nodes. A* and its subsequently mentioned variants work on finite graphs and on infinite graphs if a solution exists. This is true for all admissible single agent search algorithms.

When the heuristic is admissible and there are no negative-cost cycles in the state graph, A* is guaranteed to find an optimal solution if one exists [16]. Thus it is an admissible algorithm. Given the same admissible heuristic, no other admissible search algorithm, which is no more informed than A*, is guaranteed to expand a smaller set of distinct nodes than A* [16, 17]. If the heuristic is admissible and consistent, A* expands each node at most once [16]. If the heuristic is admissible but not consistent, nodes can be reopened and the number of expansions can be $O(2^N)$ where N is the number of expanded nodes, as shown by Martelli [32].

A family of worst-case graphs G_N can be constructed using the following scheme [32]. Given an integer N, G_N has $N+1$ nodes $(n_0, n_1, ..., n_N)$, where n_N is the start node and n_0 is goal node, and there are directed edges (n_i, n_j) such that their edge costs satisfy:

- $e(n_i, n_j) = 2^{i-2} + i - 2^{j-1} - j$, for $1 \leq j < i \leq N$, and
- $e(n_1, n_0) = 2^{N-1} + N - 2$.

The heuristic values for each node are:

- $h(n_0) = h(n_1) = 0$, and
- $h(n_i) = 2^{i-1} + 2 * i - 3$, for $1 < i \leq N$.

For example, Figure 2.4 is G_5 from Martelli [32]. The order of node expansions performed by A* on G_5 is shown in Table 2.1, which is adapted from Martelli's paper and modified. Note that the values in the table are f followed by g values, closed nodes are in square brackets, and the node to expand is marked by a *.

As seen in the table, node n_1 is expanded 8 times, node n_2 4 times, node n_3 2 times, and nodes n_4, n_5, n_0 once, totaling 17 times. In this fashion, the number of expansions is $O(2^N)$ in any G_N.

2.8 Algorithm B

Having seen the exponential behavior of A*, algorithm B was designed to bound the worst-case time complexity when using an inconsistent heuristic while maintaining admissibility [32]. The pseudocode for B is shown in Algorithm 7. It is similar to A*, with the following additions. It maintains a global variable F that keeps track of the maximum f seen so far in the nodes selected for expansion (lines 3 and 11). When choosing the next node to expand, if the minimum f in OPEN (denoted f_m) satisfies $f_m \geq F$, then the node with minimum f is chosen as in A* (line 10), otherwise the node with minimum g among the nodes with $f < F$ is chosen for expansion (line 8).

When $f_m < F$, algorithm B is essentially using Dijkstra's selection rule to select the node to expand. The rationale behind this is the following. Any node with $f < g^*(GOAL)$ in OPEN

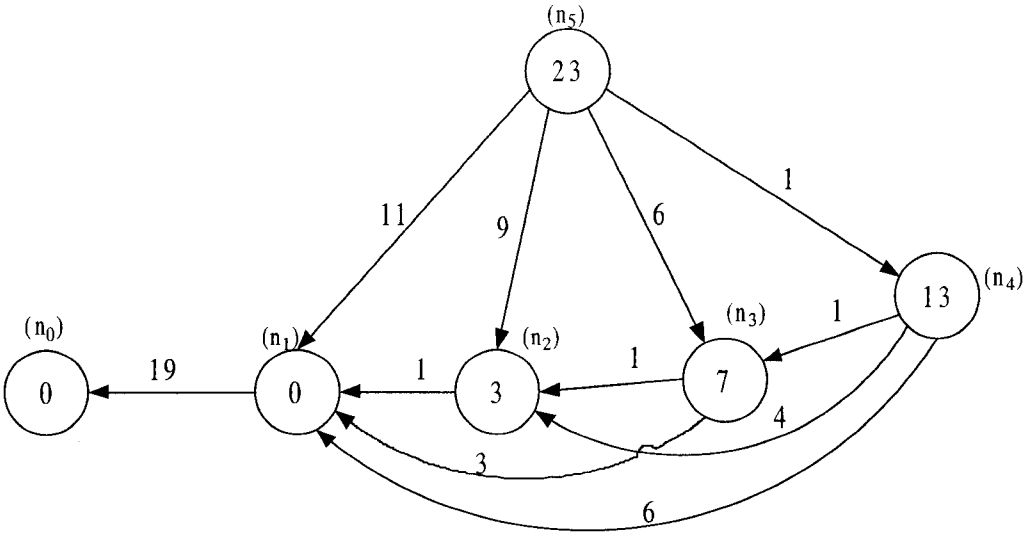

Figure 2.4: G_5 from Martelli [32].

n_5	n_4	n_3	n_2	n_1	n_0
23,0*	-	-	-	-	-
[23,0]	14,1	13,6	12,9	11,11*	-
[23,0]	14,1	13,6	12,9*	[11,11]	30,30
[23,0]	14,1	13,6	[12,9]	10,10*	30,30
[23,0]	14,1	13,6*	[12,9]	[10,10]	29,29
[23,0]	14,1	[13,6]	10,7	9,9*	29,29
[23,0]	14,1	[13,6]	10,7*	[9,9]	28,28
[23,0]	14,1	[13,6]	[10,7]	8,8*	28,28
[23,0]	14,1*	[13,6]	[10,7]	[8,8]	27,27
[23,0]	[14,1]	9,2	8,5	7,7*	27,27
[23,0]	[14,1]	9,2	8,5*	[7,7]	26,26
[23,0]	[14,1]	9,2	[8,5]	6,6*	26,26
[23,0]	[14,1]	9,2*	[8,5]	[6,6]	25,25
[23,0]	[14,1]	[9,2]	6,3	5,5*	25,25
[23,0]	[14,1]	[9,2]	6,3*	[5,5]	24,24
[23,0]	[14,1]	[9,2]	[6,3]	4,4*	24,24
[23,0]	[14,1]	[9,2]	[6,3]	[4,4]	23,23*
[23,0]	[14,1]	[9,2]	[6,3]	[4,4]	[23,23]

Table 2.1: Execution of A* on Fig 2.4 (f,g shown).

Algorithm 7 Algorithm B

```
1: procedure B(e,h,s,GOAL)
2:     g(s) ← 0
3:     F ← 0
4:     CLOSED ← ∅
5:     OPEN ← {s}
6:     while OPEN ≠ ∅ do
7:         if ∃ node t ∈ OPEN with f(t) < F then
8:             u ← EXTRACT-MIN-G(OPEN,F)          ▷ Get min-g node with f < F
9:         else
10:            u ← EXTRACT-MIN-F(OPEN)            ▷ Get min-f node
11:            F ← f(u)
12:        end if
13:        CLOSED ← CLOSED ∪ {u}
14:        if u ∈ GOAL then
15:            return BUILD-PATH(u)
16:        end if
17:        for each node v ∈ Neighbors[u] do
18:            if v ∉ CLOSED ∪ OPEN then
19:                g(v) ← g(u) + e(u,v)
20:                f(v) ← g(v) + h(v)
21:                parent(v) ← u
22:                OPEN ← OPEN ∪ {v}
23:            else if g(v) > g(u) + e(u,v) then
24:                g(v) ← g(u) + e(u,v)
25:                f(v) ← g(v) + h(v)
26:                parent(v) ← u
27:                if v ∈ CLOSED then             ▷ Reopen a closed node
28:                    CLOSED ← CLOSED − {v}
29:                    OPEN ← OPEN ∪ {v}
30:                end if
31:            end if
32:        end for
33:    end while
34:    return NIL
35: end procedure
```

must be expanded before the goal and F is always no larger than $g^*(GOAL)$. Therefore nodes with $f < F$ in OPEN will have to be expanded before the goal [32]. Dijkstra's algorithm expands each node at most once when there are no negative edges, therefore it is a good choice to use Dijkstra's selection rule in this subset of nodes for reducing the number of re-expansions [32].

n_5	n_4	n_3	n_2	n_1	n_0	F
23,0*	-	-	-	-	-	0
[23,0]	14,1*	13,6	12,9	11,11	-	23
[23,0]	[14,1]	9,2*	8,5	7,7	30,30	23
[23,0]	[14,1]	[9,2]	6,3*	5,5	30,30	23
[23,0]	[14,1]	[9,2]	[6,3]	4,4*	30,30	23
[23,0]	[14,1]	[9,2]	[6,3]	[4,4]	23,23*	23
[23,0]	[14,1]	[9,2]	[6,3]	[4,4]	[23,23]	23

Table 2.2: Execution of B on Fig 2.4 (f,g shown).

The order of node expansions performed by B on G_5 is shown in Table 2.2. Each node is expanded once and the total number of expansions is 6, compared to 17 by A* (in Table 2.1).

Algorithm B performs at most $O(N^2)$ node expansions where N is the number of unique nodes expanded. The number of expansions by B is no larger than that by A* for the same problem instance [32], if they break ties in the same way.

2.9 Algorithm C

Bagchi and Mahanti proposed a variant of algorithm B called algorithm [1]. The pseudocode for C is shown in Algorithm 8. The modifications from B include the following: changes the condition for the special case from $f_m < F$ to $f_m \leq F$ (line 7); when breaking ties in f in the normal case, instead of breaking ties favoring goal nodes, it favors goal nodes and then favors smaller g values (line 10). C guarantees $O(N^2)$ worst-case time complexity when N is the number of distinct nodes expanded. C is admissible with admissible heuristics. The authors proved some favorable properties of C, compared to A* and B, when using inadmissible heuristics: it finds a solution no worse than B with inadmissible heuristics; with so-called *proper heuristics*, A* performs at most $O(N^2)$ expansions, B performs at most $O(N)$, C expands each node at most once as well as guaranteeing the solution to be optimal [1].

2.10 Algorithm B′

Mero modified algorithm B to update heuristic values during the search while maintains admissibility, calling the algorithm B′ [33]. The pseudocode for B′ is shown in Algorithm 9. It is similar to algorithm B, except that it includes two heuristic update rules (also known as *pathmax rules*) for correcting heuristic inconsistencies. The original version of the rules contains an error, which will

Algorithm 8 Algorithm C

```
1: procedure C(e,h,s,GOAL)
2:     g(s) ← 0
3:     F ← 0
4:     CLOSED ← ∅
5:     OPEN ← {s}
6:     while OPEN ≠ ∅ do
7:         if ∃ node t ∈ OPEN with f(t) ≤ F then
8:             u ← EXTRACT-MIN-G(OPEN,F)                ▷ Get min-g node with f ≤ F
9:         else
10:            u ← EXTRACT-MIN-F-G(OPEN)  ▷ Get min-f node, break ties favoring small g
11:            F ← f(u)
12:        end if
13:        CLOSED ← CLOSED ∪ {u}
14:        if u ∈ GOAL then
15:            return BUILD-PATH(u)
16:        end if
17:        for each node v ∈ Neighbors[u] do
18:            if v ∉ CLOSED ∪ OPEN then
19:                g(v) ← g(u) + e(u, v)
20:                f(v) ← g(v) + h(v)
21:                parent(v) ← u
22:                OPEN ← OPEN ∪ {v}
23:            else if g(v) > g(u) + e(u, v) then
24:                g(v) ← g(u) + e(u, v)
25:                f(v) ← g(v) + h(v)
26:                parent(v) ← u
27:                if v ∈ CLOSED then                   ▷ Reopen a closed node
28:                    CLOSED ← CLOSED − {v}
29:                    OPEN ← OPEN ∪ {v}
30:                end if
31:            end if
32:        end for
33:    end while
34:    return NIL
35: end procedure
```

Algorithm 9 Algorithm B′

```
1: procedure B′ (e,h,s,GOAL)
2:     g(s) ← 0
3:     F ← 0
4:     CLOSED ← ∅
5:     OPEN ← {s}
6:     while OPEN ≠ ∅ do
7:         if ∃ node t ∈ OPEN with f(t) < F then
8:             u ← EXTRACT-MIN-G(OPEN,F)          ▷ Get min-g node with f < F
9:         else
10:            u ← EXTRACT-MIN-F(OPEN)            ▷ Get min-f node
11:            F ← f(u)
12:        end if
13:        CLOSED ← CLOSED ∪ {u}
14:        for each node v ∈ Neighbors[u] do
15:            h(v) ← max(h(v), h(u) − e(u,v))    ▷ PathMax rule (a)
16:        end for
17:        h(u) ← max(h(u), min_{v∈Neighbors[u]}(h(v) + e(u,v)))   ▷ PathMax rule (b)
18:        if u ∈ GOAL then
19:            return BUILD-PATH(u)
20:        end if
21:        for each node v ∈ Neighbors[u] do
22:            if v ∉ CLOSED ∪ OPEN then
23:                g(v) ← g(u) + e(u,v)
24:                f(v) ← g(v) + h(v)
25:                parent(v) ← u
26:                OPEN ← OPEN ∪ {v}
27:            else if g(v) > g(u) + e(u,v) then
28:                g(v) ← g(u) + e(u,v)
29:                f(v) ← g(v) + h(v)
30:                parent(v) ← u
31:                if v ∈ CLOSED then             ▷ Reopen a closed node
32:                    CLOSED ← CLOSED − {v}
33:                    OPEN ← OPEN ∪ {v}
34:                end if
35:            end if
36:        end for
37:    end while
38:    return NIL
39: end procedure
```

be discussed in Chapter 3. The corrected version is presented here. Let u be a node for expansion, and v be any of its successors, the pathmax rules are:

(a) $h(v) \leftarrow max(h(v), h(u) - e(u,v))$ (lines 14-16), and

(b) $h(u) \leftarrow max(h(u), min_{v \in Neighbors[u]}(h(v) + e(u,v)))$ (line 17).

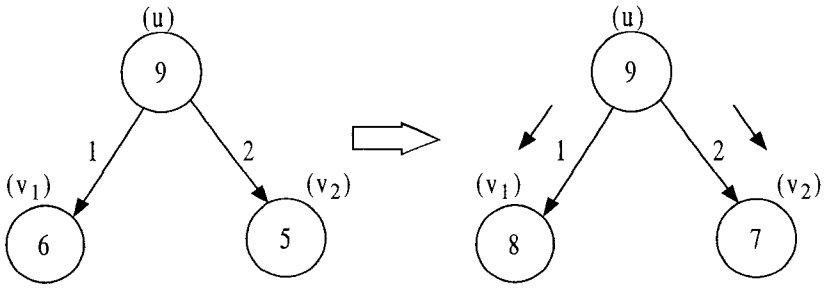

Figure 2.5: Pathmax Rule (a).

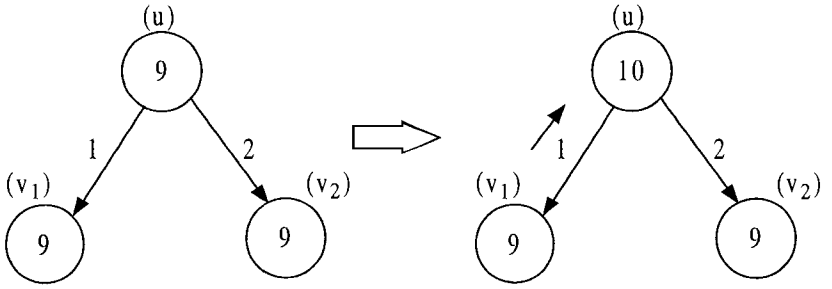

Figure 2.6: Pathmax Rule (b).

For rule (a), we know $h(u) \leq h^*(u)$ and $h(v) \leq h^*(v)$ since h is admissible, and also that $h^*(u) \leq e(u,v) + h(v)$ since u could reach the goal via v. Combining these facts we can infer that $h^*(v) \geq h(u) - e(u,v)$. Figure 2.5 shows how the parent node u updates the heuristic values of the child nodes v_1 and v_2 according to rule (a).

For rule (b), the optimal path from u to a goal must contain one of u's successors (unless u is a goal), so $h(u)$ is at least as large as $min_{v \in Neighbors[u]}(h(v) + e(u,v))$. Figure 2.6 shows how the child nodes v_1 and v_2 update the heuristic value of the parent node u according to rule (b).

Algorithm B′ performs at most $O(N^2)$ node expansions, when there are N unique nodes expanded [33]. When the update rules are used, there will never be nodes with $f < F$ in OPEN [33]. The author of B′ also claimed to have proved that B′ performs no more expansions than B in any problem instance [33], but later we will show this claim is false.

Algorithm 10 IDA*

```
1: procedure IDA*(e,h,s,GOAL)
2:     g(s) ← 0
3:     f(s) ← h(s)
4:     T ← f(s)                                              ▷ T is threshold
5:     repeat
6:         T_old ← T
7:         (P, T) ← BOUNDED-DFS(MAKE-QUEUE(s),e,h,GOAL,T)
8:     until P ≠ NIL or T = T_old                            ▷ T not updated means no solution
9:     return P
10: end procedure

11: procedure BOUNDED-DFS(path,e,h,GOAL,T)                   ▷ A recursive function
12:     u ← PEEK-TAIL(path)
13:     if u ∈ GOAL then
14:         return (path, T)
15:     end if
16:     T_new ← ∞
17:     for each node v ∈ Neighbors[u] do
18:         g(v) ← g(u) + e(u, v)
19:         f(v) ← g(v) + h(v)
20:         if f(v) ≤ T then                                  ▷ Check threshold
21:             (newpath, RT) ← BOUNDED-DFS(APPEND(path,v),e,h,GOAL,T)
22:             T_new ← min(T_new, RT)
23:             if newpath ≠ NIL then
24:                 return (newpath, T_new)
25:             end if
26:         else
27:             T_new ← min(T_new, f(v))                      ▷ Candidate value for next threshold
28:         end if
29:     end for
30:     return (NIL, T_new)
31: end procedure
```

2.11 IDA* Algorithm

Iterative-deepening A* (IDA*) is an iterative-deepening version of A* [22]. It maintains the admissibility of A*, while using an idea similar to IDS to reduce the space requirement. The pseudocode of IDA* is shown in Algorithm 10. IDA* uses an increasing threshold T, which is initially set to $f(s)$, where s is the start node (line 4). For each T value, a threshold-bounded version of DFS is used to search for the goal. If a goal is found within the threshold, the path is returned; otherwise, T is increased to the minimum f value that exceeds T (line 7) found in the last iteration.

IDA* is an admissible search algorithm. The worst-case space complexity of IDA* is $O(bd)$ where b is the max branching factor and d is the length of the optimal path with the most edges [39]. Nodes expanded under the previous T will be expanded again under the current T, so inconsistency is not a concern in IDA*. Significant progress has been made in predicting the runtime of IDA* given a specific heuristic distribution [29, 42].

Given a search problem, denote N as the number of distinct nodes expanded by A*. On trees, it has been proved that the worst-case time complexity of IDA* is $O(N^2)$ [31]. On graphs, IDA* suffers from the same exponential blow-up as IDS in extreme cases. It has been proved that the worst-case time complexity of IDA* on graphs is $O(2^{2N})$ [31].

2.12 BPMX Propagation

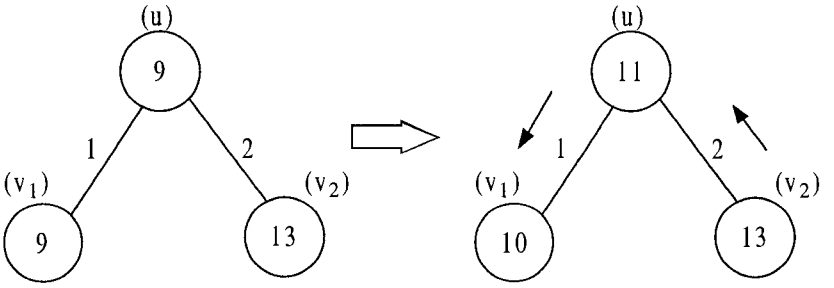

Figure 2.7: Backward Pathmax Rule.

On undirected graphs, researchers realized that pathmax rule (a) can also be used in the reverse direction, to update the heuristic value of the parent [12], since the parent can be seen as a child and the child can be seen as a parent. We call this reverse update rule the *backward pathmax rule*. They collectively call the original pathmax rules and the backward pathmax rule the *BPMX* update rules, which stands for bi-directional pathmax. The BPMX rules were introduced to speed up IDA* search when using inconsistent heuristics [12]. In Figure 2.7, we are expanding node u, and nodes v_1 and v_2 are its two neighbors. According to the backward pathmax rule, $h(v_2)$ propagates up and updates $h(u)$ to 11. Then according to pathmax rule (a), $h(u)$ propagates down and updates $h(v_1)$ to 10. In

its application to IDA*, only rule (a) and the backward pathmax rule were used, and the rules were used between the expanding node and one of its children that is on the current path being explored. In application to A*-like algorithms, we will perform backward pathmax updates from all children when we detect an inconsistency in one child during a normal expansion operation, which reduces much of the overhead of BPMX. All three rules can be used in A*-like algorithms.

In A*-like algorithms, if an h update occurs, BPMX can be performed to multiple levels to propagate the update further before resuming the normal execution of A*. In some cases, one-level BPMX outperforms multiple-level BPMX; in some cases, multiple-level BPMX outperforms one-level BPMX. There is no single optimal policy [41].

2.13 Delay Algorithm

Delay [41] is a new admissible variant of A* developed by Sturtevant during the same time as the research work of this thesis was performed.

The pseudocode of Delay is shown in Algorithm 11. Delay can be based on A* or B′, and it uses an additional priority queue called DELAY that stores the re-opened but not yet expanded nodes (line 44). There are two major features of Delay: (1) it requires that after a new expansion (i.e. not a re-expansion), there can be at most k re-expansions (lines 16-23), where k is a parameter; and (2) it chooses the minimum-g node among the re-opened nodes (line 5). The parameter k can be some constant, or a function of (an estimate of) N, for example $log(N)$ or $\sqrt{N}$. The loop in lines 7-10 is for maintaining admissibility. The motivation for adding a DELAY queue is the following. The exponential node expansions in the worst-case examples of A* are largely due to re-expansions, so if we separate the reopened nodes and restrict the number of re-expansions per each new node expansion, then we can possibly improve the worst-case complexity.

When Delay is based on B′ (i.e. uses pathmax rules) and $k = O(\sqrt{N})$, Sturtevant has proved that the worst-case runtime complexity is $O(N^{1.5})$ when using an inconsistent heuristic [41], which improves the state-of-the-art worst-case complexity of the A* family.

Algorithm 11 Delay

```
1: procedure DELAY(e,h,s,GOAL,k)
2:     g(s) ← 0
3:     CLOSED ← ∅
4:     OPEN ← {s}                                          ▷ Top node has min f
5:     DELAY ← ∅                                           ▷ Top node has min g
6:     while OPEN ≠ ∅ do
7:         while PEEK(OPEN) ∈ GOAL and g(PEEK(DELAY)) < g(PEEK(OPEN)) do
8:             delayTop ← POP(DELAY)                       ▷ Get min-g node
9:             EXPAND(delayTop,e,h)
10:        end while
11:        u ← POP(OPEN)                                   ▷ Get min-f node
12:        CLOSED ← CLOSED ∪ {u}
13:        if u ∈ GOAL then
14:            return BUILD-PATH(u)
15:        end if
16:        EXPAND(u,e,h)
17:        for i = 1 to k do        ▷ One new expansion followed by at most k re-expansions
18:            if DELAY = ∅ then
19:                break
20:            end if
21:            du ← POP(DELAY)
22:            EXPAND(du,e,h)
23:        end for
24:    end while
25:    return NIL
26: end procedure

27: procedure EXPAND(u,e,h)
28:    for each node v ∈ Neighbors[u] do
29:        h(v) ← max(h(v), h(u) − e(u,v))                 ▷ PathMax rule (a), optional
30:    end for
31:    h(u) ← max(h(u), min_{v∈Neighbors[u]}(h(v) + e(u,v)))   ▷ PathMax rule (b), optional
32:    for each node v ∈ Neighbors[u] do
33:        if v ∉ CLOSED ∪ OPEN ∪ DELAY then
34:            g(v) ← g(u) + e(u,v)
35:            f(v) ← g(v) + h(v)
36:            parent(v) ← u
37:            OPEN ← OPEN ∪ {v}
38:        else if g(v) > g(u) + e(u,v) then
39:            g(v) ← g(u) + e(u,v)
40:            f(v) ← g(v) + h(v)
41:            parent(v) ← u
42:            if v ∈ CLOSED then
43:                CLOSED ← CLOSED − {v}
44:                DELAY ← DELAY ∪ {v}
45:            end if
46:        end if
47:    end for
48: end procedure
```

Chapter 3

The Literature in Perspective

In this chapter we will clarify a minor error in the original B′ paper, show that the two pathmax rules can be executed in any order, and point out a falsely claimed property of B′.

3.1 Pathmax Formulas

Mero's paper introduced algorithm B′ as a modified version of algorithm B, by introducing the two pathmax rules. The original paper presents the formulas in the following way [33]:

(a) For each successor node m of the selected node n, if $h(m) < h(n) - e(n,m)$, then set $h(m) \leftarrow h(n) - e(n,m)$.

(b) Let m be the successor node of n for which $h(n) + e(n,m)$ is minimal. If $h(n) < h(n) + e(n,m)$, then set $h(n) \leftarrow h(n) + e(n,m)$.

Rule (a) updates the successors' heuristic values, and rule (b) updates the parent's heuristic value. Note that rule (a) is correct, but rule (b) is not. For rule (b), obviously $h(n) < h(n) + e(n,m)$ holds as long as the edge cost $e(n,m)$ is positive. Therefore for problems with positive edge costs, $h(n)$ can be increased to infinity by repeatedly applying the second rule.

The correct second pathmax rule, as we understand it, will be as follows: let m be the successor node of n for which $h(m) + e(n,m)$ is minimal. If $h(n) < h(m) + e(n,m)$, then set $h(n) \leftarrow h(m) + e(n,m)$. This is what is shown in Algorithm 9.

The reasoning behind the corrected second rule is the following. If n is not a goal node, then the minimum-cost path from n to a goal must go through one of n's successors, say m. Then the optimal distance from n to the goal is $dist(n, goal) = e(n,m) + dist(m, goal)$. Since the heuristic is admissible, we have $h(m) \leq dist(m, goal)$, and so $dist(n, goal) \geq e(n,m) + h(m)$. Thus setting $h(n)$ to the minimum of $h(n)$ and $e(n,m) + h(m)$ always preserves admissibility.

3.2 The Order of Execution of the Two Rules

Rule (a) is applied between the parent node and each of its individual child, and rule (b) is applied among the parent node and all of its children at once. Although Mero did not discuss it, we want to ask if the two rules interact with each other. If they do, is there an optimal sequence of executing them? We show here that the two rules do not interact with each other at all, so the order of execution does not matter.

Denote the selected node as n, and its successors as c_i. Assume rule (a) is active, so there is at least one successor node m with $h(m) < h(n) - e(n,m)$, and its heuristic value will be updated to $h'(m) = h(n) - e(n,m)$. Before rule (a) is applied, rule (b) cannot cause a heuristic update on n, since rule (b) can increase $h(n)$ only when $h(n) < min_i(h(c_i) + e(n,c_i))$ but $h(n) \not< h(m) + e(n,m)$. After rule (a) is applied, $h(n) = h'(m) + e(n,m)$, so rule (b) still cannot cause a heuristic update on n. Therefore we can say that rule (b) is inactive when rule (a) is active.

Now assume rule (b) is active, so $h(n) < min_i(h(c_i) + e(n,c_i))$. The heuristic value of n will be updated to $h'(n) = min_i(h(c_i) + e(n,c_i))$. Then for any successor c_i, we have $h(n) < h(c_i) + e(n,c_i)$ and $h'(n) \leq h(c_i) + e(n,c_i)$, so it is not possible to use rule (a) to update the successors' heuristic values before or after rule (b) is applied. Therefore we can say that rule (a) is inactive when rule (b) is active.

This observation implies that applying the two rules in any order is correct.

3.3 B′ Can Do More Node Expansions than B

Mero's paper analyzed several properties of algorithm B′. Theorem 3.3 in the paper reads as follows: "For every graph, algorithm B′ expands each node at most as many times as algorithm B, if both algorithms resolve ties in the same way." This theorem implies that given any problem instance, B′ will perform no more node expansions than B, as long as they break ties in the same way. The node selection mechanisms of A*, B and B′ in the original papers treat ties of f values arbitrarily, only requiring that goal nodes are preferred.

In the papers of A* and B, the tie-breaking mechanism is described as a function that takes two nodes with equal f values, and decides on which to expand first based on their current f, g and h values as well as whether they are goal nodes. This meaning has been the standard description in the literature. However, if we understand "resolve ties in the same way" in this sense, the above theorem is not correct. To disprove it, we will use a small counter example, which is shown in Figure 3.1. This graph is G_3 in Martelli's example graph family. In our simulation both algorithm B and B′ will break ties of nodes with equal f by preferring the one with larger g value (when all $f \geq F$). In this setting, algorithm B performs 4 node expansions, but algorithm B′ performs 5 node expansions. The sequence of node expansions of B is shown in Table 3.1, and that of B′ is shown in Table 3.2. Both tables show the f value and g value of each node, followed by the h value. As before, the node

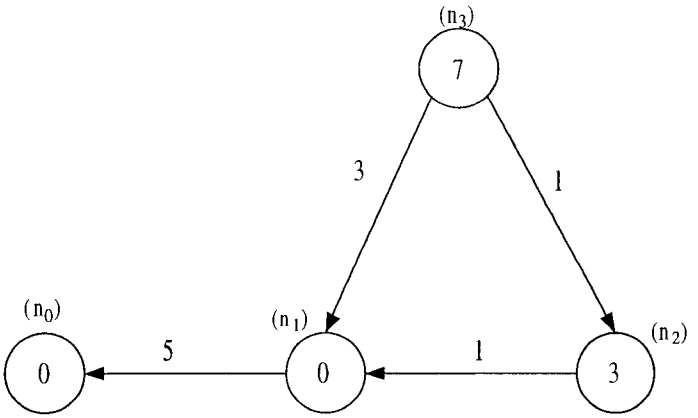

Figure 3.1: G_3 by Martelli's Scheme.

n_3	n_2	n_1	n_0	F
7,0,7*	-	-	-	0
[7,0,7]	4,1,3*	3,3,0	-	7
[7,0,7]	[4,1,3]	2,2,0*	-	7
[7,0,7]	[4,1,3]	[2,2,0]	7,7,0*	7
[7,0,7]	[4,1,3]	[2,2,0]	[7,7,0]	7

Table 3.1: Execution of B on Fig 3.1 (f,g,h shown).

n_3	n_2	n_1	n_0	F
7,0,7*	-	-	-	0
[7,0,7]	7,1,6	7,3,4*	-	7
[7,0,7]	7,1,6*	[8,3,5]	8,8,0	7
[7,0,7]	[7,1,6]	7,2,5*	8,8,0	7
[7,0,7]	[7,1,6]	[7,2,5]	7,7,0*	7
[7,0,7]	[7,1,6]	[7,2,5]	[7,7,0]	7

Table 3.2: Execution of B′ on Fig 3.1 (f,g,h shown).

selected for expansion is marked by a *, and closed nodes are bracketed.

An assertion that the proof relies on is crucial. At the bottom of page 21, it says "Notice, that algorithms B and B′ expand each node the first time in the same order". This is not true as seen in the above example. The sequence of first-time node expansions for B is $n_3 \rightarrow n_2 \rightarrow n_1 \rightarrow n_0$, but that for B′ is $n_3 \rightarrow n_1 \rightarrow n_2 \rightarrow n_0$.

It is possible that this assertion is a presumption and is part of what the author means by "resolve ties in the same way". If that is the case, a complex tie-breaking mechanism for B′ is likely needed.

Experimental results in Chapter 6 confirm that B′ can perform more node expansions than B when using inconsistent heuristics and using the same tie-breaking rule, if we understand "resolve ties in the same way" in the first-mentioned sense.

Chapter 4

Bounding A*'s Node Expansions

It has been demonstrated that A* may perform exponentially many expansions as a function of the number of distinct nodes expanded, while B and B′ perform at most quadratic expansions [32, 33]. However, exponential expansion has not been reported in real-world applications that use A*. Looking back at the G_5 example in Figure 2.4, we can observe that it has two uncommon characteristics: the nodes are fully connected, and the edge weights and heuristic values grow exponentially in the graph size. Are these the necessary conditions for the worst-case scenarios to occur when using A*?

Before going further, we define a few terms. Given two real numbers x and y, if $x = m * y$ and m is an integer, then we call y a factor of x. If a set of real numbers share some factors in common, we define the greatest among them as the *greatest common factor* (gcf). Given a set of possible edge costs, we define Δ as the greatest common factor of all the edge weights. During the search, a node can have many different g values over time, all of which are integer combinations of the edge weights. Then the difference between any two g values is still an integer combination of the edge weights. By definition, Δ is a lower bound of the minimum possible decrement of the g value of any node. When dealing with irrational numbers, Δ may not be well-defined. Since computers are finite-precision, we will only consider edge weights that are rational. To compute Δ for rational edge weights, we multiply the edge weights by a scalar to make them integers, then compute the greatest common divisor (gcd) of them, and divide the gcd by the scalar. For example, if the possible edge weights are $\{1, 2, 3\}$, then $\Delta = 1$; if the possible edge weights are $\{1, 1.2, 1.8\}$, then $\Delta = 0.2$. We define V as the set of expanded nodes, and $N = |V|$ is the size of V.

First, we show that the number of node expansions implies a lower bound on the maximum heuristic value among the nodes expanded, and implies a lower bound on the solution cost.

Theorem 1. On a graph where Δ is well-defined and edge weights are positive, if A* performs $\phi(N)$ node expansions ($\phi(N) > N$), then there must be a node with heuristic value of at least $\Delta * (\phi(N) - N)/N$, and the optimal solution path must have a cost of at least $\Delta * (\phi(N) - N)/N$.

Proof. If there are $\phi(N)$ total expansions by A*, then the total re-expansions are $\phi(N) - N$. By the

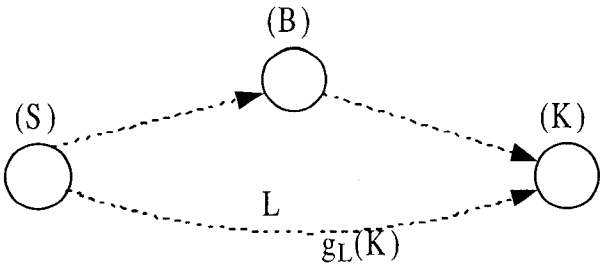

Figure 4.1: First and Last Explored Path.

pigeon-hole principle [1] there must be a node, say K, with at least $(\phi(N) - N)/N$ re-expansions. Each re-expansion corresponds to a re-opening, and each re-opening must decrease the g value of K by at least Δ, so after this process the g value of K has been reduced by at least $\Delta * (\phi(N) - N)/N$.

Figure 4.1 will serve to illustrate how the lower bound of the maximum heuristic value is derived. We will show that node B has a heuristic value of at least $\Delta * (\phi(N) - N)/N$. Suppose S is the start node, node K is the node with at least $(\phi(N) - N)/N$ re-expansions, the first expansion of K is via the path L, the last expansion of K is via the path that goes through B, and B is the node on that path with maximum f value. We denote the f and g values of K via path L as $f_L(K)$ and $g_L(K)$, and the f and g values of K via the last explored path as $f_{last}(K)$ and $g_{last}(K)$, respectively. If K is first expanded via L, then $f(B) \geq f_L(K)$. Thus we have these facts:

$$f(B) = g(B) + h(B) \tag{4.1}$$

$$f(B) \geq f_L(K) \tag{4.2}$$

$$f_L(K) = g_L(K) + h(K) \tag{4.3}$$

$$g(B) < g_{last}(K), \text{ as K's parent on the last explored path is B and edge weights} > 0 \tag{4.4}$$

$$g_L(K) - g_{last}(K) \geq \Delta * (\phi(N) - N)/N, \text{ as discussed earlier} \tag{4.5}$$

So,

$$h(B) = f(B) - g(B), \text{ by Eqn 4.1} \tag{4.6}$$

$$\geq f_L(K) - g(B), \text{ by Eqn 4.2} \tag{4.7}$$

$$\geq g_L(K) + h(K) - g(B), \text{ by Eqn 4.3} \tag{4.8}$$

$$> g_L(K) - g_{last}(K) + h(K), \text{ by Eqn 4.4} \tag{4.9}$$

$$> g_L(K) - g_{last}(K), \text{ since } h(K) \geq 0 \tag{4.10}$$

$$\geq \Delta * (\phi(N) - N)/N, \text{ by Eqn 4.5} \tag{4.11}$$

Therefore h(B) is at least $\Delta * (\phi(N) - N)/N$. Since A* expanded node B before the goal, the optimal solution cost $g^*(goal)$ must be at least $f(B)$, which is at least $\Delta * (\phi(N) - N)/N$. □

[1] If there are $m * n + 1$ pigeons in n holes, there must be a hole with at least $m + 1$ pigeons [14].

From Theorem 1 it follows that for A* to expand 2^N nodes, there must be a node with heuristic of at least $\Delta * (2^N - N)/N$, and for A* to expand N^2 nodes, there must be a node with heuristic at least $\Delta * (N - 1)$.

Next, we use this result to show that the worst-case runtime complexity is closely related to the optimal solution cost, and when edge weights are constants that do not grow with the problem size, the worst-case runtime complexity of A* is quadratically bounded.

Theorem 2. On a graph where Δ is well-defined and edge weights are positive, if the optimal solution cost $g^*(goal) \leq \lambda(N)$, then $\phi(N)$ is at most $N + N * \lambda(N)/\Delta$.

Proof. Using the result of Theorem 1,

$$\Delta * (\phi(N) - N)/N \leq g^*(goal) = \lambda(N) \tag{4.12}$$

Which implies,

$$\phi(N) \leq N + N * \lambda(N)/\Delta \tag{4.13}$$

□

Corollary 1. If the edge weights are a fixed finite set of rational constants that do not change with the problem size and the maximum edge weight is m, then the total number of node expansions is at most $N + N * m * (N - 1)/\Delta$.

Proof. Since the edge weights are a fixed finite set of rational constants and independent of the problem size, Δ is a well-defined constant. Since the edge weights are constants that are independent of the problem size, there are at most $N - 1$ edges in the solution path and the optimal solution cost $g^*(goal)$ is at most $m * (N - 1)$. Using the result of Theorem 2,

$$\phi(N) \leq N + N * \lambda(N)/\Delta \leq N + N * m * (N - 1)/\Delta \tag{4.14}$$

□

These results show that on regular problems where the edge weights are a fixed finite set of rational constants independent of the problem size, A* does not have an asymptotic disadvantage compared to B and B′. Using A* with inconsistent heuristics in these domains has a better runtime complexity upper bound than previously thought.

If the graph is a tree, then there are no multiple paths to a node (transpositions), and re-opening does not occur. If the graph is a square grid with unit edge weights and the graph size is N, then the optimal solution path cost is at most $\sqrt{2N}$, and the worst-case runtime complexity of A* using inconsistent heuristics is $N + N * \sqrt{2N}$. For many problems with rational constant edge weights, the optimal solution cost grows asymptotically slower than N, such as $\ln(N)$. In these problems, A* has a better-than-$O(N^2/\Delta)$ worst-case complexity.

Chapter 5

The DP Algorithm

Avoiding re-expansions is the key in improving A*-like algorithms' performance when using inconsistent heuristics. Re-expansions are caused by re-openings of nodes. Re-opening a node is the event that a closed nodes' g value is being reduced by one of its neighbor nodes. When searching on an undirected graph with an inconsistent heuristic, if we propagate the g values in both the child-to-parent and parent-to-child directions when expanding a node, then we can eliminate many re-openings and re-expansions. This is the motivation behind the DP (Dual Propagation) algorithm. It aims at speeding up the discovery of optimal paths. In some sense, the algorithm's propagation method is symmetric to BPMX propagation, since BPMX propagates the h values in both directions, and DP propagates the g values in both directions. By design, DP is only applicable to undirected graphs, which is similar to BPMX.

5.1 The DP Update Rule

The DP update rule for g value propagation is shown in Figure 5.1. The h value is inside a node and the current g value is shown beside a node, and the edge cost is shown beside an edge. Node A is selected for expansion because it has the lowest f value at the moment. According to DP, $g(C)$ propagates up and reduces $g(A)$ to 7, then $g(A)$ propagates down as usual and reduces $g(B)$ to 8. The solid arrow heads show the directions of the g propagation and the parent-child relationships after the propagation.

5.2 The Algorithm

The pseudocode for the DP algorithm is shown in Algorithm 12. The changes from A* appear in lines 11-17, where we propagate all the neighbors' g values to the node being expanded, and change its parent if appropriate. In actual implementation, we only need to do the backward g value propagation when we detect that there is a neighbor who can reduce the g value of the node being expanded.

Algorithm 12 DP

```
1: procedure DP(e,h,s,GOAL)
2:     g(s) ← 0
3:     CLOSED ← ∅
4:     OPEN ← {s}
5:     while OPEN ≠ ∅ do
6:         u ← EXTRACT-MIN-F(OPEN)                          ▷ Get min-f node
7:         CLOSED ← CLOSED ∪ {u}
8:         if u ∈ GOAL then
9:             return BUILD-PATH(u)
10:        end if
11:        for each node v ∈ Neighbors[u] do                ▷ Reverse g Propagation
12:            if v ∈ CLOSED ∪ OPEN and g(u) > g(v) + e(v,u) then
13:                g(u) ← g(v) + e(v,u)
14:                f(u) ← g(u) + h(u)
15:                parent(u) ← v
16:            end if
17:        end for
18:        for each node v ∈ Neighbors[u] do
19:            if v ∉ CLOSED ∪ OPEN then
20:                g(v) ← g(u) + e(u,v)
21:                f(v) ← g(v) + h(v)
22:                parent(v) ← u
23:                OPEN ← OPEN ∪ {v}
24:            else if g(v) > g(u) + e(u,v) then
25:                g(v) ← g(u) + e(u,v)
26:                f(v) ← g(v) + h(v)
27:                parent(v) ← u
28:                if v ∈ CLOSED then                       ▷ Reopen a closed node
29:                    CLOSED ← CLOSED − {v}
30:                    OPEN ← OPEN ∪ {v}
31:                end if
32:            end if
33:        end for
34:    end while
35:    return NIL
36: end procedure
```

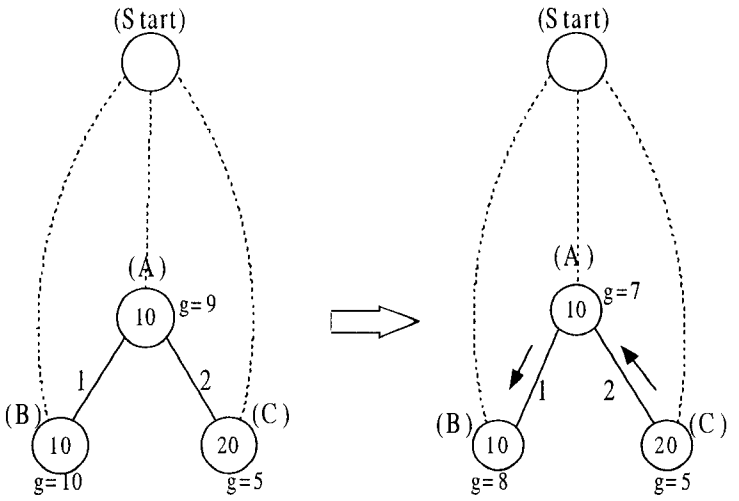

Figure 5.1: DP Rule for g Value Propagation.

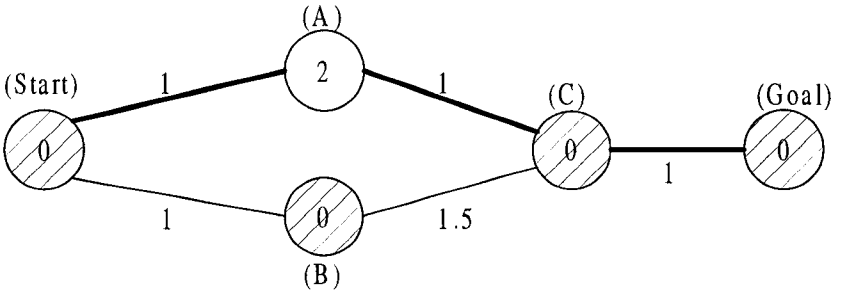

Figure 5.2: The Solution Path May Contain Open Nodes.

5.3 Properties of DP

In some cases, the solution path found by DP contains open nodes, this is illustrated in the example in Figure 5.2. Again, edge costs are shown next to the edges, and h values are shown inside nodes. The optimal solution path is $Start \rightarrow A \rightarrow C \rightarrow Goal$. But since $h(A)$ is high, the path $Start \rightarrow B \rightarrow C$ is explored first. When C is being expanded, the algorithm discovers that $g(A)$ can reduce $g(C)$, and assigns A as the parent of C, but A remains in the OPEN list. After C is expanded, the goal is found and immediately expanded, since $f(Goal) = f(A)$ and the algorithm always prefers goal nodes when breaking ties. Keeping nodes like A in OPEN is not necessary, but doing this is safe and avoids unnecessary expansions.

Note that in the above simple example DP outperforms A*. DP expands nodes in the following order: $Start \rightarrow B \rightarrow C \rightarrow Goal$. A* explores a suboptimal path first, then discovers the optimal path and has to re-expand node C. The order of node expansions by A* is: $Start \rightarrow B \rightarrow C \rightarrow A \rightarrow C \rightarrow Goal$.

Theorem 3. When using a consistent heuristic, DP is identical to A*.

Proof. If the heuristic is consistent, when a node is selected for expansion in A*, its g value is guaranteed to be optimal [16]. Therefore, the reverse g propagation in DP will never occur, and DP is identical to A*.

Theorem 4. If the heuristic function is admissible, then DP guarantees finding the optimal solution if one exists.

Proof. The proof is identical to the admissibility proof of A*. Suppose DP finds suboptimal paths, then $f(G) > f^*(G) = g^*(G)$. For any optimal path, there must be some unexpanded nodes, otherwise this optimal path will be found and used as the solution path. Consider the first open node O on this path. We have

$$\begin{aligned} f(O) &\geq f(G), \text{ since } O \text{ is open} \\ f(O) &= g(O) + h(O) \\ &= g^*(O) + h(O), \text{ since } O \text{ is first open node on an optimal path} \\ &\leq g^*(G), \text{ since } h \text{ is admissible} \end{aligned}$$

Combining the above two inequalities, we have $f(G) \leq g^*(G)$, which contradicts $f(G) > g^*(G)$ that we have before. Thus the assumption is wrong, and DP always finds an optimal solution if one exists. □

In the cases where the solution path found by DP contains open nodes, this theorem implies that these open nodes all have $f = g^*(G)$.

Since DP tries to correct g values early and avoid mistakes further down in the search tree, it is expected that DP performs no more (often less) expansions than A* in most cases, if DP performs backward propagation only when necessary. This is also hinted by the experimental results in the next chapter.

Chapter 6

Experiments

We perform experiments on two domains: pathfinding and the TopSpin puzzle. We will compare the performance of numerous algorithms using inconsistent heuristics. A* with a consistent heuristic is used as a baseline reference.

6.1 Inconsistency Measures

Zahavi et. al. introduced two measures of inconsistency on undirected graphs: IRE and IRN [43]. They are defined as follows:

(a) **Inconsistency rate of an edge (IRE)**: For an edge $e = (m, n)$, IRE$(h, e) = |h(m) - h(n)|$. The IRE of the entire graph is the average IRE over all edges.

(b) **Inconsistency rate of a node (IRN)**: For a node n, IRN$(h, n) = max_{m \in adj(n)} |h(m) - h(n)|$. The IRN of the entire graph is the average IRN over all nodes. Here $adj(n)$ denotes the set of neighbors of n.

We will use two somewhat different measures on undirected graphs:

(a) **Percentage of nodes with inconsistency (PNI)**: For a node n, if $max_{m \in adj(n)}((h(n) - h(m)) - e(m, n)) > 0$, then n has an inconsistency, and the max value is the inconsistency rate of n. PNI is the percentage of nodes that have an inconsistency.

(b) **Average inconsistency of nodes (AIN)**: AIN is the average inconsistency rate over the set of nodes that have an inconsistency.

6.2 The Pathfinding Domain

We use a collection of 116 maps from commercial games, all scaled to be 512 by 512 in size. There are blank spots and obstacles on the maps. Without obstacles and boundaries, there are 8 possible movements from a position, 4 cardinal moves, and 4 diagonal moves. Cardinal moves have cost 1, and diagonal moves have cost $\sqrt{2}$. The eight possible movements are illustrated in Figure 6.1.

The problems are categorized into different "buckets", whose bucket number is the optimal solution length divided by 4 and rounded down. There are 1160 randomly-generated problem instances in each bucket (the last few buckets have slightly fewer problems),[1] ranging from easy (bucket ID = 0) to hard (bucket ID = 127). The maps and problem instances are provided by Nathan Sturtevant. The pathfinding experiments are performed on computers with Intel P4 3.4GHz CPUs and 1GB memory.

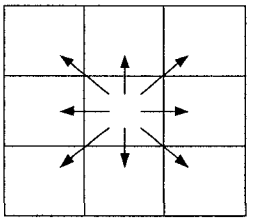

Figure 6.1: The Eight Directions of Movement.

Octile distance is an easy-to-compute consistent heuristic in this domain. If the distances along x and y coordinates between two points are (dx, dy), then the octile distance between them is $\sqrt{2} * min(dx, dy) + |dx - dy|$. In essence, this is the optimal distance between the two points if there were no restrictions from obstacles or boundaries.

To generate an inconsistent heuristic that dominates the octile heuristic for a map, we randomly choose H target points, and compute the optimal distance to every other point from each of them. The optimal distances corresponding to each of the H targets are stored in separate tables, which are a form of pattern database. For each target t, we can use its table to generate an admissible heuristic for any two points a and b, using $h(a,b) = max(|dist(a,t) - dist(b,t)|, octile(a,b))$. Because $dist(a,b) + dist(b,t) \geq dist(a,t)$ for any a, b and t, we have $dist(a,b) \geq dist(a,t) - dist(b,t)$, so $|dist(a,t) - dist(b,t)|$ is an admissible heuristic for the distance between a and b, and it is naturally consistent since it is generated according to the definition of consistency. The octile heuristic is also admissible and consistent. Taking the maximum of two admissible heuristics produces an admissible heuristic, and taking the maximum of two consistent heuristics produces a consistent heuristic. Thus our resulting heuristic in each table is admissible and consistent. $|dist(a,t) - dist(b,t)|$ can sometimes produce a heuristic value higher than the octile heuristic. For example, this can happen when b is on the optimal path from a to t, and the exact distance from a to b is larger than the octile distance. However, computing the difference does not frequently produce a larger value than the octile heuristic and sometimes produces smaller values. The use of the octile heuristic as a lower bound is to guarantee that the new heuristic dominates the octile heuristic.

At each query, we return an h value from a randomly chosen table, thus producing an inconsistent heuristic that dominates the octile heuristic. In our experiment, we use 10 (and 20) randomly selected target points on each map and thus generate 10 (and 20) lookup tables (PDBs). Since results on 10

[1]The last 41 buckets have fewer than 1160 problems, among which the last 12 buckets have problems ranging from 800 to 1000. This is due to the difficulty in generating hard problems.

PDBs and 20 PDBs are largely the same, we only present the result on 10 PDBs here. We also use the max of all PDBs and the octile distance as a heuristic for A*. This max-of-10 heuristic is consistent since each individual PDB is consistent. A* with the max-of-10 heuristic is expected to give the best performance in terms of the number of node expansions and is used as a bounding reference.

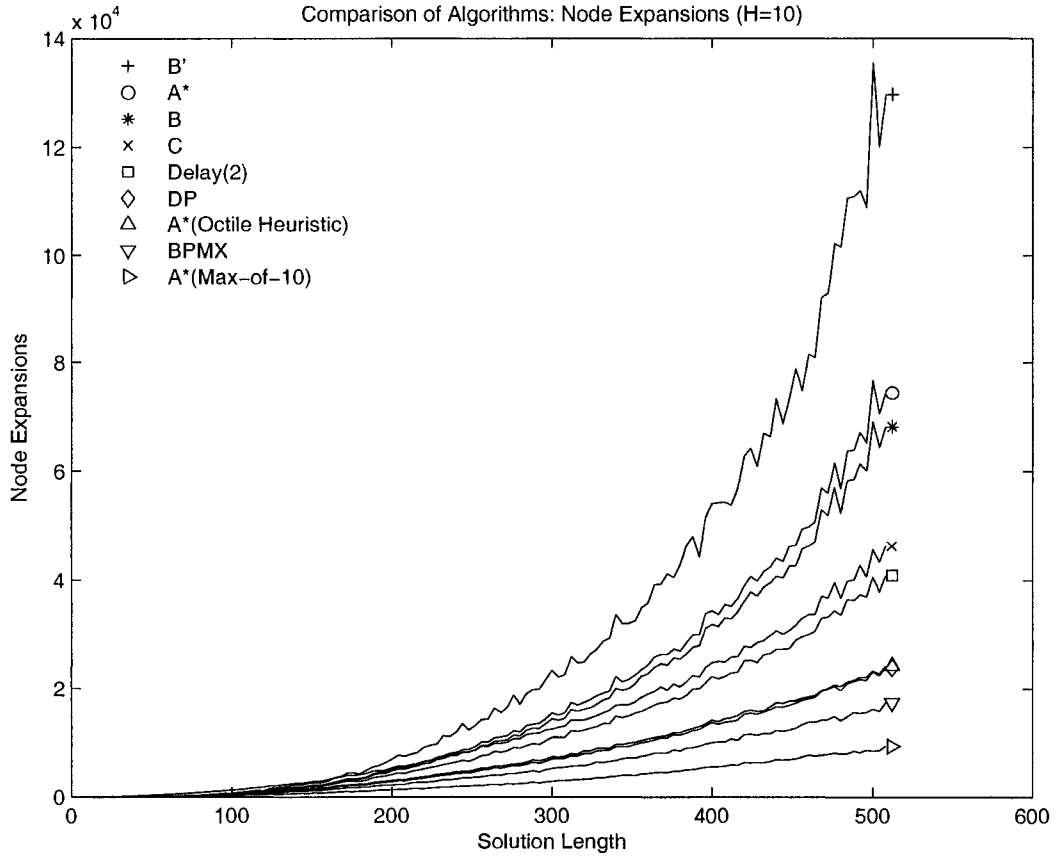

Figure 6.2: Node Expansions in Pathfinding.

By design, the random heuristic dominates the octile heuristic. We estimated that the random heuristic is higher than the octile heuristic in only 25% of the nodes. For the random heuristic, PNI= 16.20% and AIN= 35.97. The number of node expansions by algorithms A*, B, B′, C, Delay(2),[2] DP and BPMX (1-level propagation) are plotted in Figure 6.2 and their CPU times are plotted in Figure 6.3. Additionally, A* using the octile heuristic and A* using the max-of-10 heuristic are plotted for reference. When counting node expansions, the reverse h propagation in BPMX and reverse g propagation in DP are counted as one expansion as well. The x-axis in both graphs is the solution length. The y-axis is the number of node expansions and time in seconds, respectively. In the node expansions graph, the lines appear in the following descending order: B′, A*, B, C, Delay(2), DP, A*(Octile), BPMX, and A*(Max). B′ performs the most node expansions,

[2] $k = 2$ performs among the best in the choices of k we tried in both domains.

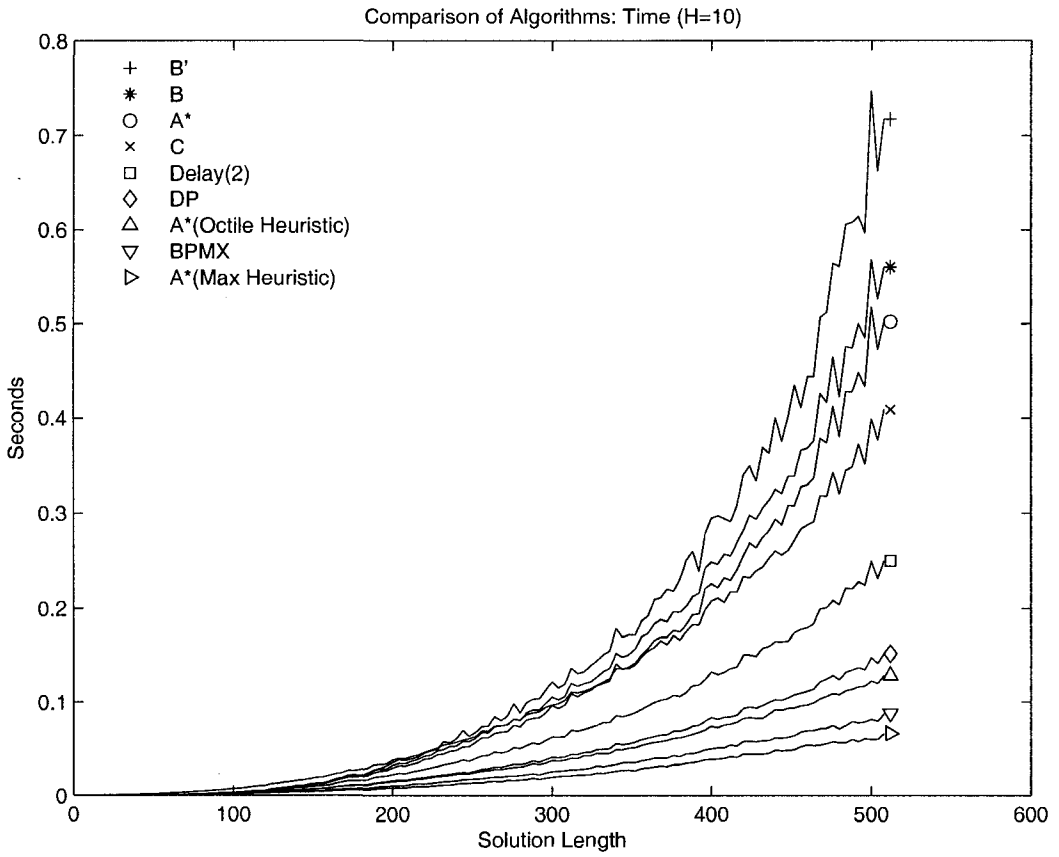

Figure 6.3: Time in Pathfinding.

which confirms that B′ can perform more node expansions than B as discussed in Chapter 3. C performs better than A*, Delay(2) performs better than C, and A*(Octile) and DP perform better than Delay(2). A*(Max) performs best, as expected. Among the lines using the inconsistent heuristic, BPMX is best and its number of node expansions is 27% smaller than that of A*(Octile) on the hardest problems, and is less than twice that of A*(Max), the best possible result. The time in seconds is plotted in Figure 6.3. It is quite similar to the node expansions graph, except for a few differences. Algorithm B uses slightly more time than A*, despite performing fewer node expansions than A*. This is because B occasionally needs to extract the node with minimum g value from the OPEN list, which is sorted by minimum f. The line for Delay(2) moves up compared to that in the node expansions graph, because it has to maintain an additional delay list, which incurs some overhead. The line of A*(Max-of-10) also moves up, because it performs multiple PDB lookups per node.

The node expansions in the hardest problems (the last bucket) are also categorized in Table 6.1. First expansions is the number of times a new node is expanded, and reverse expansions refers to the reverse h propagations in BPMX and the reverse g propagations in DP. It turns out that the number of first expansions of B′ is approximately the same as that in A* and B (using an inconsistent heuristic), but B′ performs many more re-expansions. C performs slightly more first expansions than A* and B, but it is able to largely reduce the re-expansions. Delay performs more first expansions than A* because it may need to expand some nodes with $f > g^*(Goal)$ due to the delay, but it is able to bound the number of re-expansions. DP dramatically reduces the re-expansions at the expense of a few reverse expansions. BPMX is able to dramatically reduce the first expansions as well as the re-expansions, at the expense of a few reverse expansions. The number of first expansions of BPMX is less than half that of A* with the octile heuristic, and is close to that of A* with the max-of-10 heuristic.

Algorithm	First Expansions	Re-expansions	Reverse Expansions	Sum	Seconds
A*(Octile Heuristic)	24221	0	0	24221	0.1288
A*(Max Heuristic)	9341	0	0	9341	0.0664
A*	17210	57183	0	74392	0.5027
B	17188	50963	0	68151	0.5602
B′	16660	112010	0	129680	0.7165
C	21510	24778	0	46288	0.4106
Delay(2)	19053	21886	0	40938	0.2618
DP	17784	9825	1319	28928	0.1648
BPMX	10195	4065	3108	17368	0.0887
BPMX(2)	9979	3462	5545	18986	0.0933
BPMX(3)	9997	3467	5854	19317	0.0935
BPMX(∞)	10025	3483	6207	19714	0.0938
DP+BPMX	10763	4742	4880	20385	0.0911
Delay(2)+BPMX	10374	4857	2934	18164	0.1129

Table 6.1: Categorized Node Expansions in the Hardest Pathfinding Problems.

As BPMX is most effective in the pathfinding experiment, we also experimented with combining Delay(2) and DP with BPMX, and the number of node expansions is shown in the last two rows of Table 6.1. It turns out that combining the algorithms cannot further improve the performance of BPMX in this domain, though the three algorithms' performances are very close to each other.

Furthermore, we experimented with multiple levels of BPMX propagation. BPMX propagation is purely a h value propagation. To avoid the overhead of generating new nodes while doing pure h propagation, we centered the propagation around closed nodes only. In the root level of BPMX update, if we find that a closed neighbor is able to update the h value of other nodes, or if a closed neighbor's h value has just been updated, then we push this neighbor into a queue. BPMX updates will be performed on these queued nodes, with their levels increased by 1 from the level they are enqueued. This propagation continues until the propagation has reached a preset level limit, or until there are no candidates for propagation (the queue is empty). In addition to the 1-level BPMX (normal BPMX), we tested with BPMX(2) (2-level BPMX), BPMX(3) and BPMX(∞). The results are shown in the bottom half of Table 6.1. Multiple levels of BPMX are able to slightly reduce the number of first expansions and re-expansions, but at the expense of more reverse expansions. There is no advantage doing this in our pathfinding experiments.

6.3 The TopSpin Domain

TopSpin is a combinatorial puzzle where the player is given a ring of numbers, and the goal is to sort them into increasing order. The only allowed operation is to flip a fixed length, continuous range of tokens into reverse order. The puzzle needs to specify two parameters T and K, where T is the total number of tokens, and K is the the allowed length of tokens to flip. Each flip has a unit cost. An example of a flipping operation of (8,4)-TopSpin is shown in Figure 6.4. [3] For a (T,K)-TopSpin, the flipping is solely determined by the mid-point of flipping, and so the branching factor is T. We generated 1000 random problems of the (T,K)-TopSpin puzzle for K=4 and T = 9...14. The start positions in these problems are generated by $20 * T$ random walk from the goal state, with the back move disabled. For each puzzle, a PDB was built, in which each entry is indexed by the first $\lfloor T/2 \rfloor$ tokens only and each entry stores the minimum path cost to reach that pattern from the goal state. Experiments in this domain are performed on cluster nodes with AMD Opteron 1GHz CPUs and 16GB memory. In our current implementation, no transpositions are eliminated except for not including a node's parent among its children.

The optimal solution path in this domain is short (normally with a cost no larger than the number of tiles), because the connectivity and branching factor is high. Therefore the range of possible heuristic values is narrow. Nodes with high inconsistency rates will be rare, no matter what method we use to generate the inconsistent heuristic. This is confirmed by the experiment results later.

[3] For ease of discussion, it is drawn as an array, but we can always take the ring and spread it out starting at the position of 0

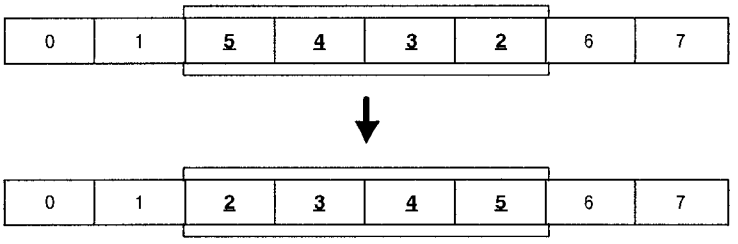

Figure 6.4: A Flipping Operation in (8,4)-TopSpin.

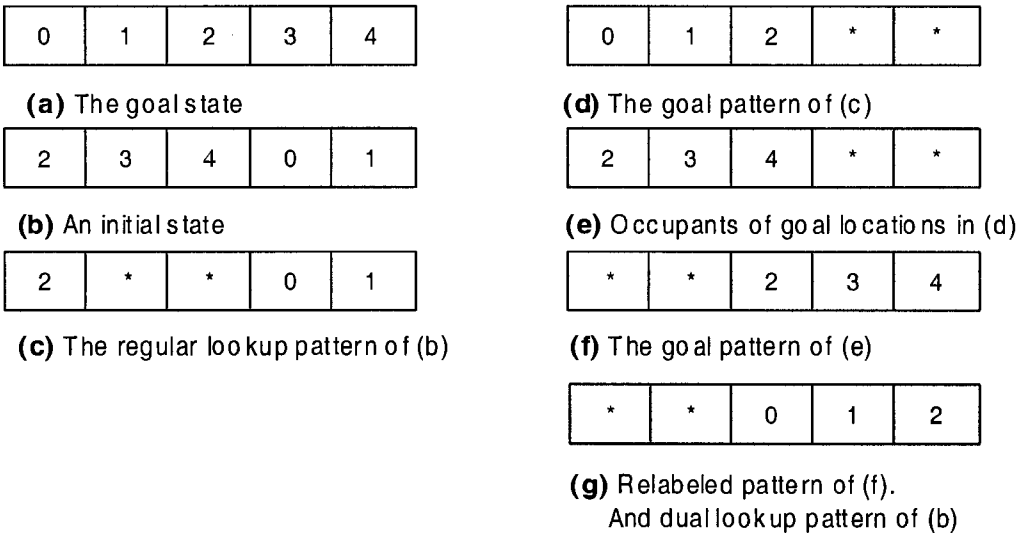

Figure 6.5: A Dual Lookup in (5,4)-TopSpin.

The state-of-the-art heuristic in the TopSpin domain is a pattern database with dual lookups [12]. Figure 6.5 explains the dual lookup in a (5,4)-TopSpin problem, and it is based on a figure in [12]. Part (a) shows the goal state, and part (b) is an initial start state. If we build a 3-piece PDB for the tiles {0,1,2}, then the regular lookup pattern for the initial state (b) is shown in (c). To solve the full problem we need to solve the subproblem of translating (c) into pattern (d). The regular PDB lookup of pattern (c) would be PDB[3][4][0], meaning that tile 0 is in location 3, tile 1 is in location 4, and tile 2 is in location 0. The goal locations of {0,1,2} are currently occupied by {2,3,4}, as shown in (e). To solve the full problem, we also need to move {2,3,4} into their goal locations, as shown in (f). If we just focus on tiles {2,3,4} and ignore the rest, then the names of the tiles can be relabeled without changing the subproblem. We want to relabel them such that their current pattern (e) becomes the goal pattern of {0,1,2}, which is (d). Thus we relabel $2 \rightarrow 0$, $3 \rightarrow 1$, and $4 \rightarrow 2$. Consequently, (f) is relabeled into (g). Because translating (g) into (d) requires the same cost as translating (e) into (f), they both provide a lower bound of the cost of solving the original problem. Now (g) is the dual lookup pattern of the initial state (b). The PDB lookup of pattern (g) is PDB[2][3][4], since tile 0 is in location 2, tile 1 is in location 3, and tile 2 is in location 4. Note that {2,3,4} are precisely the current occupants of the goal locations of {0,1,2}! The dual lookup returns a different heuristic value than the regular lookup, which can be higher. Even if a PDB stores consistent heuristic values, a dual lookup normally returns an inconsistent heuristic as the duals of two adjacent states may not be adjacent. More in-depth discussion of the dual lookup can be found in [12].

Compared to the normal heuristic, the dual heuristic is lower in 17% of the nodes generated in the search, equal in 5% of the nodes, and higher in 78% of the nodes. For the dual heuristic, PNI= 32.92% and AIN= 1.24. The average number of node expansions in TopSpin is shown in Figure 6.6. A* using a normal lookup is shown for reference. All other lines represent algorithms using a dual lookup. The x-axis is the puzzle size, and the y-axis is the number of node expansions. The figure shows that C using the dual lookup performs the most node expansions, followed by A* using the normal lookup, and then all the other algorithms using dual lookup, which all perform a similar number of node expansions. The time graph is shown in Figure 6.7. Suffering from paging (using hard disk to compensate for lack of physical memory) due to its larger search scope, the line for algorithm C is not shown. Since in our implementation of the TopSpin environment a dual lookup is about 3.5 time as expensive as a normal lookup, the line for A* using a normal lookup moves down noticeably, compared to that in the node expansions graph. Also, in this domain a reverse h propagation in BPMX and a reverse g propagation in DP are relatively inexpensive compared to a node expansion, but they are counted as node expansions, so the lines for DP and BPMX both move down noticeably in the time graph.

We categorized the node expansions in the (14,4)-TopSpin for all algorithms, including DP+BPMX and Delay(2)+BPMX and multiple levels of BPMX. The results are shown in Table 6.2. Among the

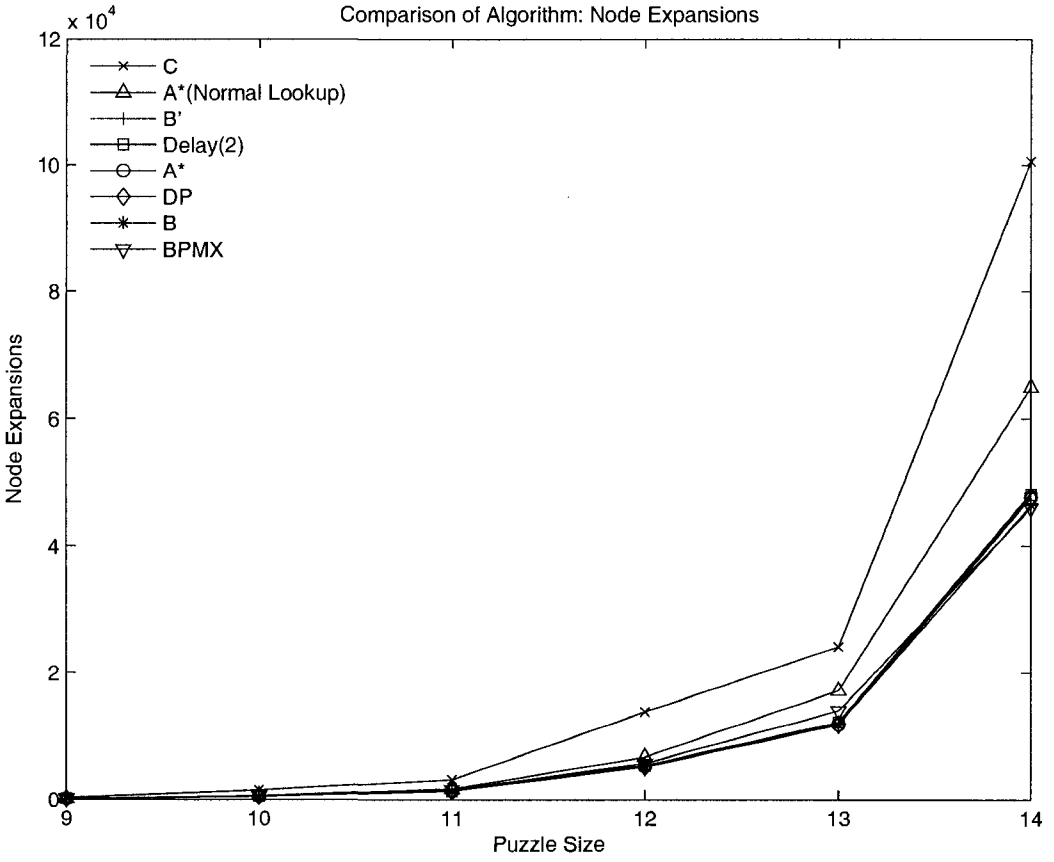

Figure 6.6: Node Expansions in TopSpin.

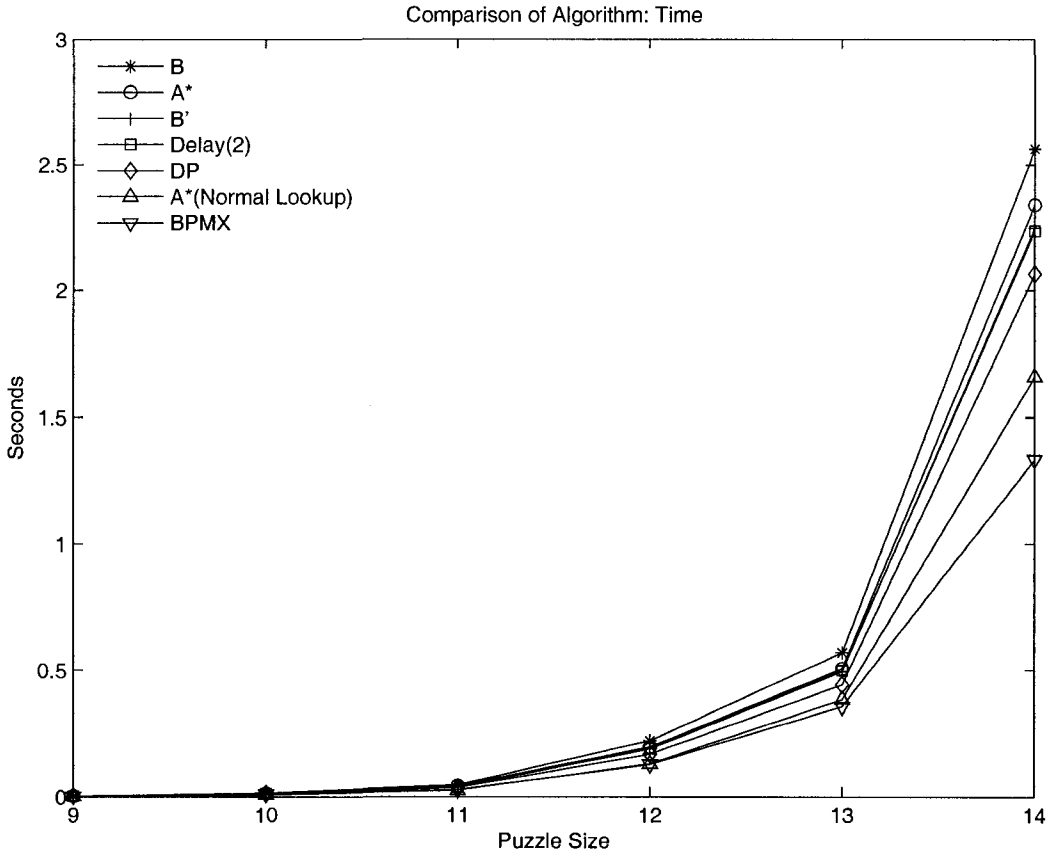

Figure 6.7: Time in TopSpin.

algorithms using the dual heuristic, the numbers of re-expansions of most algorithms are quite small. BPMX incurs many reverse expansions and re-expansions, but it is able to dramatically reduce the first expansions. Algorithm C performs the fewest re-expansions, but performs a large amount of first-expansions.

Algorithm	First Expansions	Re-expansions	Reverse Expansions	Sum
A*(Normal Lookup)	65014	0	0	65014
A*	47227	176	0	47403
B	45607	166	0	45773
B′	49454	208	0	49662
C	100550	50	0	100600
Delay(2)	48149	198	0	48347
DP	45635	159	84	45878
BPMX	23542	7019	15639	46200
BPMX(2)	24132	7242	16312	47686
BPMX(3)	24247	7277	16392	47916
BPMX(∞)	24247	7277	16392	47916
DP+BPMX	29361	7460	21935	58757
Delay(2)+BPMX	23995	7138	15903	47035

Table 6.2: Categorized Node Expansions in (14,4) Topspin.

DP+BPMX and Delay(2)+BPMX cannot improve the performance of BPMX. DP seems to have conflicts with BPMX. This is because this domain has a large branching factor, and the reverse g propagation reduces the g values of many nodes that were outside the search scope of BPMX and consequently brings them inside the search scope. This can be seen from the increased number of first expansions. Increasing the first expansions consequently increases the re-expansions and the reverse expansions.

The multiple level versions of BPMX cannot outperform the 1-level BPMX, and the extra levels of BPMX turn out to be pure overhead in our tests in this domain.

6.4 Conclusion

This chapter empirically compares various algorithms in two domains using inconsistent heuristics. According to the experiments, Delay and DP are substantial improvements to A* in the pathfinding domain, but less substantial in the TopSpin domain. BPMX has dominating improvements in both domains, but the improvement is marginal in the TopSpin domain due to its introduction of many re-expansions. Combining Delay or DP with BPMX cannot outperform BPMX in these two domains, and DP is somewhat conflicting with BPMX in the TopSpin domain. Using multiple levels of BPMX shows no advantage in the two tested domains. The performance of BPMX using a much better (by 25% and 78%, respectively) inconsistent heuristic exceeds that of A* using a regular consistent heuristic. Therefore if an inconsistent heuristic is available that is much better than the available consistent heuristics, the inconsistent heuristic is probably preferable with the help of BPMX, Delay

or DP. We also confirmed that B′ can perform more node expansions than B in the presence of inconsistency. Algorithm C has a significant advantage over A* in the pathfinding experiments, but has a significant disadvantage in the TopSpin experiments due to its larger search scope.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we have made clarifications and corrections to the research literature on inconsistent heuristics, particularly on algorithm B′. For A* algorithm using an inconsistent heuristic, we analyzed the relationship among the number of node expansions, the maximum heuristic value and the optimal solution cost. Based on these analysis, we inferred that if the maximum heuristic value or the minimum solution path cost is not an exponential function of the number of distinct expanded nodes, then the exponential blowup of A*'s worst-case cannot occur. Furthermore, if the edge costs are a fixed finite set of rational constants independent of the problem size, then the worst-case performance of A* is quadratically bounded. We also proposed a new technique called DP, that is effective when searching with inconsistent heuristics. We also performed comprehensive experiments on the pathfinding and TopSpin domains using inconsistent heuristics, and presented the first comparison among algorithms A*, B, B′, C, DP, Delay, and BPMX.

7.2 Future Work

There are several interesting directions of further research that can be seen from this dissertation, both in the technical sense and in the theoretical sense.

The pathmax rules can be generalized to longer than one step away, and more theoretical and empirical investigations into this will be worthwhile. As inconsistency can arise in learning real-time heuristic search, it will be interesting to incorporate the techniques like Delay, DP and BPMX into LRTA* and its variants. It is also promising to incorporate these techniques into memory-bounded algorithms such as SMA* when inconsistent heuristics are used, since SMA* is able to tackle much larger problems than A* and its variants, yet reported to be faster than IDA* as it can make full use of the available memory.

On the theoretical aspect, we suspect that a better worst-case bound than $O(N^2)$ can be derived for A* on undirected graphs when the edge costs are a fixed finite set of rational constants that are independent of the problem size, and possibly DP and BPMX have an even lower worst-case bound.

Furthermore, it is possible that BPMX has a better-than-$O(N^2)$ worst-case bound on general undirected graphs if it employs a particular tie-breaking rule. Also, we have attempted to characterize the degree of inconsistency of a heuristic on a specific graph, but no significant progress has been achieved. Answering this question will bring a deep insight into the inconsistency problem.

Bibliography

[1] Amitava Bagchi and Ambuj Mahanti. Search Algorithms Under Different Kinds of Heuristics-A Comparative Study. *Journal of ACM*, 30(1):1–21, 1983.

[2] Marcel Ball and Robert Holte. The Compression Power of Symbolic Pattern Databases. In *The International Conference on Automated Planning and Scheduling (ICAPS)*, pages 2–11, 2008.

[3] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[4] Vadim Bulitko and Greg Lee. Learning in Real-Time Search: A Unifying Framework. *Journal of Artificial Intelligence Research (JAIR)*, 25:119–157, 2006.

[5] Vadim Bulitko, Mitja Lustrek, Jonathan Schaeffer, Yngvi Bjornsson, and Sverrir Sigmundarson. Dynamic Control in Real-Time Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)*, 32:419 – 452, 06 2008.

[6] Partha Chakrabarti, Sujoy Ghose, Arup Acharya, and S. C. De Sarkar. Heuristic Search in Restricted Memory. *Artificial Intelligence*, 41(2):197–221, 1989.

[7] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[8] Joseph Culberson and Jonathan Schaeffer. Searching with Pattern Databases. In *Canadian Conference on AI*, pages 402–416, 1996.

[9] Edsger Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] Ariel Felner, Richard Korf, and Sarit Hanan. Additive Pattern Database Heuristics. *Journal of Artificial Intelligence Research (JAIR)*, 22:279–318, 2004.

[11] Ariel Felner, Richard Korf, Ram Meshulam, and Robert Holte. Compressed Pattern Databases. *Journal of Artificial Intelligence Research (JAIR)*, 30:213–247, 2007.

[12] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert Holte. Dual Lookups in Pattern Databases. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 103–108, 2005.

[13] Lestor Ford and Delbert Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[14] Ralph Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Addison Wesley Publishing Company, 1998.

[15] Eric Hansen and Rong Zhou. Anytime Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)*, 28:267–297, 2007.

[16] Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum-Cost Paths. *IEEE Transactions of Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[17] Peter Hart, Nils Nilsson, and Bertram Raphael. Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *SIGART Newsletter*, 37:28–29, 1972.

[18] John Hopcroft and Robert Tarjan. Efficient Algorithms for Graph Manipulation. *Communication of ACM*, 16:372–378, 1973.

[19] Donald Johnson. A Note on Dijkstra's Shortest Path Algorithm. *Journal of ACM*, 20(3):385–388, 7 1973.

[20] Sven Koenig. A Comparison of Fast Search Methods for Real-Time Situated Agents. In *AAMAS*, pages 864–871, 2004.

[21] Sven Koenig and Maxim Likhachev. Real-Time Adaptive A*. In *AAMAS*, pages 281–288, 2006.

[22] Richard Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, 1985.

[23] Richard Korf. Real-Time Heuristic Search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[24] Richard Korf. Linear-Space Best-First Search. *Artificial Intelligence*, 62(1):41–78, 1993.

[25] Richard Korf. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *AAAI Conference on Artificial Intelligence*, pages 700–705. AAAI Press, 1997.

[26] Richard Korf. Recent Progress in the Design and Analysis of Admissible Heuristic Functions. In *AAAI Conference on Artificial Intelligence*, pages 1165–1170. AAAI Press, 2000.

[27] Richard Korf and Ariel Felner. Disjoint Pattern Database Heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

[28] Richard Korf and Ariel Felner. Recent Progress in Heuristic Search: A Case Study of the Four-Peg Towers of Hanoi Problem. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 2324–2329, 2007.

[29] Richard Korf, Michael Reid, and Stefan Edelkamp. Time Complexity of Iterative-Deepening A^*. *Artificial Intelligence*, 129(1-2):199–218, 2001.

[30] Maxim Likhachev, Geoffrey Gordon, and Sebastian Thrun. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *Neural Information Processing Systems (NIPS)*, 2003.

[31] Ambuj Mahanti, Subrata Ghosh, Dana Nau, Asim Pal, and Laveen Kanal. On the Asymptotic Performance of IDA*. *Annals of Mathematics and Artificial Intelligence*, 20(1-4):161–193, 1997.

[32] Alberto Martelli. On the Complexity of Admissible Search Algorithms. *Artificial Intelligence*, 8(1):1–13, 1977.

[33] Laszlo Mero. A Heuristic Search Algorithm with Modifiable Estimate. *Artificial Intelligence*, 23(1):13–27, May 1984.

[34] Edward Moore. The Shortest Path Through a Maze. In *International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[35] Nils Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.

[36] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[37] Chris Rayner, Katherine Davison, Vadim Bulitko, Kenneth Anderson, and Jieshan Lu. Real-Time Heuristic Search with a Priority Queue. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 2372–2377, 2007.

[38] Stuart Russell. Efficient Memory-Bounded Search Methods. In *European Conference on Artificial Intelligence (ECAI)*, pages 1–5, 1992.

[39] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[40] Mehdi Samadi, Maryam Siabani, Ariel Felner, and Robert Holte. Compressing Pattern Databases with Learning. In *European Conference on Artificial Intelligence (ECAI)*, pages 495–499, 2008.

[41] Nathan Sturtevant, Zhifu Zhang, Robert Holte, and Jonathan Schaeffer. Using Inconsistent Heuristics on A* Search. In *First International Symposium on Search Techniques in Artificial Intelligence and Robotics*, pages 106–113, 2008.

[42] Uzi Zahavi, Ariel Felner, Neil Burch, and Robert Holte. Predicting the Performance of IDA* with Conditional Probabilities. In *AAAI Conference on Artificial Intelligence*, pages 381–386. AAAI Press, 2008.

[43] Uzi Zahavi, Ariel Felner, Jonathan Schaeffer, and Nathan Sturtevant. Inconsistent Heuristics. In *AAAI Conference on Artificial Intelligence*, pages 1211–1216. AAAI Press, 2007.