STEPS TOWARDS A SCIENCE OF HEURISTIC SEARCH

BY

Christopher Makoto Wilt

MS in Computer Science, University of New Hampshire 2012
AB in Mathematics and Anthropology, Dartmouth College, 2007

DISSERTATION

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science

May, 2014

ALL RIGHTS RESERVED

©2014

Christopher Makoto Wilt

This dissertation has been examined and approved.

Dissertation director, Wheeler Ruml,
Associate Professor of Computer Science
University of New Hampshire

Radim Bartŏs,
Associate Professor, Chair of Computer Science
University of New Hampshire

R. Daniel Bergeron,
Professor of Computer Science
University of New Hampshire

Philip J. Hatcher,
Professor of Computer Science
University of New Hampshire

Robert Holte,
Professor of Computing Science
University of Alberta

Date

ACKNOWLEDGMENTS

Since I began my doctoral studies six years ago, Professor Wheeler Ruml has provided the support, guidance, and patience needed to take me from "never used a compiler" to "on the verge of attaining the highest degree in the field of computer science". I would also like to acknowledge the contributions my committee have made to my development over the years.

The reality of graduate school is that it requires long hours and low pay, but one factor that makes it more bearable is good company, and I have the University of New Hampshire Artificial Intelligence Research Group to thank for making those long days more bearable.

The work in this dissertation was partially supported by the National Science Foundation (NSF) grants IIS-0812141 and IIS-1150068. In addition to NSF, this work was also partially supported by the Defense Advanced Research Projects Agency (DARPA) grant N10AP20029 and the DARPA CSSG program (grant HR0011-09-1-0021).

TABLE OF CONTENTS

ACKNOWLEDGMENTS iv

LIST OF TABLES viii

LIST OF FIGURES x

ABSTRACT xi

Chapter 1 Introduction **1**

1.1 Formal Background 1

1.2 Search Algorithm Proliferation 3

1.3 Better Searching through Science 4

1.4 Dissertation Outline 5

Chapter 2 Bidirectional Search **9**

2.1 Heuristic Improvement through Bidirectional Search 11

2.2 Incremental KKAdd 14

2.3 A* with Adaptive Incremental KKAdd 20

2.4 IDA* with Incremental KKAdd 21

2.5 Empirical Evaluation 22

2.6 Limitations 30

2.7 Conclusion 30

Chapter 3 Managing the Open List **32**

3.1 Previous Work 33

3.2 Priority Queues 35

3.3 Fringe Search 41

3.4 Selecting an Implementation 47

3.5 Conclusion 49

Chapter 4 Building Effective Heuristics for Greedy Best-First Search **51**

4.1 Introduction 51

4.2 When is increasing w bad? 53

4.3 Heuristic Requirements 55

4.4 Greedy Best-First Search with A* Heuristics 58

4.5 Quantifying Effectiveness in Greedy Heuristics 68

4.6 Building a Heuristic by Hill Climbing on Greedy Distance Rank Correlation 78

4.7 Related Work 81

4.8 Conclusion 83

Chapter 5 Why d is Better than h **85**

5.1 Operator Costs 85

5.2 The importance of d 96

5.3 Related Work 119

5.4 Conclusion 121

Chapter 6 Conclusion **123**

LIST OF TABLES

2-1 CPU seconds (T), expansions (E), and heuristic evaluations (H) (both in thousands) required to find optimal solutions. 27

3-1 Average performance of Fringe Search and A* with different open list implementations on five benchmark domains. A* (All Nodes) denotes the number of expansions done by A* expanding all nodes in the final f layer, instead of stopping at the first provably optimal solution. Bold denotes the best value for each metric. 37

3-2 Average total solving time of A* using a binary heap and a Fast Float Heap on different problems. 41

3-3 Sign test between FFH and binary heap A* solving the Towers of Hanoi 41

3-4 A* using 3 different kinds of open lists solving the inverse 15 puzzle . . 48

4-1 Domain Attributes for benchmark domains considered 53

4-2 Average number of nodes expanded to solve 51 12 disk Towers of Hanoi problems. 58

4-3 Amount of work required by Greedy best-first search and A* to solve 3x4 tile instances with different pattern databases. DNF denotes at least one instance would require more than 8GB to solve. 66

4-4 Average % error and correlation between $h(n)$ and $h^*(n)$ 69

4-5 Correlation between $h(n)$ and $d^*(n)$ 74

4-6 Average % error, correlation between $h(n)$ and $h^*(n)$, and correlation between $h(n)$ and $d^*(n)$ in City Navigation 5 5 76

4-7 Expansions to solve TopSpin problem with the stripe cost function using different PDBs 79

5-1 Difficulty of solving the puzzle from Figure 5-1 86
5-2 Failure rate of A* on tile puzzles with different cost functions, and confidence interval for failure rate 87
5-3 A* expansions on different problems 88
5-4 Sizes of local minima and average number of expansions required to find a solution. 94
5-5 Expansions required by greedy best-first search to solve different pancake puzzles using a constant size (7 cake) PDB. 95
5-6 Correlation of heuristic with search effort in various domains using Kendall's τ, Spearman's ρ, and Pearson's r 98
5-7 Algorithms on unit cost domains 115
5-8 Solving the 4x4 face^3 sliding tile puzzle 115
5-9 Solving 14-pancake (cost = sum) problems 117
5-10 Grid Path Planning with Life Costs 118

LIST OF FIGURES

1-1 Example navigation problem from Warcraft 2 2

1-2 Graph created from navigation problem 2

2-1 TopSpin puzzle 10

2-2 Towers of Hanoi problem with 4 pegs and 6 disks 10

2-3 Solving time of A* and four bidirectional searches on the Towers of Hanoi problem 12

2-4 Algorithm 1: A* with Incremental KKAdd 15

2-5 Algorithm 2: Forwards Search for Incremental KKAdd 15

2-6 Algorithm 3: Backwards Search for Incremental KKAdd 16

2-7 A city navigation problem with $n_p = n_c = 3$, with 15 cities and 15 locations in each city. 24

3-1 Fast Float Heap 39

4-1 Domains where increasing the weight speeds up search 55

4-2 Domains where increasing the weight slows down search 56

4-3 The minimum h value on open as the search progresses, using different pattern databases 59

4-4 Two Towers of Hanoi states, one near a goal (top) and one far from a goal (bottom). Note that the goal peg is the leftmost peg. 60

4-5 The minimum h value on open as the search progresses solving a Towers of Hanoi problem using disjoint pattern databases with different cost functions, square on left, reverse square on right. 63

4-6 TopSpin puzzle with different heuristics 64

4-7 Different tile abstractions. DNF denotes at least one instance would require more than 8GB to solve. 66

4-8 Plot of $h(n)$ vs $h^*(n)$, and $h(n)$ vs $d^*(n)$ for City Navigation 4 4 70

4-9 Average log of expansions with different heuristics, plotted with different GDRC 75

4-10 Average log of expansions with each of the possible 462 5/6 disjoint PDB heuristics for the 3x4 sliding tile puzzle, plotted against GDRC 76

4-11 Expansions with differing number of neighbors for cities and places . . . 77

4-12 Hill Climbing PDB Builder 78

5-1 Left: 15 puzzle instance with a large heuristic minimum. Right: State in which the 14 tile can be moved. 86

5-2 Examples of heuristic error accumulation along a path 91

5-3 Picture of one or two of local minima, depending on how they are defined 99

5-4 Example of how to remove extra nodes from a supergraph 102

5-5 An example of a shortcut tree. 103

5-6 A search tree with a local minimum. 107

5-7 The minimum h value on open as the search progresses, using a disjoint PDB. 110

ABSTRACT

STEPS TOWARDS A SCIENCE OF HEURISTIC SEARCH

by

Christopher Makoto Wilt

University of New Hampshire, May, 2014

Heuristic search is one of the most important graph search techniques, allowing people to find paths through graphs that can have more nodes than there are atoms in the universe. There has been much research in developing heuristic search algorithms, but much less work on when and why those new heuristic search algorithms work well.

Every heuristic search algorithm makes assumptions about the nature of the graph to be searched and the heuristic, and these assumptions allow the algorithm to perform well in certain circumstances, but perform poorly when those assumptions turn out to be false. The thesis of this dissertation is that understanding the assumptions behind heuristic search algorithms can be used to better select heuristic search algorithms, and to improve heuristic search algorithms.

We begin by discussing how assumptions made during optimal heuristic search impact performance, and how directly addressing these assumptions can lead to improved performance. We next turn our attention to implementation techniques, and how it is possible to squeeze additional performance out of a heuristic search algorithm by leveraging assumptions built into the data structures within the search algorithm. Last, we discuss how a better understanding of the assumptions built in to greedy best-first search allows us to tailor the heuristic so as to better meet the requirements of greedy best-first search, resulting in improved performance.

CHAPTER 1

Introduction

Heuristic search is a well known technique in artificial intelligence for solving a wide variety of problems, ranging from planning for different kinds of robots (Likhachev et al., 2003) to figuring out how to route paper in a printer (Ruml et al., 2011).

A simple example of grid path planning is navigating a character in a video game. The character is allowed to move in the cardinal directions, but is not allowed to occupy all locations, because some locations are already occupied. A simple example is shown in Figure 1-1, where the objective is to find a path that takes the footman from his start location (top right) to his goal location (bottom right).

The footman is able to move up, down, left, or right. The objective is to find a sequence of moves that take the footman from his start location to his goal location. If we create a graph representing what the footman can do, we end up with something similar to Figure 1-2. Each node in the graph represents a place the footman could be, and the edges in the graph represent different actions the footman can take.

1.1 Formal Background

Formally, a heuristic search problem consists of the following components. The first is a weighted, directed graph $< V, E >$ where V represents the set of vertices in the graph, and $E \in V \times V \times \mathbb{R}$ represents the edges in the graph. An edge consists of a starting vertex (the first part of the triple), an ending vertex (the second part of the triple), and the cost of using the edge (the third part of the triple). The cost of the edge is commonly referred to as the edge weight. The next component to a heuristic search problem is a vertex $start \in V$ which is the start vertex. The next component is a goal test, which is

Figure 1-1: Example navigation problem from Warcraft 2

Figure 1-2: Graph created from navigation problem

a function $v \to \{True, False\} : v \in V$ that returns true if v is a goal vertex, and false otherwise. The last component of a heuristic search problem is the heuristic, which is a function $v \to \mathbb{R} : v \in V$ that estimates the cost of getting from v to a node that satisfies the goal test.

A path through the graph is a sequence of nodes such that for all adjacent pairs of nodes in the sequence, there is an edge that goes from the first node to the second node. The *cost* of a path is the sum of all of the weights associated with the edges in the path. The *length* of a path is the count of edges in the path. Note that if all of the edges in the graph have the same cost, cost will be linearly proportional to length, but if the edge costs vary, cost and length can be very different.

1.2 Search Algorithm Proliferation

There are many different algorithms that solve search problems. For example, every computer scientist undergraduate knows about depth first search and breadth first search. More sophisticated alternatives that produce provably optimal solutions include Dijkstra's algorithm, A* (Hart et al., 1968), and iterative deepening A* (Korf, 1985a), and Simplified memory limited A* (Russell, 1992). If we are willing to accept suboptimal solutions, there is an even wider variety of possibilities, including Weighted A* (Pohl, 1970), greedy best-first search (Doran and Michie, 1966), MSC-k Weighted A* (Furcy and Koenig, 2005), Anytime Window A* (Aine et al., 2007), Beam Stack Search (Zhou and Hansen, 2005), Skeptical search (Thayer and Ruml, 2011a), and best first beam search (Ibaraki, 1978) just to name a few. A very important problem, then, is sifting through these algorithms to identify which algorithm to use to solve a particular heuristic search problem.

As a simple example, consider the following situation. A programmer has represented a problem he or she has as a heuristic search problem, and needs to use a heuristic search algorithm to find a solution to the problem he or she just created.

When faced with a new search problem, the first algorithm applied is often A*, since if A* is able to find solutions, the solution will be optimal, which is desirable. Unfortunately,

it is often the case that A* consumes too much time or too much memory. When this happens, people often try Weighted A*, and if Weighted A* fails to find a solution, the next algorithm used is often greedy search.

When A*, Weighted A*, and greedy best-first search all fail to solve the problem, people usually turn to the scientific literature on heuristic search for what to do. As was previously mentioned, the heuristic search literature is full of examples of different heuristic search algorithms. Most papers presenting new heuristic search algorithms follow a very predictable formula: propose the new algorithm, then show the new algorithm performing very well on a few benchmark domains. If the paper is strong, it may also contain a discussion about exactly what kind of domain features the new algorithm leverages to achieve its good performance, but this is not universally present. If the new problem is the same as a problem discussed in the literature, there is a good chance there exists a better algorithm for solving that problem, and the only issue is finding the correct paper and implementing the new algorithm.

Unfortunately, if the new problem is not in the literature, the next step is much more difficult. The brute force approach is to try algorithms until one is found that works, but this approach is extremely inefficient, as many state of the art heuristic search algorithms are quite complicated to implement, and it is not clear that there will actually be any return on that investment until after the investment has actually been made. Clearly, there is a need for a better way to identify promising search algorithms.

1.3 Better Searching through Science

When scientists look to classify things, the first step is to identify relevant features for classifications. In the context of heuristic search, is not clear what pieces of information should be used to predict search algorithm performance. Every algorithm makes implicit assumptions about the nature of the domain. When those assumptions turn out to be true, the algorithm performs well, and when the assumption turns out to be false, the algorithm is liable to perform poorly.

We analyze what domain attributes some of the most popular heuristic search algorithms leverage. We then use the results of that analysis to help figure out when different algorithms should be used, and when they should not be used. Ultimately, this will allow us to offer more accurate predictions about when various kinds of techniques are likely to be helpful, and when those same techniques are unlikely to yield performance improvements. This analysis will form a starting point for research into the question of why different heuristic search algorithms work well, covering the most popular heuristic search algorithms and the most important domain attributes for those algorithms.

The thesis of this dissertation is that understanding the assumptions behind heuristic search algorithms can be used to better select heuristic search algorithms, and to improve heuristic search algorithms.

1.4 Dissertation Outline

This dissertation will discuss how to select a heuristic search algorithm, and how different heuristic search algorithms and techniques cope with different domain features and heuristics.

The second chapter discusses optimal bidirectional search. Although the A* search algorithm (Hart et al., 1968) is the most popular optimal search algorithm, it is not the only choice for finding optimal solutions. In many domains, it is possible to search in both directions. In some of these domains (but not all) the heuristic can be corrected using knowledge gained by searching backwards. This correction can lead to substantial speedup, but in order to realize the speedup, the correct amount of backwards search must be done; too little backwards search and there is no benefit, and too much backwards search and the backwards search consumes more time than the entire uncorrected forwards search would have needed. While these may seem like obstacles to making an effective bidirectional optimal heuristic search algorithm, by directly addressing these assumptions, we are able to come up with an algorithm that is able to bound its overhead if the heuristic is not amenable to correction, and online, at runtime, figure out how much backwards search

should be done. In this case, the heuristic search algorithm is able use information about its own assumptions, and manage its own configuration at runtime, automatically tailoring itself to the domain at hand. This work was presented at AAAI 2013 (Wilt and Ruml, 2013).

The next chapter discusses implementing the open list in a heuristic search. Any search algorithm is required to expand certain nodes, and there is a limit to what is possible regarding reducing expansions. All varieties of best first search require a priority queue to order nodes according to whatever is defined as, "best". There are a number of different choices for how to implement a priority queue, and the different queues are each appropriate in different situations. We begin by discussing precisely when each kind of queue is best, and then introduce a new kind of priority queue that is optimized for the operations required in a heuristic search. We discuss what assumptions different open list implementations have, and when each kind of open list is effective. If we know what kind of problem we are dealing with, we can use this knowledge to select an effective open list, leading to a better heuristic search algorithm. We also discuss how to use this knowledge to dynamically manage the open list at runtime, switching between data structures as the search progresses and different data structures become optimal, once again leading to more effective heuristic search algorithms.

Chapter four discusses building a heuristic for greedy best-first search. Wilt et al. (2010) show that weighted A* and greedy best-first search are extremely effective heuristic search algorithms. In order to scale effectively, Weighted A* relies upon the assumption that greedy best-first search is effective, which is not always the case. We first show that as the weight is increased, in some domains the amount of effort required to find a solution goes down, but in other domains, the amount of effort required to find a solution can actually go up. This work was presented at SoCS 2012 (Wilt and Ruml, 2012). We next discuss how to construct an effective heuristic for a greedy best-first search, showing that the rules for building a good heuristic for A* can be antithetical to building a good heuristic for greedy best-first search. The reason is that A* assumes the heuristic can be used to prove that nodes are far from the goal (so expansion of those nodes can be delayed), while greedy best-

first search assumes the heuristic can be used to identify nodes that are near the goal (so those nodes can be expanded). When trying to build an effective greedy best-first search, it is important to ensure that the heuristic is able to satisfy the assumptions that the greedy best-first search is making. By examining the assumptions that greedy best-first search makes of its heuristic, we were able to identify a quick and dirty way to analyze the quality of a heuristic for greedy best-first search, the correlation between the heuristic $h(n)$ and the true distance to the goal, $d^*(n)$, which measures how many edges are between n and the nearest goal. This metric allows us to determine whether or not greedy best-first search is likely to be effective, helping us to determine whether or not we should use that algorithm. Last, we show how this metric can be used to automatically construct a quality heuristic for greedy best-first search.

In Chapter Four, we come up with recommendations for building a good heuristic for greedy best-first search. In the fifth chapter we discuss why greedy best-first search using the d heuristic that estimates distance tends to be easier to follow than using a greedy best-first search using the standard h heuristic that estimates cost. We begin by showing that within the context of optimal search how much the different operators in the domain cost relative to one another can have a huge impact on algorithm performance, and offering a theoretical explanation as to why this can happen. This work shows us that one very important assumption that should be questioned is whether an optimal solution is even necessary, because finding optimal solutions is not always practical or even possible. We then show that the same general phenomenon can also be observed in suboptimal algorithms. This work was presented at SoCS 2011 (Wilt and Ruml, 2011). Next, using a model, we show that the d heuristic is more likely to satisfy the requirements of a greedy best-first search, which is why we observe that greedy best-first using the d heuristic tends to be faster than greedy best-first search using the h heuristic. Last, we show empirically that algorithms that consider this d heuristic tend to outperform algorithms like Weighted A* on non-unit cost domains. This work was presented at SoCS 2011 (Wilt and Ruml, 2011).

This dissertation shows how understanding and acknowledging the assumptions behind

heuristic search algorithms can allow us to better select heuristic search algorithms, and to improve heuristic search algorithms.

CHAPTER 2

Bidirectional Search

If the solution is required to be provably optimal, this means that the algorithm is not only required to find the optimal solution, but it is also required to be able to prove that the resulting solution is indeed an optimal solution, and that all other solutions are of equal or greater cost. In almost all cases, the requirement that the solution be proven optimal requires expanding additional nodes beyond those that make up the solution.

One way to improve optimal heuristic search is through searching both to and from the goal. Bidirectional search has a few very basic problems that often prevent its application, and even if it is possible to do bidirectional search, it may not actually be practical.

The first and most important requirement of bidirectional search is identifying the goal state. In many problems, there is only a single goal node, as is the case in grid path planning or in the sliding tile puzzle. In other domains, however, there is more than one goal.

In the dynamic robot navigation problem, the objective is to control the gas and steering of a robot to move the robot from its start location to a goal location while respecting the dynamic constraints of the robot. For example, the robot's minimum turn radius is connected to its current speed. In this problem, the goal may be to simply arrive in a particular region, agnostic of heading or speed.

In STRIPS planning, there are facts that are either true or false in every state, and actions can change the truth value of the facts. For example, facts for a robotic bartender might be "have beer" and "served customer". An example of an action is "serve customer" which would change "have beer" from true to false, and change "served customer" from false to true. Often, the goal condition often restricts only some of the variables, allowing other variables to have any value. If this is the case, then there are a wide variety of states

Figure 2-1: TopSpin puzzle

Figure 2-2: Towers of Hanoi problem with 4 pegs and 6 disks

that are considered goals, and when searching backwards, all of these different states must be considered. In our simple example, it may make sense to only specify that the customer gets served ("customer served" = true) but not to specify whether or not the robot has a beer ("has beer" can be either true or false).

On the Towers of Hanoi (an example is shown in Figure 2-2) the objective is to rearrange the disks such that all of the disks are on the same peg. The movement of the disks is constrained such that only one disk may move at a time, and when a disk is put onto a peg, the disk must be smaller than all of the disks currently occupying the peg the disk is going on. In this problem, there is only goal state, and operators are invertible, so the backwards

expansion function is the same as the forwards expansions function.

In problems where the set of goal states is large, bidirectional search can be much more difficult, because there are a large number of nodes in the initial goal state set. For example, if a domain independent planning problem only requires some fact x_1 be true for the goal test, half of all possible are goal states, and enumerating them could be prohibitively expensive.

Another problem that sometimes plagues the backwards component of a bidirectional search is the fact that the operator inverses may not be known, or may be impractical to compute. In other situations, while it may be possible to compute inverse operators, it may be difficult to write the code to do so, as is the case for dynamic robot navigation.

There are a number of proposals for bidirectional search, but these proposals tend to be brittle. For example, one popular proposal, perimeter search (Dillenburg and Nelson, 1994) uses a large number of heuristic evaluations, and in practice often performs worse than A* (Wilt and Ruml, 2013). Another popular proposed algorithm, single frontier bidirectional search (Felner et al., 2010), does not even finish on some domains, while single directional A* is able to find solutions without any problem (Wilt and Ruml, 2013). As such, while the idea of bidirectional search seems to have intuitive merit, turning that idea into a practical heuristic search algorithm has proven to be difficult.

2.1 Heuristic Improvement through Bidirectional Search

If the basic requirements for bidirectional search are met (inverse operators can be computed and a goal set of acceptable size), one extremely effective way to leverage bidirectional search is through heuristic improvement. For example, on the Towers of Hanoi problem with 14 disks and 4 pegs, we can see in Figure 2-3 that the bidirectional searches that correct for heuristic error fare much better than A* in this domain.

Kaindl and Kainz (1997) discuss a wide variety of ways to approach bidirectional search, one of which is using the backwards search to better inform the forward search's heuristic estimates. On domains with a consistent heuristic, a method presented by Kaindl and Kainz

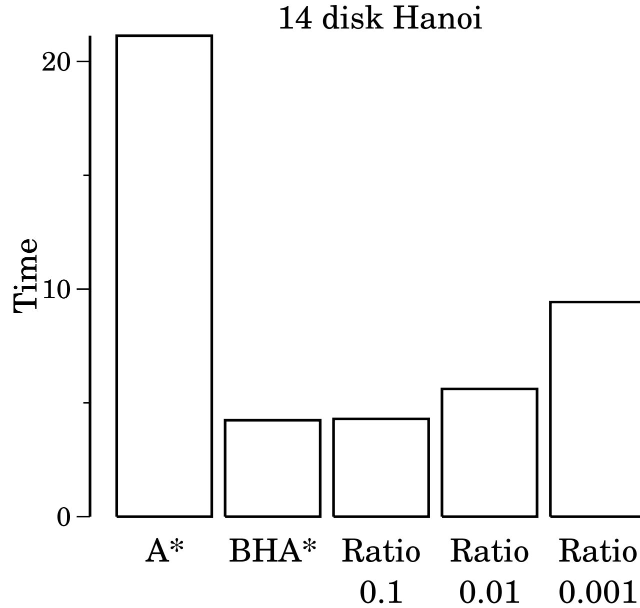

Figure 2-3: Solving time of A* and four bidirectional searches on the Towers of Hanoi problem

(1997) which we here call KKAdd learns a correction to the original heuristic by expanding nodes backwards from the goal, establishing the true $h(n)$ values (notated $h^*(n)$) for nodes that have been expanded backwards. The simplest approach is to expand the nodes in g_{rev} order, where g_{rev} is the cost of getting to the node starting at a goal using backwards expansions, but any order that expands nodes with optimal g_{rev} values ($g_{rev} = h^*(n)$) may be used. Nodes that have been generated but not yet expanded in the backwards direction form a perimeter around the goal region.

The original forward $h(n)$ value for each of the nodes in the perimeter can be compared to the $g_{rev}(n)$ values for those nodes. Each node in the perimeter has heuristic error $\epsilon(n) = g_{rev}(n) - h(n)$. The heuristic error present for all nodes that have not been reverse expanded, h_{err}, is defined as $min_{n \in open}(\epsilon(n))$. All solutions starting at a node that was not yet reverse expanded go through this perimeter, therefore all outside nodes must have at least as much heuristic error as the minimum error node in the perimeter. An admissible heuristic in the forwards search is constructed by first looking up the node in the cache of reverse expanded

nodes. If the node has been reverse expanded, its $h^*(n)$ value is used. Otherwise the node is assigned $h_{KKAdd}(n) = h(n) + h_{err}$ as its admissible heuristic estimate.

The problem with the KKAdd technique is that it requires runtime tuning. KKAdd expands N nodes in the backwards direction, then begins searching from the root to the goal. If N is set too low, there is little to no benefit from the backwards search, and if N is set too large, the backwards search can consume more time and memory than the A* would have consumed without any backwards search. Thus, it is imperative for good performance to properly tune this parameter. Despite this difficulty, we found that the technique had promise, and could be used to speed up optimal searches by a factor of up to six compared to a textbook implementation of A*.

We next introduce Incremental KKAdd, technique for improving the heuristic by using bidirectional search, adaptively managing the size of the backwards perimeter. The algorithm relies upon the key observation that the backwards searches can be done at any time in the search, not just at the beginning, meaning backwards search can be interleaved with forwards search. This means that one way to effectively manage the size of the backwards perimeter is to maintain a bounded ratio relative to the number of expansions done in the forwards search, eliminating the need for a hard number on the backwards expansions.

We show that this algorithm can substantially reduce the number of expansions if the conditions are right, and that the algorithm has a bounded overhead in the worst case. The key assumption behind both KKAdd and Incremental KKAdd is that there is a heuristic correction to be had. In some domains, there is no heuristic correction at all, because reverse expanding enough nodes to get a correction to the heuristic is impossible without running out of memory, so all of the work done in the reverse direction is wasted (although this waste can be bounded if desired). The root of this problem is the number of nodes connected to the goal nodes with $h = h^*$.

This work on heuristic improvement through bidirectional search was presented at AAAI-2013.

2.2 Incremental KKAdd

The problem with KKAdd is that it requires a runtime parameter, the amount of backwards search to do. If this parameter is set too small, the algorithm is no different from A*, but if the parameter is set too large, the algorithm expands a large number of irrelevant nodes in the backwards direction, and fails to reap a sufficiently large benefit when the search is run in the forward direction. Kaindl and Kainz (1997) report results for A* with the KKAdd heuristic, but they do not say how the size of the backwards search was set, only mentioning how many nodes each variant used in the backwards direction. It is, therefore, an open question how to make the technique practical for general use.

We introduce Incremental KKAdd, a method for correcting the heuristic by incrementally increasing the backwards perimeter's size. The approach relies upon the key observation that the backwards searches can be done at any time in the search, not just at the beginning, meaning backwards search can be interleaved with forwards searches. This means that one way to effectively manage the number of expansions in the backwards search is to maintain a bounded proportion relative to the number of expansions done in the forwards search. This is the approach taken in A* with Incremental KKAdd (see Algorithm 1).

Incremental KKAdd requires a single parameter, which is the desired proportion of backwards expansions. As we shall see in the empirical results, unlike many other algorithms with parameters we consider, the parameter used by A* with Incremental KKAdd is extremely forgiving, producing reasonable results across three orders of magnitude. A* with Incremental KKAdd initially proceeds in the backwards direction (Algorithm 1 line 9), expanding a small number of nodes backwards (we used 10). Next, the algorithm expands nodes in the forwards direction (Algorithm 1 line 11) such that when finished, the ratio is equal to the proportion of expansions done in the backwards direction. Last, the algorithm doubles the number of nodes that will be expanded in the backwards direction, and repeats the process.

fOpen is the forwards open list, and contains all nodes that have been generated but not

```
1:  function Incr KKAdd A*(ratio, initial, goal)
2:      fOpen = {Initial} sorted on base f, breaking ties on base h (not corrected)
3:      bOpen = {All Goal Nodes} sorted on ϵ_n
4:      closed = {Initial}
5:      h_err = 0
6:      incumbent = nil
7:      expLimit = 10
8:      while !solutionFound() do
9:          searchBackwards(expLimit)
10:         if !solutionFound() then
11:             searchForwards(expLimit · (1−ratio)/ratio)
12:         expLimit = expLimit * 2
13: function makeIncumbent(fwd, bwd)
14:     path = reverse path from bwd and append to fwd
15:     if path.cost < incumbent.cost then
16:         incumbent = path
17: function solutionFound(nil)
18:     if incumbent == nil then
19:         return false
20:     else if incumbent.cost > open.peek().f + h_err then
21:         return false
22:     else
23:         return true
```

Figure 2-4: Algorithm 1: A* with Incremental KKAdd

```
1:  function searchForwards(count)
2:      repeat
3:          next = fOpen.pop()
4:          for all child : next.expand() do
5:              inc, direction = closed.get(child)
6:              if inc ≠ nil and direction = REV then
7:                  makeIncumbent(child, inc)
8:                  continue
9:              else if inc == nil or inc.g < child.g then
10:                 fOpen.remove(inc)
11:                 closed.remove(inc)
12:             else if inc.g ≥ child.g then
13:                 continue
14:             fOpen.add(child)
15:             closed.add(child, FWD)
16:     until solutionFound() or at most count times
```

Figure 2-5: Algorithm 2: Forwards Search for Incremental KKAdd

```
1: function searchBackwards(count)
2:     repeat
3:         next = bOpen.pop()
4:         inc, direction = closed.get(next)
5:         if inc ≠ nil and direction = REV then
6:             continue
7:         else
8:             fOpen.remove(inc)
9:             makeIncumbent(inc, next)
10:        closed.add(next, REV)
11:        for all child : next.reverseExpand() do
12:            closedInc, dir = closed.get(child)
13:            if closedInc ≠ nil and dir = REV then
14:                continue
15:            else if closedInc ≠ nil and dir = FWD then
16:                makeIncumbent(closedInc, child)
17:            revInc = bOpen.get(child)
18:            if revInc = nil then
19:                bOpen.add(child)
20:                goalFrontier.add(child)
21:            else if revInc.g > child.g then
22:                bOpen.replace(revInc, child)
23:    until solutionFound() or at most count times
24:    h_err = min(ε(n) ∈ goalFrontier)
```

Figure 2-6: Algorithm 3: Backwards Search for Incremental KKAdd

yet expanded in the forwards direction, and have not been expanded backwards. bOpen is the open list for the backwards search. Closed contains all nodes that have been expanded or generated in the forwards direction, and all nodes that have been expanded in the backwards direction.

The forwards search is similar to A*, with a few differences. First, the definition of a goal is broader, as now any state that was reverse expanded can be treated as a goal, because its $h^*(n)$ and an optimal path to a goal is known. In addition, goal tests are done at generation, and complete paths are either stored as a new incumbent solution or deleted (if worse than the current incumbent), and in either case, are not put on the open list (Algorithm 2 line 7). The solutionFound() function (Algorithm 2 line 16, Algorithm 3 line 23) tells if an acceptable solution has been found, checking quality of the incumbent against the head of the open list. If the head of the forward search's open list ever has a corrected f value greater than or equal to that of the current incumbent, the algorithm terminates, returning the incumbent, which has now been proven to be optimal. Note that fOpen is ordered on uncorrected f values, obviating the need to resort as h_{err} rises. The forwards open list only contains nodes whose f value is of the form $f(n) = g(n) + h(n) + h_{err}$, because nodes with $f(n) = g(n) + h^*(n)$ are not included in the forwards open list. This means that sorting nodes in corrected f order and uncorrected f order produces the same ordering, because all nodes that are on the open list have the same constant added to their f value.

The backwards search is also able to generate incumbent solutions by generating states that have already been discovered by the forwards search (Algorithm 3 lines 9 and 16). Whenever the backwards search generates an incumbent solution, the quality of the incumbent is compared to the head of the forward search's open list, and if the incumbent's quality is less than or equal to the f value of the head of the forward search's open list, the algorithm immediately terminates.

The backwards search expands *count* nodes in the backwards direction in order of increasing $g_{rev}(n) - h(n)$, where $g_{rev}(n)$ is the g value of a node in the reverse direction. The goal of the backwards search is to provide a large heuristic correction, and expanding nodes

in order of heuristic error will raise the heuristic correction most quickly. This requires that the g_{rev} values be optimal, which we now prove.

2.2.1 Theoretical Properties

Lemma 1 (Lemma 5.2 of Kaindl and Kainz (1997)) *In any domain with a consistent heuristic, heuristic error is non-increasing along any optimal path to a goal.*

Theorem 1 *In any domain with a consistent heuristic, expanding nodes backwards from the goal in $g_{rev}(n) - h(n)$ order results in each node being expanded with its optimal g_{rev} value, and therefore the g_{rev} value is the same as the h^* value.*

Proof: First, let us suppose that a node $n1$ is the first node reverse expanded that does not have $g_{rev}(n1) = g^*_{rev}(n1)$. This means that there exists some positive δ such that $g_{rev}(n1) - \delta = g^*_{rev}(n1)$.

In order to expand this $n1$ with its optimal g_{rev} value, there must exist some other node $n2$ that should instead be expanded, which will eventually lead to $n1$ via the optimal reverse path. Since we assumed that $n1$ had the minimum value of $g_{rev}(n1) - h(n1)$, we know:

$$g_{rev}(n1) - h(n1) \leq g^*_{rev}(n2) - h(n2) \tag{2.1}$$

Since the optimal path backwards from the goal to $n1$ goes through $n2$, by Lemma 1 we also know:

$$g^*_{rev}(n2) - h(n2) \leq g^*_{rev}(n1) - h(n1) \tag{2.2}$$

Adding δ to both sides of Equation 2.2, and making the substitution $g^*_{rev}(n1) + \delta = g_{rev}(n1)$ on the right side yields:

$$g^*_{rev}(n2) - h(n2) + \delta \leq g_{rev}(n1) - h(n1) \tag{2.3}$$

Equations 2.1 and 2.3 yield:

$$g^*_{rev}(n2) - h(n2) + \delta \leq g^*_{rev}(n2) - h(n2) \tag{2.4}$$

which is a contradiction. □

Thus, the strategy of ordering the backwards search by error preserves the correctness of the heuristic correction.

A* with Incremental KKAdd relies upon the heuristic correction to reduce the number of expansions compared to A*. Although an improved heuristic is generally a good thing, it is known that, even when one heuristic strictly dominates another, it is still possible for A* with the weaker heuristic to terminate sooner than A* with the stronger heuristic (Holte, 2010), due to tie breaking in the final f layer. An important question to ask is if this can happen with the Incremental KKAdd heuristic. Theorem 2 tells us this will not happen.

Theorem 2 *The forwards search of A* with Incremental KKAdd will never expand a node that A* would not expand, if both algorithms are tie breaking in favor of low h, and any remaining ties are broken identically.*

Proof: A* with Incremental KKAdd sorts nodes on their f value, breaking ties in favor of low h, just as A*. Thus, A* with Incremental KKAdd will have the same nodes on the open list as A* with one exception: since nodes that have been reverse expanded have the optimal path to the goal known, these nodes are not expanded, so these nodes, and all of their children, will never be on the open list of A* with Incremental KKAdd. This pruning may cause other nodes currently on the open list of A* with Incremental KKAdd to have g values that are too large (if the optimal g value is achieved via a path going through a pruned node), delaying the node's expansion in A* with Incremental KKAdd. Since there is already a path to the goal as good as the path going through these nodes, A* gains nothing by expanding these nodes earlier, as A* with Incremental KKAdd already has an incumbent solution as good as any solution going through these nodes. □

Corollary 1 *The only nodes expanded by A* with Incremental KKAdd that A* might not also expand are the nodes expanded backwards from the goal.*

Proof: Direct consequence of Theorem 2. □

The ratio of backwards expanded nodes to forwards expanded nodes is bounded after the first iteration of forwards search by three times the ratio. When the backwards search

begins the perimeter's size is $ratio \cdot forwardExp$, and after the backwards search terminates, the perimeter's size is $ratio \cdot forwardExp + 2 \cdot ratio \cdot forwardExp$. Thus, the size of the perimeter is never more than three times the ratio times the number of forward expanded nodes.

2.3 A* with Adaptive Incremental KKAdd

Incremental KKAdd requires the user to decide what portion of effort should be dedicated to searching backwards. This raises two issues. First, while specifying a ratio is more robust than an absolute number, the ratio does make a difference, and the user might not know an ideal value, and second, the algorithm is unable to adjust to the domain or instance: sometimes backwards search is extremely helpful and more would provide further benefit, other times backwards search is not helpful at all and should be stopped. Incremental KKAdd commits to a specific ratio, which while it bounds the worst case, may not be the best ratio.

A* with Adaptive Incremental KKAdd eliminates the ratio parameter and uses experience during search to decide if it should expand nodes forwards or backwards. Intuitively, the algorithm must answer the question "will the overall search progress faster expanding backwards or forwards?". To answer this question, we begin with the observation that the termination condition of the search is when the incumbent solution's cost is less than or equal to the f value of the head of the open list. The f value of the head of the open list can be split into two independent components: the base f in the forwards direction (without the heuristic correction) and the heuristic correction from the backwards direction. The goal, therefore, is to increase the sum of these quantities using the fewest expansions possible. The algorithm must figure out which quantity is rising faster per expansion: the minimum f from the forward search's open list, or the h_{err} correction from the backwards search.

To do this, the algorithm begins by using a fixed ratio (we used 1:10) and doing three backwards/forwards iterations of A* with Incremental KKAdd using this ratio. After each forward search, it records the minimum f and number of forwards expansions done in the

forwards search and h_{err} and the number of backwards expansions done in the backwards search. After three backwards/forwards iterations, we have three data points that can be fit using linear regression to predict how future expansions can be expected to modify h_{err} and the f value of the head of the open list. We selected the number three because the minimum number of points needed for linear regression is two, and three provides additional robustness to the estimate, while four can give too much overhead. In most domains, the number of nodes with an f value below a given threshold increases exponentially with f (This is also known as the heuristic branching factor (Korf et al., 2001)). Thus, it is not unreasonable to assume that log(expansions) is linearly proportional to the f value of the head of the open list. Our experiments have also shown that the relationship between the log(backwards expansions) is also linearly proportional to h_{err}, as the correlation coefficients are reasonable (all are above .83).

Once A* with Adaptive Incremental KKAdd has completed three forwards/backwards iterations, it considers doubling the size of either the forwards search or the backwards search. In order to decide which is a better investment, it calculates the expected increase in f per expansion if the forward open list's size is doubled, and compares that to the expected increase in h_{err} per expansion if the backwards search's expansions are doubled. Once it figures out which direction looks more promising, it doubles the size of the search in that direction, adding an additional data point informing the next prediction.

2.4 IDA* with Incremental KKAdd

Iterative Deepening A* (Korf, 1985b) can also be augmented using Incremental KKAdd. For large problems with unit costs where duplicate states are rare, IDA*, rather than A*, is the state of the art algorithm. In IDA* with Incremental KKAdd, the forwards IDA* search is identical to an ordinary IDA* search, except it evaluates nodes using the KKAdd heuristic, and whenever the forwards search finds a node that has been reverse expanded, that node is turned into an incumbent solution and not expanded. The backwards search is unmodified, except it no longer needs to look for nodes that have been expanded in

the forwards direction, since the forwards search is no longer storing nodes. In order to meet its time complexity bound, the number of expansions done in each iteration should grow exponentially. IDA* with Incremental KKAdd can take advantage of this property by doing backwards expansions between forwards IDA* iterations, expanding enough nodes to achieve the desired ratio. Note that the backwards search may change the heuristic correction, and this may interfere with the forward search's ability to double. The overall performance will still be superior to that of IDA* without the improved heuristic, because each iteration will be smaller than the corresponding uncorrected IDA* iteration.

One possible complication with integrating the Incremental KKAdd heuristic into IDA* is the fact that calculating the KKAdd heuristic for a node requires looking up that node in a hash table. When doing A*, this hash table lookup is already done as a part of duplicate detection, but many implementations of IDA* omit cycle detection. In many domains, looking up a state in a hash table is substantially more expensive than expanding the state, which can cut into the overall savings.

2.5 Empirical Evaluation

We compare A* with Incremental KKAdd and Adaptive Incremental KKAdd A* against A*, A* with the KKAdd heuristic, single frontier bidirectional search, and perimeter search on seven different popular heuristic search benchmarks. We used a Java implementation of all the domains and algorithms, running on a Core2 duo E8500 3.16 GHz with 8GB RAM under 64 bit Linux.

For A* with Incremental KKAdd we selected three values for the ratio: 0.1, 0.01, and 0.001. If the ratio gets larger than 0.1, in many domains, the overhead becomes excessive. Values smaller than 0.001 would have resulted in only a handfull of backwards expansions. For perimeter search, we ran with a variety of perimeters ranging from 1 to 50, and only consider the best setting for each domain. For A* with the KKAdd heuristic, we considered backwards searches varying in size from 10 to 1,000,000. For single frontier bidirectional search, we implemented the enhancements described by Lippi et al. (2012) and tried both

enhanced single frontier bidirectional search (eSBS) and enhanced single frontier bidirectional search lite (eSBS-l), using the always jump (A) jumping policy. Our implementation of eSBS produces comparable node and expansion counts to those reported by Lippi et al. (2012), but our times are sometimes substantially slower for the grid and tiles domains, although on pancakes, both the times and node counts roughly align. To ensure that we don't discount an algorithm for which implementation tuning would provide an important advantage, we provide data on three metrics: runtime using our implementation, the number of times either a forwards or reverse expansion was requested, and the number of times the domain's base heuristic function was called. Sometimes, one or more of these algorithms ran out of either memory (7.5GB) or time (10 minutes) for one or more instances, and when that occurred, results for that configuration were marked as DNF.

The results of this comparison are shown in Table 2-1. The first domain we consider is dynamic robot path planning (Likhachev et al., 2003). We used a 50x50 world, with 32 headings and 16 speeds. The heuristic is the larger of two values. The first value estimates time to goal if the robot was able to move along the shortest path at maximum speed, which the real robot cannot do due to restrictions on its turn rate and acceleration. The second value makes the free space assumption, and estimates time to goal given the robot's current speed and the need to decelerate to zero at the goal. These worlds are unusually small, which was necessary to accommodate the algorithms that required a point-to-point heuristic (all pairs shortest paths) in this domain. To compensate for the easiness associated with having a small map, 35% of the cells were blocked randomly. All nodes in the space have a certain amount of heuristic error due to the fact that neither of these heuristics is a particularly accurate measure of the true cost of bringing the robot into the goal. As a result, the backwards search is extremely effective, correcting significant error in the heuristic. A* with all varieties of Incremental KKAdd are able to handily outperform A* in this domain. Here, perimeter search is not particularly effective, being slower than A*. The lite varieties of single frontier bidirectional search did not terminate. Across all instances, eSBS(A) averaged 123 seconds, which is substantially slower than any of the other algorithms.

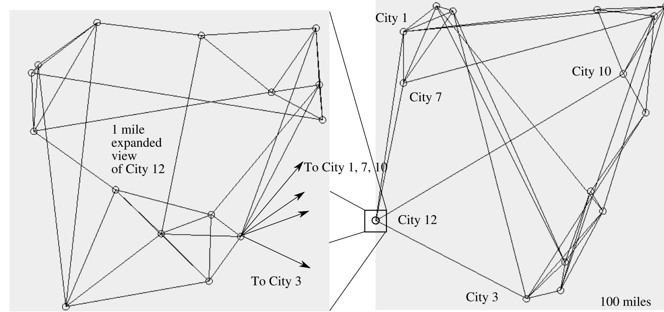

Figure 2-7: A city navigation problem with $n_p = n_c = 3$, with 15 cities and 15 locations in each city.

The next domain is city navigation, a domain which is designed to simulate a hub and spoke transportation network (Wilt and Ruml, 2012). City Navigation is designed to simulate navigation using a system similar to American interstate highways or air transportation networks. In this domain, there are cities scattered randomly on a 100x100 square, connected by a random tour which guarantees it is possible to get from any city to any other city. Each city is also connected to its n_c nearest neighbors. All links between cities cost the Euclidean distance + 2. Each city contains a collection of locations, randomly scattered throughout the city (which is a 1x1 square). Locations in a city are also connected in a random tour, with each place connected to the nearest n_p places. Links between places cost the true distance multiplied by a random number between 1 and 1.1. Within each city is a special nexus node that contains all connections in and out of this city. The goal is to navigate from a randomly selected start location to a randomly selected end location. For example, we might want to go from Location 3 in City 4 to Location 23 in City 1. Each city's nexus node is Location 0, so the goal for the example problem is to navigate from Location 3 to Location 0 in City 4, then find a path from City 4 to City 1, then a path from Location 0 in City 1 to Location 23 in City 1. An example instance of this type can be seen in Figure 2-7. The circles in the left part of the figure are locations, connected to other locations. The nexus node, Location 0, is also connected to the nexus nodes of neighboring cities. The right part of the figure shows the entire world, with cities shrunk down to a

circle.

City Navigation instances are classified by n_c and n_p, so we call a particular kind of City Navigation problem City Navigation n_p n_c. Since each location within a city has a global position, the heuristic is direct Euclidean distance. In this domain, solutions vary in length, and it is straightforward to manipulate the accuracy of the heuristic.

We used 1500 cities and 1500 places in each city, with each city connected to its 5 nearest neighbors, and each place connected to its 5 nearest neighbors. In this domain, the heuristic is euclidean distance, which assumes it is possible to move directly from one location to another. The backwards search is able to correct some of the heuristic error, which makes the A* with Incremental KKAdd variants faster than A*. Perimeter search, with its additional heuristic evaluations and lack of heuristic correction, was naturally slower. All varieties of eSBS did not terminate on this domain.

The third domain we consider is the Towers of Hanoi. We considered a 14 disk 4 peg problem with two disjoint pattern databases (Korf and Felner, 2002) tracking the bottom 12 disks and the top two disks respectively. In this domain, the backwards search is able to identify the fact that some of the states have substantial heuristic error, due to the fact that the disjoint pattern databases do not accurately track the relationship between the top two disks and the bottom 12 disks. Thus, error correction is extremely helpful, and allows the A* with Incremental KKAdd variants to terminate much faster than basic A*. Since it is not clear how to adapt the informative pattern database heuristic to give point-to-point information, we could not run either perimeter search or eSBS on this domain (N/A in Table 2-1).

The fourth domain is a TopSpin puzzle with 14 disks and a turnstile that swaps the orientation of 4 disks. For a heuristic, we used a pattern database that contained information about 7 disks. In this domain, the backwards searches were never able to find a correction to the heuristic, because there were a large number of nodes where the heuristic is perfect. Since the backwards searches were never able to find a heuristic correction, the investment A* with Incremental KKAdd variants make in the backwards search turns out to be un-

necessary. The amount of investment in the unneeded backwards search is bounded, so the overall performance of A* with Incremental KKAdd is only slightly slower than A*.

Domain		A*	KKAdd						A* with Incremental KKAdd				Peri-	SBS	SBS
			10	10^2	10^3	10^4	10^5	10^6	Adpt	10^{-1}	10^{-2}	10^{-3}	meter	(A)	(A)-1
	T	1.06	1.02	0.62	0.47	0.67	4.13	34.64	0.82	**0.45**	0.54	0.82	2.00	123	
Robot	E	53.1	40.2	21.3	**11.8**	17.9	101.0	683	**11.8**	12.6	18.4	31.1	67.3	25.8	DNF
	H	2,475	1,745	787	383	**340**	2,061	30e3	380	410	693	1,305	3,257	1,087	
	T	0.89	1.05	0.96	0.82	0.78	0.90	8.32	**0.70**	0.72	0.77	0.94	1.04		
CityNav	E	192	175	157	140	100	134	1,000	108	**92.5**	119	151	176	DNF	DNF
	H	1,184	1,056	947	845	604	826	6,005	656	**561**	716	908	6,345		
	T	23.2	21.3	16.9	10.6	5.3	**3.91**	18.4	4.79	4.33	6.27	10.88			
Hanoi	E	1,457	1,457	1,185	769	389	272	1,051	310	**266**	413	694	N/A	N/A	N/A
	H	5,583	5,583	4,542	2,951	1,513	1,259	6,194	1,251	**1,094**	1,599	2,665			
	T	3.44	**3.42**	3.43	3.57	3.81	6.77	40.63	3.51	3.97	3.50	3.61	3.71	7.99	
TopSpin	E	94	93	94	94	103	197	1,089	94	108	108	94	93	**21**	DNF
	H	1,319	1,315	1,316	1,329	1,452	2,718	15e3	1,329	1,516	1,330	1,317	1,313	**306**	
	T	17.47	18.45	18.45	18.58	18.63	19.56	25.03	18.45	18.40	19.65	19.01	**10.11**	36.56	31.26
Tiles	E	2,922	2,918	2,919	2,920	2,926	3,019	3,885	2,919	2,920	3,004	2,929	1,082	**98**	**98**
	H	5,790	5,786	5,786	5,789	5,813	6,112	9,026	5,786	8,424	6,066	5,816	32e3	**202**	**202**
	T	1.88	1.93	1.86	2.04	2.18			2.13	2.62	1.96	1.89	2.15	1.88	**1.19**
Pancake	E	19	19	19	20	29	DNF	DNF	21	25	20	19	19	**3**	**3**
	H	771	772	775	810	1,161			825	990	793	774	771	**132**	133
	T	**1.24**	1.27	1.28	1.26	1.49	1.49	4.25	1.28	1.30	1.27	1.34	1.31		
Grid	E	382	382	381	376	370	429	1,183	373	385	**371**	378	382	DNF	DNF
	H	638	638	636	629	628	819	2,977	638	686	**624**	632	638		

Table 2-1: CPU seconds (T), expansions (E), and heuristic evaluations (H) (both in thousands) required to find optimal solutions.

The fifth domain is the sliding tile puzzle (15) with the Manhattan distance heuristic. We filtered out instances that any algorithm failed to solve. The A* with Incremental KKAdd variants are actually able to solve more instances (90) than A* (83), although not as many instances as Perimeter search (94). In the sliding tile puzzles, backwards search failed to derive a heuristic error correction. In this domain, like in TopSpin, without the benefit of a heuristic error correction the A* with Incremental KKAdd variants are not able to terminate early, but once again, we see that the overhead compared to A* is bounded.

A similar phenomenon can be observed in the 40 pancake problem. We used the gap heuristic (Helmert, 2010), which is extremely accurate. None of the A* with Incremental KKAdd variants were able to get a heuristic correction, and their tie breaking was not much more effective than the tie breaking inherited from A*. Thus, without any benefit from the additional overhead, the A* with Incremental KKAdd variants performed marginally worse than A* on this problem as well. eSBSA(l) is extremely effective in this domain, and its counterpart eSBSA is competitive with A*. Perimeter search took longer than A* to find solutions.

The seventh and last domain we consider is grid path planning. The grid path planning problems we considered were 2000x1200, with 35% of the grid cells blocked. We used a 4 way motion model and the Manhattan distance heuristic. The A* with Incremental KKAdd variants were able to get a heuristic correction, and this correction allowed them to terminate the forwards search earlier than A*. Unfortunately, the backwards search required to get the correction required approximately the same number of nodes as the forwards search was able to save, resulting in no net performance gain.

Overall, we can see that in domains like dynamic robot navigation, Towers of Hanoi, and City Navigation, where there is a large heuristic error correction, the A* with Incremental KKAdd searches are all able to find optimal solutions faster than A*. In domains like TopSpin, sliding tiles, and the pancake puzzle, in which the backwards search failed to derive a heuristic correction, the overall performance was comparable to A*, with only a small amount of overhead lost to the backwards search. Once again, this can be seen in

both the overall runtime, as well as in the number of expansions and heuristic evaluations done. In domains like grid path planning, the backwards search is able to correct the forwards heuristic, but the cost of the correction is approximately equal to the savings in the forward search, so there is no net gain. In every situation, the A* with Incremental KKAdd searches are competitive with A*, due to the fact that the amount of work they do above and beyond A* is, in the worst case, bounded by a constant factor, and in the best case, there is a substantial speedup over A*. Although the other bidirectional searches are sometimes able to outperform both A* and the A* with Incremental KKAdd variants, they are plagued by brittleness: on some domains, both SBS and Perimeter Search are substantially slower than A*, or are not able to find solutions at all.

It is worth nothing that for all domains, the speedup or slowdown is observable in terms of overall runtime as well as heuristic evaluations and expansions, which do not depend on the specific implementation. This means that it is reasonable to expect a performance improvement when using any implementation where the runtime is proportional to expansions and heuristic evaluations.

We can also see that the A* with Incremental KKAdd variants do a reasonable job of capturing the potential of the KKAdd heuristic without the tedium of having to manually tune the size of the backwards search, and without the risk of sometimes getting extremely poor performance, as sometimes happens if the KKAdd heuristic is used with too much or too little backwards search.

The timing numbers for TopSpin originally reported by Wilt and Ruml (2013) had unusual results for the TopSpin domain. The numbers for A*, KKAdd(10 and 100), Adaptive Incremental KKAdd, Incremental KKAdd(0.1, 0.01, and 0.001) were unusually high for the number of expansions done. We re-ran all of the experiments for TopSpin, and it is those results that are shown in Table 2-1. We also re-ran the results for all of the other domains, but there were no significant changes for the other domains.

2.6 Limitations

As can be seen in the previous section, the major limitation of Incremental KKAdd and Adaptive Incremental KKAdd is that in some domains, there is either no heuristic correction (TopSpin, tiles, and pancake) or not enough heuristic correction to make a difference (grid path planning). If there is no heuristic correction, any investment in backwards search can only be used for tie breaking.

Another limitation that only applies to Adaptive Incremental KKAdd is the fact that the algorithm cuts off the backwards search if there is no detected heuristic correction. In a domain where there is no heuristic correction for the first N nodes expanded backwards, but at node $N+1$ the correction is large, if N is high enough, Adaptive Incremental KKAdd will have permanently given up on expanding nodes in the backwards direction, and never discover the large correction. This can be somewhat mitigated by instead reducing the proportion of backwards expansions, but not making it 0. Fortunately, this problem did not present itself in any of the standard benchmark domains we consider. For the domains in which the backwards search was permanently terminated (sliding tile puzzle and TopSpin) there was no correction, even if we exhausted all of main memory attempting to correct the heuristic.

2.7 Conclusion

A* with the KKAdd heuristic relies upon the fact that there is a specific amount of backwards search that is helpful, and assumes that the user knows what that amount is. The problem with this assumption is that it can fail in one of two ways. First, the user might have no idea what is an appropriate value for the parameter. The second problem is that for some instances, lots of backwards search might be optimal, but for other instances, only a little backwards search is optimal. In either case, the core of the problem is knowing how much backwards search to do is critical.

Incremental KKAdd and Adaptive Incremental KKAdd address this assumption by first tying the amount of backwards expansions to do not to an arbitrary constant, but rather

to a proportion of all work done thus far, which results in improved performance when the KKAdd heuristic is able to help speed up search, and bounded overhead in the cases when the KKAdd heuristic is not able to speed up search. In this case, by acknowledging and addressing the assumption the KKAdd heuristic makes, we are able to come up with a more robust bidirectional heuristic search algorithm.

We have introduced a significantly improved incremental variant of the KKAdd procedure of Kaindl and Kainz (1997) that preserves its benefits while provably bounding its overhead. A* with Incremental KKAdd is empirically an effective way to speed up optimal heuristic search in a variety of domains. We have also shown that, in the worst case, the amount of additional work done is bounded by a constant factor selected at runtime, and that numbers ranging from 0.1 to 0.001 all result in reasonable performance, producing an algorithm that is extremely effective in the best case, and only induces a minimal constant factor additional work in the worst case. We also introduce A* with Adaptive Incremental KKAdd, which manages the ratio of forwards expansions to backwards expansions on its own and reliably yields good performance.

Both A* with Incremental KKAdd and A* with Adaptive Incremental KKAdd are able to provide substantial speedup as compared to A*. Unlike other bidirectional search algorithms that are sometimes much slower than A* or unable to find solutions at all, in the worst case, the A* with Incremental KKAdd varieties always return a solution in comparable time to A*, making Incremental KKAdd a robust and practical alternative to previous state-of-the-art bidirectional searches and A*.

CHAPTER 3

Managing the Open List

If it is not possible to speed up the search by improving the heuristic, as the Incremental KKAdd heuristic does, there are other areas to look for performance gains. The A* search algorithm (like many other best-first heuristic search algorithms) requires a priority queue so that the most promising node on the open list can be selected for expansion. As such, selecting the best kind of data structure for the priority queue is essential for good performance.

Almost all best-first heuristic search algorithms (like A*, weighted A*, and greedy best-first search) require a priority queue governed by the algorithm's definition of "best". For domains like the Towers of Hanoi where states are small and the heuristic is computed quickly, much of the runtime of the algorithm (30% in our implementation) can be consumed by the priority queue. The priority queues play an important role in the time complexity of the search algorithm, and as such, selecting the proper kind of priority queue is important.

There are many choices for data structures to implement a priority queue. The most common is a binary heap, but there are other data structures that can be used. As we will see below, exactly which data structure is the best choice depends largely on domain characteristics such as the transition cost function and the number of nodes that will need to be put into the priority queue. In this chapter, we discuss the benefits of optimizing the choice of priority queue to match the domain in question. We also discuss a new data structure we call a Fast Float Heap that is specially optimized to handle the requirements of a priority queue for a best-first search with a large open list and many distinct f values. We also discuss how to deploy this data structure for maximum performance.

A radically different approach to best-first search is Fringe Search (Björnsson et al., 2005), which uses an unsorted linked list for the open list. This algorithm has been shown to be effective for grid path planning, but we show that this is attributable to the fact that

grid path planning is a domain with unit cost and a heuristic that always changes by 1. We also discuss extensions to fringe search, and show how these extensions can not solve the underling problems with non-unit cost domains and heuristics that change quickly. We then relate these problems with fringe search to Iterative Deepening A* (Korf, 1985b), showing that IDA* exhibits the same problems.

Last, we discuss how a heuristic search algorithm can take advantage of the information available at runtime to select the most appropriate data structure for its open list, and how this can result in improved algorithm performance. This allows us to tune the open list on a per-instance basis, using the best kind of open list for the difficulty of the instance at hand.

3.1 Previous Work

The basis, as well as comparison point, for almost all work on optimal heuristic search is the A* algorithm (Hart et al., 1968). This algorithm expands nodes in $f(n) = g(n)+h(n)$ order, where $g(n)$ is the best known path from the start state s to n, and $h(n)$ is a lower bound on the cost of the best path from n to a goal. The A* algorithm requires a priority queue, and the priority queue is an important factor in the overall runtime of the A* algorithm. Priority queues are also central to other heuristic search algorithms, such as weighted A* (Pohl, 1970) and greedy best-first search (Doran and Michie, 1966).

The most complete recent work comparing different data structures for open lists in the context of a heuristic search was done by (Edelkamp et al., 2012). They compared the runtime of Dijkstra's algorithm using various comparator-based data structures for the open list. Their analysis included the standard array-based binary heap that is taught in almost all data structures and algorithms textbooks (Cormen et al., 2009; Sedgewick and Wayne, 2011), as well as more advanced heaps that are able to perform some of the priority queue operations in constant time. Their analysis showed that despite the fact that an array based binary heap has operations that take $O(log(n))$ time, the constant factors associated with the more exotic heaps render them unable to offer any benefit over

a basic array-based binary heap. (Edelkamp et al., 2012) show that if we insist upon using an arbitrary comparator for the heap, the best choice of data structure is an array-based binary heap. This does not, however, settle the issue of what data structure to use for the open list within the context of heuristic searches. Heuristic searches define the priority of search nodes with numbers, and as such, it is possible to take advantage of this to develop a faster implementation. This is the direction that we pursue in this chapter.

Fringe search (Björnsson et al., 2005) is a best-first search algorithm that expands all nodes with minimum $f(n)$ value, and if it fails to find a goal, the minimum $f(n)$ required for expansion is set to be that of the lowest $f(n)$ value from among all the nodes that were not expanded. Instead of storing the nodes in a heap, fringe search puts the nodes in an unsorted linked list, and iterates through the linked list expanding nodes within the bound, and passing over nodes that are not within the bound. In this way, it is reminiscent of iterative deepening A*. For fringe search, the best case is when the domain is such that f layers are generated one at a time. If this happens, in the worst case fringe search will process each node exactly twice: once when the node is generated, and once when the node is expanded, if the nodes is expanded. As we will see below, if many f layers coexist on the open list simultaneously, a node may sit on the open list through several passes, which is why fringe search is not always faster than A*.

(Burns et al., 2012) discusses how to implement A* and iterative deepening A* efficiently. They discuss which data structures to use for creating a fast sliding tile solver. Although the work details precisely how to create a state of the art sliding tiles solver, many of the results have only limited applicability to other heuristic search problems. For example, if the problem in hand does not have unit costs, an array based priority queue simply cannot be used. In addition to that, the work only discusses aggregate performance across a suite of instances. In this context, the "best" data structure is the one whose aggregate performance was best, giving the more difficult instances disproportionate influence over the definition of "best", leaving open the possibility of dynamically selecting the most appropriate data structure at runtime.

3.2 Priority Queues

A priority queue is often implemented using a binary heap backed by a dynamically resizing array of pointers. This approach, for example, is taken by the C++ standard template library and the built in java.util priority queue. A binary heap, however, is not the only choice of data structure. If the domain has unit cost, a priority queue can be implemented by indexing into an array. If the domain uses arbitrary floating point numbers to represent cost, it is also possible to perform better than a binary heap by leveraging the fact that priority is defined by numbers.

We begin with a discussion of a number of popular priority queues, then introduce a new kind of priority queue whose operations are optimized for heuristic searches. We present a mixture of both theoretical and empirical results, as both aspects of runtime analysis are relevant. All algorithms and data structures were implemented in Java and unless otherwise stated, all experiments were performed on a Core2 duo E8500 running at 3.16 GHz with 8 GB RAM under 64-bit Linux.

3.2.1 Benchmarks

We run experiments on a variety of heuristic search benchmark domains to show how the different priority queues perform when faced with various kinds of problems. The first domain we consider is dynamic robot navigation (Likhachev et al., 2003). In dynamic robot navigation, the objective is to route a robot from its start pose to its goal pose while respecting the dynamic motion constraints of the platform.

We also consider the inverse sliding tile puzzle (15), which is the same as an ordinary sliding tile puzzle, except the cost of moving a tile is $\frac{1}{n}$ where n is the face of the tile that was moved. We selected these domains to use for an evaluation because both of these domains have a wide variety of operator costs, and there are many distinct f values.

We consider both large and small problems from each domain. For the dynamic robot navigation domain, the small problems were 50x50 grids with 35% random obstacles, and the large problems were 200x200 grids with line obstacles. For the sliding tile problem, the

large problems were instances that required at least 1 million expansions, and the small problems were instances that required fewer than 1 million expansions.

We also consider five domains that have unit cost. The first is the 3x4 sliding tile puzzle. We used the 3x4 sliding tile puzzle because for all domains, we wanted to know how many nodes would be expanded by A* in the absolute worst case, and finding this piece of information for a larger sliding tile puzzle is impossible due to memory constraints. The next domain we consider is the 40 pancake puzzle with the gap heuristic (Helmert, 2010). We also consider the TopSpin problem with 14 disks, a turnstile that swaps 4 disks, and a pattern database containing information on 7 of the disks. The next problem is the Towers of Hanoi, where we use a problem with 14 disks, 4 pegs, and a disjoint pattern database consisting of 12 and 2 disks. The last problem we consider is grid path planning, where we use a grid that is 1200x2000 with 35% of the cells blocked, and four way motion.

3.2.2 Heaps

The most general kind of priority queue is a heap, because the only requirement is that the objects have a total order. (Edelkamp and Schrödl, 2012) evaluate a number of sophisticated varieties of heaps, but by far the most important of these are binary heaps. The empirical study done by (Edelkamp et al., 2012) convincingly shows that for Dijkstra's algorithm, the best data structure to use for the open list is a binary heap.

3.2.3 Integer Priority queues

If all operators in the domain have the same cost, the priority of a node can be defined using an integer, and bucket data structures can be employed (Dial, 1969). Bucket data structures use an array where each index in the array corresponds to a priority. A* simply requires that nodes come out of the priority queue in f order, and as such, it would be correct to keep the nodes with the same f value in an unordered collection, returning any valid node. This approach is certainly the fastest approach from the perspective of the priority queue operations, but leaving tie breaking beyond f to the priority queue implementation is risky, especially when using an integer priority queue.

Algorithm	Metric	Tiles	Pancake	TopSpin	Hanoi	Grid
A* (Heap)	Time	0.357	1.880	3.497	23.201	1.206
	Exp	**51,358**	**19,786**	94,281	**1,457,667**	**382,535**
	Nodes/Sec	143,859	10,524	26,960	62,827	317,193
A* (Bucket)	Time	0.373	5.943	9.156	18.788	1.137
	Exp	77,266	75,671	292,946	1,795,934	382,533
	Nodes/Sec	207,147	**12,732**	31,994	**95,589**	336,440
A* (Nested Bucket)	Time	**0.352**	**1.644**	3.082	**16.802**	1.084
	Exp	51,448	19,819	94,127	1,458,681	383,385
	Nodes/Sec	146,159	12,055	30,540	86,815	353,676
Fringe	Time	0.364	6.906	**2.878**	18.720	**1.034**
	Exp	76,606	75,604	**92,893**	1,487,347	383,383
	Nodes/Sec	**210,456**	10,947	**32,276**	79,452	**370,776**
A* (All Nodes)	Exp	141,658	486,450	517,437	1,782,655	384,253

Table 3-1: Average performance of Fringe Search and A* with different open list implementations on five benchmark domains. A* (All Nodes) denotes the number of expansions done by A* expanding all nodes in the final f layer, instead of stopping at the first provably optimal solution. Bold denotes the best value for each metric.

Forgoing tie breaking is risky because poor tie breaking can cause a very large number of nodes that are not necessarily needed to be expanded. Using only a single bucket queue sacrifices any ability to do tie breaking on g, which results in more expansions, which in turn leads to a longer runtime. The exact magnitude of this effect can be seen in Table 3-1. For example, on the sliding tile puzzle, bucket A* expands nodes at about 1.5 times the rate of A*, but this additional speed does not result in an overall speedup, because the algorithm expands more nodes because of its poor tie breaking. The superior speed of the single bucket open list is insufficient to compensate for the higher number of node expansions.

To combat this poor behavior, ties in f are often broken in favor of the node with the lower h value, with the idea being that nodes with a small h are probably better than nodes with the same f value but a larger h value, because the nodes with the smaller h are likely to be closer to a goal. In the last f layer, such nodes are goals. If both f and g values are integers it is reasonable to use another integer priority queue to break ties.

Bucket data structures are popular because insertion and deletion can both be done in constant time. The downside of bucket data structures is that they require the priority to be defined by integers, which is often not the case, as many domains have real valued costs.

3.2.4 Fast Float Heaps

We now introduce a new data structure that is optimized for implementing a priority queue where objects have the most important aspect of their priority defined by a real number, and arbitrary tie breaking criteria. The critical observation behind a Fast Float Heap is that when doing a heuristic search, the new items that are added are highly likely to belong near, if not at, the front of the priority queue. The other important observation inspiring the Fast Float Heap is the fact that the majority of nodes are never popped off the open list, so there is little need to have the lower priority nodes completely ordered. Algorithms like Partial Expansion A* (Yoshizumi et al., 2000) and Enhanced Partial Expansion A* (Felner et al., 2012) leverage this fact by not putting the nodes with higher f value on the open list in the first place.

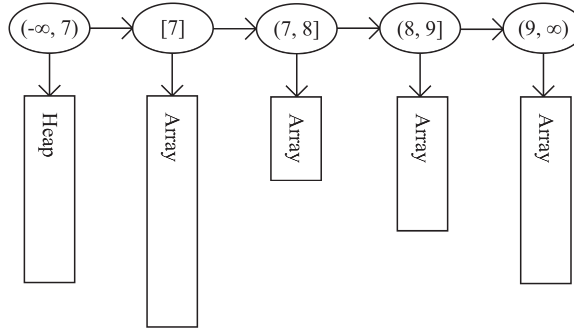

Figure 3-1: Fast Float Heap

A Fast Float Heap uses buckets to store at most b items, or all nodes with a single f value, in a skip list. Initially, there is only a single bucket that contains all the items. Each bucket contains disjoint set of floating point values, and some buckets only contain a single floating point value. When the bucket fills up, the bucket is split into two buckets, with the goal being that each bucket will contain roughly half the items that the original bucket contained. The first bucket in the skip list has its items stored in a binary heap backed by a dynamically resizing array, and the other buckets have their items stored in a dynamically resizing array that is not ordered. This means any node can be accessed by its position index in constant time. Nodes can be added or removed from the unsorted buckets in constant time. A Fast Float Heap uses the same tie breaking criteria as a binary heap would use in the heap for the heap in the head bucket. An example of a Fast Float Heap is shown in Figure 3-1.

Inserting an item involves two steps: first finding the correct bucket, then putting the item in the bucket. If the item belongs in the first bucket, it is put into the first bucket's heap, consuming $O(log(b_{head}))$ time where b_{head} is the size of the head bucket. Ordinarily, b_{head} will be bounded by b, but buckets containing only a single f value are allowed to grow arbitrarily large. If the item belongs in any other bucket, finding the correct bucket will take expected $O(log(\frac{n}{b}))$ time, where n is the number of items in the Fast Float Heap. Inserting the item into the bucket can be done in amortized constant time. Inserting the item into the bucket can be done in amortized constant time because $b-1$ items are inserted

in constant time, and the bth item will take b time, because the bucket has to be split.

Splitting a bucket requires first estimating the median, then creating two new buckets, one for items at or below the median and one for items above the median. We estimated the median by sampling 10% of the bucket's f values, and computing the median based upon this sample. The sample's median value is then used to divide the bucket into two new buckets. Next, items are removed from the old bucket, and put into the correct new bucket. When this process has completed, the new buckets are checked to make sure that the split had the desired effect of splitting the bucket into two roughly equal parts.

If the bucket splitting process failed, it is either because the median estimate was highly inaccurate, or more likely because there is a large number of some f value present. If bucket splitting fails, we assume it is because there is a large number of some f value present rather than a bad estimate of the median. With this in mind, we instead split the original bucket on the mode, making a new dedicated bucket for the mode value, and additional buckets to cover the rest of the interval originally occupied by the original bucket. Last, the new buckets are put into the skip list, and the old bucket is removed from the skip list.

Removing the best item from the Fast Float Heap simply requires popping a node from the heap that is the head bucket, which can be done in $log(b_{head})$ time. If the head bucket is not a heap, it can be turned into a heap in linear time. If the majority of nodes are generated but never expanded, most nodes are only partially sorted into the skip list, and are never put into an actual heap.

Next, we compare the performance of the Fast Float Heap with a binary heap on three different domains: dynamic robot navigation, 15 puzzle with inverse costs, and the Towers of Hanoi.

As can be seen in Table 3.4.1, using the Fast Float Heap results in faster solving times as compared to a basic binary heap on large problems. In Table 3.4.1, we can see that the results in the first table are due to the fast float heap consistently outperforming the binary heap, as evidenced by a sign test.

For smaller problems, the overhead associated with operating the Fast Float Heap is

Problem	Heap	FFH	Exp
Small Robot	**1.087**	1.143	53,111
Large Robot	21.048	**19.558**	978,530
Small Tile	**2.419**	2.491	492,579
Large Tile	44.684	**43.792**	7,263,723
Small Hanoi	**1.99**	2.12	239,653
Medium Hanoi	24.99	**24.89**	472,478
Large Hanoi	49.98	**45.07**	709,477

Table 3-2: Average total solving time of A* using a binary heap and a Fast Float Heap on different problems.

Problem	FFH Faster (count)	Heap Faster (count)
Small Hanoi	1	**50**
Medium Hanoi	**30**	21
Large Hanoi	**50**	1

Table 3-3: Sign test between FFH and binary heap A* solving the Towers of Hanoi

high enough that the net result is a performance loss. Since we are changing only the open list implementation, exactly the same nodes will be expanded assuming the tie breaking function produces a total ordering over all search nodes. If ties are broken in favor of low g but there is no tie breaking criterion required beyond that, the Fast Float Heap will always produce a node of equal quality to A* if the same nodes are present in both data structures, but depending on how the ties in both f and g are broken, the exact set of nodes expanded can vary in practice.

3.3 Fringe Search

Fringe search uses a linked list to represent the open list. The advantage of Fringe Search over A* is the lack of a priority queue. In Fringe Search, adding a node to the open list is as simple as adding a node to a linked list, which can be done in constant time. As a result, Fringe Search has the potential to outperform A*, as long as the other overhead is kept under control.

In domains like grid path planning and sliding tile problems, the heuristic either increases or decreases by the operator cost and all operators cost the same amount. In such domains, every node that is generated either has the same f value as its parent, or an f value that is 1

larger than its parent. These domains are optimal for Fringe Search, because children nodes are either expanded immediately if their f value is the same as the parent, or expanded on the next iteration if the f value is larger than that of the parent. Every node that is generated is either expanded immediately, or expanded on the next pass through the open list.

If a node has an f value that is larger than the next f bound, the node will be passed over on the next iteration, which is clearly a waste. If this waste never or rarely happens, then fringe search can be superior to A*, returning the optimal solution faster.

Since fringe search treats the open list as an unordered list, it does not do any tie breaking. As such, exactly how many nodes the algorithm will expand depends largely upon how the nodes happen to be ordered on the open list. In some domains, the last f layer is not particularly large relative to the other f layers, but in other domains, the last f layer can contain more nodes than all of the previous f layers combined.

On some of these problems, Fringe search consistently outperforms A*, but on other problems, Fringe search is consistently beaten by A*. In Table 3-1, we can see that on all the domains except TopSpin, Fringe Search expands more nodes than A*, and when A* beats Fringe search, it is always because Fringe search expanded substantially more nodes than A*.

One way to determine if Fringe Search is likely to outperform A* is to consider how important tie breaking in the last f layer is. In grid path planning and Towers of Hanoi, the number of nodes expanded by A* is fairly close to the total number of nodes whose f value is less than or equal to the f value of the optimal solution. Thus, in these domains, tie breaking is less important, because the penalty for poor tie breaking is minimal. On domains like this, even if Fringe search has terrible tie breaking, the algorithm will not do much worse than A* in terms of expansions because tie breaking is of minimal importance.

In the other domains, however, tie breaking in the final f layer is critical. In these domains (Tiles, Pancake, and TopSpin) the last f layer contains several times as many nodes as are required by A* to find a solution, so it is important to find the solution early

in the f layer. Another factor to consider is the riskiness of deploying Fringe search. Both Fringe search and A* with a double bucket list have constant time open list operations, but A* with a double bucket list is able to do g tie breaking efficiently. It is possible to do tie breaking with Fringe Search, but doing so can be inefficient. If we force Fringe Search to do tie breaking on g Fringe Search is required to iterate through the open list multiple times at the same f value, which is inefficient.

3.3.1 Fringe Search on Non-Unit Domains

The original Fringe Search paper only discusses unit cost domains, but the authors point out that the same procedure will still work correctly on domains that do not have unit cost operators. The problem with this approach is that nodes might get iterated over more than once. Fringe Search has the same core problem as IDA* in domains with non-unit costs: the next bound might expand only a single new node, while iterating over the entire open list.

The usual antidote to overly conservative bound setting for iterative deepening search algorithms is improved bound selection. In the $IDA{*}_{CR}$ algorithm (Sarkar et al., 1991), the bound is set using a histogram to estimate where the bound should be set in order to double the number of nodes expanded on the next iteration. When this idea is combined with fringe search, the performance is actually dramatically worse as compared to basic fringe search because of the phenomenon of duplicate re-expansion.

When a node is encountered through a suboptimal path, fringe search using conservative bound setting is able to ignore the suboptimal node, because fringe search expands nodes in the same order as A* modulo tie breaking. This means that when the heuristic is consistent, the node is expanded first with its optimal g value. If the bound is more lax, when the suboptimal node is first generated, there is a chance it is still within the bound, and therefore the suboptimal node gets expanded as well. Eventually, the algorithm will find the suboptimal node via the optimal path, and will be forced to re-expand the node, causing a major increase in the total number of required expansions. We implemented fringe search using the same bound setting mechanism from IDA*-CR and the result was

substantially worse than fringe search with basic bound setting.

For Fringe search-CR we did experiments on the small dynamic robot path planning problems and found that 99% of all expansions were re-expansions, representing a large amount of wasted time and computation. We also tried seeding Fringe Search with the optimal solution cost. With the optimal solution cost, Fringe search-CR would not need to estimate what the bound should be, but was rather given the exact number it needed at the outset. Once again, however, the performance very poor due to re-expansions. Even with the optimal solution cost as the initial bound, Fringe search-CR (using an oracle) still re-expands a large number of nodes, requiring more than four times as many node expansions than are required to solve the problem using A*. Based upon this analysis, we conclude that aggressive bound setting for Fringe Search is not a workable solution to the problem of iterating over nodes more than once.

We also tried running Fringe search with the original bound setting rules. We found that for Fringe search running on the same set of small dynamic robot instances, the average iteration expanded 1.39 nodes per iteration. This means that each iteration the algorithm would expand an average of only 1.39 nodes, while iterating over the entire open list, which contains hundreds or even thousands of nodes. On larger on dynamic robot navigation problems, Fringe Search took more than 4 hours to solve an average problem, as compared to 21 seconds for A*.

Fringe Search can be modified to have more than one open list, with the individual open lists corresponding to ranges of f values. If this modification is made, inserting a node is now a two step process: first find the correct open list, and then add the node to that list. If the open lists correspond to individual f values and the f values are all integers then the algorithm is identical to Dial's Algorithm 360 (Dial, 1969).

If using Dial's Algorithm 360 for the open list is not possible because the f values do not line up to integers the problem of bucketing the nodes becomes more complex. The first question is whether or not individual open lists should be allowed to be heterogeneous, and allowed to contain more than one f layer. If this is the case and open lists are allowed to

be heterogeneous the first problem is to figure out exactly which range of values should be placed in each open list such that the nodes are distributed across all of the open lists, but care must be taken not to put too many nodes in the same open list, otherwise the original problem we were trying to solve, iterating over nodes multiple times without expanding them, will return.

If open lists are to be homogeneous and contain only one f layer the problem then becomes sorting the open lists themselves; if the domain is such that every single node has a unique f value this algorithm will be identical to A* with a $\log(n)$ open list.

While Fringe Search may be reasonably competitive in unit cost domains under certain circumstances, we believe that Fringe Search is poorly suited to non-unit cost domains due to the catch-22 regarding how to break up the open list.

3.3.2 Connection to IDA*

Fringe search is very closely related to IDA*. As such, the problems that plague Fringe search on non-unit cost domains are also present when using IDA*. For Fringe search on a non-unit cost domain, there is a choice between re-expanding nodes and passing over nodes on the open list more than once, both of which are detrimental to performance. When faced with a non-unit cost domain, IDA* faces the exact same problem, and is forced to either lose its time complexity bound by failing grow the search frontier exponentially (set the bound conservatively), or to re-expand nodes first encountered along a suboptimal path (set the bound aggressively), both of which are undesirable.

When IDA* expands a node after the first time, it could be expanding the node for one of two reasons (or possibly both). The first possibility is that the node was encountered on a previous iteration, and we are now on a subsequent iteration with a higher bound. A simple example of a node in this category is the root, which is expanded on every iteration. The second reason that a node would be re-expanded is if there are multiple paths to the node. If this is the case, the re-expansion could either be necessary, if the g value has been improved, or unnecessary, if the g value has not been improved.

Problems with the first kind of re-expansion due to the overhead of the first n itera-

tions is well known, and reducing these expansions is the primary objective of the IDA*$_{CR}$ (Sarkar et al., 1991). Problems with re-expansions of duplicates can be ameliorated if we eliminate the unnecessary expansions of duplicates by using a transposition table, but some re-expansions are still necessary if the g value improves.

In order to estimate the effect of duplicate re-expansion, we propose a simple experiment. First, we completely eliminate internal node re-expansions by using a single iteration with a bound of the optimal solution cost. In order to eliminate unnecessary duplicate re-expansions we use a complete global closed list. The reason we use a global closed list instead of a transposition table is the fact that the best performance we can expect out of a transposition table is identical to that of a closed list. We call this algorithm IDA*$_{CR-closed-oracle}$ because it has a full closed list, and uses an oracle to set the bound. Tracking re-expansions done by this algorithm will allow us to ascertain how much of a problem suboptimal duplicate re-expansions can be.

We ran IDA*$_{CR-closed-oracle}$ on a dynamic robot navigation problem where the world is 10x10, and 35% of the cells are blocked. Even with the optimal solution cost as a bound and the benefit of a full closed list to detect duplicates globally, IDA*$_{CR-closed-oracle}$ still performs poorly in this domain. IDA*$_{CR-closed-oracle}$ expands an average of about 500,000 nodes, while the A* solver expands only about 5,000 nodes solving the same problems.

The reason that IDA*$_{CR-closed-oracle}$ performs poorly is directly attributable to duplicate re-expansion. IDA*$_{CR-closed-oracle}$, just like Fringe search-CR, often discovers nodes through suboptimal paths that are still within the bound. Since the suboptimal paths are within the bound, the nodes are expanded, but eventually must also be re-expanded when the node is found through a better path.

One benefit of doing more than one iteration is that the f^* value for nodes whose f^* value is less than the latest iteration is known. For example, if the $f(root) = 10$ and $f^*(root) = 12$ and we do an iteration with a bound of 11, the transposition table could contain data for all nodes whose f^* value is less than or equal to 11, which would allow for the possibility of eliminating some of the suboptimal paths. The down side to this approach

is that by doing more than one iteration, we may expand nodes on more than one iteration. One interesting question is if it is possible to balance these two kinds of re-expansions by setting the bounds appropriately.

While IDA*'s difficulty with real-valued costs and internal node re-expansions are widely known, this phenomenon of suboptimal duplicates was news to us.

3.4 Selecting an Implementation

3.4.1 Problem Features

Before we address the question of how to manage the open list at runtime, an important first step is analyzing how to manage the open list if we had an oracle that was capable of providing information about the problem beforehand. This will help us identify information to look for and estimate at runtime.

One question that must be addressed first is the question of whether or not a priority queue should be used in the first place, deciding between A* and fringe search. As we have seen, due to re-expansions, fringe search is impractical in domains where the operator costs are not uniform. For unit-cost domains, Fringe search can outperform A*, but it does not always do so. Compared to A* with a double bucket list, Fringe search is sometimes marginally faster (7% faster on TopSpin), but in the worst case can be several times slower, as occurs on the pancake problem. For this reason, the safest course of action is to use A* with a double bucket queue.

Another important question that must be addressed is how large the open list is going to be. As we saw in Table and Table , as the open list increases in size, the fast float heap is able to outperform a basic binary heap. A fast float heap is the data structure of choice when the open list is large, but if the search will terminate before reaching the break-even point, we would be better off with some kind of heap. Finding the exact break even point depends on the implementation of the Fast Float heap and the binary heap, but once the data structures have both been implemented, this point can easily be found through empirical analysis.

	Heap	FFH	Open Changes
Fastest	9	21	24
Average CPU time (s)	38.422	37.827	37.656

Table 3-4: A* using 3 different kinds of open lists solving the inverse 15 puzzle

3.4.2 Selection at Runtime

Each of the data structures mentioned previously is best under certain circumstances, but not always best. The information needed to determine what data structure to use is available at runtime. For example, the search algorithm can easily figure out if the domain is unit cost or not by monitoring how h, and g change. Likewise, knowing if the Fast Float Heap will manage the open list better than a binary heap is simply a question of knowing how many nodes are on the open list.

Since a binary heap is the most general kind of open list, it is reasonable to start the search using a binary heap. Another advantage of starting the search with a binary heap is the fact that in the beginning, a binary heap's $O(log(n))$ operations are much closer to the constant time alternatives, so the performance loss is minimal. An alternative is to take an optimistic approach, and start the search with a double bucket queue, and if the double bucket queue encounters problems enqueuing nodes due to an unacceptable f value, the algorithm can switch to a binary heap.

Switching the priority queue on the fly can turn out to be a worthwhile investment. We experimented with switching the open list on the fly for the sliding tile puzzle. Our implementation used a binary heap until the binary heap contained 100,000 nodes, at which point it copied the entire open list to a Fast Float Heap and continued searching using the Fast Float Heap. We then compared the run time of A* with a binary heap, A* with a Fast Float Heap, and A* that switched the open list. The results of this comparison can be seen in Table 3-4. The table shows that swapping the open list at runtime is a worthwhile investment, outperforming both heaps and Fast Float Heaps in terms of overall solving time and instances on which the approach is the fastest overall.

Of course, switching the open list at runtime carries a certain degree of risk. If, for

example, the search terminates when the open list contains 100,001 nodes, there is no benefit to switching, so the time spent changing the open list is wasted. This risk can be mitigated by considering how likely it is that the investment in switching the open list will pay off using estimates of how much search time remains, as was done by Thayer et al. (2012).

3.5 Conclusion

These improvements are subject to the caveat that bookkeeping within the heap is not the only thing going on in a heuristic search. For example, the computer is also allocating and deallocating memory in support of expanding, generating, and deleting nodes. The speedup associated with using a better open list is bounded by the amount of time that the open list consumes.

Despite this theoretical limitation, it is nonetheless possible to extract performance gain out of heuristic search algorithms by selecting the best data structure for the open list. Burns et al. (2012) show that for unit-cost sliding tile puzzles a double bucket heap yields the fastest heuristic search, and our analysis suggests that this conclusion carries to other unit-cost domains such as Towers of Hanoi, pancake. Burns et al. (2012) only discusses A*, but we show that on some unit-cost domains (grid path planning and TopSpin) Fringe search is able to outperform A* with a double bucket heap. By questioning the need for an arbitrary comparator to order nodes in the open list, we were able to extract a speedup of as much as 30% (on Towers of Hanoi).

We have also introduced a new data structure optimized to serve as the open list of a heuristic search, the Fast Float Heap. While the operations on a Fast Float Heap are still $O(\log(n))$, the n is divided by a large constant which helps the speed, and the Fast Float Heap leverages the fact that in a successful best-first search, it is often the case that there are more generations as compared to expansions. Using this data structure, it is possible to speed up A* on non-unit cost domains when the problem is large. We also discussed the benefits of optimizing the data structure at runtime. Once search algorithm identifies the

domain as unit-cost, it can deploy a double bucket queue, which is the best data structure to use for unit-cost domains. Likewise, we showed that using a binary heap for domains with non-unit cost that are small but switching to a Fast Float Heap once the problem gets large can also speed up the search.

Last, we showed that while Fringe search is sometimes able to outperform A*, but sometimes the algorithm's lack of effective tie breaking causes it to expand a large number of nodes, resulting in poor performance. We also discussed how Fringe search assumes that the delay between when a node is generated and when it is expanded is minimal, and that the node will not sit on the open list for a long time continually being passed over because it is outside of the bound. We also saw that changing the bound introduces a different problem, which is that nodes are subject to being re-expanded. We also demonstrated that these exact same problems plague IDA*, making the algorithm unsuited to unit-cost domains. This knowledge helps us to better understand when to deploy fringe search (domains with unit-cost) and IDA* (domains with unit-cost and few duplicates).

CHAPTER 4

Building Effective Heuristics for Greedy Best-First Search

Wilt et al. (2010) showed that Weighted A* is an extremely useful heuristic search algorithm, and there are many situations when it makes sense to use that algorithm. The reason is simple: by increasing the weight on the heuristic, it is often possible to speed up the search. There is, however, a caveat to this general trend: increasing the weight ad infinitium is effective only if greedy best-first search is effective. In some domains, greedy best-first search performs worse than A*, taking more time to find a lower quality solution, a disastrous situation. When we stop to ask what causes greedy best-first search to perform poorly, the only real variable is the heuristic function, meaning that when greedy best-first search performs poorly, the blame lies squarely on the heuristic. The question, then, is how to craft a heuristic that is going to lead to an effective greedy best-first search.

4.1 Introduction

Many of the most effective algorithms in satisficing search have a weight parameter that can be used to govern the balance between solution quality and search effort. The most famous of these is Weighted A* (Pohl, 1970), which expands nodes in f' order, where $f'(n) = g(n) + w \cdot h(n) : w \in (1, \infty)$. Weighted A* is used in a wide variety of applications. For example, the Fast Downward planner (Helmert, 2006) uses Weighted A*, and LAMA (Richter and Westphal, 2010) also uses a variant of Weighted A*. Weighted A* is also used extensively for planning for robots (Likhachev et al., 2003).

In addition to that, Weighted A* is a component of a number of anytime algorithms. For example, Anytime Restarting Weighted A* (Richter et al., 2009) and Anytime Repairing A* (Likhachev et al., 2003) both use Weighted A*. Anytime Nonparametric A* (van den Berg et al., 2011) doesn't directly use Weighted A* like the preceding anytime algorithms, but this algorithm requires greedy search (Doran and Michie, 1966) to find a solution quickly.

All of these anytime algorithms have, built in, the implicit assumption that Weighted A* with a high weight or greedy search will find a solution faster than A* or Weighted A* with a small weight.

In many popular heuristic search benchmark domains (e.g., sliding tile puzzles, grid path planning, Towers of Hanoi, TopSpin, robot motion planning, traveling salesman problem) increasing the weight does lead to a faster search, until the weight becomes so large that Weighted A* has the same expansion order as greedy search, which results in the fastest search. As we shall see, however, in some domains greedy search performs worse than Weighted A*, and is sometimes even worse than A*.

We show that the failure of greedy best-first search is not merely a mathematical curiosity, only occurring in hand crafted counterexamples, but rather a phenomenon that can occur in real domains. Our contribution is to identify conditions when this occurs, knowledge which is important for anyone using a suboptimal search. This is also an important first step in a predictive theoretical understanding of the behavior of suboptimal heuristic search.

The root cause of the failure of greedy search can be ultimately traced back to the heuristic, which is used to guide a greedy best-first search to a goal. For A*, there are a number of well documented guidelines as to what constitutes an effective heuristic. In this chaper we revisit these guidelines in the context of greedy best-first search. We begin by showing that if one follows the well established guidelines for creating a quality heuristic for A*, the results are decidedly mixed. We present numerous examples where following the A* wisdom for constructing a heuristic leads to slower results for greedy search.

Next, we use the lessons learned from the examples to understand the requirements that greedy search places on its heuristic, developing some rules of thumb for building heuristics for greedy best-first search. These methods allow one to identify promising (and unpromising) heuristics for greedy best-first search, helping us to further improve the scalability of one of the most robust, effective heuristic search algorithms.

We present a quantitative metric for assessing a greedy heuristic, Kendall's τ. Kendall's

Domain	Solution Length	Total States	Branching Factor
Dynamic Robot	187.45	20,480,000	0-240
Hanoi (14)	86.92	268,435,456	6
Pancake (40)	38.56	8×10^{47}	40
11 Tiles (unit)	36.03	239,500,800	1-3
Grid	2927.40	1,560,000	0-3
TopSpin (3)	8.52	479,001,600	12
TopSpin (4)	10.04	479,001,600	12
11 Tiles (inverse)	37.95	239,500,800	1-3
City Navigation 3 3	15.62	22,500	3-8
City Navigation 4 4	14.38	22,500	3-10
City Navigation 5 5	13.99	22,500	3-12

Table 4-1: Domain Attributes for benchmark domains considered

τ can be used to predict whether or not greedy best-first search is likely to perform well. Kendall's τ can also be used to compare different heuristics for the same domain, allowing us to make more informed decisions about what heuristic to select, if there is are a variety of choices, as is the case for abstraction-based heuristics like pattern databases. This quantitative metric can be used to automatically construct a heuristic for greedy best-first search by iteratively refining an abstraction and measuring how good each candidate heuristic is.

This work provides both qualitative and quantitative metrics for assessing the quality of a heuristic for the purposes of greedy best-first search. This increases our understanding of one of the most popular and scaleable heuristic search techniques.

4.2 When is increasing w bad?

In order to get a better grasp on the question of when increasing the weight is bad, we first need some empirical data, examples of when increasing the weight is either good, speeding up search, or bad, slowing down search. We consider six standard benchmark domains: the sliding tile puzzle, the Towers of Hanoi puzzle, grid path planning, the pancake problem, TopSpin, and dynamic robot navigation. Since we need to know the optimal solution, we are forced to use somewhat smaller puzzles than it is possible to solve using state of the art suboptimal searches. Our requirement for problem size was that the problem be solvable

by A*, Weighted A*, and greedy search in main memory (eight gigabytes). We selected these domains because they represent a wide variety of interesting heuristic search features, such as branching factor, state space size, and solution length. Basic statistics about each of these domain variants are summarized in Table 4-1.

For the 11 sliding tile puzzle (3x4), we used random instances and the Manhattan distance heuristic. We used the 11 puzzle, rather than the 15 puzzle, because we also consider the sliding tile puzzle with non-unit cost functions, which are not practical to solve optimally for larger puzzles. In addition to that, A* runs out of memory solving a 15 puzzle for the more difficult instances. The non-unit version of sliding tile puzzle we consider uses the inverse cost function where the cost of moving a tile n as $1/n$. The Manhattan distance heuristic, when weighted appropriately, is both admissible and consistent. For the Towers of Hanoi, we considered the 14 disk 4 peg problem, and used two disjoint pattern databases, one for the bottom 12 disks, and one for the top two disks (Korf and Felner, 2002). For the pancake problem, we used the gap heuristic (Helmert, 2010). For grid path planning, we used maps that were 2000x1200 cells, with 35% of the cells blocked, using the Manhattan distance heuristic with four way movement. For the TopSpin puzzle, we considered a problem with 12 disks with a turnstile that would turn either three or four disks, denoted by TopSpin(3) or TopSpin(4) to differentiate between the two varieties. For a heuristic, we used a pattern database with 6 contiguous disks present, and the remaining 6 disks abstracted. For the dynamic robot navigation problem, we used a 200x200 world, with 32 headings and 16 speeds.

We also consider City Navigation problems with varying numbers of connections, but always having 150 cities and 150 places in each city.

Figure 4-1 and 4-2 show the number of expansions required by A*, greedy search, and Weighted A* with weights of 1.1, 1.2, 2.5, 5, 10, and 20. These plots allow us to compare greedy search with Weighted A* and A*, and to determine whether increasing the weight speeds up the search, or slows down the search.

Looking at the plots in Figure 4-1, it is easy to see that as we increase the weight

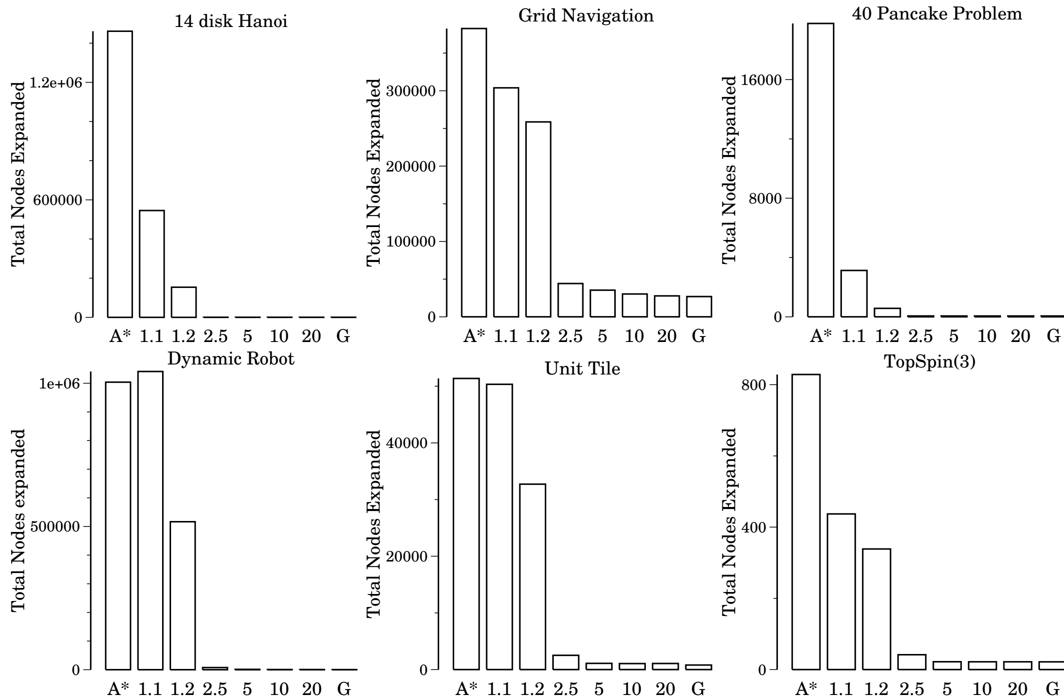

Figure 4-1: Domains where increasing the weight speeds up search

the number of expansions goes down. In Figure 4-2, the opposite is true. In each of these domains, increasing the weight initially speeds up the search, as A* is relaxed into Weighted A*, but as Weighted A* transforms into greedy search, the number of nodes required to solve the problem increases. In two of the domains, TopSpin with a turnstile of size 4 and City Navigation 3 3, the number of nodes expanded by greedy search is higher than the number of nodes expanded by A*.

4.3 Heuristic Requirements

We have established that increasing the weight in Weighted A* does not always speed up the search, and in some situations can actually slow down search. The fact that A* is sometimes faster than greedy best-first search and sometimes slower than greedy best-first search suggests that some heuristics work well for A* and poorly for greedy best-first search, and that some heuristics work well for greedy best-first search but not for A*. Thus, the question is precisely what is driving this difference, and what each algorithm, A* and greedy best-first search, needs out of the heuristic.

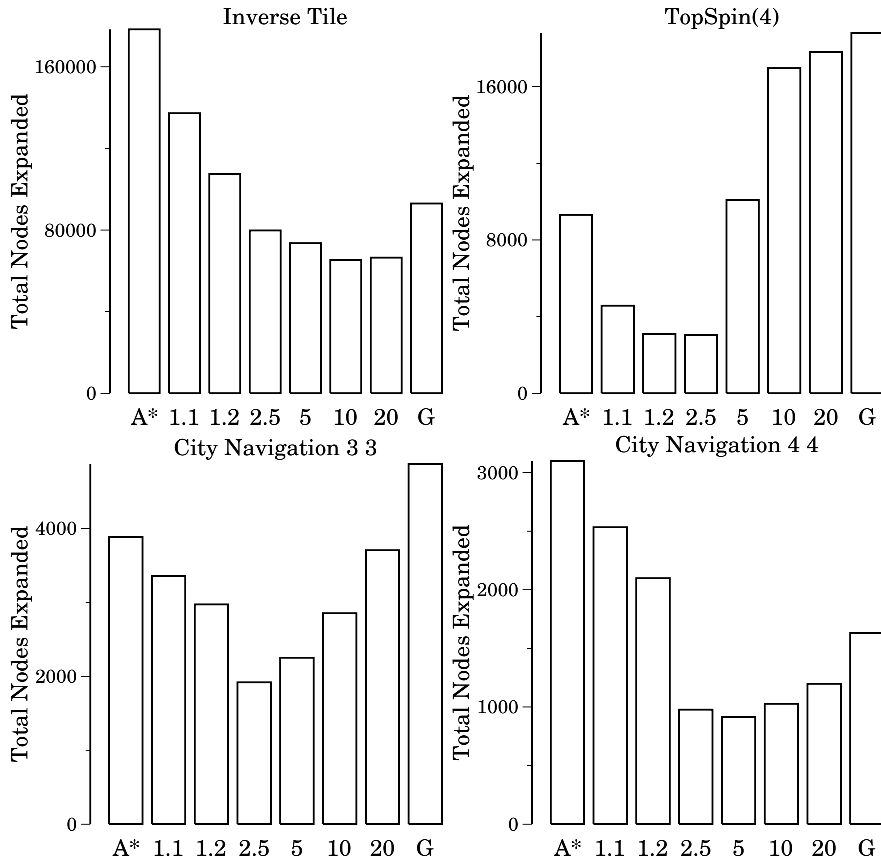

Figure 4-2: Domains where increasing the weight slows down search

We begin by reviewing the literature for suggestions about how to make a good heuristic for A*. With this in mind, we then apply the A* rules for constructing an effective heuristic to greedy best-first search. This leads us to a number of alternative qualitative recommendations for how to build an effective heuristic for greedy best-first search that are different from the recommendations for building a good heuristic for A*.

4.3.1 Building a Heuristic for A*

Much of the literature about what constitutes a good heuristic centers on how well the heuristic works for A*. For finding optimal solutions using A*, the first and most important requirement is that the heuristic be admissible, meaning for all nodes n, $h^*(n)$ – the true cheapest path from n to a goal – is greater than or equal to $h(n)$. If the heuristic is not admissible, A* degenerates into A (no star) which is not guaranteed to find the shortest

path.

It is generally believed that consistency is also important, due to the fact that inadmissible heuristics can lead to an exponential number of re-expansions (Martelli, 1977). Despite the fact that in the general case, an inconsistent heuristic can cause an exponential number of re-expansions, this situation rarely arises in practice (Felner et al., 2011). While inconsistency is not ideal, Felner et al. (2011) argue that it is generally not as much of a problem as is widely believed.

The most widespread rule for making a good heuristic for A* is: dominance is good (Nilsson, 1980; Pearl, 1984). A heuristic h_1 is said to dominate h_2 if $\forall n \in V : h_1(n) \geq h_2(n)$. This makes sense, because due to admissibility, larger values are closer to h^*. Furthermore A* expands every node n it encounters where $f(n) < f(opt)$, so large h often reduces expansions. Dominance represents a gold standard for comparing two heuristics; in fact, heuristics are often informally evaluated by their average value or by their value at the initial state over a benchmark set. In either case, the general idea remains the same: bigger heuristics are better.

If we ignore the effects of tie breaking as well as the effects of duplicate states, A* and the last iteration of IDA* expand the same number of nodes. This allows us to apply the formula from Korf et al. (2001). They predict that the number of nodes we expect IDA* to expand at cost c is:

$$E(N, c, P) = \sum_{i=0}^{c} N_i P(c - i)$$

The variable P in the KRE equation represents the equilibrium heuristic distribution, which is "the probability that a node chosen randomly and uniformly among all nodes at a given depth of the brute-force search tree has heuristic value less than or equal to h" (Korf et al., 2001). This quantity tends to decrease as h gets larger, depending on how the nodes in the space are distributed. The dominance relation also transfers to the KRE equation, meaning that if a heuristic h_1 dominates a different heuristic h_2, the KRE equations predicts that the expected expansions using h_1 will be less than or equal to the expected expansions

Cost	Heuristic	A* Exp	Greedy Exp
Unit	8/4 PDB	2,153,558	36,023
	8/0 PDB	4,618,913	771
Square	8/4 PDB	239,653	4,663
	8/0 PDB	329,761	892
Rev Square	8/4 PDB	3,412,080	559,250
	8/0 PDB	9,896,145	730

Table 4-2: Average number of nodes expanded to solve 51 12 disk Towers of Hanoi problems.

using h_2.

When considering pattern database (PDB) heuristics, assuming the requirements of A* and IDA* are the same also allows us to apply Korf's Conjecture (Korf, 1997), which tells us that we can expect $M \times t = n$, where $M = \frac{m}{1+log(m)}$ with m being the amount of memory the PDB in question takes up, t is the amount of time we expect an IDA* search to consume, and n is a constant (Korf, 2007). This equation tells us that we should expect larger pattern databases to provide faster search. To summarize, bigger is better, both in terms of average heuristic value and pattern database size.

4.4 Greedy Best-First Search with A* Heuristics

As we shall see, these improvements to the A* heuristic are all very helpful when considering only A*. What happens if we apply these same improvements to greedy best-first search? We answer this question by considering the behavior of greedy search on three benchmark problems: the Towers of Hanoi, the TopSpin puzzle, and the sliding tile puzzle.

4.4.1 Towers of Hanoi Heuristics

The first domain we consider is the Towers of Hanoi. The most successful heuristic for optimally solving 4 peg Towers of Hanoi problems is disjoint pattern databases (Korf and Felner, 2002). Disjoint pattern databases boost the heuristic value by providing information about the disks on the top of the puzzle. For example, consider 12 disks, split into two disjoint pattern databases: eight disks in the bottom pattern database, and four disks in the top pattern database. With A*, the best results are achieved when using the full disjoint

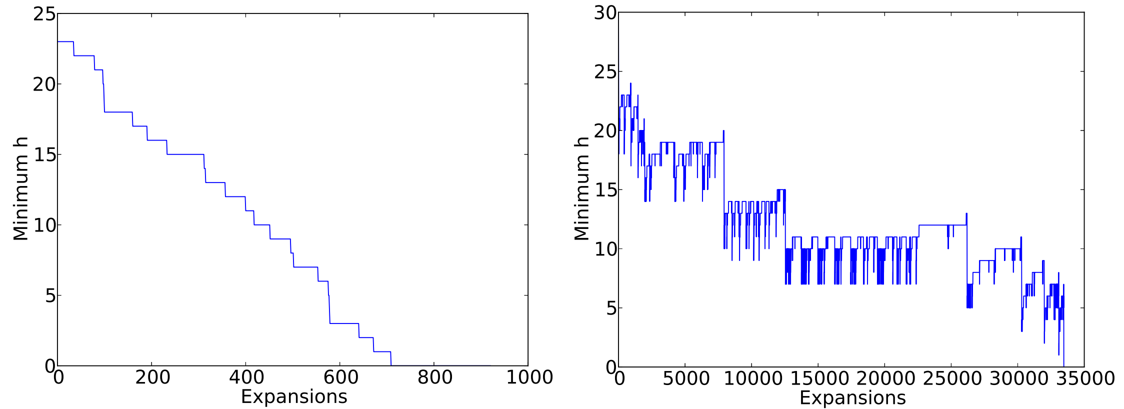

Figure 4-3: The minimum h value on open as the search progresses, using different pattern databases

pattern database. With greedy search, however, observe fewer expansions (and therefore shorter overall solving time) when we do not use a disjoint pattern database, and instead only use the result of the 8 disk pattern database. The exact numbers are presented in the Unit rows of Table 4-2.

The theory for A* corroborates the empirical evidence observed here: the disjoint pattern database dominates the single pattern database, so absent unusual effects from tie breaking, it is no surprise that the disjoint pattern database results in faster A* search.

The reason for the different behaviour of A* and greedy best-first search is simple. With greedy best-first search using a single pattern database, it is possible to follow the heuristic directly to a goal, having the h value of the head of the open list monotonically decrease. To see this, note that every combination of the bottom disks has an h value, and all possible arrangements of the disks on top have will also share that same h value. Since the disks on top can be moved around independently of where the bottom disks are, it is always possible to arrange the top disks such that the next move of the bottom disks can be done, while not disturbing any of the bottom disks, thus leaving h constant, until h decreases because more progress has been made putting the bottom disks of the problem in order. This process repeats until $h = 0$, at which point greedy search simply considers possible configurations of the top disks until a goal has been found.

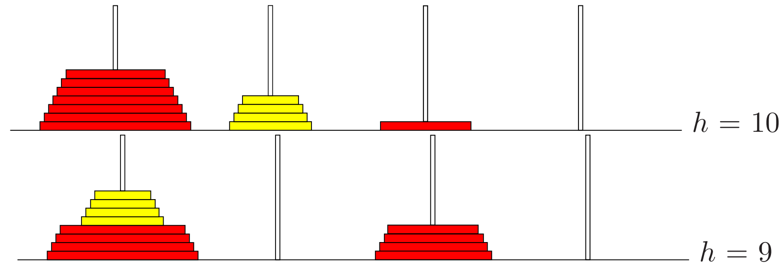

Figure 4-4: Two Towers of Hanoi states, one near a goal (top) and one far from a goal (bottom). Note that the goal peg is the leftmost peg.

This phenomenon can be seen in the left pane of Figure 4-3, where the minimum h value of the open list monotonically decreases as the number of expansions the search has done increases. The heuristic created by the single pattern database creates an extremely effective gradient for the greedy search algorithm to follow for two reasons. First, there are no local minima at all, only the global minimum where the goal is. In this context, we define a minimum as a region of the space M where $\forall n \in M$, every path from n to a goal node has at least one node n' with $h(n') > h(n)$. In addition to that, there are exactly 256 states associated with each configuration of the bottom 8 disks. This means that every 256 expansions, h is guaranteed to decrease. In practice, a state with a lower h tends to be found much faster.

In the right pane of Figure 4-3, the heuristic is a disjoint pattern database. We can see that the h value of the head of the open list fluctuates substantially when using a disjoint pattern database, indicating that greedy best-first search's policy of "follow small h" is much less successful. This is because those states with the bottom disks very near their goal that are paired with a very poor arrangement of the disks on top are assigned large heuristic values, which delays the expansion of these nodes.

This becomes more apparent when we consider a simple example. Consider the two Towers of Hanoi states in Figure 4-4. The top state is significantly closer to a goal, despite having a higher h value than the bottom state. If we ignore the top disks completely, the top state has $h = 1$ compared to the bottom state's $h = 9$, which correctly conveys the fact

that the top state is significantly closer to a goal.

This causes substantial confusion for greedy best-first search, because prior to making any progress with any of the 8 bottom disks, the greedy search considers states where the top 4 disks are closer to their destination. If the bottom state is expanded, it will produce children with lower heuristic values which will be explored before ever considering the top state, which is the state that should be explored first. Eventually, all descendants of the bottom state with $h \leq 9$ are explored, at which point the top state is expanded, but this causes the h value of the head of the open list to go up and down.

To summarize, the disjoint pattern database makes a gradient that is more difficult for greedy search to follow because nodes can have a small h for more than one reason: being near the goal because the bottom pattern database is returning a small value, or being not particularly near the goal, but having the top disks arranged on the target peg. This brings us to our first recommendation.

Recommendation 1 *All else being equal, greedy search tends to work well when it is possible to reach the goal from every node via a path where h monotonically decreases along the path.*

While this may seem self-evident, our example has illustrated how it conflicts with the common wisdom in heuristic construction.

Another way to view this phenomenon is in analogy to the Sussman Anomaly (Sussman, 1975). The Sussman anomaly occurs when one must undo a subgoal prior to being able to reach the global goal. In the context of Towers of Hanoi problems, the goal is to get all of the disks on the target peg, but solving the problem may involve doing and then undoing some subgoals of putting the top disks on the target peg. The presence of the top pattern database encourages greedy searches to privilege states where subgoals which eventually have to be undone have been accomplished.

Korf (1987) discusses different kinds of subgoals, and how different kinds of heuristic searches are able to leverage subgoals. Greedy best-first search uses the heuristic to create subgoals, attempting to follow the h to a goal. For example, in a unit-cost domain, the first

subgoal is to find a node with $h = h(root) - 1$. If the heuristic follows Recommendation 1, these subgoals form a perfect serialization, and the subgoals can be achieved one after another. As the heuristic deviates from Recommendation 1, the subgoals induced by the heuristic cannot be serialized.

These effects can be exacerbated if the cost of the disks on the top is increased relative to the cost of the disks on the bottom. If we define the cost of moving a disk as being proportional to the disks's size, we get the Square cost metric, where the cost of moving disk n is n^2. We could also imagine the tower being stacked in reverse, requiring that the larger disks always be on top of the smaller disks, in which case we get the Reverse Square cost function. In either case, we expect that the number of expansions that greedy search will require will be lower when using only the bottom pattern database, and this is indeed the effect we observe in Table 4-2. However, if the top disks are heavier than the disks on the bottom, greedy search suffers even more then when we considered the unit cost problem, expanding an order of magnitude more nodes. This is because the pattern database with information about the top disks is returning values that are substantially larger than the bottom pattern database, due to the fact that the top pattern database tracks only expensive operators. If the situation is reversed, however, and the top pattern database uses only low cost operators, the top pattern database's contribution to h is a much smaller proportion of the total expansions. Since greedy search performs best when the top pattern database isn't even present, it naturally performs better when the contribution of the top pattern database is smaller.

An example of this can be seen in Figure 4-5. In the left of the figure, the disks in the top pattern database are much cheaper to move than the disks in the bottom pattern database, and are therefore contributing a much smaller proportion of the total value of h. In the right part of the figure, the disks in the top pattern database are much more expensive to move than the disks in the bottom pattern database, so the top pattern database makes a much larger contribution to h, causing substantially more confusion, because states with a low bottom PDB value that are near the goal are sometimes combined with a poor value

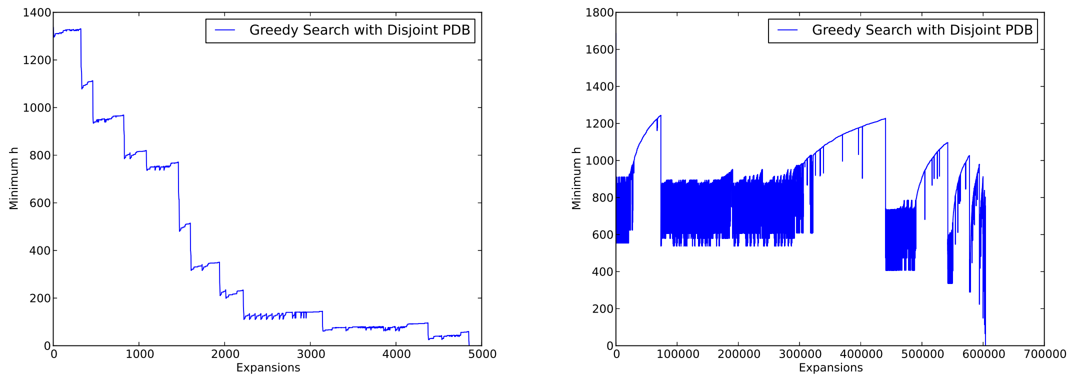

Figure 4-5: The minimum h value on open as the search progresses solving a Towers of Hanoi problem using disjoint pattern databases with different cost functions, square on left, reverse square on right.

for the top pattern database, leading to an overall poor h value for nodes that, in reality, are along a short (measured in node count) path to the goal.

4.4.2 TopSpin Heuristics

In the TopSpin puzzle, the objective is to sort a permutation by iteratively reversing a continuous subsequence of fixed size. We used a simplified problem with 12 disks and a turnstile that flipped 4 disks, because for larger puzzles, greedy search would sometimes run out of memory. We considered pattern databases that contained 5, 6, 7, and 8 of the 12 total disks.

Korf's conjecture predicts that the larger pattern databases will be more useful for A*, and should therefore be considered to be stronger heuristics, and indeed, as the PDB becomes larger, the number of expansions done by A* dramatically decreases. In addition to that, the heuristic created by using a larger pattern database dominates the heuristic from the smaller pattern databases. This can be seen in Figure 4-6. Each box plot (Tukey, 1977) is labeled with either A* or G for (for greedy best-first search), and a number, denoting the number of disks that the PDB tracks. Each box denotes the middle 50% of the data. The horizontal line in the middle of the box represents the median. The circles denote points that are more than 1.5 times the interquartile range away from either the first quartile or

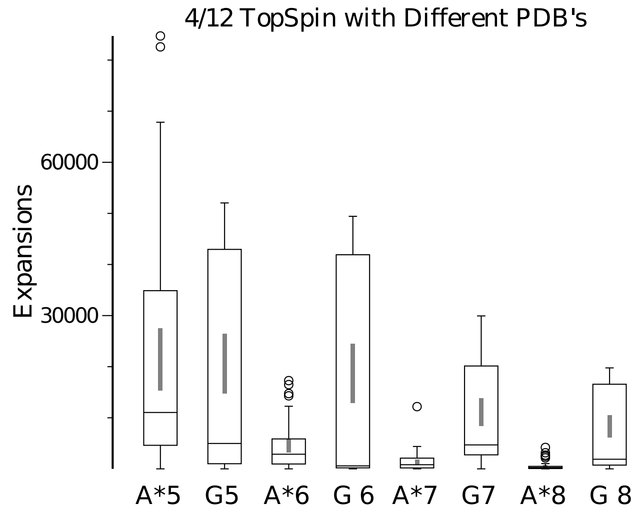

Figure 4-6: TopSpin puzzle with different heuristics

the third quartile, and the whiskers representing the range of the non-outlier data. The gray line represents the confidence interval about the mean. As we move from left to right, as the PDB heuristic tracks more disks, it gets substantially better for A*. While there are also reductions for greedy search in terms of expansions, the gains are nowhere near as impressive as compared to A*.

The reason that greedy best-first search does not perform better when given a larger heuristic is that, with the larger heuristic, states with $h = 0$ may still be quite far from a goal. For example, consider the TopSpin state represented as follows, where A denotes an abstracted disk:

0 1 2 3 4 5 A A A A A A

The turnstile swaps the orientation of 4 disks, but there are configurations such that putting the abstracted disks in order requires moving a disk that is not abstracted, such as:

0 1 2 3 4 5 6 7 8 9 11 10

Moving a disk that is not abstracted will increase the heuristic, so this means that the subgraph consisting of only nodes with $h = 0$ in the TopSpin problem is disconnected. Thus, when greedy search encounters a state with $h = 0$, it may turn out that the node is nowhere near the goal. If this is the case, greedy search will first expand all the $h = 0$ nodes

connected to the first $h = 0$ node, and will then return to expanding nodes with $h = 1$, looking to find a different $h = 0$ node.

The abstraction controls the number and size of $h = 0$ regions. For example, if we abstract 6 disks, there are two strongly connected regions with only $h = 0$ nodes, each containing 360 nodes. If we instead abstract 5 disks, there are 12 strongly connected $h = 0$ regions, each with 10 nodes. For the heuristic that abstracts 6 disks, there is a 50% chance that any given $h = 0$ node is connected to the goal via only $h = 0$ nodes, but once greedy search has entered the correct $h = 0$ region, finding the goal node is largely up to chance. For the heuristic that abstracts 5 disks, the probability that any given $h = 0$ node is connected to the goal via only $h = 0$ nodes is lower. Once the correct $h = 0$ region is found, however, it is much easier to find the goal, because the region contains only 10 nodes, as compared to 360 nodes. Empirically, we can see that these two effects roughly cancel one another out, because the total number of expansions done by greedy search remains roughly constant no matter which heuristic is used. This brings us to our next recommendation.

Recommendation 2 *All else being equal, nodes with $h = 0$ should be connected to goal nodes via paths that only contain $h = 0$ nodes.*

One can view this as an important specific case of Recommendation 1.

4.4.3 Sliding Tiles Heuristics

The sliding tile puzzle is one of the most commonly used benchmark domains in heuristic search. As such, this domain is one of the best understood. Pattern database heuristics have been shown to be the strongest heuristics for this domain (Korf and Taylor, 1996). We use the 11 puzzle (4x3) as a case study because the smaller size of this puzzle allows creating and testing hundreds of different pattern databases. The central problem when constructing a pattern database for a sliding tile puzzle is selecting a good abstraction.

The abstraction that keeps only the outer L, shown in the left part of Figure 4-7 is extremely effective for greedy search, because once greedy search has put all abstracted tiles in their proper places, all that remains is to find the goal, which is easy to do using even a completely uninformed search on the remaining puzzle as there are only $\frac{6!}{2} = 360$

	A	A	3
A	A	A	7
8	9	10	11

	1	A	3
4	A	6	A
A	9	A	11

Figure 4-7: Different tile abstractions. DNF denotes at least one instance would require more than 8GB to solve.

Abstraction	Greedy Exp	A* Exp
Outer L (Figure 4-7 left)	258	1,251,260
Checker (Figure 4-7 right)	11,583	1,423,378
Outer L Missing 3	3,006	DNF
Outer L Missing 3 and 7	20,267	DNF
Instance Specific	8,530	480,250
τ Generated	427	1,197,789
Average 6-tile PDB	17,641	1,609,995
Worst 6-tile PDB	193,849	2,168,785

Table 4-3: Amount of work required by Greedy best-first search and A* to solve 3x4 tile instances with different pattern databases. DNF denotes at least one instance would require more than 8GB to solve.

states with $h = 0$, and the $h = 0$ states form a connected subgraph. Compare this to what happens when greedy search is run on a checkerboard abstraction, as shown in the right part of Figure 4-7. Once greedy search has identified a node with $h = 0$, there is a very high chance that the remaining abstracted tiles are not configured properly, and that at least one of the non abstracted tiles will have to be moved. This effect can be seem in Table 4-3, where the average number of expansions required by A* is comparable with either abstraction, while the average number of expansions required by greedy search is larger by two orders of magnitude.

The sheer size of the PDB is not as important for greedy best-first search as it is for A*. In Table 4-3, we can see that as we weaken the pattern database by removing the 3 tile, the number of expansions required increases only by a factor of 10 for greedy best-first search. When we also remove the 7 tile, the number of expansions required by greedy best-first search increases only by an additional factor of 10. For A*, the PDB with the 3 tile missing 3 instances are unsolvable within 8 GB of memory (approximately 25 million nodes). With both the 3 and the 7 tile missing, A* is unable to solve 16 instances within the same limit.

It is worth noting that even without the 3 tile, the outer L abstraction is still more effective for greedy best-first search as compared to the checkerboard abstraction.

The underlying reason behind the inefficiency of greedy search using certain kinds of pattern databases is the fact that the less useful pattern databases have nodes with $h = 0$ that are nowhere near the goal. This provides further evidence in favor of Recommendation 2; greedy best-first search concentrates its efforts on finding and expanding nodes with a low h value, and if some of those nodes are, in reality, not near a goal, this clearly causes problems for the algorithm. A* is able to eliminate some of these states from consideration by considering the high g value that such states may have.

The checkerboard pattern database also helps to make clear another problem facing greedy search heuristics. Once the algorithm discovers a node with $h = 0$, if that node is not connected to any goal via only $h = 0$ nodes, the algorithm will eventually run out of $h = 0$ nodes to expand, and will begin expanding nodes with $h = 1$. When expanding $h = 1$ nodes, greedy best-first search will either find more $h = 0$ nodes to examine for goals, or it will eventually exhaust all of the $h = 1$ nodes as well, and be forced to consider $h = 2$ nodes. A natural question to ask is how far the algorithm has to back off before it will be able to find a goal. This leads us to our next recommendation.

Recommendation 3 *All else being equal, greedy search tends to work well when the difference between the minimum h value of the nodes in a local minimum and the minimum h that will allow the search to escape from the local minimum and reach a goal is low.*

This phenomenon is clearly illustrated when considering instance specific pattern databases (Holte et al., 2005). In an instance specific pattern database, the tiles that start out closest to their goals are abstracted first, leaving the tiles that are furthest away from their goals to be represented in the pattern database. This helps to maximize the heuristic values of the states near the root, but can have the undesirable side effect of making states that are required to be included in a path to the goal have high heuristic values as well. Raising the heuristic value of the initial state is helpful for A* search, as evidenced by the reduction in the number of expansions for A* using instance specific abstractions of the same size,

shown in Table 4-3. Unfortunately, this approach is still not as powerful for greedy search as the simpler outer L abstraction. This is because some instance specific pattern databases use patterns that are difficult for greedy search to use effectively, similar to the problems encountered when using the checkerboard abstraction.

4.5 Quantifying Effectiveness in Greedy Heuristics

In the previous section, we identified three recommendations for building effective heuristics for greedy best-first search. While these qualitative recommendations are helpful, it is also useful to have a simple, quantitative metric that can robustly compare the aggregate quality of two heuristics. As we will see later, a method can be leveraged to automatically construct heuristics useful for greedy best-first search.

We begin by considering two intuitively reasonable quantitative metrics, the percent error in h, and the correlation between h and h^*. For each of these metrics, we show that the metric cannot be used to predict whether or not greedy best-first search will perform worse than Weighted A*. Then we show that the correlation between h and d^* can be used to predict when greedy best-first search will perform poorly. $d^*(n)$ the same as h^* if we change the graph by making all edges cost 1.

We next explain why the correlation between h and d^* corresponds to our recommendations, and why this metric can be used to differentiate between good and bad heuristics for greedy best-first search.

4.5.1 Percent Error in $h(n)$

The first metric we consider is the percent error in h. The percent error in the heuristic is defined as $\frac{h^*(n)-h(n)}{h^*(n)}$. Since greedy search increases the importance of the heuristic, it is reasonable to conclude that if the heuristic has a large amount of error, relying upon it heavily, as greedy search does, is not going to lead to a fast search.

In Table 4-4, we have the average percent error in the heuristic for each of the domains considered. Surprisingly, the average percentage error bears little relation to whether or not greedy search will be a poor choice. Towers of Hanoi, unit tiles, and TopSpin(3), three

	Domain	Heuristic % Error	$h(n)$-$h^*(n)$ Correlation (Pearson)	$h(n)$-$h^*(n)$ Correlation (Spearman)	$h(n)$-$h^*(n)$ Correlation (Kendall)
Greedy Works	Towers of Hanoi	29.47	0.9652	0.9433	0.8306
	Grid	25.11	0.9967	0.9958	0.9527
	Pancake	2.41	0.9621	0.9593	0.9198
	Dynamic Robot	15.66	0.9998	0.9983	0.9869
	Unit Tiles	33.37	0.7064	0.7065	0.5505
	TopSpin(3)	25.95	0.5855	0.4598	0.4158
Greedy Fails	TopSpin(4)	32.86	0.2827	0.3196	0.2736
	Inverse Tiles	29.49	0.6722	0.6584	0.4877
	City Nav 3 3	44.51	0.5688	0.6132	0.4675
	City Nav 4 4	37.41	0.7077	0.7518	0.6238

Table 4-4: Average % error and correlation between $h(n)$ and $h^*(n)$

domains where greedy search is effective, have as much or more heuristic error than domains where greedy search works poorly. This leads us to conclude that you cannot measure the average heuristic percent error and use this to determine whether or not increasing the weight will speed up or slow down search.

Intuitively, this makes sense, as greedy search only really requires that the nodes get put in $h^*(n)$ order by the heuristic. The exact size, and therefore error, of the heuristic is unimportant, but size has a huge effect on the average percent error. This can be seen if we consider the heuristic $h(n) = \frac{h^*(n)}{N}, N \in \mathbb{N}$, which will always guide greedy search directly to an optimal goal, while exhibiting arbitrarily high average percent error in the heuristic as N increases.

4.5.2 $h - h^*$ Correlation

The next metric we consider is the correlation between h and h^*. While considering the percent error in h as a metric, we noted that greedy best-first search has run time linear in the solution length of the optimal solution if the nodes are in $h^*(n)$ order. Looking at the plot of $h(n)$ vs $h^*(n)$ in the left half of Figure 4-8, we can see that for City Navigation 4 4 there is a reasonable relationship between $h(n)$ and $h^*(n)$, in that the nodes with low $h(n)$ tend to have small $h^*(n)$ values. One way to quantify this observation is to measure

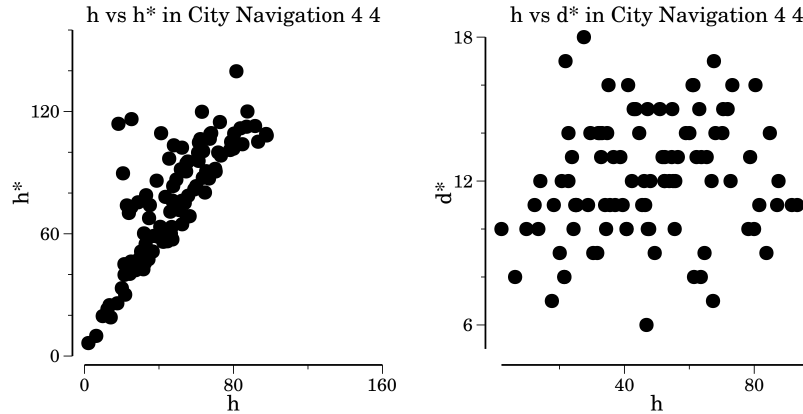

Figure 4-8: Plot of $h(n)$ vs $h^*(n)$, and $h(n)$ vs $d^*(n)$ for City Navigation 4 4

the correlation between the two values. We will do this in three different ways.

The most well known correlation coefficient is Pearson's correlation coefficient r, which measures how well the relationship between $h(n)$ and $h^*(n)$ can be modeled using a linear function. Such a relationship would mean that weighting the heuristic appropriately can reduce the error in the heuristic, which could reasonably be expected to lead to a faster search. In addition, if the relationship between $h(n)$ and $h^*(n)$ is a linear function, then order will be preserved: putting nodes in order of $h(n)$ will also put the nodes in order of $h^*(n)$, which leads to an effective greedy search. For each domain, we calculated the Pearson's correlation coefficient between $h^*(n)$ and $h(n)$, and the results are in the second column of Table 4-4.

Another reasonable way to measure the heuristic correlation is to use rank correlation. Rank correlation measures how well one permutation (or order) respects another permutation (or order). In the context of search, we can use this to ask how similar the order one gets by putting nodes in h order is to the order one gets by putting nodes in h^* order. Rank correlation coefficients are useful because they are less sensitive to outliers, and are able to detect relationships that are not linear.

Spearman's rank correlation coefficient (ρ) is the best known rank correlation coefficient. ρ is Pearson's r between the ranked variables. This means that the smallest of N heuristic values is mapped to 0, the largest of the N heuristic values is mapped to N. This is done

for both h and h^*, at which point we simply calculate Person's r using the rankings. In the context of greedy search, if Spearman's rank correlation coefficient is high, this means that the $h(n)$ and $h^*(n)$ put nodes in very close to the same order. Expanding nodes in $h^*(n)$ order leads to greedy search running in time linear in the solution length, so it is reasonable to conclude that a strong Spearman's rank correlation coefficient between $h^*(n)$ and $h(n)$ would lead to an effective greedy search. For each domain, we calculate the Spearman's rank correlation coefficient between $h^*(n)$ and $h(n)$, and the results are in the third column of Table 4-4.

A much more natural metric for measuring this relationship can be achieved by using Kendall's τ (Kendall, 1938). Kendall's τ is another rank correlation coefficient, but it measures the amount of concordance between the two rankings. Concordance is having the rankings for two elements agree. In the context of greedy search, a concordant pair is a pair of nodes such that $h(n_1) > h(n_2)$ and $h^*(n_1) > h^*(n_2)$ or $h(n_1) < h(n_2)$ and $h^*(n_1) < h^*(n_2)$. Kendall's τ is the proportion of pairwise comparisons that are concordant. If h puts nodes in h^* order, all pairwise comparisons will be concordant, and Kendall's τ will be 1. If h puts nodes in reverse h^* order, all comparisons will be discordant, and Kendall's τ will be -1. If sorting nodes on h puts nodes in a random order, we expect that half of the comparisons will be concordant and half of the comparisons will be discordant.

Kendall's τ can also be understood in the context of bubble sort. The Kendall τ distance is the number of swaps that a bubble sort would do in order to change one list into the other. In this case, it is the number of swaps that a bubble sort would do rearranging a list of nodes sorted on h so the list is sorted on h^*. Kendall's τ is calculated by normalizing the Kendall τ distance, which is done by dividing by $N(N-1)/2$.

Since τ and ρ are both rank correlation coefficients, they are related, but we argue that τ is the more natural statistic. Consider this question: given an open list containing n nodes, how likely is it that the node with the smallest h^* will be at the front of the open list, given that the nodes are ordered on h? We can use τ to predict that the node will, on average, be in the middle of the list if h and h^* are completely unrelated, and closer to the

front of the open list the stronger the correlation between h and h^* is. The reason is that if we assume that the nodes on the open list are a random selection of nodes, τ tells us how often a random comparison is correct. We can therefore use τ to predict how far back the node with the minimum h^* is, which will give us a rough idea of how long we should expect to wait to expand that node. The node with minimum h^* wants to be at the front of the open list, and we can imagine comparing this node to all other nodes on the open list to figure out if the node with minimum h^* is placed in front of the other node, or behind the other node. By hypothesis, the node belongs in front of every other node, and τ measures how often a pairwise comparator returns the correct pairwise ranking. For example, if τ is 0, we expect the pairwise comparisons to be correct half of the time, so this would lead us to expect the node with the minimum h^* would be halfway between the front and the back of the open list. If τ is 0.5, we expect about 75% of all pairwise comparisons are correct, so we would expect to have to look through about 1/4 of the nodes on open before finding the node with minimum h^*. ρ has no such natural interpretation, making τ the more natural statistic.

For Kendall's τ, if we move a node to the wrong side of n nodes, Kendall's τ counts those errors one at a time. This means if the node is put in position 4 by h, when the node really belongs in position 10, there are 6 errors. Kendall's τ tracks this as 6 errors, relative to the total number of errors that are possible. Spearman's ρ takes the error from each individual node, squares that error (turning the 6 into 36), then adds all of the squared errors together, and takes the square root, having no natural interpretation the way Kendall's τ does. It is worth nothing that τ and ρ are generally related to one another, in that one can be used to reliably predict the other (Gibbons, 1985). This relationship means that in practice, it is generally possible to use either metric.

Looking to the data in Table 4-8 leads us to reject the correlation between h and h^* as a metric for predicting how well greedy best-first search will work. For all three correlation coefficients, there are examples of domains where greedy search fails with high $h(n)$-$h^*(n)$ correlations, and examples of domains where greedy search works well with

poor $h(n)$-$h^*(n)$ correlations. For example, in TopSpin(3), we have a Kendall's τ of .41, but this is lower than the τ for Inverse Tiles and both City Navigation problems we consider.

4.5.3 $h - d^*$ Correlation

The objective of greedy search is to discover a goal quickly by expanding nodes with a small $h(n)$ value. If nodes with a small $h(n)$ are far away from a goal, and therefore have a high $d^*(n)$ value (high count of edges to goal), it is reasonable to believe greedy search would perform poorly. The right half of Figure 4-8 shows a plot of $h(n)$ vs $d^*(n)$. We can clearly see that in the City Navigation 4 4 domain, there is almost no relationship between $h(n)$ and $d^*(n)$, meaning that nodes that receive a small $h(n)$ value can be found any distance away from a goal, which could explain why greedy search works so poorly for this domain.

For each domain, we quantify this concept by calculating Pearson's correlation coefficient, Spearman's rank correlation coefficient, and Kendall's τ between $d^*(n)$ and $h(n)$. Intuitively, this is reasonable: greedy search expands nodes with small $h(n)$ values. If nodes with small $h(n)$ values are also likely to have a small $d^*(n)$ values (and these nodes are therefore close to a goal, in terms of expansions away) expanding nodes with small $h(n)$ values will quickly lead to a goal. The converse is also reasonable. If the nodes with small $h(n)$ value have a uniform distribution of $d^*(n)$ values (and thus many of these nodes are far away from a goal in terms of expansions away), expanding these nodes will not quickly lead to a goal.

Looking at Table 4-5, we can see that using both Kendall's τ and Pearson's r, we are finally able to separate the domains on which greedy best-first search performs well and the domains on which greedy best-first search performs poorly. For Kendall's τ, we can draw a line at approximately 0.4 that can be used to separate the domains where greedy best-first search works well and the domains where greedy best-first search works poorly. Likewise, for Pearson's r, we can draw a line at approximately .55.

Because it is a quantitative metric, the Kendall's τ of a heuristic h with d^* can be used to compare heuristics for the same domain. We call this metric the *Goal Distance Rank Correlation*, or GDRC for short. For reasons previously discussed, we believe that Kendall's

	Domain	$h(n)$-$d^*(n)$ Correlation (Pearson)	$h(n)$-$d^*(n)$ Correlation (Spearman)	$h(n)$-$d^*(n)$ Correlation (Kendall)
Greedy Works	Towers of Hanoi	0.9652	0.9433	0.8306
	Grid	0.9967	0.9958	0.9527
	Pancake	0.9621	0.9593	0.9198
	Dynamic Robot	0.9998	0.9983	0.9869
	Unit Tiles	0.7064	0.7065	0.5505
	TopSpin(3)	0.5855	0.4598	0.4158
Greedy Fails	TopSpin(4)	0.2827	0.3196	0.2736
	Inverse Tiles	0.5281	0.5173	0.3752
	City Nav 3 3	0.0246	-0.0338	-0.0267
	City Nav 4 4	0.0853	0.1581	0.1192

Table 4-5: Correlation between $h(n)$ and $d^*(n)$

τ is the best rank correlation coefficient to use, but because of the close relationship between different rank correlation coefficients, it is generally possible to use any rank correlation coefficient.

We ran experiments on the Towers of Hanoi problem using 17 disjoint and non-disjoint pattern databases. We considered pattern databases with between 3 and 8 disks, as well as a selection of pairings of the PDBs where the total number of disks is less than or equal to 12. For each pattern database, we calculated the GDRC for the heuristic produced by the PDB. In Figure 4-9 we plot, for each PDB, the GDRC of the PDB on the X axis and the average of the log of the number of expansions required by greedy best-first search to solve 51 12 disk Towers of Hanoi problems on the Y axis. As we can see from the figure, when the GDRC is below roughly 0.4, greedy search performs very poorly, but as the GDRC increases, the average number of expansions done by greedy best-first search decreases. This suggests that it is possible to use Kendall's τ to directly compare heuristics against one another.

We can see this in the left part of Figure 4-10. Each dot represents one of the 462 possible disjoint pattern databases for the 3x4 sliding tile puzzle with inverse costs. On the Y axis is the log of the average expansions required to solve 100 random instances. On the X axis is the GDRC. Since we are using a non-unit problem, h^* and d^* are not the same, so we can also calculate the correlation between h and h^*. In the right part of Figure 4-10,

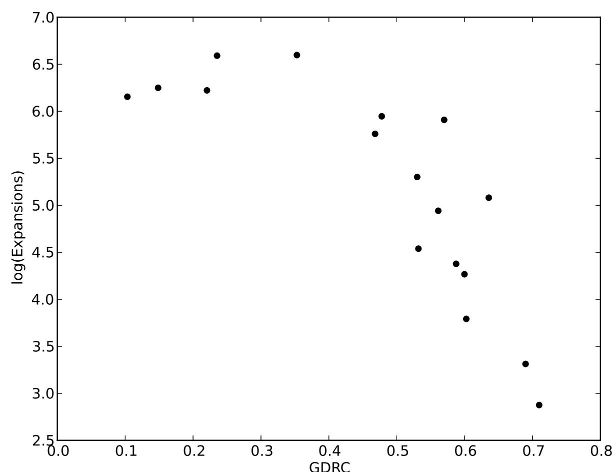

Figure 4-9: Average log of expansions with different heuristics, plotted with different GDRC

this correlation is on the X axis.

In Figure 4-10 we can see that the GDRC when used with h^* is very similar to the GDRC when used with d^*, but that is somewhat an artifact of the fact that h^* and d^* are so closely related in this domain. Despite this relationship, the correlation between expansions and GDRC is larger (and the difference between the correlation coefficients is statistically significant), and the plot with h^* appears to be slightly more skewed to the left.

The correlation between h and d^* neatly predicts when greedy search performs worse than Weighted A* (or A*). It is not perfect, however. If we consider the heuristic $h(n) = h^*(n)$, any measure of the correlation between $h(n)$ and $h^*(n)$ will be perfect, but the relationship between $h(n)$ and $d^*(n)$ for this heuristic can be arbitrarily poor. As the heuristic approaches truth, the $h(n)$-$h^*(n)$ correlations will approach 1, which allows Weighted A* to scale gracefully, as greedy search will have linear run time, no matter what the correlation between $h(n)$ and $d^*(n)$ is. In this situation, looking solely to the correlation between $h(n)$ and $d^*(n)$ to determine whether or not greedy search will be faster than Weighted A* may produce an incorrect answer.

This can be seen in the City Navigation 5 5 domain. City Navigation 5 5 is similar to the other City Navigation problems we consider, except that the cities and places are better

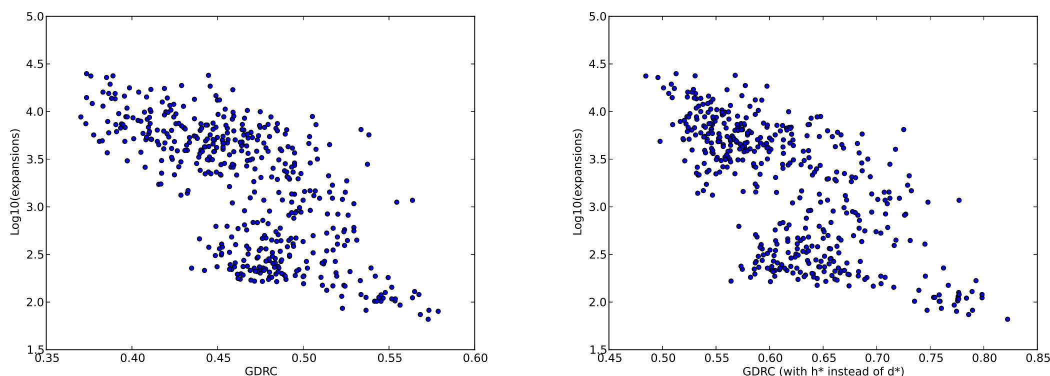

Figure 4-10: Average log of expansions with each of the possible 462 5/6 disjoint PDB heuristics for the 3x4 sliding tile puzzle, plotted against GDRC

Domain	Heuristic % Error	$h - h^*$ r	$h - h^*$ ρ	$h - h^*$ τ	$h - d^*$ r	$h - d^*$ ρ	$h - d^*$ τ
City Nav 5 5	31.19	0.9533	0.9466	0.8230	0.0933	0.0646	0.0501

Table 4-6: Average % error, correlation between $h(n)$ and $h^*(n)$, and correlation between $h(n)$ and $d^*(n)$ in City Navigation 5 5

connected, allowing more direct routes to be taken. Since the routes are more direct, and thus shorter, the heuristic is more accurate.

Table 4-6 shows the various correlations and percent error in $h(n)$ for City Navigation 5 5. Figure 4-11 shows that as we increase the weight, despite the very weak correlation between $h(n)$ and $d^*(n)$, there is no catastrophe: greedy search expands roughly the same number of nodes as Weighted A* with the best weight for speed. This occurs because of the extreme strength of the heuristic, which correlates to $h^*(n)$ at .95, an extremely strong correlation.

The next question is which correlation matters more: $h^*(n)$ or $d^*(n)$. Clearly, a perfect correlation between $h^*(n)$ and $h(n)$ or $d^*(n)$ and $h(n)$ will lead to a fast greedy search, which leads us to the conclusion that in order for greedy search to be effective, nodes with small $h(n)$ that get expanded are required to have at least one virtue: they should either be close to the goal measured in terms of search distance (small $d^*(n)$) or close to the goal

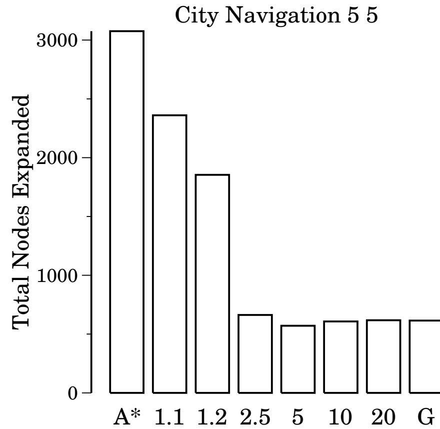

Figure 4-11: Expansions with differing number of neighbors for cities and places

measured in terms of graph distance (small $h^*(n)$). We have seen empirically that as the two correlations break down, the $d^*(n)$ correlation allows greedy search to survive longer: in domains where the $d^*(n)$-$h(n)$ is above .58, greedy search does well. In domains where the $h^*(n)$-$h(n)$ correlation is as high as .70 (or .75, depending on which correlation metric is being used), we have domains where greedy search performs poorly.

The importance of the correlation between $h(n)$ and $d^*(n)$ shows the importance of node ordering for greedy search. In optimal search, the search cannot terminate when a solution is found, but rather when the solution is known to be optimal because all other paths have been pruned. The larger the heuristic values, the sooner nodes can be pruned. This means that in optimal search, heuristic size is of paramount importance: bigger is better. With greedy search, the heuristic is used to guide the search to a solution, so relative magnitude of the heuristic (or the error in the heuristic) has no bearing on the performance of the search, as we saw when we considered the percent error in h.

Some heuristics are able to satisfy both the needs of A* and greedy best-first search simultaneously. For example, the dynamic robot navigation heuristic works extremely well for both A* and greedy best-first search, because it is both big, and therefore good for A*, and good at differentiating nodes that are near the goal and far away from the goal, helping greedy best-first search.

```
1:  function BuildPDB
2:      AllTokens = {Tokens in problem that can be abstracted}
3:      RemainingTokens = AllTokens
4:      BestPDB = build PDB by abstracting AllTokens
5:      BestTau = 0
6:      function tryPDB(tokens)
7:          pdb = build PDB by abstracting AllTokens \ tokens
8:          allNodes = nodes discovered by breadth first search backwards from the goal
    state(s)
9:          sample = randomly sample 10% of the nodes from allNodes
10:         hdCorr = []
11:         for all node ∈ Sample do
12:             hdCorr.append(node.depth, pdb.getH(node))
13:         return calcTau(hdCorr), pdb
14:     while BestPDB.size < Max Allowed Size do
15:         LocalBestPDB, LocalBestTau, LocalBestToken = (None, BestTau, None)
16:         for all token ∈ RemainingTokens do
17:             CurrentTau, PDB = tryPDB(RefinedTokens ∪ {token})
18:             if CurrentTau > LocalBestTau then
19:                 LocalBestTau = CurrentTau
20:                 LocalBestPDB = PDB
21:                 LocalBestTokens = token
22:         if LocalBestPDB ≠ None then
23:             BestPDB = LocalBestPDB
24:             BestTau = LocalBestTau
25:             RemainingTokens = RemainingTokens \ LocalBestTokens
26:         else
27:             Break
28:     return BestPDB
```

Figure 4-12: Hill Climbing PDB Builder

4.6 Building a Heuristic by Hill Climbing on Greedy Distance Rank Correlation

Given a metric for assessing the quality of a heuristic for greedy search, we can use that metric to automatically construct effective abstraction-based heuristics for greedy search. In many domains, a heuristic can be constructed by initially abstracting everything, and slowly refining the abstraction to construct a heuristic.

For example, for the TopSpin problem, we begin with a heuristic that abstracts all disks. We then consider all PDBs that can be devised by abstracting everything except one disk,

PDB	Greedy Exp	A* Exp	Avg. Value
Contiguous	411.19	10,607.45	52.35
Big Operators	961.11	411.27	94.37
Random	2,386.81	26,017.25	47.99

Table 4-7: Expansions to solve TopSpin problem with the stripe cost function using different PDBs

and measure the GDRC of each pattern database. The GDRC can be effectively estimated by doing a breadth-first search backwards from the goal (we used 10,000 nodes for a 12 disk problem) to establish d^* values for nodes, and the h value can be looked up in the pattern database. We then sample 10% of the nodes generated in this way, and used the sample to calculate an estimate of Kendall's τ. Last, we take the PDB with the highest value as the incumbent PDB. This process repeats until either all PDB's have a worse GDRC than the previous best PDB, or until the PDB has reached the desired size. The process is detailed in Algorithm 1.

When used to generate unit-cost TopSpin pattern databases, this process always produced PDBs where the abstracted disks were all connected to one another, and the refined disks were also all connected to one another. This prevents the abstraction from creating regions where $h = 0$, but where the goal is nowhere near the $h = 0$ nodes, per Recommendation 2.

With unit-cost TopSpin problems, abstractions where all of the disks are connected to one another work well for both greedy search and A*. If we change the cost function such that moving an even disk costs 1 and moving an odd disk costs 10, we get the stripe cost function, so called because the costs are striped across the problem. The most effective PDBs for A* are those that keep as many odd disks as possible, because moving the odd disks is much more expensive than moving an even disk. If we use such a pattern database for greedy search, the algorithm will align the high cost odd disks, but will have great difficulty escaping from the resulting local minimum. If we use hill climbing on GDRC to build a heuristic for greedy search on such a problem, we end up with a heuristic that keeps the abstracted and the refined disks connected to one another. In other words, hill climbing

on GDRC produced a pattern database where all of the abstracted disks were adjacent to one another, and all of the refined disks were also adjacent to one another.

Table 4-7 shows the results of using Greedy best-first search and A* on a 10 disk TopSpin puzzle with the stripe cost function using various 6 disk pattern databases. The Contiguous PDB selected by hill-climbing on GDRC works much better for greedy search as compared to the Big Operators pattern database that tracks only the expensive disks. A*, on the other hand, performs much better when using the Big Operators pattern database, because the average values found in this pattern database are much higher. We can see the importance of creating a good pattern database when we consider the random row from Table 4-7, which contains the average number of expansions from 20 different randomly selected 6 disk pattern databases.

We can also compare the GDRC-generated PDBs to instance specific PDBs for the 3x4 sliding tile puzzle (Holte et al., 2005). For the sliding tile puzzle, we refined one tile at a time, selecting the tile that increased τ the most. On this domain, in order to get an accurate estimate of τ, we had to increase the number of nodes expanded going backwards from 10,000 to 1,500,000. Following the hill climbing procedure, the algorithm selected a pattern database that tracked the 1, 3, 4, 7, 8, and 11 tiles. The results of using this PDB are shown in Table 4-3. While this abstraction is not as strong as the outer L abstraction, it is the fourth best PDB for minimizing the average number of expansions done by greedy search of the 462 possible 6-tile pattern databases. The automatically constructed PDB results in greedy best-first search running faster by two orders of magnitude faster as compared to an average 6-tile PDB, and 3 orders of magnitude faster than the worst 6-tile PDB for greedy best-first search. The GDRC-generated PDB works substantially better for greedy best-first search as compared to instance specific PDBs, requiring about 1/20 of the expansions. One additional advantage the GDRC-generated PDB has over instance specific PDBs is the fact that GDRC produces a single PDB, unlike instance specific PDBs which produce a new PDB for every problem. One interesting question is if it is possible to extract additional performance out of greedy best-first search by selecting a pattern database for each problem.

This shows that GDRC is useful for predicting the relative quality of heuristics for greedy best-first search. It also showed that it is possible to leverage this quantitative metric to automatically construct a heuristic for greedy best-first search, and that the automatically created heuristics are extremely effective for greedy best-first search.

4.7 Related Work

Gaschnig (1977) describes how to predict the worst case number of nodes expanded by A*, and also discusses how weighting the heuristic can affect the worst case final node expansion count. His predictions, however, have two limitations. First, the predictions assume the search space is a tree, and not a graph, as is the case for many applications of heuristic search. In addition to that, the worst case predictions only depend on the amount of error present in the heuristic, where error is measured as relative deviation from $h^*(n)$. For A*, this criterion makes a certain amount of sense, but for greedy best-first search, we have seen that relative deviation from $h^*(n)$ cannot be used to predict when greedy best-first search will perform poorly. Gaschnig points out that increasing the weight ad infinitium may decrease performance, which is something we have observed here in practice.

Chenoweth and Davis (1991) show that if the heuristic is "rapidly growing with logarithmic cluster", a greedy best-first search can be done in polynomial time. A heuristic is rapidly growing with logarithmic cluster if, for every node n, $h(n)$ is within a logarithmic factor of a monotonic function f of $h^*(n)$, and f grows at least as fast as the function $g(x) = x$. On a graph of finite size, any heuristic with the property that $h(n) = 0$ if and only if n is a goal and $h(n) = \infty$ only if $h^*(n) = \infty$ will qualify as a rapidly growing with logarithmic cluster. The claims presented by Chenoweth and Davis (1991) are significant when $h^*(n)$ can grow arbitrarily large, which can only occur when the graph is infinite. For all benchmark domains we consider, this is not the case. It remains an open question to ascertain whether or not the asymptotic analysis of Chenoweth and Davis (1991) is of significance in practice on finite graphs. For example, there are problems for which Weighted A* and Greedy best-first search perform poorly, despite the existence of this bound.

A number of works consider the question of predicting search algorithm performance (Korf et al., 2001; Pearl, 1984; Helmert and Röger, 2007), although the subject attracting by far the most attention is determining how many nodes will be expanded by an optimal search algorithm, which does not lend any insight into the question of whether or not greedy search is likely to perform well or poorly. Lelis et al. (2011) did empirical analysis of suboptimal search algorithms, predicting the number of nodes that would be expanded by Weighted IDA*, but it remains an open question determining whether or not the Weighted IDA* predictions can tell us if increasing the weight too far can be detrimental for that algorithm. For example, one open question is to ascertain whether or not the predictions are able to predict higher number of expansions could be associated with a higher weight.

Korf (1993) provides an early discussion of how increasing the weight may actually be bad, showing that when recursive best first search or iterative deepening A* is used with a weight that is too large, expansions actually increase. This paper is also an early example of exploring how the weight interacts with the expansion count, something central to this chapter.

Hoffmann (2005) discusses why the FF heuristic (Hoffmann and Nebel, 2001) is an effective way to solve many planning benchmarks when used in conjunction with enforced hill climbing. The paper shows that in many benchmark problems, the breadth-first search part of the enforced hill climbing algorithm is bounded, which means that those problems can be solved quickly, sometimes in linear time. Although enforced hill climbing is a kind of greedy search, its behaviour is very different from greedy best-first search when a promising path turns into a local minimum. Greedy best-first search considers nodes from all over the search space, possibly allowing very disparate nodes to compete with one another for expansion. Enforced hill climbing limits consideration to nodes that are near the local minimum (with nearness measured in edge count), which means that the algorithm only cares about how the heuristic performs in a small local region of the space. Hoffmann (2011) extends this concept, describing a process for automatically proving that a domain will have small local minima.

Xu et al. (2009) discuss construction heuristics for a greedy search, but the algorithm they consider is a beam search. Beam searches inadmissibly prune nodes to save space and time, so their function is ultimately being used not to rank nodes, but to make a decision as to whether or not to keep any one node. The function that Xu et al. create can be used to rank nodes, so one natural question to ask is how well the learned function works for ranking nodes for greedy best-first search, which is an interesting avenue for future work. Another natural question to ask is if the learned function is good for guiding a greedy best-first search, how well the learned function is able to replicate d^*.

4.8 Conclusion

We first showed that greedy search can sometimes perform worse than A*, and that although in many domains there is a general trend where a larger weight in Weighted A* leads to a faster search, there are also domains where a larger weight leads to a slower search. It has long been understood that greedy search has no bounds on performance, but our work shows that poor behavior can occur in practice.

We then showed that the domains where increasing the weight degrades performance share a common trait: the true distance from a node to a goal, defined as $d^*(n)$, correlates very poorly with $h(n)$. This information is important for anyone running suboptimal search in the interest of speed, because it allows them to identify whether or not the assumption that weighting speeds up search is true or not, critical knowledge for deciding what algorithm to use.

We have shown several examples in which the conventional guidelines for building heuristics for A* can actually harm the performance of greedy best-first search. We used this experience to develop alternative recommendations and desiderata for heuristics for use with greedy best-first search. The first is that from every node, there should be a path to a goal that only decreases in h. The second, an important special case of the first, is that nodes with $h = 0$ should be connected to a goal via nodes with $h = 0$. The third recommendation is that nodes that require including high h nodes in the solution should have as high an h

value as possible.

We showed that Kendall's τ is a useful metric to compare different heuristics for greedy search, and demonstrated how it can be used to construct appropriate heuristics for greedy best-first search.

Given the importance of greedy best-first search in solving large problems quickly, we hope this investigation opens the door to further analysis of suboptimal search algorithms and the heuristic functions they rely on.

Understanding the assumptions that greedy best-first search makes of its heuristic function are critical to getting the algorithm to perform well. This understanding can be used to do two extremely useful tasks. First, it helps us to design heuristics that are compatible with greedy best-first search. Second, if we cannot change the way the heuristic is built to be more amenable to greedy best-first search, this knowledge tells us that we should consider alternative kinds of heuristic, or alternative kinds of heuristic search.

CHAPTER 5

Why d is Better than h

5.1 Operator Costs

In the previous section, we saw that a strong correlation between h and d^* is important for greedy best-first search. This suggests that there is something special about d^*, which represents a count of edges, and treats the graph as if all edges have the same cost.

Wilt et al. (2010) showed that operator costs are an extremely important domain feature, so a natural extension of this work is to figure out why this was the case. This has the added benefit of connecting back to why heuristics that correlate well with d^* are so effective for greedy best-first search.

We begin by discussing how the operator costs affect optimal heuristic search. For A* (and Weighted A*) the fact that g is incorporated into the evaluation function induces a bound on the number of extra expansions in the local minimum, as long as there are no zero cost operators (Wilt and Ruml, 2011). Unfortunately, for greedy best-first search there is no bound on the upper bound of the number of nodes that can be introduced onto the open list by a single large heuristic evaluation error, because greedy best-first search does not consider g, so if the local minimum is infinite, greedy best-first search will never terminate.

Theoretical analysis of how the operator costs interact with different kinds of heuristic searches can inform us about the worst case behavior, which in turn can lend insight into what is happening in the more common situations we tend to observe in practice.

5.1.1 Costs Make Optimal Search Difficult

Optimal heuristic search is much more restrictive as compared to satisficing search, so we begin by analyzing how operator costs affect optimal heuristic search.

The first domain we consider is the sliding tile puzzle. In a normal sliding tile puzzle, the cost of moving any tile is 1. We now consider a variant where the cost of moving a

	2	1	3
4	5	6	7
8	9	10	11
12	13	15	14

?	?	?	?
?	?	?	?
?	?	?	
?	?	15	14

Figure 5-1: Left: 15 puzzle instance with a large heuristic minimum. Right: State in which the 14 tile can be moved.

Cost (ratio)	Algorithm	Exp	Cost	Length
Unit (1:1)	A*	76,599	28	28
Face (1:15)	A*	482,948	210	28
Face^2 (1:225)	A*	3,575,939	DNF	DNF

Table 5-1: Difficulty of solving the puzzle from Figure 5-1

tile is the face value of the tile, or the face value of the tile squared. On the instance shown in the left part of Figure 5-1, the Manhattan distance heuristic will underestimate the cost of the root node by a very large margin. In order to get to the solution, the 14 and the 15 tiles have to switch places. In order for this to occur, one of the tiles has to move out of the way. Let n denote a configuration as in the right part of Figure 5-1, where we can move the 14 tile up 1 slot, to make room for the 15 tile, and the other tiles are arranged in any fashion. If we assume the optimal path involves first moving the 14 out of the way (as opposed to first moving the 15 out of the way), this node, or one like it, must eventually make its way to the front of the open list. Let s_{14} denote the node representing the state where we have just moved the 14 tile. If we have unit cost, $f(s_{14})-2=f(n)$, since $g(s_{14})=g(n)+1$ and $h(s_{14})=h(n)+1$. If cost is proportional to the tile face, then we have $f(s_{14})-28=f(n)$, since $g(s_{14})=g(n)+14$ and $h(s_{14})=h(n)+14$. If cost is proportional to the square of the tile face, then we have $f(s_{14})-392=f(n)$, since $g(s_{14})=g(n)+14^2$ and $h(s_{14})=h(n)+14^2$, and $2\cdot 14^2=392$. Since s_{14} is along the optimal path, eventually it must be expanded by A*. Unfortunately, in order to get this node to the front of the open list A* must first expand enough nodes such that the minimum f on the open list is $f(n)+\{2,28,392\}$, where the appropriate value depends upon the cost function under consideration.

Cost Function	Failure Rate	95% Confidence Interval
Unit	21%	7.9%
Cost = $\sqrt{face}$	48%	9.6%
Cost = $\frac{1}{face}$	66%	9.5%

Table 5-2: Failure rate of A* on tile puzzles with different cost functions, and confidence interval for failure rate

The core problem is that raising the f value of the head of the open list in non-unit domains can require expanding a very large number of nodes due to the low cost operators. We can observe this phenomenon empirically by considering Table 5-1, where we can see that the more we vary the operator costs, more nodes must be expanded to get out of the local minimum associated with the root node.

The same general problem presents itself when we consider random 15-puzzles. We ran A* on the 100 instances published by Korf (1985a), using three different cost functions: unit, square root, and inverse. As the cost of the cheapest action compared to the cost of the most expensive action increases, the number of failures increases. We define failure of a search to be exhausting main memory, which is 8 GB on our machines, without finding a solution. As can be seen in Table 5-2, as we increase the ratio of operator costs from 1 in unit to 3.9 in square root tiles to 15 in inverse tiles, the problems get more difficult. These differences are statistically highly significant, in that there is almost no overlap in the confidence intervals.

In Table 5-3 we can see how the raw expansions are affected for A* on a slightly easier problem, the 3x4 sliding tile puzzle, where A* is always able to find a solution. For non-unit problems, here we use square root, in which the cost of moving a tile n is $\sqrt{n}$, and inverse, in which the cost of moving tile n is $\frac{1}{n}$. Here, we see the same general trend, where unit cost problems are easiest to solve, and as we increase the ratio of the largest operator to the smallest operator, the problems become more difficult for A* to solve.

The same phenomenon can be seen in grid path planning. As can be seen in Table 5-3, the unit cost problems are substantially easier to solve using A*. For grid path planning, the

Problem	Heuristic	Cost	Ratio	Expansions
3x4 Sliding Tile	Manhattan	Unit	1:1	51,358
		Sq Root	$\sqrt{15}$:1	62,402
		Inverse	15:1	178,357
Grid	Manhattan	Unit	1:1	382,536
		Life	1199:1	994,437
13 Pancake	7 cake PDB	Unit	1:1	23,822
		Sum	78:1	217,289
12 disk 4 peg Hanoi	8/4 disjoint PDB	Unit	1:1	2,153,558
		Square	144:1	239,653
		Rev Sq	144:1	3,412,080
10 disk 4 turnstile TopSpin	6 disk PDB	Unit	1:1	208
		Stripe	10:1	411
		Sum	3.4:1	11,463

Table 5-3: A* expansions on different problems

non-unit cost function we consider is Life, in which the cost of exiting a cell is proportional to the cell's Y value.

For the pancake problem, the non-unit cost problem we consider is Sum, where the cost of making a flip is the sum of the indexes of the pancakes that are being flipped. For the pancake problem with sum costs, we can see that the non-unit sum cost function requires about 10 times the number of expansions than its unit cost cousin.

For the Towers of Hanoi, we consider the square cost function, and the reverse square cost function. For both problems, the cost of moving a disk d is d^2, proportional to the size of the disk. In the square problem, disks are stacked normally, but in the reverse square problem, the stack of disks is inverted, and large disks are required to be on top of smaller disks. On the Towers of Hanoi, the reverse square problems are more difficult to solve than the unit cost problems, providing us another example of a non-unit cost problem that is more difficult to solve than its unit-cost counterpart. When we consider the square costs, however, we see that this trend does not always hold, and there are at least some situations where having a wide variety of operator costs makes the problem easier to solve.

For the TopSpin problem, we consider two non-unit cost functions. The first is Sum, in which the cost of using the turnstile is the sum of the indexes of the disks being turned. In

the TopSpin problem with stripe costs, the cost of using the turnstile is proportional to the sum of the costs of the disks being moved, except the even disks cost 1 and the odd disks cost 10. On the TopSpin problem, we once again see that the unit-cost problems are easier to solve than non-unit cost problems, but the stripe problems are easier to solve optimally than the sum problem, despite the fact that the sum problem as a smaller operator cost ratio.

We have seen that a simple modification to the cost function of five standard benchmark domains makes the problems require significantly more computational effort to solve optimally using A*, with one notable exception, the Towers of Hanoi with square costs. These examples help to establish the fact that non-unit cost domains can be much more difficult as compared to a unit cost domain with the same state space and connectivity.

Heuristic Errors are More Costly with h

We next show one reason why A* runs into problems on non-unit cost problems, even when all other aspects of the problem remain the same as in the unit-cost counterpart.

The problem occurs when the amount the heuristic can change across a transition is bounded, which we show is quite common in heuristic search problems. This condition holds when we have a consistent heuristic in a domain with invertible operators, but we argue that some heuristics have this property even if the operators in the domain are not invertible. We use this bound on the rate at which h can change to prove bounds on the rate at which error in the heuristic can change. We then apply these results to draw conclusions about the size of heuristic depressions in non-unit cost domains.

Felner et al. (2011) show (in their Equation 3) that in a graph with invertible operators, an alternative definition of consistency is:

$$\forall n, s : \Delta_h(n, s) = |h(n) - h(s)| \leq c(n, s) \tag{5.1}$$

This is one method of establishing a bound on how $\Delta_h(n, c)$ can change along a path. Note that having invertible operators is not the only way in which a bound on Δ_h can be

established. Some heuristics inherently have this property. For example, in the dynamic robot motion planning domain, operators are often not reversible. In this domain, the usual heuristic is calculated by solving the problem without any dynamics. Without dynamics, operators can be reversed, so the rate at which the heuristic can change is bounded by the cost of the transition.

Establishing a bound on Δ_h allows us to establish properties about not only the heuristic error, but also about the behavior of search algorithms that rely upon the heuristic for guidance.

Bounds on the rate at which h can change have an important consequence for how the error in $h(n)$, defined as $\epsilon(n) = h^*(n) - h(n)$, can change across any transition.

Theorem 3 *In any domain with a consistent heuristic and invertible operators, for any pair of nodes n, s such that s is a successor of n:*

$$\Delta_\epsilon(n, s) = |\epsilon(n) - \epsilon(s)| \leq 2 \cdot c(n, s) \tag{5.2}$$

Proof: This equation can be rewritten as

$$|(h^*(n) - h(n)) - (h^*(s) - h(s))| \leq 2 \cdot c(n, s) \tag{5.3}$$

which itself can be rearranged to be

$$|(h^*(n) - h^*(s)) - (h(n) - h(s))| \leq 2 \cdot c(n, s) \tag{5.4}$$

The fact that h^* is consistent and operators are invertible implies that the most h^* can change between two nodes n and s where s is a successor of n is $c(n, s)$. The same applies to h. Since the change in both h and h^* is bounded by the $c(n, s)$, the difference between the $\epsilon(n)$ and $\epsilon(s)$ is at most twice $c(n, s)$. The 2 is necessary because $(h^*(n) - h^*(s)) = \Delta_{h^*}$ and $(h(n) - h(s)) = -\Delta_h$ may have opposite signs. □

This bound Δ_ϵ places important restrictions on the creation of a local minimum in the heuristic. When we have a consistent heuristic and invertible operators and all edges have similar cost, in order to accumulate a large heuristic error, the heuristic has to provide incorrect assessments for several transitions. For example, in order for a heuristic in a unit

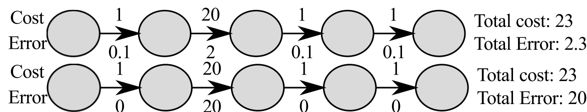

Figure 5-2: Examples of heuristic error accumulation along a path

cost domain to have an error of 4, there must be at least two transitions where the heuristic does not change correctly; it is not possible for any single transition to add more than 2 to the total amount of error present in h, because h^* can change by at most 1, and h can also only change by at most 1.

In a non-unit cost domain, the error associated with a high cost operator can contribute much more to the overall heuristic error as compared to the error potentially contributed by a low cost operator. The result of this is that, in a domain with non-unit cost, heuristic error does not accumulate uniformly because the larger operators can contribute more to the heuristic error. This does not occur in domains with unit costs, because each operator can contribute exactly the same amount to the heuristic error. A simple example of this phenomenon can be seen in the top part of Figure 5-2; all operators contribute the same percent to the total error, but due to size differences, a single large operator contributes almost all of the error in h. If we consider a different model of error where any operator can either add its size to the total error, or not contribute to the error, we have the example in the bottom part of Figure 5-2, where a single large operator can contribute so much error to a path that the effects of the other operators is insignificant. This can occur when, for example, an important aspect of the domain has been abstracted away in the heuristic. In either case, the large operators have the ability to contribute much more error than the smaller operators. If there are no large operators, the rate at which error can accumulate is much slower.

Any time the heuristic of an expanded node deviates from $h^*(n)$ A* is liable to expand extra nodes. We next show that the number of extra nodes is exponential in the ratio of the size of the error to the smallest cost operator.

Corollary 2 *In a domain where* $\Delta_h(n,s) \leq c(n,s)$ *with an operator of cost* δ*, if a node* n *with heuristic error* $\epsilon(n)$ *is expanded by A* and all descendants of* n *have* b *applicable operators of cost* δ*, at least* $b^{\epsilon(n)/2\cdot\delta}$ *extra nodes will be introduced to the open list as compared to the number of expansions that would be done by A* with* $\epsilon(n) = 0$*, unless there are duplicate nodes or dead ends to prematurely terminate the search within the subtree of descendants of* n*.*

Proof: Consider the situation where an A* search encounters a node n that has a heuristic error of size $\epsilon(n)$. This means that $g(n) + h(n) + \epsilon(n) = g_{opt}(goal)$. In order to expand the goal, all descendants s of n which have $f(s) < g_{opt}(goal)$ must first be expanded.

We have assumed that the change in h going from the parent to the child is bounded by the cost of the operator used to get from the parent to child. We have also assumed that we can apply b operators with cost δ. Since b operators with cost δ can be applied, then we have the result that f can increase by at most 2δ per transition, at least while in the subtree consisting of operators of cost δ. Note that there can be additional nodes reachable by different operators, but these nodes are not counted by this theorem.

This bound on the change in f stems from the fact that $g(s)$ always increases from $g(n)$ by δ, and in the best case $h(s)$ will also increase by δ over $h(n)$. This means that in order to raise $f(s)$ to something higher than $g_{opt}(goal)$, we must apply a cost δ operator at least $\frac{\epsilon(n)}{2\cdot\delta}$ times to make it so that $f(s) > g_{opt}(goal)$.

The best case scenario is that the heuristic always rises by δ across each transition, along with g. If this happens, we can apply the δ cost operator $\frac{\epsilon(n)}{2\cdot\delta}$ times before $f(s) > g_{opt}(goal)$. If we assume there are no duplicate states, this produces a tree of size $b^{\epsilon(n)/2\cdot\delta}$ □

Limitations

The exact number of nodes that A* is going to have to expand as a result of the heuristic error is more difficult to calculate. First, it might not always be possible to apply an operator with cost δ. If this is the case, then the local minimum might not have $b^{\epsilon(n)/\delta}$ nodes in it, because δ will be replaced by the cost of the smallest operator that is applicable. If the domain has small operators that are not always applicable, then this equation will

overestimate the number of nodes that will be introduced to the open list.

The closed list can provide protection against expanding $b^{\epsilon(n)/\delta}$ nodes. If the local minimum does not have $b^{\epsilon(n)/2\cdot\delta}$ unique nodes in it, then the closed list will detect the duplicate states, allowing search to continue with a more promising area of the graph. This observation plays a critical role in determining whether cost-based search will perform well in domains with cycles.

Conclusion

The key insight here is that the high cost operators are capable of introducing much more error into the heuristic than any low cost operator, because the change in heuristic error is bounded by the cost of the operator. On the other hand, when A* goes to prove that no better path exists, we have to explore all paths that can be created using the low cost operators, which produces a tree exponential in the branching factor, of height proportional to the size of the error divided by the smallest cost operator. The assumption that the A* search algorithm will expand fewer nodes than Dijkstra's algorithm is based upon the assumption that it is possible to use the heuristic to prove that certain parts of the space do not need to be explored. We have shown that when there are a wide variety of operator costs, the effectiveness of this pruning can be dramatically reduced.

This analysis helps us to understand why finding a cost-optimal solution can be much more difficult when the domain had operators that vary widely in cost, but it also suggests an important tip for constructing heuristics in a domain where operator costs vary: make sure the heuristic can accurately track the effects of the high-cost operators, because these operators allow error to accumulate.

In addition to that, this knowledge helps us to manage expectations. The unfortunate reality is that finding provably optimal solutions is computationally intensive, and finding optimal solutions using a non-unit cost function can be even more computationally intensive, but this gives us some idea as to why this is so often the case. Fortunately, while provably optimal solutions may be difficult to find, this leaves open a very important alternative: solutions that may not be provably optimal, but are (ideally) substantially easier to find.

Domain	Heuristic	Cost	Max Local Min Size	Expected Min Size	Expansions
Hanoi (12 disk, 4 peg)	8/4 disjoint PDB	Unit	7,587	1,892.41	36,023
		Rev Sq	35,874	4,415.71	559,250
		Square	2,034	200.82	4,663
10/4 TopSpin	6 disk PDB	Unit	296	250.00	933
		Sum	922	2.65	749
		Stripe	240	2.64	441
3x4 Sliding Tile	Manhattan	Unit	392	2.01	801
		Inverse	51,532	87.23	93,010
11 Pancake	7 cake PDB	Unit	1	1	78
		Sum	8	8	149
Grid	Manhattan	Unit	2248	3.62	18,642
		Life	25,159	4.36	68,778
Robot	Path w/o dynamics	Unit	351	8.48	156
		Nonunit	341	7.27	153

Table 5-4: Sizes of local minima and average number of expansions required to find a solution.

Questioning the assumption that the solution has to be optimal is critical, because in many situation, what is needed is a good solution, not necessarily the best solution that there is.

5.1.2 Costs Make Satisficing Search Difficult

It is often the case that finding a cost optimal solution is too computationally expensive, exceeding limits on either time, memory, or both. When that happens, practitioners turn to one of a variety of algorithms that attempt to exchange provable optimality for improved runtime. One natural question to ask is if these satisficing algorithms are also plagued by the same problems as their optimal cousins.

In the rightmost column of Table 5-4, we can see the average number of expansions done by greedy best-first search using different heuristics. For each problem, we consider a standard unit-cost problem, as well as other cost functions.

On the sliding tile puzzle, we can see that with inverse costs, greedy best-first search requires substantially more work to find a solution when the operators do not all have the same cost.

For the pancake problem, we can again see that the non-unit cost problems are more

No. Cakes	Unit	Sum
11	78	148
12	416	776
13	2,571	6,382

Table 5-5: Expansions required by greedy best-first search to solve different pancake puzzles using a constant size (7 cake) PDB.

difficult to solve. Table 5-4 uses an 11 cake puzzle so it is possible to calculate the size of all of the local minima in the space, but the non-unit problems continue to be more difficult to solve as we increase the size of the space. This effect can be seen in Table 5-5 which shows the number of expansions required to solve both unit cost problems (left column) and non-unit cost problems (right column) with different numbers of disks.

For grid path planning, we can again observe the trend that unit-cost problems are easier to solve than non-unit cost problems, and once again, the unit-cost heuristic has smaller local minima as compared to the life-cost heuristic. Note that for this domain, we used a single instance to calculate these figures, because in this domain, the connectivity of the graph is not the same across all instances.

The same general trend can be observed in dynamic robot path planning, but it is worth noting that in this domain, the difference between the unit-cost heuristic and the standard heuristic is extremely small. The actual cost heuristic counts diagonal moves as $\sqrt{2}$, whereas the unit-cost heuristic counts diagonal moves as 1.

On the Towers of Hanoi problems, we can see that the unit cost problems are easier to solve than the reverse square cost problems, but more difficult to solve than the square cost problems. This example shows that while some problems are easier to solve when the operators all have the same cost, this trend is by no means absolute.

Examining the results of running greedy best-first search on a wide variety of benchmark domains with a wide variety of cost functions, we can see that the overall it appears that much of the time, unit-cost problems are easier for greedy best-first to solve, generally requiring fewer expansions. Unfortunately, this is not a hard and fast rule, because there

are examples of unit-cost problems that are more difficult to solve than non-unit problems. This leads to the question of why this occurs, and what is behind the variation in the amount of expansions that greedy best-first search requires to find solutions.

5.2 The importance of d

5.2.1 Hypotheses

We showed that non-unit problems can be substantially more difficult than a unit-cost problem, even when we have the same underlying connectivity and abstracting assumptions built into the heuristic. We also saw that this general trend covered both optimal and satisficing best-first heuristic search algorithms. We next review hypotheses that have been presented in the literature to explain this phenomenon.

Unit cost domains are easier

Cushing et al. (2010) and Cushing et al. (2011) argue that searching using a cost-based heuristic function, like h, is yields slower runtimes than searching using a distance-based heuristic function, like d. They go so far as to state "as a rule, cost-based search is harmful". While we have observed that this tends to be true, we have found exceptions to this as well.

The Towers of Hanoi with square costs are much easier for greedy best-first search to solve as compared to Towers of Hanoi problems with unit cost. The same applies to TopSpin problems; in this domain, the most difficult problems to solve are the unit-cost problems.

These counterexamples show that cost-based search is not always harmful. These counterexamples also serve as a place to start looking as to why cost-based search is sometimes less difficult.

The speedy heuristic better predicts search effort

One hypothesis that explains why greedy best-first search using h tends to be faster in unit-cost domains advanced by Thayer (2012) is that expanding nodes with small d (which is the same as h in a unit-cost problem) is the fastest way to get to a goal. Thayer writes, "$\hat{d}$ (an inadmissible estimate of d) is a proxy for search effort, and is used to ensure EES

(Explicit Estimation Search) pursues solutions that can be found quickly." A^*_ϵ uses d^*, the true distance to go, as a way to estimate, "the computational effort required to complete the search starting from n" (Pearl and Kim, 1982). If it is indeed possible to use d as a proxy for remaining search effort, then the d heuristic should be a better predictor of the remaining search effort as compared to the h heuristic. The reason that we expect d to predict search effort better than h is that if h is just as good at predicting remaining search effort as d, then there is no reason to bother using d for ascertaining remaining search effort, because h would, hypothetically, suffice.

If it is true that for nodes with a low d, the amount of work necessary to find a goal is low, and for high d nodes, the amount of work necessary to find a goal is high, the correlation between d value of the initial state and the number of nodes required to solve the problem should be strong. If the d heuristic is a better predictor of remaining search effort, this correlation should be stronger for d as compared to h.

To test this hypothesis, we considered three different ways to quantify the relationship between two variables: Kendall's τ (Kendall, 1938) and Spearman's ρ, two measures of rank correlation, as well as the standard linear correlation statistic, Pearson's r. We considered a number of standard benchmark domains with a variety of edge costs. For each domain, we considered a unit-cost variant, where the d and h are the same, and at least one non-unit cost variant. For each problem, we calculated the correlation between the heuristic of the initial state (h for the non-unit cost domains, and d or h for the unit-cost domains, where d and h are the same) and the number of nodes required to solve the problem using greedy best-first search.

The overall results can be seen in Table 5-6. The first section of the table shows the results for the Towers of Hanoi using the same cost functions and heuristics as before. With this problem, we can see that the h heuristic (using square costs) is a better predictor of remaining search effort as compared to the d heuristic (unit costs). h is not always better than d for predicting remaining search effort, as the h heuristic (using reverse square costs) is a very poor predictor of remaining search effort.

Domain (cost)	Heuristic	τ	ρ	r
Hanoi 12/4 (square)	h	0.51	0.61	0.69
Hanoi 12/4 (unit)	d	0.46	0.43	0.62
Hanoi 12/4 (reverse square)	h	0.08	0.05	0.14
3x4 Tiles (unit)	d	-0.03	-0.08	0.14
3x4 Tiles (inverse)	h	0.06	0.14	0.14
3x4 Tiles (reverse inverse)	h	0.01	0.06	0.14
TopSpin 10/4 (unit)	d	-0.00	0.00	0.00
TopSpin 10/4 (sum)	h	0.08	0.13	0.12
TopSpin 10/4 (stripe)	h	0.18	0.25	0.26
13 Pancake (unit)	d	-0.03	-0.04	-0.08
13 Pancake (nonunit)	h	0.11	0.16	0.15
Robot (unit)	d	0.95	0.99	0.99
Robot (nonunit)	h	0.96	1.00	0.99

Table 5-6: Correlation of heuristic with search effort in various domains using Kendall's τ, Spearman's ρ, and Pearson's r

For both the TopSpin and Pancake problems, we can see that there is very little relationship between either h or d and the number of expansions required to solve the problem using either greedy or speedy search.

We also considered three variants of the 11 tile sliding tile puzzle (3x4). The first variant is the standard unit cost function, where h and d are the same, so speedy search and greedy best-first search are the same. We also consider inverse costs, where the cost of moving tile n is $\frac{1}{n}$, and reverse inverse, where the cost of moving tile n is $\frac{1}{12-n}$. We can see that neither h nor d seems related to the number of expansions that greedy or speedy search will require to solve the problem.

The last domain we consider is dynamic robot path planning. In this domain, both h and d are very strongly correlated to the number of expansions required to solve the problem using greedy or speedy search, but it is worth noting that h is never worse than d at predicting remaining search effort.

To summarize, our examples do not provide support for the hypothesis that d better predicts remaining search effort than h.

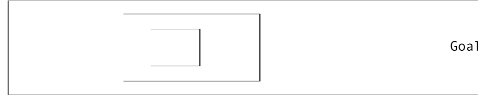

Figure 5-3: Picture of one or two of local minima, depending on how they are defined

d has smaller local minima

The last hypothesis we discuss argues that the d heuristic that one gets when all operators cost the same has smaller local minima than the h heuristic, and that the smaller local minima cause greedy best-first search and A* to run faster.

The concept of a local minimum in $h(n)$ is not rigorously defined, so the first step is to precisely define what a local minimum is. The first, and simplest definition, is to define a local minimum as a maximal set of connected nodes N such that for each node $n \in N$, every path from n to a node $n' \notin N$ includes at least one node n_{max} such that $h(n) < h(n_{max})$.

The "size" of a local minimum under this metric has little bearing on how effective greedy search is. For example, in the sliding tile domain with unit cost, an average local minimum contains 2.48 nodes, and in the sliding tile puzzle with inverse costs, where greedy search performs very poorly, an average local minimum contains 2.47 nodes. Thus, the size of a local minimum measured in this way cannot be used to predict whether optimal or satisficing best-first heuristic search is going to perform poorly. The reason is that local minima can be nested: once the search escapes the first local minimum, it is possible that it is inside another local minimum.

The solution, then, is to change the definition of a local minimum to verify that the path out of the first local minimum actually terminates in a goal node, as opposed to possibly leading into another local minimum. One modified definition of a local minimum is a maximal set of connected nodes N such that for each node $n \in N$, every path from n to a goal node includes at least one node n_{max} such that $h(n) < h(n_{max})$. The difference between this definition, and the previous one, is what happens to the small cup inside the large cup shown in Figure 5-3. Under the first definition, the small inner cup is its own

local minimum, but under the second definition, the small inner cup is not a local minimum on its own.

Using this definition of a local minimum, there is a clear benefit to having small local minima. Unless the initial state is located in a global minimum (a local minimum that contains a goal node), greedy best-first search will begin by expanding all of the nodes in the current local minimum, and will then proceed to look for the goal outside the local minimum. When local minima are large, the likelihood of starting in a local minimum increases, assuming start states are distributed uniformly at random. In addition to that, the penalty for encountering a large local minimum (expanding all of the large number of nodes in the local minimum) is high.

The preceding paragraphs lead us to believe that having small local minima is beneficial to best-first searches, leading us to the hypothesis that unit-cost problems are easier to solve because unit-cost heuristics have smaller local minima than non-unit cost heuristics.

If we limit analysis to a small search space, it is possible to calculate the size of every local minimum in an entire search space by searching backwards from the goal, expanding nodes in increasing h order, allowing us to directly test our hypothesis. Any node whose h value is less than the highest h value seen thus far is inside a local minimum, since nodes were expanded in h order The results of this analysis are shown in Table 5-4. Recall that if the initial state is inside a local minimum, greedy best-first search will expand every single node in the local minimum prior to exiting the local minimum and attempting to find a path to the goal.

Looking at Table 5-4, we can see that the hypothesis that the unit-cost problems have smaller local minima is not supported by the data, because there are a number of examples of unit-cost problems that have larger local minima than the non-unit cost problems. Specifically, in Towers of Hanoi, the square cost function has a heuristic with smaller local minima than the unit cost function, and for TopSpin, both the sum and the stripe cost functions have an expected local minima size that is substantially smaller than that of the unit cost problem, providing three counterexamples to our hypothesis.

Despite the fact that our hypothesis about d producing smaller local minima appears to be incorrect, the size of the local minima is able to provide useful information regarding how difficult the problems are to solve. In particular, if we consider how large the expected local minimum is and how large the largest local minimum is, we can make a very informed prediction about difficult the problem is going to be to solve for greedy best-first search. It is possible to calculate the size of every local minimum in a search space by searching backwards from the goal, expanding nodes in h order. Every node that has an h value that is less than that of the highest previously expanded h value is inside of a local minimum. Every time the h value is greater than or equal to the highest h value encountered thus far, the current local minimum is completely explored.

To summarize, the unit-cost heuristic doesn't always produce smaller local minima, but the data suggest that whichever heuristic does produce smaller local minima will be more useful for guiding a greedy best-first search. In most of the previous examples, the unit-cost heuristic had smaller local minima, but it is more accurate to say that a greedy best-first search has universal requirements on its heuristic, and whichever heuristic, h or d, better satisfies those requirements will produce a faster algorithm.

5.2.2 Why d is better than h

We now turn to the crucial question raised by these results: why d tends to produce smaller local minima as compared to h, proving itself to be a more useful heuristic.

Local Minima are More Likely using h

We begin this analysis by introducing a model of how heuristics are constructed which can be applied to any admissible heuristic. The model was originally created by Gaschnig (1979). We call this model the shortcut model of heuristic construction.

In any graph g, a node's h^* value is defined as the cost of a shortest path through the graph from the node to a goal node. In calculating the h value of the node, the shortcut model stipulates that the heuristic constructs a shortest path on a graph g' which is the same as the original graph, with the exception that additional edges have been added to

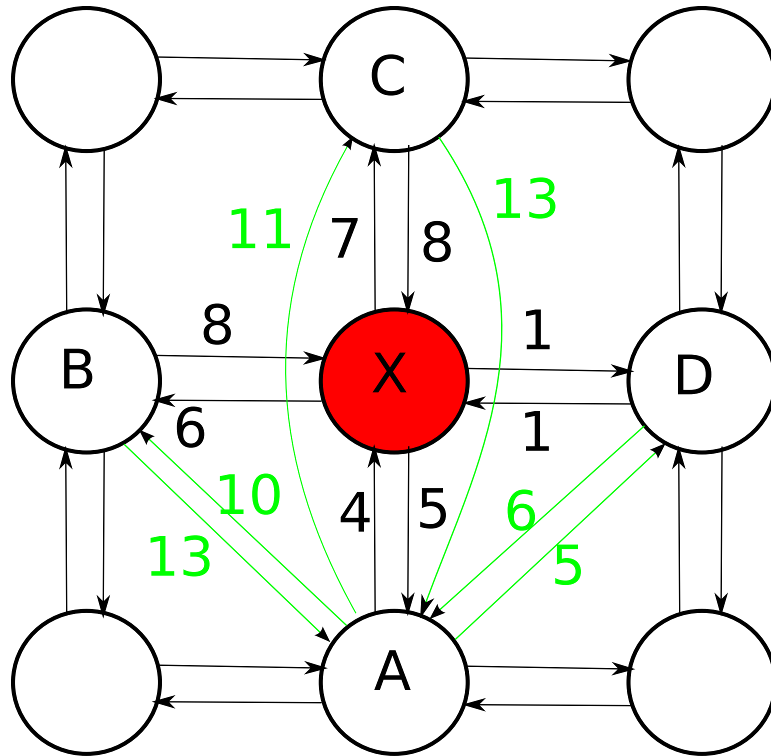

Figure 5-4: Example of how to remove extra nodes from a supergraph

the graph. Gaschnig (1979) calls g' the supergraph. The heuristic sometimes includes these edges from g' in its path, which is why it is not always possible to follow the heuristic directly to a goal. Any admissible heuristic can be modeled using a supergraph using the degenerate mapping of connecting every node n directly to the goal via an edge with cost $h(n)$. In the context of a pattern database, all nodes that map to the same abstract state are connected to one another by zero cost edges in the supergraph.

In some domains, it is easiest to conceptualize how the supergraph creates the heuristic by adding both nodes and edges. For example, in grid path planning, the Manhattan Distance heuristic adds nodes for all blocked cells, and edges connecting all of the blocked cells the way these cells would be connected if the cell was not blocked. This same general principle can be applied to the Manhattan Distance heuristic for the sliding tile puzzle where we add nodes that represent states where tiles share the same location.

If desired, we can then remove the additional nodes one at a time by replacing all length

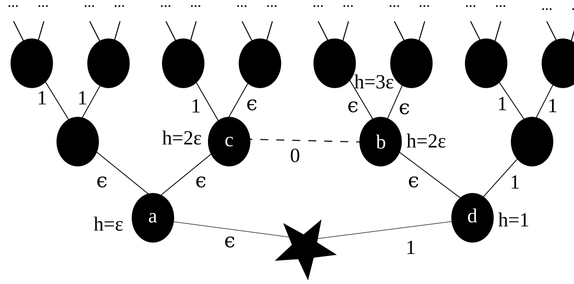

Figure 5-5: An example of a shortcut tree.

2 simple paths that go through the node being removed with single edges whose cost is the sum of the edges in the length 2 simple path. At this point, we can remove the node and all edges that go through the node in question, without changing the length of simple paths in the original graph. An example of this can be seen in Figure 5-4. In this example, we are in the process of removing node X. If node X is removed, we eliminate paths that go through node X. In order to allow the original graph to maintain the connectivity it would have had if node X had been present, we consider all simple paths of length 2 that go through node X, and replace each path with a new edge with the same start and end point as the simple path and the same cost as the simple path. In Figure 5-4 we show the edges that would get added to connect node A to nodes B, C, and D are shown in green. Note that to reduce clutter, the edges that would connect B, C, and D are not shown, but they are analogous to the edges involving node A.

An example tree is shown in Figure 5-5. Note that the heuristic value of a node is determined by either the heuristic value the node inherits from its lower neighbor node that is closer to the goal, or from a zero cost shortcut from a node with an even lower h. In the example of Figure 5-5, node c gets its h value of 2ϵ from its lower neighbor a and the ϵ cost edge between the two nodes. Node b in the tree gets its h value of 2ϵ via node c and the zero cost edge between the two nodes. Since we have a tree, any node that has an h value that is smaller than its lower neighbor in the original tree is inside of a local minimum.

Now, we will introduce a special kind of tree which we will use to model heuristic search trees, called a *shortcut tree*, an example of which is shown in Figure 5-5. A shortcut tree

has edge costs assigned uniformly at random from a categorical distribution *opset* such that the lowest cost edge costs ϵ and the highest cost edge costs 1. Each edge in the shortcut tree is assigned a weight independently from *opset*.

We require that the count of edges in all paths from a leaf to the goal be at least $\frac{1}{\epsilon}$. This means that all paths from a leaf to the root have a cost of at least 1. We model the heuristic of a shortcut tree as a supergraph heuristic that adds edges uniformly at random to the shortcut tree. With some fixed probability $0 \leq p \leq 1$ edges have zero cost, but if edges do not have zero cost, they are assigned a cost drawn from *opset*. Supergraph edges are also allowed to be the sum of more than one item from *opset*. A simple example can be seen in Figure 5-5, where *opset* only has two possible edge costs, ϵ, and 1. The star represents the goal, which is also the root.

In Figure 5-5, all paths from the node b to a goal go through node d, but node d has a heuristic value of 1, while node b has a heuristic value of 2ϵ, so node b is inside a local minimum, because going from b to a goal requires at least one node n with $h(n) > h(b)$. The local minimum was caused because node b is connected to node c via a zero cost edge. If node c had a heuristic value greater than 1, the zero cost edge between b and c would not cause a local minimum. Thus, the question of whether or not node b will be in a local minimum is equivalent to asking what the likelihood is that node b is connected to a node whose heuristic value is less than 1.

Shortcut trees have their edge weights and supergraph edges assigned randomly based upon *opset* and the probability that a supergraph edge is assigned zero cost. As a result, it is impossible to predict exactly what will happen with a particular shortcut tree. It is meaningful, however, to discuss the expected value over all possible assignments of edge weights and supergraph edges. Theorem 4 discusses how the expected probability of a local minimum forming changes as *opset* changes.

Theorem 4 *Let T be a shortcut tree with distribution opset of fixed height. Let us also assume that T is always tall enough to be a shortcut tree, even after opset has been changed. As the average value of the items in opset approaches 0, the expected value of the probability*

that a node whose parent's h value (parent is the neighbor closer to the goal) is at least 1 is inside a local minimum increases. As we increase the prevalence of operators whose cost is not 1, we also increase the expected value of the probability that a node whose parent's h value is at least 1 is inside a local minimum.

Proof: We need to figure out what the expected probability is that a node whose parent's h value is at least 1 is connected to a node whose h value is less than 1. By the definition of shortcut trees zero cost supergraph edges are added uniformly at random to the space, we simply need to figure out how the expected number of nodes whose h value is less than 1 changes.

For every node, the supergraph heuristic is constructed from a combination of regular edges and supergraph edges, allowing the possibility of using zero edges from either category. The expected contribution from all nonzero edges decreases, because the expected cost of a single edge has decreased because of the changes made to *opset*. Thus, the expected value of h for every node must decrease for all nodes whose h value is not zero. The expected number of nodes with $h = 0$ remains constant no matter how *opset* is changed because the number of $h = 0$ nodes depends only on how prevalent zero cost supergraph edges are.

The expected h value of a node is a function of the expected number of nodes with $h = 0$, and the expected h value for nodes that are not 0. Since the number of nodes with $h = 0$ remains constant, the number of nodes with $h \neq 0$ must also be constant, since the number of nodes in T is constant. Thus, the expected heuristic of all nodes decreases, because the expected heuristic of $h \neq 0$ decreases, while the proportion of $h \neq 0$ nodes remains constant. □

Every node in the tree needs to have an h value that is higher than its parent, otherwise the node will be inside of a local minimum. In particular, nodes whose parents have h values that are higher than 1 that receive h values that are smaller than 1 will be in a local minimum. Theorem 4 shows that two factors contribute to creating local minima in this way: a wide range of operator costs, and an overabundance of low cost operators. Both of these factors make sense. When the cheap edges are relatively less expensive, there are going

to be more nodes in the tree whose cost is smaller than 1. This increases the likelihood that a node that needs a high heuristic value is connected in the supergraph to a node with a low heuristic value because there are more nodes with low heuristic values. Likewise, when the prevalence of low cost edges increases, there are more parts of the tree with deceptively low heuristic values that look promising for a best-first search to explore.

Another natural question to ask is if increasing the prevalence of low cost operators will decrease the prevalence of nodes with high h^* to the point that Theorem 4 does not matter, because there are very few nodes with $h^* \geq 1$, therefore there are very few nodes with $h \geq 1$. Fortunately, this is not a problem, as long as the tree has reasonable height. In the worst case, there are 2 costs: 1 and ϵ, and the only way to get, h^* larger than 1 is to include at least one edge of cost 1. As the depth of the tree increases, the proportion of nodes with $h^* \geq 1$ increases exponentially. For example, if 90% of the edges have cost ϵ, at depth 10, only 34.9% of the nodes will be reached by only cost ϵ edges.

Theorem 4 offers an explanation of why guiding a best-first search with d is likely to be faster than guiding a best-first search with h to the extent that shortcut trees model the heuristic. With d, the heuristic pretends that all nodes have the same cost, which makes it so that the wide variety of edge costs cannot exacerbate the errors in the heuristic, as can happen when edge costs vary significantly (Cushing et al., 2010; Wilt and Ruml, 2011).

Theorem 4 also tells us that when doing best-first search, one possible source of inefficiency is the presence of many low cost edges, because these edges cause local minima. Low cost edges increase the probability that the h is computed from a supergraph path that completely bypasses a high h region, causing a local minimum, which best-first search on h will have to fill in.

One limitation of the analysis of Theorem 4 is that it considers only trees, while most problems are better represented by graphs. Fortunately, the analysis done in Theorem 4 is also relevant to graphs. The only difference between a graph and a tree are the duplicate nodes that are allowed to exist in a graph. These duplicate nodes can be modeled using zero cost edges between the nodes in the tree that represent the same node in the graph.

This makes it so that there are two kinds of zero cost edges: ones that were added because the problem is a graph, and zero cost edges from the supergraph. If we assume that the zero cost edges that convert the tree to a graph are also uniformly and randomly distributed throughout the space just like the zero cost edges from the supergraph, we arrive at precisely the same conclusion from Theorem 4.

Theorem 4 assumes the shortcut zero cost edges are uniformly distributed throughout the space, but edges may not be uniformly distributed throughout the space. If we do not know anything about a particular heuristic, applying Theorem 4, which discusses the expected properties of an arbitrary heuristic, may be the best we can do.

Local Minima can be costly for Greedy Best-First Search

In the previous section, we saw that local minima were more likely to form when the difference in size between the large and small operators increased dramatically. We also saw that as the low cost operators increased in prevalence local minima also became more likely to form. In this section we address the consequences of the local minima, and how those consequences are exacerbated by increasing the size difference between the large and small operators and the increased prevalence of low cost operators.

We begin the analysis for greedy best-first search with the same starting point as for A*, which is that the change in h between two adjacent nodes n_1 and n_2 is often bounded by the cost of the edge between n_1 and n_2.

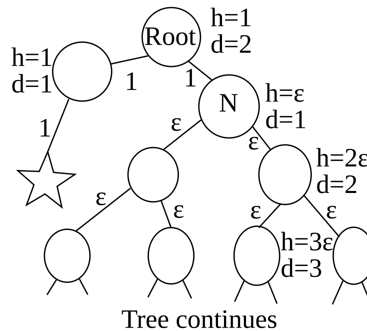

Figure 5-6: A search tree with a local minimum.

Consider the tree in Figure 5-6. In this tree, we have to expand all of the nodes whose heuristic value is less than 1, because the only goal in the space is a descendant of a node whose h value is 1. The core of the problem is the fact that node N was assigned a heuristic value that is way too low, an error made possible by the fact that the edge between N and the root costs 1. The tree rooted at N has height $1/\epsilon$, so it contains $2^{1/\epsilon}$ nodes, all of which would be expanded by a greedy best-first search. The tree in Figure 5-6 represents the best possible outcome for greedy best-first search, where the heuristic climbs as fast as it can. In a more antagonistic case, h could either fall or stay the same, which would exacerbate the problem, adding even more nodes to the local minimum.

If we substitute d for h, the ϵ edges change to cost 1, which makes it so the tree only contains 1 node. The number of nodes that can fit in a local minimum caused by a single error is much larger if the low cost edges in the graph have very low cost. The idea here is very similar to previously (Corollary 2 on page 92), except in this case, g is not contributing to escaping the local minimum, because greedy best-first search does not consider g when evaluating nodes. f could rise by 2ϵ per transition (ϵ from g, and ϵ from h), but h can only rise by ϵ across each transition.

5.2.3 Heuristic Gradients

In the previous section, we saw why the d heuristic tends to produce smaller local minima as compared to the h heuristic. The next question, then, is why this would be useful, and exactly why greedy best-first search and A* would benefit from having a heuristic with small local minima. A* uses the heuristic for pruning, and when doing A*, any time the heuristic fails to return h^*, A* is liable to expand additional nodes to prove that there does not exist a better solution under the node with the heuristic error. We now turn to a more formal analysis of how local minima connect to heuristic error.

High Water Mark Pruning

For every node n, there is a minimum h value, which we denote as h_{hw}, such that all paths from n to a goal include at least one node whose h value is at least h_{hw}. Note that if

there are no paths to a goal from n, this value should be infinity. Formally, this quantity is defined as

$$h_{hw}(n) = \min_{paths\ from\ n\ to\ a\ goal}(\max_{p \in path} h(p))$$

In order to find a path to a goal from node n, it is necessary to expand at least one node with an h value of $h_{hw}(n)$ and sufficient to expand all nodes x with $h(x) \leq h_{hw}(n)$.

On problems where there is a solution, greedy search takes advantage of this by never expanding any nodes whose h value is greater than $h_{hw}(root)$. Greedy best-first search terminates when it discovers a path from the start to the goal. Because of this, nodes on the open list whose h value is higher than the $h_{hw}(root)$ will never be expanded. As greedy search expands additional nodes, the minimum h_{hw} of all nodes on the open list decreases, decreasing the maximum h of nodes that will be expanded from that point onwards.

Theorem 5 *On problems for which greedy best-first search terminates, greedy best-first search will expand at least one node with $h(n) = h_{hw}(root)$. Greedy best-first search will not expand any nodes with $h > h_{hw}(root)$.*

Proof: All paths from the root to a goal node contain at least one node with $h \geq h_{hw}(root)$, per the definition of the high water mark of a node. This means that greedy best-first search must expand at least one such node. At least one such path must exist by hypothesis in order for greedy best-first search to find a solution. Prior to expanding any nodes with $h > h_{hw}(root)$, greedy search will expand at least one node with $h = h_{hw}(root)$. By the definition of $h_{hw}(root)$, it is possible to reach the goal starting at the root using only nodes with $h \leq h_{hw}(root)$. Since greedy best-first search terminates when it finds a solution, nodes whose h is higher than $h_{hw}(root)$ will not be expanded. □

The effectiveness of high water mark pruning is driven largely by the relationship between $h_{hw}(n)$ and $h(n)$. For example, suppose $\forall n : h(n) = h_{hw}(n)$. If this is the case, greedy search will be able to expand nodes along a single path leading directly to a goal, assuming optimal tie breaking.

If $h_{hw}(n)$ is significantly higher than $h(n)$ for many nodes in the space, the likelihood that

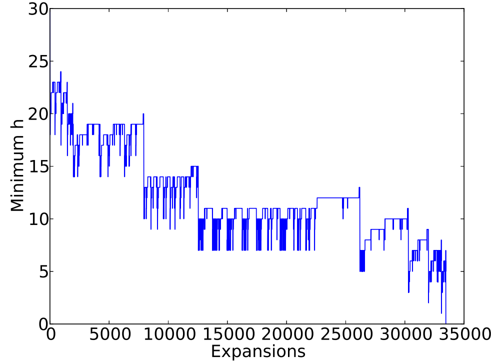

Figure 5-7: The minimum h value on open as the search progresses, using a disjoint PDB.

$\min_{n \in Open} h(n) \ll \min_{n \in Open} h_{hw}(n)$ increases. As $\min_{n \in Open} h_{hw}(n)$ approaches infinity, the number of nodes that will have $h < \min_{n \in Open} h_{hw}(n)$ will increase. This number is exactly the number of nodes that are inside of the local minima reachable from the nodes on the open list. As we recall from earlier, this quantity plays a major role in determining search time. Thus, it would be beneficial to assess this error in heuristics.

Heuristic Error

The right part of Figure 4-3, reproduced as Figure 5-7 shows the h value of the head of the open list of a greedy best-first search as the search progresses solving a Towers of Hanoi problem with 12 disks, 4 pegs, and a disjoint pattern database, with one part of the disjoint PDB containing 8 disks, and the other containing 4 disks. From this figure, we can see that the h value of the head of the open list of greedy search can fluctuate significantly. These fluctuations can be used to assess inaccuracies in the heuristic function. For example, at about 1,000 expansions the search encounters a node n_{bad} with a heuristic value that is 14, but we can show that the true h value of n_{bad} is at least 26.

After expanding n_{bad}, greedy search then expands a node with an h value of 20 at roughly 7,500 expansions. This allows us to establish that it costs at least 26 to get from n_{bad} to a goal because h is consistent. The general case is expressed as:

Theorem 6 *Consider a node n_{bad} that was expanded by greedy search, and n_{high}, the node with the highest h value that was expanded after n_{bad}. If $h(n_{high}) > h(n_{bad})$, then*

$h^*(n_{bad}) \geq h(n_{high})$ *if* h *is admissible.*

Proof: If there were a path from n_{bad} to a goal containing only nodes with $h < h(n_{high})$, greedy search would have expanded all nodes along this path prior to expanding n_{high}. Since all paths from n_{bad} to a goal contain at least one node with $h \geq h(n_{high})$, we know that n_{bad} is at least as far away from a goal as n_{high}, by the admissibility of h. □

The genesis of this problem is the fact that n_{bad} is in a local minimum. As discussed earlier, greedy best-first search will expand all nodes in a local minimum in which it expands one node, so clearly larger local minima pose a problem for greedy best-first search. Local minima also cause problems for A*, because every node in a local minimum has nonzero heuristic error.

The example in Theorem 4 is relevant here, as n_{bad} here is analogous to nodes like b in Figure 5-5. b is behind a node with a high heuristic value, but itself has a low heuristic value, which makes it at the bottom of a local minimum. As per Theorem 4, decreasing the size of the of the low cost edges or increasing the prevalence of the low cost edges can both increase the likelihood of nodes like b, which we have seen are problematic for both A* and greedy best-first search.

Heuristic error, defined as deviation from h^*, in and of itself, is not the root cause of the phenomenon visible in Figure 5-7. For example, $h(n) = h^*(n) \times 1000$ and $h(n) = h^*(n)/1000$ both have massive heuristic error, but either of these heuristics would be very effective for guiding a best-first search. The problem is the fact that to actually find a goal after expanding n_{bad}, all nodes with $h < h(n_{high})$, and the descendants of those nodes that meet the same criteria, must be cleared from the open list. It is the unproductive expansion of these nodes that causes greedy search to perform poorly.

From the perspective of greedy search, the core of the problem is the number of nodes that have a heuristic value lower than $h(n_{high})$ that are reachable from the root, independent of what $h^*(n)$ is. Bringing $h(n_{bad})$ closer to its true value could make it so that n_{bad} is not expanded, but there is another possibility: lowering $h(n_{high})$. For A*, the error in the heuristic really is the whole of the problem. Lowering the heuristic of $h(n_{high})$ will not fix

the problem for A*; the only solution is to make the heuristic aware of the fact that n_{bad} needs a high heuristic value.

Greedy search attempts to find a goal by following the heuristic directly to a goal, but this approach often fails, forcing greedy search to consider other nodes from the open list. If we view the search graph as a two dimensional topological space, we can conceptualize the greedy search algorithm as pouring water onto the root node where the water attempts to run downhill one node at a time, following the heuristic gradient (Cushing et al., 2011). If it is not possible for the water to run downhill, it will pool up, attempting to fill in a local minimum and eventually find a new place to continue flowing downhill. The only counter-intuitive aspect of this analogy is that a best-first search will simultaneously fill all minima that it has visited, as all their nodes share the same open list. With this metaphor in mind, we know that greedy search needs the heuristic to provide a gradient that is amenable to flooding by rolling downhill.

Heuristic Gradient Requirements

High water mark pruning and the nature of heuristic error are very closely related, as both describe things that can go wrong with the heuristic for greedy search. In both cases, the problem is that high h nodes should not separate regions of nodes with lower h.

In the context of high water pruning, it means that for every node, it is best if the node with the highest h value on the path to a goal is the first node on the path. In terms of heuristic error, nodes that have a high error according to Theorem 6 require traversing through a high h region to get to a goal. In either case, the low h nodes on the wrong side of the high h nodes are in a local minimum, which causes inefficiency in greedy best-first search.

Put in an alternative light, this means that if we eliminate all high h nodes from the state space, each separate region of the subgraph should contain at least one goal node. If this is not the case, greedy search is liable to enter one of the low h regions with no goal, and expand all of the nodes in that region prior to ever considering a different low h region to expand nodes in.

Theorem 4 discusses how likely a local minimum is to form, and shows that as we increase the prevalence of low cost edges or decrease the cost of the low cost edges, the likelihood of creating a local minimum increases. The local minima created have high water marks that are determined by the high cost edges in the graph. We then showed that if we have a local minimum whose height is the same as the high cost edge, the number of nodes that can fit inside of the local minimum can be exponential in the ratio of the high cost edge to the low cost edge, demonstrating that the performance penalty associated with even a single error in the heuristic is very severe, and grows exponentially as the low cost edges decrease in cost. Using the d heuristic instead of h mitigates these problems, because there are no high cost edges or low cost edges.

While it is generally true that doing greedy best-first using the d heuristic is faster than greedy best-first search using the h heuristic, note that some heuristics do not follow this general trend. For example, the h heuristic for the Towers of Hanoi using the reverse square cost function is faster than the d heuristic. The reason behind this trend is the fact that Theorem 4 only discusses the expected value across all possible heuristics that add the same number of zero cost edges to the graph. Which zero cost edges get added clearly has a major effect on how well a particularly heuristic will work.

5.2.4 A Solution: Search Distance

In the previous section, we saw why both A* and greedy best-first search encounter problems when there are local minima in the heuristic, and in Section 5.2.2 we saw that local minima are less likely using a unit-cost function. This observation suggests an obvious solution to the fact that non-unit cost problems can be so difficult to solve: pretend the problem has unit-cost to find a solution, and take advantage of the fact that a solution to the unit-cost problem can also be used as a solution to the non-unit cost problem. The obvious flaw with this approach is the fact that the path found by the unit-cost search may be extremely costly. Thayer et al. (2009) and Thayer and Ruml (2011b) investigate algorithms that exploit $d(n)$ to speed up search, but simultaneously attempt to maintain solution quality by enforcing a bound on solution quality.

We consider six algorithms, four of which make use of a $d(n)$ heuristic. The first algorithm we consider is greedy search, where nodes are expanded in $h(n)$ order. Second, we consider weighted A*, where nodes are expanded in $f'(n) = g(n) + w \cdot h(n)$ order. The third algorithm we consider is a best-first search that expands nodes with small d, disregarding cost completely. We call this method Speedy search, since the objective is to find a solution quickly by ignoring the costs. Since we elect to drop duplicate states in order to further speed things up, we call the specific variant used here Speedier (Thayer et al., 2009). Fourth, we consider a breadth-first beam search that orders nodes on $d(n)$. Fifth we consider Explicit Estimation Search (EES), which is a bounded suboptimal algorithm that uses inadmissible estimates of $d(n)$ and $h(n)$ to guide search, as well as an admissible $h(n)$ that is used to prove the quality bound. A detailed description of the algorithm is given by Thayer and Ruml (2011b). The last algorithm we consider is Skeptical search (Thayer and Ruml, 2011a), an algorithm that orders nodes on $\hat{f}(n)' = g(n) + w \cdot \hat{h}(n)$, where $\hat{h}(n)$ is an improved, but inadmissible, estimate of $h^*(n)$, as its evaluation function, using the admissible $h(n)$ to prove the bound on the initial solution.

We show that searching on $d(n)$ leads to very fast solutions, but there is a cost: as would be expected, solutions found disregarding all cost information are of very poor quality. Fortunately, the bounded suboptimal searches that leverage both distance and cost information are able to provide high quality solutions very quickly.

5.2.5 Empirical Results

Sliding Tiles

We consider the variant of the sliding tile puzzle where the cost of moving a tile is the face value of the tile raised to the third power, and use the same instances used by Korf (1985a). We use the Manhattan distance heuristic, which is weighted for the non-unit cost problems.

Raising the tile's number to the third power to get the cost of moving the tile dramatically increases the size of the largest operator relative to the smallest operator. Theorem 4 predicts this will increase the likelihood that local minima will form. Corollary 2 predicts with the increased operator cost ratio the local minima will contain more nodes. These

Domain	Algorithm	Expansions	Cost
Tiles	Greedy/Speedier	2,117	519
	WA* 3	13,007	74
	$d(n)$ beam (50)	4,207	90
	EES 3	6,816	85
	Skeptical 3	5,620	84
Pancake	Greedy/Speedier	26,174	7,532
	WA* 3	18,225	15
	$d(n)$ beam (50)	11,940	241
	EES 3	12,358	16
	Skeptical 3	15,640	15

Table 5-7: Algorithms on unit cost domains

Algorithm	Parameter	CPU Time	Solution Cost
Greedy		DNF	DNF
Weighted A*	1-1000	DNF	DNF
Speedier		0.010	485,033
$d(n)$ beam	10	0.228	199,709
$d(n)$ beam	50	0.086	77,223
$d(n)$ beam	100	0.098	65,187
$d(n)$ beam	500	0.315	57,122
EES	100	0.037	87,682
EES	10	0.038	87,682
EES	3	46.836	74,162
Skeptical	100	0.770	85,030
Skeptical	10	1.084	80,027
Skeptical	3	6.917	68,171

Table 5-8: Solving the 4x4 face^3 sliding tile puzzle

effects combine to lead us to predict that greedy best-first search and Weighted A* with a high weight will perform poorly.

Table 5-8 shows the results of running a variety of heuristic search algorithms on this problem. CPU time refers to the average CPU time needed to solve a problem. Solution cost refers to the average cost of the solution found by an algorithm. DNF denotes that the algorithm failed to find a solution for one or more instances due to running out of memory. With this cost function, speedier is the clear winner in terms of time to first solution, but it lags badly in terms of solution quality. EES and skeptical search, on the other hand, are

able to find high quality solutions. Interestingly, the most successful algorithm we were able to find for this domain is a breadth-first beam search that orders nodes on $d(n)$ (ordinary unweighted Manhattan distance) where ties are broken in favor of nodes with small $g(n)$. We believe this is due to the fact that the beam searches found short solutions in terms of path length, and that solutions short in terms of length also tend to have low cost. For example, in 31% of 3x4 sliding tile puzzle instances we sampled, following the unit-cost optimal solution returned by A* also resulted in an optimal weighted cost solution. In the worst case, the solution obtained by following the optimal unit-cost path was 32% higher than the of the cost of the optimal solution. The average cost of following the optimal unit-cost solution was 8.7%.

EES, Skeptical, speedier, and $d(n)$ beam search all solve the problem, but the most important fact to note from Table 5-8 is the fact that both weighted A* and greedy search were not able to solve the problem at all, showing that consideration of $d(n)$ is mandatory in this domain.

Sum Pancake Puzzle

For our evaluation on the pancake puzzle, we consider a 14 pancake problem where the pattern database contains information about seven pancakes, and the remaining seven pancakes are abstracted away. We used the same 100 randomly generated instances from the unit cost experiments. The 14 pancake problem was the largest problem we could consider because we were unable to easily generate a pattern database for a pancake problem that was any larger.

Once again, we have raised the operator cost ratio substantially, which we expect to cause more local minima to form, each containing more nodes, which we expect to cause problems for greedy best-first search.

The results of running the selected algorithms on this problem can be seen in Table 5-9. Once again, we observe weighted A* and greedy search are unable to solve all problems. EES and skeptical are able to solve all instances with larger weights. We had to use a weight of 4 because the weight of 3 used in the previous domain proved to be too aggressive. Speedier

Algorithm	Parameter	CPU Time	Solution Cost
Greedy		DNF	DNF
Weighted A*	1-1000	DNF	DNF
Speedier		0.679	508,131
$d(n)$ beam	10	36.666	100,507
$d(n)$ beam	50	1.810	8,191
$d(n)$ beam	100	1.642	5,184
$d(n)$ beam	500	1.777	1,905
EES	4	18.401	817
EES	10	1.619	934
EES	100	1.619	934
Skeptical	4	3.435	854
Skeptical	10	3.379	889
Skeptical	100	3.233	951

Table 5-9: Solving 14-pancake (cost = sum) problems

is able to solve all instances once again, but this comes at a cost: extremely poor quality solutions. Beam search on $d(n)$ is able to provide complete coverage over the instances, but neither solution quality or time to solution are particularly impressive in this domain. Overall, $d(n)$ based searches perform very well in the pancake puzzle with sum costs. Again, the most important thing to note about this domain is that, although there was no clear winner able to pareto dominate the all other algorithms, weighted A* and greedy search were once again unable to solve all problems.

Grid Path Planning

For grid path planning, we consider a variant of standard grid path planning where the cost to transition out of a cell is equal to the y coordinate of the cell, which has two benefits. First, there is a clear difference between the shortest solution and the cheapest solution. In addition, it allows us to simulate an environment in which a being in a certain part of the map is undesirable. The boards are 1,200 cells tall and 2,000 cells wide. 35% of the cells are blocked. Blocked cells are distributed uniformly throughout the space. In this domain, the size of a local minimum associated with any given node is bounded very tightly. For any node n in this problem, the descendants of n are all reached by applying an operator that is within 1 of the cost of the operator used to generate n. Given this, even if the heuristic

Algorithm	Parameter	Expansions	Solution Cost
Greedy		68,778	2,992,826
Weighted A*	3	148,407	2,556,569
Weighted A*	10	97,002	2,825,369
Weighted A*	100	71,704	2,904,727
Speedier		18,642	2,978,587
$d(n)$ beam	10-500	DNF	DNF
EES	3	113,413	2,721,338
EES	10	108,044	2,936,708
EES	100	108,044	2,936,708
Skeptical	3	165,146	2,514,215
Skeptical	10	127,079	2,603,531
Skeptical	100	117,056	2,630,606

Table 5-10: Grid Path Planning with Life Costs

errs in its evaluation of n, very few levels of nodes will have to be expanded to compensate for this heuristic error.

Another factor that makes grid world with life costs a particularly benign example of a non-unit cost domain is the fact that duplicates are so common. In order to observe the worst case exponential number of states described in Corollary 1, we assume the descendants of the node in question are all unique. In grid world with life costs, this assumption is not the case.

These mitigating factors allow cost-based best-first searches to perform very well, despite a very wide range of operator costs. This can be seen in Table 5-10 where weighted A* performs very well, in stark contrast to the other domains considered where weighted A* and greedy search are unable to solve all problems. It is a known problem that beam searches perform poorly in domains like grid path planning where there are lots of dead ends (Wilt et al., 2010), so it is not surprising that beam searches were unable to solve all instances in this domain. In Table 5-10, the rows for Speedier, EES, Skeptical, and Weighted A* are all pareto optimal, showing that in this domain, there is no clear benefit to using $d(n)$ the way there is in the weighted tiles domain and the heavy pancake domain, where the searches that did not consider $d(n)$ did not finish.

Summary

The examples in the previous section show the practical empirical consequences of Corollary 2 and Theorem 4. As the range of operator costs increases, local minima become more likely to form. Simultaneously, as the range of operator costs increases the number of nodes we expect to find in a local minimum also increases.

The sliding tile domain is one where low cost operators are often applicable and duplicates are rare, so there is little to mitigate the effect of the high cost ratio. In the sum pancake domain, duplicates are only mildly more common than in the sliding tiles domain, as the minimum cycle in that domain has length 6. Both of these domains prove to be very problematic for best-first searches when the costs are not all the same.

We can contrast this with what we observe in grid path planning. In grid path planning, duplicates are very common, and although operator costs vary across the space, there is very little local variation in operator costs, which places a very strict bound on the number of nodes that are within any single local minimum. Thus, the range of operator costs is only a part of what makes a non-unit cost domain more difficult.

5.3 Related Work

Other researchers have demonstrated that domains with high cost ratios can cause best-first search to perform poorly. The problem, in its most insidious form, was first identified by Benton et al. (2010). They discuss g value plateaus, a collection of nodes that are connected to one another, all with the same g value. Domains with g value plateaus have zero cost operators, so these domains have an infinite operator cost ratio. Such plateaus arise in temporal planning with concurrent actions and the goal of minimizing makespan, where extraneous concurrent actions can be inserted into a partially constructed plan without increasing its makespan, creating a group of connected nodes with the same g value. Benton et al. (2010) prove that when using either an optimal or a suboptimal search algorithm, large portions of g value plateaus have to be expanded, and they then demonstrate empirically that this phenomenon can be observed in modern planners. They propose a solution that

uses a node evaluation function considering estimated makespan to go, makespan thus far, and estimated total time consumed by all actions, independent of any parallelism. The estimate of the total time without parallelism is weighted and added to the estimated remaining makespan and incurred makespan to form the final node evaluation function, which is then tested empirically in a planner, which performs well on the planning domains they discuss. Benton et al. (2010) discuss the effects of zero cost operators, but do not discuss problems associated with small cost operators.

Cushing et al. (2010) argue that cost based search performs poorly when the ratio of the largest operator cost to the smallest operator cost is large, and that cost-based search can take prohibitively long to find solutions when this occurs. They first argue that it is straightforward to construct domains in which cost-based search performs extremely poorly simply by having a very low cost operator that is always applicable, but does not immediately lead to a goal. They empirically demonstrate this phenomenon in a travel domain where passengers have to be moved around by airplanes, but boarding is much less expensive than flying the airplane. They provide empirical evidence using planners to demonstrate that the presence of low cost operators causes traditional best-first search to perform poorly, and show that best-first searches that use plan size or hybridized size-cost metrics to rank nodes performs much better as compared to when using cost alone. Their analysis of how the heuristic factors into the analysis is limited to an empirical analysis of two planning systems. This work invited questions about the theoretical underpinnings of precisely why the range of operator costs was causing such a huge problem, and also questions about why the presence of an informed heuristic was not ameliorating the problem.

Cushing et al. (2011) extended their previous analysis, discussing examples of domains that exhibit the problematic operator cost ratios. Once again, their theoretical analysis is almost exclusively confined to uniform cost search, and does not consider the mitigating effects an informed heuristic can have, although they speculate that heuristics can help improve the bound, but not asymptotically. Their consideration of heuristics is purely empirical, limited to extensive analysis of the performance of two planning systems, discussing

how the various components of the planners deal with a wide variety of operator costs, and either exasperate or mitigate the underlying problem.

In this chapter, we extend this previous work in two ways. First, we illustrate the problem empirically in a number of standard benchmark domains from the search literature. Second we deepen the theoretical underpinnings of the problem associated with a wide range of operator costs. We do this by showing that, given a heuristic with a bound on its change between any parent and child node, the number of extra nodes introduced into the search by a heuristic error can, in the worst case, be exponential, but that this can be mitigated by change in the branching factor (e.g. branching factor is not uniform) or the detection of duplicates; the heuristic cannot rise sufficiently quickly to change the underlying asymptotic complexity. We also show that the property of bounded change of heuristic error arises in a number of common robotics heuristics, and it is known that any domain with a consistent heuristic and invertible operators will also satisfy this requirement.

A number of algorithms make use of a distance-based heuristic. For example, Explicit Estimation Search (Thayer and Ruml, 2011b) uses a distance-based heuristic to try and find a goal quickly. Deadline Aware Search (Dionne et al., 2011) is an algorithm that uses distance estimates to help find a solution within a specified deadline. The LAMA 2011 planner (Richter et al., 2011), winner of the 2011 International Planning Competition, uses a distance-based heuristic to form its first plan.

5.4 Conclusion

It is well known that searching on distance can be faster than searching on cost, but the precise mechanics of when and why this occurs were poorly understood. We have shown the hypotheses presented in the academic literature to explain this phenomenon to be incorrect. We have suggested that one reason behind the conventional wisdom is the fact that the d heuristics that measure distance tend to produce smaller local minima in popular benchmark domains.

We have shown that when the h heuristic has smaller local minima, best-first search on

h will outperform greedy best-first search on d. This is because best-first search assumes that it will not get bogged down in the local minima in the heuristic. We show that when local minima in the heuristic are small, the expected number of expansions done by a greedy best-first search is low, and when the local minima in the heuristic are large, the expected number of expansions done by a greedy best-first search is high.

We suggested that this phenomenon is due to the fact that the expected size of a local minimum is smaller for the d heuristic as compared to the h heuristic, which tells us that we should expect that algorithms that use d will tend to be faster than algorithms that use h. We also propose a model that can be used to explain why this phenomenon occurs. This model shows why best-first search on d tends to be faster than best-first search on h.

Lastly, we proposed a solution for problems where cost-based search performs poorly, which is to consider an additional heuristic, $d(n)$, and use this heuristic to help guide the search. We then showed that when best-first search proves impractical, searches that consider $d(n)$ can still find solutions. This demonstrates that algorithms that exploit $d(n)$ represent a fruitful direction for research. These results are significant because real-world domains often exhibit a wide range of operator costs, unlike classic heuristic search benchmarks.

CHAPTER 6

Conclusion

Although there are a wide variety of heuristic search algorithms, there is little theoretical understanding about why the various algorithms work well, and under what circumstances these algorithms will perform poorly. Even well understood algorithms like A* have situations in which they perform arbitrarily poorly (e.g. Martelli graphs) (Martelli, 1977). Every useful algorithm has a collection of assumptions that are made about the nature of the domain that are used to speed the search up.

We began by applying this idea to bidirectional heuristic search. Bidirectional search has long been believed to hold significant potential to speed up heuristic search, but researchers have never been able to come up with a reliable, robust way to capture this potential. We examined one way of leveraging bidirectional heuristic search, which was using the backwards search to correct the forwards heuristic, but we saw that once again, as is often the case with bidirectional search, it does not always help. In addition to that, when the heuristic can be corrected, the technique assumes that precisely the right number of nodes are going to be expanded going backwards, otherwise there is no benefit (too few nodes) or insufficient benefit (too many nodes). By acknowledging these assumptions, we are able to modify the original technique to directly compensate for these weaknesses, providing us with a robust bidirectional heuristic search that is able to automatically configure itself for maximal benefit when bidirectional search is helpful, and automatically configure itself for minimal performance loss when bidirectional search proves to be unhelpful.

In Chapter Three, we saw that within a best-first heuristic search, different ways of implementing the open list each have different assumptions about the domain. For example, some implementations assume that all operators in the domain have integer costs, and that the range of those costs is small. Other implementations assume that the vast majority of nodes that are put on the open list are never removed from the open list. In this chapter,

we saw that the performance of the different implementations varied significantly, and that it was possible to extract performance gains out of a best-first search by making a change as simple as selecting a new data structure to use for the open list.

In Chapter Four, we address the question of what a greedy best-first search needs in a heuristic. The objective of this analysis is twofold. First, if we know that a domain has a heuristic that is not going to work well with greedy best-first search, that tells us that we should consider other heuristic search algorithms than greedy best-first search. Second, this knowledge about what greedy best-first search requires out of a heuristic can help us to identify and build better heuristics for greedy best-first search, which in turn leads to better greedy best-first searches.

In Chapter Five, we discuss how problems where edges all have the same cost tend to be substantially easier to solve as compared to problems where edge costs vary, and that this trend repeats itself in many domains, in both optimal and suboptimal searches. Within the context of suboptimal search, we performed a comparison between two different kinds of heuristics, h heuristics that measure cost, and d heuristics that measure distance. Empirically, we saw that it is usually the case that using a d heuristic to guide a greedy best-first search (speedy search) results in faster search as compared to guiding the greedy best-first search with an h heuristic. We then show that the reason for this is that the d heuristic tends to better satisfy the assumptions of greedy best-first search, which is why speedy search tends to outperform greedy best-first search guided by h.

One interesting alternative to the approach taken in this dissertation is selecting algorithms by building a portfolio of heuristic search algorithms, and taking advantage of the fact that modern hardware is becoming more and more amenable to parallel processing. While it may seem like this is a radically different approach, in reality it is quite similar. In any portfolio algorithm, there is a collection of different heuristic search algorithms, but unless the collection contains every algorithm ever invented, decisions still have to be made about which algorithms should be included in the portfolio. In addition to that, we ideally want the portfolio to be built in such a way as to maximize the likelihood that one of the

algorithms in the portfolio is going to work well, and that objective can only be achieved through having a very good understanding of when the different algorithms work well.

Another approach related to using a portfolio algorithm is using a machine learner to figure out what algorithm should be used. The critical problem with this approach is that machine learning algorithms require a set of features that will allow the machine learner to select the correct algorithm for the problem. The question then becomes how to identify the features. In this case, our analysis about what different heuristic search algorithms need to perform well are precisely the features required by the machine learner.

This dissertation seeks to develop a theoretical understanding of some of the most popular heuristic search algorithms, in the hope that the knowledge about when different algorithms are appropriate leads to more accurate algorithm selection. The ultimate goal is to allow practitioners to avoid the labor intensive trial and error process that currently dominates.

BIBLIOGRAPHY

Sandip Aine, P.P. Chakrabarti, and Rajeev Kumal. AWA* - a window constrained anytime heuristic search algorithm. In *Proceedings of IJCAI-07*, 2007.

J. Benton, Kartik Talamadupula, Patrik Eyerich, Roburt Mattmueller, and Subbaro Kambhampati. G-value plateuas: A challenge for planning. In *Proceedings of the Twentieth International Conference on Plannin and Scheduling*, 2010.

Yngvi Björnsson, Markus Enzenberger, Robert C. Holte, and Jonathan Schaeffer. Fringe search: beating A* at pathfinding on game maps. In *In Proceedings of IEEE Symposium on Computational Intelligence and Games*, pages 125–132, 2005.

Ethan Andrew Burns, Matthew Hatem, Michael J. Leighton, and Wheeler Ruml. Implementing fast heuristic search code. In *SOCS*, 2012.

Stephen V. Chenoweth and Henry W. Davis. High-performance A* search using rapidly growing heuristics. In *Proceedings of the Twelfth International Joint Conference on Articial Intelligence*, pages 198–203, 1991.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. ISBN 978-0-262-03384-8.

William Cushing, J. Benton, and Subbarao Kambhampati. Cost based search considered harmful. In *Proceedings of the Third Symposium on Combinatorial Search*, 2010.

William Cushing, J. Benton, and Subbarao Kambhampati. Cost based search considered harmful. http://arxiv.org/abs/1103.3687, June 2011.

Robert B. Dial. Algorithm 360: shortest-path forest with topological ordering [h]. *Communications of the ACM*, 12(11):632–633, 1969.

John F. Dillenburg and Peter C. Nelson. Perimeter search. *Artificial Intelligence*, 65(1): 165–178, 1994.

Austin Dionne, Jordan T. Thayer, and Wheeler Ruml. Deadline-aware search using on-line measures of behavior. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search*, 2011.

J. E. Doran and D. Michie. Experiments with the graph traverser program. In *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, pages 235–259, 1966.

Stefan Edelkamp and Stefan Schrödl. *Heuristic Search - Theory and Applications*. Academic Press, 2012. ISBN 978-0-12-372512-7.

Stevan Edelkamp, Amr Elmasry, and Jyrki Katajainen. The weak-heap family of priority queues in theory and praxis. In *Computing: The Australasian Theory Symposium (CATS)*, 2012.

Ariel Felner, Carsten Moldenhauer, Nathan R. Sturtevant, and Jonathan Schaeffer. Single-frontier bidirectional search. In *AAAI*, 2010.

Ariel Felner, Uzi Zahavi, Robert Holte, Jonathan Schaeffer, Nathan R. Sturtevant, and Zhifu Zhang. Inconsistent heuristics in theory and practice. *Artif. Intell.*, 175(9-10): 1570–1603, 2011.

Ariel Felner, Meir Goldenberg, Guni Sharon, Roni Stern, Tal Beja, Nathan R. Sturtevant, Jonathan Schaeffer, and Robert Holte. Partial-expansion a* with selective node generation. In *AAAI*, 2012.

David Furcy and Sven Koenig. Scaling up WA* with commitment and diversity. In *International Joint Conference on Artificial Intelligence*, 2005.

John Gaschnig. Exactly how good are heuristics?: Toward a realistic predictive theory of

best-first search. In *Proceedings of the Fifth International Joint Conference on Articial Intelligence*, pages 434–441, 1977.

John Gaschnig. A problem similarity approach to devising heuristics: First results. In *Proceedings of the Sixth International Joint Conference on Articial Intelligence (IJCAI-79)*, 1979.

Jean Dickinson Gibbons. *Nonparametric Statistical Inference*. Marcel Decker, Inc, 1985.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, July 1968.

Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Malte Helmert. Landmark heuristics for the pancake problem. In *Proceedings of the Third Symposium on Combinatorial Search*, 2010.

Malte Helmert and Gabriele Röger. How good is almost perfect? In *Proceedings of the ICAPS-2007 Workshop on Heuristics for Domain-independent Planning: Progress, Ideas, Limitations, Challenges*, 2007.

Jörg Hoffmann. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *Journal of Artifial Intelligence Research*, 24:685–758, 2005.

Jörg Hoffmann. Analyzing search topology without running any search: On the connection between causal graphs and h^+. *Journal of Artificial Intelligence Research*, 41:155–229, 2011.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

R. Holte, J. Grajkowskic, and B. Tanner. Hierachical heuristic search revisitied. In *Symposium on Abstracton Reformulation and Approximation*, pages 121–133, 2005.

Robert C. Holte. Common misconceptions concerning heuristic search. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, pages 46–51, 2010.

Toshihide Ibaraki. Depth-m search in branch-and-bound algorithms. *International Journal of Parallel Programming*, 7(4):315–343, 1978.

Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

R.E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134: 9–22, 2002.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985a.

Richard E. Korf. Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1034–1036, 1985b.

Richard E Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1): 65–88, 1987.

Richard E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41–78, 1993.

Richard E. Korf. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 700–705, 1997. ISBN 0-262-51095-2.

Richard E. Korf. Analyzing the performance of pattern database heuristics. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 2*, AAAI'07, pages 1164–1170, 2007. ISBN 978-1-57735-323-2.

Richard E. Korf and Larry A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *AAAI/IAAI, Vol. 2*, pages 1202–1207, 1996.

Richard E. Korf, Michael Reid, and Stefan Edelkamp. Time complexity of iterative-deepening-A*. *Artificial Intelligence*, 129:199–218, 2001.

Levi Lelis, Sandra Zilles, and Robert C. Holte. Improved prediction of IDA*'s performance via epsilon-truncation. In *SOCS*, 2011.

Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of the Seventeenth Annual Conference on Neural Information Processing Systems*, 2003.

Marco Lippi, Marco Ernandes, and Ariel Felner. Efficient single frontier bidirectional search. In *Proceedings of the Fifth Symposium on Combinatorial Search*, 2012.

Alberto Martelli. On the complexity of admissible search algorithms. *Artificial Intelligence*, 8(1):1–13, 1977.

Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co, 1980.

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

Judea Pearl and Jin H. Kim. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(4):391–399, July 1982.

Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1: 193–204, 1970.

Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artifial Intelligence Research*, 39:127–177, 2010.

Silvia Richter, Jordan T. Thayer, and Wheeler Ruml. The joy of forgetting: Faster anytime search via restarting. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, 2009.

Silvia Richter, Matthias Westphal, and Malte Helmert. LAMA 2008 and 2011. In *International Planning Competition 2011 Deterministic Track*, pages 117–124, 2011.

Wheeler Ruml, Minh Binh Do, Rong Zhou, and Markus P. J. Fromherz. On-line planning and scheduling: An application to controlling modular printers. *JAIR*, 40:415–468, 2011.

Stuart Russell. Efficient memory-bounded search methods. In *Proceedings of the 10th European conference on Artificial intelligence*, ECAI '92, pages 1–5, 1992. ISBN 0-471-93608-1.

U.K Sarkar, P.P. Chakrabarti, S. Ghose, and S.C. De Sarkar. Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence*, 50:207–221, 1991.

Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011. ISBN 978-0-321-57351-3.

G. J. Sussman. *A Computer Model of Skill Acquisition*. New York: New American Elsevier, 1975.

Jordan T. Thayer and Wheeler Ruml. Learning inadmissible heuristics during search. In *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling*, 2011a.

Jordan T. Thayer, Wheeler Ruml, and Jeff Kreis. Using distance estimates in heuristic search: A re-evaluation. In *Proceedings of the Second Symposium on Combinatorial Search*, 2009.

Jordan Tyler Thayer. *Heuristic Search Under Time and Quality Bounds*. PhD thesis, University of New Hampshire, May 2012.

Jordan Tyler Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proceedings of the Twenty Sixth International Joint Conference on Articial Intelligence (IJCAI-11)*, pages 674–679, 2011b.

Jordan Tyler Thayer, Roni Stern, and Levi H. S. Lelis. Are we there yet? - estimating search progress. In *SOCS*, 2012.

John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, Reading, MA, 1977.

Jur van den Berg, Rajat Shah, Arthur Huang, and Kenneth Y. Goldberg. Anytime non-parametric A*. In *Proceedings of the Twenty Fifth National Conference on Articial Intelligence*, 2011.

Christopher Wilt and Wheeler Ruml. Cost-based heuristic search is sensitive to the ratio of operator costs. In *Proceedings of the Fourth Symposium on Combinatorial Search*, July 2011.

Christopher Wilt and Wheeler Ruml. When does weighted A* fail? In *Proceedings of the Fifth Symposium on Combinatorial Search*, July 2012.

Christopher Wilt and Wheeler Ruml. Robust bidirectional search via heuristic improvement. In *AAAI*, 2013.

Christopher Wilt, Jordan Thayer, and Wheeler Ruml. A comparison of greedy search algorithms. In *Proceedings of the Third Symposium on Combinatorial Search*, July 2010.

Yuehua Xu, Alan Fern, and Sungwook Yoon. Learning linear ranking functions for beam search with application to planning. *The Journal of Machine Learning Research*, 10: 1571–1610, 2009.

Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. A* with partial expansion for large branching factor problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 923–929, 2000. ISBN 0-262-51112-6.

Rong Zhou and Eric A. Hansen. Beam-stack search: Integrating backtracking with beam search. In *Proceedings of ICAPS-05*, 2005.