Heuristic search under time and cost bounds

Jordan Tyler Thayer
University of New Hampshire, Durham

HEURISTIC SEARCH UNDER TIME AND COST BOUNDS

BY

Jordan Tyler Thayer

BS of Computer Science, Rose-Hulman Institute of Technology, 2006

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, 2012

This dissertation has been examined and approved.

[illegible]

Dissertation director. Wheeler Ruml.
Assistant Professor of Computer Science
University of New Hampshire

[illegible]

Radim Bartôs.
Associate Professor. Chair of Computer Science
University of New Hampshire

[illegible]

Michel Charpentier.
Associate Professor of Computer Science
University of New Hampshire

[illegible]

Phillip J. Hatcher.
Professor of Computer Science
University of New Hampshire

[illegible]

Sven Koenig.
Professor of Computer Science
University of Southern California

May 7th, 2012
Date

DEDICATION

For my father, who taught me to set the bar just out of reach, my mother, who taught me there is more to life than work, and my wife who makes sure I remember both lessons.

ACKNOWLEDGMENTS

A dissertation is a huge undertaking and to say that all of the work here is mine and mine alone would be nonsense. While the work is mine, it would not have been possible without the support, guidance and patience of many people who I'd now like to thank.

Peer review is a larger and more important piece of the scientific process than I realized six years ago. It makes sure that the content of what we publish is correct, that the writing makes sense, and that is worth reading. I've had anonymous reviewers correct me on all three accounts over the years, and the contents of this thesis are much improved for it.

Obviously, much of this work wouldn't have been possible without the support and guidance of my advisor, Wheeler. Without his encouragement and persistence I would have likely given up somewhere along the way.

My lab mates, particularly Ethan, made the long hours of graduate school go by just a bit faster. They've also commented on my work, often to its betterment, for which I am grateful.

Finally, my wife Rachel, who was either too understanding or too timid to divorce me for neglecting her in favor of my research. Surely I would have subcomed to scurvy or exhaustion without her constant insistence that I eat something other than instant noodles and her firm belief that coffee cannot be substituted for sleep.

The work contained in this thesis has been funded primarily by the NSF and DARPA. I gratefully acknowledge their financial support over the years. I would particularly like to thank the NSF for the support it has given me to attend various doctoral consortia, as this has resulted in substantial improvements to this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

HEURISTIC SEARCH UNDER TIME AND COST BOUNDS

by

Jordan Tyler Thayer

University of New Hampshire, May, 2012

Intelligence is difficult to formally define, but one of its hallmarks is the ability find a solution to a novel problem. Therefor it makes good sense that heuristic search is a foundational topic in artificial intelligence. In this context "search" refers to the process of finding a solution to the problem by considering a large, possibly infinite, set of potential plans of action. "Heuristic" refers to a rule of thumb or a guiding, if not always accurate, principle. Heuristic search describes a family of techniques which consider members of the set of potential plans of action in turn, as determined by the heuristic, until a suitable solution to the problem is discovered.

This work is concerned primarily with suboptimal heuristic search algorithms. These algorithms are not inherently flawed, but they are suboptimal in the sense that the plans that they return may be more expensive than a least cost, or optimal, plan for the problem. While suboptimal heuristic search algorithms may not return least cost solutions to the problem, they are often far faster than their optimal counterparts, making them more attractive for many applications.

The thesis of this dissertation is that the performance of suboptimal search algorithms can be improved by taking advantage of information that, while widely available, has been overlooked. In particular, we will see how estimates of the length of a plan, estimates of plan cost that do not err on the side of caution, and measurements of the accuracy of our estimators can be used to improve the performance of suboptimal heuristic search algorithms.

CHAPTER 1

INTRODUCTION TO HEURISTIC SEARCH

The focus of this dissertation is heuristic search, so we begin with a description of the technique aided by a simple example. Then we discuss some of the most basic techniques for systematic heuristic search before providing an outline of the dissertation and its contributions.

1.1 A Simple Example

Search is a technique used to automatically find solutions to a wide variety of problems ranging from finding high quality alignments for sequences [79] of DNA to automatically driving an automobile [36]. The following example is most like finding a path for a character in a video game, another popular use of heuristic search algorithms [7].

Although the kinds of problems that can be solved by heuristic search are very different from one another, they do share several important features. Typically, the problems can be described by some initial configuration, or state, some goal state, or set of goal states, and a set of actions which can convert one state into another. Heuristic search algorithms then systematically consider plans until they find one which converts the initial state into the goal state.

Figure 1-1 shows a simple problem that can be solved with heuristic search: pathfinding in a four-neighbor grid. We have an agent in the starting state, the lower left hand corner of the grid labeled "Start", and they would like to be in the upper right hand cell of the grid labeled "Goal".

Figure 1-1 is filling several roles. It shows us an entire problem, it shows us a possible configuration of the world, the initial configuration or starting state, and it shows an in-

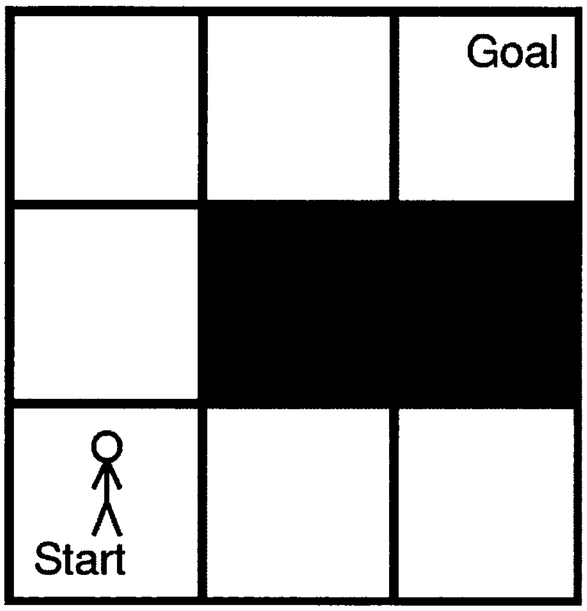


Figure 1-1: Starting state

complete solution to the problem. In particular, it shows the partial solution in which we take no action. We will be referring to a partial solution under consideration by a search algorithm as a *node.*

It is important to differentiate between a *state* and a *node.* A state is simply a configuration of the world, for example the stick figure in the lower left hand cell. A node is both a configuration of the world and the path by which it was reached. This will become important later on, especially in Chapter 7, when we discuss how to handle the situation in which a search algorithm reaches the same state by multiple paths, resulting in two nodes representing the same configuration of the world.

Figure 1-2 shows the actions available to us in the initial state shown in Figure 1-1. From this state, we might move to the north, or we might move to the east. Taking either of these actions would result in a new state, one in which the agent was one cell north of the start and one where the agent was one cell east respectively.

When a search algorithm selects a node and considers adding actions to the partial solution it represents, we say that the search has *expanded* that node. A single expansion is

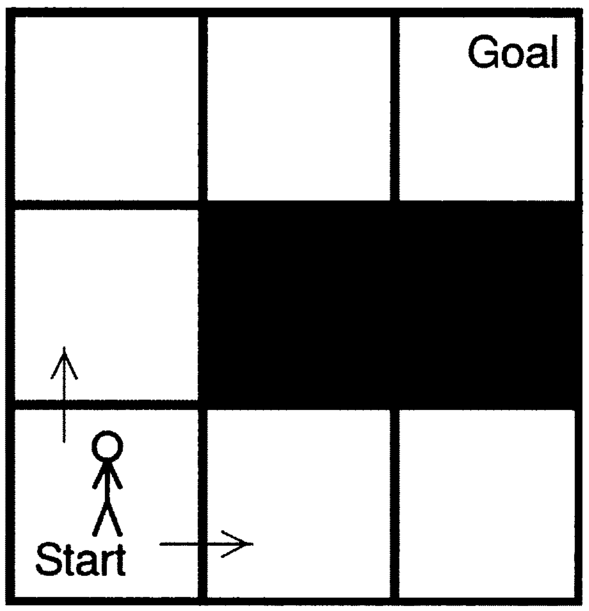


Figure 1-2: Actions

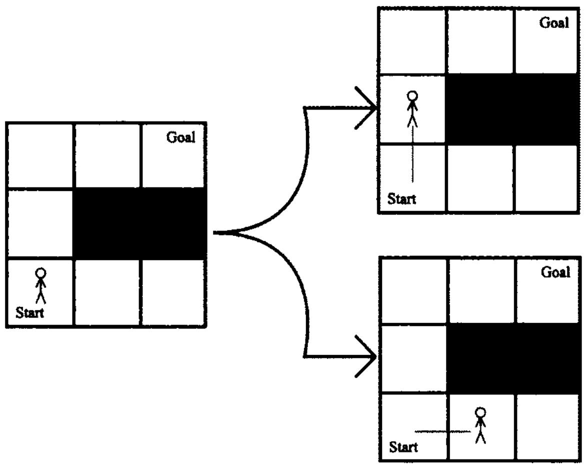


Figure 1-3: Search tree

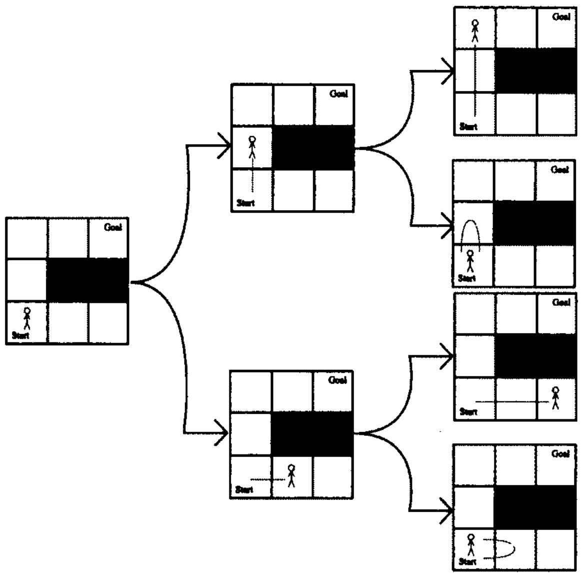


Figure 1-4: Search tree

shown in Figure 1-3. We see the initial search node, often called the root, generating two successor, or child, nodes. The search state is shown, and the path by which the state was achieved is drawn in red. As we've shown here, an expansion considers applying all legal actions to the end of the current plan represented by the parent node. There are techniques which consider only applying a subset of the available actions [78] and those which consider inserting actions at places other than the end of a partial solution [38], however in this dissertation we will focus on algorithms that consider adding all legal actions to the end of a partial plan.

Figure 1-4 shows another pair of expansions on the nodes generated by the expansion of the root, shown in Figure 1-3. It shows a couple of interesting features of heuristic search. We should first note that two of the grand-children of the root are plans which take us from the starting state back to the starting state. Paying attention to such duplicate states in search is important for performance.

We can also see that the search basically creates a tree of possible plans for solving the problem in question. We can think about a heuristic search algorithm as inducing a tree over the problem, where each edge of the tree represents an action and each element of the

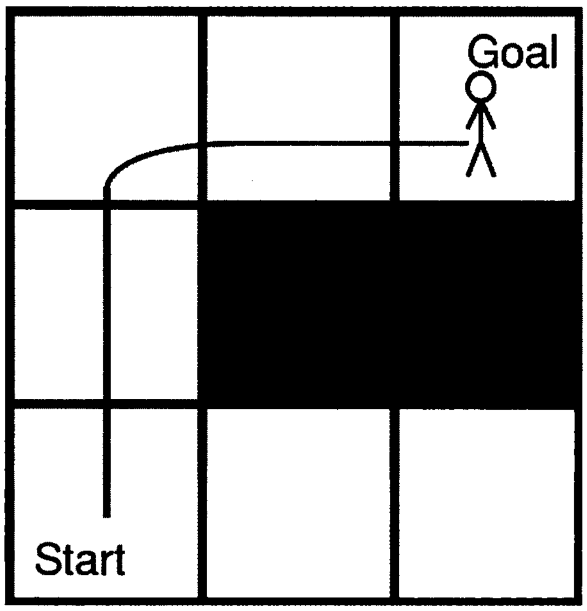


Figure 1-5: Solved problem

tree is a state of the problem. A path from the root of the tree to a goal state represents a solution to the problem, shown in Figure 1-5.

1.2 Basic Search Strategies

It is important to note that trees like the one shown in Figure 1-4 get very large very quickly. The study of heuristic search is in part concerned with techniques for efficiently constructing and navigating these large trees. We now discuss three of the most basic methods for heuristic search. The first two are optimal search algorithms. They are optimal in the sense that they return cost-optimal solutions to the problem should one exist. The last algorithm we will discuss in this section is a suboptimal search algorithm which provides no guarantee on solution cost relative to optimal.

1.2.1 Uniform Cost Search

Uniform cost search is one of the simplest techniques for finding optimal cost solutions to problems. It works by systematically considering potential solutions in order of increasing

```
**UniformCostSearch(root)**
1.  $open \leftarrow \{root\}$
2.  while $open \neq \{\}$
3.    let $n = \text{argmin}_{n \in open}\, g(n)$ in
4.      if $goal_p(n)$
5.      then return $n$
6.      else $open \leftarrow open - \{n\}$
7.          for each child $c$ of $n$, $open \leftarrow open \cup \{c\}$
8.  return no solution
```

Figure 1-6: Uniform cost search pseudo code

cost until a goal is found. Pseudo code for the algorithm is presented in Figure 1-6.

On line 1, we initialize the open list of uniform cost search to contain the initial node, the root. The open list is simply a collection of all nodes being considered by the search currently. As we see in line 2 , search proceeds so long as there are plans that have yet to be considered. Sometimes a node may have no children, that is there are no legal actions to append to the partial plan. Other times, a node may only generate children with states that the search has already encountered by a better path, in which case those children are discarded[1]. If no nodes are left for consideration, the search algorithm has exhausted the space, showing that there is no solution to the problem. The ability of a search algorithm to always find a solution to a problem should one exist and correctly report that a problem has no solutions when it does not is called completeness.

On line 3 of Figure 1-6, we select the cheapest potential solution for consideration. Specifically we select the node with the smallest g-value from the open list. $g(n)$ simply tells us what the cost of executing the partial, or complete in the case of a node with a goal

[1]The pseudo code in Figure 1-6 doesn't include duplicate detection. It would be performed on line 7. We will cover duplicate detection in detail in Chapter 7.

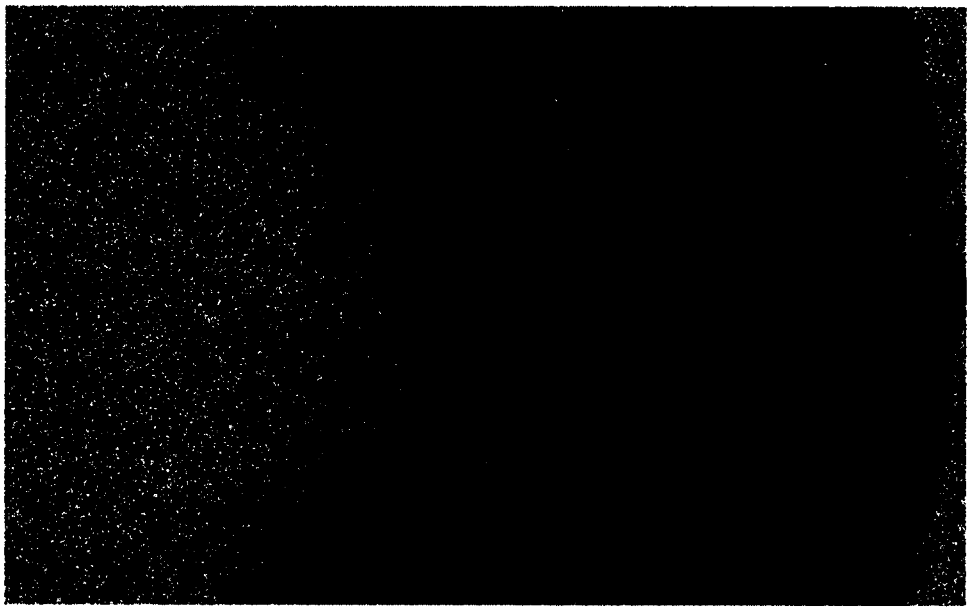


Figure 1-7: Expansion order of uniform cost search on a pathfinding problem

state, solution represented by a node is. There may actually be many nodes which share the smallest g-value, so strictly speaking the pseudo code is wrong. Tie-breaking is actually a very important part of a heuristic search algorithm, and can have a large impact on the performance of search algorithms [42, 66]. In the case of uniform cost search, breaking ties arbitrarily or in a first in first out order are both reasonable strategies.

Figure 1-7 shows the order in which uniform cost search considers nodes on a pathfinding problem of the variety we considered in Figure 1-1, albeit on a slightly larger scale. In this problem, the start state is in the middle of the left-hand side of the grid (near the yellow cells) and the goal is on the right-hand side (near the red cells), also in the middle. If a cell is black, it was an obstacle in the search, a place where the agent couldn't move. If the cell is white, the agent could have moved into it, but the agent never considered a plan moving through there. If a state was part of a plan considered by the search, then it is colored according to when the search considered it. If the state was considered early on, it is colored in yellow. As time progresses, the color of the cell becomes redder.

We can see that uniform cost search considers the nodes in a circular pattern, radiating

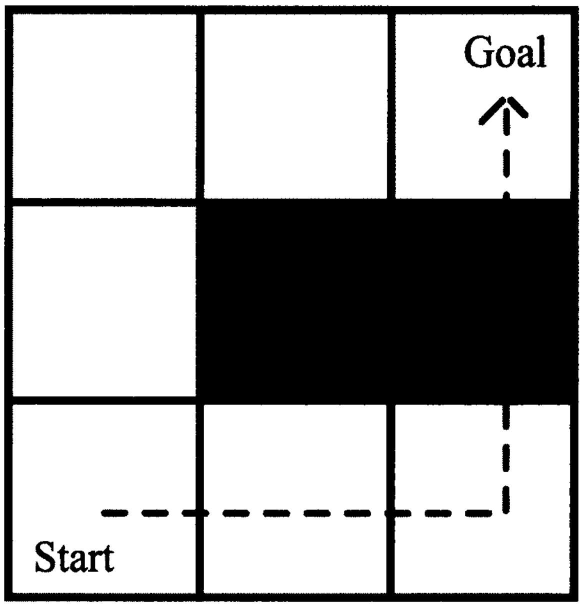


Figure 1-8: An example heuristic

outwards from the start state in the center left. Cells with approximately the same color were reached at about the same time by the search, and therefore have about the same cost. The reason that cost radiates almost evenly outward from the start state is that actions in this domain all have identical cost. If the actions had different costs, the expansion order would be quite different. Paying attention to action cost can be a determining factor in getting good performance out of heuristic search algorithms and is a topic that this dissertation will return to frequently.

1.2.2 A*

There is something particularly unsatisfying about the expansion order of uniform cost search as seen in Figure 1-7. The search algorithm considers paths going through many states which we can clearly see are not good choices. If the goal is to move through the grid from center-left to center-right, it makes little sense to consider states in the upper or lower left-hand corners. Certainly, we may need to consider them, what if the only solution to the problem was through there, but it is unsatisfying to see them considered so early on.

Our intuition about which states should and shouldn't be expanded by a search is more

```
**A*(root)**
1. $open \leftarrow \{root\}$
2. **while** $open \neq \{\}$
3.    let $n = \text{argmin}_{n \in open} f(n) = g(n) + h(n)$ in
4.       **if** $goal_p(n)$
5.       **then return** $n$
6.       **else** $open \leftarrow open - \{n\}$
7.          **for each** child $c$ of $n$, $open \leftarrow open \cup \{c\}$
8. **return** no solution
```

Figure 1-9: A* search pseudo code

formally called a heuristic. There are many sources of heuristic information that can be brought to bare when solving a problem, this is the primary focus of Chapters 3, 4 and I. One of the simplest, and probably most widely used, techniques for constructing heuristics for search is that of solving a relaxed version of the problem.

Relaxing a problem means that we ignore all of the interesting parts that made it difficult to solve in the first place. In the case of a pathfinding problem, we make the unrealistic assumption that no obstacles exist, and then we compute the cost of a path from the state being considered to the goal. An example of this for the starting state shown in Figure 1-1 is shown in Figure 1-8. Often we don't need search to construct the solution to a relaxed problem. As we can see for these grid navigation problems, the sum of the horizontal and vertical displacement of the current state from the goal state gives the exact cost of an optimal path from the state to the goal assuming no obstacles. This is called the Manhattan distance heuristic.

The pseudo code in Figure 1-9 shows a best-first search algorithm like uniform cost search modified to take a heuristic evaluation function into account. In fact the pseudo code is identical to that presented in Figure 1-6 but for a small change in line 3. Previously, nodes

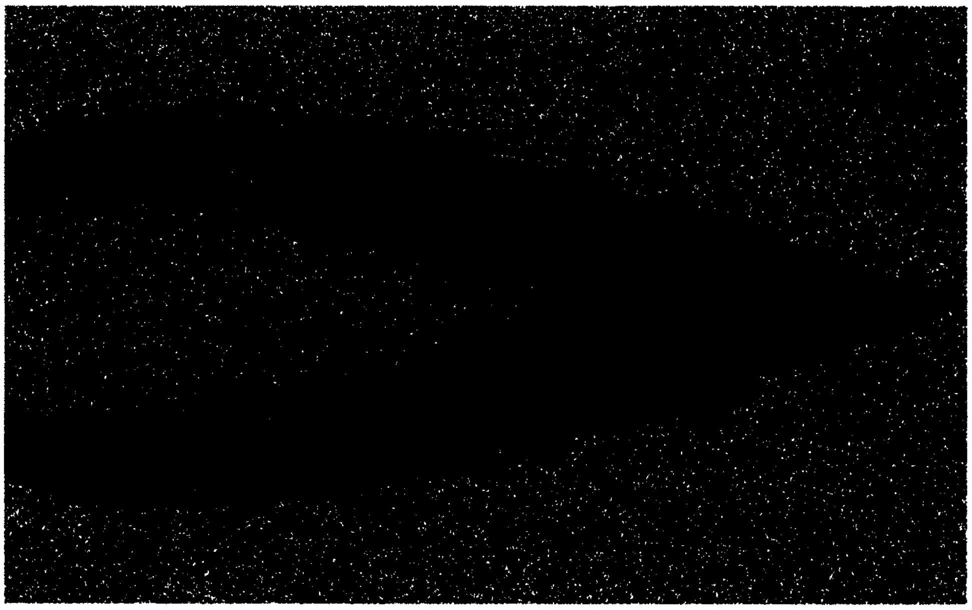


Figure 1-10: Expansion order of A* search on a pathfinding problem

were selected for having the smallest g-value, the smallest costing partial solution. This algorithm augments that by considering not only the cost of the current partial solution, but a heuristic estimate of the cost of completing that solution, $h(n)$. This is the A^* search algorithm.

Figure 1-10 shows the expansion order of A^* on the same problem as we saw in Figure 1-7. We can clearly see the influence of the heuristic on the search algorithm. The heuristic prevents us from exploring portions of the space that we recognize as unpromising which produces the flame-like search order seen in the visualization.

A^*[43] search is a heuristic search algorithm that, like uniform cost search, produces optimal cost solutions to a problem should one exist, provided the heuristic is *admissible.* Admissible heuristics always underestimate the true cost-to-go from the state on which they are computed to the goal. A formal proof of the cost-optimality of solutions returned by A* can be found in either [43] or [41], but the core of the argument is as follows. If $h(n)$ is an underestimate of the cost-to-go, then $f(n) = g(n) + h(n)$ must be an underestimate of the total cost of a solution through n. If we evaluate nodes in order of increasing $f(n)$,

```
A*(root)
1.  $open \leftarrow \{root\}$
2.  while $open \neq \{\}$
3.    let $n = \text{argmin}_{n \in open} h(n)$ in
4.      if $goal_p(n)$
5.      then return $n$
6.      else $open \leftarrow open - \{n\}$
7.          for each child $c$ of $n$, $open \leftarrow open \cup \{c\}$
8.  return no solution
```

Figure 1-11: Greedy search pseudo code

then when we do encounter a solution it will have an estimated total cost no greater than any other potential solution to the problem, and thus be cost-optimal.

While admissible heuristics are very useful for proving that a solution is cost-optimal, or that it has cost within some bounded factor of the cost of an optimal solution as we will see in Chapter 7, there is nothing inherently wrong with inadmissible heuristics, that is, heuristics which may potentially overestimate the cost-to-go from a state to the goal. Chapter 5 focuses on constructing powerful inadmissible heuristics, and in Chapter 7 and on we will see that inadmissible heuristics can be used to improve search performance substantially.

1.2.3 Pure Heuristic Search

We've been talking about optimal heuristic search algorithms, but the focus of this work is suboptimal search algorithms. Greedy search [15], sometimes called pure heuristic search is the simplest suboptimal heuristic search algorithm. Pseudo code is presented in Figure 1-11. Again, the primary difference between greedy search and the previous two algorithms is in line 3, where we determine the order in which nodes are considered by search. In greedy

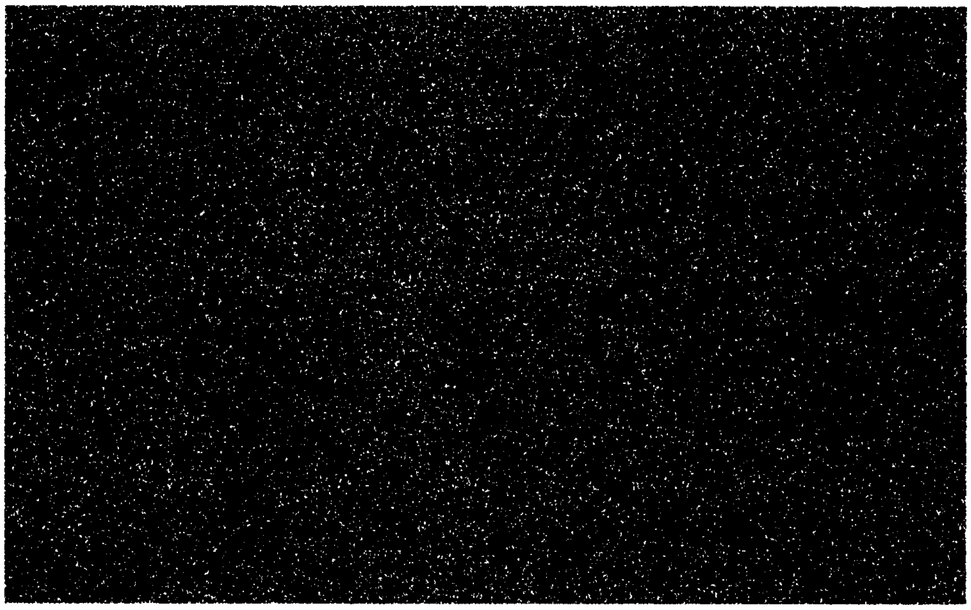


Figure 1-12: Expansion order of greedy search on a pathfinding problem

search, we only consider the cost-to-go heuristic, hence the name "pure heuristic search".

The order in which greedy search expands nodes is shown in Figure 1-12. In comparison to the previous two algorithms, greedy search expands very few nodes, proceeding nearly directly from the start to the goal. Unlike uniform cost search and A*, greedy search provides no guarantee on the cost of the solution it returns relative to optimal. Since it provides no guarantees, it does not spend any time considering solutions which may, in total be cheaper. It simply pursues the cheapest-to-complete solution in an effort to be fast. As we will later see in Chapters I and 7, pursuing solutions with low estimated cost-to-go is not the best approach to solving problems quickly, we should instead pursue solutions with few estimated actions-to-go.

1.3 Outline and Contributions of Dissertation

Optimal heuristic search is a well understood part of the artificial intelligence landscape. There are many algorithms for the optimal search setting that are well understood. More specifically, they are well understood both empirically, that is in terms of what the performance trade offs between the algorithms are, and they are well understood theoretically. We know what information the algorithms can rely, ie admissible heuristics, and we know how to derive powerful forms of that information directly from the description of the problem. Further, we know under what conditions which algorithms are guaranteed to be efficient and under what conditions they are likely to be inefficient.

The theoretical understanding of optimal search is the motivation of this work. While optimal search enjoys a well established theory, and the insight and improved algorithms that such an understanding brings, there is still no theory of suboptimal search. We do not have a listing of the sources of information that suboptimal search may find helpful, nor do we have a solid understanding of how these sources of information may be derived directly from the problem. Beyond reasoning backwards from empirical results, the field of suboptimal search has very little to say about the reasons why our algorithms perform well or poorly.

This dissertation is, hopefully, the beginning of such a foundation for the theory of suboptimal heuristic search. The dissertation falls roughly into two parts. In the first, we will discuss several kinds of information useful for suboptimal search algorithms and show how to construct these sources of information from the problem being solved. In the second, we will discuss suboptimal search settings and algorithms. In particular, we will show how relying on the sources of information constructed in the first portion of the dissertation lead to improved performance for suboptimal search. We will discuss a variety of suboptimal search settings: bounded suboptimal search, bounded cost search, and anytime search will be discussed in great detail, while other settings such as pure heuristic search and beam search will receive less attention.

The following provides a summary of the contents of each chapter, as well as its contributions to the field.

1.3.1 Estimating Actions-to-go

In this chapter, we discuss the construction of heuristics estimating the number of actions-to-go between a state and the goal. Actions-to-go estimates have appeared several times throughout the history of suboptimal search [41, 19], however they are not as commonly discussed as estimates of cost-to-go. The reasons for this are twofold. First, cost-to-go heuristics are needed for any search algorithm that wants to provide guarantees about the cost of a solution, either absolutely or relative to optimal cost. Secondly, many domains, particularly the puzzle like domains frequently used by the heuristic search community. There, estimating cost-to-go is identical to estimating actions-to-go.

The chapter serves a second role, namely it provides a detailed description of the domains used for evaluation throughout the rest of the dissertation. In addition to discussing the way in which the actions-to-go estimates are computed, we will also discuss how the admissible cost-to-go estimates are constructed, as well as other interesting aspects of the problems such as average branching factor and the number of goal states. Such features play a large role in determining the performance of the search strategies discussed in the latter half of the dissertation.

1.3.2 Constructing Inadmissible Estimates by Hand

In this chapter, we will discuss the simplest technique for constructing inadmissible estimates of cost-to-go, namely constructing them by hand using insight into the domain. We will discuss three general techniques for building inadmissible heuristics: book keeping while computing the admissible heuristic, taking the midpoint of an under-estimate and over-estimate, and combining multiple heuristics in potentially inadmissible ways.

While we know very well how to construct admissible heuristics from the description of a problem, the construction of effective inadmissible estimates is more of an art, having no formulaic approach like those enjoyed by admissible heuristics. The contribution of this chapter is to provide an outline for three general ways of constructing inadmissible heuristics from the description of a problem. The approaches here are not as automatic as those for constructing admissible heuristics, but they share many parallels with the automated construction of admissible heuristics.

1.3.3 Learning Inadmissible Estimates

This chapter discusses three techniques for constructing inadmissible estimates of cost and actions-to-go automatically using techniques from machine learning. We will look at three times when learning could produce an improved inadmissible estimate: before any search takes place, in between solving problems, and during the solving of a single instance. The latter of these is a major contribution of this dissertation to the field. Specifically, the idea that the search should inform the heuristic just as the heuristic informs the search is very important.

While the inadmissible estimates computed in the previous chapter require some amount of human ingenuity, those discussed in this chapter do not. Further, the heuristics contributed by this dissertation, those learned online, during search, have several desirable properties that their forerunners lacked. Namely, they do not require a large set of homogeneous instances to work, and they can learn corrections tailored to a specific instance of a problem without impacting performance on other instances from the same domain.

Further, the online heuristic learning could easily be used on top of the offline or interleaved learning approaches to improve the quality of already strong heuristics. The online correction techniques presented in this chapter make no strong assumptions about the properties, so there is no reason that they can't be used on top of previous techniques for constructing powerful heuristics before or in between searches.

1.3.4 Bounded Suboptimal Search

This chapter begins the second half of the dissertation wherein we discuss suboptimal heuristic search strategies. In this chapter, we will discuss bounded suboptimal heuristic search. These algorithms return solutions that are guaranteed to have cost within a bounded factor of the optimal solution cost to the problem. The chapter contains a definition of the problem of bounded suboptimal search and a lengthy discussion of many, if not all, of the algorithms for this problem setting.

This chapter contains three major contributions to the field of heuristic search: a definition of the goal of bounded suboptimal heuristic search, the explicit estimation search algorithm, and a study of much of the previous work in the area of bounded suboptimal search. The study of previous work is broken into two parts, an empirical evaluation of the algorithms on a wide set of benchmark domains and a more theoretical evaluation of the algorithms on a set of explicit graphs. The empirical evaluation shows that Explicit Estimation Search is generally faster and more robust than previous approaches, and the theoretical evaluation explains that this is the result of actually attempting to solve the problem of search under a suboptimality bound directly.

The problem definition and the theoretical evaluation hopefully are the beginnings of a theory of suboptimal search. Having a formal definition of the desired performance of algorithms, a concept of optimal behavior for bounded suboptimal search, is necessary for forming a theoretical foundation for the area.

1.3.5 Bounded Cost Search

This chapter investigates a relatively new setting for suboptimal heuristic search. Unlike algorithms in the previous chapter, which return solutions within a bounded factor of the optimal cost solution, these algorithms seek to find any solution beneath a user-supplied cost bound C as quickly as possible. The main contribution of this chapter are a bounded-cost variant of the Explicit Estimation Search algorithm unimaginatively called Bounded-cost Explicit Estimation Search, or BEES. The construction of BEES shows that the approach taken when constructing explicit estimation search can be applied effectively to a range of other settings.

1.3.6 Anytime Search

This chapter investigates the anytime search setting, in which search algorithms must find the best possible solution within an unknown time. In this chapter, we present a study of many algorithms for the anytime search setting. In particular, we look at three general frameworks for converting bounded suboptimal search algorithms into anytime algorithms. We examine the bounded suboptimal search algorithms presented in the earlier chapter within these frameworks.

When originally published, the study of frameworks and bounded suboptimal search algorithms was the first of its kind. The *d*-Fenestration algorithm presented here was an original contribution, although in the end it turned out the algorithm was not particularly competitive with other previously proposed work. Anytime Explicit Estimation Search, AEES, is another major contribution of this work. AEES is to anytime search what BEES is to bounded cost search: an application of the ideas that gave rise to EES to the problem of anytime heuristic search.

1.3.7 Summary

This dissertation attempts to lay the groundwork for a theoretical understanding of the area of suboptimal heuristic search algorithms. Such an understanding is important because it allows us to predict when suboptimal search algorithms will work well and when they will work poorly. A theory of bounded suboptimal search necessarily includes a formal definition of the problems solved using suboptimal search methods. Further, it requires an understanding of the kinds of information useful to suboptimal search and then techniques for constructing this information. Finally, it needs a set of baseline algorithms designed to solve the various problems that were previously laid out.

Part I

Sources of (Additional) Information

CHAPTER 2

INTRODUCTION

As we previously noted, heuristic search is a widespread approach to automated planning and problem solving. If time and memory permit, we can use algorithms such as A* [22] to find solutions of minimal cost. These algorithms require an admissible heuristic evaluation function, that is, a heuristic which never over-estimates the true cost-to-go from a node to a goal. Under mild assumptions it can be shown that no similarly informed algorithm can find provably optimal solutions while performing less work than A* [13]. Unfortunately, problems are often too large and deadlines are often too short for finding provably optimal solutions [24]. When optimally solving a problem is impractical, suboptimal search can be a practical alternative. Suboptimal search algorithms sacrifice solution optimality in an attempt to reduce the resources needed for solving problems.

In this dissertation, I will be talking about four varieties of suboptimal search: greedy best-first search algorithms, bounded suboptimal search algorithms, bounded cost search algorithms, and anytime search algorithms. While all four algorithms solve slightly different problems and are tailored towards different applications of suboptimal heuristic search, they do have at least one common point: they can consider inadmissible sources of heuristic guidance without sacrificing whatever guarantees about the solution they already provide. Greedy search provides no guarantees, so this is trivial, and we will discuss how to make use of inadmissible cost-to-go estimates in bounded suboptimal, bounded cost, and anytime search without losing guarantees of bounded suboptimality, bounded cost, and convergence to an optimal solution in Chapters 7, 8, and 9.

The chapters in this section of the dissertation investigate ways of constructing potentially inadmissible heuristics to guide search. We look at three sources of heuristic guidance:

estimates of actions-to-go, hand crafted inadmissible cost-to-go estimates, and estimates constructed by automated learning techniques. These heuristics all have differing sources, and even slightly different applications in the case of actions-to-go estimates and cost-to-go estimates.

Estimates of actions-to-go may be inadmissible because there are many domains where actions may have cost less than 1, one of example of this is a TSP problem laid out on a unit square. The distance between the towns is a number that must be larger than 0 but less than $\sqrt{2}$. There are many values in that range with cost less than 1. In Section 3, we will discuss ways of constructing distance-to-go estimates for all of the problems considered in this dissertation. We will additionally be discussing the construction of the cost-to-go heuristics of the domains as well as other interesting properties of these domains, such as average branching factor and the number of potential solutions to a problem.

Anyone who has taught an introductory course in artificial intelligence can attest to the ease of constructing an inadmissible estimate of cost-to-go by hand. Even when asked to produce an admissible heuristic, many students will produce inadmissible heuristics because they are more in line with the non-technical definition of a heuristic: a general rule that may occasionally be violated. In Section 4, we will discuss techniques for constructing inadmissible estimates of cost-to-go by hand.

Finally, we consider learning as way to construct these inadmissible estimates of cost-to-go in Section 5. There are three possible settings where inadmissible estimates of cost-to-go can be learned automatically from data: before any search begins, interleaved with the solving of many instances, and during the solving of a single instance. We will discuss all three approaches in Section 5, although the focus will be on the online learning of cost-to-go estimates, as that is the primary contribution of this work to the area of learning inadmissible heuristics.

CHAPTER 3

ESTIMATING ACTIONS-TO-GO

3.1 Introduction

This chapter discusses the derivation of estimates of actions-to-go for use in suboptimal heuristic search algorithms. Before we go too far along the path of finding out how to compute estimates of the remaining actions, we should first consider why it is we want those estimates. As we discuss in detail in Chapter 7, Chapter 8, and Chapter 9, in suboptimal search settings, the time required to find a solution is often incredibly important. We will argue in Chapter 7 that bounded suboptimal search algorithms should find a solution within the user supplied bound as quickly as possible. A nearly identical argument will be made for the bounded cost setting in Chapter 8, and a similar discussion will be part of Chapter 9.

Since suboptimal search algorithms are often quite concerned with the time consumed while solving problems, we should have some way of estimating the difficulty of converting the partial solution represented by a node into a complete solution. Estimating the actual time required to solve a problem is an open problem in heuristic search, but the number of actions remaining will provide a good proxy for the required effort to complete.

The difficulty of solving a problem using heuristic search is strongly tied to the size of the tree induced by search. The size of this tree is determined by two things, the branching factor and the depth. If the size of the tree is roughly b^d, where b is the branching factor and d is the estimated depth of the tree, then clearly d plays a very important role in determining the difficulty of solving a problem.

What holds for the whole search tree is also true of the nodes in that tree. We can roughly guess how difficult it will be to convert some node in our search tree into a complete solution by estimating the number of actions between the state that node represents and

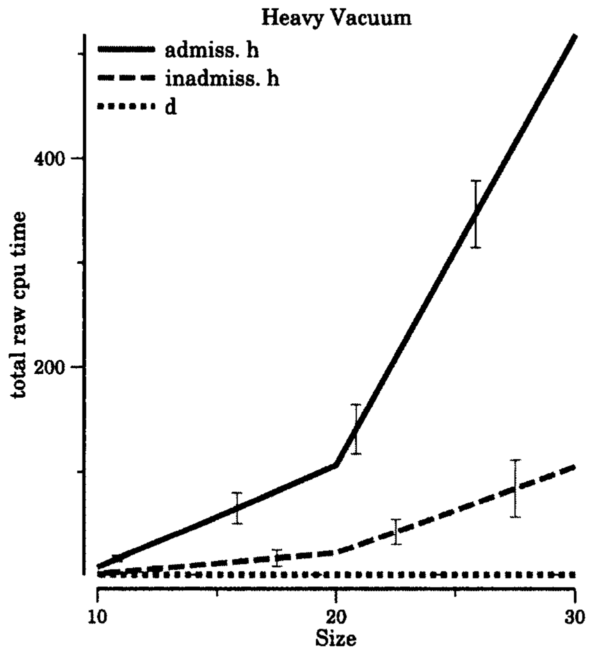


Figure 3-1: Inadmissible estimates of cost and actions-to-go improve speed

the goal. By ordering nodes on $d(n)$, their estimated actions-to-go, we are able to order nodes roughly on their cost of completion.

In Figure 3-1, we see that ordering nodes in terms of actions-to-go in greedy search results in faster search for the heavy vacuum domain which we will discuss later in this chapter. In this plot, we show the size of the instance on the x-axis, and the amount of time required by greedy search to find a solution in seconds on the y-axis. Not only is greedy search on actions-to-go, d in the plot, faster than search on either admissible cost-to-go or inadmissible cost-to-go, admissible h and inadmissible h respectively, but it also scales better than either of these on the heavy vacuum problem. The bulk of the dissertation will be interested in how estimates of actions-to-go allow us to speed up search algorithms in a variety of settings by allowing us to order nodes roughly by their cost of completion. The scaling behavior is interesting as well, but will not be investigated thoroughly here.

Now that we have motivated the need for actions-to-go estimates, we will discuss techniques for constructing them directly from the description of a problem. This chapter will also serve as a description of all of the domains used in the evaluations throughout the dissertation, as we will have to discuss the domain in detail to understand how to construct $d(n)$ for a given domain.

3.2 N-Puzzle

The n-puzzle, sometimes the $n^2 - 1$-puzzle, or the sliding tile puzzle is the fruit fly of the heuristic search community. That is, it is probably the most commonly experimented upon problem in heuristic search literature, and for good reason. The puzzle is simple to describe, simple to represent, and while small versions of the puzzle are easy to solve, as n becomes large, the problem becomes incredibly difficult to solve. Large, of course, depends on the kind of solution we want to find, ie optimal, bounded suboptimal, and so on.

In general, the n-puzzle refers to a sliding tiles puzzle with n pieces and 1 blank. Initially the tiles have some unknown configuration, and the goal is to, by sliding tiles from their current position into the blank, to convert the initial configuration of the puzzle into the goal configuration. Sometimes the tiles are simply numbered, sometimes they are pieces of an image that must be reformed, sometimes parts of words appear on the tiles, but no matter the goal, the problem is essentially equivalent. Although many configurations exist, we will consider only square puzzles in this dissertation.

3.2.1 Eight Puzzle

The eight puzzle is the smallest variant of the sliding tiles puzzle we consider in this chapter. It is a 3x3 grid containing the numbers 1 through 8. The goal configuration is to put the blank in the upper left-hand corner, and the numbers 1 through 8 following from left to right behind the blank. If we think of the blank as having the number 0, then the idea is to convert whatever original permutation of the numbers existed into the sequence 0..8.

In any state of the puzzle, our available actions are dictated by the position of the blank. All we can do in a given state is move one of the tiles adjacent to the blank into the blank. Later, we will discuss a variant of the n-puzzle that more accurately reflects the physical puzzle in that multiple tiles may be moved at once. In the tiles puzzle, generally actions all have the same cost, 1. Such domains are referred to as unit-cost domains. In these domains, estimating the cost-to-go is identical to estimating the actions-to-go.

In the following evaluations, we will predominantly rely on the Manhattan Distance heuristic for estimating the cost-to-go in tiles puzzle. For each tile, we compute the horizontal and vertical distance between it and its home location, ie x moves left or right, y moves up or down, report a value for the tile of $x + y$, and we sum this value for all tiles on the board. In practice, we do not compute these values anew for each state. Instead, we construct a look-up table before search begins so that we can simply look-up the distance of a tile from its home position, rather than going to the trouble of performing simple arithmetic. This heuristic is a relaxation of the original problem in the sense that it assumes we can simply slide one tile through another, which is obviously not true in the real puzzle.

3.2.2 Fifteen Puzzle

We will also examine the 100 instances of the 15-puzzle presented by Korf[32]. Again, we will predominantly use the Manhattan distance heuristic for both $h(n)$ and $d(n)$. Other more informed heuristics exist, for example we could add Manhattan distance and linear conflicts [39] or use memory based heuristics like pattern databases [11], but these techniques have drawbacks such as being more expensive to compute or requiring large amounts of precomputation to construct.

This implementation of the 15-puzzle is not as fast as others. Expansion rates upwards of a million nodes a second have been reported in the literature, however the implementation used here is capable of handling arbitrary sized puzzles, macro-actions, and interesting cost functions. Each of these comes at the cost of increased per-node overhead. In relation to the other domains in the study, this solver is the second "fastest" in the sense of nodes per second.

Another interesting feature of the n-puzzle is that it has incredibly long cycles, three to four times as long as the other cycle-containing domains considered in the dissertation. When we discuss the length of cycles in this dissertation, what we mean is cycles excluding the trivial two-action cycle of making and immediately undoing an action, for example moving a tile to the left and then immediately moving the same tile back into the blank it created. Such actions are not helpful and can be safely pruned in almost all cases. When computing the average branching factor of a domain, we will also use this optimization. The importance of cycles and the duplicate nodes they create will be a recurring theme in Chapter 7.

3.2.3 Macro Fifteen Puzzle

The macro fifteen puzzle is a more faithful representation of the sliding tile puzzle than the previously discussed domains. In this variant, we might move one, two, or even three tiles at a time. That is, we can slide a single tile, a portion of a row or column, or the entire row or column one cell in the direction of the blank, much like we can move multiple tiles in the real puzzle with a single slide of our finger. However, even when moving multiple tiles, we only charge a single unit of cost for the action.

Being able to move multiple tiles at the same time does change the way in which we compute the admissible estimates of cost-to-go. We still rely on the Manhattan distance of all of the tiles from their home location, but this value may now over estimate the true cost-to-go as a result of the macro actions. In order to keep the Manhattan distance estimates admissible, we must account for the fact that we can now move up to three tiles one space closer to their goal locations in a single action. That is, we can simply divide the Manhattan distance by three in order to get an admissible heuristic estimate for the cost-to-go. The resulting heuristic is admissible and still consistent, but it is no longer integer-valued, which can be important in obtaining an efficient implementation of a search algorithm, something we will discuss in greater detail in Chapter 7.

Now that we have discussed how to compute the cost-to-go for this domain, we consider how to compute the number of actions remaining between a given state and the goal state [1]. The simplest way of computing the number of actions remaining is to have solutions that

[1]The goal state because the fifteen puzzle has a single canonical goal state.

always exist at a fixed depth, such as they do in the traveling salesman problem. When solutions do not exist at a fixed depth, the simplest way of computing an estimate of the number of actions between a state and the goal is to perform some book-keeping while estimating cost-to-go in order to compute the number of actions-to-go simultaneously.

In the case of the macro fifteen puzzle we are considering moving each tile individually one space at a time from its current position into its goal position. For each space moved we charge it $\frac{1}{3}$ because in the ideal case we could be moving up to three tiles at a time at unit cost, and we can not allow a potential over-estimation of the cost. As we tally $\frac{1}{3}$ for the cost of the action for each move, we can also tally 1 for the number of actions we suspect we will have to take to solve the problem. In this case this ends up being exactly the Manhattan distance heuristic that we used for the previous versions of the sliding tile puzzle.

The macro fifteen puzzle is unique in the domains evaluated here in that it is the only problem which has unit-cost actions and differing base estimates of cost and actions-to-go, h and d respectively. Any action could move multiple tiles, so we must divide the costs of all movement under the assumption that all tiles will be moved at the same time as two others in order to maintain admissibility. However, having probably solved a number of these problems ourselves as children, we recognize that many of the actions will not be to move all tiles in a row or column simultaneously, and that such moves are often not beneficial. Thus, the standard, undivided Manhattan distance provides a reasonable estimate of the length of solutions for this domain.

Certainly we could construct different estimates of the length of the solution for the standard 15 puzzle, without macro actions. We might consider using a more informed estimate of the length of the solution, one that could potentially over-estimate the number of actions required. This is really more like an inadmissible estimate of cost-to-go, the subject of the next two chapters. In the end, the distinction between inadmissible estimates of cost-to-go and inadmissible estimates of actions-to-go on unit cost problems is really a purely academic one. Still, it is a distinction that is important to make because it helps us think about more complicated problem settings where the cost of individual actions differ.

3.2.4 Inverse Cost Fifteen Puzzle

We will examine the same 100 instances of the 15-puzzle used in the standard 15 puzzle and the macro 15 puzzle discussed in the previous two sections, but with yet another function used to determine the cost of actions. In the inverse tile puzzle, the cost of moving a tile into the blank is $\frac{1}{face}$ where $face$ is the number on the tile. So moving the 15 tile costs $\frac{1}{15}$ and moving the 8 tile costs $\frac{1}{8}$. The heuristic is simply a modified Manhattan distance. For each tile we compute its displacement from its goal location, and then multiply this distance by the action-cost for moving that particular tile. This is then summed up for all tiles.

While it may seem ridiculous to study nearly identical problems with slightly differing cost functions initially, it allows us to separate out the impact of action costs from basically all other aspects of a domain when evaluating the impact of action-costs on heuristic search algorithms. By holding the branching factor, number of goals, average solution depth, cycle length, and other important domain features constant while only varying the cost-function for actions, we can get a better idea of how exactly the cost of actions impacts heuristic search.

The inverse tiles problem has a relatively wide spread of action costs, but what is particularly interesting is that all of these costs are less than 1. This means that, strictly speaking, the distance-to-go estimate for this domain frequently over estimates the cost of the solution for this problem. In other domains with action costs, for example the heavy vacuum domain we are about to discuss, estimates of actions-to-go are generally far lower than the cost-to-go.

As before, to compute the actions-to-go estimate for the inverse 15 puzzle, we must simply keep track of the number of actions we estimate we will take while computing the cost-to-go estimate $h(n)$. In this case, as with the macro 15 puzzle, that ends up being the Manhattan distance for all of the tiles, summed together.

3.2.5 Twenty-Four Puzzle

We also consider a 5x5 sliding tile puzzle with 24 tiles in our evaluation. These puzzles are considered primarily in the context of learning heuristics for search in Chapter 5. Increasing the size of the problem, even by such a small amount, increases the difficulty of solving the problem dramatically. h and d are computed identically in the 24 puzzle using the Manhattan distance.

3.3 Vacuum World

The vacuum world is domain motivated by the first search space described in [52]. In it, a small vacuum must navigate a room, modeled as a grid, and vacuum up all of the piles of dirt. Naturally the room is not completely free of furniture, so we model these obstructions to the movement of the vacuum robot as blocked cells on the grid. The robot can turn on a dime, but can only move in the cardinal directions. The problem is solved when no piles of dirt remains. We consider two variants of the vacuum world problem, one with unit action costs and one with actions of varying cost.

The vacuum world problem is much like a mixture of the traveling salesman problem and grid world navigation. In fact, at least for the unit cost variant of the problem, we could solve these problems by computing all pairs shortest paths for all points on the grid, or at least the vacuum and all dirty cells, and then solving the resulting problem as if it were a TSP with a number of cities equal tot he number of dirts plus the vacuum. For problems of the size we consider in this chapter, solving such a TSP problem is pretty simple, however the point isn't to construct the fastest solver for an imagined problem, but rather to understand the impact of domain features on solver performance and to have a wide variety of domain features present in our evaluation. In the case of the vacuum problem, these features are the inconsistency of the cost-to-go heuristic, the relatively low branching factor, the tight cycles and large number of duplicates, and the fact that there exist multiple goal states.

3.3.1 Unit-Cost Vacuum

We consider two variants of the vacuum problem, one with unit-cost actions and one without. We will discuss the variant with unit-cost actions first. We consider two sizes of unit-cost vacuum worlds. For measuring the relative performance of bounded suboptimal and anytime search algorithms, we used 100 instances that are 500 cells tall by 500 cells wide, each cell having a 35% probability of being blocked. We place twenty piles of dirt and the robot randomly in unblocked cells and ensure that the problem can be solved. When measuring the accuracy of heuristics, we look at 100 instances that are 200 by 200 with 5 piles of dirt. These smaller instances can be exhaustively searched on our computers, while the larger problems cannot be exhaustively enumerated.

3.3.2 Heavy Vacuum

We examine 150 instances of vacuum problems in our evaluations. Each instance is on a 200 by 200 grid. Each cell has a 35% chance of being occluded. Once the obstacles are laid down, 10 piles of dirt and the vacuum are placed randomly on the board. We then check to make sure the problem is solvable by making sure that the robot and dirt piles are in the same connected component of the grid. The cost of taking an action is 1 plus the number of dirt piles that the vacuum has already cleaned up. So initially all actions cost 1, then 2, and so on up to a cost of 10.

The cost-to-go heuristic is computed as a minimum spanning tree of the robot and dirt piles. Once the minimum spanning tree is computed, the edges in the tree are sorted in order of length, longest first. We then weight the edges based on the current action cost. The longest edge is weighted by the current cost of acting, the next longest edge gets the current cost plus one, and so on.

Estimates of actions-to-go are computed by assuming the problem contains no obstacles, and then computing a greedy traversal of the dirt piles. That is, the vacuum moves to the nearest pile, then the next nearest, and so on. We compute most of this information while constructing the spanning tree, so computing this more informed action-to-go heuristic is surprisingly cheap. While we could compute the actions to go simply by counting the length of each arc in the spanning tree instead of the weighted arc length as we do for computing $h(n)$, this estimate of distance-to-go ends up being more informed for little additional cost.

One interesting thing about the heavy vacuum domain is that the heuristic for this domain is inconsistent. That is, the heuristic between two states will often differ by more than the cost of the transition between them. This is because the heuristic is based on a spanning tree including the agent. Moving the agent can alter the cost of all edges in the spanning tree, which is what gives rise to the inadmissibility. Pilot experiments showed that less informed admissible heuristics lead to longer solving times.

3.4 Life Cost Grids

Life-cost grids were first proposed by Ruml and Do[51]. They are a standard 4-connected grid with a slightly different cost function, moving out of a cell has cost equal to the y-coordinate of the cell. The instances studied here are 2000 by 1200 cells, with the starting location in the lower left hand corner of the grid and the goal location in the lower right. As a result of the cost function, cheap paths involve moving up from the starting location towards the top of the grid, cutting across, and coming back down to the goal. It is called the "Life" cost function because cheap solutions incorporate many economizing steps, much like many tasks in real life.

Computing cost-to-go for life cost grids is slightly more complicated than using simple Manhattan distance. It is easiest to think of the heuristic as ignoring all obstacles on the board and computing the cost of the cheapest solution from the current state. In the case of life cost grids, the cheapest solution will take one of two forms. Either an 'L' shape is produced where the agent makes a string of horizontal moves and a string of vertical moves, horizontal followed by vertical if the agent is north of the goal, vertical then horizontal if the agent is south of the goal or alternatively the agent moves in a 'n' shape, straight up for some number of moves, then across and down.

We compute the cost of both solutions and take the cheaper of the two. The choice of which solution to take also impacts the actions-to-go estimate for the state. In the case where we take the 'L' shaped path, the actions-to-go can be estimated by Manhattan distance. In the 'n' shaped paths, we must count the up, down, and horizontal actions which are usually much larger than the Manhattan distance between the agent and the goal.

The life cost grids have the largest spread of action costs, spreading over a range at least an order of magnitude larger than other domains with action costs. Despite this, algorithms which paid attention to the difference between solution length and solution cost did not fare as well on this domain as they did in others, as we saw in the evaluation in Section 7. We suspect that this is because the cost-to-go heuristic for this domain is particularly strong. Paying attention to an additional source of information has several benefits, but one of them is to shore up weaknesses in some of the heuristics [50].

3.5 Dynamic Robot Navigation

This domains follows that used by Likhachev, Gordon and Thrun [35]. The goal is to find the fastest path from the starting location of the robot to some goal location and heading, taking momentum into account. We perform this search in worlds that are 500 by 500 cells in size. We scatter 75 lines, up to 70 cells in length, with random orientations across the domain and present results averaged over 100 instance. The cost-to-go heuristic is constructed by computing the optimal distance of every location of the board to the goal location, call this $h_{static}(n)$. $h(n) = \frac{h_{static}(n)}{maxvelocity}$ and $d(n) = h_{static}(n)$. That is, the admissible heuristic is simply the length of the shortest static path divided by the maximum speed of the robot, and the estimated number of actions is the length of the static path.

Dynamic robot navigation has by far the larges branching factor of all of the domains considered in this study, with a maximum branching factor two orders of magnitude larger than other algorithms, and an average branching factor one order of magnitude larger than other domains. This is because we are considering a large number of potential headings and speeds for the robot, and any of these could change between two search nodes.

3.6 Dock Robot

We implemented a dock robot domain inspired by Ghallab et al[20] and the depots domain from the International Planning Competition. Here, a robot must move containers to their desired locations. Containers are stacked at a location using a crane, and only the topmost container on a pile may be accessed at any time. The robot may drive between locations and load or unload itself using the crane at the location. We tested on 150 randomly configured problems having three locations laid out on a unit square and fifteen containers with random start and goal configurations. Driving between the depots has a cost of the distance between them, loading and unloading the robot costs 0.1, and the cost of using the crane was 0.05 times the height of the stack of containers at the depot. h was computed as the cost of driving between all depots with containers that did not belong to them in the goal configuration plus the cost of moving the deepest out of place container in the stack to the robot. d was computed similarly, but 1 is used rather than the actual costs.

The dock robot domain has a large number of legal goal states, far larger than most of the problems here. While tiles, inverse tiles, life grids, and dynamic robots all have a single canonical goal, dock robots only specifies in which pile the crates must be at the end of search. It says nothing about the ordering of those crates in the goal pile, which is why there are so many legal configurations. It is rare that all crates would need to be moved to one pile, which has the largest number of legal configurations at 1,307,674,368,000, and it is also rare that each pile would contain five crates, which has the smallest number of legal goal configurations at 360. This makes computing the heuristic particularly challenging for this domain.

3.7 Summary

There are a wide variety of problems that can be solved using suboptimal search techniques like the kind discussed in this dissertation. Table 3-1 gives a brief summary of many of the important properties of the domains used in this dissertation. The domains themselves

	Tiles	Inv. Tiles	Life	H. Vacuum	Dyn. Robot	Dock
Max Branch	4	4	4	5	150	4
Avg Branch	2.13	2.13	1.66	1.67	51.41	3.08
Action Costs	1	$\frac{1}{15}$–1	0–1200	1 – 10	$\frac{1}{20}$ – 1	0 – 15
Shortest Cycle	12	12	4	4	None	3
Consistent h	Yes	Yes	Yes	No	Yes	No
# Goals	1	1	1	10	1	$3 \cdot 5! \leq i \leq 15!$
$\frac{\text{Nodes}}{\text{sec}}$	642577	52460	771680	20958	568859	108495

Table 3-1: Properties of the domains under investigation

span navigation problems for a vehicle with dynamics, organizing crates at a ship yard, and finding the solution to a puzzle. This doesn't even begin to cover the spectrum of problems approachable with heuristic search techniques, but it does provide a decent range of important domain properties.

The table presents the domains and columns, and attributes as rows. "Max Branching" reports the maximum possible branching factor for the domain. "Avg Branching" reports the average branching factor experienced by a uniform cost search run to completion or until it exhausted memory on all of the instances. "Action Costs" reports the range of action costs for the domain, from least cost action to most expensive action. "Shortest Cycle" reports the length of the shortest route by which the search may leave and return to a give node, assuming that we disallow the trivial two-step cycle of doing and undoing a move. "Consistent h" denotes whether the base cost-to-go heuristic was consistent for the domain. "Number of Goals" reports the number of goals to a given problem from the domain. "$\frac{\text{Nodes}}{\text{sec}}$" reports the rate with which search can, on average, generate states. We computed this by examining the number of nodes per second generated by greedy search, as this is the algorithm with the least overhead that also computes the heuristic of all states. Nodes per second will obviously differ from algorithm to algorithm, but this provides a sort of lower bound.

CHAPTER 4

CONSTRUCTING INADMISSIBLE ESTIMATES BY HAND

4.1 Introduction

In this chapter we consider several techniques for constructing inadmissible estimates of cost-to-go, which we will refer to as $\widehat{h}$, by hand. The techniques contained in this chapter are not specific to any of the domains considered here, the domains for which we exhibit the techniques are simply illustrative.

4.2 Book Keeping

The simplest technique by which we can compute an inadmissible estimate of the cost-to-go from a state to the goal shares much in common with the technique by which we computed the distance to go. When computing the admissible cost-to-go, we just need to perform a small amount of book keeping in order to construct an inadmissible estimate as well. Specifically, we will be looking at the relaxed solution constructed by the admissible heuristic and charging it for any violation of the rules of the real problem that it is trying to solve. We will use the Life-cost grid navigation problem as an example.

The idea that a heuristic is computing a solution to a relaxed version of the problem is a common one, as we have already briefly discussed. In the example in the introduction, in Figure 1-8, we showed that we could think of the Manhattan distance heuristic on a grid as the solution to a relaxed version of that problem, Similar ideas come up in all areas of heuristic search. In the dynamic robot problem, we are ignoring the dynamics of the robot and instead solving the simpler static version of the problem. In dock yard robots, we are assuming that we only care about one crate on each pile, the deepest one. In tiles, we assume the tiles can move through one another, even though they are physically unable to do so. The same observations can be made of nearly any admissible heuristic.

In life cost grids, we estimate the cost-to-go by constructing one of two paths through the grid, assuming that there are no obstacles. Either the path goes up and over, or up, across, and back down. Let's assume for the moment that the path goes up and over in an 'L' shape. Now, normally we wouldn't explicitly construct the path, we would simply use Manhattan distance and the current y-value of the agent's location to compute the cost, but let's assume that we construct the whole path.

If we were to look at every grid-cell traversed by the relaxed solution to the problem, we would see that some of the cells are free and some of them are blocked. Every time the relaxed solution passes through a blocked cell, it has violated the real constraints of the problem. This is why the heuristic underestimates the true cost-to-go, it takes cheap moves that are not actually legal. If we could charge the relaxed plan for each illegal move it makes, we would likely get a more powerful heuristic.

The reason that such a technique is a by-hand construction of an inadmissible heuristic and not an automated construction of the inadmissible heuristic, as we will be discussing in the following chapter, is that it is not clear how we should charge the relaxed plan for this violation. In the case of life cost grids, we might consider charging it twice the cost of moving through a free cell in the same row. This assumes that we will have to make some additional moves in a previous or subsequent row in order to avoid the occluded cell here.

There are, however, obvious problems with just charging twice the cost of the row. Obviously, we will not be passing through this cell in the real solution, because we can't, and yet we have not altered the rest of the relaxed plan. When computing the remainder of the heuristic, we may be adding on penalties that we will no longer experience because we will need to deviate from the relaxed plan early on. Similarly, it is unclear if twice the cost is the best choice. Three of four times the cost could also perform well. The need to do such tuning to the inadmissible heuristic computations is the reason that such heuristics are constructed by-hand, and not truly automatically constructed.

4.3 Mean of Under and Over Estimates

Sometimes, a benchmark or problem where we would normally apply heuristic search is only difficult to solve because we want optimal or near optimal solutions. In these situations, it is often the case that a suboptimal solution to the problem can be constructed in polynomial time. The traveling salesman problem and the sliding tile puzzle are excellent examples of this. In the case of the traveling salesman problem, we can simply greedily go to the next nearest city. This constructs a valid, often expensive, solution to the problem. Similarly, there exists a recursive decomposition of the sliding tiles puzzle, where the right most column and bottom row are solved, and then we recur inward to the $n-1$-puzzle, and so on until the problem is solved[55]. The solutions computed this way are often quite expensive but they are legal.

In these situations, we can compute an upper bound on the cost of an optimal solution to the problem. Specifically, the cost of a solution to the problem must be at least as large as the cost of the optimal solution to the problem, so it acts as a natural upper bound on optimal solution cost. We also have a lower bound to the cost of the optimal solution in the form of $h(n)$, the admissible heuristic. Obviously the true cost-to-go, $h^*(n)$ must be somewhere between this upper and lower bound.

If we have no idea how far off the pre-computed solution is from optimal, a simple and rational choice is to simply compute the mid point between the two values and use this as the cost to go heuristic. If we have some notion of the cost of the suboptimal solution to the problem relative to the optimal cost solution, then we could perform a weighted average of the two values to get a more reasonable estimate of the true cost-to-go.

It is interesting to note that we do not necessarily need a poly-time solution to the problem to be able to employ an approach like this. Consider the dynamic robot navigation problem, for which we do no know of a poly-time solution. We could substitute a solution found with greedy search for the polynomial solution to the problem, if we're relative certain that such a solution can be found quickly. The greedy solution has all of the desirable properties of the previously discussed constructions save for one: we do not have any bound on the amount of time it can take to find a greedy solution to the problem. There are many situations in which simply solving the problem greedily is quite challenging, for example the 35-puzzle.

Now, we should not simply use the cost of solution computed from the root of the problem when computing $\widehat{h}$ for all states in the problem. This would simply inflate the cost-to-go estimate for all states evenly, and would have very little benefit in most search algorithms. We also, realistically, can't compute a complete solution from each state in the space. Although we may be able to solve the problem in poly-time, we would like our search algorithms to expand tens of thousands to millions of nodes per second, so constructing a complete solution from each node is right out. Instead, we can subtract the cost of arriving at a node, $g(n)$, from the cost of the suboptimal solution computed at the root to get a quick estimate of the cost of a suboptimal solution from this node. Of course, this assumes that the search is moving towards, and not away from, the goal.

4.4 Weighted Sum of Features

The final approach for constructing an inadmissible estimate of the cost-to-go from a node to the goal by hand is simply a more general version of the previous approach. Rather than take a weighted sum of the admissible heuristic and an upper bound on the true cost to solve a problem, we can take the weighted sum of a set of arbitrary features, include these two elements or not as we see fit.

One of the first examples of inadmissible heuristics for search is of this variety. Nilsson[39] once suggested that we could use the Manhattan distance plus three times the number of linear conflicts in a state of the eight puzzle to estimate the true cost-to-go rather accurately. While the Manhattan distance heuristic and the linear conflicts summed together is a powerful admissible heuristic, by weighting the linear conflicts component, Nilsson[39] produced a powerful inadmissible heuristic to the problem. Finding the proper weighting requires either expert insight into the domain, a fair amount of testing and revising, or large amounts of data and machine learning.

This approach is really the foundation of much of the next chapter. We will see that there are many ways of automatically finding a good set of weights for a given set of features if we want to accurately estimate the true cost-to-go for search. Typically, we will find these weight by writing down for many states the values of the features and the true cost-to-go, and then performing machine learning to find a set of weights that most closely reproduces the true-cost-to go from the features.

4.5 Summary

In this chapter we discussed three techniques for computing inadmissible estimates of the cost-to-go from a description of the problem. While the techniques should be easy to apply to any domain of interest, they must be carefully applied. When charging for violations of the real problem in the relaxed solution computed by an admissible heuristic, we must think carefully about how much we will charge. The idea of using the mid-point between an admissible cost-to-go estimate and the cost of a suboptimal solution to a problem is a powerful one, but we may not be able to easily construct a suboptimal solution to the problem. Finally, the weights and the features in a weighted sum of features must be carefully selected if we want the resulting heuristic to be an effective one. That is not to say these techniques are not all useful for constructing inadmissible heuristics, they certainly are, but they cannot be automatically derived in the same way that the heuristics discussed in the next chapter can.

CHAPTER 5

LEARNING INADMISSIBLE ESTIMATES OF COST-TO-GO

5.1 Introduction

Heuristic search is a widespread approach to automated planning and problem solving. If time and memory permit, we can use algorithms such as A* [22] to find solutions of minimal cost. These algorithms require an admissible heuristic evaluation function, that is, a heuristic which never over-estimates the true cost-to-go from a node to a goal. Under mild assumptions it can be shown that no similarly informed algorithm can find provably optimal solutions while performing less work than A* [13]. Unfortunately, problems are often too large and deadlines are often too short for finding provably optimal solutions [24]. When optimally solving a problem is impractical, suboptimal search can be a practical alternative. Suboptimal search algorithms sacrifice solution optimality in an attempt to reduce the resources needed for solving problems.

We will focus on two types of suboptimal search algorithms: greedy best-first search algorithms that attempt to find solutions of high quality as quickly as possible while providing no guarantees on solution quality [15], and bounded suboptimal search algorithms that return solutions whose cost is guaranteed to be within some user-provided factor of optimal. Suboptimal search algorithms tend to be faster than their optimal counterparts because they do not need to prove that the solutions they return are optimal. By not proving solution optimality, they avoid having to expand all nodes that could potentially lead to a solution of lower cost. Because suboptimal search algorithms do not prove solution optimality, they can consider inadmissible sources of heuristic guidance.

This chapter investigates learning as way to construct these inadmissible estimates of cost-to-go. We are not the first to consider guiding search algorithms with inadmissible learned heuristics. As we later discuss in detail, several authors have proposed learning informed inadmissible heuristics by recording for many states the true cost-to-go, which we call h^*, and a set of features. They then learn a function from the features to a potentially inadmissible estimate of the cost-to-go, which we call $\widehat{h}$. Such an approach makes the limiting assumption that we either have access to a representative training set, or the ability to generate one automatically and sufficient resources to find h^* for many states. It further assumes that the training instances and test instances are similar enough to one another for the learning on the training instance to transfer effectively to the instances we truly care about solving. This can be problematic in settings, such as STRIPS planning, where instances can be very different from one another because of the expressivity of the problem description language.

In this chapter, we demonstrate that learning heuristics during search itself is a practical and effective alternative to learning before search or learning interleaved with search. In Section 5.2, we present a new technique for improving heuristics during the execution of search, called single-step correction. It improves a given initial heuristic based on observing its behavior over paths in the search tree. We prove that, assuming knowledge of the heuristic's behavior over the entire search space, our techniques will produce perfect heuristic estimates. Although this assumption will rarely be met in a real problem, it does demonstrate that the technique is theoretically sound. In Section 5.2.3, we demonstrate that it works well in practice in an empirical study across eight benchmark domains. Heuristics learned during search find solutions up to three orders of magnitude faster than the base heuristic when used in greedy best-first search, and they also tend to improve solution quality substantially. In Section 5.2.6, we show how inadmissible heuristics can be used in bounded suboptimal search. We introduce a new algorithm, skeptical search, that is capable of using arbitrary inadmissible heuristics. Skeptical search improves upon the performance of the state-of-the-art optimistic search algorithm [65] while removing the need for parameter tuning. In Section 5.3.2, we show that, although heuristics learned either offline or in between search episodes are often substantially more accurate than those learned online, they provide worse guidance, leading to slower solving of instances. To close, in Section 5.4 we compare against work aimed a learning heuristics using a set of instances. Other related work is summarized in Section 5.6.

5.2 Learning During Search

Heuristic evaluation functions are the distinguishing component of heuristic search algorithms. Notated $h(n)$, these functions estimate the cost of the cheapest completion of a given node n, that is, the cost of the cheapest sequence of actions transforming the state represented by node n into a goal state. Our starting observation is that the optimal cost of a solution beneath some node p is the cost of completing its best child plus the cost of transition to that best child. More formally, let $h^*(n)$ represents the perfect heuristic function that exactly predicts the cost-to-go for all nodes. For any parent node p, if $bc(p)$ is the next node along an optimal path from p to a goal and $c(p, bc(p))$ is the cost of the transition between p and $bc(p)$, then:

$$h^*(p) = h^*(bc(p)) + c(p, bc(p)) \tag{5.1}$$

This is a slight generalization of move invariance [9], which holds that the entire node evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of arriving at node n, should not vary between a parent and its best child. Here, rather than trying to hold $f(n)$ constant across nodes, we're trying to force the heuristic to differ by exactly $c(p, bc(p))$. A little algebra shows us that these are equivalent:

$$\begin{aligned}
f^*(p) &= f^*(bc(p)) \\
g(p) + h^*(p) &= g(bc(p)) + h^*(bc(p)) \\
h^*(p) &= h^*(bc(p)) + (g(bc(p)) - g(p)) \\
h^*(p) &= h^*(bc(p)) + c(p, bc(p))
\end{aligned}$$

Obviously, during the course of search, we do not have access to perfect heuristics. If we did, search would be unnecessary. We would simply perform hillclimbing from the root, expanding only those nodes along the solution. However, every time an imperfect heuristic deviates from the relationships described above, we have observed a mistake. Every observed mistake is an opportunity to learn an improvement to the underlying heuristic function. In this way, our perspective is that of temporal difference learning [62]. Using temporal difference learning to improve heuristics has been suggested before [40, pages 172-175], but to our knowledge never actually implemented and evaluated until this work. In the next section, we present the details of our approach.

5.2.1 Single-Step Error Corrections

We can measure the error in a heuristic for a single step by comparing heuristic values between the parent and the best child. With a measurement of the error across a single step, we can attempt to correct for the error by estimating the number of steps to go and adjusting the heuristic estimates accordingly. As shown in Equation 5.1, there is a relationship between the cost-to-go estimates of a parent and its best child. This allows us to define the single-step error in h at p as:

$$\epsilon_{h_p} = (h(bc(p)) + c(p, bc(p))) - h(p) \tag{5.2}$$

The sum of the cost-to-go heuristic and the single-step errors from a node p to the goal equals the true cost-to-go:

Theorem 1 *For any node p with a goal beneath it:*

$$h^*(p) \quad = \quad h(p) + \sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n} \tag{5.3}$$

where $p \rightsquigarrow goal$ is the set of nodes along the path between the node p and the goal, including p and excluding the goal. ϵ_{h_n} is the single-step error in h between a node n and its best child.

Proof: The proof is by induction over the nodes in the path. For our base case, we show that when $bc(p)$ is the goal, Equation 5.3 holds:

$$\begin{aligned}
h^*(p) &= c(p, bc(p)) && \text{because } bc(p) \text{ is the goal} \\
&= h(p) + c(p, bc(p)) - h(p) && \text{by algebra} \\
&= h(p) + c(p, bc(p)) + h(bc(p)) - h(p) && \text{because } h(bc(p)) = 0 \\
&= h(p) + \epsilon_{h_p} && \text{by Eq. 5.2} \\
&= h(p) + \textstyle\sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n} && \text{because } p \rightsquigarrow goal = \{p\}
\end{aligned}$$

As the best child of p was a goal, the optimal cost of completing p is exactly the arc cost from p to its best child.

For the inductive case, assuming that Equation 5.3 holds for $bc(p)$, we show that it holds for its parent p as well:

$$\begin{aligned}
h^*(p) = {} & c(p, bc(p)) + h^*(bc(p)) && \text{by Eq. 5.1} \\
= {} & c(p, bc(p)) + h(bc(p)) + \textstyle\sum_{n \in bc(p) \rightsquigarrow goal} \epsilon_{h_n} && \text{by inductive assumption} \\
= {} & h(p) + \epsilon_{h_p} + \textstyle\sum_{n \in bc(p) \rightsquigarrow goal} \epsilon_{h_n} && \text{by Eq. 5.2} \\
= {} & h(p) + \textstyle\sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n} && \text{by def. of } \rightsquigarrow
\end{aligned}$$

which is exactly Equation 5.3, completing the proof. □

We define the mean one-step error $\bar{\epsilon}_h$ along the path from p to the goal as:

$$\bar{\epsilon}_{h_p} = \frac{\sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n}}{d^*(p)} \tag{5.4}$$

where $d^*(p)$ is the length of the cost-optimal path between p and a goal. It is important to remember that the mean single-step error is defined in terms of the true length (number of arcs) of the remaining path, $d^*(p)$, and not the cost (sum of weights) of the remaining path, $h^*(n)$. We will reconsider this decision in Section 5.3.1, and while both approaches can be shown to be technically correct, using path length provided better performance empirically. Solving Equation 5.4 for $\sum_{n \in p \rightsquigarrow goal} \epsilon_{hn}$ yields:

$$\sum_{n \in p \rightsquigarrow goal} \epsilon_{hn} = d^*(p) \cdot \bar{\epsilon}_{h_p} \tag{5.5}$$

Substituting Equation 5.5 into Equation 5.3 we see that:

$$h^*(p) = h(p) + d^*(p) \cdot \bar{\epsilon}_{h_p} \tag{5.6}$$

This forms the basis for the *single step* heuristic correction method.

In a realistic setting, we are not going to have access to the true distance-to-go $d^*(n)$, and so we cannot use Equation 5.6 to produce an improved cost-to-go estimate directly. Given the important role that distance plays in Equation 5.6, we will assume that a heuristic estimate of search distance-to-go, call it $d(n)$, is available (In Section 5.3.1, we will consider heuristic correction without $d(n)$). If this assumption seems strong, note that in domains in which all actions have equal cost, $d(n) = h(n)$. In other domains, one can usually construct a distance-to-go heuristic using methods very similar to those for the cost-to-go heuristic. For example, one can track the number of actions required to solve a simplified version of the problem, in addition to the cost of those actions. Further examples are given by [42], [19], and [66].

Just as we correct a given $h(n)$, we will want to correct $d(n)$. We take a similar strategy as before. In analogy to Equation 5.1, the perfect distance-to-go estimate $d^*(n)$ obeys:

$$d^*(p) = 1 + d^*(bc(p)) \tag{5.7}$$

Notice that $c(p, bc(p))$ has been replaced with 1 in the previous equation. That is because while a transition between two nodes may have a wide range of weights assigned to it, a distance estimate only cares about the number of transitions. When we are not working with perfectly informed heuristics, we must introduce a term that represents the error ϵ_{d_p} present in the heuristic when evaluated at a parent p and its best child:

$$d(p) = 1 + d(bc(p)) + \epsilon_{d_p} \tag{5.8}$$

Solving for the one-step distance error of the parent ϵ_{d_p}, we get:

$$\epsilon_{d_p} = (1 + d(bc(p))) - d(p) \tag{5.9}$$

Note that the single-step error is specific to the node p. Imagine a situation where several nodes, each with a different distance-to-go estimate, all generate the same goal node as their only child. All nodes share a best child, but each has a different single-step error. As a result, the error is specific to the generating node. We require that the best child selected for this calculation not represent the parent state of p. Thus, states with no children other than the inverse action back to their parent have no associated ϵ_d. Goals also have no best child. Using Equation 5.9, we prove the following analogue of Theorem 1:

Theorem 2 *For any node p with a goal beneath it:*

$$d^*(p) = d(p) + \sum_{n \in p \rightsquigarrow goal} \epsilon_{dn} \tag{5.10}$$

where p $\rightsquigarrow$ goal is the set of nodes along an optimal path between the node p and a goal, including p and excluding the goal.

Proof: The proof is by induction over the nodes in the path. For our base case, we show that Equation 5.10 holds when $bc(p)$ is the goal:

$$\begin{aligned}
d^*(p) &= 1 && \text{because } bc(p) \text{ is a goal} \\
&= d(p) + 1 - d(p) && \text{by algebra} \\
&= d(p) + 1 + d(bc(p)) - d(p) && \text{because } d(bc(p)) = 0 \\
&= d(p) + \epsilon_{d_p} && \text{by Equation 5.9} \\
&= d(p) + \textstyle\sum_{n \in p \rightsquigarrow goal} \epsilon_{d_n} && \text{because } p \rightsquigarrow goal = \{p\}
\end{aligned}$$

As the best child of p was a goal, p is obviously a single step away from the goal and the base case holds.

For the inductive case we show that by assuming that Equation 5.10 holds for $bc(p)$, we can show that it holds for its parent p as well:

$$\begin{aligned}
d^*(p) = {} & 1 + d^*(bc(p)) && \text{by Eq. 5.7} \\
= {} & 1 + d(bc(p)) + \textstyle\sum_{n \in bc(p) \rightsquigarrow goal} \epsilon_{d_n} && \text{by inductive assumption} \\
= {} & d(p) + \epsilon_{d_p} + \textstyle\sum_{n \in bc(p) \rightsquigarrow goal} \epsilon_{d_n} && \text{by Eq. 5.9} \\
= {} & d(p) + \textstyle\sum_{n \in p \rightsquigarrow goal} \epsilon_{d_n} && \text{by def. of } \rightsquigarrow \text{ and } bc
\end{aligned}$$

which is exactly Equation 5.10, completing the proof. □

We can define the mean one-step error $\bar{\epsilon}_{d_p}$ along the path from p to the goal as:

$$\bar{\epsilon}_{d_p} = \frac{\sum_{n \in p \rightsquigarrow goal} \epsilon_{d_n}}{d^*(p)} \tag{5.11}$$

Using Equations 5.10 and 5.11, we can define $d^*(p)$ in terms of $\bar{\epsilon}_d$:

$$d^*(p) = d(p) + d^*(p) \cdot \bar{\epsilon}_{d_p} \tag{5.12}$$

Solving Equation 5.12 for $d^*(p)$ yields:

$$d^*(p) = \frac{d(p)}{1 - \bar{\epsilon}_{d_p}} \tag{5.13}$$

Another way to think of Equation 5.13 is as the closed form of an infinite geometric series that recursively accounts for error in $d(p)$:

$$\begin{aligned} d^*(p) &= d(p) + d(p) \cdot \bar{\epsilon}_{d_p} + (d(p) \cdot \bar{\epsilon}_{d_p}) \cdot \bar{\epsilon}_{d_p} + \ldots && (5.14) \\ &= d(p) \cdot \textstyle\sum_{i=1}^{\infty} (\bar{\epsilon}_{d_p})^i && (5.15) \end{aligned}$$

This series takes the average single-step error, $\bar{\epsilon}_d$, and assumes that we will observe that error during each step that $d(n)$ is predicting. This results in some number of additional steps. Unfortunately, the mean single-step error will also be observed in the additional steps. Naturally, this results in more steps, during which the error will again be observed. This process recurs, resulting in the infinite series.

Substituting our compact equation for d^* (Equation 5.13) into our equation for h^* (Equation 5.6), we have:

$$h^*(p) = h(p) + \frac{d(p)}{1 - \bar{\epsilon}_{d_p}} \cdot \bar{\epsilon}_{h_p} \tag{5.16}$$

Given Equations 5.16 and 5.13, if we had both $\bar{\epsilon}_{d_n}$ and $\bar{\epsilon}_{h_n}$, we could construct perfect estimates of both the distance-to-go and cost-to-go beneath an arbitrary node n. The quantities $\epsilon_{\bar{d}_n}$ and $\epsilon_{\bar{h}_n}$ are the mean one-step errors along an optimal path through n to a goal in the distance-to-go and cost-to-go heuristics respectively. During a search, these values are unknown, although they are bounded. The average error can never be less than 0, and can never be larger the largest arc-cost in the case of $\epsilon_{\bar{h}_n}$ or 1 in the case of $\epsilon_{\bar{d}_n}$. The heart of our proposed method for learning during search is to estimate $\bar{\epsilon}_{h_n}$ and $\bar{\epsilon}_{d_n}$ using the observed errors described in Equations 5.9 and 5.2. We then use these estimated values to improve the performance of the cost-to-go and distance-to-go heuristics during the same search. To complete the approach, we now discuss two techniques for estimating $\epsilon_{\bar{d}_n}$ and $\epsilon_{\bar{h}_n}$ online.

Global Error Model

The *Global Error Model* assumes that the distribution of one-step errors across the entire search space is uniform and can be estimated by a global average of all observed single-step errors. We need only keep a running global sum of observed error in h and d as well as a running count of the number of observations taken. This is roughly equal to the number of expanded nodes, although some nodes may have no children and thus generate no observations. The one difficulty in employing the global error model is that we must estimate which child of node p is $bc(p)$. We assume it is the node with minimum $f(n) = g(n) + h(n)$ among all of p's children, breaking ties on $f(n)$ in favor of low $d(n)$. Pilot experiments showed this to be just as effective as using $\widehat{f}(n) = g(n) + \widehat{h}(n)$, where $\widehat{h}$ is the current corrected heuristic. We then calculate the corrected heuristics $\widehat{d}$ and $\widehat{h}$ using Equations 5.13 and 5.16 respectively:

$$\widehat{d}^{global}(n) = \frac{d(n)}{1 - \bar{\epsilon}_d^{global}} \tag{5.17}$$

$$\widehat{h}^{global}(n) = h(n) + \widehat{d}^{global}(n) \cdot \bar{\epsilon}_h^{global} \tag{5.18}$$

This approach has the benefit of gaining information on average single-step error very quickly and the drawback of the values constantly fluctuating. Our estimates of single-step error change every time we receive an observation, which is at nearly every expansion. If we really want to expand nodes in the order dictated by the cost function, this would require resorting our open list after every expansion. In most benchmark search domains, heuristic computation and node expansion are cheap enough that the cost of the search would be dominated by the cost of constantly resorting the open list. In preliminary experiments, we investigated several approaches, including constantly resorting, a logarithmic resorting schedule, and no resorting. We found that no resorting performed the best empirically and those are the results presented below.

Path-based Error Model

The *Path-based Error Model* calculates the mean one-step errors, $\bar{\epsilon}_d^{path}$ and $\bar{\epsilon}_h^{path}$, separately along each search path. This allows the model to capture variations in the heuristics' accuracy in different parts of the search space. This is done by passing the cumulative single-step error experienced by a parent node down to all of its children. We can then use the depth of the node to determine the average single-step error along this path. $\widehat{d}^{path}$ and $\widehat{h}^{path}$ and computed analogously to Equations 17 and 5.18:

$$\widehat{d}^{path}(n) = \frac{d(n)}{1 - \bar{\epsilon}_d^{path}} \tag{5.19}$$

$$\widehat{h}^{path}(n) = h(n) + \widehat{d}^{path}(n) \cdot \bar{\epsilon}_h^{path} \tag{5.20}$$

The path-based model has a distinct conceptual (and practical) advantage over the global error model: we need not estimate which node is the best child at the time that a parent node is expanded in order to compute average error. In the path-based model, we can simply say that every child of a node is the best child, as this is what the search has determined at the time of expansion. For when a node is expanded by best first search, the search (and evaluation function) have decided that this particular node, among all other nodes available for consideration, is best. If a node is best among all nodes, it must also be best among its siblings (or its siblings descendants, from which the siblings would derive their values). The practical effect of this is that we need not worry about resorting the open list, because the heuristic corrections of nodes in the path-based model never change.

In either model, if our estimate of $\bar{\epsilon}_d$ is ever as large as one, we assume we have infinite distance and cost-to-go. Because these are estimates, and not bounds, we don't discard nodes which we guess have infinite cost. This preserves the completeness of algorithms using the corrected heuristics. An alternate approach, that we do not pursue in this chapter, would be to put these nodes in a reserve list that is only considered when nodes with finite

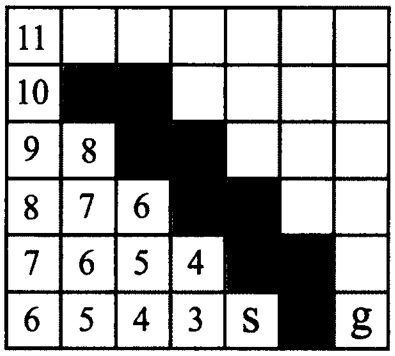


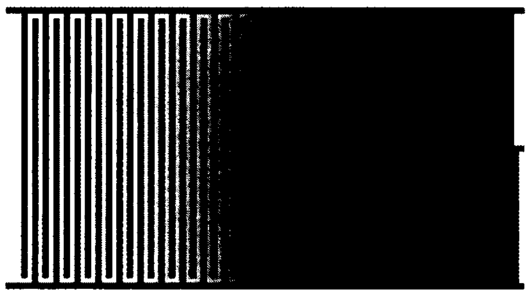
Figure 5-1: A worst-case domain for single-step corrections

estimated cost have been exhausted. The alternate list could then be sorted on another criteria, for example, the base heuristic or $g(n)$.

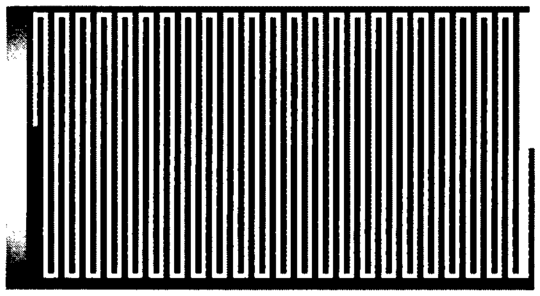
5.2.2 Worst and Best Case Scenarios

The single-step correction techniques presented above do have limitations. Figure 5-1 shows a grid pathfinding problem where single-step corrections perform poorly. The start state is marked with 's', and the goal is marked with 'g'. The grid is 4-connected. The numbers in the cells show the value of $d(n)$, the distance-to-go estimate. In this instance we use the Manhattan distance in a 4-connected grid under the free-space assumption for $d(n)$.

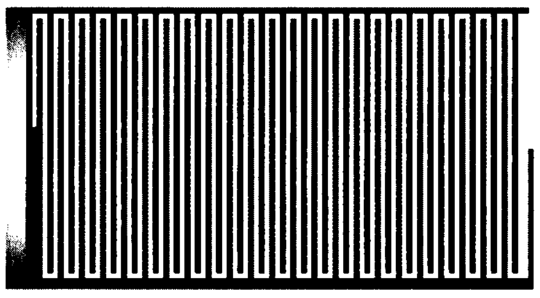
In this example, each move that could take us out of the beginning section into the half of the grid with the goal is a move that will increase the estimated distance-to-go. For any search to escape the beginning of the problem, it must experience a single-step error of two repeatedly. When we reach the state marked with a distance-to-go of eleven, the estimated single-step error will be two for both the global and path-based methods. Until the estimate is lowered below one by expanding many additional nodes with no single-step error, $\widehat{h}(n) = \widehat{d}(n) = \infty$, and our search will expand nodes in uniform cost order due to tie breaking (in the search algorithms presented here, we break ties in favor of low $g(n)$). Thus, if the cost-to-go estimates become infinitely large, we will perform a best-first-search on $g(n)$.



Greedy search



A* search



Greedy search with learning

Figure 5-2: A best-case domain for single-step corrections

If we had just been doing a greedy search on the base heuristic in this example, we would go straight to the goal from the state marked eleven rather than performing uniform-cost-search. Therefore, greedy search on the corrected heuristic will perform much worse than the uncorrected heuristic. In fact, we can make the example above arbitrarily large, and so the performance gap could be made arbitrarily large as well. Any heuristic with large plateaus or local minima between the start and a goal can demonstrate this behavior. If the plateaus and minima are larger than the areas where the heuristic performs well, we would expect to see this pathology. It should be noted that this is arguably correct, albeit undesirable, behavior. If the heuristic is woefully uninformed, or worse yet misleading, it may be preferable to ignore it entirely and search according to cost incurred.

In contrast, the images in Figure 5-2 provide an example of structured error that works strongly in favor of the single-step correction method. In this ladder-like navigation problem, the error is, as before, highly structured and there are many nodes for which the heuristic is very poorly informed (those in between the 'rungs') and nodes for which the heuristic is perfectly informed (those on the outside of the ladder). Greedy search without correction is much slower than even A* for this problem. However, when learning is added to the solving process, as it is in the bottom panel of the figure, the performance is identical in this case.

This example demonstrates two things. The first is that the corrections can work incredibly well in some domains. The second is that, in order to produce the poor behavior noted in Figure 5-1, the heuristic must be incorrect early on *for all nodes leading to a reasonable goal.* It is not enough for the heuristic to merely be very incorrect early.

In the eight benchmark domains considered in the evaluation below, we observed neither of the behaviors present in these hand-crafted examples. This suggests that it is often the case that the heuristic is neither consistently misinformed, nor is it perfectly informed. This is to be expected, as heuristics are generally heavily engineered functions designed to work well in practice.

5.2.3 Performance of Single-Step Corrections

We will consider two ways of evaluating the quality of our learned heuristics. First, we look at how accurately they predict the true cost-to-go. We then consider their success in guiding a heuristic search algorithm towards a goal.

Absolute Accuracy

For the accuracy study, we consider three small benchmark domains:

Sliding Tiles Puzzles We examined 100 random 8-puzzle instances. In our implementation, the goal state has the blank in the upper-left, with the numeric tiles laid out in sequence left to right, top to bottom. All actions have unit cost. We do not consider moving back to the parent node's state, so very few duplicate states are encountered during search. Manhattan distance is used to estimate the cost and distance-to-go for all states.

Grid-world Navigation We tested on grid pathfinding problems using the "life" cost function. This cost function produces problems where actions have a large range of costs, short solutions are more costly than longer ones, and the search space includes several large g-value plateaus. These properties have recently seen significant interest [2, 75]. We examined 200 by 200 grids with 35% of cells blocked randomly. The cost function means that standard heuristics like Manhattan distance are no longer an accurate (or even admissible) estimate of cost-to-go for these grid problems. To compute a heuristic for these problems, we assume that there are no obstacles and analytically compute the cheapest solution from a node to the goal.

Vacuum World In this domain, which follows the first state space presented by Russell and Norvig[53], a robot is charged with cleaning up a grid world. Movement is in the cardinal directions, and when the robot is on top of a pile of dirt, it may vacuum. The cost of movement is one plus the number of dirt piles that have already been vacuumed up. Cleaning has unit cost. We used 100 instances that are 200 by 200 with 5 piles of dirt and 35% of cells blocked randomly. An admissible cost-to-go heuristic is found by computing the spanning tree of all dirty cells and the robot. The edges in the spanning tree are then weighted, with the longest edge receiving the current robot weight, the next longest the robot weight plus one, and so on. The length of the solution is estimated inadmissibly by making a free space assumption and computing a greedy traversal of the dirty cells.

In each domain, we examined the following single-step correction techniques:

SS Path The path-based corrections based on single-step error computed as in Equation 5.20.

SS Global The global corrections based on single-step error computed as in Equation 5.18. The best child of a node is computed using $f(n)$ rather than the improved estimate $\widehat{f}(n)$ as mentioned previously.

All algorithms were implemented in Objective Caml, compiled to 64-bit native code, and run on Linux systems with 3.16 GHz Intel Core2 duo processors and 8 GB of RAM. All of the algorithms share the same domain functions and data structures to help ensure fair comparisons.

Figure 5-3 shows the performance of the learned heuristics relative to truth on our small benchmark domains. The y-axis represents error, computed as $h^*(n) - \widehat{h}(n)$ where $\widehat{h}$ is the heuristic labeled on the x-axis. We present the data in the form of a box plot. The whiskers extend to the extreme values. The box shows data between the first and third quartile, and the line in the box shows the median value. The gray rectangle shows 95% confidence intervals about the mean. The intervals are so tight for most of the plots this rectangle will appear as a short line. Occasionally this line overlaps with the median, and can not be seen.

In all three plots, we see that the baseline, the admissible heuristic has all of its error above zero because it is required to underestimate the true cost-to-go. It is also relatively accurate when compared to the two learned heuristics. The extreme values for the admissible

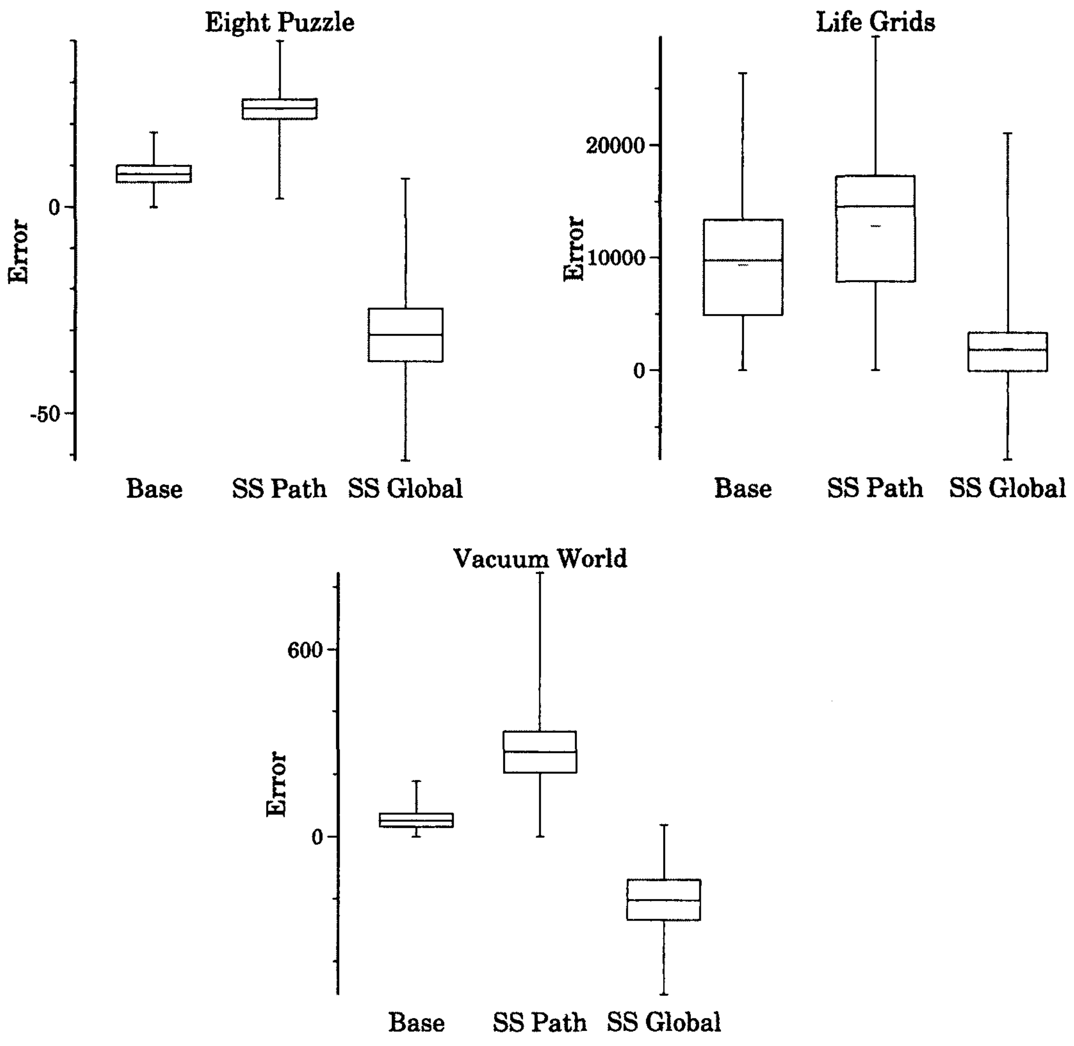


Figure 5-3: Accuracy of single-step corrections on the 8-puzzle, "life" grids, and vacuum world

heuristic are always smaller than that of the learned heuristics. Further, the total range of values is also always smaller than that for the learned heuristics. In the eight puzzle and vacuum world, the base heuristic is the most accurate, it has a mean error closer to zero than any of the other heuristics being considered.

In all three domains the path based correction has worse performance, in terms of error, than the base heuristic it is attempting to correct as it has median and mean values further away from 0 error and more extreme error values.

Given the performance of these heuristics relative to truth, we might expect a search algorithm guided by global corrections to perform best in life grids, while the base heuristic would perform best in the eight puzzle and in vacuum worlds.

Guidance

We now turn from the absolute accuracy and evaluate the performance of these heuristics inside of search algorithms. While absolute accuracy may give us some indications as to how a heuristic will perform inside of a search algorithm, it doesn't tell the whole story, and this is one of the most common misconceptions in heuristic search [26]. We will see that, surprisingly, path-based corrections provide superior guidance despite being less accurate in absolute terms. We delay our evaluation of heuristics in bounded suboptimal search until Section 5.2.6 so that we can evaluate the guidance of the heuristics alone before examining their interaction with the admissible heuristics which are needed to provide guarantees on solution quality.

Greedy search [15] is a best-first heuristic search where best is determined solely by the heuristic. While this estimate may be admissible, greedy search can provide no guarantees on the quality of the solutions it returns, so there is no need to limit the heuristic by restricting it to be admissible. For the guidance study, we use four additional benchmark domains. They were omitted from the accuracy study because we cannot measure accuracy for all states as the search spaces are too large to be enumerated on our machines.

Fifteen Puzzle We examined the 100 instances of the 15-puzzle presented by [32]. We use the Manhattan distance heuristic for both $h(n)$ and $d(n)$, just as we did in the 8-puzzle.

Dynamic Robot Following [35], the goal is to find the fastest path from the initial state of the robot to some goal location and heading, taking momentum into account. We use worlds that are 200 by 200 cells in size. We scatter 25 lines, up to 70 cells in length, with random orientations across the domain and present results averaged over 100 instances. We precompute the shortest path from the goal to all states, ignoring dynamics. To compute h, we take the length of the shortest path from a node to a goal and divide it by the maximum velocity of the robot. For d, we use the number of actions along that path.

Dock Robot We implemented a dock robot domain inspired by the running example of [20] and the depots domain from the International Planning Competition. Here, a robot must move containers to their desired locations. Containers are stacked at a location using a crane, and only the topmost container on a pile may be accessed at any time. The robot may drive between locations and load or unload itself using the crane at the location. We tested on 150 randomly configured problems having three locations laid out on a unit square and ten containers with random start and goal configurations. Driving between the depots has a cost of the distance between them, loading and unloading the robot costs 0.1, and the cost of using the crane was 0.05 times the height of the stack of containers at the depot. h was computed as the cost of driving between all depots with containers that did not belong to them in the goal configuration plus the cost of moving the deepest out of place container in the stack to the robot. d was computed similarly, but 1 is used rather than the actual costs.

Vacuums This differs from the accuracy study in that now there are 10 piles of dirt to remove instead of 5. The size of the state space is exponential in the number of dirt piles, so these problems are considerably more difficult than the previous ones. h and d are computed as before.

	Eight Puzzle		Life Grids		Small Vacuums	
	generated	cost	$\frac{sec}{1000}$	$\frac{cost}{1000}$	secs	cost
Baseline	582	*128*	*169*	2993	*0.990*	*2673*
SS Global	*763*	43	74	*3050*	0.405	2457
SS Path	**463**	**33**	**71**	**2795**	**0.260**	**2100**

Table 5-1: Performance of single-step corrections in greedy search on domains from accuracy study

Table 5-1 presents the results of using the learned heuristics within a greedy best-first search for the domains used in the accuracy study. Algorithms are run until a solution is found, memory is exhausted, or 10 minutes have passed. We report the mean CPU time required to find a solution, except for the eight puzzle where we report nodes generated because the times are extremely small, and the mean cost of that solution. The worst entry in a column is *italicized*, and the best value in each column is **bolded**. The table reveals that the more accurate predictors do not always lead to improved performance within a search algorithm. If they did, the global corrections, which were often more accurate than the path-based single-step approach, would have better performance. We see that, despite its relatively poor accuracy, path-based corrections provide the best performance in terms of both solving time and solution cost in a greedy search on these three small benchmarks. Further, global correction, which was more accurate than the base heuristic in two domains, often provides worse performance in terms of solving time.

We show results on the more difficult problems in Table 5-2. These problems are difficult enough that not all heuristics can guide greedy search to a solution using the machines we had at our disposal. When an algorithm fails to find a solution within system memory or within 10 minutes, we say that it failed. So, for more difficult instances, the cost column either reports the mean solution cost or the number of instances the algorithm failed to solve. Seconds is mean elapsed time for all instances, regardless of why the algorithm halted (i.e.

	Fifteen Puzzle		Dynamic Robot		Dock Robot		Large Vacuums	
	$\frac{secs}{1000}$	cost	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Baseline	29	*302*	60	522	*169*	*Failed 55*	*9.07*	*9635*
SS Global	*177*	136	*563*	*1321*	77.2	Failed 24	3.56	6808
SS Path	**15**	**90**	**14**	**47**	**0.38**	**29**	**1.22**	**6063**

Table 5-2: Performance of single-step corrections when used in greedy search on larger problems

15 Puzzle

Heuristic	$\frac{secs}{1000}$	cost
Manhattan Distance	29	302
7-8 PDB	*44*	85
Manhattan Distance SS Path	15	90
7-8 PDB SS Path	**13**	**65**

Table 5-3: Performance of learned heuristics compared to that of pattern databases

timeouts score 600 seconds, memory exhaustion as long as it takes to exhaust memory, and so on). We see that path-based single-step corrections provide the best guidance, holds for these larger and more diverse benchmarks. The dockyard robot domain is particularly interesting. Here, the single-step path corrections solve more instances than the other approaches. By observing the performance of the heuristic on a single instance we can solve problems that we could not solve with the base heuristic alone.

5.2.4 Impact of Base Heuristic Accuracy

One might wonder if these observed improvements are limited to relatively weak heuristics like Manhattan Distance. In Table 3 we compare our best learning method with a modern pattern database for the 15-puzzle, the 7-8 PDB [31]. The 7-8 PDB is the sum of PDB heuristics that have been computed such that they can be added together without becoming inadmissible. Rather than computing the distance of every tile from its goal location, a PDB heuristic works by enumerating the state space for a relaxed version of the problem, in this case one where all of the tiles other than 1 through 7 have no symbol on them. The space is enumerated using all of the actions available in the real problem, and the cost of reaching a state from the goal is recorded. During search, we then abstract the state we are examining into the pattern used in the pattern database, that is we imagine all of the tiles other than 1-7 have been wiped clean, and then ask the PDB how expensive our current configuration is. In the case of the 7-8 PDB, this abstraction and lookup is done twice, and then the values are summed up to provide an estimate of cost-to-go.

We see that using the pattern database heuristic has mixed results with respect to greedy search performance, solving times are longer (although fewer nodes are generated), and solution cost is reduced. However, our path-based heuristic finds solutions faster than the PDB heuristic and those solutions are not much worse on average, and on some instances our heuristic can find better solutions. This is accomplished without the benefit of the pre-computation needed to construct the pattern databases. If we add our path-based correction to the PDB heuristic (the last line of Table 5-3), it further improves performance, finding better solutions faster than ether the PDB alone or path-based corrections on top of Manhattan distance. From this we conclude that single-step correction can improve the performance of even strong heuristics.

5.2.5 Instance Specific Heuristics

One advantage of online corrections is that they do not require the use of a set of training instances. This means we can avoid the problem of ensuring that our training instances are similar enough to our test instances for the learning to generalize. Since all of our learning is being performed online during the solving of a single instance, we needn't worry about generalization. However, we might wonder if the information being learned during the search is specific to one instance, or if it can be used to seed the estimated error values

	Vacuums		Life Grids	
Learning	$\frac{nodes}{1000}$	$\frac{cost}{1000}$	$\frac{nodes}{1000}$	$\frac{cost}{1000}$
Base	206	10	115	2993
SS Global	62	7	42	3049
Same Instance	48	7	36	2992
Random Instance	64	7	36	2983

Table 5-4: The learning is instance specific

for searches on other instances in the same domain.

Table 5-4 shows the performance of our global single-step model used in greedy search in two new ways. The first row of the table shows the performance of the base heuristic and the second row of the table shows the performance of the global single-step model learned on line. The third line, "Same Instance" shows the performance of the global single-step model values for error learned by the global model on the same instance of the problem being solved but now being used statically (with learning turned off). "Random Instance" is similar, but the learned values come from a random instance. We use the global model because it is clear how to transfer the information learned from one instance to another: we simply take the final values we computed for $\bar{\epsilon}_h$ and $\bar{\epsilon}_d$ and use those as the average error in a new problem. In this table, we present results in terms of nodes generated in order to focus on search guidance and ignore the overhead of learning (Table 5-2 already demonstrated that using online learning can improve the speed of search algorithms). We present two domains, the vacuum domain, where learned heuristic errors are very different between instances, and life grids, where learned error is similar between instances.

As we saw before in Table 5-2, the online corrections produce better results than the base heuristic. Additionally, for both domains, using the errors learned previously for the same instance improves performance substantially. This shows us that the improved performance is not because of some fortuitous synergy between learning and search. If it were, the online model would out-perform the same errors fed into a static model. As it does not, we conclude that we are learning a meaningful ordering over the nodes.

We see that the heuristic learned from the same instance performs better than one from a random instance in the vacuum domain. This indicates that the technique is learning an instance-specific model online, and that instance-specific information is beneficial to our searches. Interestingly this is not the case for the vacuum problem. Recall that for our life grid instances, the start and goal state are always in the same location, and the obstacles are placed down uniformly at random. This suggests that the error in the heuristic is likely to be similar between any two random instances, and thus the learning should generalize well from one instance to another.

5.2.6 Bounded Suboptimal Search

So far we have seen how on-line learning can improve greedy best-first search, we now turn to the setting of bounded suboptimal search. Here, we require solutions whose quality is within a fixed factor of optimal. This will be the main focus of Chapter 7, but we introduce some bounded suboptimal search algorithms here to show that inadmissible heuristics are useful in many applications.

Bounded suboptimal search algorithms like weighted A* [44] rely on the admissibility of their base heuristic to obtain their suboptimality bound. However, some algorithms such as optimistic search [65] can use arbitrary heuristics for at least a portion of their search. Optimistic search works by running weighted A* with a weight higher than the desired suboptimality bound. This can be hand tuned per problem or per domain, although we found that a weight twice as large as the desired bound worked well in the domains they evaluated the algorithm in. However, looking closely at the algorithm will reveal that optimistic search can take advantage of any inadmissible heuristic. After finding an incumbent, additional nodes are expanded in A* order until we can prove the solution found was within the desired suboptimality bound.

Optimistic search proves that the incumbent is within the bound by comparing its cost

```
OptimisticSearch($root, b, w$)
1.  $incumbent \leftarrow$ null
2.  $open \leftarrow \{root\}$
3.  while($incumbent =$ null and $open \neq \{\}$)
4.    remove $n$ from $open$ with minimum $f'(n) = g(n) + w \cdot h(n)$
5.    if $n$ is a goal
6.      $incumbent \leftarrow n$
7.      otherwise, expand $n$ and insert children into $open$
8.  while($open \neq \{\}$)
9.    $f_{min} \leftarrow n \in open$ with minimum $f(n) = g(n) + h(n)$
10.   $f'_{min} \leftarrow n \in open$ with minimum $f'(n) = g(n) + w \cdot h(n)$
11.   if $b \cdot f(f_{min}) \geq g(incumbent)$
12.     return $incumbent$
13.   otherwise, if $f'(f'_{min}) \leq g(incumbent)$
14.     if $f'_{min}$ is a goal
15.       $incumbent \leftarrow min(f'_{min}, incumbent)$
16.       otherwise, remove $f'_{min}$ from $open$, expand it, and insert its children.
17.   otherwise, remove $f_{min}$ from $open$, expand it and insert children into $open$
18. return $incumbent$
```

Figure 5-4: Optimistic Search pseudo code with escape hatch

to $f(f_{min})$, the estimated cost of the node with the smallest f value. The f value of a node acts as a lower bound on the cost of a solution through that node, so the f value of the node with the smallest f value acts as a lower bound on the cost of an optimal solution to a problem. Therefor, if $f(f_{min})$ is within a factor b of the cost of the incumbent solution, we know that the incumbents quality is within a bounded factor of the cost of an optimal solution.

Pseudo code for the algorithm is provided in Figure 5-4. In lines 3 through 7, weighted A* using a weight w (presumably higher than the bound b) is used to find an initial solution. The remainder of the code is focused on proving that the incumbent is within the desired suboptimality bound (lines 11, 12, and 17) or opportunistically improving the quality of the incumbent solution. In lines 11 and 12, we test to see if the incumbent solution can be shown to be within the current bound, and if it is, then we return it. In line 17, we remove f_{min} from open and expand it. This may raise the lower bound on the cost of an optimal solution to the problem, allowing us to return the current incumbent in the next iteration. Lines 13–16 seek to improve the current incumbent solution. If it ever appears that a node might lead to a better incumbent solution, it is pursued. In practice, this case is rarely, if ever, used. For a node to be expanded by this case, it must first be generated by an f_{min} expansion, otherwise it would have been expanded before an incumbent was found in lines 1–7. This can happen if w and b are selected such that the solution initially found is outside of the bound. In practice, we prove the quality of a solution long before such a node becomes a candidate for expansion in line 13.

When searching for an incumbent solution, optimistic search can use any inadmissible heuristic and still retain its guarantees of bounded suboptimality as long as an admissible heuristic is available for proving that the incumbent was within the desired bound. While, at first glance, it may not be obvious that optimistic search is using an inadmissible heuristic, we can show that it is by closely examining line 4. Rather than writing $f'(n) = g(n)+w\cdot h(n)$, we could instead write $f'(n) = g(n)+b\cdot\frac{w}{b}\cdot h(n)$. We can think of $\frac{w}{b}\cdot h(n)$ as an inadmissible heuristic which attempts to correct for the under-estimating nature of $h(n)$ by scaling it

```
SkepticalSearch($root, w$)
1.  $incumbent \leftarrow$ null
2.  $open \leftarrow \{root\}$
3.  while($incumbent =$ null and $open \neq \{\}$)
4.     remove $n$ from $open$ with minimum $\widehat{f'}(n) = g(n) + w \cdot \widehat{h}(n)$
5.     if $n$ is a goal
6.        $incumbent \leftarrow n$
7.        otherwise, expand $n$ and insert children into $open$
8.  while($open \neq \{\}$)
9.     $f_{min} \leftarrow n \in open$ with minimum $f(n) = g(n) + h(n)$
10.    $\widehat{f'}_{min} \leftarrow n \in open$ with minimum $\widehat{f'}(n) = g(n) + w \cdot \widehat{h}(n)$
11.    if $w \cdot f(f_{min}) \geq g(incumbent)$
12.       return $incumbent$
13.    otherwise, if $\widehat{f'}(\widehat{f'}_{min}) \leq g(incumbent)$
14.       if $\widehat{f'}_{min}$ is a goal
15.          $incumbent \leftarrow min(\widehat{f'}_{min}, incumbent)$
16.          otherwise, remove $\widehat{f'}_{min}$ from $open$, expand it, and insert its children.
17.    otherwise, remove $f_{min}$ from $open$, expand it and insert children into $open$
18. return $incumbent$
```

Figure 5-5: Skeptical search pseudo code

up uniformly (recall that $w > b$). We can replace the weighted admissible heuristic from the first phase of optimistic search with any learned heuristic. We call this modification of optimistic search *skeptical search*, and we provide pseudo code for it in Figure 5-5. It is skeptical in that it does not place absolute trust in the base heuristic. Note that the ad hoc additional weight parameter of optimistic search has been removed, and so skeptical only accepts two parameters instead of three. As we will see in the following evaluation, skeptical search offers two benefits over optimistic search. It removes the need for parameter tuning and provides improved performance in several benchmark domains.

The pseudo-code makes no attempt to specially handle duplicate states, that is states re-encountered by a cheaper path. Avoiding re-expanding duplicate states often improves the performance of weighted A* [37, 71]. If the heuristic being used is consistent, dropping duplicates has no impact on the suboptimality bound. (If the heuristic is inconsistent, dropping duplicates forces us to loosen the suboptimality bound dramatically, see [16] for details.) In skeptical search, we cannot drop duplicates entirely. They must be retained so that $f(f_{min})$ provides an accurate lower bound on optimal solution cost. At best, we can choose to delay duplicates during the first iteration of skeptical search, when we are looking for a potential solutions. This leads us to find potential solutions faster, but they tend to be of lower quality. This makes the step of proving solution quality take longer. Preliminary experiments showed that delaying duplicate expansions until the cleanup phase provided better performance, and this is the approach taken in the results reported here.

Figures 5-6, 5-7, 5-8, and 5-9 compare several optimism settings, the factor by which w exceeds b, for the original optimistic search [65], weighted A* and skeptical search. The x-axis of the plot is the suboptimality bound, the desired guarantee on solution quality. The y-axis represents the amount of time needed to solve problems for the given bound. We show results for skeptical with path-based correction as it produced the best results.

Although many of the algorithms are often difficult to distinguish in detail, what is clear is that skeptical search is always at least competitive with optimistic search for any of the optimism settings examined. On the fifteen puzzle (Figure 5-6), and dynamic robot

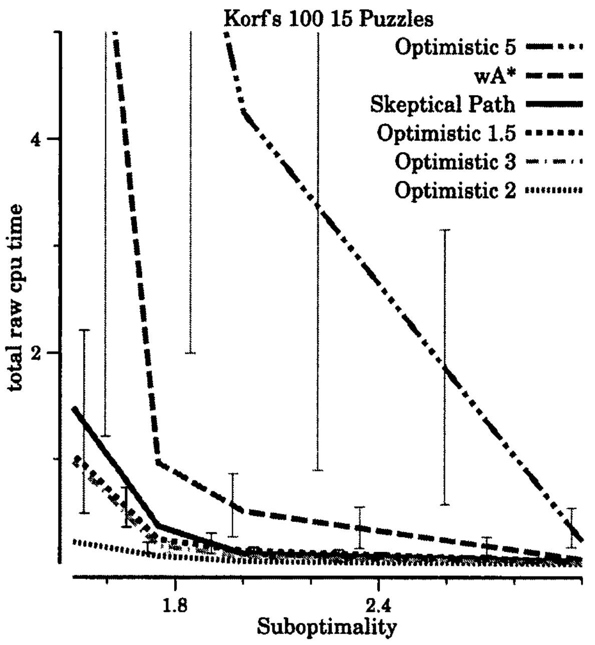


Figure 5-6: 15 puzzle

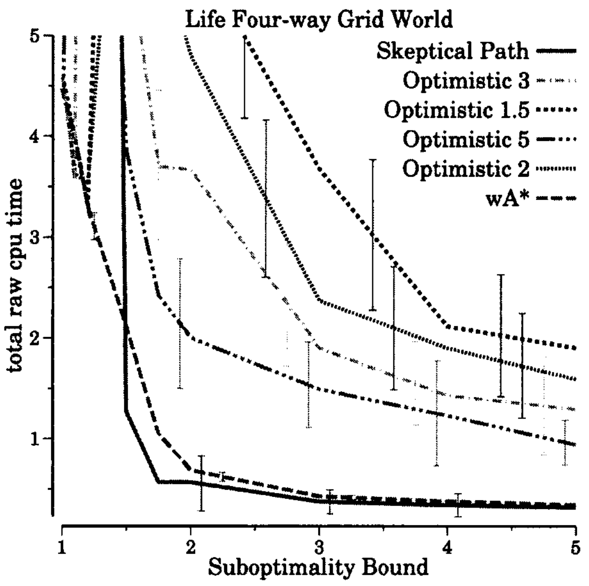


Figure 5-7: life grid navigation

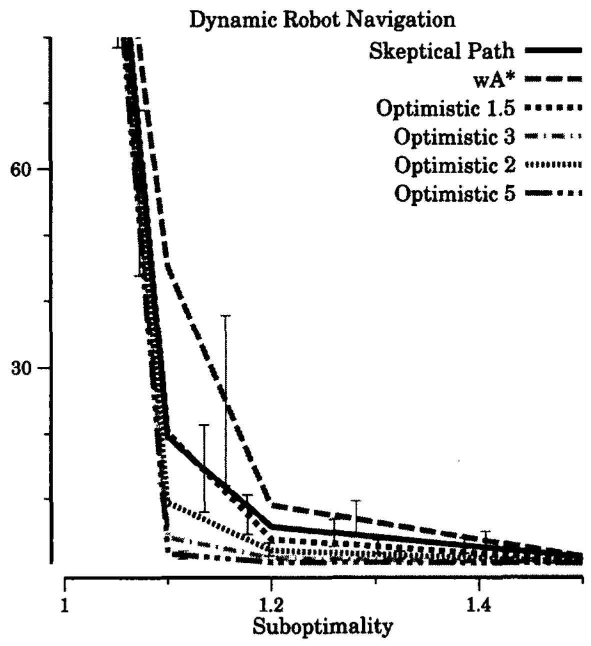


Figure 5-8: Dynamic robot navigation

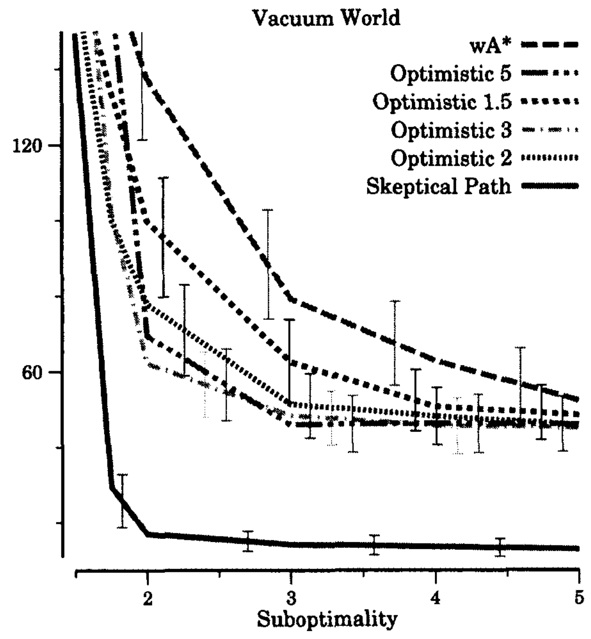


Figure 5-9: Vacuum problems

navigation (Figure 5-8) because the confidence intervals on the search time between skeptical search and the best configuration for optimistic search overlap. For life cost grids (Figure 5-7), we see that skeptical search takes between half and a third of the time needed by any optimistic search and and it is three times faster in vacuum world (Figure 5-9).

In addition to out-performing optimistic search, skeptical search removes the need for parameter tuning. Optimistic search requires two parameters, the desired suboptimality bound and an optimism factor. The optimism factor tells optimistic search how aggressive it should be in pursuing the initial solution. If it is set too high, the incumbent solution will be outside of the desired bound, and the performance of the algorithm will suffer. If it is set too low, finding the initial solution will take too long, pulling down overall algorithm performance. Skeptical search has only the desired suboptimality bound as a parameter. Rather than requiring an explicit optimism factor, skeptical search constructs $\widehat{h}$ using its experience during problem solving. It's best suited to domains where expanding nodes and computing heuristics is relatively inexpensive. If computing heuristics and generating successors are very expensive, more complicated techniques like explicit estimation search [68] are more appropriate. Of the domains presented here, explicit estimation search only outperforms skeptical search in vacuum world.

5.2.7 Summary

As we have just seen, our approach to learning heuristic corrections online during the solving of a single instance produces heuristics with strong guidance and poor overall accuracy. We saw that the strong guidance led to good performance in both suboptimal and bounded suboptimal search, improving substantially on the performance of the base heuristics. Tests of transfer provided evidence that on-line learning learns something specific to the instance being solved. This may be particularly useful when the instances of interest have substantially different properties despite being from the same domain.

Finally, we should note that we make no assumptions about the characteristics of the heuristics used as the basis for learning, This allows our technique to be as general as possible. The equations showing that the learning of single-step corrections is theoretically sound rely on only two assumptions about the basic nature of the underlying heuristics: $h(n)$ estimates the cost-to-go from n to a goal, and $d(n)$ estimates the number of actions in that solution. We did not make, nor do we need make, any assumption as to the consistency, admissibility, or accuracy of the underlying heuristic.

5.3 Alternate Approaches to Learning During Search

The single-step corrections presented in the previous section are not the only way that we can learn improved heuristics on-line. This section of the chapter focuses on alternative approaches that can be used on-line and while similarly justified and natural, do not appear to work as well in practice, as we will see in the accompanying evaluations.

5.3.1 Single-step Correction Without Distance Estimates

We might naturally wonder how much the distance-to-go heuristic $d(n)$ is contributing to the singles-step correction process. To evaluate this we altered the single-step error model to use only cost-to-go estimates, removing the need for distance-to-go estimates entirely. Rather than measuring the error in $h(n)$ per-step, we measure it per-cost:

$$\epsilon_{h_p}^{cost} = \frac{(h(bc(p)) + c(p, bc(p))) - h(p)}{c(p, bc(p))} \tag{5.21}$$

This can also be rewritten using Equation 5.2:

$$\epsilon_{h_p}^{cost} = \frac{\epsilon_{h_p}}{c(p, bc(p))} \tag{5.22}$$

Then, we compute the mean cost-step error at p as:

$$\bar{\epsilon}_{h_p}^{cost} = \frac{\sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n}^{cost}}{h^*(p)} \tag{5.23}$$

We then compute the corrected heuristic as:

$$\widehat{h}^{cost}(n) = \frac{h(n)}{1 - \bar{\epsilon}_{h_n}^{cost}} \tag{5.24}$$

using, as we did in Equations 5.18 and 5.20, either a path-based or global average to estimate $\bar{\epsilon}_{h_p}^{cost}$. The following proof shows that this is a legitimate correction:

Theorem 3 *For any node p with a goal beneath it:*

$$h^*(p) \quad = \quad h(p) + h^*(p) \cdot \bar{\epsilon}_{h_p}^{cost} \tag{5.25}$$

where $\bar{\epsilon}_{h_p}^{cost}$ is the average per-cost error in the cost-to-go estimate $h(p)$.

Proof: The proof is by induction over the nodes in $p \rightsquigarrow goal$, the optimal path from p to a goal node. For our base case, we show that when $bc(p)$ is a goal, Equation 5.25 holds:

$$\begin{aligned}
h^*(p) &= c(p, bc(p)) && \text{because } bc(p) \text{ is the goal} \\
&= h(p) + c(p, bc(p)) - h(p) && \text{by algebra} \\
&= h(p) + c(p, bc(p)) \cdot \tfrac{c(p,bc(p))) - h(p)}{c(p,bc(p))} && \text{by algebra} \\
&= h(p) + c(p, bc(p)) \cdot \tfrac{(h(bc(p)) + c(p,bc(p))) - h(p)}{c(p,bc(p))} && \text{h(bc(p)) = 0} \\
&= h(p) + c(p, bc(p)) \cdot \bar{\epsilon}_{h_p}^{cost} && \text{by Equation 5.21} \\
&= h(p) + h^*(p) \cdot \bar{\epsilon}_{h_p}^{cost} && \text{because } bc(p) \text{ is the goal}
\end{aligned}$$

For the inductive case we show that, assuming that Equation 5.25 holds for $bc(p)$, we can show that it holds for its parent p as well:

$$\begin{aligned}
h^*(p) &= c(p, bc(p)) + h^*(bc(p)) && \text{by Equation 5.1} \\
&= c(p, bc(p)) + h(bc(p)) + h^*(bc(p)) \cdot \bar{\epsilon}_{h_{bc(p)}}^{cost} && \text{by inductive assumption} \\
&= h(p) + \epsilon_{h_p} + h^*(bc(p)) \cdot \bar{\epsilon}_{h_{bc(p)}}^{cost} && \text{by Equation 5.2} \\
&= h(p) + \epsilon_{h_p} + \textstyle\sum_{n \in bc(p) \rightsquigarrow goal} \epsilon_{h_n}^{cost} && \text{by Equation 5.23} \\
&= h(p) + \textstyle\sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n}^{cost} && \text{by definition of } \rightsquigarrow \\
&= h(p) + h^*(p) \cdot \tfrac{\sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n}^{cost}}{h^*(p)} && \text{by algebra} \\
&= h(p) + h^*(p) \cdot \bar{\epsilon}_{h_p}^{cost} && \text{by Equation 5.23}
\end{aligned}$$

□

Solving Equation 5.25 for $h^*(p)$, we can arrive at something nearly identical to Equation 5.24. The difference is that here we have the exact single-step error, and in Equation 5.24 single-step error is being estimated.

$$\begin{aligned}
h^*(p) &= h(p) + h^*(p) \cdot \bar{\epsilon}_h^{cost} \\
h^*(p) - h^*(p) \cdot \bar{\epsilon}_h^{cost} &= h(p) \\
h^*(p) \cdot (1 - \bar{\epsilon}_h^{cost}) &= h(p) \\
h^*(p) &= \frac{h(p)}{(1 - \bar{\epsilon}_h^{cost})}
\end{aligned}$$

As with the single-step model, there are many ways we could choose to aggregate the observed error in the heuristic. In this work we evaluate two:

Cost Global Computes $\widehat{h}$ based on the cost-based error in $h(n)$, computed as in Equation 5.24 using a global average to estimate the error in h. The best-child is estimated as in the global model.

Cost Path Computes $\widehat{h}$ based on the cost-based error in $h(n)$, computed as in Equation 5.24. Error in the cost-to-go heuristic is aggregated along paths as in the previous path-based model.

We now evaluate the cost-step model. This will allow us to see the influence of distance estimates on our single-step corrections.

Accuracy

Figure 5-10 shows the absolute accuracy of the cost-step models on "life" grid navigation and small vacuum problems. The eight-puzzle is omitted because it has unit cost, and the cost-step models are identical to the single-step models for such domains. Additionally, the cost-based global model is omitted from Figure 5-10 as it occasionally estimated the heuristic to be infinitely large. The figure shows that, like the single-step approach to learning, the heuristics constructed online using cost-step error are less accurate than the base heuristic that they are being built from. As we saw in the previously presented distance based corrections, the global model appears to be less accurate in general than the path based corrections.

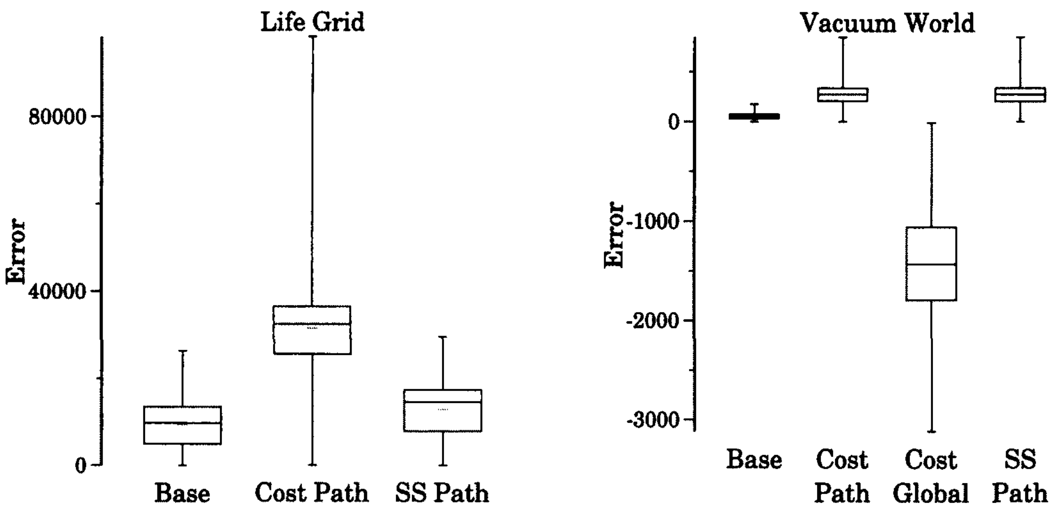


Figure 5-10: Accuracy of cost-step model on "life" grids (left) and small vacuum problems (right)

	Life Grids		Small Vacuums	
	$\frac{sec}{1000}$	$\frac{cost}{1000}$	secs	cost
Baseline	169	2993	0.990	2673
Cost-Global	*5140*	*9846*	1.042	*2786*
Cost-Path	2509	3246	*1.725*	1910
SS Path	**71**	**2795**	**0.260**	**2100**

Table 5-5: Performance in greedy search on domains from accuracy study

Guidance

Table 5-5 shows the performance of the cost-step heuristics in a greedy best-first search on the domains used in the accuracy study. While we might expect that, like the single-step models, cost-step heuristics would provide better guidance than the baseline, the experiments reveal that they do not. We see that, for these domains, both cost-based approaches are worse than the base heuristic in terms of time and solution cost.

Table 5-6 shows the performance of the cost-step heuristics on larger benchmark prob-

	Dynamic Robot		Dock Robot		Vacuums	
	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Baseline	60	522	169	Failed 55	*9.07*	9635
Cost-Global	*600000*	*Failed 40*	*349*	*Failed 73*	1.75	Failed 1
Cost-Path	**14**	**46**	11.8	Failed 2	1.94	*Failed 22*
Path	**14**	47	**0.385**	**29**	**1.22**	**6063**

Table 5-6: Comparing cost-step corrections to single-step corrections on larger problems

lems. We see that although the global version of the cost-step approach is consistently worse than the base heuristic, the path-based approach often makes substantial improvements, solving problems faster and providing solutions of lower cost. The single-step path-based heuristic is still substantially better in that it is never slower and it never failed to solve one of our benchmark instances. From this we can conclude that using the distance-to-go estimate $d(n)$ is important to the good performance of our corrected heuristics.

5.3.2 Comparison to Generic Regression Algorithms

The techniques we have considered so far were derived specifically for learning heuristic values on-line. We also evaluated the use of generic linear regression techniques. These can be applied, noting that if our corrected heuristics were perfect, we would see that the estimated cost of the parent, $\widehat{f}$, was exactly that of the estimated cost of the best child. If we were computing the corrected heuristic as a feature vector $\phi(n)$ weighted by a vector $\vec{w}$, we could expand this equation to be:

$$\begin{aligned}
\widehat{f}(p) &= \widehat{f}(bc(p)) \\
g(p) + \widehat{h}(p) &= g(bc(p)) + \widehat{h}(bc(p)) \\
\widehat{h}(p) &= g(bc(p)) - g(p) + \widehat{h}(bc(p)) \\
\widehat{h}(p) &= \widehat{h}(bc(p)) + c(p, bc(p)) \\
\widehat{h}(p) - \widehat{h}(bc(p)) &= c(p, bc(p)) \\
(\phi(\vec{p}) - \phi(b\vec{c}(p))) \times \vec{w} &= c(p, bc(p))
\end{aligned}$$

This shows that, so long as we can determine which node is the best child, we can use linear regression to compute an improved estimate of cost-to-go. To do this, we use the difference of a set of features between a parent and its best child and learn a function from them onto the cost of the transition between them. The same function for estimating the cost of the transition from the differences in features will be an estimator of the full cost-to-go from any node, as shown by the above algebra.

Unfortunately, this does not work for all regression algorithms. If the learned function is not a linear combination of the features, then we cannot perform the transformation in between Equation 5.26 and Equation 5.26. We can still use regression techniques in these situations, so long as we are willing to assume that the heuristic values of nodes deeper in the search tree are more likely to be accurate than that of nodes higher in the tree. Equation 5.1 suggests that we can approximate $h^*(p)$ as $h(bc(p)) + c(p, bc(p))$. It may be reasonable to assume that the heuristic of the child has a more accurate heuristic because the best child is one step closer to a goal, and therefor has less to be uncertain about. What this effectively provides us is a target value for standard regression techniques that can be used during the search itself. For all nodes (save the root), we can collect a set of features of the parent and then train them to estimate the heuristic of the best child plus the cost of arriving at that child, which should be more accurate than the original heuristic.

To provide a fair comparison with our previous techniques, we use only the following four features for learning:

$g(n)$ the cost of arriving at n from the root

$h(n)$ an estimate of the cost-to-go from n to a goal along a cost-optimal path from n to the goal

$depth(n)$ the number of actions between the root and n

$d(n)$ an estimate of the number of actions along a cost-optimal path from n to the goal

We take care to try to normalize the features between 0 and 1 based on an estimate of their range (using the h and d values of the root), as this typically improves the performance of learning. We cannot always normalize the values between 0 and 1 because we do not always know what the maximum value for a feature is a priori. For example the maximum $depth(n)$ and $g(n)$ are tied to the execution of search and thus unknowable. We evaluate the following learning techniques:

LMS Least means squared linear regression can be used to train an improved estimator of cost-to-go. In the offline setting, this is typically done with batched regression using a library like LAPACK. However, in our online setting, batched regression is impractically slow. We use streamed regression which will still converge *provided the data are presented in a random order.* Since an online approach will present the data to the learner in an order related to the search order, we are violating one of the assumptions that guarantees our learning will converge. Therefore we can only make observations as to the empirical performance of online regression, not its correctness.

ANN We trained a three layer neural network with three hidden nodes and used it to compute $\widehat{h}$. This learning technique was also used by [27]. We used a back-propagation learning rate of 0.01. To initialize the network, we collected the first 100 training pairs and performed a batch regression for 1000 epochs or until the network converged. Doing the batched regression any shorter or longer had a negative impact on performance. After this initial period, we began streaming subsequent features and target values to the learner.

ANN Offline We used the same network architecture and training algorithm as before, but now in the offline setting. We used at least 500,000 feature-target pairs taken from 10 random instances, with the exact number of pairs varying by domain. We used $h^*(n)$ as the target value and used $g^*(n)$, the optimal cost of arriving at a node from the initial state, as features in addition to $d(n)$, $h(n)$, *depth*(n), and a constant. We trained the network for 10,000 epochs or until it converged.

LMS Offline Using the same data as we did when training the offline ANN, we optimally solved a least mean squared linear regression using $h^*(n)$ as the target value and $g^*(n)$, $d(n)$, $h(n)$, *depth*(n) and a constant as features.

Accuracy

Figure 5-11 shows the performance of the regression techniques in terms of absolute accuracy on the 8 puzzle, "life" grid navigation, and our small vacuum benchmark. We see, most notably in the 8 puzzle plot, that the offline estimators are more accurate predictors of cost-to-go than the base heuristic or their online counterparts. We also see that online LMS corrections has varied performance. As before, we now need to see how well accuracy translates into search performance.

Guidance

Table 5-7 shows the performance of the regression-based heuristics in greedy search for the same domains that we used in the accuracy study. We see that the offline ANN tends to outperform its online counterpart. This isn't particularly surprising. The offline learners have better data available as they are learning against true cost-to-go values.

For the Eight puzzle, Offline LMS finds solutions faster than any other approach, while the Offline ANN finds the best solutions but requires more expansions. For permutation puzzles like the 8 and 15-puzzle, the state space for all problems is identical and a heuristic learned on one instance of the problem transfers perfectly to new instances of that problem. These two offline techniques benefit by knowing the "correct" answer at the beginning of

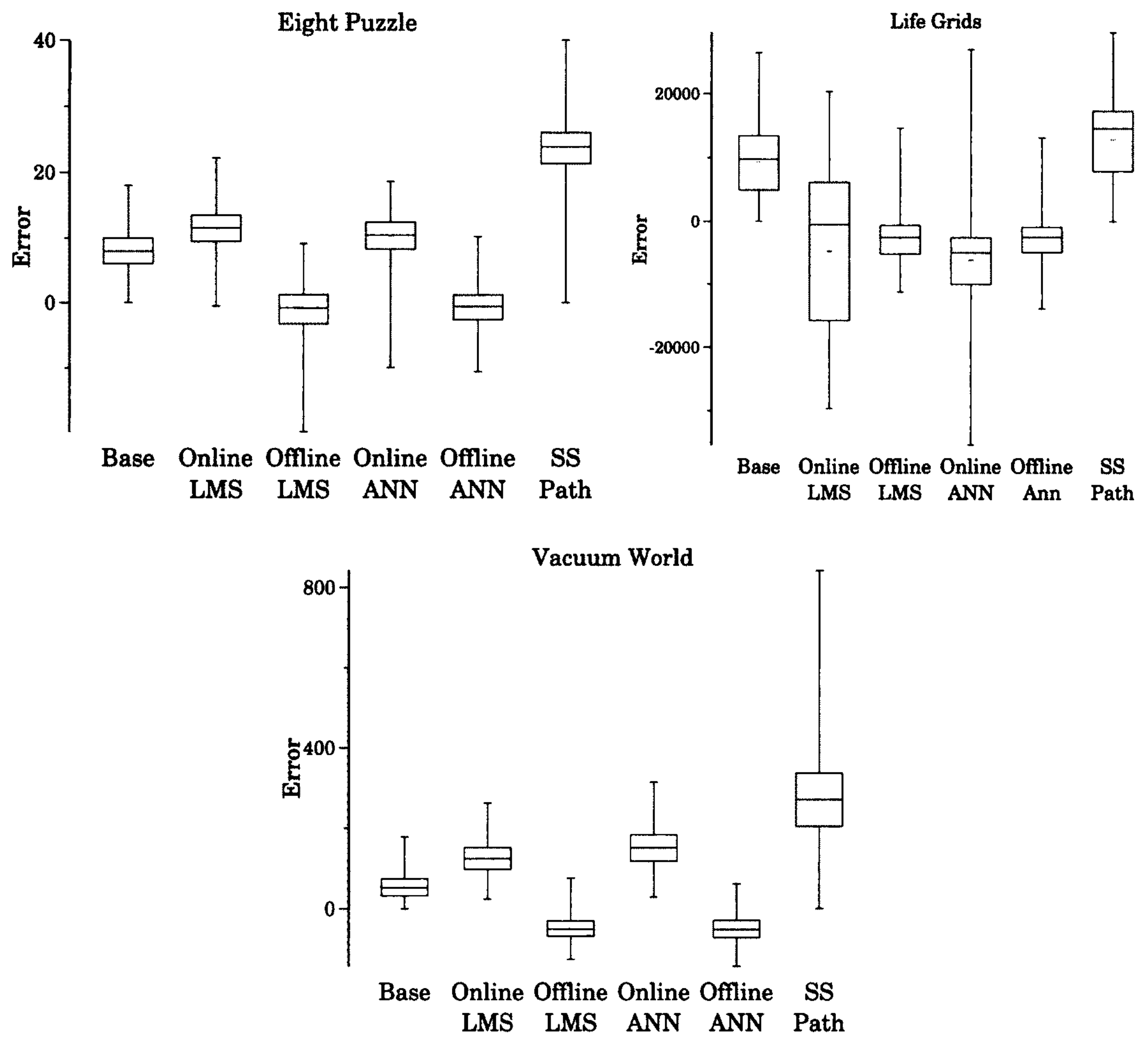


Figure 5-11: Accuracy of heuristics constructed with standard regression techniques on Eight-puzzles, "life" grids, and small vacuum problems

	Eight Puzzle		Life Grids		Small Vacuums	
	generated	cost	$\frac{sec}{1000}$	$\frac{cost}{1000}$	secs	cost
Baseline	582	*128*	169	2993	0.990	2673
Offline LMS	**337**	113	275	3967	*6.266*	1573
Online LMS	514	108	216	2993	**0.158**	**1368**
Offline ANN	798	**31**	323	2809	0.390	2459
Online ANN	610	56	919	5056	0.995	*5415*
SS Path	463	33	**71**	**2795**	0.260	2100

Table 5-7: Performance in greedy search on small domains

search while the online technique must learn the improved heuristic on the fly. That is, the offline techniques have already performed all of their learning and converged on a set of weights to produce $\widehat{h}$. This function will be used on all nodes in search. In contrast, the online techniques are learning their weights, and so $\widehat{h}$ will fluctuate over time leading to potentially unfair comparisons of nodes.

We see that for these small benchmarks, the online LMS correction is competitive with the single-step path corrections. It is nearly as efficient for the Eight Puzzle, and produces better solutions in less time on the small Vacuum World benchmarks. It is interesting to note that online LMS performs best when it is least accurate on these benchmark domains. It is, however, just over three times slower on the Life Grid benchmarks. We now turn towards to larger benchmarks.

Table 5-8 shows the performance of the learned heuristics in greedy best-first search on problems that are too large to enumerate. As these problems are so large, we can not perform offline learning directly. The LMS heuristics now outperform the ANN heuristics which had less variance and a better mean. Why is this? First, recall that the target values for both learners are very different. The offline techniques are allowed to see truth, while the online techniques must approximate the target value for learning using the f-values of

	Fifteen Puzzle		Dynamic Robot		Dock Robot		Vacuums	
	$\frac{secs}{1000}$	cost	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Baseline	29	302	60	522	*169*	*Failed 55*	9.07	9635
Online LMS	8	520	75	522	73	Failed 17	9.42	9635
Reverse LMS	25	150	17128	95	–	–	6.556	9648
Online ANN	*2444*	719	3418	*881*	135	Failed 47	7.40	6155
Reverse ANN	531	*831*	*465133*	254	–	–	*14.28*	*13525*
SS Path	**15**	**90**	**14**	**47**	**0.385**	**29**	**1.22**	**6063**

Table 5-8: Comparing online LMS and ANN's to single-step corrections on larger problems

their children. We posit that the ANN is more sensitive to noise in the target values. Since it is capable of learning a more expressive range of functions than linear regression, it is also more prone to over-training. It may be learning to predict the noise in our prediction of the true cost-to-go instead of predicting h^* as we would desire.

That linear regression and neural network-based heuristics perform so poorly is especially surprising considering how well these techniques have performed in previous work on learning in heuristic search and their high accuracy in our own evaluation. Our explanation is that previous work has mostly focused (with the notable exception of Xu, Fern, and Yoon [77], discussed in Section 5.5) on learning heuristics for optimal search algorithms, namely iterative deepening A*. The role, and therefore the desired properties, of the heuristic in IDA* and greedy best-first search differ substantially. IDA* uses heuristics primarily for pruning, and in many implementations only pruning, while greedy best-first search uses the heuristic solely for guidance. IDA* works by expanding all nodes within a cost bound, and iteratively increasing this cost bound until a solution is contained within it. In all but the final iteration, the relative ordering of nodes is of no consequence, with the exception of the final iteration, and many implementations ignore child ordering as a result [1]. The

[1]The current state-of-the-art is to run IDA* with multiple action orderings in parallel [73],which takes

child ordering is of limited consequence because, excepting the final iteration, IDA* must exhaust the entire f-layer to show that no solution exists within the current bound. This, along with the way the bound is updated, guarantee that when a solution is found it will be an optimal solution.

If our goal is to exhaust all nodes with some property and not, instead, to find a goal, then we don't care what order we expand the nodes in. Accurate cost estimates allow IDA* to prune unpromising nodes early, dramatically reducing the size of these exhausted layers, and therefore dramatically reducing the search effort. In contrast, greedy search cares not one whit for accuracy in the absolute sense. Any heuristic that can correctly sort the set of all open nodes so that nodes leading to good solutions are explored earliest is acceptable even if it is incredibly inaccurate. By way of example, the following heuristic results in perfect performance despite being infinitely inaccurate: the heuristic returns 1 on any optimal path from the root to the goal, and infinity for any other state.

5.3.3 Estimating $h^*(n)$ Using Backwards-looking Heuristics

If we find ourselves in a domain where the heuristic estimate of cost can be computed between two arbitrary points, we have an alternate technique for gathering information about heuristic error: we can compare the heuristic estimate of the cost-to-go from a node n to the initial state with the cost of arriving at that node from the initial state during this search, $g(n)$. This would be especially appealing if we knew that we had arrived at a node by an optimal path, as we would have if we were performing uniform cost search or A* search with a consistent heuristic [41]. $g(n)$ is very likely to be suboptimal in the kinds of searches we consider in this dissertation, but we can still use it as an approximation of the true cost between an arbitrary node n and the root.

We can learn $\widehat{h}(n)$ as a weighted combination of features pointing from n to the initial search stat using any of the previously described regression techniques. The target value of these weighted features is $g(n)$, an approximation of the optimal cost of navigating between

advantage of child ordering, but doesn't use the heuristic to order the children.

	Eight Puzzle		Life Grids		Small Vacuums	
	generated	cost	$\frac{sec}{1000}$	$\frac{cost}{1000}$	secs	cost
Baseline	582	*128*	169	2993	0.990	2673
Online LMS	514	108	216	2993	**0.0.158**	**1368**
Reverse LMS	623	36	168	**2763**	1.065	2956
Online ANN	610	56	919	5056	0.995	*5415*
Reverse ANN	*5032*	83	*1996*	*6829*	1.884	4590
SS Path	463	33	**71**	2795	0.260	2100

Table 5-9: Performance in greedy search on domains from accuracy study

a given node n and the initial state. When we want to produce a forward looking estimate (ie from n to the goal), we simply use features that relate n to the goal rather than to the root. If our forwards and backwards looking features are similarly informed, as we would suspect them to be if they were heuristics computed using the same relaxation, then this should produce a reasonable estimate for $\widehat{h}(n)$.

More concretely, assume that we have a cost-to-go and distance-to-go heuristic that can be computed between arbitrary states, $h(n, m)$ and $d(n, m)$ respectively. When we present training examples to these learning algorithms, we present $g(n)$ as the target value, and $h(n, root)$ and $d(n, root)$ as features. When we want to compute $\widehat{h}(n, goal)$, then we use $h(n, goal)$ and $d(n, goal)$ as features. All of the previously used features have a corresponding backwards looking feature. $g(n)$ can be estimated by $h(n, goal)$, $h(n)$ can be mapped to $h(n, root)$, $depth(n)$ as $d(n, goal)$, and $d(n)$ as $d(n, root)$. It should be noted that such an approach is not nearly as general as those discussed previously. It limits us to domains where we can efficiently compute heuristics between arbitrary states.

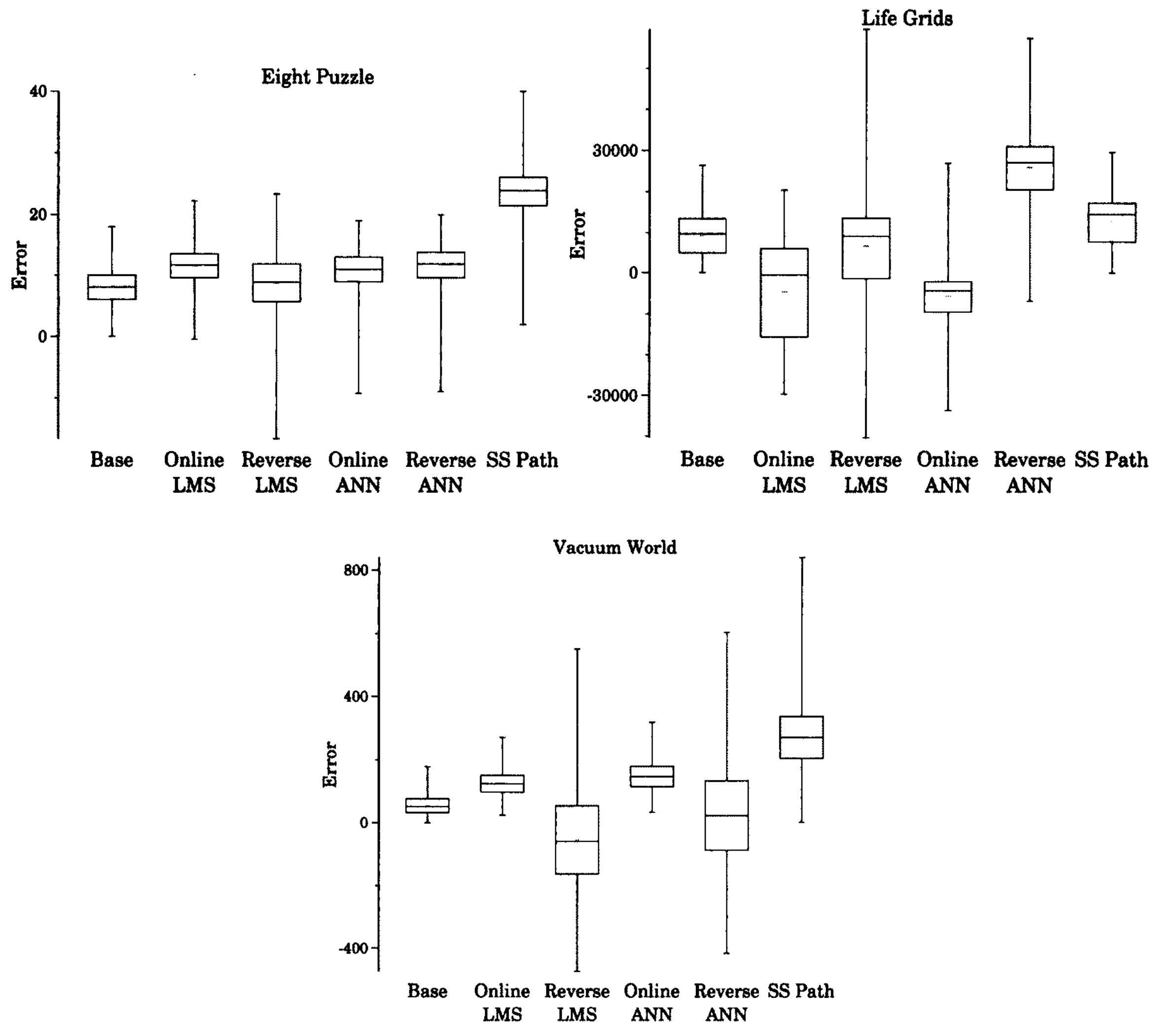


Figure 5-12: Accuracy of heuristics constructed with standard machine learning techniques and backwards looking heuristics on 8-puzzles, "life" grids, and small vacuum problems

	Fifteen Puzzle		Dynamic Robot		Dock Robot		Vacuums	
	$\frac{secs}{1000}$	cost	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Baseline	29	302	60	522	*169*	*Failed 55*	9.07	9635
Online LMS	8	520	75	522	73	Failed 17	9.42	9635
Reverse LMS	25	150	17128	95	–	–	6.556	9648
Online ANN	*2444*	719	3418	*881*	135	Failed 47	7.40	6155
Reverse ANN	531	*831*	*465133*	254	–	–	*14.28*	*13525*
SS Path	**15**	**90**	**14**	**47**	**0.385**	**29**	**1.22**	**6063**

Table 5-10: Comparing online LMS and ANN's to single-step corrections on larger problems

Evaluation

that look towards the initial state of the search space. We use backward looking features (the heuristics computed towards the root for h and d, and the heuristics computed towards the goal for g and $depth$) and $g(n)$ as a target value. We examine the following regression techniques:

Reverse LMS Least mean squared linear regression.

Reverse ANN Estimating the remaining cost-to-go using an Artificial Neural Network. The ANN is constructed as before, with the same random weights and the same initial training period.

Figure 5-12 shows the absolute accuracy of the backwards looking regression approaches over three benchmark domains. While they can produce more accurate estimates, most noticeable in the vacuum world domain where both reverse LMS and reverse ANN heuristics have better means than their forward looking counterparts, they tend to have a much wider variance than the other techniques, something that holds for all three domains. While they can produce better estimates, they don't always, as is the case for life grids where the reverse looking ANN heuristic produces a substantially less accurate estimator than its forward looking counterpart.

Table 5-9 shows the performance of the backwards-looking heuristics in terms of absolute accuracy on the 8 puzzle, "life" grid navigation, and our small vacuum benchmark. We see that, perhaps surprisingly, they do not perform substantially better than similar techniques that look forwards. This is likely because the target values being used for training, the g-values of the nodes being expanded, are much higher than their optimal values. When a node is expanded by an A* search on an admissible and consistent heuristic, we know it is expanded with its optimal g-value. Greedy search on potentially inadmissible heuristics enjoys no such guarantee. It appears that, empirically, this harms the performance of the algorithm. When we consider the additional overhead of computing the backwards looking heuristics together with the large variance of the resulting estimators, it is unsurprising that they perform worse when used in search.

We see similar results for the larger domains in Table 5-10. The learning algorithms that rely on heuristics that look towards the root are omitted for the dock robot domain because we cannot construct similarly informed heuristics in both directions, highlighting a limitation of the approach. The backwards looking corrections rely on our ability to compute a *similarly informed heuristic* between arbitrary states in the space efficiently. The base heuristic we use in this domain isn't from state to state, but from one state to a set of states, since many states satisfy the goal. Thus it is asymmetric.

5.3.4 Summary

One might ask what we lose, in terms of guidance and accuracy, by restricting ourselves to only the information available in the online setting. In this section we compared the performance of the online techniques to heuristics similarly trained offline. We found that the offline techniques generally produced heuristics that were more accurate than those learned during the course of the search itself. Despite being more accurate, these heuristics actually produced worse performance when used in best-first heuristic search algorithms. This was especially surprising considering the success such approaches have enjoyed in previous work on learning heuristics for optimal or near optimal search. We pointed out that the purpose of a heuristic in an optimal search is substantially different than that in a suboptimal search. Specifically, in optimal search we need the heuristic to be accurate so that we can effectively prune away unpromising portions of the space early allowing us to prove solution optimality. In suboptimal search we merely need the heuristic to guide us towards a goal, and the accuracy of the estimations with respect to truth is a secondary concern at best.

5.4 Learning Interleaved with Search

This chapter is primarily concerned with the problem of learning heuristics online during search on a single instance. A strongly related problem is that of learning heuristics while solving a large set of problems. Techniques for this setting are closely related for two reasons. First, single-instance methods can be directly applied to multiple instances individually or, as discussed in Section 5.2.5, heuristics learned while solving one instance can sometimes be transferred to other instances. In our case, the learned single-step errors $\bar{\epsilon}_h$ and $\bar{\epsilon}_d$ can be passed between instances. Second, any technique that learns an improved heuristic while solving multiple instances can be made to work on a single instance by first constructing a training set. We now discuss two techniques designed specifically for the multiple-instance setting.

5.4.1 Bootstrap Learning Of Heuristic Functions

[27] showed that the process of solving a set of instances can be shortened by interleaving learning with solving. Their bootstrapping method attempts to solve all of the instances in a set within a time bound using a base heuristic, h_0. It then uses information from the solved instances, including the true cost-to-go for states along optimal paths and a set of features, to train a new heuristic using an ANN. This process then iterates, using the newly constructed heuristic as a feature, over the unsolved instances until all instances are solved. In addition to solving the instances, this procedure also results in new heuristics. If

an insufficient number of the instances are solved in any given iteration, new easy-to-solve instances are automatically generated by random walks backwards from the goal.

Unlike the techniques discussed previously, bootstrapping learns in between episodes of search, not concurrently with it. When faced with a single target instance, bootstrapping generates a set of instances of progressively increasing difficulty to solve along with the target instance. Effectively, it takes the single problem setting and reduces it to the multi-problem setting by generating a set of instances to solve and learn from. The actual generation process cleverly constructs a set of problems that are almost guaranteed to be of increasing difficulty, a property that bootstrapping finds beneficial. It does this by using a series of longer and longer random walks backwards from the goal state of the problem. Further details are given by [28].

While bootstrapping avoids the need for a set of training instances, it still assumes that the instances are similar enough for the learning to transfer effectively. It also makes two additional assumptions that may not be immediately obvious. The first is that there is some function that allows us to expand nodes backwards. In domains with reversible actions, this exists trivially, in others we must construct such a function. The second assumption is that a fixed goal state exists. There are some problems, such as STRIPS planning, for which the goal is only partially specified, leading to a potentially huge set of goal states from which we must regress in order to generate training instances. It is also implicitly assumed that the base heuristic is too weak to solve the instances we care about, as otherwise no learning ever occurs.

Comparison of Bootstrapping and Single-Step Corrections

Table 5-11 compares the performance of bootstrapping and single-step corrections on the 24-puzzle (a 5x5 sliding tile puzzle). The results for Bootstrapping are taken from [28] and personal communications with the authors. The table is split into two halves. The top shows results for the search algorithm when solving 500 random instances of the 24-puzzle, the second shows results for a larger set of 5000 instances. In both cases, we use

	Total Time	Total Cost
500 instances		
Bootstrapping	*42180 seconds	*73878
Greedy	1921 seconds	Failed 1
Greedy Path Adapt	87 seconds	139674
5000 instances		
Bootstrapping	*421200 seconds	*575402
Greedy	21596 seconds	Failed 17
Greedy Path Adapt	828 seconds	1387004

Table 5-11: Comparison of bootstrapping and single-step corrections on the 24-puzzle. Results with a * taken from [28]

the same instances used in [28]. The columns show the time consumed while solving all instances, and the cost of all solutions summed together appears in the final column. We must take care to note that the algorithms were implemented in different languages and run on different machines, so the timing results are not directly comparable. This table reveals two huge disparities between these two approaches to learning for heuristic search. The path-based corrections are three orders of magnitude faster than bootstrapping, but they produce solutions of much higher cost. Bootstrapping takes nearly 12 hours to solve 500 random instances of the 24-puzzle, whereas path-based corrections take around 90 seconds. For 5000 random instances, this gap widens proportionally with bootstrapping taking several days and path-based corrections solving all 5000 instances in 14 minutes. While the timing results are not directly comparable, the gaps in solving time are so large that we can reasonably conclude that greedy search on path-based corrections is much faster than bootstrapping on the 24-puzzle.

The huge disparities in solving time and solution quality reflect a fundamental difference in the goals of the two approaches. This difference is clearly outlined by the choice of search algorithm the learned heuristic is used in. Bootstrapping relies on a search algorithm designed for finding optimal solutions; it was proposed using IDA*. The optimality (or near optimality) of the solutions returned by a search in bootstrapping are fundamental to the technique because we assume that the solutions returned are optimal (or near optimal) and are thus suitable for use as target values for learning better informed heuristics. Using wildly inflated costs would lead to inaccurate heuristics, which goes against the intent of the technique.

In contrast, we evaluate our approaches in suboptimal and bounded suboptimal algorithms. As we discussed in Section 5.3.2, the desired qualities of a heuristic differ for these two search paradigms. Optimal solvers like IDA* want very accurate heuristics, the type of heuristics that the learning in bootstrapping tends to produce. In suboptimal search, accuracy is unimportant, and ordering is key. In end effect, the two approaches are solving distinct problems: Bootstrapping wants to find nearly optimal (but not provably optimal) solutions quickly, and build an accurate estimator as a side effect, and our approaches seek to find any solution as quickly as possible, with quality being a secondary consideration. In another sense, the techniques are directly comparable as neither provides any guarantee on suboptimality bounds before search begins.

The Statistical Learning of Accurate Heuristics

[5] also proposed a technique, called SACH , that iteratively improves a heuristic used for solving a batch of problems. Using the current heuristic, they attempt to solve all of the problems in a set of instances within a given expansion bound using A* search. Any instances that are solved are used to train a new heuristic using linear regression against h^*. The process then repeats until all instances are solved. If all of the remaining instances are too difficult to solve using the current heuristic, it applies a weight to the current heuristic. Again, we must be able to assume that all of the instances we are trying to solve are similar enough to one another to allow learning to transfer across instances.

In addition to the interleaved approach proposed in [5], a related paper shows how to

	Fifteen Puzzle		Dynamic Robot		Dock Robot		Vacuums	
	$\frac{secs}{1000}$	cost	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Baseline	29	302	60	522	169.297	Failed 55	9.073	9635
SACH	*149613*	*Failed 21*	*236815*	*Failed 6*	*238.142*	*Failed 83*	*85.824*	*Failed 2*
SS Path	**15**	**90**	**14**	**47**	**0.385**	**29**	**1.218**	**6063**

Table 5-12: Performance of SACH compared to other search algorithms

perform SACH online for a single instance [6]. To learn during a search, SACH looks at the nodes on the search frontier. It uses parent pointers to trace backwards from these nodes to the root of the search. For each state along the path from the fringe to the root, it records the difference in g-values and a set of features. It uses these to learn an estimate of the cost-to-go from arbitrary nodes to the goal. The technique used for learning by SACH can learn from arbitrary states, and so it does not need to completely solve an instance to perform learning in the same way that bootstrapping does. The learning is very similar to what we proposed in Equation 5.26, except that instead of using differences between a parent and its best child, it uses differences between a fringe node and all of its ancestors to create training data.

Table 5-12 shows the performance of online, single-instance SACH on the larger benchmark domains from our evaluations. SACH doesn't perform very well when compared to the other algorithms, especially the single-step path-based corrections shown in the table. Again, a large part in the difference in performance is due to the underlying search algorithm. At the heart of SACH is a search algorithm intended to find optimal solutions, A*.

5.5 Learning Search Orderings Directly

The previously discussed techniques attempt to learn an improved estimate of cost-to-go to be used in guiding the search towards goals. While learning cost estimates is quite popular [56, 27, 5, 17], [77] point out that it is not the only approach. They propose two search algorithms, LaSO-BR and LaSO-BST, that rely on a technique that directly learns an ordering over nodes based on the performance of that ordering in a beam search.

A beam search is a form of breadth-first search where the size of the open list, the nodes which have been generated but not yet expanded, is kept to a fixed size. This size is referred to as the beam width of the search, typically denoted b. All nodes on the beam are expanded simultaneously, all children are added to the open list, and then the open list is pruned until it is no larger than the beam width. Plain beam search is a form of memory limited search; by controlling the width of the beam, you can limit how many nodes need to be considered at any time, thereby limiting the maximum amount of memory consumed by a beam search. For domains with many duplicate paths to the same state and many potential cycles, beam searches need to implement a closed list to be effective [76]. Having a closed list removes the limited-memory property of beam search algorithms, but allows them to solve a wider variety of problems.

Rather than performing a linear regression from the features of a node to truth, the LaSO technique learns a weighting over the features that would prevent a beam search with a given beam width b from pruning away all nodes leading to optimal solutions. In essence, the algorithm works by simulating a beam search forward from the root of the search problem. It repeatedly expands all nodes in the current beam and sorts them based on the current weight vector and features of the node. If, when forming the next beam based on the expansion of the previous beam, all nodes that lead to an optimal solution have been pruned, the weights are updated. The weights are updated to promote nodes on optimal paths that could have been in the beam but were not because of the weight vector. Then, the current beam is set to be the remaining optimal nodes on open. This process is performed offline before the algorithm is used to solve problems. It requires a set of training instances that can be optimally solved by some other technique such as A* or IDA*.

This ranking function can be learned from either best-first beam search or breadth-first beam search. We refer to these approaches as LaSO-BST and LaSO-BR respectively.

Training for LaSO-BST often takes far longer than training for LaSO-BST. The first reason for this is that it often takes best-first beam search longer to solve a problem than breadth-first beam search. The second is that it is rarer for a best-first beam search to prune away all nodes leading to an optimal solution because it only expands a single node at a time. If there are multiple paths to an optimal solution, as there are in all of the domains considered in this dissertation, it is likely that several optimal nodes exist in the beam. Unless the children of the node being expanded manage to drive them all out, LaSO-BST will perform no learning in this step. LaSO-BR, on the other hand, expands all nodes at once. Presumably, the optimal nodes are a small portion of the existing beam and they likely only have one or two children on the optimal path. Thus, the optimal nodes must beat out many competitors to be included in the next beam, they often don't, and so learning occurs more frequently in practice.

LaSO learns weights over a set of features such that they prevent a beam search with a given beam width b from pruning away all nodes leading to good solutions. This is done by solving training instances[2], recording all nodes lying on a path to good solutions. [77] point out that any solution path can be used for training. However, if we want to find solutions of minimal cost (high quality) we should also train on optimal solutions. Additionally, the authors point out that a smaller version of a problem may be trained from, then larger problems can be solved using the same learned ordering. A beam search is simulated on the same training instances, and the weights for the features are updated whenever the beam search would prune away all promising nodes from the beam.

Pseudo-code for updating the weights in the breadth-first beam search variant of LaSO is provided in Figure 5-13. In essence, the algorithm works by simulating a breadth-first beam search. It repeatedly expands all nodes in the current beam (line 3) and sorts them based on the current weight vector and features of the node (lines 5 & 6). If, when forming the next beam based on the expansion of the previous beam, all nodes that lead to a good solution have been pruned (lines 7 & 8), the weights are updated (line 9). The weights

[2]This evaluation only considers domains where we can solve the training instances optimally.

```
Update-BR($\mathbf{S_i}, \mathbf{P_i}, \mathbf{b}, \mathbf{w}$)
//$S_i = \langle I_i, s_i(\cdot), f(\cdot), <_i \rangle$ and $P_i = \{P_{i,0}, ..., P_{i,maxdepth}\}$
//$I_i$ is the root node, $s_i(\cdot)$ is the successor function
//$f_i(\cdot)$ generates features of a node
//$P_i$ is the set of all nodes along a desirable path to the goal
1.  $B \leftarrow I_i$
2.  for $depth = 1$ to $maxdepth$
3.      $C \leftarrow$ BreadthExpand($B, s_i(\cdot)$)
4.      for every $v \in C$
5.          $H(v) \leftarrow w \cdot f(v)$ // compute heuristic value of v
6.      Order $C$ according to $H$ and the total ordering $<_i$
7.      $B \leftarrow$ the first $b$ nodes in $C$
8.      if $B \cap P_{i,depth} = \emptyset$ then
9.          $w \leftarrow w + \alpha \cdot (\frac{\sum_{v^* \in P_{i,depth} \cap C} f(v^*)}{|Pi,depth \cap C|} - \frac{\sum_{v \in B} f(v)}{b})$
10.         $B \leftarrow P_{i,depth} \cap C$
```

Figure 5-13: Update rule for LaSO-BR

	Unit 15 Puzzle		Vacuum World		Dock Robot	
Algorithm	$\frac{sec}{1000}$	Cost	sec	Cost	sec	Cost
Base	*29*	*302*	9.073	9635	169.297	Failed 55
LaSO-BR	85	391.95	*142.4*	*Failed 7*	*576.83*	*Failed 98*
SS Path	15	90	**1.218**	**6063**	**0.385**	**29**

Table 5-13: Heuristic performance in greedy search

are updated to promote nodes on good paths that could have been in the beam, $P_{i,j} \cap C$, but were not because of the weight vector. The code for LaSO-BST is similar, but the breadth-first beam search is replaced with a best-first beam search.

Note that the beam width to be trained for is a parameter of the LaSO learning technique (line 7 of Figure 5-13). This makes adapting the learning technique of LaSO to general heuristic search difficult. How to set the beam width to get the best performance for our learned heuristic in a different search algorithm is an open question. For our evaluation, we tried multiple beam widths, 1, 3, 5, 10, 50, 100, 500, and 1000, and then report results for the best-performing beam width for the algorithm.

5.5.1 Evaluation: Greedy Search

Table 5-13 shows the performance of the learned heuristics in greedy best-first search across five benchmark domains. The rows represent the learned heuristic, and the columns are domains. Each major column is divided into two minor columns showing mean solving time and mean solution cost respectively. In the event that a search algorithm failed to solve all instances in the set, the mean solution cost would be infinity, and so we instead report the number of instances it failed to solve.

Table 5-13 shows that greedy best-first search on single-step path corrections performs best in terms of time and solution quality for all domains save the unit-cost fifteen puzzle where it finds better solutions at the cost of increased solving time. We see that it has better coverage in our experimental domains than either of the other two heuristics when used in greedy search; it never fails to solve an instance whereas the baseline and LaSO heuristic do, in inverse tiles and heavy vacuums and dockyard robots respectively. We should also note that the heuristic learned for use in LaSO-BR performs substantially worse than the baseline in several domains. There are two reasons behind this. The first is that in domains where the LaSO heuristic is performing poorly, the learning is unlikely to generalize well. Consider the tiles domains, where the LaSO heuristic substantially outperforms the baseline. Here, the underlying state space is identical (unit-cost) or incredibly similar (inverse cost) across problems, and therefore the learned ordering generalizes well. Contrast that with the dockyard robot domain, where the goal configuration and the cost of transition between depots changes across instances. Here the learned node ranking performs poorly.

Secondly, the LaSO heuristic was trained to be used in beam search, not a best-first search. The role of the heuristic is different in these two kinds of search algorithms, just as the role of the heuristic in greedy search and IDA* differs. In best-first search, we want to push goals, or nodes leading to goals, all the way to the front of *open*. In a beam search the heuristic need only prevent us from pruning away all promising nodes. We can see in line 8 of Figure 5-13 that is exactly what we are training the LaSO heuristic to do. The weights are only updated when all of the promising nodes are pruned away from the beam. In light of that, we shouldn't expect the LaSO heuristic to perform well in greedy best-first search because it isn't designed to provide the right kind of guidance.

5.5.2 Evaluation: Bounded Suboptimal Search Search

Figure 5-14 shows the relative performance of LaSO-BR and the best single-step correction technique in a bounded suboptimal search. Since LaSO-BR does not learn a cost-to-go estimate, we perform the initial search on the learned heuristic directly, and then perform cleanup on $f(n)$. We see that for all suboptimality bounds shown, skeptical search on single-step path corrections outperforms skeptical search relying on the LaSO heuristic. While skeptical search can construct an initial solution much faster when using the LaSO heuristic

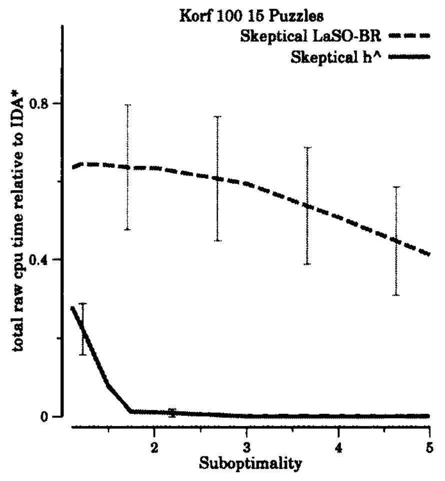


Figure 5-14: Single-step path corrections versus LaSO-BR in skeptical search on tiles

the solution found is much more expensive. Even if this incumbent is within the bound, proving this requires more effort than showing that a solution with lower cost is within the same bound. Results for the other domains are similar.

5.5.3 Evaluation: Beam Search

The previous comparison of learned heuristics is, in some sense, unfair because LaSO wasn't designed to be used in general search algorithms. It was designed to be used in beam search. Table 5-14 shows the relative performance of the learned heuristics in breadth-first beam search for differing beam widths and domains. In the table, rows are the heuristic used to sort the beam, and major columns show the beam width. As before, each major column is divided into two minor columns that report the time required to find a solution and the solution cost respectively.

The first results, those showing the performance on fifteen puzzle (first row of Table 5-14), are particularly surprising because the techniques which learn their heuristics appear to be dominated by search on the base heuristic. The mean time to solution for beam search on $h(n)$ are indeed lower, but this is primarily a result of reduced overhead. The solution costs for each beam are within noise of one another, meaning that the depths to which each beam search is going in this domain are incredibly similar and therefore the number

	Beam Width					
	10		100		1000	
Ordering	Time	Cost	Time	Cost	Time	Cost
			15 Puzzle			
Base	**0.0029**	**173**	**0.0249**	**73**	**0.2999**	**59**
SS Path	*0.0076*	*204*	*0.0360*	*74*	*0.4286*	**59**
LaSO BR	0.0061	191	0.0296	74	0.3345	**59**
			Vacuum World			
Base	*576.0024*	*Failed 145*	*41.3884*	*Failed 17*	*32.9744*	**Failed 1**
SS Path	576.0022	Failed 141	3.5336	Failed 2	32.6410	**Failed 1**
LaSO BR	**372.1086**	**Failed 104**	**3.5102**	**Failed 1**	**30.9520**	**Failed 1**
			Dock Robot			
Base	*97.1460*	*Failed 8*	24.2612	Failed 1	25.2046	Failed 1
SS Path	**0.1460**	**112**	**0.3962**	**29**	**1.6446**	**15**
LaSO BR	97.1544	Failed 6	*151.6342*	*Failed 10*	*299.7842*	*Failed 21*

Table 5-14: Learning techniques in breadth-first beam search

of nodes generated are similar. They are in fact not statistically distinct for many of the beam widths. We see a similar phenomenon for the vacuum world domain, where LaSO BR appears to outperform search on single-step path corrections, but the values are statistically indistinguishable from one another (the confidence intervals overlap significantly).

Table 5-14 also reveals that, as was the case in greedy search, LaSO-BR fails to solve many of the instances in the dockyard robot domain. Again, we attribute this to the fact that the underlying instances are very different from one another. This impedes the performance of techniques which perform all of their learning offline. Note that greedy search on the LaSO heuristic (Table 5-13, second row) also performs incredibly poorly, and so this performance is likely the fault of the heuristic and not the search algorithm itself.

Learning, both LaSO-BR and single step corrections, generally improve the performance of our beam search algorithms. At worst, it does not appear to harm performance. Single-step path corrections provides better guidance in beam search than the LaSO heuristic. From Table 5-14, we see that it solves more instances across the beam widths examined than the other two heuristics. Not only does it solve more instances, but it tends to have lower mean solving time and better mean solution quality. While these times are not always distinguishable from search performed on the LaSO heuristic, in the dockyard robot domain search on single-step path corrections is clearly better than search on the LaSO heuristic.

5.5.4 Summary

Suboptimal search algorithms need functions that can effectively discriminate between nodes to guide search. Learning exactly what we need is very appealing. The previous technique for learning search orders directly, LaSO, is designed for beam search. Unfortunately it does not appear to work as well when used in other kinds of search algorithms. Although the single step techniques proposed in this work provide the largest advantage in best-first search algorithms, they can be competitive with LaSO techniques when used in beam search.

5.6 Other Related Work

We now present previously proposed alternative and complementary techniques for learning before any search begins, and in between multiple runs on a single instance. We say that the techniques are complementary as the single-step error corrections presented here could be added to these techniques to improve performance.

The most popular, or at least the most frequently proposed in the literature, technique for learning heuristics for search is to learn those heuristics before any search of the target instances begins, offline, from training data. All such techniques assume that training instances are abundant, or at least that they are easily generated. Further, several of the following approaches make use of strong domain-specific features to use for the learning of heuristics. Both of these assumptions limit the applicability of the techniques.

Samuel's checker playing program [58] used learning techniques to construct good static evaluators to be used in his alpha-beta pruning game tree search and it is the earliest to make use of learning techniques for constructing heuristic evaluation functions. Positional strength is not the same quality as cost-to-go, so this technique is not directly comparable, or even easily combined, with those presented here.

[59] learn to identify sets of nodes with interesting properties, such as nodes that are likely to lie on a path to a solution or nodes that are more likely to be near to solutions. They then use this classification to perform efficient tie-breaking in optimal search algorithms. Learning is performed offline, from training data, before any search over the target instances begins.

[56] present a technique for combining an arbitrary number of features into a single cost-to-go estimate. In their implementation, these features are pre-computed pattern databases, powerful heuristics in their own right. They train an artificial neural network (ANN) to map these values to an estimate of the cost-to-go using h^* as the target value. When problems are too large to solve optimally, they substitute the optimal solution of a relaxed problem for h^*. Naturally, this lessens the quality of the training data and leads to slightly worse estimates as a result. [57] provide a technique for compressing pattern databases efficiently

that could be used here to ensure admissibility. While we do not rely on admissibility, such a powerful cost-to-go estimate would likely make a good starting heuristic for our online technique.

[17] proposes a technique that learns an improved heuristic for multiple searches over the same instance of a pathfinding problem. Specifically, he assumes that the same graph is being searched every time, but that the start and goal nodes may change. A cost-to-go heuristic is learned in between search episodes using information recorded during the previous search. Features of a node are recorded and a heuristic is learned by performing a regression from these features to the true cost-to-go. As more problems are solved, more data becomes available and the quality of the heuristic improves as a result of that. While bootstrapping and the original implementation of SACH were exclusively evaluated on permutation puzzles, where each solution shares the same underlying search space, it can be run without alteration on problems where the underlying state space differs between instances. This isn't obviously the case for the technique proposed by Fink.

5.7 Discussion

There are three times when learning can happen: before any search, in between solving instances of a batch, or during the execution of a search. We do not thoroughly investigate the possibility of combining offline or interleaved learning with online learning in this dissertation. As we've shown that the online technique works with the base heuristic and generally improves when the accuracy of the underlying heuristic improves, it is likely that a combination of the techniques would be very beneficial.

Nearly all of the previous work has focused on finding optimal or near optimal solutions. There have been very few techniques that consider speed as the primary figure of merit. Learning heuristics generally leads to a certain amount of inadmissibility, preventing us from guaranteeing cost-optimality. There are many applications of search, and while many demand solutions of the highest possible quality there are important settings that require us to solve problems quickly.

Online techniques for improving heuristics allows us to take advantage of the information present in every expansion. By definition, search algorithms tend to spend a majority of their time searching. Every expansion, of which there will in the worst case for search (but the best case for learning) be many, provides an opportunity to learn a potentially improved evaluation function.

A point that arose several times in our investigation is that different kinds of search algorithms have differing requirements for their heuristics. For finding optimal search, the problem that nearly all previous work focuses on, we need the heuristic to be extremely accurate in terms of absolute error. That is, the heuristic must be able to very accurate predict the true cost-to-go, h^*. This is because in optimal search the heuristic is used to prove that the returned solution is optimal (ie expand all nodes where $f(n) \leq g(opt)$). The absolute magnitude of the heuristic determines what portion of the search space we must exhaustively search before we can prove that the solution we find is of a sufficient quality. If we are unconcerned with proving quality bounds, or if time is at a larger premium than quality, we should use search algorithms that rely on the guiding power of a heuristic. Here we are not exhausting large portions of the space to prove quality and the limiting factor of the search is how quickly we can guide the algorithm into goals. Any heuristic that assigns a node close to a solution a relatively smaller value than one far away will work well here, regardless of how far away its estimates are from truth.

5.8 Conclusions

Learning for heuristic search had previously considered primarily in two settings: learning an improved heuristic offline, before any search begins, and learning an improved heuristic in between the solving of instances in a large batch. The technique presented in this chapter, learning corrections from single-step error, learns during the execution of the search itself. It can be easily combined with either, or both, of the other two settings to improve performance. Our technique has the advantage of making few assumptions. Specifically, we do not assume a training set or the ability to generate one, we do not assume we can solve problems optimally, and we needn't assume that all of the instances being solved are similar. We merely require that a heuristic search algorithm is being used, and we need a cost-to-go and a distance-to-go heuristic. Both are likely to exist for any given domain. This allows the described approach to be widely and immediately applicable. In our evaluation, we found that, our technique produces better solutions faster than the base heuristics when used in greedy best-first search across a wide range of benchmark domains. The technique also proved to be beneficial in bounded suboptimal search, improving upon the performance of previous state of the art algorithms while removing the need for parameter tuning.

Part II

Search Strategies

CHAPTER 6

INTRODUCTION

The previous section of the dissertation was primarily concerned with the construction of heuristic information for guiding heuristic search algorithms. The following chapters investigate suboptimal search strategies in three main settings: bounded suboptimal search, bounded cost search, and anytime heuristic search. As we previously noted, part of the common thread between algorithms for all of these settings is that the can, and should, consider the inadmissible estimates of cost and actions-to-go that we discussed in the previous section of the dissertation.

The first chapter in this section discusses the setting of bounded suboptimal search. Algorithms which address the problem of bounded suboptimal search must find a solution whose cost is provably within the user-supplied factor of optimal. The first major contribution of this chapter is an argument that suggests they should also perform this task as quickly as possible. Much of our discussion of the performance of bounded suboptimal search algorithms from the literature and from this thesis will be focused on under what circumstances, if any, the algorithm is capable of minimizing solving time subject to a suboptimality bound.

The second major contribution of Chapter 7 is the Explicit Estimation Search algorithm, first mentioned in Thayer et al [68], and fully presented in Thayer and Ruml [69]. The explicit estimation search algorithm is the goal-statement of bounded suboptimal search made expansion order. It provides state of the art performance for many benchmark domains, and the general framework that it lays out provides efficient algorithms for other suboptimal search setting, including bounded cost and anytime search, as we will see in Chapters 8 and 9.

The second chapter of this section, Chapter 8 covers a relatively new variant of suboptimal search, the bounded cost search domain. In bounded cost search the goal is to find any solution within a user specified cost-bound C as quickly as possible. This differs from the bounded suboptimal domain in that we no longer care what the cost of the optimal solution is, as we must prove an absolute rather than a relative bound.

The final chapter of this section covers the anytime search setting. Anytime search is one of three major methods for controlling the amount of time consumed by a heuristic search algorithm. Anytime search is designed for situations where some unknown amount of time is available for solving the problem. Since the deadline is unknown, anytime search algorithms must expand to make use all available time, or at least as much time as is required to find the optimal cost solution. Although anytime search algorithms are designed for unknown deadlines, they are also popularly used in settings where the deadline is known before hand, as it is in the international planning competition.

CHAPTER 7

BOUNDED SUBOPTIMAL SEARCH

7.1 Introduction

As we previously discussed, when time is not a concern, we can solve heuristic search problems optimally with algorithms like A* [22] or IDA* [32]. These algorithms work by slowly increasing a lower bound on the cost of an optimal solution to the problem under until a solution is contained within their bound. If the bound was increased slowly enough, this solution has provably optimal cost.

Proving that a solution has optimal cost can be very expensive: the search algorithm must examine all nodes that could potentially lead to a solution of lower cost. To determine which nodes might lead to a solution of lower cost we employ a heuristic evaluation function. Even if we have heuristics that err in their estimate of the cost-to-go from a node to the goal by no more than a constant (which is unrealistically accurate), finding cost-optimal solutions is still intractable [50]. The cost of optimal solving is fundamentally incompatible with many applications that require fast response times.

The requirements of an application may require us to abandon optimal search as too expensive, but that does not mean we must accept poor solutions. By requiring that solution cost be less than a pre-specified factor of optimal (even if the optimal cost is unknown) we can retain some control over the cost of solutions returned by a search algorithm while potentially increasing the speed with which those solutions are found. Algorithms that meet this requirement are called bounded suboptimal search algorithm. These algorithms have also been referred to as ϵ-admissible or w-admissible search algorithms, as the user-supplied parameter is often named ϵ or w. The goal of bounded suboptimal search is to return a solution that is within a factor w (or alternatively $1 + \epsilon$) of optimal as quickly as possible.

This Section proceeds as follows: We begin with a discussion of bounded suboptimal search, focusing on the properties an algorithm must have and a discussion of what the overall goal of bounded suboptimal search is. In chapter 7, we argue that the goal of bounded suboptimal search is to minimize solving time with respect to a user-supplied suboptimality bound.

In Chapter 7.12, we introduce the explicit estimation search algorithm (EES), a new algorithm designed to optimize the goal we proposed for bounded suboptimal search. EES works by combining potentially over-estimating heuristics for solution cost and solution length to find solutions provably within a user-provided suboptimality bound as quickly as possible.

After describing EES, we relate it to previous work in the field of bounded suboptimal search in Chapter 7. Our discussion of previous bounded suboptimal search algorithms includes a discussion of how much previous work does not strictly minimize solving time under a suboptimality bound. An empirical evaluation that shows EES is frequently far more efficient for a given suboptimality bound than previous algorithms. Not only is EES often faster, but it is more robust than previous approaches as well. We will see that the mean solving time for EES across all benchmarks considered in this dissertation is lower than that of other algorithms because EES never fails catastrophically for any of the domains considered, while all other algorithms have at least one domain where they perform exceptionally poorly.

In Chapters 8 and 9 we discuss EES in the context of related heuristic search settings. In particular, we will look at how EES relates to the bounded-cost search setting, where we would like algorithms to produce a solution with cost less than C as quickly as possible, and to the anytime search setting, where we would like algorithms that provide the best possible solution under some unknown deadline. EES can be adapted directly to either of these domains, resulting in performance exceeding that of previous approaches in these areas.

7.2 Problem Definition

Bounded suboptimal search attempts to address a shortcoming of optimal cost heuristic search: optimal search is often prohibitively, and perhaps needlessly, expensive. Finding provably optimal solutions to problems takes much longer than finding suboptimal solutions in general. If the time requirements of an application are short, optimal search is not always an option. Even if the time constraints of our application could permit optimal search, suboptimal solutions may be "good-enough" in a variety of situations and we may wish to spend our resources on parts of the task other than search, making suboptimal solving a better decision than finding provably optimal solutions.

Bounded suboptimal search fixes this problem by allowing the user to trade increased solution suboptimality for potentially decreased solving time. Generally, though not always as we will see in the empirical evaluation, an increase in suboptimality bound produces a reduction in solving time for a bounded suboptimal search algorithm on a given problem. The reduction in solving time is also generally quite similar across similar instances. Thus, a user typically plays with the suboptimality bound of a search algorithm until it is fast enough.

This suggests the following goal for bounded suboptimal search algorithms: for a given suboptimality bound, find a solution as quickly as possible. If the user is going to raise the suboptimality bound until solving is sufficiently fast, we would like our algorithms to be sufficiently fast with the smallest increase in the suboptimality bound.

There are really two tasks that any bounded suboptimal search must solve. First, it must find a solution, should one exist. Of course, we would like search to find that solution as quickly as possible. Secondly, it must be able to prove that this solution is within a user-specified factor of optimal. We now discuss each task in turn.

7.2.1 Finding a Solution as Quickly as Possible

Heuristic search algorithms must find a solution, or prove that none exists by examining all states in the search space and showing that none of them are a goal. However, if a solution to the problem does exist, then we want to find that solution as quickly as possible.

Search algorithms have used a variety of approaches to find solutions quickly. Algorithms like weighted A* [43] and greedy best first search place additional emphasis on the heuristic estimate of cost-to-go to encourage search for a solution to complete quickly. In domain independent planners such as FF [25] potentially over-estimating estimates of cost-to-go, which we will refer to as $\widehat{h}(n)$, are used in place of admissible cost-to-go estimators in an effort to speed search. Reducing the number of potential solutions under consideration can also speed search, so long as the subset we choose still contains solutions. This is the approach taken by algorithms such as beam search [18, 3], A_ϵ^* [42], and A_ϵ [19].

Focusing on cost-to-go overlooks the difficulty of completing a partial solution. To find solutions as quickly as possible, we would ideally rank nodes by how quickly they can be completed. Although it is difficult to see how we can estimate the difficulty of finding a solution under a node directly, it is relatively straight forward to estimate the length of a solution path passing through a node. All else being equal, solutions with fewer actions tend to require less search to find, as the complexity of search is often a function of solution length. Despite this natural relationship between solution length and solving difficulty, many heuristic search algorithms, including weighted A*, fail to take this quantity into account explicitly.

7.2.2 Proving Bounds

Suboptimal search algorithms are generally faster than cost-optimal search algorithms because they expend less effort proving that their solutions are of sufficiently low cost. Optimal search algorithms must show that there is no possible solution to the problem with smaller cost whereas bounded suboptimal search algorithms must only show that there is no solution whose cost is more than a factor w smaller than the solution returned. By lowering the standard to which we hold solutions in search, we reduce the cost of proving the bound precipitously. Under certain assumptions, Davis showed that bounded suboptimal search can be run in time linear in the length of the returned solution [12].

There are two ways by which we can show a solution lies within a given suboptimality bound. These are by exhaustion, and by construction. We will discuss the approaches briefly now and in depth later in connection with specific algorithms.

Optimal search algorithms such as IDA* show that a solution is within a desired suboptimality bound (e.g. $w = 1$) by exhaustion. That is, they exhaust all potential solutions that could have cost less than a factor w times the solution. To do this, we must compute a lower bound on the complete cost of a partial solution. If $g(n)$ is the cost of executing the actions in a partial solution and $h(n)$ is a lower-bound on completing that solution, then $f(n) = g(n) + h(n)$ is a lower bound on a complete solution using the prefix n. Algorithms that work by exhaustion must merely extend all partial solutions until $f(n) \geq w \cdot g(sol)$ where *sol* is the solution we would like our algorithm to return.

Proving that a solution lies within a suboptimality bound by construction is slightly different. We must show that at the time a search algorithm was considering a node it could show the solution represented by that node was within a bounded factor of the optimal-cost solution. Generally such a proof relies on the order in which partial solutions are considered by the search algorithm and properties of $h(n)$, our estimator of cost-to-go. Proving a solution is within a suboptimality bound by construction is neither explicitly more difficult nor easier than proving a solution is bounded by exhaustion.

7.3 Weighted A*

Weighted A* [43] is the oldest and simplest bounded suboptimal search algorithm. It modifies the standard node evaluation function of A*, $f(n) = g(n) + h(n)$ into $f'(n) = g(n) + w \cdot h(n)$. Weighting the cost-to-go heuristic encourages the search algorithm to prefer states where there is little estimated cost remaining to the goal, as they tend to be closer to the goal. In unit-cost domains, this corresponds to preferring nodes with low $d(n)$.

```
weightedAstar(root, w)
1.  $open \leftarrow \{root\}$
2.  while $open \neq \{\}$
3.      let $n = \text{argmin}_{n \in open} f'(n) = g(n) + w \cdot h(n)$ in
4.          if $goal_p(n)$
5.          then return $n$
6.          else $open \leftarrow open - \{n\}$
7.              for each child $c$ of $n$, $open \leftarrow open \cup \{c\}$
8.  return no solution
```

Figure 7-1: Weighted A* pseudo code

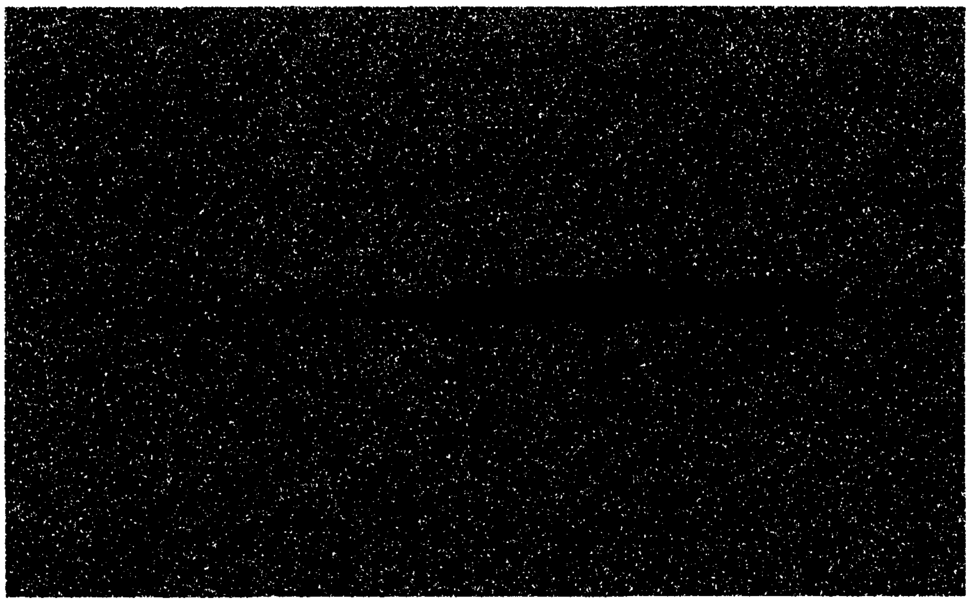


Figure 7-2: Expansion order of weighted a* ($w = 1.5$)search on a pathfinding problem

Figures 1-10 and 7-2 shows the order in which weighted A* expanded nodes when solving a unit-cost grid world navigation problem with four-way movement. Nodes in yellow were expanded early on in the search, and as the nodes become redder they were expanded later on in the search. The starting state for this problem is in the middle of the left-hand side of grid, and the goal state is in the middle of the right hand side. When comparing the two expansion orders, we see that weighted A* and A* are quite similar. Figure 7-2 looks much like a thinned version of the expansion order of A* shown in Figure 1-10.

7.3.1 Implementation Concerns

An equivalent formulation, $f'(n) = w_1 \cdot g(n) + w_2 \cdot h(n)$ allows the use of an integer bucket-based open list for a larger range of weights than is possible with the standard single-weight conception of weighted A*. This is more efficient than a heap based open-list, but less general. For example, at a weight of 1.25, very few nodes are going to have integer values, even for domains with unit cost actions. However, $4 \cdot g(n) + 5 \cdot h(n)$ produces an identical node expansion order and all resulting node-evaluations will be integer if the underlying g and h-values are also integer. Thus, any rational weight can be done with an integer based queue. Tie-breaking is then handled by the order of nodes in the buckets.

7.3.2 Proof of Bounded Suboptimality

Before discussing the strength and shortcomings of weighted A*, we reproduce the proof from [43] showing that it obeys a suboptimality bound.

Theorem 1 *If $h(n)$ is an admissible heuristic, then the solution returned by weighted A* has cost within a factor w of the optimal solution.*

Proof: The proof is based on the construction of the open list. Let p be the deepest node along the cheapest path to a goal. Initially it is the root, and when the root is expanded, it is one of the generated children. Since we never discard a node in this version of weighted A*, p is on the open list at all times, including when a solution is returned:

```
Weighted A*($root, w$)
1.  $open \leftarrow \{root\}$
2.  while($open \neq \{\}$)
3.    remove $n$ from $open$ with minimum $f'(n) = g(n) + w \cdot h(n)$
4.    if $n$ is a goal
5.      return $n$
6.      else for each child $c$ of $n$
7.        if another node is in $open$ with the same state as $c$
8.          then keep the node with the smallest $g$-value
9.          otherwise insert $c$ into open
10. return no solution
```

Figure 7-3: Weighted A* pseudo code with duplicate dropping

$$
\begin{aligned}
g(sol) &= f'(sol) && \text{By admissibility of } h(n) \\
f'(sol) &\leq f'(p) && \text{By Line 3 of Figure 7-1} \\
&\leq g(p) + w \cdot h(p) && \text{By definition of } f' \\
&\leq w \cdot (g(p) + h(p)) && \text{By algebra} \\
&\leq w \cdot f(p) && \text{By definition of } f \\
&\leq w \cdot f(opt) && \text{By admissibility of } h
\end{aligned}
$$

$\square$

7.3.3 Dealing with Duplicates

Of all the algorithms we discuss in this dissertation, weighted A* is the only one that can ensure bounded suboptimality when electing to not re-open previously expanded states. That is, it can ignore, or drop, duplicate states even when they are encountered by a better path. We provide pseudo-code for weighted A* with duplicate dropping in Figure 7-3. The proof of bounded suboptimality for the duplicate dropping variant of weighted A* requires that the heuristic used is consistent.

Theorem 2 *Following [71], if $h(n)$ is an admissible and consistent heuristic, then during search with duplicate dropping weighted A* there always exists an open node p that is the deepest node along an optimal solution path that has $g(p) \leq w \cdot g^*(p)$, where $g^*(p)$ is the optimal cost of arriving at p.*

Proof: The proof is by induction over iterations of the search algorithm. For the base case, we consider the first expansion, that of the root. One of its children must be along an optimal path and therefore it must also have its optimal g value. For the inductive step, assume that there is a node along an optimal path, p_{i-1}, whose g value is within a factor w of its optimal g value. Consider its fate during expansion. If it is not selected for expansion, then p_{i-1} is still the deepest node along an optimal path on open, it obeys the inequality $g(p_{i-1}) \leq w \cdot g^*(p_{i-1})$, and the proof holds trivially. If p_{i-1} is selected for expansion, one of two things happens: 1) the next node along an optimal path, p_i, is inserted into open, or 2) p_i was already expanded by another path and the new duplicate version is discarded. We now proceed by cases:

1) p_i is inserted: If p_{i-1} is expanded and p_i is inserted into open, then $g(p_i) = g(p_{i-1}) + c^*(p_{i-1}, p_i) \leq w \cdot g^*(p_{i-1}) + c^*(p_{i-1}, p_i) \leq w \cdot g^*(p_i)$ and the theorem holds.

2) p_i is discarded: This can only happen because p_i is already in closed after having been expanded along another path. If p_i was expanded before whichever w-admissible ancestor p_{i-j} was on the open list at that time, this means that $f'(p_i) \leq f'(p_{i-j})$. But then:

$$\begin{array}{rcll}
f'(p_i) & \leq & f'(p_{i-j}) & \text{by expansion order} \\
g(p_i) + w \cdot h(p_i) & \leq & g(p_{i-j}) + w \cdot h(p_{i-j}) & \text{by definition of } f' \\
g(p_i) + w \cdot h(p_i) & \leq & g(p_{i-j}) + w \cdot (c^*(p_{i-j}, p_i) + h(p_i)) & \text{by consistency of } h \\
g(p_i) + w \cdot h(p_i) & \leq & g(p_{i-j}) + w \cdot c^*(p_{i-j}, p_i) + w \cdot h(p_i) & \text{by algebra} \\
g(p_i) + w \cdot h(p_i) & \leq & w \cdot g^*(p_{i-j}) + w \cdot c^*(p_{i-j}, p_i) + w \cdot h(p_i) & \text{by inductive assumption} \\
g(p_i) + w \cdot h(p_i) & \leq & w \cdot (g^*(p_{i-j}) + c^*(p_{i-j}, p_i)) + w \cdot h(p_i) & \text{by algebra} \\
g(p_i) + w \cdot h(p_i) & \leq & w \cdot g^*(p_i) + w \cdot h(p_i) & \text{by definition of optimal path} \\
g(p_i) & \leq & w \cdot g^*(p_i) & \text{by algebra}
\end{array}$$

□

Theorem 3 *Following [71], if $h(n)$ is an admissible and consistent heuristic, then weighted A* may drop duplicates during search while still adhering to the suboptimality bound w.*

The proof of bounded suboptimality under duplicate dropping is nearly identical to the previous one, except for the third step: ***Proof:***

$$\begin{array}{llllll}
1) & g(sol) = f'(sol) & & & & \text{By admissibility of } h(n) \\
2) & & f'(sol) \leq f'(p) & & & \text{By expansion order} \\
3) & & g(p) + w \cdot h(p) & \leq & w \cdot (g^*(p) + h(p)) & \text{By Theorem 2} \\
4) & & & \leq & w \cdot f(p) & \text{By definition of } f \\
5) & & & \leq & w \cdot g(opt) & \text{By admissibility of } h
\end{array}$$

□

The additional step is required because we are dropping duplicates, and cannot guarantee that p was arrived at by an optimal path anymore. Theorem 2 reassures us that the path to p is not so costly as to ruin the proof of bounded suboptimality. Weighted A* is the only algorithm of which we are aware that can drop duplicates without sacrificing its suboptimality bound, and it can only do so as long as h is consistent.[1]

[1]If we were to drop duplicate states without a consistent heuristic, we would suffer a loosening of our suboptimality bounds. In [16], it was shown that other bounded suboptimal algorithms, and even weighted

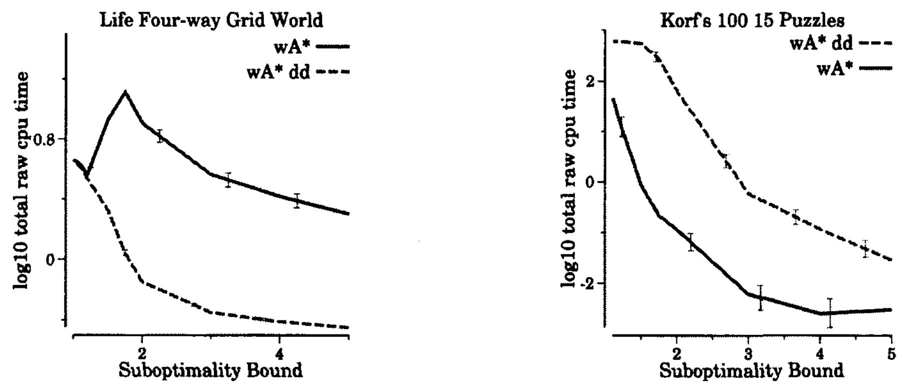


Figure 7-4: Impact of duplicate dropping on weighted A* performance

Figure 7-4 shows the impact duplicate dropping has on performance in two domains: grid navigation with life costs, a domain with tight cycles and thus many duplicates, and the fifteen puzzle, a domain that has few cycles and few duplicates. Although results vary strongly between these two domains, as we see in Figure 7-4. For grids, it has a strong benefit as ordinary weighted A* is almost an order of magnitude slower that weighted A* with duplicate dropping. In tiles, we see the opposite: weighted A* with duplicate dropping is nearly a full order of magnitude slower than ordinary weighted A*. It's difficult to be certain a priori which strategy will perform best, but, in our experience, it tends to be the case that for domains with tight cycles and many duplicates, dropping duplicates is beneficial, while for domains with few cycles, duplicates should be re-expanded.

7.3.4 Solving Time vs Suboptimality Bound

We now turn to evaluating weighted A*, the most well knows bounded suboptimal search strategy, with the new EES approach. Figure 7-5 shows the time weighted A* required to solve a problem across a variety of suboptimality bounds (1, 1.0005, 1.001, 1.01, 1.05, 1.1, 1.15, 1.2, 1.3, 1.5, 1.75, 2, 2.5, 3, 4, and 5). Experiments were run until the problem was

A* can drop duplicates without completely losing their suboptimality bound. The resulting suboptimality bound is w^{length}, where *length* is the length of the returned solution. For example, in the heavy vacuum domain, where solutions are generally around a thousand actions long, if we were to run weighted A* with duplicate dropping at $w = 1.1$, the resulting bound would be about $2.47 \cdot 10^{41}$, rendering it useless.

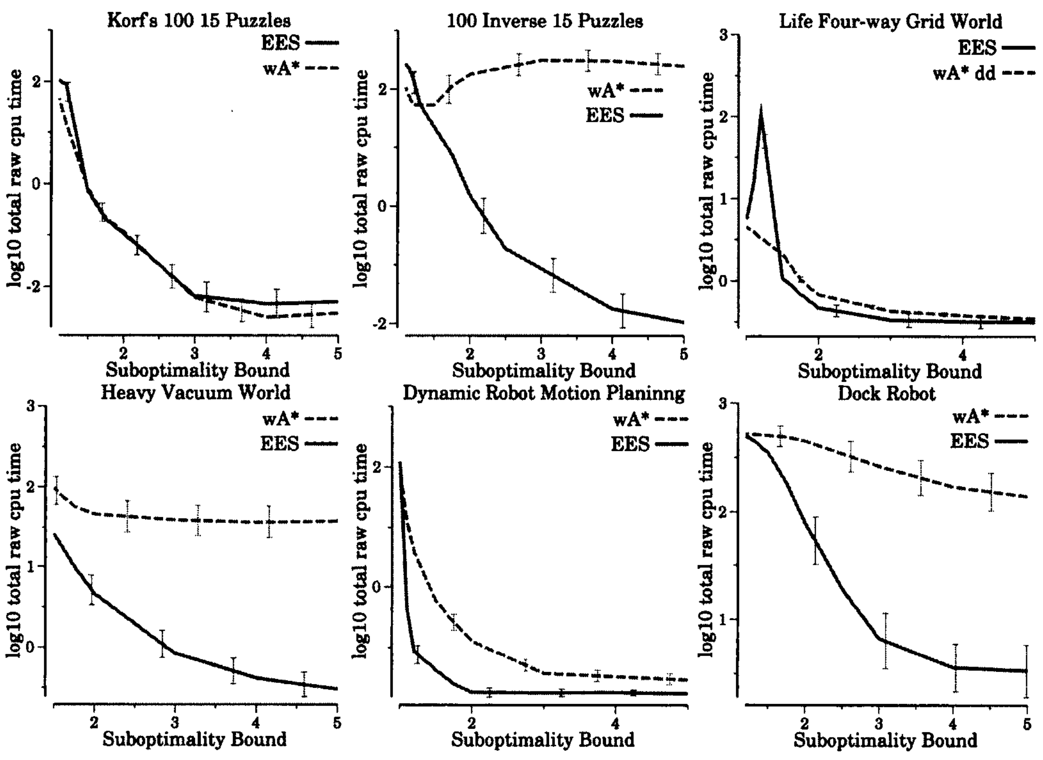


Figure 7-5: Performance of weighted A*: suboptimality bound vs. solving time

solved, memory was exhausted, or more than ten minutes had passed. The reported times are therefore optimistic, as weighted A* will report a time shorter than what is needed to solve the problem whenever it fails to return a solution.

Generally, as the suboptimality bound supplied to weighted A* increases, the time the algorithm requires to find the solution decreases. That is the intended behavior of weighted A*. It is supposed to scale gracefully between A*-like behavior, cost-optimal solutions and long solving times, and greedy search behavior, with expensive solutions but short solving times. There are, however, three domains where this trend is not observed: inverse cost tiles, heavy vacuum world, and dock robots.

In the heavy vacuum domain, weighted A* performance plateaus early on. That is, for suboptimality bounds larger than two, the time required to find a solution doesn't noticeably decrease. This is because weighted A* has already converged on the performance of greedy best first search. Greedy search takes on average 94 seconds to solve one of these instances.

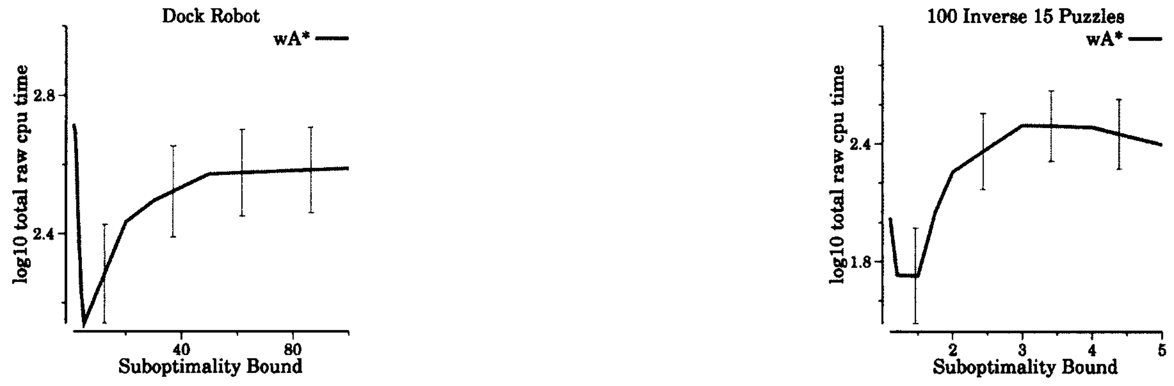


Figure 7-6: Weighted A* doesn't always improve with larger w

No matter how much focus weighted A* shifts from cost-incurred to cost-to-go, it can never become greedier greedy search, and so we would expect the performance of greedy search on a problem to be a bound on the performance of weighted A* (as w becomes very large, ordering on $f'(n)$ and $h(n)$ become identical). For this domain, EES is substantially faster, up to two orders of magnitude, because it is using additional information, an estimate of actions-to-go, to pursue easy-to-find solutions. Outside of tie breaking, it is unclear how to incorporate such information into weighted A* in a general way without losing guarantees of bounded suboptimality.

In the inverse tiles problems and dock robot problems weighted A* demonstrates a U-shaped performance curve (not shown in dock robots because it first occurs at $w > 10$). This is surprising because it defies the conventional wisdom that as suboptimality bounds are relaxed, heuristic search algorithms take less time to solve problems. weighted A* still converges on greedy search performance, it just so happens that greedy search performs very poorly for these problems. EES avoids bad behavior by using online corrections of the misleading admissible cost-to-go heuristic and by relying on multiple sources of information. Thus, EES is faster on domains with varying action costs and in domains where the admissible heuristic is poorly informed.

In the inverse tiles problems and dock robot problems shown in Figure 7-5 weighted A* demonstrates a U-shaped performance curve. That is, it starts of needing large amounts

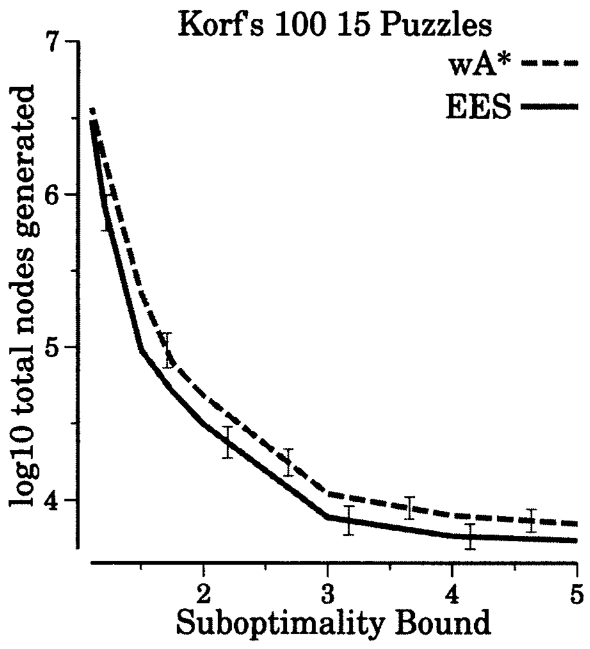


Figure 7-7: Comparing EES and weighted A* based on nodes generated

of time, improves for a while, and then becomes worse. We show weighted A* alone in these domains to highlight the effect in Figure 7-6. This is surprising because it defies the conventional wisdom that as suboptimality bounds are relaxed, heuristic search algorithms take less time to solve problems. It turns out this notion of heuristic search performance is itself a heuristic in that it generally, not always, holds. In these problems weighted A* is still converging on the performance of greedy best-first search, it just so happens that greedy search performs very poorly for these problems because the heuristic can be quite misleading. By putting too much focus on cost-to-go, weighted A* ends up ignoring cost-incurred and is mislead by the heuristic. There is a 'sweet-spot' where it performs quite well, but where this is will vary by domain and instance. EES avoids this potentially bad behavior by using online corrections of the misleading admissible cost-to-go heuristic and by relying on multiple sources of information. The latter is known to improve the performance of satisfying search substantially [50], but it is not known if a direct adaptation of this technique to bounded suboptimal search results in improved performance.

Figure 7-7 shows the performance of Weighted A* and EES as a function of the number

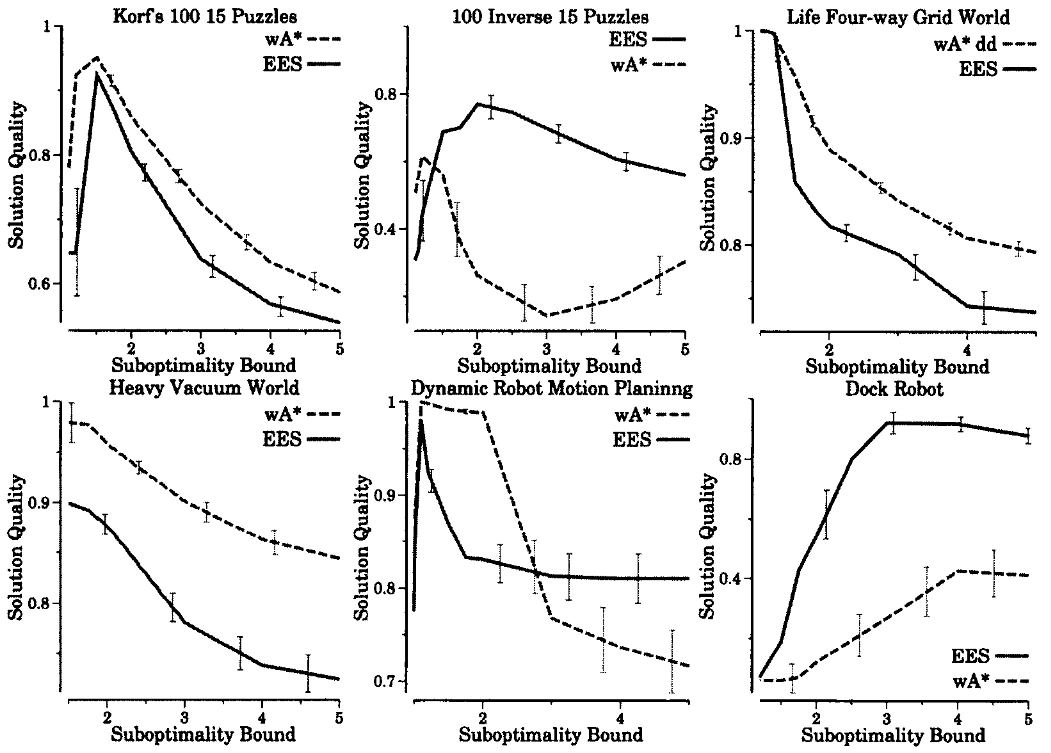


Figure 7-8: Performance of weighted A*: suboptimality bound vs. solution quality

of nodes generated during a search at a given suboptimality bound on the standard unit-cost fifteen puzzle. When we look at the two algorithms in terms of time-to-solution, as in Figure 7-5, we see that the two algorithms are barely distinguishable from one another, with weighted A* having a slight advantage for high suboptimality bounds. However, if we look at the results in terms of states generated, then EES has a clear and consistent advantage over weighted A*. This demonstrates the importance of search overhead. In domains like the sliding tiles puzzle, where it is not uncommon for a well tuned implementation to generate millions of nodes per second. The overhead of a search algorithm can be a determining factor in performance. When generating nodes becomes expensive, as it is in domain independent planning for example, algorithm overhead becomes less of a determining factor.

7.3.5 Solution Cost vs Suboptimality Bound

Another performance metric we might care about for bounded suboptimal search algorithms is how cheap the returned solution is. While all solutions returned by bounded suboptimal search algorithms are, by definition, provably within a pre-specified factor of the optimal cost solution, often they are better than this bound may imply. To measure this, we use *solution quality*, a standard metric used in the international planning competition.A Figure 7-8 presents solution quality relative to the suboptimality bound. Solution quality is computed as the cost of the best known solution to the problem divided by the cost of the solution returned by the algorithm. Finding no solution has infinite cost, so this normalizes the cost of solutions between 1 (best known) and 0 (no solution returned).

In Figure 7-8 we see that the solution quality of both algorithms generally has an inverted V shape. Generally, for very tight suboptimality bounds low solution quality are reported, then solution quality increase as suboptimality is increased, eventually reaching a peak. Beyond this peak, the solution quality begins to decrease again. The initial stage of low to high solution quality is a result of moving from a suboptimality bound where weighted A* fails to solve many instances within time and memory to a suboptimality bound where it can find the solution to most of the problems under consideration. The second phase, of moving from high solution quality to low solution quality, is exactly what we should expect from a bounded suboptimal search algorithm. As the suboptimality bound is relaxed, worse costing solutions are permissible and returned because they are easier to find.

This trend isn't seen in life cost grids, where all problems can be solved by A*. Thus, there is no initial phase of low quality caused by a failure to solve algorithms. It is not present in heavy vacuums for a similar reason, we do not show suboptimality bounds below 1.5 because all algorithms fail to solve any problems here. In the inverse domain, we see that weighted A* exhibits an inverted V early on, but then becomes worse again after a second peak in solution quality. This is directly related to the u-shaped performance of weighted A* in this domain.

Comparison of solution quality between weighted A* and EES we see that, in three of the domains under investigation (the original 15 puzzle, Life grid navigation, and heavy vacuum problems) weighted A* consistently has higher solution quality than explicit estimation search. This is because EES is explicitly trying to minimize solving time by pursuing partial solutions believed to be w-admissible in order of fewest estimated actions-to-go. If there is not a direct correlation between solution length and solution cost, then searching in order of $\widehat{d}$ could lead to low quality solutions. Weighted A*, on the other hand, only ever considers cost-to-go for search guidance, so we should expect it to return solutions of high quality so long as it can solve the problem being considered.

There are three domains where the solution cost achieved by EES is not dominated by weighted A*'s. These are the inverse sliding tiles domain and dock robot domain, where EES consistently finds more solutions for every suboptimality bound than weighted A*, and the dynamic robot domain. In the latter domain, we cannot ascribe the good performance of EES to simply solving more instances. Here, the good performance is likely the result of using more accurate cost-to-go estimates than those used by weighted A*. We justify this by noting that greedy best-first search on $\widehat{h}$ finds substantially better (i.e. cheaper) solutions than those returned by greedy best-first search on h.

It is important to note that we could have made the solution quality comparison more favorable to EES by picking a different range for normalization. We compute solution quality as $\frac{g(\text{best solution})}{g(\text{my solution})}$ but $100 \cdot \frac{g(\text{best solution})}{g(\text{my solution})}$ or $100000 \cdot \frac{g(\text{best solution})}{g(\text{my solution})}$ are equally legitimate. The proper range of normalization hinges on how costly it is to have no solution to a problem. This varies from setting to setting. By virtue of relying on multiple heuristics and directly trying to minimize solving time (and by proxy memory consumption), EES will almost always solve more problems for a give setting than weighted A*. If we place a high cost on having no solution, EES is clearly the better approach. If, however having no solution is about as good as having a very costly solution, an approach like weighted A* becomes more attractive.

From the perspective of bounded suboptimal search, evaluating solution quality post-hoc isn't exceptionally useful. We can say what tends to happen, but not what will happen.

This is problematic for unknown domains and novel problem instances. If hard bounds on solution cost are required, there is an area of suboptimal search, bounded-cost search, that addresses this problem directly. We discuss these algorithms briefly in Section 5.6.

Often, people hand tune a suboptimality bound for a bounded suboptimal search until it returns solutions of sufficient quality sufficiently quickly. EES is typically faster for a given bound than other bounded suboptimal search algorithms, and as a result EES is likely to hit "sufficiently fast" with tighter bounds on solution quality than weighted A*.

7.4 Dynamically Weighted A*

Dynamically weighted A* [44] is based on the second justification for weighting in weighted A*, that weighting makes the search prefer nodes further along in the search and therefor presumably nearer to a goal. Assuming that this preferential treatment is the reason for the good performance of weighted A*, dynamically weighted A* attempts to improve upon weighted A* by giving more preferential treatment to nodes far along in the search by scaling the weight by which their heuristic is multiplied down.

The cost function for dynamically weighted A* is provided in Equation 7.2. Here, $\epsilon = w - 1$, so ϵ is the portion of the weight beyond optimal. There are, of course, many ways to write the same expression, but this one highlights that the node evaluation function of dynamically weighted A* is the evaluation function of A* plus an extra term based on the depth of a node relative to the goal depth ($scale(n)$ defined in Equation 7.1), and the maximum allowable deviation from optimal, ϵ.

We can see that as the depth of a node increases, the value returned by Equation 7.1 approaches 0, so the deepest nodes in the search have no weight applied to them, while nodes early on in the search have nearly the full suboptimality bound used ($g(n) + h(n) + (w-1) \cdot h(n) = g(n) + w \cdot h*(n)$). This means that, for dynamically weighted A* to prefer a node higher in the search tree to one lower in the tree, the higher node likely has a much lower f-value. This results in a search algorithm which is loathe to reconsider previous decisions, that is it will spend most of it's time expanding nodes deep in the search tree

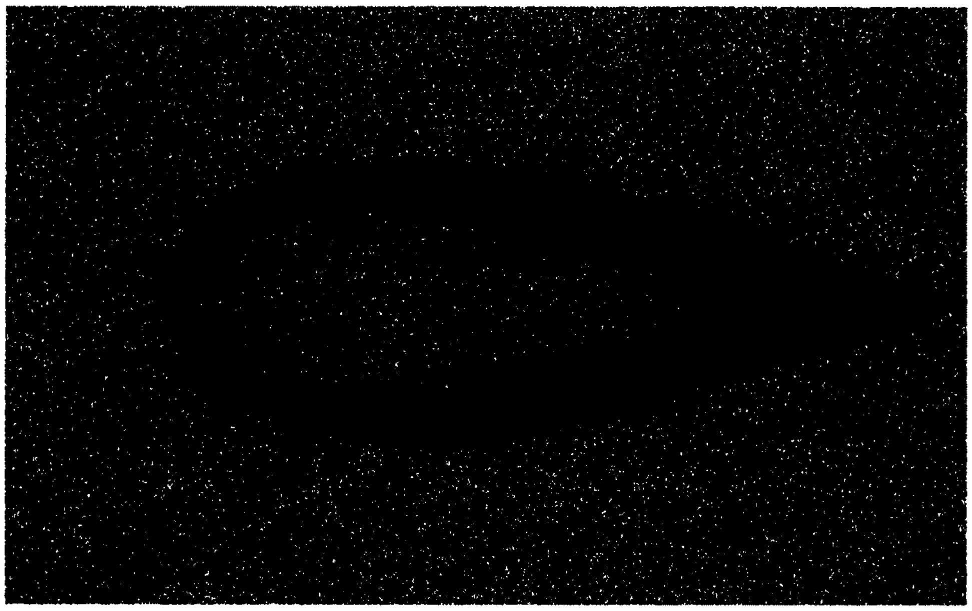


Figure 7-9: Expansion order of dynamically weighted A* search on a pathfinding problem

because they're receiving preferential treatment.

Figure 7-9 shows the expansion order of dynamically weighted A* on a unit cost grid-world pathfinding problem. The visualization shows first time a node was expanded by dynamically weighted A* search ($w = 5$). Nodes that were expanded early are colored yellow, nodes that were expanded later on are colored red. We choose to color the first time a node is expanded because dynamically weighted A* re-expands a great many nodes. You can see that the dynamically weighted expansion order is much like the A* expansion order (Figure 1-10, but shifted towards the goal, that is to the right. If we take a close look at the node evaluation function used by dynamically weighted A*, this makes perfect sense. As the depth of nodes increases, the weight applied to the heuristic increases. Eventually, the depth of nodes will exceed $MaxDepth$, and $f_{dwA*}(n)$ will be equivalent to $f(n)$.

$$scale(n) = 1 - min(1, \frac{Depth(n)}{MaxDepth}) \tag{7.1}$$

$$f_{dwA*}(n) = g(n) + h(n) + \epsilon \cdot scale(n) \cdot h(n) \tag{7.2}$$

7.4.1 Implementation Concerns

When looking at Equations 7.1 and 7.2, one might notice the *MaxDepth* value is not supplied to the algorithm. While it is not supplied, it is required to perform the weight scaling and thus the search as well. For some domains, like the traveling salesman problem, the maximum depth of a search node is known, in that case it is the number of cities. For the domains here, if we ignore cycles there is no maximum depth. If we disallow cycles, there is a maximum depth, but it is far, far larger than the depth we would expect to encounter solutions at. Having a huge *MaxDepth* relative to actual expected solution depth would cause dynamically weighted A* to behave almost exactly like weighted A*, effectively defeating the point of the algorithm. In our evaluation, we estimate the depth of the solution using $d(root)$.

7.4.2 Proof of Bounded Suboptimality

The proof of bounded suboptimality for dynamically weighted A* hinges on the fact that Equation 7.2 is bounded from below by $f(n)$ and from above by $f'(n)$. To see this, we must merely observe that Equation 7.1 only returns values between 0 and 1. Thus, when ϵ is 0, $f_{dwA*}(n) = f(n)$ and when ϵ is 1, $f_{dwA*}(n) = f'(n)$. Given this, we can use the same chain of inequalities used in our proof of Theorem 1 to show the bounded suboptimality of dynamically weighted A*. More generally, any node evaluation function obeying the inequality $f(n) \leq \widetilde{f}(n) \leq w \cdot f(n)$ can be shown to produce bounded suboptimal solutions.

Theorem 4 *A best-first search on a node evaluation function $\widetilde{f}(n)$ returns solutions within a bounded factor w of optimal so long as $f(n) \leq \widetilde{f}(n) \leq w \cdot f(n)$.*

Proof: The proof is based on the construction of the open list. Let p be the deepest node along a bath to the optimal solution. This node must exist. Initially it is the root, and when the root is expanded, it is one of the generated children. Since we never discard a node in this search, p is always on the open list. When a best-first search expands a node, we know it had the smallest node evaluation of all nodes on open. From this we can conclude:

$$\begin{array}{lll} g(sol) = f(sol) & & \text{By admissibility of } h(n) \\ f(sol) \leq \widetilde{f}(sol) & & \text{By construction of } \widetilde{f} \\ \widetilde{f}(sol) \leq w \cdot f(p) & & \text{By construction of } \widetilde{f} \\ \leq w \cdot f(opt) & & \text{By admissibility of } h \end{array}$$

□

Unlike the previous algorithm, dynamically weighted A* cannot drop duplicates if they are found along a better path even when the base heuristic is consistent. To see why this is, imagine that we had written the node evaluation function of dynamically weighted A* in this equivalent formula:

$$f_{dwA*}(n) = g(n) + w \cdot scale(n) \cdot h(n) \tag{7.3}$$

Now consider combining $scale(n)$ and $h(n)$ into a single value, $\widetilde{h}(n)$. We are then left with a weighted A* search on the new heuristic $\widetilde{h}(n)$. The new heuristic is admissible, as $\widetilde{h}(n) \leq h(n)$ because $0 \leq scale(n) \leq 1$. However, the new heuristic is no longer guaranteed to be consistent. Consider a pair of nodes, n_1 and n_2 where n_1 is the parent of n_2, the base heuristic $h(n)$ changes exactly by the cost of the transition between the two nodes $c(n_1, n_2)$, and $Depth(n_2) < MaxDepth$. For $\widetilde{h}(n)$ to be consistent, the following must be true:

$$\widetilde{h}(n_1) - \widetilde{h}(n_2) \leq c(n_1, n_2)$$

$$scale(n_1) \cdot h(n_1) - scale(n_2) \cdot h(n_2) \leq c(n_1, n_2)$$

$$(1 - \tfrac{Depth(n_1)}{MaxDepth}) \cdot h(n_1) - (1 - \tfrac{Depth(n_2)}{MaxDepth}) \cdot h(n_2) \leq c(n_1, n_2)$$

$$h(n_1) - \tfrac{Depth(n_1)}{MaxDepth} \cdot h(n_1) - h(n_2) + \tfrac{Depth(n_2)}{MaxDepth} \cdot h(n_2) \leq c(n_1, n_2)$$

$$c(n_1, n_2) - \tfrac{Depth(n_1)}{MaxDepth} \cdot h(n_1) + \tfrac{Depth(n_1)+1}{MaxDepth} \cdot h(n_2) \leq c(n_1, n_2)$$

$$c(n_1, n_2) - \tfrac{Depth(n_1)}{MaxDepth} \cdot h(n_1) + \tfrac{Depth(n_1)}{MaxDepth} \cdot h(n_2) + h(n_2) \leq c(n_1, n_2)$$

$$\tfrac{Depth(n_1)}{MaxDepth} \cdot (h(n_2) - h(n_1)) + h(n_2) \leq 0$$

$$h(n_2) - \tfrac{Depth(n_1)}{MaxDepth} \cdot c(n_1, n_2) \leq 0$$

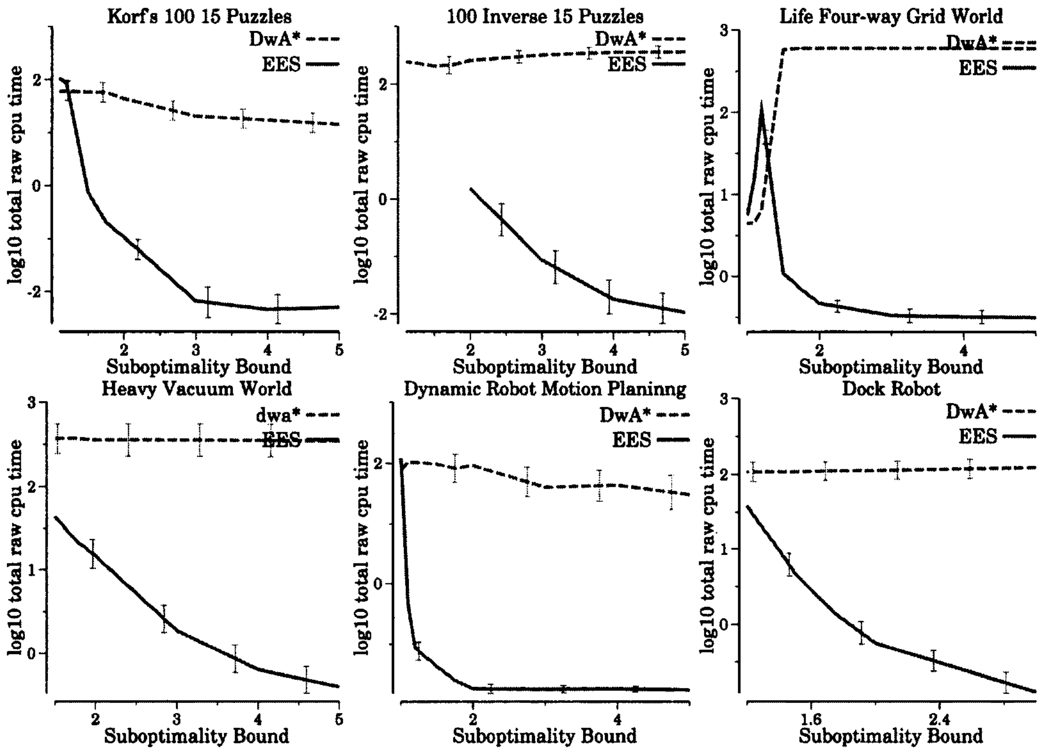


Figure 7-10: Performance of dynamically weighted A*: suboptimality bound vs. solving time

If $h(n_2)$ is ever larger than the cost of transitioning between n_1 and n_2, $\widetilde{h}$ could violate the inequality and thus be inconsistent. Since the consistency and admissibility of $h(n)$ don't guarantee this property, we cannot guarantee that $\widetilde{h}$ will be a consistent heuristic, therefore dynamically weighted A* cannot ignore duplicate states arrived at via duplicate paths without having an immense negative impact on the suboptimality bound [16].

7.4.3 Solving Time vs Suboptimality Bound

Figure 7-10 shows the time required by dynamically weighted A* to find a solution as a function of the suboptimality bound supplied to the algorithm. We also place EES on the plots for reference. Results for the heavy vacuum domain are omitted because dynamically weighted A* solved no problems for any of the suboptimality bounds considered. The plots reveal that for no suboptimality bound and for no domain is dynamically weighted

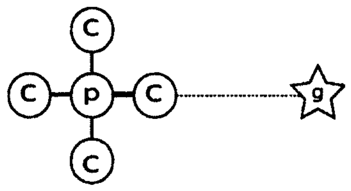


Figure 7-11: Moving away from the root is not the same as making progress towards a goal

A* competitive with EES. In fact, for no suboptimality bound is dynamically weighted A* competitive with weighted A*. Part of this is that the domains we consider in our evaluation don't have known solution depths, and so *MaxDepth* must be estimated. If we have estimates of *MaxDepth* that are too conservative, dynamically weighted A* will spend much of it's time doing a mini-A* search near the goal state. If they are too large, dynamically weighted A* will spend too much time exploring depths where no solutions exist. Even if we pick *MaxDepth* well, dynamically weighted A* will be running an A*-like search as it approaches the goal, and like A*, these searches will be expensive.

7.5 Revised Dynamically Weighted A*

Dynamically weighted A* [44] is built around the idea of rewarding progress away from the starting node of the search space. And there are domains, such as the traveling salesman problem or the knapsack problem where this is exactly the right thing to do. In these domains every step away from the root is a step towards some goal, and so dynamically weighted A* is always rewarding progress towards a goal. However in many domains we can make steps away from the goal. Consider a single expansion in a completely empty 4-connected grid, shown in Figure 7-11, world where the agent has the same y-coordinate as the goal, but is still to the left of the goal. When we expand the root of this problem, we generate four children, only one of which is actually closer to the goal. Despite only one child making any real progress, all four children have the same depth. Dynamically weighted A* will give the same preferential treatment to all children, even those that moved away

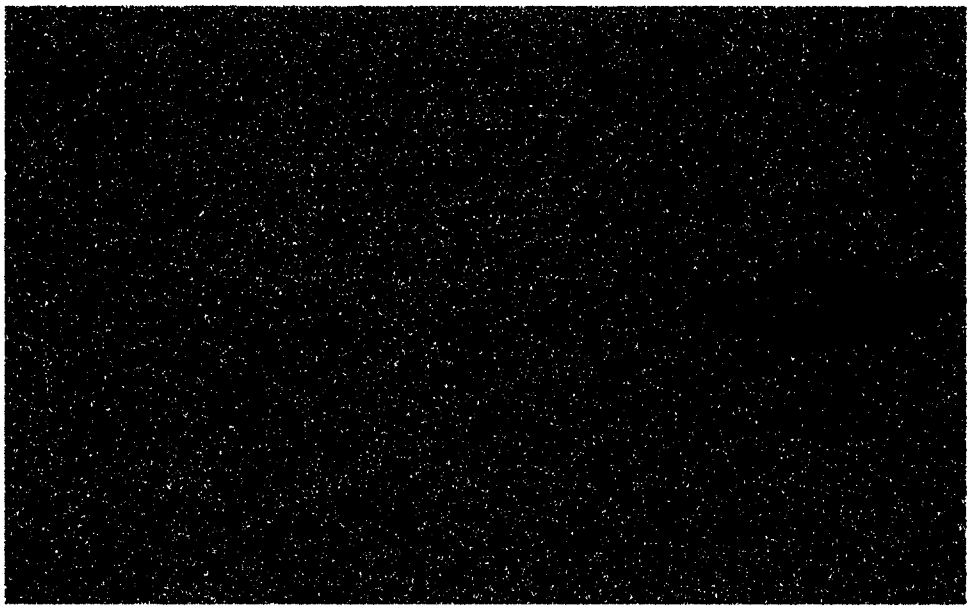


Figure 7-12: Expansion order of revised dynamically weighted A* search on a pathfinding problem

from the goal.

$$scale_{rdwA*}(n) = max(1, \frac{d(n)}{d(root)}) \tag{7.4}$$

To try and correct for this degenerate behavior, we proposed revised dynamically weighted A* [66, 71]. Revised dynamically weighted A* scales the heuristic values based on estimated distances to the goal rather than the depth of the node as we see in Equation 7.4. If $d(n)$, an estimate of the length of a cost-optimal path beneath node n, is accurate then revised dynamically weighted A* will only reward progress towards a goal instead of rewarding all movement away from the root. We will see in the evaluation that this results in substantially improved performance over dynamically weighted A*.

$$f_{rdwA*}(n) = g(n) + h(n) + \epsilon \cdot scale_{rdwA*}(n) \cdot h(n) \tag{7.5}$$

Figure 7-12 shows the expansion order of Revised Dynamically Weighted A* (RDwA*) search for the same suboptimality bound used by Dynamically weighted A* search in Figure 7-9 ($w = 5$). We can see that while RDwA* does perform a mini-A* search near the goal, the size of this search is much smaller than that of the previous dynamic weighting scheme. This is because the revised dynamic weighting recognizes that not all progress away from the root is progress towards a goal. By recognizing that nodes at the same depth may represent solutions of radically different costs, we end up with a much improved expansion order as we see here and in the empirical evaluation.

7.5.1 Proof of Bounded Suboptimality

The proof of bounded suboptimality is identical to that of the proof of Theorem 4. Since Equation 7.4 always returns values between 0 and 1, $f_{rdwA*}(n)$ is always between $f(n)$ and $w \cdot f(n)$. Any node evaluation function obeying this inequality will produce solutions within a factor w of the optimal cost solution when used in a standard best-first search.

7.5.2 Solving Time vs Suboptimality Bound

Figure 7-13 shows the performance of revised dynamically weighted A* in terms of the time needed to find solutions relative to the suboptimality bound provided to the algorithm. EES is also included in the plots for reference. We see that revised dynamically weighted A*, while significantly improving upon dynamically weighted A*, is substantially worse than explicit estimation search for all domains and nearly all suboptimality bounds. We say nearly all suboptimality bounds because for very tight suboptimality bounds in the standard tiles problem and life grid pathfinding, revised dynamically weighted A* has performance that is marginally better than that of explicit estimation search. Even though it corrects the conflation of moving away from the root and moving towards a goal, revised dynamically weighted A* is not a competitive bounded suboptimal search algorithm.

7.6 Clamped Adaptive

Clamped adaptive [70] is very similar in spirit to weighted A*. It has the same philosophy that making the cost-to-go seem more important than the cost already incurred is likely to

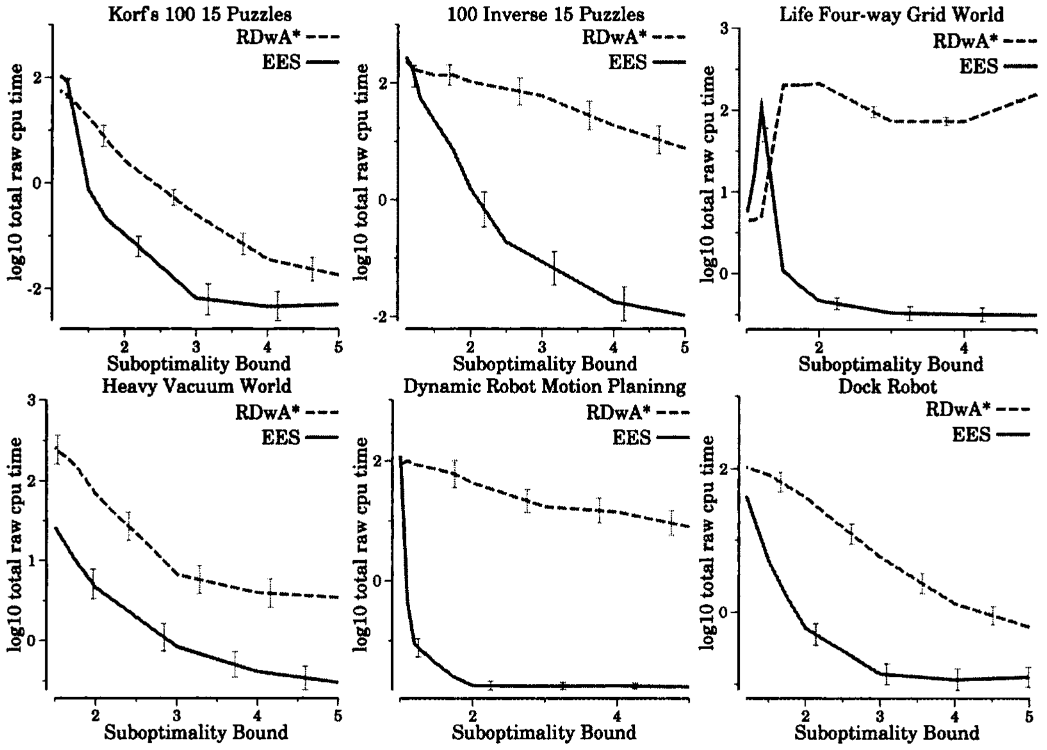


Figure 7-13: Performance of revised dynamically weighted A*: suboptimality bound vs. solving time

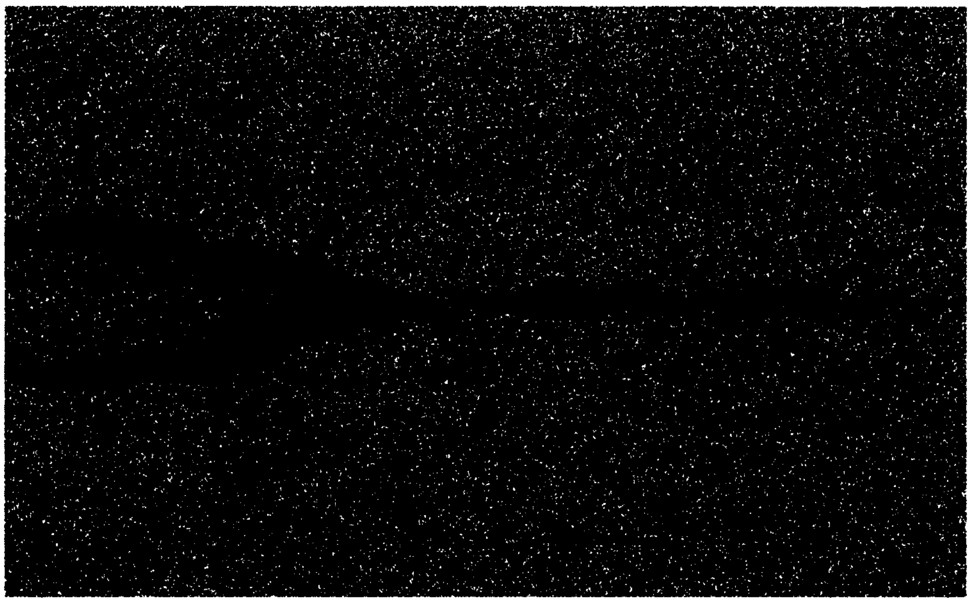


Figure 7-14: Expansion order of clamped adaptive search ($w = 1.25$) on a pathfinding problem

lead to a faster search algorithm. However it bases its search on an inadmissible heuristic $\widehat{h}(n)$ rather than on $h(n)$ as seen in Equation 7.6. This heuristic could be a hand-crafted inadmissible heuristic, or it could be learned during search.

A heuristic unfettered by the requirements of admissibility could potentially be more accurate than an admissible heuristic. This makes intuitive sense as an admissible heuristic has to deal with an unlikely best-case scenario so that it can guarantee that it will never over-estimate the true cost to go from a node, while an inadmissible heuristic could consider what is likely to work in the majority of problems even if this may occasionally over-estimate the true cost to go. Thus, it should be the case that by allowing ourselves to consider inadmissible estimates of cost-to-go we could find a more informative estimate than the base admissible cost-to-go estimate. As in EES, we refer to the inadmissible cost-to-go estimate as $\widehat{h}$. It has the same potential sources as before, and in our evaluation of clamped adaptive $\widehat{h}$ is learned online, during the course of search using single step corrections.

$$\widehat{f}'(n) = g(n) + w \cdot \widehat{h}(n) \tag{7.6}$$

One might wonder why clamped adaptive is a best-first search based on Equation 7.6 instead of a best first search on the same $\widehat{f}$ used by EES. The argument for using a weighted variant of $\widehat{f}$ is very similar to the argument against *selectNode*$_{l2}$: it never becomes sufficiently greedy. If $\widehat{h}$ were perfect, this wouldn't matter, as search on $\widehat{f}$, $\widehat{f}'$, and $\widehat{h}$ are nearly equivalent if $\widehat{h} = h^*$. However, heuristics are rarely perfect, hence the need for search algorithms. When the heuristic is inaccurate, search will not proceed directly to a goal, but it will fill in minima in the heuristic value by raising the g-value of nodes with low $\widehat{f}$ or $\widehat{f}'$ values. By placing additional emphasis on the heuristic, we can hope directly address the problem of failing to become greedy under the realistic assumption that the heuristic is imperfect.

Naturally, weighted A* run on an inadmissible heuristic is not guaranteed to return solutions within the desired suboptimality bound. This means that we cannot simply run a best-first search on $\widehat{f}'$. Instead the node evaluation function of clamped adaptive search must be slightly modified to ensure bounded suboptimality, as we see in Equation 7.7. By restricting the node evaluation function to never be larger than $w \cdot f(n)$ we will be able to prove bounded suboptimality in much the same manner that weighted A* (without duplicate dropping) proves suboptimality bounds.

$$f_{ca}(n) = max(f(n), min(w \cdot f(n), \widehat{f}'(n))) \tag{7.7}$$

There is a large potential problem with the node evaluation function proposed in Equation 7.7. If $\widehat{h}(n)$ is consistently much larger than $h(n)$, then $\widehat{f}'(n)$ will consistently be larger than $w \cdot f(n)$. In this situation, Equation 7.7 states that most nodes will be sorted in order of $w \cdot f(n)$. If all nodes are sorted in order of $w \cdot f(n)$, that is equivalent to sorting them in order of $f(n)$. Effectively, if the inadmissible heuristic consistently reports values much larger than the admissible heuristic, then clamped adaptive search converges to an A* search order, regardless of suboptimality bound. While $\widehat{f}$ is more conservative than $\widehat{f}'$, and thus less likely to revert to A* expansion order, it has problems with not becoming sufficiently greedy as the suboptimality bound is relaxed, as we just discussed.

Obviously such behavior is undesirable, but for an arbitrary inadmissible heuristic it is also unavoidable for some sets of problems. The evaluation will show that such behavior is not merely theoretical, it is experienced in practice for online heuristic corrections like those used by EES. EES avoids this problem partially by not limiting the range of values that $\widehat{f}$ can take, but as we previously noted, if $\widehat{f}$ is needlessly large this can result in too many bound-proving expansions and poor performance for EES. That is, EES may experience a similar failure relating to the relative magnitudes of $\widehat{h}$ and h, but in the case of EES it is non-catastrophic, while clamped adaptive search will revert to an A* search order.

7.6.1 Proof of Bounded Suboptimality

The cost function for clamped adaptive obeys the inequality $f(n) \leq f_{ca}(n) \leq w \cdot f(n)$ and therefor returns bounded suboptimal solutions by the same argument as the previous algorithms.

Theorem 5 *If $h(n)$ is an admissible heuristic, and $\widetilde{f}$ is an arbitrary node evaluation function obeying the inequality $f(n) \geq \widetilde{f}(n) \leq w \cdot f(n)$ then the solution returned by the algorithm has cost within a factor w of the optimal solution.*

Proof: The proof is based on the construction of the open list. Let p be the deepest node along a bath to the optimal solution. This node must exist. Initially it is the root, and when the root is expanded, it is one of the generated children. Since we never discard a node in this version of weighted A*, p is on the open list at all times, including when a solution is returned. When a best-first search algorithm on $\widetilde{f}$, that node has the smallest value of all nodes on the open list (Line 3). From this, we can conclude:

$$\begin{array}{rcll}
g(sol) = & \widetilde{f}(sol) & & \text{By admissibility of } h(n)\text{, definition of } \widetilde{f} \\
\widetilde{f}(sol) \leq & \widetilde{f}(p) & & \text{By Definition of } p\text{, best-first} \\
& \widetilde{f}(p) & \leq \; w \cdot f(p) & \text{By definition of } \widetilde{f} \\
& & \leq \; w \cdot f(opt) & \text{By admissibility of } h
\end{array}$$

□

As we can see, the proof presented here is nearly identical to the proof of bounded suboptimality for weighted A*. The node evaluation function ($f'(n)$ for weighted A*) has been replaced with a generic node evaluation function $\widetilde{f}(n)$. The definition of $\widetilde{f}$ is left open save for the fact that it is bounded from below by $f(n)$ and from above by $w \cdot f(n)$. As shown by the proof, any node evaluation function obeying such inequalities is guaranteed to provide a solution within a bounded factor of optimal. $f'(n)$ obviously obeys such an inequality, and f_{ca} does so by construction.

7.6.2 Dealing with Duplicates

Unlike weighted A*, an algorithm that may avoid re-expanding duplicate states so long as the base heuristic is consistent, clamped adaptive must re-expand duplicate nodes as they are encountered. We have no guarantee that $\widehat{h}$ is consistent. Even if we did, it isn't obvious that the resulting node-evaluation function allows for a proof similar to what we did to

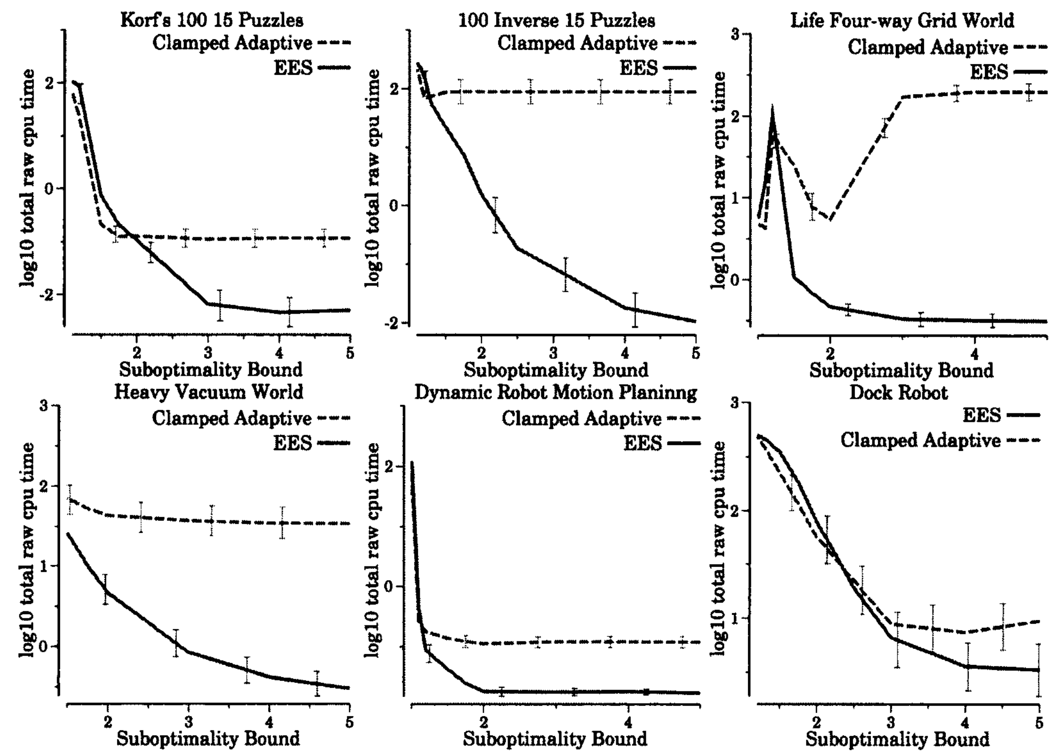


Figure 7-15: Performance of clamped adaptive: suboptimality bound vs. solving time

prove Theorem 2. As such, clamped adaptive must re-expand duplicate states encountered by a better path or suffer a sever loosening of its bounds.

7.6.3 Solving Time vs Suboptimality Bound

Figure 7-15 shows the performance of the clamped adaptive algorithm, in terms of time to solution, as a function of the suboptimality bound. EES is also included in the plot for reference. We see in the results that the behavior of Clamped Adaptive search is almost bimodal. In some domains (dynamic robots, dock robots, standard tiles) it performs relatively well, perhaps failing to become sufficiently greedy as the suboptimality bound increases. For other domains, it fails to solve a majority of the instances within time and memory for some or all suboptimality bounds. The bad behavior of clamped adaptive search can be almost entirely ascribed to the clamping performed in Equation 7.7. We previously noted that when there is a large gap between $h(n)$ and $\widehat{h}(n)$, that clamped adaptive will assign

$w \cdot f(n)$ to most nodes being considered by search, leading to a very A*-like search order. A* search is ideal for optimal search, but it is a very poor approach to finding suboptimal solutions quickly.

7.7 AlphA*

AlphA* [45] is a best first search which tries to improve the performance of search by separating nodes into two groups, good nodes who will be sorted according to their f-value, and bad nodes who will be sorted on $w \cdot f(n)$. Thus, bad nodes are maximally penalized, we cannot give them a value larger than $w \cdot f(n)$ or we could not prove bounded suboptimality. Any measurement of goodness could be used, but the paper introducing AlphA* suggests four, shown in Equations 7.10 through 7.13. In these equations, $\pi(n)$ is the parent of node n and $\hat{n}$ is the last node expanded.

$$f_\alpha(n) = w_\alpha(n) \cdot f(n) \tag{7.8}$$

$$w_\alpha(n) = \begin{cases} 1 & \alpha(n) \text{ is true} \\ w & \alpha(n) \text{ is false} \end{cases} \tag{7.9}$$

$$\alpha_g = g(\pi(n)) \geq g(\hat{n}) \tag{7.10}$$

$$\alpha_h = h(\pi(n)) \leq h(\hat{n}) \tag{7.11}$$

$$\alpha'_g = g(\pi(n)) \geq \max_{n \in closed} g(n) \tag{7.12}$$

$$\alpha'_h = h(\pi(n)) \leq \min_{n \in closed} h(n) \tag{7.13}$$

Equation 7.10 says that a node is good so long as the cost of arriving at its parent is at least as much as the cost of arriving at the last node expanded by the search. This

```
AlphA*($root, w, \alpha$)
1.  $open \leftarrow \{root\}$
2.  while($open \neq \{\}$)
3.      remove $n$ from $open$ with minimum $f_\alpha(n)$
4.      if $n$ is a goal
5.          then return $n$
6.          else expand $n$, inserting children into $open$
7.  return no solution
```

Figure 7-16: AlphA* pseudo code

function ends up encouraging progress away from the root, not unlike dynamically weighted A*. It gives the search a kind of forward momentum, because nodes with lower g-values are presumably elsewhere in the search space, and abandoning the current avenue of search is made expensive by this alpha function.

Equation 7.11 is very similar to that of Equation 7.10, except now instead of preferring nodes which have incurred lots of cost since leaving the root, we give nodes which appear to be at least as close as the last node expanded preferential treatment. If a node appears to be better (based on $f(n)$) but further away in terms of cost-to-go, then it must be much better than the last expanded node to be considered. Again, this policy is aimed at giving the search a sort of forward momentum.

Equations 7.12 and 7.13 are more aggressive version of the previous two rules in that they consider the values of all nodes ever expanded instead of just the previously expanded node. Not only are these rules stricter, they also incur less overhead. For the previous rules, the *alpha*-value of a node would change at every expansion, potentially requiring a resorting of the open list for ever node expanded.

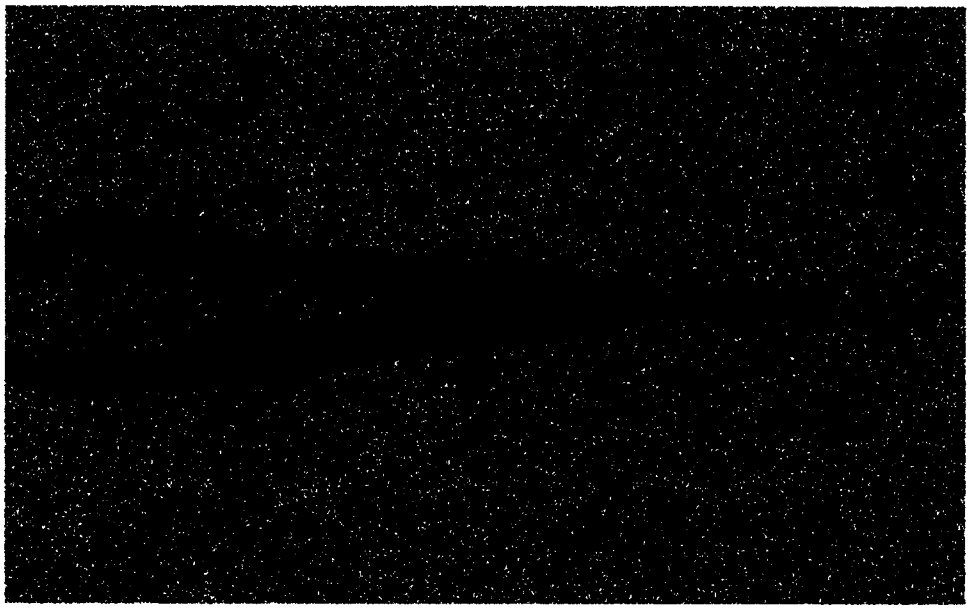


Figure 7-17: Expansion order of alpha* search on a pathfinding problem

7.7.1 Implementation Concerns

The way that f_α is defined leads to large concerns for the efficient implementation of AlphA*. The truth of $\alpha(n)$ can change from expansion to expansion for a great many nodes. If many nodes change α and thus f_α-values every expansion, we might have to resort the entire open list if it stored nodes in order of f_α.

Clearly, resorting the entire open list every expansion is impractical. The simplest way to implement AlphA* is to perform a linear scan of an open list sorted on $f(n)$. For every node, determine its α and f_α-value. The first node with $\alpha(n) = \mathit{true}$ is going to have the lowest f_α value and can be returned. If we must return some node with $f_\alpha(n) = w \cdot f(n)$, we may have to scan a large portion of the open list in order to determine that we can return this node.

Reese[45] has a yet-more-efficient way of determining the best node for an AlphA* search algorithm. In order to improve the efficiency of the above approach, AlphA* may maintain a pointer, or marker, to the first open node whose $f_\alpha(n) = f(n)$, or the first active node. When this node is selected for expansion, the pointer must be updated using a sweep of a

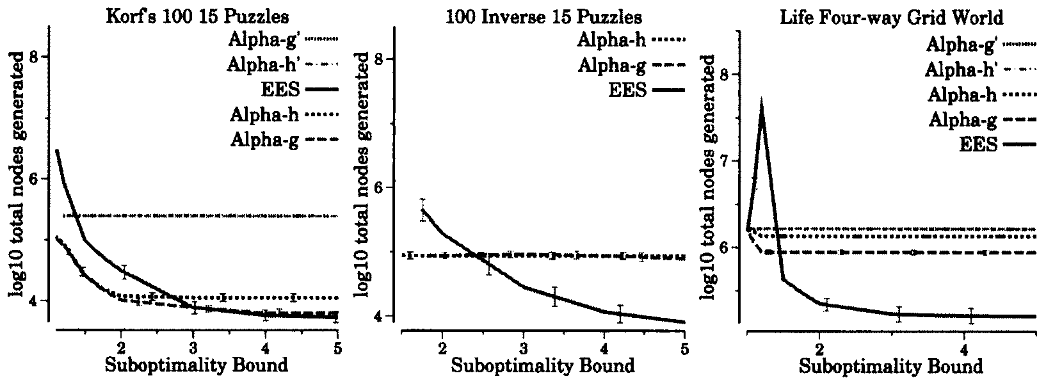


Figure 7-18: Performance of AlphA*: suboptimality bound vs. nodes generated

prefix of the open list, but maintaining and updating the marker saves time on iterations in which an active node is note expanded, reducing search overhead.

Of course, none of this really matters. Even if we could completely eliminate the overhead of AlphA*, it is not a particularly effective algorithm, at least not on the sorts of domains investigated here with the α-functions suggested by the original paper. Figure 7-18 shows a performance evaluation of selected AlphA* algorithms in terms of nodes generate. Examining algorithms in terms of nodes generated removes all over-head from the evaluation. Even absent the considerable overhead of AlphA*, it is an uncompetitive algorithm.

Figure 7-18 shows the performance of the AlphA* algorithm in terms of nodes generated while solving a problem. We report results for the three domains where AlphA* was able to solve problems within memory and the ten minute time limit. Two things become immediately apparent: α_g has the best overall performance, although it is only slightly better than α_h, and AlphA* does not perform well on domains without unit cost actions. For domains with unit cost actions, ie the standard fifteen puzzle reported in the leftmost panel of Figure 7-18, AlphA* is competitive in terms of the number of nodes generated.

7.7.2 Proof of Bounded Suboptimality

No matter what alpha is, we know that the following inequality holds:

$$f(n) \leq f_\alpha(n) \leq w \cdot f(n) \tag{7.14}$$

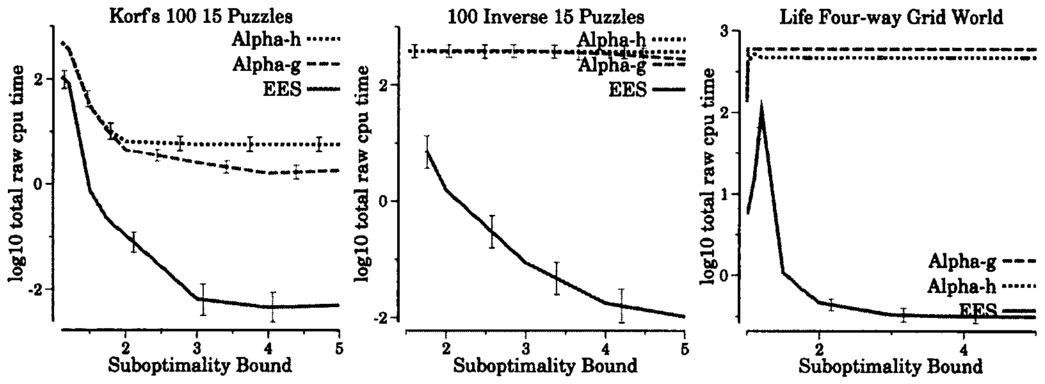


Figure 7-19: Performance of AlphA*: suboptimality bound vs. solving time

This comes from the way f_α is constructed; all of the rules are only capable of returning either $f(n)$ or $w \cdot f(n)$ for any node. Any cost function obeying this inequality can use the same proof of bounded suboptimality provided in the proof of Theorem 5. By the same line, we know that AlphA* must re-expand duplicate nodes in order to maintain it's suboptimality bounds.

7.7.3 Solving Time vs Suboptimality Bound

Figure 7-19 shows the performance of AlphA* in terms of time rather than in terms of nodes generated. Even an efficient implementation of AlphA* is dramatically slower than other search algorithms, including EES which as we previously discussed has non-negligible overhead. If we were to compare AlphA* with even more streamlined algorithms, such as weighted A*, the comparison would be even more one-sided than the one seen in Figure 7-19.

7.8 A_ϵ

A_ϵ [19] was published slightly before A^*_ϵ, but is easier to describe in terms of A^*_ϵ, so we delayed its presentation until now. It is easiest to conceptualize A_ϵ as a blend of A^*_ϵ and local search. We can see in Figure 7-20 that the pseudo-code for A_ϵ is nearly identical to that for A^*_ϵ. A_ϵ build exactly the same *focal* list as the one used by A^*_ϵ.

Where they differ is in what is done with the node selected for expansion. While

```
$A_\epsilon(root, w)$
1.  $open \leftarrow \{root\}$
2.  while $open \neq \{\}$
3.  $focal \leftarrow \{n \in open : f(n) \leq f(f_{min})\}$
4.      remove $n$ from $focal$ with minimum $d(n)$
5.      **if** $n$ is a goal return $n$
6.      **else** $pursue(n)$
7.  return no solution
```

Figure 7-20: A_ϵ pseudo code

```
$pursue(n)$
1.  **if** $f(n) > w \cdot f(best_f)$
2.      **if** $persevere(open, n)$
3.          expand $best_f$, inserting children into $open$
4.          $pursue(n)$
5.      **else** insert $n$ into $open$
6.  **else if** $n$ is a goal
7.      return $n$
8.  **else** $children \rightarrow expand(n)$
9.      $n' \rightarrow \text{argmin}_{children}\, d(n)$
10.     insert $children - \{n'\}$ into $open$
11.     $pursue(n')$
```

Figure 7-21: *pursue* subroutine of A_ϵ

```
$persevere_{A^*_\epsilon}(open, n)$
1.  return false

$persevere_{close}(threshold, open, n)$
1.  return $d(n) \leq threshold$

$persevere_{fgap}(threshold, open, n)$
1.  $best_f \rightarrow \text{argmin}_{f(n)}\, open$
2.  $best'_f \rightarrow \text{argmin}_{f(n)}\, open - \{best_f\}$
3.  return $f(best'_f) - f(best_f) \geq threshold$

$persevere_{fgap'}(threshold, open, n)$
1.  $best_f \rightarrow \text{argmin}_{f(n)}\, open$
2.  return $\frac{f(n)}{f(best_f)} \leq threshold$
```

Figure 7-22: Possible persevere predicates

A^*_ϵ would simply expand the node and insert it's children into the *open* list, A_ϵ invokes the *pursue* function, shown in Figure 7-21. *pursue* is essentially a small local search run starting from every node selected for expansion. This search hill-climbs on $d(n)$ (lines 5 and on) so long as the next state can be shown to have cost within a bounded factor w of optimal (line 1). If the node that hill-climbing would like to expand cannot be pursued because we would lose our guarantees of bounded suboptimality, A_ϵ might still pursue this node by choosing to persevere. In this case, that means attempting to raise the lower bound on solution cost until the node given to *pursue* can be shown to have cost within a bounded factor w of optimal (lines 1 through 4).

persevere largely dictates the performance difference between A_ϵ and A^*_ϵ, as it determines to what extent A_ϵ will use its local search behavior. We replicate the predicates suggested by Ghallab and Allard[19] in Figure 7-22, and add an additional predicate that further highlights the similarity between A^*_ϵ and A_ϵ. $persevere_{A^*_\epsilon}$ which A_ϵ to A^*_ϵ. If we never attempt to raise $f(best_f)$ in order to extend our local from the node selected for expansion, A_ϵ will behave exactly as A^*_ϵ.

The other suggested *persevere* rules are designed to actually allow A_ϵ to do some limited amount of local search depending on the situation. $persevere_{close}$ allows for hill-climbing if a node is estimated to be sufficiently close to being expanded into a goal. $persevere_{fgap}$ and $persevere_{fgap'}$ take the point of view that hill-climbing should be continued if raising the lower-bound on optimal solution cost so that the node can be expanded while maintaining a bound won't be too expensive. Where they differ is in their definition of too expensive. $persevere_{fgap}$ looks at the potential difference that could be gained as a result of a single expansion of $best_f$, while $persevere_{fgap'}$ looks at the relative difference between n and $best_f$ to determine if we should persevere. Note that, with the exception of the rule reducing A_ϵ to A^*_ϵ, all of the suggested rules require an additional parameter, which if selected improperly could prove disastrous for performance.

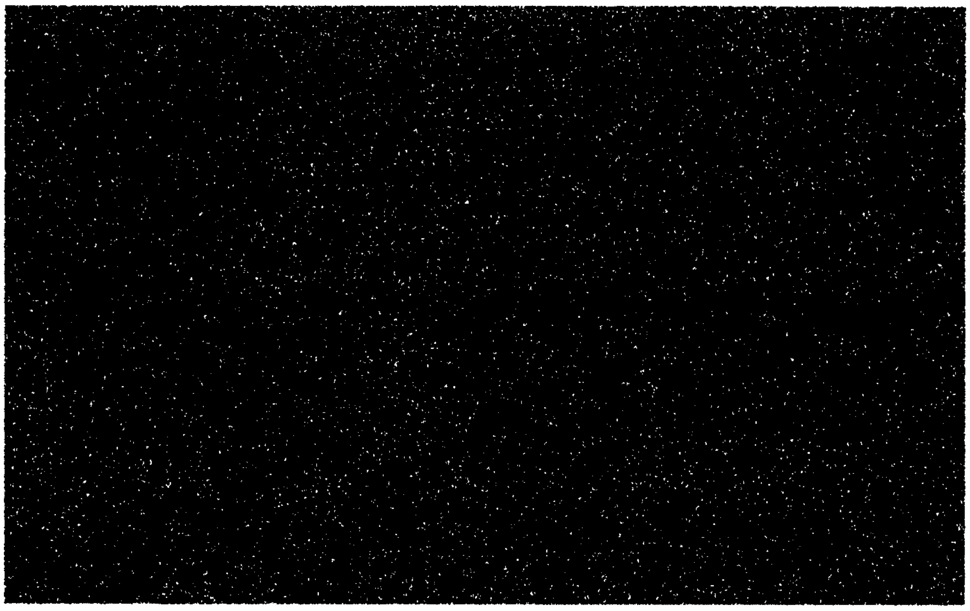


Figure 7-23: Expansion order of A_ϵ search on a pathfinding problem

7.8.1 Proof of Bounded Suboptimality

The proof of bounded suboptimality for A_ϵ follows the same line of reasoning as that for A_ϵ^* and EES. Anytime a node is expanded, we know that $f(n) \leq w \cdot f(best_f)$. Since $f(best_f)$ is a lower bound on the cost of an optimal solution to the problem, we know that the cost of any returned solution will be within a bounded factor w of optimal.

7.8.2 Solving Time vs Suboptimality Bound

Figure 7-24 shows the performance of A_ϵ on the three benchmark domains where it was able to solve problems within memory or ten minutes: both examined variants of the tiles puzzle and life-cost grid navigation problems. We show results for the best performing perseverance rule, that of using a relative threshold to deciding when to persevere. We can see from the plots that, even for the best performing rule, A_ϵ has poor performance. For small suboptimality bounds, where bounded suboptimal search is most like A*, it has performance that is on par with or, in tiles, slightly better than, EES. However, once the bound is loosened and A_ϵ is allowed to search through nodes in the order of its choosing,

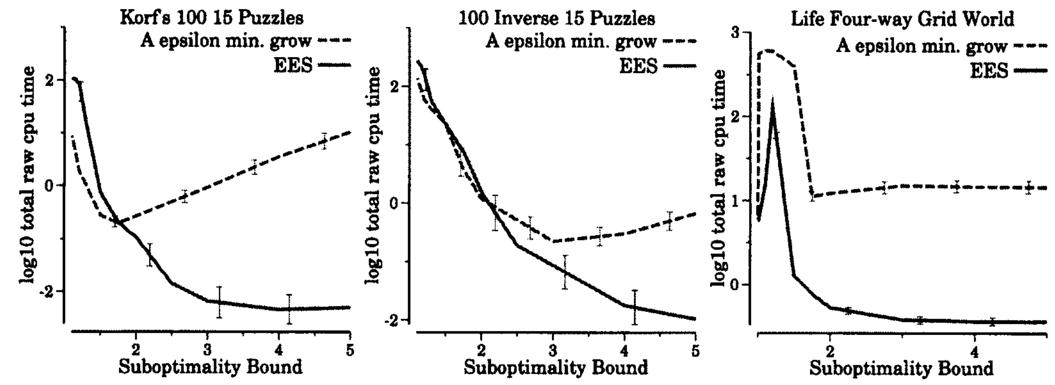


Figure 7-24: Performance of A_ϵ on three benchmark domains

its performance becomes far worse than that of EES.

7.9 A_ϵ^*

A_ϵ^* [42] is a bounded suboptimal search algorithm that seeks to find solutions of bounded suboptimality as quickly as possible by constructing a subset of all the nodes that could be considered by search, and expanding only nodes out of this subset. As we have previously noted, creating a subset of the nodes for consideration can speed up search by reducing the size of the space that needs to be considered and by reducing overhead in key data structures for a search algorithm. In A_ϵ^*'s case, it is creating a subset of all nodes that, if expanded at the current time, could be shown to lead to a solution within the desired suboptimality bound. Of these nodes, it expands the node that is estimated to be closest to a goal, based on $d(n)$. Pseudo code for this algorithm is provided in Figure 7-25.

A_ϵ^* is the previously proposed algorithm that is most similar to explicit estimation search, as is obvious from the algorithm description. The key distinction between the two approaches is that EES builds a subset of all search nodes it estimates to lead to solutions that are within a bounded factor w of optimal, while A_ϵ^* builds a subset of all nodes it can prove lead to a solution within a bounded factor w of optimal. More directly, EES uses $\widehat{f}$ to determine if a node is included on *focal*, while A_ϵ^* uses f. As we will see, this leads A_ϵ^* to have less than ideal performance in practice.

```
$A^*_\epsilon(root, w)$
1. $open \leftarrow \{root\}$
2. while $open \neq \{\}$
3. $focal \leftarrow \{n \in open : f(n) \leq f(best_f)\}$
4.     remove $n$ from $focal$ with minimum $d(n)$
5.     if $n$ is a goal
6.         then return $n$
7.         else expand $n$, inserting children into $open$
8. return no solution
```

Figure 7-25: A^*_ϵ pseudo code

The reason A^*_ϵ can perform poorly in practice is as follows: When using an admissible cost-to-go estimate, the f-values of nodes cannot decrease, and typically increase, as the search proceeds outward from the root. In contrast, along a path towards a goal, the d-values of nodes will usually decrease. Thus, nodes with low d-values will often have relatively high f-values.

This is not, in of itself, a problem. However, it leads to a behavior we refer to as thrashing. Let the best node on A^*_ϵ's *focal* list, that is the node with the smallest d-value, be $best_d$. Because f-values tend to rise as nodes move away from the root of the search problem, it is often the case that nodes with low d-values have higher f-values. As a result, $best_d$ often has an f-value that is so high that it only barely qualifies for inclusion into the *focal* list, while the node with the smallest f-value is all the way at the end of *focal*. When $best_d$ is expanded, its children will often have higher f-values than it did. As a result, they may no longer qualify for inclusion in focal until $best_f$ is expanded and $f(best_f)$ raises. $best_f$ will be expanded and $f(best_f)$ will raise, but generally only after all of the other nodes on *focal* have been expanded. A more thorough discussion of this phenomena is available in

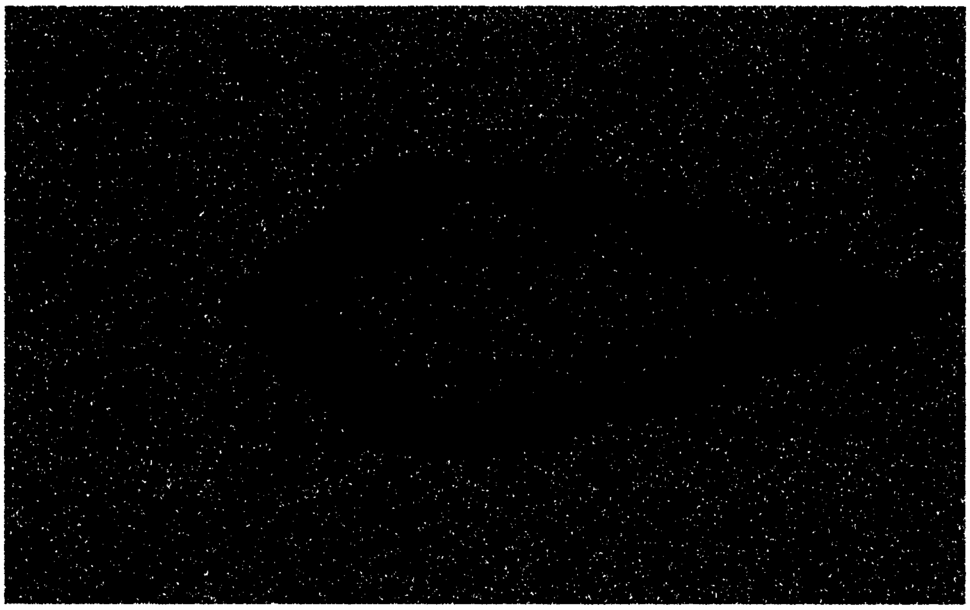


Figure 7-26: Expansion order of A^*_ϵ search on a pathfinding problem

Thayer et al[71].

7.9.1 Implementation Concerns

Open and *focal* are separate lists, at least conceptually. There are several ways we could build them, but an inefficient implementation will harm the performance of the algorithm. One might consider only maintaining the open list, and iterating through the first handful of nodes on every expansion to select d_{min}. Unfortunately when the bound is loose such an algorithm would be examining every node in open at every expansion. Alternatively, we might keep both open and a list of all nodes ordered on d in memory, iterating back on this d-list until a node near enough to $best_f$ is discovered, but again, this is inefficient.

To make A^*_ϵ a practical algorithm, we use a more sophisticated data structure. Nodes in the open list are stored in a balanced binary tree totally ordered by f. In our implementation, we used a red-black tree following Cormen, Leiserson, Rivest, and Stein[10]. Those nodes within ϵ of the node with minimum f are also stored in a heap ordered on d. We used a binary heap stored in an array, following Sedgewick [60]. Using this arrangement, it takes constant time to identify the node to expand, logarithmic time to remove it from the heap and tree, and logarithmic time to insert each child resulting from the expansion. However, if the node with minimum f changes, then nodes may need to be added or removed from the heap. (All nodes are stored in the tree.) While it is easy to find the nodes whose f-values fall between w times the old minimum f and w times the new one (because the tree is ordered on f), there might be many such nodes that need to be added or removed from the heap. Removal is easy because we maintain, in each node, its index in the heap array. Using this more sophisticated data structure speeds up A^*_ϵ in practice by an enormous factor which increases as problem become more difficult.

7.9.2 Proof of Bounded Suboptimality

The proof of bounded suboptimality is identical to that for optimistic and skeptical search. At the time a solution is returned in Line 5 of Figure 7-25 we know that the cost of that solution is within a bounded factor w of $f(best_f)$ by the construction of the focal list (Line 3). Since $f(best_f)$ is a lower bound on the cost of an optimal solution, we know that the solution returned by A^*_ϵ is within a bounded factor of optimal. Conceptually, it is identical to the proof of bounded suboptimality for EES.

7.9.3 Solving Time vs Suboptimality Bound

Figure 7-27 shows the performance of A^*_ϵ as a function of the provided suboptimality bound in terms of the time needed to find a solution. Generally, A^*_ϵ performs worse than or is at best competitive with EES in terms of time to a solution. We will shortly see that EES, however, has consistently better solutions. Whenever A^*_ϵ does outperform EES, it is for large suboptimality bounds. We can see this when we look at life cost grids, heavy vacuum problems, or large inverse cost tiles problems. Here, for suboptimality bounds larger than 3, A^*_ϵ finds solutions faster, on average, than EES. Interestingly, high suboptimality bounds are exactly the bounds where the detrimental thrashing behavior of A^*_ϵ does not occur. For tighter bounds, where thrashing is a problem, we see remarkably poor performance for

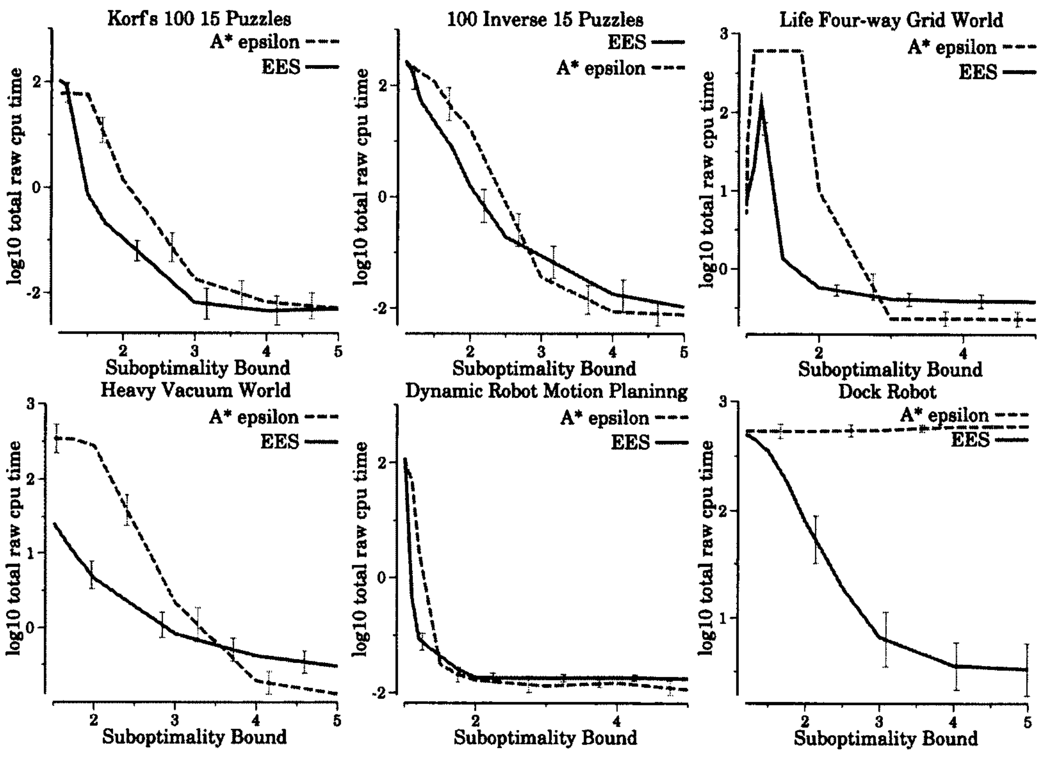


Figure 7-27: Performance of A_ϵ^*: suboptimality bound vs. solving time

A_ϵ^*. For example, for $1 \leq w \leq 2.5$, A_ϵ^* fails to solve most instances of the grid pathfinding problem, exhausting memory or timing out at ten minutes. For reference, the mean A* time is about 4.5 seconds, so A_ϵ^* is a full two orders of magnitude slower than optimal search in some cases.

In Figure 7-28 we see that a portion of the competitive behavior of A_ϵ^* can be ascribed to its reduced per-node overhead when compared to EES. In Figure 7-28 we show the performance of algorithms as measured by the number of nodes generated while solving a problem. This removes search overhead from the consideration. We show three domains where the performance of A_ϵ^* and EES was closest, the inverse cost fifteen puzzle, life cost grid-world pathfinding, and dynamic robot motion planning. When we remove search overhead from consideration, A_ϵ^* no longer has a clear advantage over EES for high suboptimality bounds. The confidence intervals of the two lines overlap strongly for all domains. EES must maintain an additional sorting over the nodes to perform search. The cost of maintaining this

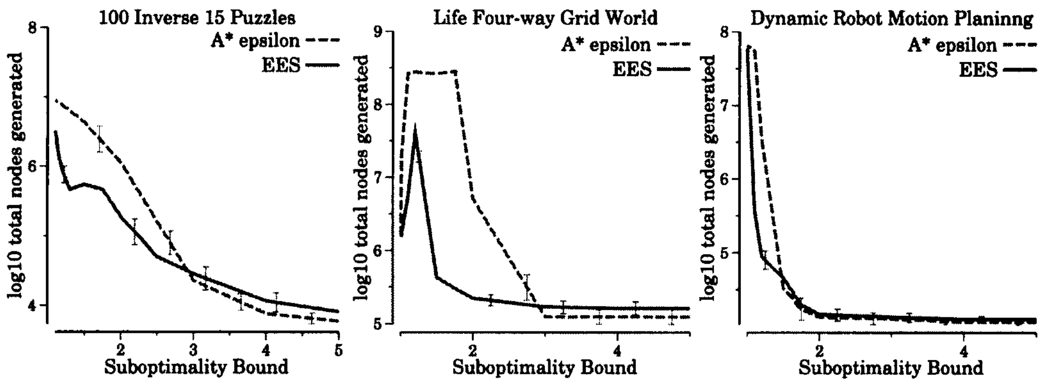


Figure 7-28: Performance of A_ϵ^*: suboptimality bound vs. nodes generated

open list is well worth it, as it helps EES avoid the thrashing problem experienced by A_ϵ^*, but for high suboptimality bounds where thrashing is not experienced, it can lead to marginally worse performance.

7.10 Optimistic Search

Optimistic search is based on the following observation of the performance of weighted A*: weighted A* often returns a solution much better than the bound would imply. Consider the proof of weighted A*'s suboptimality bound presented in Proof 7.3.2. In the third line of this proof, we state that, by algebra, $g(sol) \leq f'(p) \leq w \cdot f(p)$. Essentially, we are weighting $g(p)$ by a factor w that was not present in the node evaluation function f'. This introduces a looseness in the provable suboptimality bound for weighted A* that often allows this algorithm to return solutions much better than the suboptimality bound suggests.

We show an example of this behavior in one domain in Figure 7-29. Here, the x-axis is the suboptimality bound (or weight) weighted A* was run with and the y-axis represents actual solution quality, computed by solving the instance optimally and then dividing the cost of the solution returned by weighted A* by that of the optimal solution. To show the bound, we draw the line $y = x$.

As we see in the left panel of Figure 7-29, we should often be able to run weighted A* with a weight much higher than the desired suboptimality bound and still be able to

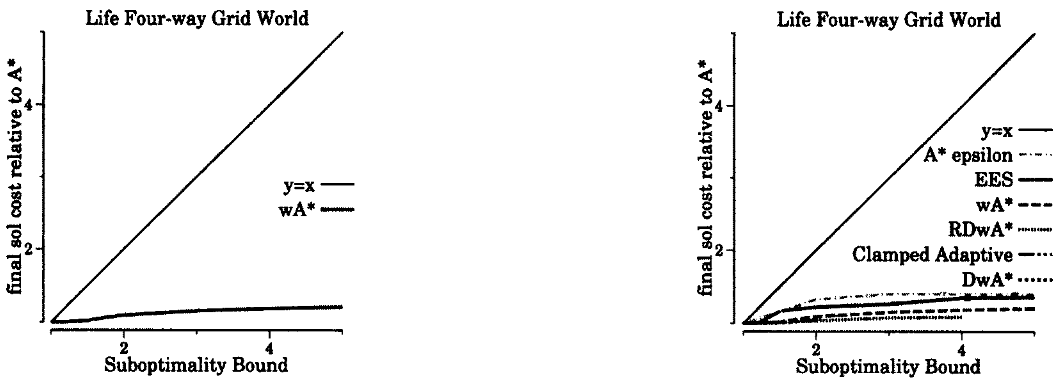


Figure 7-29: Actual solution quality versus bound in bounded suboptimal search

find a solution within the bound. However, the guarantee of bounded suboptimality for weighted A* is based on the supplied weight as we saw in the proof of Theorem 1. So even if the returned solution is likely to be within the bound, we won't know for certain, and the algorithm would no longer be a bounded suboptimal algorithm[2]. In order to provide a bound on the quality of solutions returned by such a search, we would need to find a way of proving the quality of solutions that was independent from the search order that generated those solutions. The right panel of Figure 7-29 shows that, although optimistic search was proposed with weighted A* in mind, a similar approach is still valid for any of the bounded suboptimal search algorithms investigated here, even EES.

Since we cannot prove the quality of the solution in the first phase of search, when we find the solution, we will have to prove the quality of the solution using an additional phase of search. In their paper on anytime heuristic search, Hansen and Zhou [21] point out that, if we are not discarding duplicate states, the node on the open list with the smallest f-value acts as a lower bound to the cost of an optimal solution. Thus, we can compute the quality of an incumbent solution by dividing its cost by the smallest f-value on open as in $\frac{g(inc)}{f(best_f)}$. We can determine which node has the smallest f-value among all nodes by either

[2]There is some work concerning search algorithms that provide probabilistic bounds, where the solution is within the desired bound with some probability. Such algorithms have different applications that the ones discussed here and are thus outside the scope of this work.

performing a linear scan of the open list or maintaining a separate, synced priority queue sorted in order of increasing f-value. We take the latter approach and refer to this set of nodes as the cleanup list.

This only shows how we can compute the bound on the current incumbent solution. If the incumbent is within the bound, then we could simply return it, but the more interesting case is the one where we cannot immediately show that the incumbent is within the bound. In this case, there are two reasons why the solution might not appear to be within the bound. Either the solution isn't within the desired bound, or the solution is within the bound but our lower bound on optimal solution cost is not tight enough to prove that the incumbent is indeed within the bound. In the former case, we must abandon our current solution and attempt to find a new one, in the latter, we must merely raise the lower bound.

To raise the lower bound, we need to increase the f-value of $best_f$. The most direct way to do this is by expanding $best_f$ and inserting its children into both the open and cleanup lists. Since the heuristic used for computing f-values is admissible, expanding the node with minimum f-value will either leave that value the same or will increase it. Eventually, if the solution is within the desired bound, expanding $best_f$ will raise the minimum f-value to the point where we can show the incumbent solution is within the desired bound.

If the solution wasn't within the bound, expanding $best_f$ will eventually generate a cost-optimal solution that is guaranteed to be within the bound. However, such a search would, in terms of nodes evaluated, be less efficient than simply running A* in the first place. It might be faster in terms of CPU time because of reduced overhead from having an incumbent for pruning. In order to avoid this degenerate behavior, optimistic search implements an 'escape hatch' which fires when it appears that there is a solution better than the incumbent. Specifically, in lines 13-16 of Figure 5-4, we see that optimistic search may pursue a new incumbent solution if its f'-value appears to be less than the cost of the incumbent. Such a node would have to have an f-value smaller than the cost of the incumbent solution, so it could potentially lead to a better solution. In practice, these rules are rarely, if ever, used. For a node to be expanded by these rules, it must first be generated

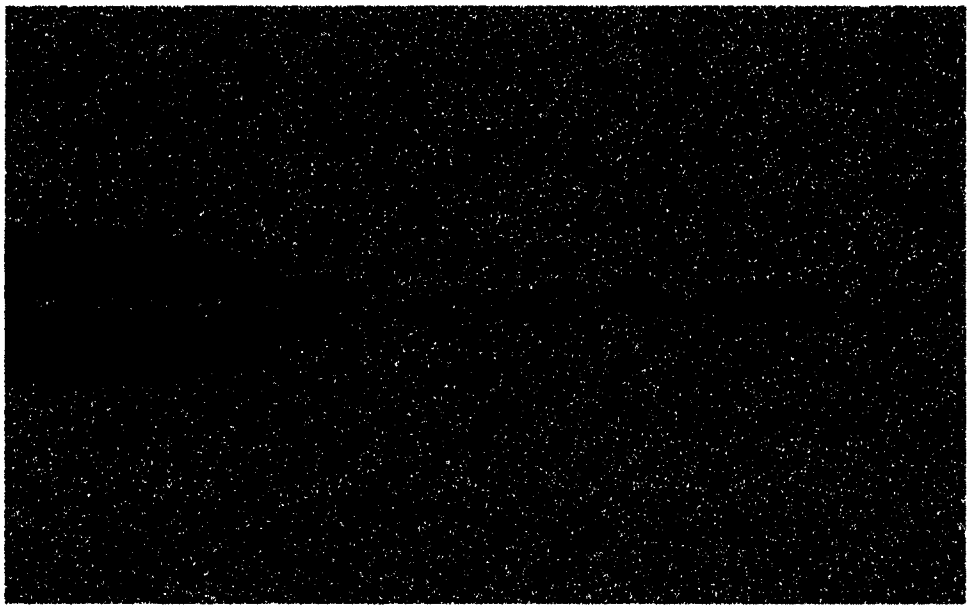


Figure 7-30: Expansion order of optimistic search (w = 1.5, *opt* = 2) on a pathfinding problem

by an $best_f$ expansion, otherwise it would have been expanded before an incumbent was found in lines 1–7. In practice, we prove the quality of a solution long before such a node becomes a candidate for expansion in line 13. If w and b are selected such that the solution initially found is outside of the bound, these rules will be used.

Figure 7-30 shows a visualization of the expansion order of optimistic search on a unit cost grid pathfinding problem. This particular visualization shows the first time a node was expanded by search (recall that optimistic search may need to re-open nodes). Nodes that were expanded for the first time early on are colored yellow, and as the color of a cell approaches red, that node was expanded later on in the search. Nodes that were untouched remain white, and obstacles on the grid are colored in black. In the visualization we can see the two phases of optimistic search, the greedy pursuit of the goal and the cleanup phase. We visualize each phase of optimistic search separately in Figure 7-31 and Figure 7-32.

The first phase, lines 3 through 7 of Figure 5-4 and shown in Figure 7-31 is the search for a first solution, when the incumbent solution is found. This shows optimistic search run

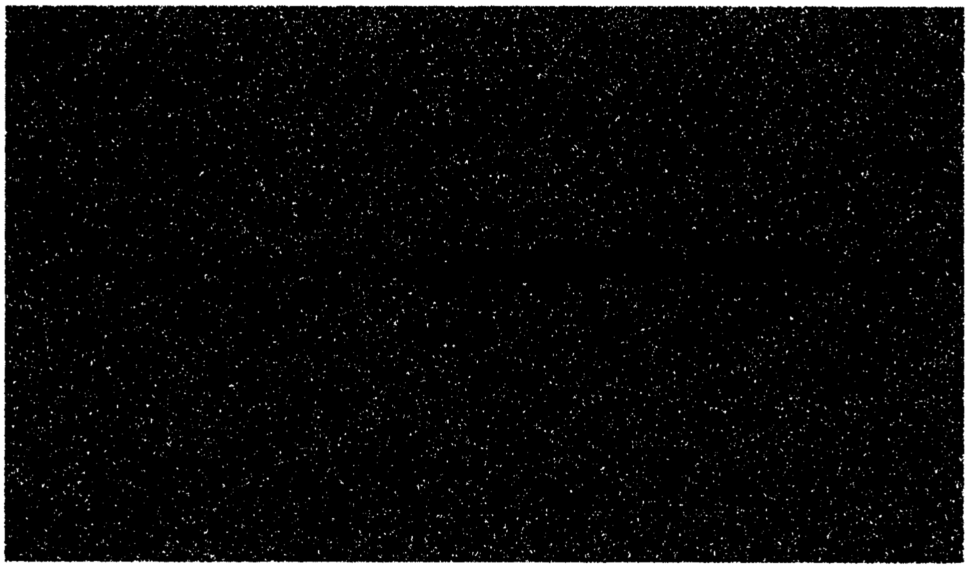


Figure 7-31: Initial phase of optimistic search ($w = 1.5$, $opt = 2$) on a pathfinding problem

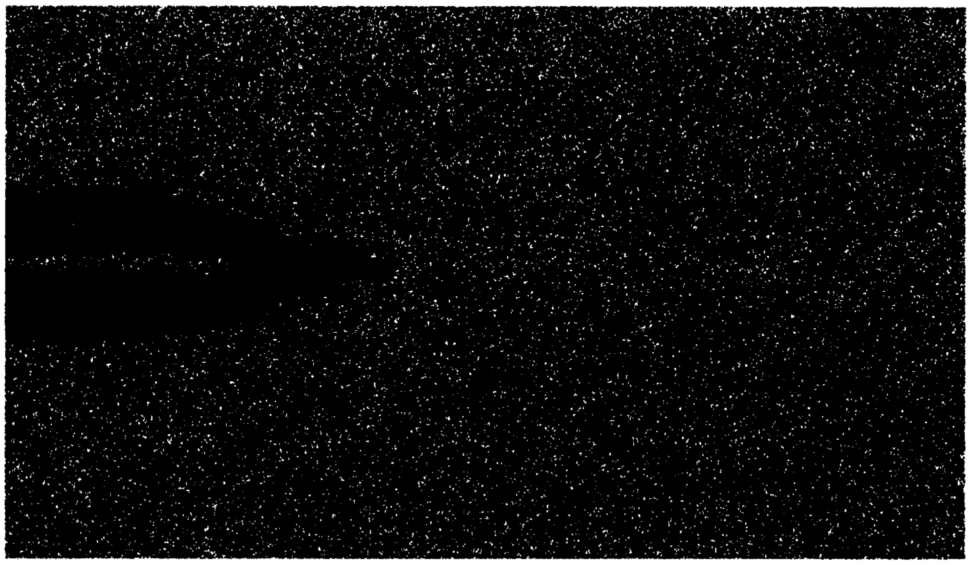


Figure 7-32: Cleanup phase of optimistic search ($w = 1.5$, $opt = 2$) on a pathfinding problem

with a suboptimality bound of 1.5 and and optimism of 2, which means that we're seeing the same expansion order as weighted A* run with a weight of 2. Optimistic search runs weighted A* with a weight that is optimism times as generous as the suboptimality bound, which means we take the portion of the weight beyond optimal, and multiply by optimism (ie $(1.5 - 1) \cdot 2 + 1$).

The second phase, the red colored onion around the beginning of the search space, is where the quality of the solution found by the first phase of search is proved. While in Figure 7-30, the red bulb did not form a closed region, here it does. That is because previously we were only showing the first time a node was expanded, now we are showing the first time a node was expanded during the cleanup phase. Some of the nodes expanded in the first phase shown in Figure 7-31 are re-expanded in the cleanup phase shown in Figure 7-32. These nodes must be re-opened to prove the bound later on.

If we compare the expansion order of optimistic search (Figure 7-30) with that of EES (Figure 7-40), we notice that the two algorithms expand a nearly identical set of nodes, but they do so in opposite orders. Optimistic search expands a thin strip of nodes leading towards the goal, and then expands a set of nodes near the root to prove that the returned solution was indeed within the bound. EES on the other hand, expands a small set of nodes near the root in order to raise the lower-bound on optimal solution cost. It then expands a thin strip of nodes from the tip of this A*-like initial phase into the goal. Because EES expands the cleanup nodes first, in A* order, it will often have to do less repeated work than optimistic search.

7.10.1 Implementation Concerns

An efficient implementation of optimistic search requires us to have easy access to both $best_{f'}$ and $best_f$. This suggests two open lists, one sorted on f'-values, and the other sorted on f-values. We refer to these as *open* and *cleanup* respectively. Although *open* must exist for the entire lifetime of the search, *cleanup* is only important after an initial solution is found. Thus, *cleanup* should only be constructed once the first solution is found by iterating over all nodes in *open*, discarding those nodes with $f(n) \geq g(incumbent)$, and inserting all other nodes into *cleanup*. Once *cleanup* is constructed, search can continue. Keeping *cleanup* and *open* synchronized after this step, and performing the pruning on the incumbent solution while building *cleanup* require data structures that support arbitrary element removal.

7.10.2 Proof of Bounded Suboptimality

Optimistic search only returns a solution at three places, in Lines 12, 17, and 18 of Figure 5-4. We can only return a solution in line 18 if the search space is exhausted without finding a solution. In line 17, we return $best_f$, a cost-optimal solution. Obviously the optimal solution is within a bounded factor of optimal. The more interesting case is when a solution is returned in line 12.

Theorem 6 *If $h(n)$ is an admissible heuristic, then the solution returned by optimistic has cost within a factor b of the optimal solution.*

Proof: The proof is based on the construction of the cleanup list. Let p be the deepest node along a path to the optimal solution. This node must exist. Initially it is the root, and when the root is expanded, it is one of the generated children. Since we never discard a node in optimistic search, p is on the cleanup list at all times, including when a solution is returned. Unfortunately, we do not know which node is p. However, we can determine which node has the smallest f-value, and this will allow us to prove the quality of the solution.

$$\begin{array}{lllll} g(incumbent) \leq b \cdot f(best_f) & & & & \text{By line 11 of Figure 5-4} \\ & b \cdot f(best_f) \leq b \cdot f(p) & & & \text{By definition of } best_f \\ & & b \cdot f(p) \leq b \cdot f(opt) & & \text{By admissibility of } h(n) \end{array}$$

By this chain of inequalities, we can see that when a solution is returned on Line 12 in optimistic search, that solution is within a bounded factor b of the cost of an optimal solution. We previously showed that the other situations produce optimal solutions. This completes the proof of bounded suboptimality. □

7.10.3 Dealing with Duplicates

Although optimistic search cannot discard duplicates as a result of the way it computes its suboptimality bounds, it can delay their expansion until the second phase of search, the bound proving phase. Effectively, we can run a variant of optimistic search where duplicate states are only ever inserted into the *cleanup* list, but are never held on *open*. This, generally, will result in finding an incumbent solution faster, although with slightly higher cost. If the incumbent solution has higher cost, proving it to be within the bound will be harder. So it is unclear if speeding the search to the first solution will always be beneficial. We show results for duplicate delaying in Life grids, the only domain where it had a substantial impact on algorithm performance.

7.10.4 Solving Time vs Suboptimality Bound

Figure 7-33 presents the time required by optimistic search and optimistic search with duplicate delaying to find a solution or a given suboptimality bound across six benchmark domains. EES is also included in the plots. As before, time is displayed on a log scale, and 95% confidence intervals about the mean are also displayed in the plot. These results show optimistic search run with an optimism parameter of 2, as was used in the first conference paper on optimistic search [65]. We will investigate the impact of optimism parameter on performance shortly.

These plots reveal that duplicate delaying in optimistic search is nearly universally beneficial to search performance, or at the very least it does no harm. The largest speedup gained by duplicate delaying is in Life cost grids, while in some domains we see small increases in mean solving time. In these cases, the two algorithms are indistinguishable because the confidence intervals overlap so strongly. We will see in the comparison of suboptimality bound versus solution quality that the improved solving times do come at

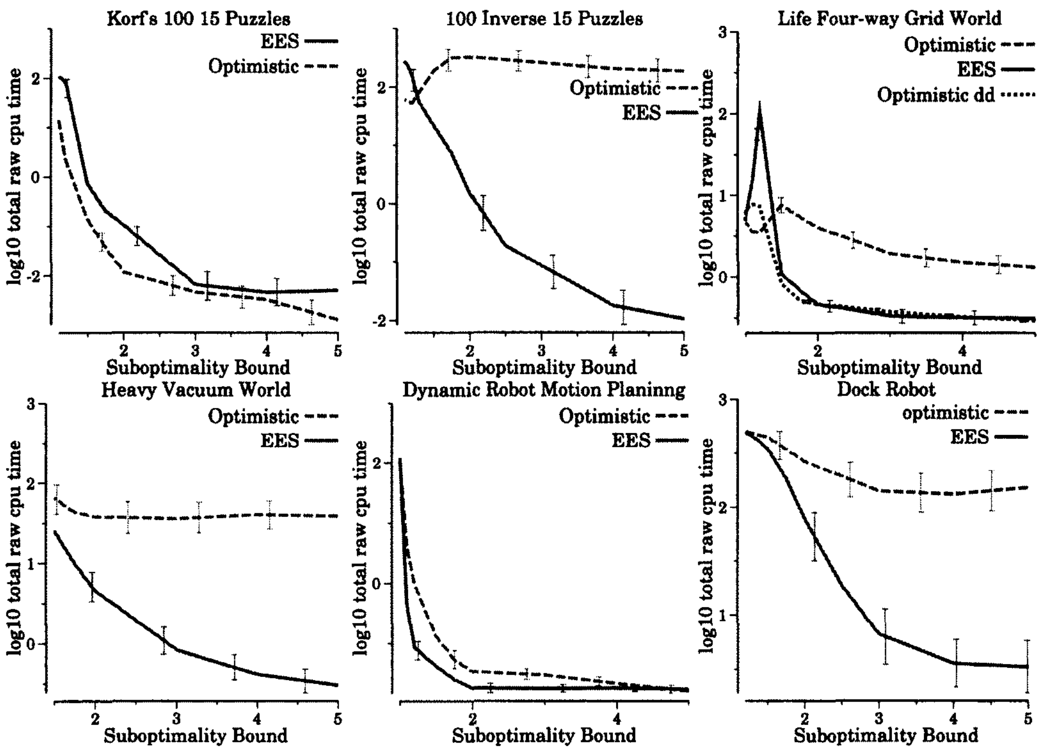


Figure 7-33: Performance of optimistic search: suboptimality bound vs. solving time

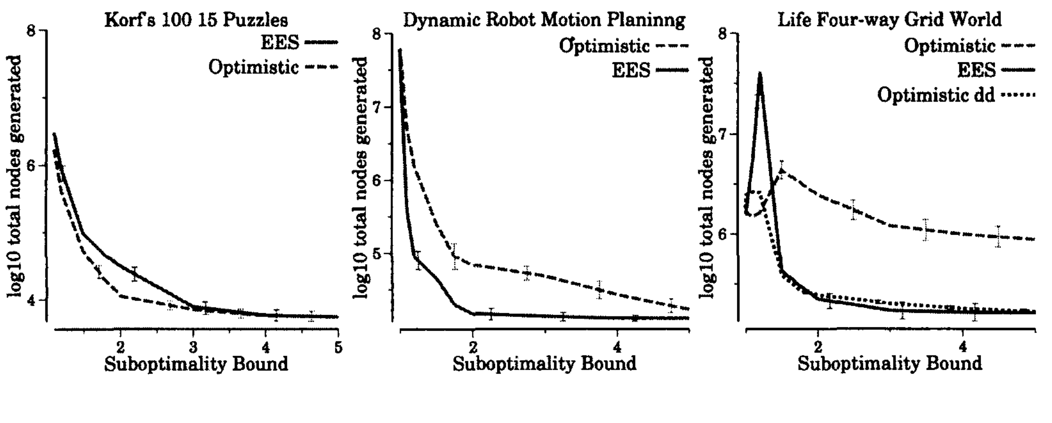


Figure 7-34: Performance of optimistic search: suboptimality bound vs. nodes generated

the cost of an increase in average solution cost and thus a decrease in solution quality.

For two thirds of the domains evaluated, EES is faster than either variant of optimistic search, sometimes by small amounts, as in dynamic robot motion planning, and sometimes by wide margins (3 to 4 orders of magnitude) as we see in the inverse tiles problems. EES appears to have a substantial edge on performance in domains with a wide range of operator costs, as is the case for the inverse tiles problem, heavy vacuum world, and dock robot domain. In these domains, optimistic search pursues cheap solutions over short solutions, resulting in increased solving time, but also decreased solution cost.

In dock robots, we also see the previously discussed U-shaped profile of weighted A* manifesting inside of the optimistic framework. As the suboptimality bound increases, the performance of optimistic search does not always get better. It improves up to a point (a bound of 3., an effective weight of 5.) and then begins to perform worse as the weight increases.

In the domains where performance between EES and optimistic search is closest, the fifteen puzzle, Life cost grids, and dynamic robot pathfinding, we also examine the number of nodes generated by the algorithms while solving a problem to get a feeling for how much of the competitive performance of optimistic search is a result of reduced overhead. These results are displayed in Figure 7-34, and they reveal that a portion, though not all, of the competitive nature of optimistic search is a result of reduced overhead. The only domain where optimistic search is consistently the better choice when evaluating search algorithms in terms of the number of states evaluated is the standard 15 puzzle. This is also the only domain with unit cost actions in the evaluation. As the domain has unit cost actions, EES receives no benefit from distinguishing between solution cost and solution length because there is no difference here.

7.10.5 Impact of Optimism Parameter

Figure 7-35 shows the impact that the optimism parameter has on the performance of optimistic search for the dynamic robot domain, though results for other domains are similar.

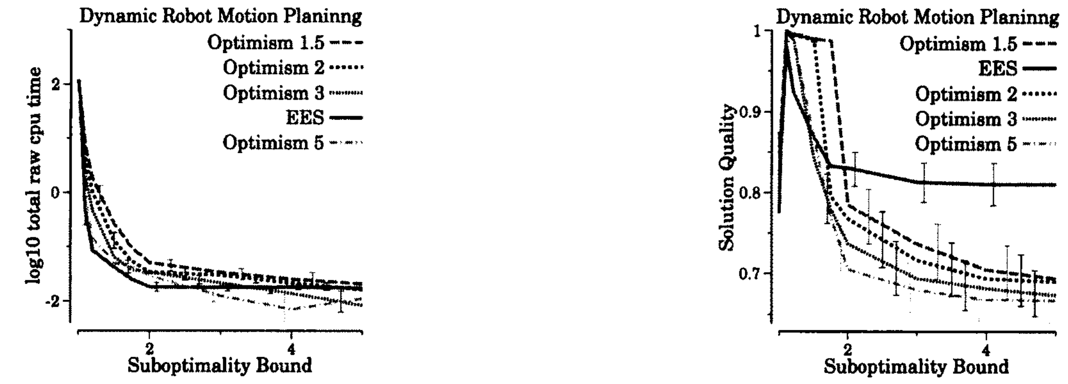


Figure 7-35: Impact of optimism parameter on optimistic search performance

We report only optimistic search, not the duplicate dropping variant. Again, the results are similar across duplicate handling techniques. As the optimism parameter is increased, solving times generally lower. Similarly, solution qualities generally lower as the optimism parameter is raised. There are of course exceptions to this. We could, for example, pick an optimism parameter so large that optimistic search reverted to A* search.

7.11 Skeptical Search

We previously introduced Skeptical Search in 5.2.6. As we will see in the following evaluation, skeptical search offers two benefits over optimistic search. It removes the need for parameter tuning and provides improved performance in several benchmark domains.

The implementation details, proof of suboptimality and consideration of duplicates for skeptical search are identical to that of optimistic search. A visualization of its expansion order is shown in Figure 7-36. If we compare the expansion order of Skeptical search with that of Optimistic, we see that the two approaches are very similar, differing primarily in their greedy search phase. This is because optimistic search works with a fixed inadmissible heuristic, whereas skeptical search is learning its inadmissible heuristic online, during search. These two algorithms are cut from the same cloth. They differ only in that it is obvious skeptical search is using an inadmissible heuristic, while with optimistic search we must make an argument that applying an additional weight over the suboptimality bound on h

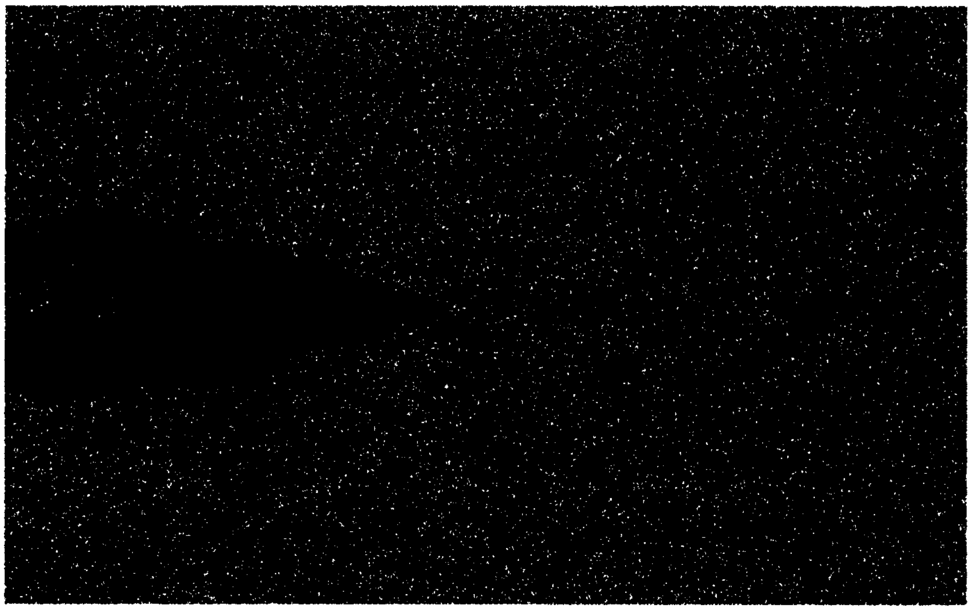


Figure 7-36: Expansion order of skeptical search on a pathfinding problem

is an elementary attempt to correct for an underestimating heuristic, and thus optimistic search does indeed search on inadmissible estimates of cost-to-go.

7.11.1 Solving Time vs Suboptimality Bound

Figure 7-37 shows the relative performance of EES and skeptical search in terms of time to return a solution for the given suboptimality bound. As before, the y-axis reports the mean time to solution on a log scale, with 95% confidence intervals about the mean. The relative performance of EES and skeptical search largely mirrors the relative performance of EES and optimistic search. For two thirds of the domains considered, EES provides obviously better performance than skeptical search. In the remaining two domains, EES and skeptical search are about at parity with one another, with skeptical search being slightly better in the unit cost tiles puzzle. Here, distinguishing between solution cost and solution length, something EES does but skeptical search does not, provides no additional advantage when determining search order.

What is interesting is that the gap between the performance of the search algorithms

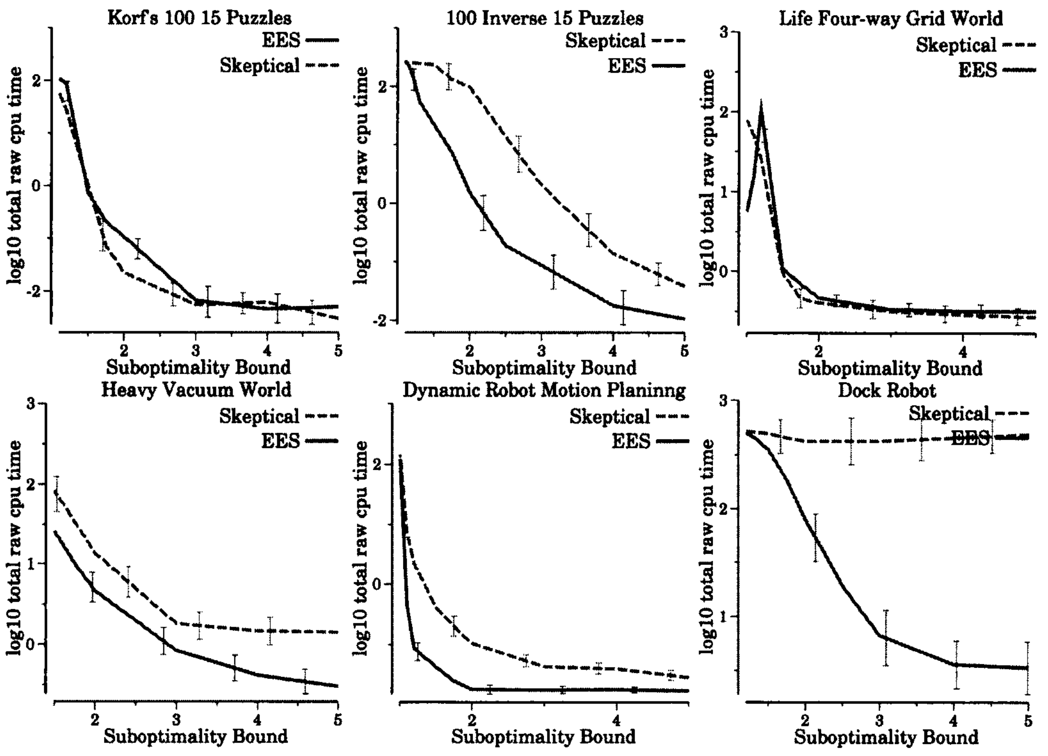


Figure 7-37: Performance of skeptical search: suboptimality bound vs. solving time

is narrower for EES and skeptical search than it was for EES and optimistic search. This is in part because skeptical search, in this evaluation, uses the same inadmissible cost-to-go heuristic that EES uses when determining search order. This inadmissible estimate of cost-to-go is based, in part, on an estimate of actions-to-go [64]. Incorporating action-to-go estimates is known to be particularly beneficial in domains where actions may have a wide range of costs, as is discussed in Chapter 3. The two domains where skeptical is substantially better than optimistic search, heavy vacuum problems and the inverse cost tiles puzzles, also have a wide variance in action cost. So it is likely that skeptical search is getting some performance benefits from incorporating action-to-go estimates via the "backdoor" of how it constructs its inadmissible cost-to-go estimates.

```
EES(root, w)
1.  $open \leftarrow \{root\}$
2.  while $open \neq \{\}$
3.    let $n = selectNode(open, w)$ in
4.      if $goal_p(n)$
5.      then return $n$
6.      else $open \leftarrow open - \{n\}$
7.          for each child $c$ of $n$, $open \leftarrow open \cup \{c\}$
8.  return no solution
```

Figure 7-38: Pseudo code for explicit estimation search

7.12 Explicit Estimation Search

Explicit Estimation Search (EES) is a new bounded suboptimal search algorithm that incorporates the objectives of bounded suboptimal search directly into its search order. It uses inadmissible, or potentially over-estimating, heuristics for cost and actions-to-go in order to pursue the shortest w-admissible solution to the problem. As we will discuss later, shorter solutions should be easier to find, and so EES is attempting to minimize solving time within a given bound by proxy. To ensure that the suboptimality bound is met, EES also relies on the more traditional admissible heuristics for cost-to-go.

Pseudo code for EES is provided in figure 7-38. We can see that explicit estimation search is a standard best-first bounded suboptimal search algorithm. It takes as input an initial state and a suboptimality bound and returns a bounded suboptimal solution should one exist (line 6) or no solution if the space contains no solution (line 9). The most interesting part of EES, and indeed any best-first search algorithm, is how it selects the next node for expansion. We now discuss this portion of EES in detail.

7.12.1 Explicit Estimation Search Order

EES keeps track of three values for every node. $f(n) = g(n) + h(n)$, an admissible estimate of the total cost of a solution passing through node n. $f(n)$ will be used to construct a lower bound on the cost of a solution to the problem, and it is how EES shows that returned solutions are within the bound. $\widehat{f}(n) = g(n) + \widehat{h}(n)$ is similar to $f(n)$, but inadmissible. $\widehat{f}(n)$ is EES's best guess as to the cost of a solution through n. EES will use $\widehat{f}(n)$ to estimate which nodes will lie within the suboptimality bound. Finally, EES uses $\widehat{d}(n)$, an inadmissible estimate of the number of actions required to complete a solution beginning with node n. $\widehat{d}$ is a proxy for search effort, and is used to ensure EES pursues solutions that can be found quickly.

Using these three measurements, EES keeps track of three special nodes in the search space. $best_f$, $best_{\widehat{f}}$, and $best_{\widehat{d}}$. $best_f$ is the node with the smallest $f(n)$ for all nodes that have been generated but not yet expanded. $best_f$ is interesting because $f(best_f)$ represents a lower bound on the cost of a solution to the problem under consideration. As we previously noted in Section 7, expanding all nodes with $w \cdot f(n) < g(sol)$ allows us to show that sol has cost within a bounded factor w of optimal.

$best_{\widehat{f}}$ is an inadmissible doppelganger of $best_f$. Where $best_f$ is used to find a lower-bound on optimal solution cost, we use $best_{\widehat{f}}$ to construct our best estimate of optimal solution cost as in $\widehat{f}(best_{\widehat{f}})$. EES will use this estimate of optimal solution cost to construct a subset of apparently w-admissible nodes to consider for expansion.

$best_{\widehat{d}}$ is selected from this suspected to be w-admissible subset. Among those nodes estimated to be w-admissible, it is the node with the smallest $\widehat{d}$-value, the least estimated actions-to-go. Effectively, of all potentially w-admissible solutions, $best_{\widehat{d}}$ is estimated to be the easiest to complete. "Easiest to complete" naturally refers to computation time, and while actions-to-go is not a direct measurement for computation time, it has been shown to be a reasonable proxy for it [14].

Slightly more formally the three nodes can be defined as:

$$best_f = \operatorname*{argmin}_{n \in open} f(n) \tag{7.15}$$

$$best_{\widehat{f}} = \operatorname*{argmin}_{n \in open} \widehat{f}(n) \tag{7.16}$$

$$best_{\widehat{d}} = \operatorname*{argmin}_{n \in open \wedge \widehat{f}(n) \leq w \cdot \widehat{f}(best_{\widehat{f}})} \widehat{d}(n) \tag{7.17}$$

It is of course possible that there are multiple nodes with the smallest f-value, smallest $\widehat{f}$-value, or smallest $\widehat{d}$-value. In this case, the above formal definition is incorrect, though it still provides the intuition behind these nodes. For $best_f$ and $best_{\widehat{f}}$, we are primarily interested in the value associated with the node, and ties are of little consequence. For $best_{\widehat{d}}$, tie breaking is very important. We now discuss tie breaking for all three nodes.

In the case of $best_f$, we recommend breaking ties in favor of low g-values. Nodes with more of their f-values in cost-to-go, $h(n)$, than in cost-incurred, $g(n)$, are less likely to have been reached by a suboptimal path because they are, generally, the result of fewer expansions. This is absolutely true for unit-cost domains, but if actions may have differing costs, this is only a heuristic and not always the case. The intuition is that by preferring nodes with low g-values we are more likely to improve a path to a node already on our open list, improving the chances that it may be selected for expansion.

For $best_{\widehat{f}}$ ties should be broken in favor of low f-values. If two nodes have the same estimated total cost, then we should prefer the node with lower $\widehat{d}(n)$. By preferring the node with lower $\widehat{d}$, we may end up converting $best_{\widehat{f}}$ into $best_{\widehat{d}}$, thereby allowing the search to pursue a solution rather than busying itself with book keeping and bound proving. An argument can also be made for breaking ties in favor of low f-values. A node with low f-values is more likely to be expanded as $best_f$ later on in the search. By expanding it now, we save ourselves the effort of doing it later. Pilot experiments showed that tie-breaking on low $\widehat{d}$-values was slightly better performing.

For $best_{\widehat{d}}$, we should break ties in favor of low $\widehat{f}$-values. Nodes with lower $\widehat{f}$-values are more likely to stay on the focal list of EES, and they are more likely to be legal for expansion by the criteria $\widehat{f}(n) \leq w \cdot f(best_f)$. By preferring the node with the smaller $\widehat{f}$-value, we

```
selectNode
1.  if $\widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f)$ then $best_{\widehat{d}}$
2.  else if $\widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ then $best_{\widehat{f}}$
3.  else $best_f$
```

Figure 7-39: Node selection function of EES

are attempting to pursue lower-cost solutions. All else being equal, low-cost solutions are easier to show to be within the desired suboptimality bound, thus potentially speeding up search.

At every expansion, EES chooses from among these three nodes using the function described in Figure 7-39. EES first considers $best_{\widehat{d}}$, as pursuing nearer goals should lead to a goal fastest, satisfying the "as quickly as possible" objective of bounded suboptimal search. $best_{\widehat{d}}$ is selected if its expected solution cost can be shown to be within the suboptimality bound. That is, if $\widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f)$. In prose, this conditional says pursue $best_{\widehat{d}}$ only if we suspect we could convert it into a complete solution that we could return without raising the lower-bound on optimal solution cost. If $best_{\widehat{d}}$ is unsuitable, $best_{\widehat{f}}$ is examined. We suspect that this node lies along a path to an optimal solution. Expanding this node can also expand the set of candidates for $best_{\widehat{d}}$ by raising $\widehat{f}(best_{\widehat{f}})$. We only expand $best_{\widehat{f}}$ if it is estimated to lead to a solution within the bound. If neither $best_{\widehat{f}}$ nor $best_{\widehat{d}}$ are thought to be within the bound, we return $best_f$. Expanding it can raise our lower bound $f(best_f)$, allowing us to consider $best_{\widehat{d}}$ or $best_{\widehat{f}}$ in the next iteration.

Figure 7-40 shows the order in which EES expands nodes on a unit-cost grid navigation problem with suboptimality bound $w = 1.3$. The root of the problem is in the middle of the left-hand-side of the grid, and the goal is in the middle of the right-hand-side. Nodes are colored by the time they were last examined by the search algorithm. We say last as EES may re-expand some states. Nodes that were expanded early on are colored yellow, and as the search progresses, the color changes from yellow to red.

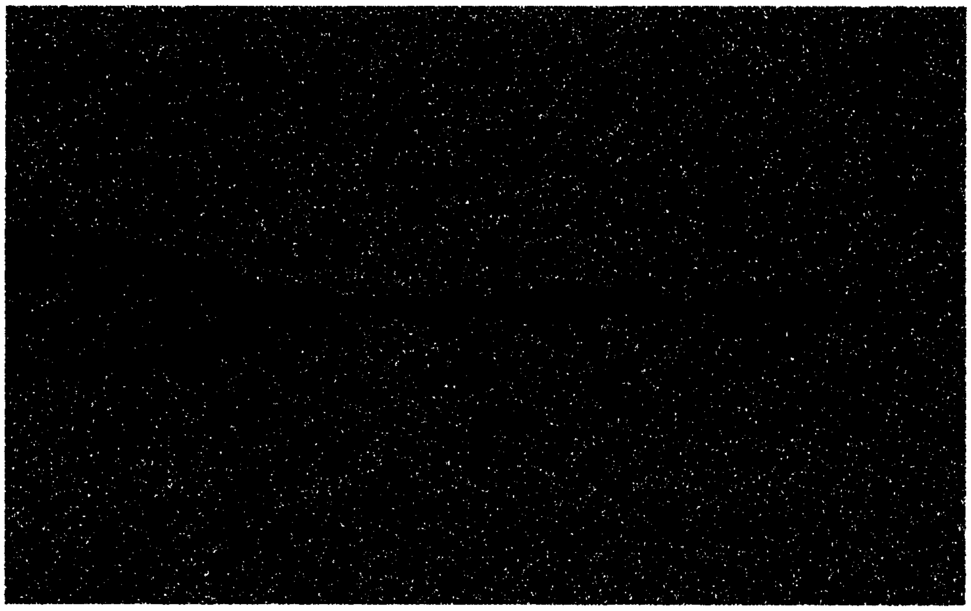


Figure 7-40: Expansion order of explicit estimation search ($w = 1.3$) on a pathfinding problem

In Figure 7-40 we see that order in which nodes are expanded is strongly related to their proximity to the root and the goal. Nodes near the root are all expanded early (these nodes are primarily yellow), and as we approach the goal, nodes become orange and then finally red. States that were never explored by search remain white, and the obstacles appear as blackened cells.

Obviously, at the beginning of the search many nodes will be near the root. However, the search staying near the root early on is also a result of the inadmissible heuristics we're using and the relative power of the admissible heuristic. Early on the online estimators used by EES in this dissertation are unstable, as they are based on observed error in the heuristic and very few observations have been made. This leads to a situation where $\widehat{h}$-values for nodes are often very high, especially relative to the h-value of nodes. This means that most nodes appear to exist outside of the currently provable suboptimality bound, causing EES to expand $best_f$ repeatedly until estimates of $\widehat{h}$ calm down and some nodes appear to lead to solutions within the bound. These are then expanded until a goal is produced, often

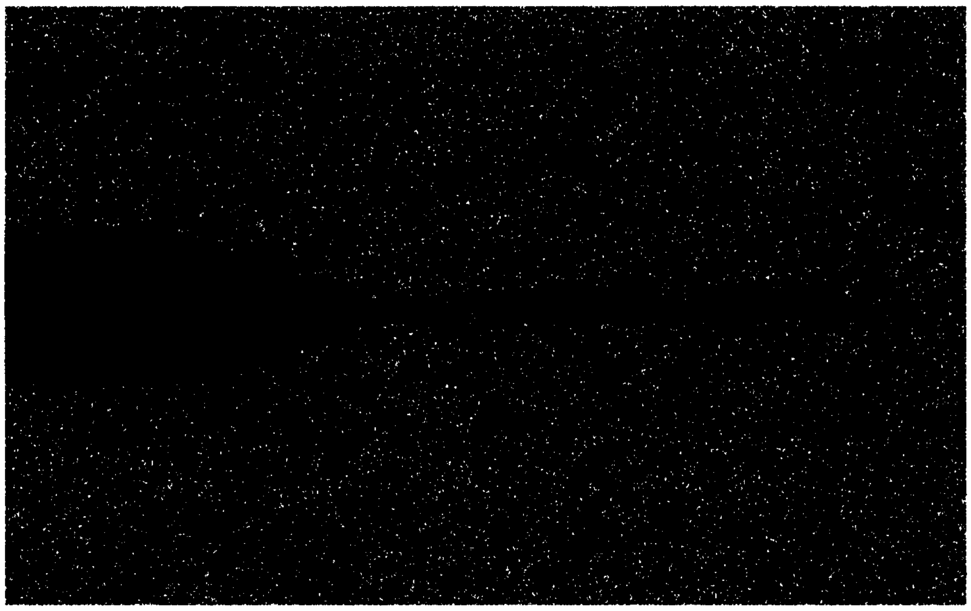


Figure 7-41: Visualization of which node was selected by *selectNode*

without the need to go back and improve our lower bound on optimal cost solutions.

While this makes good intuitive sense, we will see that such behavior is not common to previous work in bounded suboptimal search. There are some exceptions to this general observation on expansion order, for example some nodes near the root are colored orange. These nodes were expanded later on in order to prove the suboptimality bound for the solution, as we will now see.

Figure 7-41 provides an alternative perspective on the search order of EES run with $w = 1.3$ on this grid pathfinding problem. In this image, we see not the order in which nodes were expanded, but rather the rule in *selectNode* by which they were selected. If *selectNode* returned $best_f$, then the node is colored blue, if it was returned by $best_{\widehat{f}}$, then the node is green, and if the node was returned by $best_{\widehat{d}}$, then the node is colored red. To deal with duplicates, we only show the first rule by which a node was expanded.

Nodes near the root are selected when they are $best_f$. We expand $best_f$ in order to allow for the expansion of other nodes, and to prove the suboptimality bound. Admissible heuristics are, by definition, optimistic. As search progresses, the f-value of nodes tends to rise, so nodes near the root often have the smallest f-values. Therefore, nodes near the starting state are often going to be expanded as $best_f$ rather than as any other node.

Nodes expanded as $best_{\widehat{d}}$, colored in red, are a thin strand proceeding in almost a straight line towards the goal. This is exactly what we should expect from the expansion order. If $\widehat{d}$ and $\widehat{f}$ were perfect, then some child of $best_{\widehat{d}}$ would always be the next $best_{\widehat{d}}$, and search could proceed directly towards a goal, if we were to ignore the suboptimality bound. While $\widehat{f}$ and $\widehat{d}$ aren't exactly perfect in practice, they aren't particularly inaccurate on this problem either. If they were, the strip of nodes expanded by as $best_{\widehat{d}}$ would be much wider as a result of vacillation [14].

Vacillation is a measure of the indecision experienced by search when deciding what is best. If we think of searches as inducing a tree of possible solutions from the initial state using the expand function, then vacillation is a measurement of how frequently the search algorithm hops between subtypes. Since heuristics are not truth, a systematic best-first search will occasionally need to abandon one line of inquiry for another. Accurate heuristics can reduce the amount of vacillation experienced by search and thus improve performance.

There is another surprising thing about the nodes expanded as $best_{\widehat{d}}$, none of them are near the root, in the area expanded by $best_f$. In this chapter, as we will soon discuss, we use online correction to produce $\widehat{h}$ and $\widehat{d}$ from base heuristic estimators. As a result of the online correction, $\widehat{h}$ and $\widehat{d}$ are very volatile early on in the search, having estimates that differ wildly between parent and child. The estimates are also, generally, quite high. As a result, all nodes appear to be outside of the provably suboptimality bound early on, and so nodes are expanded in $best_f$ order until the online estimators settle down.

Finally, there are a surprisingly large number of expansions of $best_{\widehat{f}}$. $best_{\widehat{f}}$ is only chosen for expansion in the event that $best_{\widehat{d}}$ isn't expected to lead to a w-admissible solution currently, but $best_{\widehat{f}}$ is. For tight suboptimality bounds, such as the one considered here, that can occur quite frequently. As we will discuss shortly, we could ignore $best_{\widehat{f}}$, and only consider $best_{\widehat{d}}$ and $best_f$ when selecting nodes for expansion. This would remove all of the green nodes in Figure 7-41, but it actually harms performance for tight suboptimality bounds, as we will see.

7.12.2 Sources of $\widehat{h}$ and $\widehat{d}$

The expansion order of EES relies heavily on an admissible estimate of cost-to-go and inadmissible estimates of cost-to-go and actions-to-go, $\widehat{h}$ and $\widehat{d}$ respectively. Yet we have not discussed where these estimates come from. Admissible estimates of cost-to-go are a staple of heuristic search, and we therefor know many ways of efficiently computing lower bounds on solution cost. These include abstraction based techniques like pattern databases [11] and relaxation techniques like ignoring obstacles in grid pathfinding.

Actions-to-go estimates of any stripe are slightly rarer in bounded suboptimal search, in part because they are not required (admissible heuristics are needed to prove suboptimality bounds), but are nonetheless simple to construct for an arbitrary problem. Several of the bounded suboptimal search algorithms we will discuss as previous work rely on actions-to-go estimates, so such estimates are not particularly novel, simply less common. The simplest way of constructing an estimate of actions-to-go is to take the same approach that we would for constructing estimates of cost-to-go, but rather than using the cost of actions, simply replace action costs with 1.

There are several methods of constructing inadmissible estimates of cost or actions-to-go. Hand crafted heuristics constructed by a domain expert are perhaps the oldest. As we previously noted, it has been suggested that we could use the Manhattan distance of all tiles plus three times the linear conflicts measure as a heuristic for the sliding tiles puzzle. Such a heuristic is no where near admissible, but it does provide reasonable guidance on the puzzle for which it was proposed. Inadmissible heuristics can also be automatically constructed. This can be done offline, using training instances to learn improved evaluators [56], in between instances when solving a large number of problems from the domain [29, 6], or over the course of a single search algorithm by evaluating the accuracy of a heuristic on the search tree [64].

7.12.3 Proof of Bounded Suboptimality

We've stated that explicit estimation search is a bounded suboptimal search algorithm, but we have yet to demonstrate that the solutions returned by EES are guaranteed to be within a bounded-factor w of optimal. We now present a proof that EES is guaranteed to only produce solutions whose cost is within a bounded factor w of the optimal cost solution.

Theorem 7 *If $h(n) \leq \widehat{h}(n)$ and $g(opt)$ represents the cost of an optimal solution, then for every node n expanded by EES, it is true that $f(n) \leq w \cdot g(opt)$, and thus EES returns w-admissible solutions.*

Proof: *selectNode* will always return one of $best_{\widehat{d}}$, $best_{\widehat{f}}$ or $best_f$. No matter what node n is selected we will show that $f(n) \leq w \cdot f(best_f)$. This is trivial when $best_f$ is chosen. When $best_{\widehat{d}}$ is selected:

$$\begin{aligned}
\widehat{f}(best_{\widehat{d}}) &\leq w \cdot f(best_f) && \text{by } selectNode \\
g(best_{\widehat{d}}) + \widehat{h}(best_{\widehat{d}}) &\leq w \cdot f(best_f) && \text{by definition of } \widehat{f} \\
g(best_{\widehat{d}}) + h(best_{\widehat{d}}) &\leq w \cdot f(best_f) && \text{by } h \leq \widehat{h} \\
f(best_{\widehat{d}}) &\leq w \cdot f(best_f) && \text{by definition of } f \\
f(best_{\widehat{d}}) &\leq w \cdot g(opt) && \text{by admissibility of } h \\
g(best_{\widehat{d}}) &\leq w \cdot g(opt) && \text{by admissibility of } h \text{ and } best_{\widehat{d}} \text{ being a goal}
\end{aligned}$$

When $best_{\widehat{d}}$ is a solution, $h(best_{\widehat{d}}) = 0$ and $f(best_{\widehat{d}}) = g(best_{\widehat{d}})$, thus the cost of the solution represented by $best_{\widehat{d}}$ is within a bounded factor w of the cost of an optimal solution. The

$best_{\widehat{f}}$ case is analogous:

$$\begin{aligned}
\widehat{f}(best_{\widehat{f}}) &\leq w \cdot f(best_f) && \text{by } selectNode \\
g(best_{\widehat{f}}) + \widehat{h}(best_{\widehat{f}}) &\leq w \cdot f(best_f) && \text{by definition of } \widehat{f} \\
g(best_{\widehat{f}}) + h(best_{\widehat{f}}) &\leq w \cdot f(best_f) && \text{by } h \leq \widehat{h} \\
f(best_{\widehat{f}}) &\leq w \cdot f(best_f) && \text{by definition of } f \\
f(best_{\widehat{f}}) &\leq w \cdot g(opt) && \text{by admissible } h \\
g(best_{\widehat{f}}) &\leq w \cdot g(opt) && \text{by admissible } h \text{ and } best_{\widehat{f}} \text{ being a goal}
\end{aligned}$$

□

EES only expands nodes returned by *selectNode*, and since any of the nodes returned by *selectNode* must have cost within a bounded factor w of $g(opt)$, any solution returned by EES must be within a bounded factor w of optimal, thus completing the proof.

7.12.4 Implementation Considerations

Explicit Estimation Search is structured like a classic best-first search: insert the initial node into *open*, and at each step, we select the next node for expansion using *selectNode*. To efficiently access $best_{\widehat{f}}$, $best_{\widehat{d}}$, and $best_f$, EES maintains three orderings over the nodes: the *open* list, *focal* list, and *cleanup* list. These lists are used to access $best_{\widehat{f}}$, $best_{\widehat{d}}$, and $best_f$ respectively. The *open* list contains all generated but unexpanded nodes sorted on $\widehat{f}(n)$. The node at the front of the *open* list is $best_{\widehat{f}}$. *focal* is a prefix of the *open* list ordered on $\widehat{d}$. *focal* contains all of those nodes where $\widehat{f}(n) \leq w \cdot \widehat{f}(best_{\widehat{f}})$. The node at the front of *focal* is $best_{\widehat{d}}$. *cleanup* contains all nodes from *open*, but is ordered on $f(n)$ instead of $\widehat{f}(n)$. The node at the front of *cleanup* is $best_f$. We need to be able to quickly select a node at the front of one these queues, remove it from all relevant data structures, and reinsert its children efficiently. To accomplish this, we implement *cleanup* as a binary heap, *open* as a red-black tree, and *focal* as a heap synchronized with a left prefix of *open*. This lets us perform most insertions and removals in logarithmic time except for transferring nodes from *open* onto *focal* as $best_{\hat{f}}$ changes. This requires us to visit a small range of the red-black tree in order to put the correct nodes in *focal*.

For domains with integer action costs, performance could be improved by using a bucketed open list instead of a binary heap for *cleanup* and *open*. In this restricted case, we can use integer-based bucketed open lists to get constant insertion and removal times instead of the log times that we have with heap backed priority queues. This can result in substantial speedups, as open lists can be quite large. In the empirical evaluation in this chapter, we will ignore this potential optimization because it is not general.

Search algorithms should perform duplicate detection on node generation (ie on line 7 of Figure 7-38), rather than on node expansion. There are many domains with huge numbers of duplicates, and maintaining duplicate nodes on open increases the cost of all operations needlessly, as most operations have complexity logarithmic in the number of nodes in the list . To do detection on insertion into node lists, you have to have arbitrary removal and replacement for all of your node structures. This is not particularly difficult if one is willing to write their own data structures, but many standard libraries, C++'s SOL for example, do not provide this ability.

Goal tests should be done on generation if goal tests are inexpensive. The small amount of pruning this gives you can actually improve performance in some settings. With goal testing on generation, the pseudo code for the algorithm changes as shown in Figure 7-42. The proof of suboptimality for this variant of EES is very simple. We can see in line 4 of Figure 7-42 that EES with goal tests on node generation only exits the search loop when no solution exists or the cost of the incumbent solution can be shown to be within a bounded factor w of $f(best_f)$. Since f-values are based on admissible heuristics, this proves the incumbent is within a bounded factor w of the optimal cost solution as well, completing the proof.

```
EESGoalGen(root, w)
1.  $open \leftarrow \{root\}$
2.  $cost \leftarrow \infty$
3.  $incumbent \leftarrow None$
4.  while $open \neq \{\} \wedge w \cdot f(best_f) < cost$
5.    let $n = selectNode$ in
6.      $open \leftarrow open - \{n\}$
7.      for each child $c$ of $n$
8.        if $f(c) < cost$
9.          if $goal_p(c)$
10.           $incumbent \leftarrow c$
11.           $cost \leftarrow g(c)$
12.         else $open \leftarrow open \cup \{c\}$
13. return $incumbent$
```

Figure 7-42: EES with goal testing on node generation

7.12.5 Performance vs. Heuristic Accuracy

Explicit Estimation Search relies on three heuristic functions, an admissible cost-to-go heuristic h as well as $\widehat{h}$ and $\widehat{d}$. These latter two estimate the true cost-to-go and true actions-to-go respectively. Since they are not bound by the constraint of admissibility, we hope that they can be more accurate predictors of these values. We now consider what happens in several extreme cases and examine the performance of EES as the corrections are degraded in a controlled experiment, arriving at the expected outcome that better corrections lead to better behaviors. Our analysis of the behavior of these algorithms relies on the assumption that our actions-to-go estimations are estimating the length of the shortest optimal solution beneath a node, and not simply the shortest solution beneath a node.

While the algorithm will work with either interpretation of actions remaining in practice, the intention of the algorithm encourages us to use the cost-optimal variant. When we determine whether or not a node can be included in the set of all likely w-admissible solutions, we decide so optimistically. That is, we consider all nodes whose cost-optimal completion is estimated to have cost within the bound. In the worst-case, this is the only solution beneath a node within the desired suboptimality bound. As a result, we should tailor our proxies for completion-cost to this worst case. Otherwise we run the risk of substantially underestimating the cost of finding a solution beneath a node, harming search performance.

7.12.6 $\widehat{h} = h^*$ & $\widehat{d} = d^*$

In an ideal world, both inadmissible heuristics would be equal to the true cost to go heuristic h^* and the true distance to go heuristic d^*. Surprisingly, in this situation we are neither guaranteed to find the optimal solution, nor will we always find the shortest solution within the bound. We will however find one of these two solutions. In this situation, *selectNode* will repeatedly expand $best_f$, since $best_f$ is based on an admissible h. h may be much smaller than h^*, and the difference in these two heuristics determines just how many $best_f$ expansions are performed. As $best_f$ continues to be expanded, $f(best_f)$ will steadily rise as

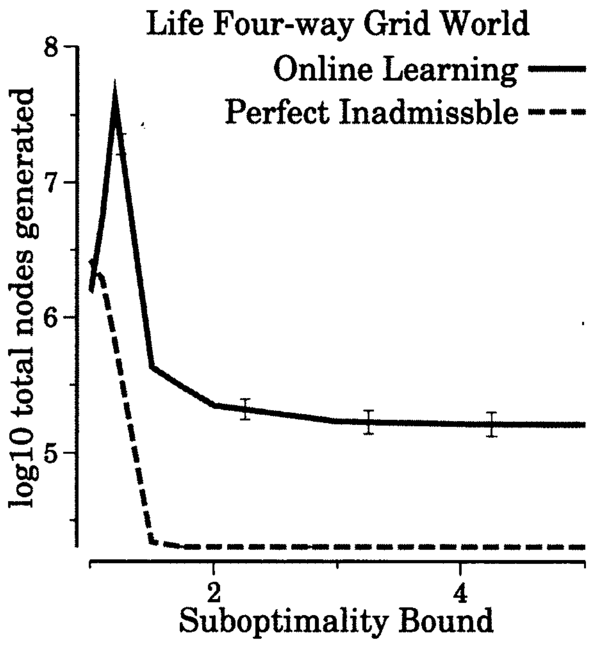


Figure 7-43: Performance of ees when $\widehat{h} = h^*$ and $\widehat{d} = d^*$

a result of the admissibility of h. At some point this value will be large enough so that one or both of $w \cdot f(best_f) \leq \widehat{f}(best_{\widehat{f}})$ and $w \cdot f(best_f) \leq \widehat{f}(best_{\widehat{d}})$ will hold. If both become true, EES will expand $best_{\widehat{d}}$ into the shortest w-admissible goal. However, if we manage to show only that $best_{\widehat{f}}$ is provably within the bound then we will expand it. At some point, it will become the new $best_{\widehat{d}}$, and will be expanded into the optimal goal.

Figure 7-43 shows the performance of EES on a Life-cost grid-world pathfinding problem, described in detail in Chapter 3 with a summary of features presented in Table 3-1, when both inadmissible heuristics are perfectly accurate. The y-axis of the plot shows mean number of nodes generated across 100 instances on a log scale, with 95% confidence intervals about the mean, and the x-axis shows the suboptimality bound with which the algorithm was run. We report node generations to remove overhead from the consideration; the two algorithms are implemented slightly differently due to the source of their heuristics.

Unsurprisingly, EES using perfect inadmissible heuristics, Perfect Inadmissible in the plot, outperforms EES using online learning techniques to produce its inadmissible estimates of cost and actions-to-go. This is the version of EES used throughout the evaluation in this chapter. What we might find surprising are the peak in online learning and the fact that the performance difference is limited to a single order of magnitude for most suboptimality bounds, ie outside of the peak. Recall that only the inadmissible heuristics, $\widehat{h}$ and $\widehat{d}$, have perfect information. h is still the admissible augmented Manhattan distance described in Appendix 3. While the EES with perfect information can find a w-admissible solution in time linear to the length of that solution, proving the solution is within the bound is still difficult because the admissible heuristic is imperfect. This limits the potential difference in performance. The relationship between h and $\widehat{h}$ plays a very important role in the performance of EES.

The peak for the realistic implementation of EES, and its absence for perfect inadmissible heuristics, is also of interest. The peak is the result of node re-expansion, that is it is the result of reopening nodes that are encountered with a better path. Using perfect heuristics ensures that we never encounter a node by a suboptimal path when selecting $best_f$ or $best_{\widehat{f}}$ for expansion ($best_{\widehat{d}}$ may still encounter a node by a suboptimal path, even with perfect information). However, even though EES with perfect heuristics can (and does) encounter nodes by suboptimal paths, it will never re-open a node. With perfect information, EES will expand nodes from $best_f$ until a provably w-admissible solution is on the open list. It will then directly pursue this w-admissible solution until the goal is returned. Since solutions contain no cycles, and since $best_f$ expansions always expand nodes with their optimal g-value, no node can require re-expansion when perfect inadmissible heuristics are used. Realistically, EES will interleave proving the bound and solving the problem. It may also make mistakes when estimating if a node has a solution beneath it whose cost is within the suboptimality bound, resulting in a large number of re-expansions for some suboptimality bounds. This manifests as a peak for tighter suboptimality bounds, as we see in Figure 7-43.

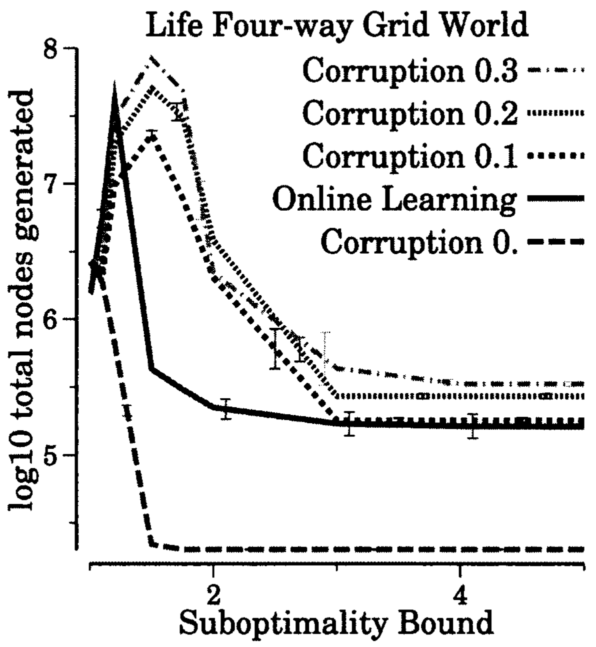


Figure 7-44: Performance of EEC both $\widehat{h}$ and $\widehat{d}$ may be inaccurate

7.12.7 $\widehat{h} \neq h^*$ & $\widehat{d} \neq d^*$

This situation represents reality, where neither $\widehat{h}$ nor $\widehat{d}$ are perfectly accurate. We present results for this setting in Figure 7-44. To construct heuristics with controlled amounts of error, we compute h^* for all states. Then, when computing the heuristic for a given state, we introduce noise. Since inadmissible heuristics can err in either direction, that is we expect $h^*(n) \geq \widehat{h}(n)$ to be just as likely as $h^*(n) \leq \widehat{h}(n)$, we must be sure that our corrupted heuristic is equally likely to err on both sides of truth. We set some maximum magnitude, say 0.1, and then select a value, called c at random between -0.1 and 0.1. Then, the reported heuristic for a node is $h_{corrupt}(n) = h^*(n) \cdot (1 + c)$ where c is selected independently for each node. EES with various maximum corruptions, ranging from no corruption to 0.3, are shown in Figure 7-44. We also include online learning for sake of comparison. Again, results are reported in terms of states examined on a log scale to control for overhead.

In Figure 7-44 we see that, as the error introduced to the heuristic increases, performance of the algorithm decreases. Similarly, as we introduce noise, the number of nodes needed to be re-expanded increases, seen in the size of the peaks for small suboptimality bounds. Surprisingly, the online corrections used in this evaluation provide better are better performance than any of the evaluated corruptions. This is surprising because the online estimators are known to be inaccurate, as we discussed in Chapter I, in fact more inaccurate than the corrupted estimators studied here. It is important for $\widehat{h}$ to be accurate in absolute terms, as it forms the set of all nodes estimated to lead to w-admissible solutions. Further we use it to determine if a node can be extended into a complete solution without needing to perform the bound-proving $best_f$ expansions, so $\widehat{h}$ may harm performance in this way as well. $\widehat{d}$ need only provide good relative orderings over nodes, as we want to purse the easiest to find solution, but we don't really care how difficult it is to find in absolute terms.

7.12.8 Alternate Expansion Rules

The *selectNode* function is the heart of EES in that it determines the search order and thus the behavior of the algorithm. We previously argued that *selectNode* was directly motivated by the goal of suboptimal search outlined in Section 7: find a solution within the suboptimality bound as quickly as possible. While *selectNode* is nearly a direct transcription of this goal into an algorithm, that does not preclude the usefulness of alternative *selectNode* functions. We investigate several of these alternate functions below.

7.12.9 Conservative

Although the formulation of *selectNode* is directly motivated by the stated goal of bounded suboptimal search, it is natural to wonder if there exist other formulations of select node that may have better performance or be beneficial in particular settings. We consider a more conservative approach called *selectNode*$_{con}$, but find that it produces an expansion order identical to that of *selectNode*.

selectNode$_{\mathbf{con}}$

```
1.  if $\widehat{f}(best_{\widehat{f}}) > w \cdot f(best_f)$ then $best_f$
2.  else if $\widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f)$ then $best_{\widehat{d}}$
3.  else $best_{\widehat{f}}$
```

$selectNode_{con}$ is a more conservative approach in that it wants to do the bound-proving expansions, those on $best_f$, as early as possible and so it considers expanding $best_f$ before any other node. If $best_f$ wasn't selected for expansion, it then considers $best_{\widehat{d}}$ and $best_{\widehat{f}}$ in the same order as before. This expansion order produces a solution within the desired bound by the same argument as that for *selectNode.*

selectNode$_{\textbf{con}}$

```
1.  if $\widehat{f}(best_{\widehat{f}}) > w \cdot f(best_f)$ then $best_f$
2.  if $\widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f) \wedge \widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ then $best_{\widehat{d}}$
3.  if $\widehat{f}(best_{\widehat{d}}) > w \cdot f(best_f) \wedge \widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ then $best_{\widehat{f}}$
```

Using the ordering of the rules and the properties of $best_f$, $best_{\widehat{f}}$, and $best_{\widehat{d}}$ we can rewrite $selectNode_{con}$ as seen above. The rule for selecting $best_f$ is unchanged. The rule for selecting $best_{\widehat{d}}$ has been strengthened. If we are considering selecting $best_{\widehat{d}}$, then it must have been the case that $best_f$ was unsuitable for expansion. This gives us the second half of the rule for selecting $best_{\widehat{d}}$. This is simply the negation of the rule for selecting $best_f$. We then apply the same strengthening to the rule for selecting $best_{\widehat{f}}$. As it is the last node to be considered, the first two rules must have failed.

selectNode

```
1.  if $\widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f)$ then $best_{\widehat{d}}$
2.  if $\widehat{f}(best_{\widehat{d}}) > w \cdot f(best_f) \wedge \widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ then $best_{\widehat{f}}$
3.  if $\widehat{f}(best_{\widehat{d}}) > w \cdot f(best_f) \wedge \widehat{f}(best_{\widehat{f}}) > w \cdot f(best_f)$ then $best_f$
```

We can apply the same techniques to obtain a more precise definition of *selectNode* as well. As before, we leave the first rule, that for selecting $best_{\widehat{d}}$, untouched. The rule for selecting $best_{\widehat{f}}$ is strengthened by adding the negation of the first rule to its condition. This makes sense because we would only consider the second rule if the first failed. Finally, we form the rule for selecting $best_f$ by stating what the relationship between $best_{\widehat{d}}$, $best_{\widehat{f}}$ and $best_f$ must be for the first two rules to fail.

```
selectNode_opt
1.  if $incumbent = None \vee \widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f)$ then $best_{\widehat{d}}$
2.  else if $\widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ then $best_{\widehat{f}}$
3.  else $best_f$
```

Figure 7-45: Optimistic node selection function for EES

```
selectNode
1.  if $\widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f) \wedge \widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ then $best_{\widehat{d}}$
2.  if $\widehat{f}(best_{\widehat{d}}) > w \cdot f(best_f) \wedge \widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ then $best_{\widehat{f}}$
3.  if $\widehat{f}(best_{\widehat{f}}) > w \cdot f(best_f)$ then $best_f$
```

The proceeding is a simple restatement of the strengthened *selectNode*. We have added the condition $\widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ to the rule for the selection of $best_{\widehat{d}}$. This adds no new restrictions of the rule, as $\widehat{f}(best_{\widehat{d}}) \geq \widehat{f}(best_{\widehat{f}})$, but it does make it identical to the rule from *selectNode*$_{con}$. The rule for $best_f$ has been altered for the same reason. We've removed a redundant statement rather than adding one. It is now obvious that *selectNode* and *selectNode*$_{con}$ are equivalent, modulo order of course.

7.12.10 Optimistic

In contrast to the above 'conservative' approach for selecting nodes, the optimistic node selection function we are about to discuss actually produces a different search order from the original *selectNode* function. This particular select node function, described in Figure 7-45, is optimistic in that it will expand nodes that it cannot immediately prove to be within the current suboptimality bound. We see this in line 1 of Figure 7-45, where $best_{\widehat{d}}$ may always be selected for expansion so long as there is no incumbent solution. Once an incumbent solution is found, *selectNode*$_{opt}$ and *selectNode* are equivalent.

As *selectNode*$_{opt}$ cannot guarantee the w-admissibility of the nodes it returns in the same way *selectNode* does, we must find another way to ensure returned solutions have

```
EESGoalGen(root, w)
1.  $open \leftarrow \{root\}$
2.  $cost \leftarrow \infty$
3.  $incumbent \leftarrow None$
4.  while $open \neq \{\} \wedge w \cdot f(best_f) < cost$
5.    let $n = selectNode_{opt}$ in
6.      $open \leftarrow open - \{n\}$
7.      for each child $c$ of $n$
8.        if $f(c) < cost$
9.          if $goal_p(c)$
10.           $incumbent \leftarrow c$
11.           $cost \leftarrow g(c)$
12.         else $open \leftarrow open \cup \{c\}$
13. return $incumbent$
```

Figure 7-46: Optimistic EES with goal testing on node generation

bounded suboptimality. Effectively, any optimistic version of EES must be a sort of limited anytime algorithm. Potentially, we could produce two solutions, the first solution that is not w-admissible, and a second solution that is. Pseudo-code for an optimistic EES algorithm is provided in Figure 7-46. This algorithm differs from EES with goal testing on node generation only in the function used for node selection.

There are two reasons to prefer an optimistic node selection function to the original $selectNode$. The first is that $selectNode_{opt}$ may encounter its first solution far faster than $selectNode$. This can happen because $selectNode$ will always prove suboptimality bounds, and thus select $best_f$ for expansion quite frequently, only in the very extreme case where $w = 1$ would we want to return a solution by $best_f$, as it will always be an expensive to find optimal cost solution. Finding a solution early on is beneficial because it allows for more opportunities to prune nodes. This, as we noted with EES with goal testing on node generation, can reduce solving time.

The second reason to prefer an optimistic approach is that it may actually reduce the cost of proving a solution is within a bounded factor w-of optimal. If we have a solution in hand, we know exactly what nodes need to be expanded in order to prove that the solution is within the bound. This is the case in an optimistic variant of EES. The normal $selectNode$ rule uses $\widehat{f}$ to guess what nodes must be expanded in order to prove the suboptimality bound. If our guesses are bad, and they may well be, this could lead to needless effort.

On the other hand, the normal $selectNode$ function will never find solutions outside of the desired suboptimality bound. Because $selectNode_{opt}$ may, it runs the risk of needing to do two disjoint searches over the space, resulting in far more expansions for the optimistic EES than the regular EES. The chance that $selectNode_{opt}$ will return a solution outside of the desired suboptimality bound on the first iteration hinge on the accuracy of $\widehat{h}$. If $\widehat{h}$ is very accurate, or consistently underestimates h^*, then the chances that the solution found by $selectNode_{opt}$ will be outside of the suboptimality bound are low. If $\widehat{h}$ is inaccurate, the opposite is true.

In Figure 7-47, we see three plots comparing the performance of explicit estimation

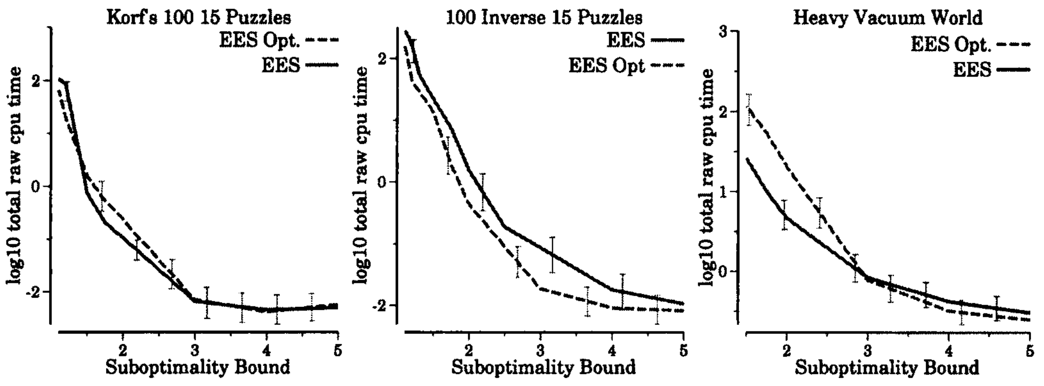


Figure 7-47: A performance comparison of EES and EES Opt

search using *selectNode* and $selectNode_{opt}$, labeled in the plots as EES and EES Opt. respectively. We present results for three of our benchmark domains: 100 instances of the 15-puzzle originally used in Korf's paper on IDA* [32], the same puzzles with a different set of action costs, and a robotic vacuuming domain. The domains are explained in detail in Chapter 3 with their most interesting features described in Table 3-1. The plots in this figure all follow the same layout: the suboptimality bound is listed on the x-axis. On the y-axis, we show the time required to find a solution at the given suboptimality bound in seconds on a log scale. The line presents the mean value of solving time, and the error bars show a 95% confidence interval about the mean.

As we can see from the three plots here, there is little difference between *selectNode* and $selectNode_{opt}$ in terms of performance on these three problems. There is no discernible difference in the standard 15-puzzle (left panel), EES using $selectNode_{opt}$ has slightly better performance in the inverse cost 15-puzzle shown in the center panel, and EES using the original *selectNode* dominates EES using $selectNode_{opt}$ for the heavy vacuum problems shown in the right panel. While, for very accurate $\widehat{h}$'s and domains with many cycles, we would expect $selectNode_{opt}$ to outperform *selectNode*, realistically we don't know the accuracy of our heuristics a priori, making it difficult to know which of EES and EES Opt. will have the best performance. For the domains and heuristics considered here, the difference between the two approaches is not particularly large, nor is it consistently in one direction as we saw in Figure 7-47

7.12.11 Lesion Study Of Expansion Order

We now turn to the question of whether *selectNode* is more complicated than it needs to be. One way of reducing the complexity of the algorithm is to reduce the number of nodes being considered by *selectNode*. This reduces the complexity of selecting the next node for expansion. It may also remove the need for maintaining the set of nodes from which $best_{\widehat{d}}$ is selected. This would reduce the overhead of the algorithm. We consider three lesioned versions of *selectNode*; each ignores one rule.

selectNode$_{l1}$

```
1.  if $\widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f)$ then $best_{\widehat{d}}$
2.  else $best_f$
```

In *selectNode$_{l1}$*, we expand either $best_{\widehat{d}}$, the node that is nearest to a solution that we estimate to be w-admissible, or $best_f$. We would expand this node in order to prove the bounds on the solution represented by $best_{\widehat{d}}$. Note that within this formulation $best_{\widehat{f}}$ is still present. We use it to define $best_{\widehat{d}}$. While this lesioned version of *selectNode* performs well at high weights, it can have trouble at tight suboptimality bounds. This is because the gap in quality between $best_f$ and $best_{\widehat{d}}$ can be much larger than the gap in quality between $best_{\widehat{f}}$ and $best_{\widehat{d}}$, which is fixed at w. It may be difficult to prove that $best_{\widehat{d}}$ is within the bound precisely because of this gap. In these situations, expanding $best_{\widehat{f}}$ and pursuing the solution estimated to have optimal cost, as EES would, is the best course of action as we will soon see.

selectNode$_{l2}$

```
1.  if $\widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ then $best_{\widehat{f}}$
2.  else $best_f$
```

In *selectNode$_{l2}$* we ignore $best_{\widehat{d}}$, choosing instead to pursue the node that appears to be the furthest along on a path to an optimal solution, $best_{\widehat{f}}$, and those nodes needed to prove that the optimal solution is within our desired bound, represented by $best_f$. While this approach is effective for tight suboptimality bounds where even the suboptimal solutions must be nearly optimal, for generous bounds, the search fails to become sufficiently greedy. If $\widehat{f}$ is wrong by even a small amount, the effort required to find the optimal solution becomes quite large [24]. The ability to select from all nodes that appear to be w-admissible allows us to skirt this problem and provides considerable utility in domains where the shortest and the cheapest solutions are very different.

selectNode$_{l3}$

```
1.  if $\widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f)$ then $best_{\widehat{d}}$
2.  else $best_{\widehat{f}}$
```

l3 is essentially *selectNode$_{opt}$*, except that it has no mechanism for ever enforcing the suboptimality bound. We mention *selectNode$_{l3}$* merely for completeness sake; it is the last function resulting from removing a single rule from *selectNode*. The previous discussion of *selectNode$_{opt}$* covers a more realistic implementation of such a lesioned expansion rule.

We might also consider lesioned variants where two rules are removed. These provide A*, a greedy search on $\widehat{d}$, and a greedy search on $\widehat{f}$. None of these is particularly interesting when discussing bounded suboptimal search algorithms as the first isn't suboptimal and the last two aren't bounded. We will discuss the later two in Section 5.6.

Figure 7-48 compares the performance of the standard *selectNode* with the two lesioned variants on the standard 15 puzzle. Results are presented in terms of nodes generated (on a log scale) in order to remove differences caused by differing overheads. *selectNode$_{l2}$*, for example, must not maintain *focal* as it never expands $best_{\widehat{d}}$. We show three algorithms in the plot, EES using the standard *selectNode* function, labeled EES, EES using *selectNode$_{l2}$*, labeled EES - L2, and EES using *selectNodel*1, labeled EES - L1.

There are two interesting phenomena displayed in Figure 7-48. First, there are places where both lesioned *selectNode* functions converge on the original *selectNode*. For *selectNode$_{l1}$*, this is for loose suboptimality bounds, where $best_{\widehat{d}}$ is very likely to be selected for expansion. For *selectNode$_{l2}$*, it is for tight suboptimality bounds, where $best_{\widehat{d}}$ is unlikely to be

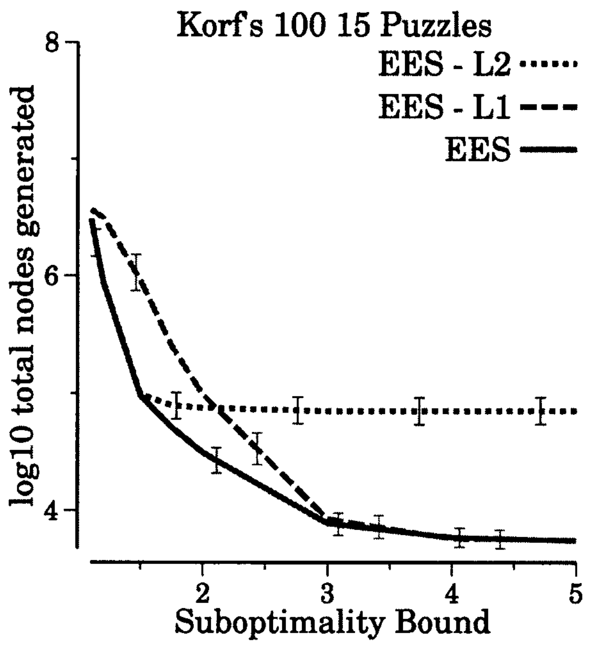


Figure 7-48: Comparing the performance of lesion *selectNode* functions

selected. A particular suboptimality bound in combination with our inadmissible heuristic estimators may effectively lesion our expansion order.

The second thing to note is that $selectNode_{l2}$ never becomes as greedy as the other two approaches. While we might initially suspect this is because it never considers $\widehat{d}$, this evaluation is performed on a unit-cost domain where $\widehat{h} = \widehat{d}$, so this isn't strictly true. The difference is more nuanced; $selectNode_{l2}$ never considers the inadmissible estimate on its own. If our corrections were perfect, this wouldn't matter; a greedy search on f^* is the same as a greedy search on h^* (and d^*) in unit-cost domains. However, because our inadmissible estimates are often imperfect, incorporating cost-incurred into the node evaluation function can lead to a more conservative search order, as we see in the plot.

7.12.12 Summary

This section presented the explicit estimation search algorithm in detail. We discussed how its search order, defined by *selectNode* shown in Figure 7-39 is nearly a direct implementation of the stated goal of bounded suboptimal search. It would be identical, but we use estimates of solution length as a proxy for estimating search effort. We also discussed visualizations of the expansion order of EES. This showed that EES does indeed behave as our intuitions about the algorithm would suggest. Early on, the algorithm expands nodes to raise its lower bound on solution cost to be more in line with it's inadmissible estimate of optimal solution cost. Once this is done, EES proceeds more or less directly towards a goal.

Our discussion of alternate expansion orders showed that *selectNode* is indeed the proper definition of "best" for bounded suboptimal search. An apparently more conservative approach, $selectNode_{con}$ was shown to be equivalent to *selectNode* upon further examination. Although *selectNode* and $selectNode_{opt}$ did differ in which nodes they would consider for expansion, we saw in Figure 7-47 that the difference in performance between EES and EES Optimistic was not particularly large, nor was it consistently in favor of one algorithm over the other. Finally, our discussion of lesioned expansion orders show that *selectNode* is exactly as complicated as it needs to be in order to have good performance.

7.13 Suboptimality Bound vs. Nodes

The previous evaluations of the bounded suboptimal search algorithms compared their performance in terms of actual running time. That evaluation was fair in the sense that it took algorithm overhead into account. While we are often concerned with the question of which algorithm takes less time to solve a problem, we may also care about the number of states that need to be considered by a search. Such an evaluation is interesting because it says something about the scalability of the algorithms, as a search which examines more nodes will exhaust memory faster. Similarly, search is fundamentally limited by the cost of examining states, as much of the other computation in a search algorithm can be optimized away or tuned to the point of not introducing too much cost. Thus, looking at the number of nodes generated by a search tells us something about their relative performance in the limit of infinite optimization. Such an evaluation follows.

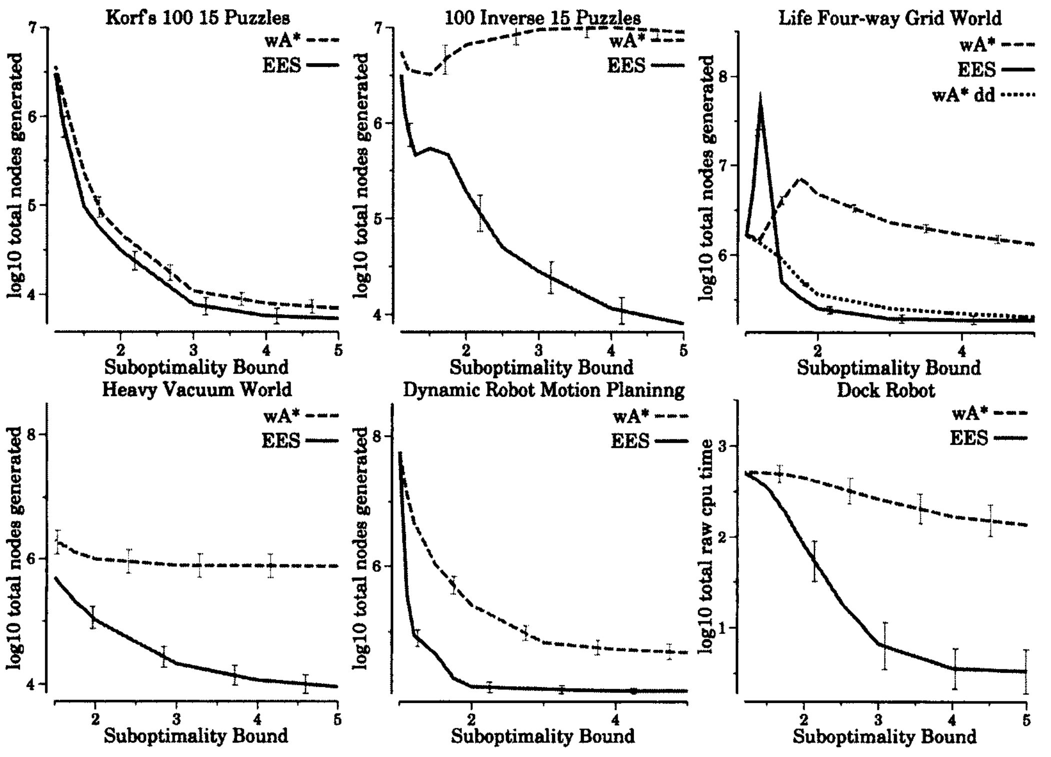


Figure 7-49: Performance of weighted A* search: suboptimality bound vs. nodes generated

7.13.1 Weighted A*

Figure 7-49 compares the weighted A* search algorithm with explicit estimation search across a wide variety of benchmarks. Here, we are examining the performance of algorithms as a function of the number of nodes they generate, shown on the y-axis in log scale. In these evaluations, a node is generated if it was generated by expanding a node; that is, we count duplicates as generated, even though they will be discarded before being inserted into open.

Unlike the previous comparison in Figure 7-5, which was quite favorable for weighted A*, we see here that EES effectively dominates weighted A* with the notable exception of a small range of suboptimality bounds in life-cost grid pathfinding. We will discuss the brief exception momentarily. Recall that the plots here ignore algorithm overhead entirely. As we discussed previously, weighted A* has very little overhead, whereas EES must compute additional heuristics and maintain additional orderings over nodes. Thus, when ignoring algorithm overhead, we ignore the primary advantage weighted A* has over EES.

The exception to EES' dominance over weighted A* is in the problem of life-cost grids. In this domain, there are a great many duplicates, which weighted A* can ignore (wA* dd in the legend). However, we see that even weighted A* with re-expanding duplicates is better than EES for a small range of weights. Why should EES re-expand more nodes than weighted A*?

The answer comes from an examination of their expansion orders. For low w, weighted A* will expand nodes in approximately A* order. Since A* requires no re-expansions, we would expect a search order that is almost identical to require relatively few expansions. EES, on the other hand, deviates as much as possible from an A* search order. When expanding a node it effectively shoots out a greedy search on d from that node until all of the children look to be outside the suboptimality bound. Greedy searches often reach nodes by suboptimal paths. In the case of a near optimal search, many of those nodes will have to be re-expanded in order to find a solution within the bound. This is why EES is slower than even weighted A* in this small area of the life-grid problems.

7.13.2 Dynamically Weighted A*

Figure 7-50 compares the dynamically weighted A* search algorithm with explicit estimation search across a wide variety of benchmarks. This evaluation isn't particularly revealing, except that the performance difference between dynamically weighted shown in the plots here is larger than that shown in Figure 7-10. This is because, like weighted A*, dynamically weighted A* has low per-node overhead relative to EES. However, unlike weighted A*, dynamically weighted A* was never competitive with EES on the benchmarks evaluated here.

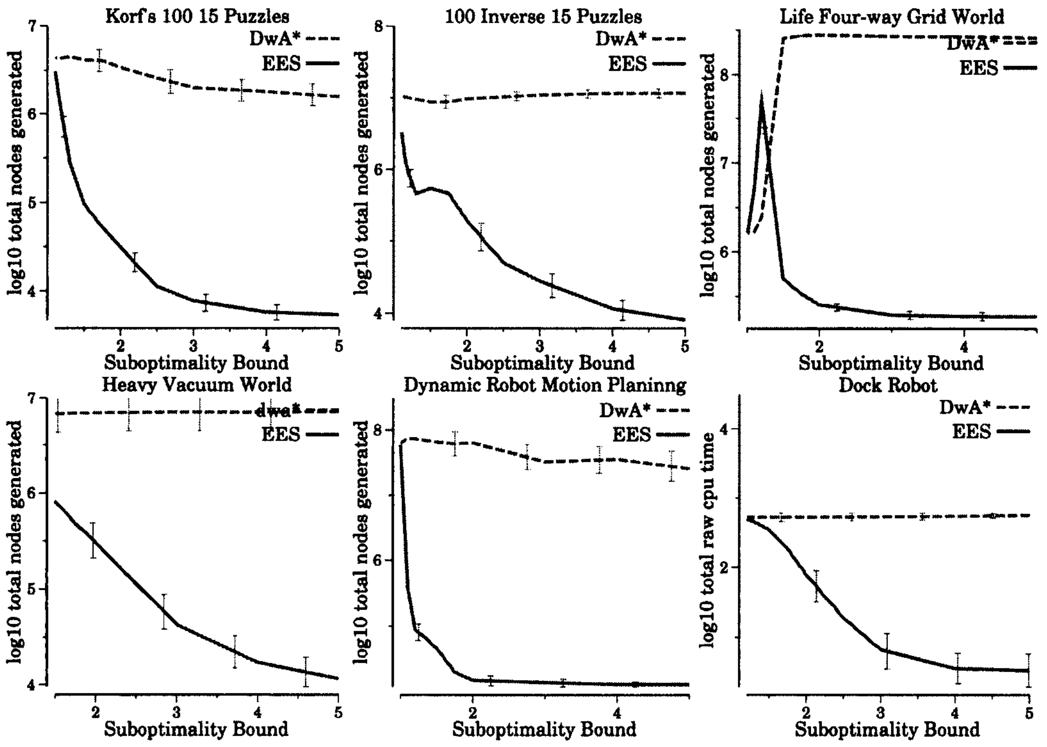


Figure 7-50: performance of dynamically weighted A* search: suboptimality bound vs. nodes generated

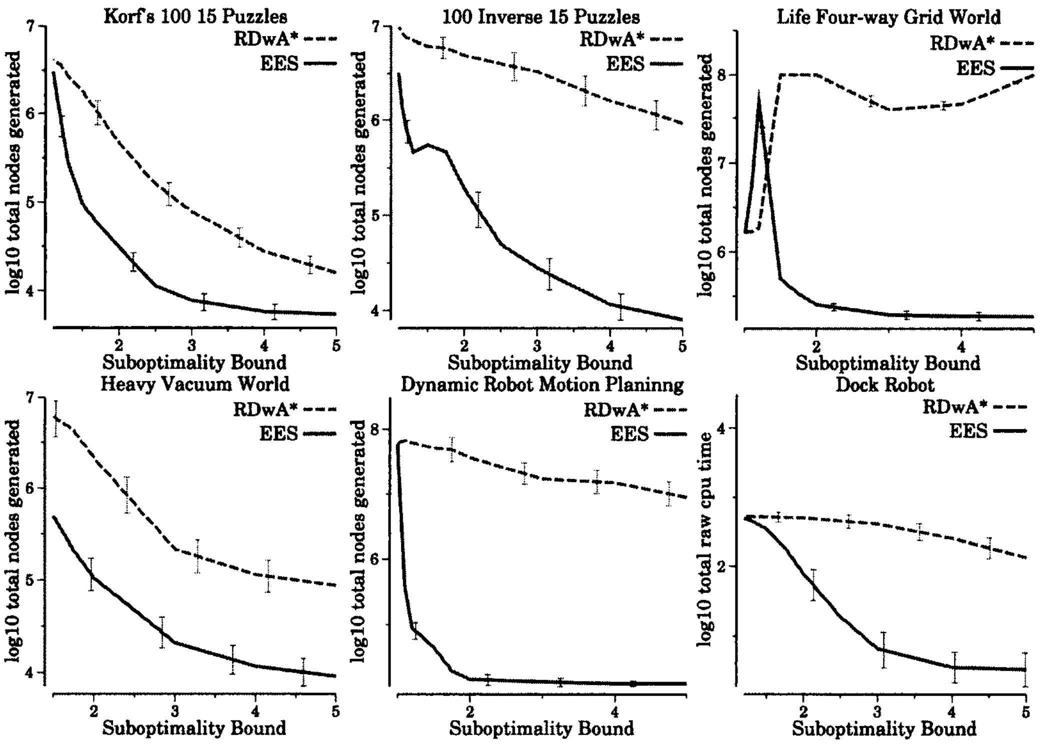


Figure 7-51: Performance of revised dynamically weighted A* search: suboptimality bound vs. nodes generated

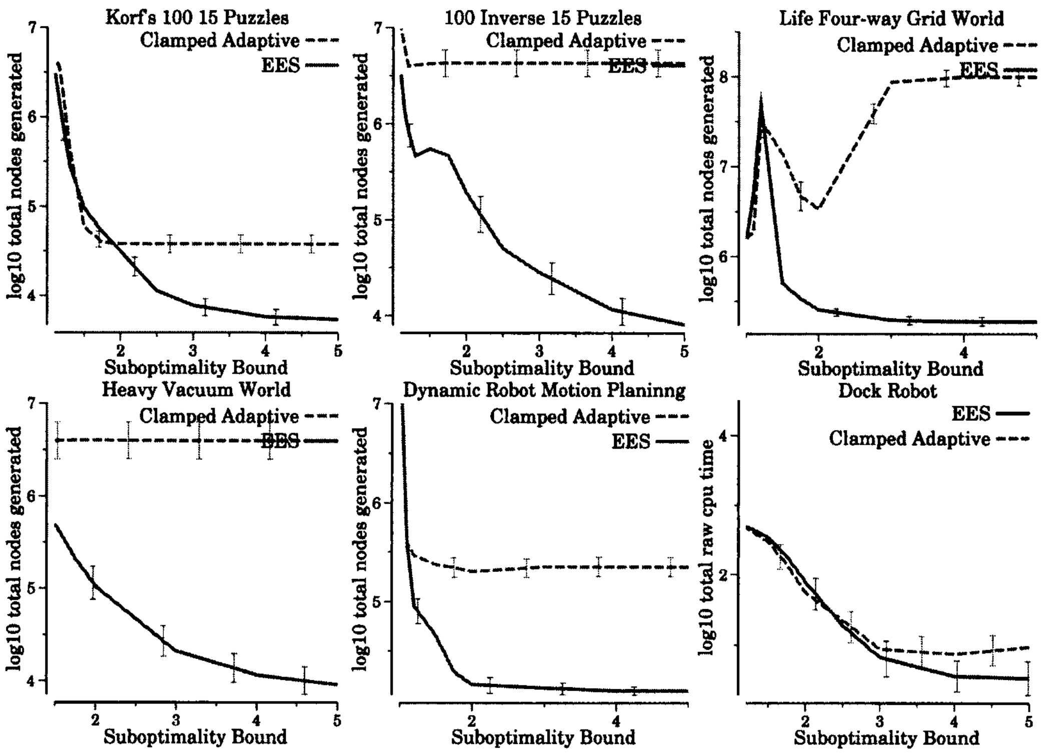


Figure 7-52: Performance of clamped adaptive search: suboptimality bound vs. nodes generated

7.13.3 Revised Dynamically Weighted A*

Figure 7-51 compares the revised dynamically weighted A* search algorithm with explicit estimation search across a wide variety of benchmarks. The comparison is much like that provided for dynamically weighted A*. It is not particularly surprising because revised dynamically weighted A* was not an especially competitive algorithm on the benchmarks considered in this dissertation.

7.13.4 Clamped Adaptive

Figure 7-52 compares the clamped adaptive search algorithm with explicit estimation search across a wide variety of benchmarks. The interesting results here are in the dock robot domain, where we see that the good performance of clamped adaptive relative to EES is only partially a result of reduced overhead. For small suboptimality bounds ($w < 2$), we see that clamped adaptive search has performance on par with that of explicit estimation search.

Recall that when we performed the lesioned evaluation of EES, we saw that for small suboptimality bounds, search on $best_{\widehat{f}}$ and $best_f$ exclusively performed about as well as the full fledged EES algorithm. The expansion order for clamped adaptive is not that different than that of the lesioned EES. Clamped adaptive will assign a value $w \cdot f(n)$ to a node if $g(n) + w \cdot \widehat{h}(n)$ is too large to be provably within the bound at the time of expansion. These are exactly the nodes that EES would deem unqualified for expansion by $best_{\widehat{f}}$. Further, all of these nodes will be sorted in f-order because a linear scaling of all f-values does nothing to impact the order of nodes. The ordering of nodes qualified for expansion by $best_{\widehat{f}}$ will differ between EES and clamped adaptive, as only scaling the heuristic portion of the node evaluation function can change search order. This is why we see some deviation in terms of the number of nodes expanded.

7.13.5 A^*_ϵ

Figure 7-53 compares A^*_ϵ with explicit estimation search across a wide variety of benchmarks. The interesting thing to note here is that the good performance of A^*_ϵ is not entirely attributable to overhead. For large suboptimality bounds $w > 3$, it appears that search on d is actually sometimes faster than search on $\widehat{d}$. We see this in life grids, heavy vacuums, and the inverse 15 puzzle. This seems to be in conflict with the results reported in the previous chapter, where we showed that the learned heuristic was almost always better than the base heuristic. However, there we were talking about h and $\widehat{h}$, not d and $\widehat{d}$.

Further, we aren't really searching greedily on either of these values. We're searching greedily over an, admittedly very large, subset of nodes. Taking this subset is likely pruning away many of the poorer options that the learned heuristic would not expand, but that the base heuristic might. Essentially, we're reducing the advantage that the learned heuristic has over the base heuristic by building subsets of the nodes.

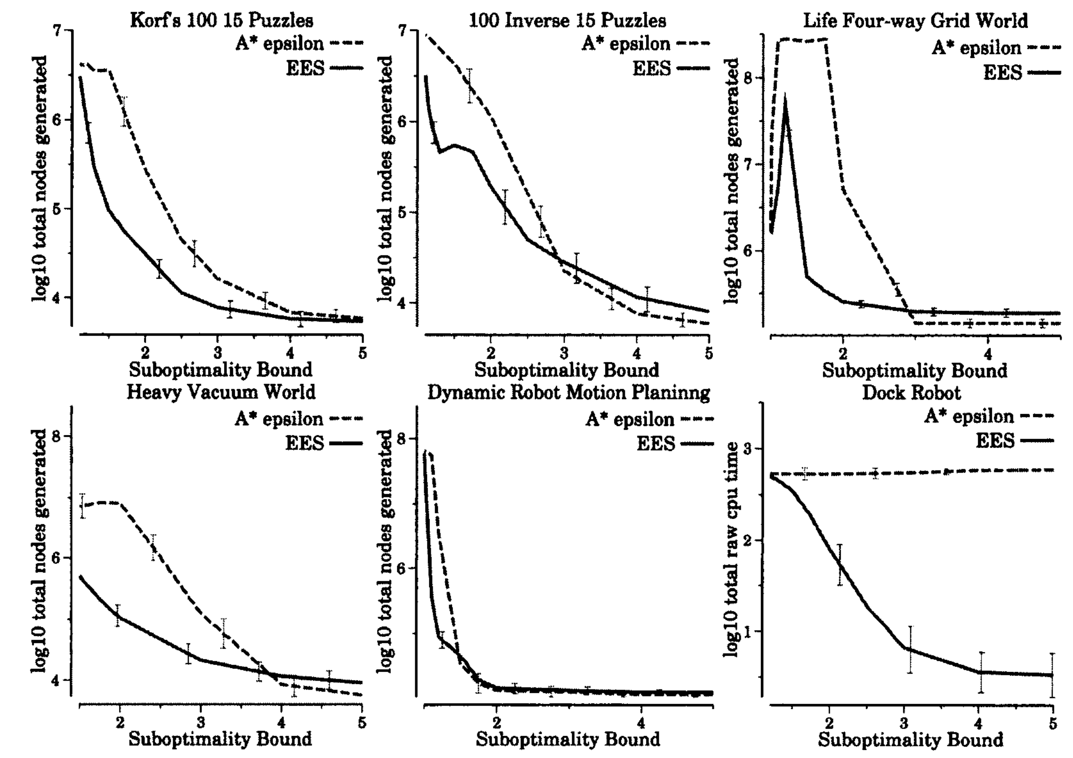


Figure 7-53: Performance of A^*_ϵ search: suboptimality bound vs. nodes generated

7.13.6 Optimistic Search

Figure 7-54 shows the performance of optimistic search relative to the suboptimality bound, where performance is measured by the number of nodes expanded during the search. The interesting thing to note from these results is that the good performance of optimistic search in the sliding tiles domain is not entirely the result of reduced overhead. Optimistic search consistently expands fewer nodes than EES.

For lower bounds, where the difference is noticeable, we suspect that this is the result of $\widehat{h}$ being too high. We saw in Chapter 5 that path based correction often far over-estimated the cost-to-go on tiles puzzles (see for example Figure 5-3. If $\widehat{h}(n) > h^*(n)$ for many nodes, then EES will do too much cleanup as it will incorrectly assume that most solutions actually lie outside of the bound. Optimistic search, on the other hand can not do too much cleanup because it waits until a solution is in hand to start. That way it can always do the minimum amount of cleanup necessary for the solution it finds.

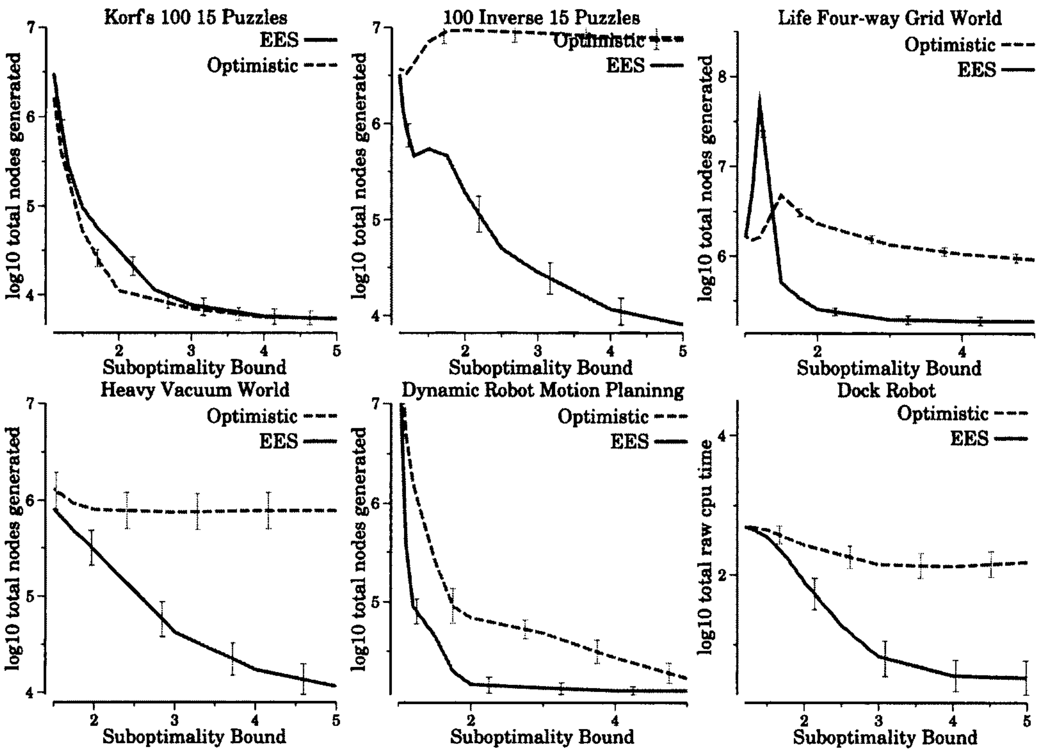


Figure 7-54: Performance of optimistic search: suboptimality bound vs. nodes generated

7.13.7 Skeptical Search

Figure 7-55 shows the performance of skeptical search relative to the suboptimality bound, where performance is measured by the number of nodes expanded during the search. The results here are very much in line with those in Figure 7-37 and reveal that in two domains, the competitive performance of skeptical search is a result of having less per-node overhead than EES.

7.14 Analysis on Explicit Graphs

In previous sections of this chapter, we have examined the empirical performance of bounded suboptimal search algorithms on a variety of benchmark domains. We saw that, in general, algorithms which took estimates of the number of actions remaining in a solution into account outperformed those that did not. Similarly, algorithms that took inadmissible

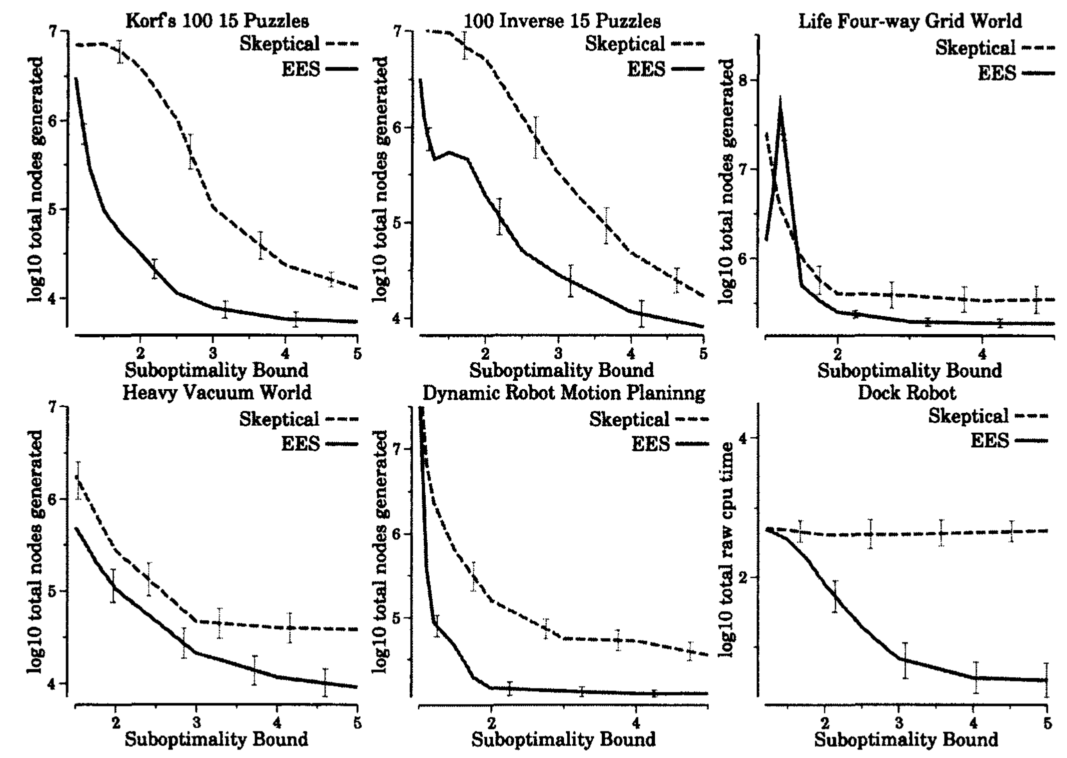


Figure 7-55: Performance of skeptical search: suboptimality bound vs. nodes generated

estimates of the cost-to-go into account tended to perform better than those algorithms that only relied on admissible heuristics for this value.

In this section, we will look at algorithm performance on two families of explicit graphs. The graphs are constructed to make two points: even if we had perfect information, algorithms that weight the cost-to-go heuristic cannot ever minimize solving time under a bound because they do not prefer shorter paths; algorithms that use actions-to-go estimates to prefer shorter paths are also fatally flawed and cannot always prefer the shortest path in the bound.

7.14.1 An Inconvenient Graph

Figure 7-56 shows a template for a family of graphs, each of which has exactly two solutions. The first, cost-optimal solution that goes from the starting node S to the goal node G over a, and the second solution that goes from S to G over b. We will refer to these as $path_a$

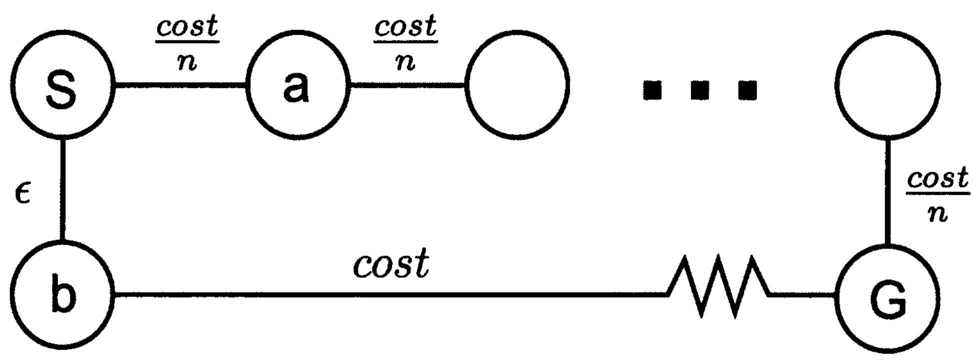


Figure 7-56: Explicit graph that thwarts cost-focused search

and $path_b$ respectively.

$path_a$ and $path_b$ are related in the following ways: $path_a$ is marginally cheaper than $path_b$, $cost(path_b) = cost(path_a) + \epsilon$, however $path_b$ has a length of 2, while $path_a$ has a length of n. Thus, $path_b$ is arbitrarily shorter than $path_a$. Obviously, for all suboptimality bounds other than 1, we would prefer our search algorithms to find $path_b$ rather than $path_a$. Unfortunately nearly every algorithms we have previously discussed will find $path_a$ regardless of the suboptimality bound, even when the heuristics are perfectly informed.

To show that an algorithm will always find the longer solution, we need merely show that it's expansion order prefers node a to node b. So long as a is considered favorable to b, then all nodes beyond a will be favorable to b. Any algorithm that works by placing additional emphasis on cost-to-go estimates will be fooled by the above graph because $h^*(a) < h^*(b)$.

Weighted A*

When S is expanded, a and b are placed in the open list. As $h^*(a) < h^*(b)$, $f'(a) < f'(b)$, and thus weighted A* will prefer node a. For all nodes n along $path_a$ beyond node a, $h^*(n) < h^*(a) < h^*(b)$, and thus $f'(n) < f'(b)$. Thus, weighted A* finds the longer, but cheaper path.

Dynamically Weighted A*

When S is expanded, a and b are placed in the open list. As $h^*(a) < h^*(b)$ and a and b are at the same depth, dynamically weighted A* will prefer node a to node b. All nodes n along $path_a$ beyond a will have lower cost-to-go values and they will be at deeper depths. Since dynamically weighted A* rewards depth, all nodes n along path $path_a$ will look better than node b as they will be deeper and have smaller h^*-values.

Clamped Adaptive

As we are discussing a world in which $h(n) = h^*(n)$, clamped adaptive search is equivalent to weighted A* search. This is because the clamping behavior will never be observed. Since $\widehat{h}(n) = h^*(n)$, clamped adaptive will take one of $w \cdot f^*(n)$ or $g(n) + w \cdot h^*(n)$, whichever is smaller. The two values are equivalent at the root, but beyond node s, $g(n) + w \cdot h^*(n)$ will always be less than $w \cdot f^*(n)$. If it wasn't, weighted A* wouldn't be guaranteed to return a solution within the bound given an admissible heuristic (h^* never over-estimates the cost-to-go), and thus we would have a contradiction. Clamped adaptive runs into the trap because in this setting it is running weighted A* with a perfect heuristic.

Optimistic Search

In the initial phase, optimistic search runs weighted A* search with a higher weight than the desired suboptimality bound. It therefore falls into the trap by the same argument as weighted A* search.

Skeptical Search

In the initial phase, skeptical search runs weighted A* on an inadmissible heuristics. Since the heuristic used in this discussion is perfect, $h^*(n) = \widehat{h}(n) = h(n)$. Thus, skeptical search falls into the trap by the same argument that weighted A* does.

7.14.2 AlphA*

AlphA* always evaluates a node with either $f(n)$ or $w \cdot f(n)$. In this setting, since we are dealing with perfect heuristics, this becomes $f^*(n)$ and $w \cdot f^*(n)$. None of the proposed α-functions would make us penalize node a and not node b. Therefore they will be sorted according to their f^*-values, which will make AlphA* prefer node a to node b. As search progresses along $path_a$ from a, the f^*-values of nodes on this path will remain constant, and thus be preferable to node b even if we do not penalize b. Thus, AlphA* finds the cheaper, but longer path.

Revised Dynamically Weighted A*

Is the only algorithm which scales cost-to-go values that does not fall into the trap demonstrated by Figure 7-56. There are some values of w for which revised dynamically weighted A* will find the long solution, but there are many more where it will find the longer, albeit more expensive solution.

The reason revised dynamically weighted A* has different behavior is because it scales the cost-to-go estimate based on the actions-to-go estimate. While $h^*(a) < h^*(b)$, $d^*(b) < h^*(a)$. As $d^*(n)$ is defined to be the number of actions along the optimal cost path from n to a goal, $f'_{rdwa*}(a) = f^*(a) + (w-1) \cdot \frac{n-1}{n} \cdot \frac{(n-1) \cdot cost}{n}$ while $f'_{rdwa*}(b) = f^*(b) + \frac{w-1}{n} \cdot cost$. We can show via algebra that a will often be preferable to b:

$$\begin{array}{rcl}
f'_{rdwa*}(b) & < & f'_{rdwa*}(a) \\
f^*(b) + \frac{w-1}{n} \cdot cost & < & f^*(a) + (w-1) \cdot \frac{n-1}{n} \cdot \frac{(n-1) \cdot cost}{n} \\
f^*(a) + \epsilon + \frac{w-1}{n} \cdot \frac{n \cdot cost}{n} & < & \\
\epsilon + \frac{w-1}{n} \cdot \frac{n \cdot cost}{n} & < & (w-1) \cdot \frac{n-1}{n} \cdot \frac{(n-1) \cdot cost}{n} \\
n^2 \cdot \epsilon + (w-1) \cdot (n \cdot cost) & < & (w-1) \cdot (n-1) \cdot ((n-1) \cdot cost) \\
\frac{n^2 \cdot \epsilon}{w-1} + (n \cdot cost) & < & (n-1) \cdot ((n \cdot cost) - cost) \\
\frac{n^2 \cdot \epsilon}{w-1} + (n \cdot cost) & < & n^2 \cdot cost - n \cdot cost - n \cdot cost + cost \\
\frac{n^2 \cdot \epsilon}{w-1} & < & n^2 \cdot cost - 3 \cdot n \cdot cost + cost \\
\frac{n^2 \cdot \epsilon}{w-1} & < & (n^2 - 3 \cdot n + 1) \cdot cost \\
\frac{\epsilon}{w-1} & < & (1 - \frac{3}{n} + \frac{1}{n^2}) \cdot cost
\end{array}$$

We now have that, in all situations when $\frac{\epsilon}{w-1} < (1 - \frac{3}{n} + \frac{1}{n^2}) \cdot cost$, b is preferred over a. Since ϵ is supposed to be very small (but non-zero), the largest value the left hand side of the equation could ever have is 1, in the case where $w = 1 + \epsilon$. Since we wanted to show that b is preferable to a for all weights over many graphs, we need to show that $(1 - \frac{3}{n} + \frac{1}{n^2}) \cdot cost > 1$.

As the figure is drawn, n is at least 4 and *cost* is always larger than 0. Thus, a will be preferred to b in all cases where *cost* is larger than $\frac{16}{5}$. As the size of the graph increases, the minimum value of *cost* for the equation to hold also decreases.

A_ϵ^* and EES Do the Right Thing

A_ϵ^* and EES do the right thing on the graph shown in Figure 7-56. That is, they find the shorter, but ϵ more expensive path. After expanding S, the root, a and b are both on open. Both A_ϵ^* and EES build a focal list based on what nodes are estimated to be within the suboptimality bound. In the case of perfect information, they build a focal list based on

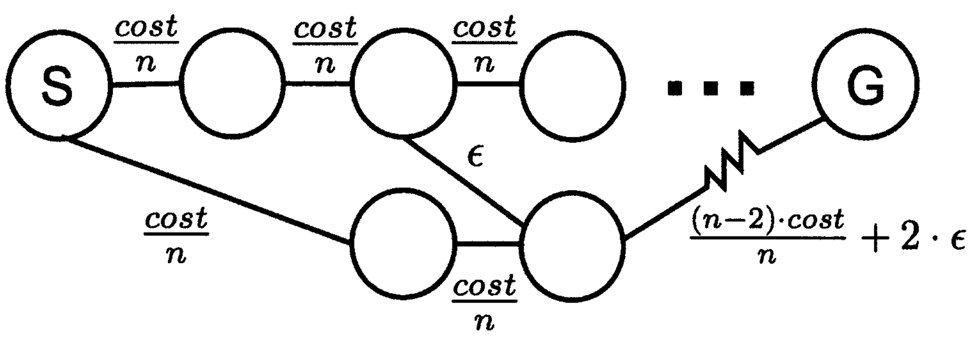


Figure 7-57: Explicit graph that thwarts EES and A_ϵ^*

what nodes are known to be within the suboptimality bound. Since $path_b$ is only ϵ worse than $path_a$, $path_b$ will be within the suboptimality bound for all bounds greater than 1. While both a and b will be on focal, both A_ϵ^* and EES will prefer b, as it has a smaller d^*-value.

7.14.3 Confusing EES and A_ϵ^*

The graph shown in Figure 7-57 is meant to demonstrate a flaw in algorithms which do not weight the cost-to-go heuristic in order. There are three paths through the graph shown in Figure 7-57, a length n cost-optimal path over a, an ϵ suboptimal path over nodes b and b' of length $n+1$, and a $2 \cdot \epsilon$ suboptimal path over node b'. In this graph, all algorithms discussed in the dissertation will prefer the long cost-optimal path over a.

This is because the graph is constructed to give a misleading value of actions-to-go for node c. Recall that $d^*(n)$ is defined to be the number of actions in the *cost-optimal* solution beneath node n. Thus, $d^*(c) = d^*(a)$, and so a will be preferred under any reasonable tie-breaking scheme. Once a is expanded, all other nodes considered along $path_a$ will have lower d-values than node c, and thus we will get the cheaper but longer solution.

7.14.4 Summary

In this section we discussed a pair of explicit graphs which highlighted weaknesses in the algorithms described in this chapter. The first graph showed that algorithms which work by weighting cost-to-go estimates can easily be tricked into preferring arbitrarily long paths even when a very short path within the desired suboptimality bound exists. This is because they do not consider the number of actions remaining when determining expansion order. The second graph showed how our definition of d can lead algorithms which explicitly consider actions to go to be mislead. In reality, we need more expressive heuristics that would let us estimate the length of the shortest solution beneath a node within the bound. Unfortunately, it is unknown how to compute such a heuristic.

7.15 Discussion

Explicit estimation search provides a substantial improvement over the previously proposed algorithms for bounded suboptimal search. It is faster than previous approaches for a given suboptimality bound across a wide range of suboptimality bounds and domains. However, while it is not always the fastest algorithm, it is robust in a way that previous proposals were not. This is why EES had the lowest mean solving time of all algorithms as we saw in Figure 7-33.

However, to say that the contribution of EES is limited to faster solving times and more robust behavior is to miss the point. When designing search algorithms, we should make sure we're solving the right problem. That is the largest lesson to be taken away from explicit estimation search, and this work in general. By looking at what it was we wanted from a bounded suboptimal search algorithm and crafting a search strategy tailored to that, we ended up with a more effective approach to the problem. Similar results will be seen in anytime search and bounded-cost search.

A fixation on admissible heuristics and optimal solving has likely been harmful to the field of heuristic search as a whole. Admissible heuristics are useful for proving that solutions returned by an algorithm have certain desirable properties, for example cost-optimality or being within a bounded factor of optimal. Admissible heuristics are also incredibly useful for permanently pruning away unpromising areas of the search space. However, these two tasks represent a small part of what search algorithms need to do. They also have to find solutions, and they may need to do so quickly in order to obey restrictions on solving time. These tasks are neither easily nor best accomplished relying on admissible cost-to-go heuristics alone.

More effort needs to be put into inadmissible cost-to-go estimates, both hand crafted techniques, and ways of deriving such automatically. These heuristics are useful in all search settings, excluding perhaps cost-optimal search. Having techniques for automatically constructing inadmissible heuristics from the definition of a problem is a key part of building a theory of suboptimal search.

If we want fast algorithms, we need to be able to estimate the relative speed with which various partial solutions can be brought to completion. We didn't notice that we didn't have one in heuristic search for a great many years because we've been too focused on unit-cost domains like the sliding tiles puzzle. We still don't really have an estimate of the time it will take to convert a partial solution into a complete solution in heuristic search. We have reasonable proxies in the form of action-to-go estimates, and paying attention to these does indeed improve the speed of search algorithms. It seems likely that if we construct estimators for the desired value and guide search based on those that search can be sped up even further.

7.16 Conclusions

In this Chapter we introduced a new state of the art bounded suboptimal search algorithm, Explicit Estimation Search. Explicit Estimation Search relies on inadmissible estimates of solution cost and solution length to guide search towards easy-to-find solutions within the bound. While EES is a significant improvement over the previous state of the art in bounded suboptimal search, it is not the most important contribution of this work. This chapter, and particularly the discussion of how EES addresses the problem of bounded suboptimal search, forms a part of the foundation of the theory of suboptimal search. Specifically, we put forth a definition of the goal of bounded suboptimal search, and we pointed out several sources of information needed to address that goal. EES uses inadmissible estimates because efficient suboptimal search requires inadmissible estimates to determine what solutions are likely to be within the bound and to determine what solutions are easy to find. EES isn't the best performing bounded suboptimal search algorithm because of brilliant insight or clever optimization, it's simply the first algorithm to attempt to optimize the goal of bounded suboptimal search directly. Hopefully this is an approach that will prove useful for many areas of heuristic search and AI in general.

CHAPTER 8

BOUNDED COST SEARCH

Recently, Stern, Puzis, and Felner[61] began studying a slightly different variation on bounded suboptimal search called *bounded-cost search*: given a user-specified cost bound C, find a plan with cost less than or equal to C as fast as possible. Bounded-cost search corresponds to many realistic cost-constrained scenarios such as constructing an interesting air show for model planes, or planning a trip within a budget. They also introduced an algorithm called Potential search, abbreviated as PTS, designed for the bounded-cost search setting. PTS is a best-first search on potential — the probability that a given node will be part of a solution whose cost is no more than C. Nodes that are more likely to have a goal node beneath them are preferred.

8.1 Potential Search

Stern, Puzis, and Felner[61] define bounded-cost search in the context of heuristic shortest-path graph search: Given a description of a state space, a start state s, a goal test function and a constant C, find a path from s to a goal state with cost less than or equal to C. Potential search [61] (PTS) is a bounded cost search algorithm based on considering the potential of all nodes that have been generated but not yet expanded (i.e. nodes on *open*). The potential of a node is the probability that a solution of cost no more than C exists beneath that node. The *potential* of a node n is $PT_C(n) = Pr(g(n) + h^*(n) \leq C)$.

PTS is a best-first search on PT_C. In general, we don't know how to calculate the potential of a node. However, for some cases it is possible to order the nodes according to their potential without being able to calculate it. If we know how $h(n)$ and $h^*(n)$ are related then there are situations where we can order the nodes by their potential without being able to calculate it directly.

In particular, if $h(n)$ and $h^*(n)$ are related linearly, then we can order the nodes in order of their relative potentials without bothering to compute the potential of the two nodes. If the error in h with respect to h^* grows linearly, then we can order nodes on $f_{lnr}(n) = \frac{h(n)}{C-g(n)}$ and end up exploring the nodes in order of increasing potential without needing to compute the potential directly.

8.1.1 Potential Search on Inadmissible Heuristics

In the following evaluation, we consider two variants of Potential Search, the original (PTS in the plot) and a newer variant presented in Thayer, Stern, Felner, and Ruml [72] which uses inadmissible estimates of cost-to-go in order to calculate the potential of a node. Assuming that the inadmissible heuristic has the same relationship to true cost-to-go as the admissible heuristic, there is no reason not to use an inadmissible estimate of cost-to-go in order to estimate the potential of a node. In their evaluation and the one conducted here, Potential Search with inadmissible heuristics (PTS hîn the plot) uses the same online correction technique that we have discussed before. It is identical to PTS except that we sort nodes on $\hat{f}_{lnr}(n) = \frac{\hat{h}(n)}{C-g(n)}$.

8.1.2 Implementation Concerns

In practice, we implement $\widehat{PTS}$ as a best-first search on $\hat{f}_{lnr}(n) = \frac{\hat{h}(n)}{1-\frac{g(n)}{C}}$, where we have effectively divided the potential score of all nodes by the cost bound C. This does not affect node ordering. When we divide the potential scores of all nodes by a constant, in this case C, we preserve the relative ordering. Order is exactly what matters in a best-first search.

Restating the node ordering function this way does two things. First, it makes it clear that for large values of C, PTS and $\widehat{PTS}$ will behave like a greedy search on cost-to-go estimates (sometimes called pure heuristic search). Secondly, it avoids precision issues caused by large C values. For large C, implementing $f_{lnr}(n)$ as $\frac{h(n)}{C-g(n)}$ will result in $f_{lnr}(n) =$

0 for all nodes. For $f_{lnr}(n) = \frac{h(n)}{1-\frac{g(n)}{C}}$, we have $f_{lnr}(n) = h(n)$ for large C.

8.1.3 Drawbacks

One small drawback of the potential search technique is that, if the heuristic does not exhibit linear relative error, the above cost functions are no longer valid. This leaves us with several options. Either we construct a heuristic for every domain which we know will have linear relative error, we construct a node evaluation function that will work with whatever error model our heuristic has, or we accept that we will only expand nodes in approximate order of improving potential.

The first two are difficult. It's unclear how to construct a heuristic with linear relative error, though it appears that many admissible heuristics exhibit this property naturally, as PTS does not need to consider exceptional heuristics for the domains it uses for evaluation in the original paper. Similarly, it isn't obvious that a potential ordering function can be constructed for arbitrary heuristic error models. While it is obvious that we could simply accept an approximate best-first order for our bounded cost search, that is less than ideal. We will now discuss a technique for bounded cost search that does not rely on a measurement of potential, and thus does not suffer from these drawbacks.

8.2 Bounded Cost Explicit Estimation Search

Bounded-Cost Explicit Estimation Search [72] (BEES) considers both admissible and inadmissible estimates of cost-to-go (h and $\widehat{h}$) as well as inadmissible estimates of actions-to-go ($\widehat{d}$). BEES is inspired by EES in that both rely on estimates of solution cost and actions remaining to guide search rather than exclusively relying on lower bounds as PTS does. To suit the goal of bounded-cost search, instead of considering $best_{\widehat{d}}$ like EES, BEES considers the following node:

$$best_{\widehat{dc}} = \operatorname*{argmin}_{n \in open \wedge \widehat{f}(n) \leq C} \widehat{d}(n)$$

```
$selectNode_{bees}$
1.  if there exists $n \in open$ s.t. $\widehat{f}(n) \leq C$
2.      then return $best_{\widehat{dc}}$
3.      else return $best_f$
```

Figure 8-1: BEES node selection strategy

Note that $best_{\widehat{dc}}$ is a member of the set of all nodes in open whose estimated total cost is less than that of the cost bound. Of these, $best_{\widehat{dc}}$ is the node with the smallest $\widehat{d}(n)$. $best_{\widehat{dc}}$ is the node we estimate has the fewest actions remaining between it and a goal, among all the nodes whose estimated total cost is less than the cost bound. Again, tie breaking is an important consideration. If multiple nodes could be $best_{\widehat{dc}}$, then we should prefer the node with the least $\widehat{f}$. If this still doesn't eliminate all ties, then we should prefer nodes with lower f-values.

BEES chooses to expand either $best_{\widehat{dc}}$ or $best_f$ according the rule described above. Using this rule, BEES attempts to pursue the shortest solution estimated to be within cost bound C if it estimates that such a node exists (line 2). If BEES thinks there are no solutions within the cost bound, it expands nodes in A^* order to efficiently prove no solution exists (line 3).

This differs from the *selectNode* function of EES in that $best_{\widehat{f}}$ is never returned. In the context of bounded-cost search, it doesn't make sense to expand $best_{\widehat{f}}$, because we've estimated that the cost of the optimal solution is beyond the cost-bound C for the given problem; that is we predict the problem has no solution. If we assume the problem isn't solvable, the right thing to do is to prove that the problem isn't solvable by raising the lower-bound on optimal cost above C as quickly as possible. Thus, BEES will be more effective for bounded cost search than a modified EES because it actually addresses the problem at hand, much in the same way EES was better than previous approaches to bounded suboptimal search because it focused on addressing the problem as directly as possible.

```
$selectNode_{beeps}$
1.  if there exists $n \in open$ s.t. $\widehat{f}(n) \leq C$
2.      then return $best_{\widehat{dc}}$
3.      else return $best_p$
```

Figure 8-2: BEEPS node selection strategy

8.2.1 BEEPS

While straightforward, the previous approach ignores the potential measurement suggested by Stern et al[61]. To takes this new quantity into account, we propose Bounded-Cost Explicit Estimation Potential Search (BEEPS). In addition to $best_{\widehat{dc}}$, BEEPS considers expanding the node with the highest potential, or in other words the lowest $\widehat{f}_{lnr}(n)$.

The node selection strategy of BEEPS is exactly the same as that of BEES, differing only in the last line. When BEES decides to return $best_f$, BEEPS will return $best_p$. BEES assumes that $\widehat{f}$ is accurate and so if $best_{\widehat{dc}}$ does not exist, then there must not exist a solution to this problem within cost bound C. If that is true, then the optimal way of proving it is by expanding nodes in A* order until we have shown that there is no node with $f(n) \leq C$, proving the problem unsolvable. In contrast, BEEPS acknowledges that $\widehat{h}$ is not a perfect estimator and thus even when $best_{\widehat{dc}}$ does not exist, the problem may well be solvable. For solvable bounded-cost problems, PTS was shown to be superior to A* with pruning [61], so BEEPS reverts to $\widehat{PTS}$ instead.

Naturally, BEEPS has all of the same drawbacks as potential search. This is because it relies on the same measurement of potential. However, BEEPS only uses the potential values in the case that our inadmissible heuristics lead us to suspect that no solution exists within the desired cost bound. BEEPS only relies on potential in what is hopefully an exceptional case, while potential search relies on measurements of potential all the time. As a result, the drawbacks of using potential are less severe for BEEPS.

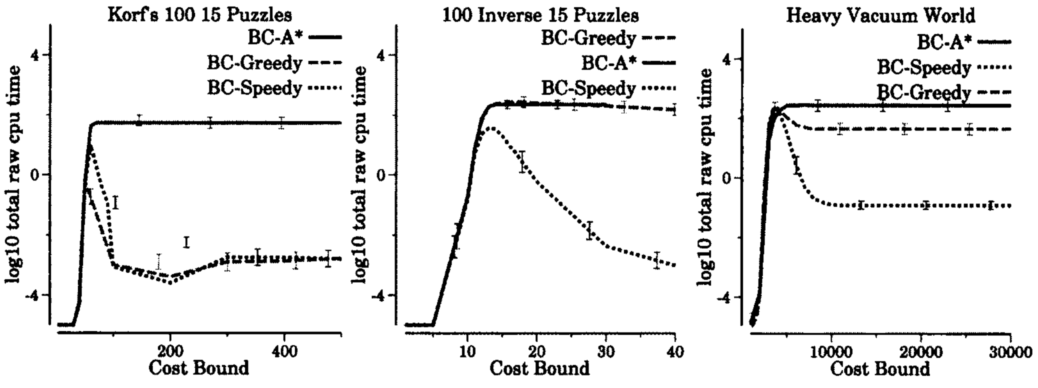


Figure 8-3: Relative performance of baseline bounded cost algorithms

8.3 Empirical Evaluation

8.3.1 Baseline Algorithms

To better understand the performance of the more advanced techniques for bounded cost search presented here, we compare the algorithms to some natural baselines for the problem of bounded cost search. These baselines work by taking algorithms for other search settings and simply adapting the algorithms to work in the bounded cost setting.

We consider three baselines: A* with pruning, greedy search with pruning, and speedy search with pruning. When we say "with pruning" what we mean is that whenever a node is generated, it's f-value is compared to the cost-bound C. Any node n with $f(n) > C$ is discarded because it cannot result in a solution within the desired cost bound.

Figure 8-3 shows the relative performance of the three previously described baseline algorithms on several heuristic search benchmarks. On the y-axis, we report the time taken to find a solution, or prove no solution exists, on a log scale. As with all previous evaluations, algorithms were run until memory was exhausted, or until ten minutes had passed. In the plot shown, the x-axis is the cost-bound supplied to the algorithm. The cost bounds are designed to start at a point where no instance under consideration contains a solution, and then scale up well beyond the point where all instances contain solutions.

There are two general trends in the results shown in Figure 8-3. The first thing to note is that for tight cost-bounds, ie low values of C, the bounded cost adaptation of A*, A*-BC in the legends, is the ideal algorithm. In fact, we can easily see that bounded cost A* is the best way of showing no solution exists within cost bound C. To prove that no solution exists to the problem with cost no more than C, we must show that any solution to the problem must have cost greater than C. To wit, we must raise the lower bound on the cost of the optimal solution of the problem to be larger than C. Starting with the root, we must expand all nodes whose f-value is no more than C. A* has the most efficient way of raising the lower bound on optimal solution cost [13].

The second thing to note is that, when many solutions exist within the cost bound, bounded cost speedy search, BC-Speedy in the legend, has the best performance. When the bound is loose, the problem becomes that of simply finding any solution, as nearly all solutions will be acceptable. When attempting to find any solution, the fastest way of doing so is generally to search in order of the estimated number of actions remaining between a node and a solution, exactly what speedy search does. In the following evaluation of bounded cost search algorithms, we will only present the speedy baseline. As we see in Figure 8-3, for the wide majority of bounds, it is the most effective baseline.

The reader may have noticed that three benchmark domains appear to be missing. Specifically, we do not show results for life-cost grids, dock robots, or dynamic robots with the baselines. That is because these search algorithms are incredibly brittle, as we see in Figure 8-4, where we show the performance of the algorithms on the life-cost grid benchmark. These problems are relatively simple, as they can be solved optimally in several seconds (abound 5) by A* search.

We see, however, that speedy and greedy bounded cost searches have incredibly difficult times with these problems. They actually exhaust time on nearly all of the instances; that is they run for ten minutes and still find no solution in the bound. Remember, part of the reason greedy and speedy searches are so fast is that they will accept absolutely any solution. In the bounded cost case, the solutions we can consider are constrained by C. Greedy and speedy search will often encounter a node by a suboptimal path, and to ensure

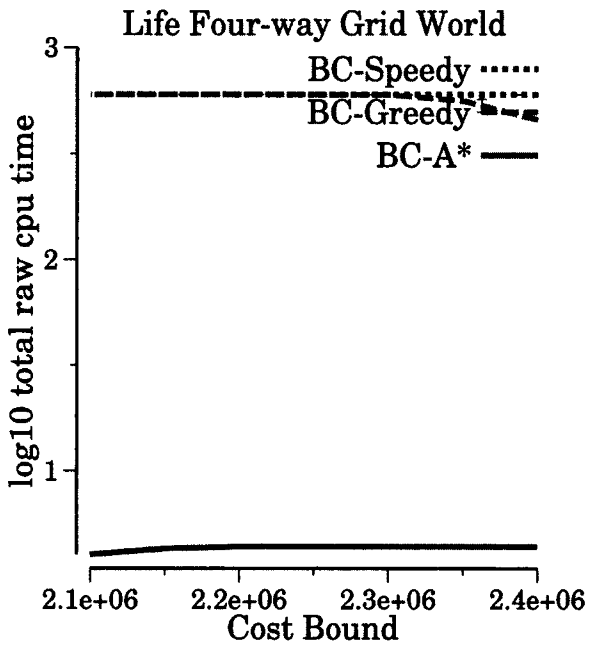


Figure 8-4: Baselines are brittle

completeness they must reconsider that node again every time it is reached by a better path. Considering there are exponentially many paths to a given state, this can take a while. For domains with many duplicate paths, like grids, dynamic robots, and dock robots, greedy and speedy bounded cost searches will not work well unless the heuristic is very powerful.

A* bounded cost search has a similar problem, but for a different reason. Finding the optimal solution to a problem is known to be incredibly difficult. This difficulty is actually the entire justification for the dissertation. We cannot expect bounded cost A* to perform well on problems that cannot be solved optimally in memory by A*, ie every domain used in this dissertation save life-cost grid navigation.

8.3.2 Performance in Terms of Time to Solve

Figure 8-5 shows the relative performance of the bounded cost search algorithms on the benchmark domains considered in this dissertation. On the x-axis, we show the cost-bound that each algorithm was run with. The y-axis shows the mean time consumed by each search algorithm on a log scale. We don't show the performance of the simple baselines, as

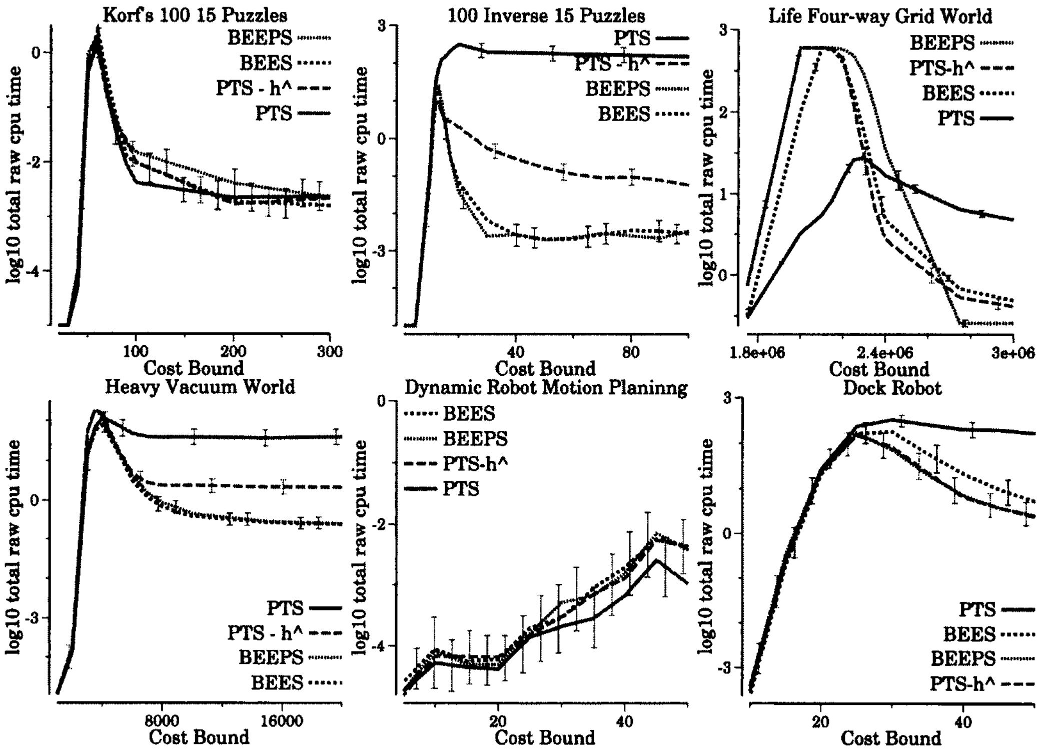


Figure 8-5: Performance of bounded cost search algorithms

they perform quite poorly. Instead, we show four algorithms designed from the ground up for the bounded cost search problem, potential search (PTS in the legend), potential search using an inadmissible cost-to-go estimate ($\widehat{\text{PTSh}}$), bounded cost explicit estimation search (BEES), and bounded cost explicit estimation search with potential (BEEPS).

The first plot in Figure 8-5 shows the relative performance of the algorithms on the standard fifteen puzzle. The algorithms are difficult to differentiate on this plot. In Figure 8-6, we will see that the differences that do exist between the algorithms are primarily a result of algorithm overhead. Although the algorithms are not easy to distinguish between, we do see an important phenomena of bounded-cost search: a often observed easy-hard-easy transition.

Early on, when C is much lower than the average optimal cost of a problem, problems are very easy to prove unsolvable, and search is quite fast. Then, as C grows and approaches the average cost of an optimal solution, it becomes quite difficult to prove that no solution exists, or alternatively, that some optimal or near-optimal cost solution satisfies the cost-bound C. This manifests as a large peak in the performance profiles of the algorithm. Then, as C grows to be much larger than the average cost of an optimal solution, problems once again become easy, although not as easy as those instances which were trivial to prove unsolvable within C.

In the next domain in the plot, inverse cost fifteen puzzles, all algorithms follow this general rule with the notable exception of the potential search algorithm. It starts off solving many problems quite quickly, as no solution exists within C, However, once the other algorithms have peaked and begun their descent, potential search fails to become fast again. As we saw in previous chapters, search focused purely on cost-to-go, especially greedy search on h, does not perform particularly well in the inverse cost tiles problems. Potential search becomes greedy search on cost-to-go for large values of C, and so it never becomes as fast as the algorithms that search on actions-to-go directly (ie BEES and BEEPS), nor the algorithm that includes d indirectly (PTSh$\hat{)}$. Of all the algorithms, BEES and BEEPS are the fastest and are difficult to distinguish between.

The results in life-cost grids highlight the importance of an efficient search order for portions of the space where solutions are difficult or impossible to find within the given cost-bound. We again see the same easy-hard-easy transition as C moves from a point where few solutions exist, towards the cost of optimal solutions to the problem, and then finally beyond the cost of optimal solutions into a space where many solution exist within C. However, for the smaller values of C, we see that potential search is far and away more efficient than the other bounded cost search algorithms. This is because, as we previously noted, potential search has a very good search order for proving no solution exists. BEES has an ideal approach, provided it correctly estimates that no solution exists within the bound. It obviously does not guess correctly for this domain.

The heavy vacuum domain, seen in the bottom row, left most panel of Figure 8-5 returns to the more standard shape of the tiles domains. That is, problems start out easy to solve, then become harder to solve as C approaches the average cost of an optimal solution. As

C becomes larger, most of the algorithms speed up, eventually converging on the speed of a greedy search, either on cost-to-go in the case of potential search and potential search on inadmissible heuristics, or a greedy search on estimated actions-to-go in the case of BEES and BEEPS. Just as was the case in the inverse tiles problems, search on actions-to-go ends up being much faster than search on cost-to-go, and so BEES and BEEPS outperform the other algorithms in this setting.

The next panel in the figure shows the performance of the algorithms on the dynamic robot navigation problem. The results in this domain do not look at all like the results in the other five domains. This is because the domain is quite different from our other domains. We generate the instances with random start and goal locations, so there is a wide range of optimal solution costs to these problems. This prevents us from getting the well formed peaks seen in the other domains, as there is no C for which most problems have an optimal or near optimal solution.

In the final panel of Figure 8-5, we have results on the dock robot domain and a return to the easy-hard-easy pattern seen in most of the domains in our study. We note that the peak is less pronounced in this data set than it is in the tiles puzzles. This is because there is a wider range of optimal solution costs for the problems in this domain than there are for the tiles domain. This results in a different point for the peak in each problem. Further, we can't come close to solving these problems optimally using the search techniques in this dissertation, meaning many of the algorithms time out rather than returning a solution or showing that none exists within C. That results in a more shelf-like appearance rather than a peak in terms of running times and nodes generated.

This data set is also interesting because both BEEPS and potential search on inadmissible heuristics perform better than BEES, in contrast to the other domains where BEES and BEEPS are generally better than potential search with or without inadmissible heuristics. This is because BEES is often estimating that no solution lies within the bound despite there being a solution within the bound. BEES reverts to an A* search order when no solution is estimated to be within the bound, as A* is the most efficient way of proving this

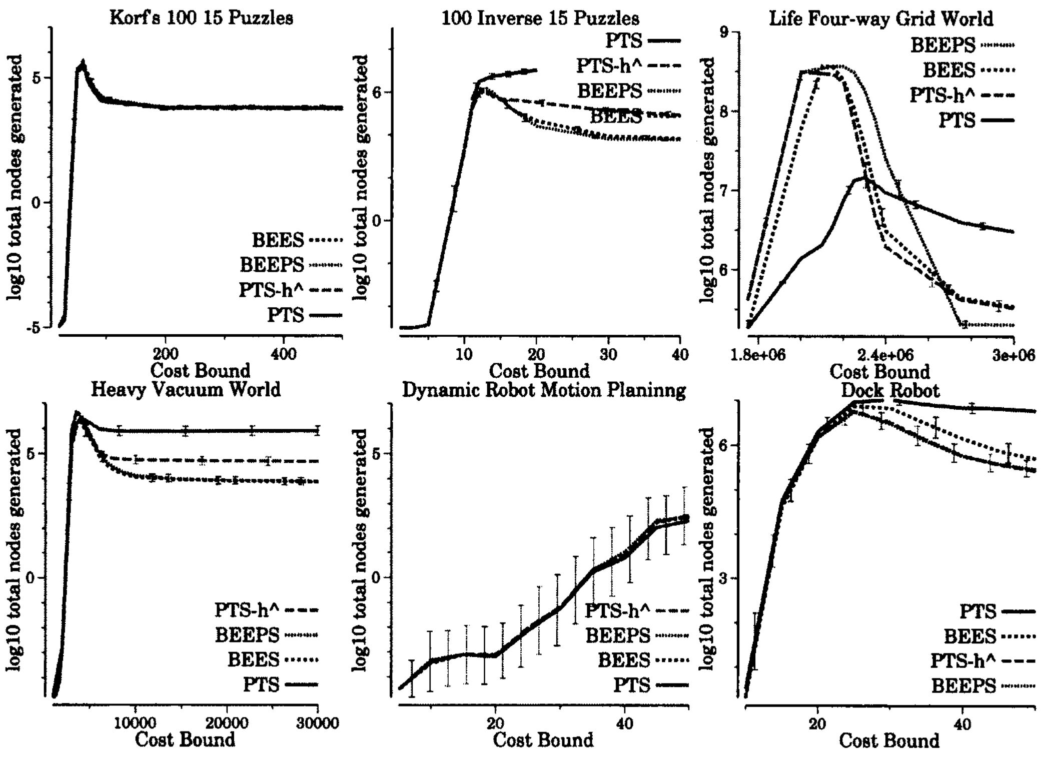


Figure 8-6: Performance of bounded cost search algorithms

to be true. However, BEEPS falls back to potential search on inadmissible heuristics in this case, and this is why the performance of these two algorithms align in this domain.

This points out the differing requirements of heuristic properties that bounded cost algorithms like BEES and BEEPS have when compared to bounded suboptimal algorithms like EES and Skeptical. EES and Skeptical can perform well so long as the inadmissible heuristics provide a good relative ordering over the nodes, while BEES and BEEPS really need accurate estimates of cost-to-go in order to determine if a solution exists within C or not.

8.3.3 Performance in Terms of Nodes Generated

Figure 8-6 shows the performance of the bounded cost algorithms in the same domains as those in Figure 8-5, but now we are examining performance as number of nodes generated rather than time consumed. This removes algorithm overhead from the comparison, and provides a direct comparison of the search strategies of the algorithms in the evaluation.

The results seen in the timing plots are relatively unchanged in the nodes generated evaluation. We do see that the algorithms are essentially indistinguishable in two domains: the sliding tiles puzzle and dynamic robot navigation.

8.4 Discussion

Consider the two best performing algorithms here, BEES and potential search. These two algorithms take radically different approaches to the problem of bounded cost search. Potential search explores nodes in order of their chances of containing a solution within the bound beneath them. BEES makes a binary decision about a node leading to a solution within the cost bound, and then explores this subset of all nodes in order of increasing estimated actions-to-go.

Both approaches initially seem well founded. Pursuing nodes more likely to lead to acceptable solutions seems ideal. Remember though, we don't simply want to find a solution in the cost bound. We want to find a solution within the cost bound *quickly.* The node most likely to contain a solution beneath it may be talking about a solution thousands of steps away, while a node that is marginally less likely to contain a solution beneath it may be discussing a solution tens of steps away. Not taking the shorter, albeit less likely, node in favor of the more certain bet seems unreasonable, and as the empirical results showed, it is unreasonable.

BEES has a similar flaw. It makes a binary decision about within the cost bound or outside the cost bound, and then searches all of those nodes as if they were equal. If we assume error in $\widehat{f}$-values is similar across all nodes, then if we have two nodes n_1 and n_2 where $\widehat{f}(n_1) \leq \widehat{f}(n_2) \leq C$, n_1 is obviously more likely to lead to a solution within the cost bound than n_2. However, BEES doesn't use this information in any way. Anytime a search algorithm doesn't take advantage of information it has computed, we should immediately wonder if there is some way to bring this information to bear efficiently.

Ideally, we would consider both likelihood of containing a solution beneath it and the proximity of that solution together as a kind of expected effort measurement. Simple combinations, like BEEPS, or simply multiplying $\widehat{d}(n)$ by potential do little to improve search performance, in some cases they harm it, and they also incur additional overhead over the approach of BEES. This of course doesn't preclude the usefulness of this information, but we have yet to find an effective way to bring it to bear.

8.5 Summary

In this Chapter we discussed the bounded cost search problem, where we want to find a solution within a user specified cost-bound C as quickly as possible. We examined two main approaches to the problem, potential search and bounded cost explicit estimation search. Potential search sorted nodes in an order determined by how likely they were to have a solution within the cost bound. BEES, on the other hand, first constructed a set of nodes estimated to be within the cost-bound and then sorted these based on estimates of the remaining number of actions-to-go. The empirical evaluation revealed again that if we want to have fast searches, we need to consider estimates of actions remaining to prefer solutions that are easier to find. Simply put, we should be careful to optimize the problem we are solving when applying heuristic search.

CHAPTER 9

ANYTIME SEARCH

When abandoning cost-optimality as infeasible, there are still many ways we can retain some measure of control over the search algorithms being used to find solutions. In the previous two chapters we've discussed ways of managing the cost of solutions returned by a solver, either by restricting solution cost to be within some bounded factor of optimal or to be beneath some absolute user specified bound. We now consider an alternate setting, where time, and not solution cost, is the value we want to retain some control over.

There are three primary ways we can retain control over the time used by heuristic search. In the setting where we are willing to interleave finding a plan and executing that plan, we can limit the amount of time taken by search per-action. Alternatively, we can construct search algorithms that are designed to work under a fixed, known deadline, for example half an hour. Finally, we could construct algorithms that work with an unknown amount of time. This final setting, the anytime search setting [4], is the focus of this chapter.

We will begin by setting out the anytime search setting. In particular, we will take care to differentiate it from the setting in which a deadline is known before hand. The literature has shown that knowing the deadline before search begins should change our search strategy [14], and we will discuss why that is the case here. After introducing the problem of anytime search, we will discuss three general frameworks for converting bounded suboptimal search algorithms like those discussed in Chapter 7 into anytime search algorithms. The strengths and drawbacks of the frameworks will be the focus of this section, and we will show the performance of Explicit Estimation Search in each of the frameworks as well. Finally, we will discuss anytime search algorithms that are not obviously frameworks for converting other algorithms into anytime search algorithms. This section includes discussion of the anytime explicit estimation search algorithm which attempts to minimize the time between improving solutions in anytime search. Although we will not see that AEES is always the best performing algorithm for anytime search, we will see that the algorithm with the least time between improving solutions often is.

9.1 Anytime Search Setting

Anytime search is so named because an anytime search algorithm could be interrupted at any time and be required to return its best known solution. Anytime search is designed to solve problems under some unknown deadline. That is, we do know that there is a limit on the amount of compute time that will be given to us, but we do not know what that limit is. We could be asked to stop our algorithm at any point. In this setting, the desired behavior for an algorithm is to find some solution quickly, and then produce a stream of improved solutions until the cutoff arrives or until we can prove that we have the optimal solution in hand.

This suggests the following approach: find any solution as quickly as possible, then find the next improving solution as quickly as possible, and so on.

9.1.1 Ideal Performance of Anytime Search Algorithms

When discussing the performance of anytime algorithms, we need to make an important distinction between ideal performance and dominance. Dominance is when one anytime algorithm always has a better solution in hand at a given time than another anytime algorithm. More formally, for two anytime algorithms χ_1 and χ_2, let $\chi(t)$ be the solution returned by the algorithm at time t. Then χ_1 is said to dominate chi_2 if, for all t, $g(\chi_1(t)) \leq g(\chi_2(t))$. In this chapter we will assume that the cost of no solution is infinite, ie $g(chi(0)) = \infty$. Realistically, χ_1 dominates χ_2 if, when the algorithms are halted χ_1 has the better solution.

Clearly, dominance is what we want of our anytime search algorithms. We want the best possible solution at time t for any conceivable anytime search algorithm. However, it is incredibly difficult to optimize "better than every other algorithm", especially when many algorithms have yet to be constructed. As a result, we will now examine an alternate measure of anytime search performance, minimizing regret.

In this context, regret will be the amount of "wasted" compute time. That is, when the anytime search is interrupted, how long ago did it find its best solution. The time between finding the returned solution and returning that solution is wasted in the sense that the additional compute resources did not directly improve the quality of the incumbent solution. The time may not actually be wasted, as it may improve the bound on solution quality or show that large portions of the space contain no answers, but from the perspective of the consumer of the returned solution, it is effort that resulted in no improvement, or more colloquially a waste.

Regret can be formalized as follows. Let t_{stop} be the time at which the algorithm was halted, then

$$regret(\chi, t_{stop}) = t_{stop} - (\operatorname*{argmin}_{t \leq t_{stop}} \chi(t) = \chi(t_{stop})) \tag{9.1}$$

This is exactly the difference in time between when the solution was returned and when the solution we returned was found. While minimizing regret does not guarantee dominance, it does minimize the amount of wasted compute cycles, which is a desirable trait.

Minimizing the regret for an algorithm under an unknown deadline is simple. As the deadline could come at any time, we must simply minimize the time between improving solutions which we will refer to as δ_t. Minimizing δ_t has a lot in common with the quantities we were trying to optimize for in Chapters 7 and 8. There we wanted to find some solution within the relative or absolute cost bound as quickly as possible. We were trying to minimize the time between the initial, infinitely expensive incumbent and the time when we found our acceptable solution. Minimizing time between solutions requires looking at the time it will take to find a solution. Just as in bounded suboptimal and bounded cost search, we will see that algorithms which take time to solution into account are rare in anytime search.

9.1.2 Difference from Contract Search

It is important to differentiate between anytime search problems and contract or deadline search problems if only because anytime search algorithms are often applied to deadline search problems even though they are not ideally suited to the problem. The key difference between anytime search and deadline search is that in the former we do not know when the algorithm will be halted, however in the latter this information is part of the problem description. Deadlines are actually quite common in a variety of settings, but perhaps the most common is competitions such as the bi-annual international planning competition. Here, competitors are given about half an hour per instance to solve a variety of vary challenging planning problems, yet most algorithms take an anytime approach to the problem.

The use of anytime search algorithms for deadline search problems conflicts with what has been a central tenant of this thesis: use all available information. In this case, the impending deadline is the available information not being used by the algorithms. Anytime search algorithms should, as we just discussed, seek to minimize the time between solutions in order to reduce potential regret. However, when the deadline is known, we should seek to have the best solution possible in hand at that deadline. These two tasks differ, as we just discussed.

There exist algorithms for the deadline search setting, for example deadline aware search [14], but they are not the winning algorithms for the international planning competition, anytime algorithms are. This speaks to the difficulty of designing good contract search algorithms. We all know that we are trying to minimize solution cost within a given deadline, but taking the deadline into account requires us to estimate how difficult it will be to solve a problem. This is very hard to do. Even deadline aware search, which is currently the state-of-the-art, is actually an anytime algorithm because it frequently fails to accurately predict when it will be able to reach a given solution. Improving these predictors is an important and open problem.

9.2 Three General Frameworks

We now turn from the definition of the problem of anytime search to the algorithms designed to address it. Although anytime algorithms can take any form, they tend to be based on best-first heuristic search algorithms and can loosely be classified into one of three frameworks: the continued search framework, the repairing search framework, and the restarting framework. All of the previously discussed algorithms have been best-first heuristic search algorithms.

9.2.1 Continued Search

Continued search runs a bounded suboptimal search beyond the first encountered solution was introduced by Hansen and Zhou [21]. If the search is continued it will produce a stream of ever improving solutions, eventually finding the optimal solution. Continued search is sensitive to the configuration of the underlying bounded suboptimal search. There will naturally be some sensitivity to the underlying algorithm for all frameworks, but unlike repairing or restarting search, continued search never reconsiders the initial configuration of the underlying algorithm. As a result it is very reliant on pruning for performance. Thus, it performs best in domains with strong admissible heuristics where greedy search produces good solutions, and many nodes can be pruned once an incumbent solution is in hand. It has difficulties in domains where there are many cycles because it cannot ignore improved paths to an already visited state. Although the underlying bounded suboptimal algorithms may be able to ignore duplicate states while still respecting a suboptimality bound [16], ignoring these nodes during a continued anytime search would prevent us from converging on optimal.

9.2.2 Repairing Search

Repairing search differs from continued searches in two ways. First, they have a special way for handling duplicate nodes. When repairing search encounters a better path to a state which it has already expanded, it places this state onto a list of inconsistent nodes rather

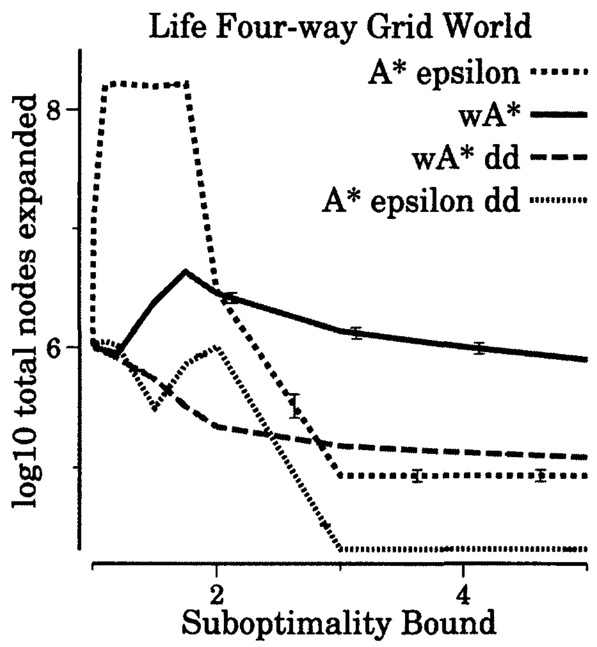


Figure 9-1: Impact of ignoring duplicates

than immediately re-expanding it. These nodes will not be selected for expansion until the next iteration of repairing search. While this may decrease the quality of the solution found on any iteration of repairing search, it leads to improved performance in domains with many cycles by decreasing the time it takes to find a solution on any iteration, as seen in Figure 9-1. Here, we show the performance of A^*_ϵ [42] and weighted A* [43] on a grid pathfinding problem. The y-axis represents the number of nodes generated while finding a solution on a log scale. The x-axis represents the parameter that the algorithm was run with. Algorithms with 'dd' appended do not re-expand duplicate states, instead they ignore duplicate states whenever they are encountered. While this can decrease solution quality, and even quality bounds for some algorithms, ignoring duplicates allows both of these algorithms to solve the problems while generating orders of magnitude fewer nodes. In the event that ignoring duplicate nodes loosens the desired suboptimality bound, as it does in every algorithm but weighted A*, the anytime nature of the framework will ensure that we still converge on an optimal solution, but the speedup will still extend to every iteration of the search.

Second, repairing searches rely on parameter schedules. These are typically constructed by selecting a starting parameter and a decrement for the parameter, although they may also be specified by hand. Every time a new solution is encountered the parameters are updated. There are now two parameters that need tuning: the starting weight and the decrement. Set the decrement to be too large, and the next iteration may never finish; however, if the decrement is too small, the open list will be resorted a large number of times, and this is also inefficient. Changing the parameters used by the search requires updating the evaluation of every node the search is currently considering. While touching every node will take time, it also allows for the immediate pruning of every node that cannot lead to an improved solution. This considerably reduces the size of the open list and thus reduces overhead.

An alternative to the above approach is to compute a new bound dynamically every time a new incumbent solution is found. As we pointed out several times in Chapter 7, the node with the smallest f-value, $best_f$, can be used to construct a lower-bound on the cost of an optimal solution. Using this lower-bound, an upper-bound on solution suboptimality can be computed as $\frac{g(incumbent)}{f(best_f)}$ as we discussed in the sections on optimistic and skeptical search. Likhachev et al [35] point out that while we could compute such a bound dynamically, it is likely to create jumps that are too large in the parameters used by the anytime search. We have not observed this behavior in our searches, but we do evaluate on different benchmarks than they did. This could be the reason for the difference. As we have already established, the behavior of weighted A* differs from domain to domain, and thus what would constitute a large jump in w would also differ from domain to domain.

9.2.3 Restarting Search

Restarting search is one of the simplest frameworks for anytime search. Restarting weighted A* (RwA*) [46], the search strategy at the center of the award winning LAMA planning [48, 49], is an example of an algorithm in the restarting framework. RwA* runs a sequence of weighted A* searches, each with a parameter picked from a hand-crafted parameter schedule. The subsequent searches do not throw away all of the effort of previous searches,

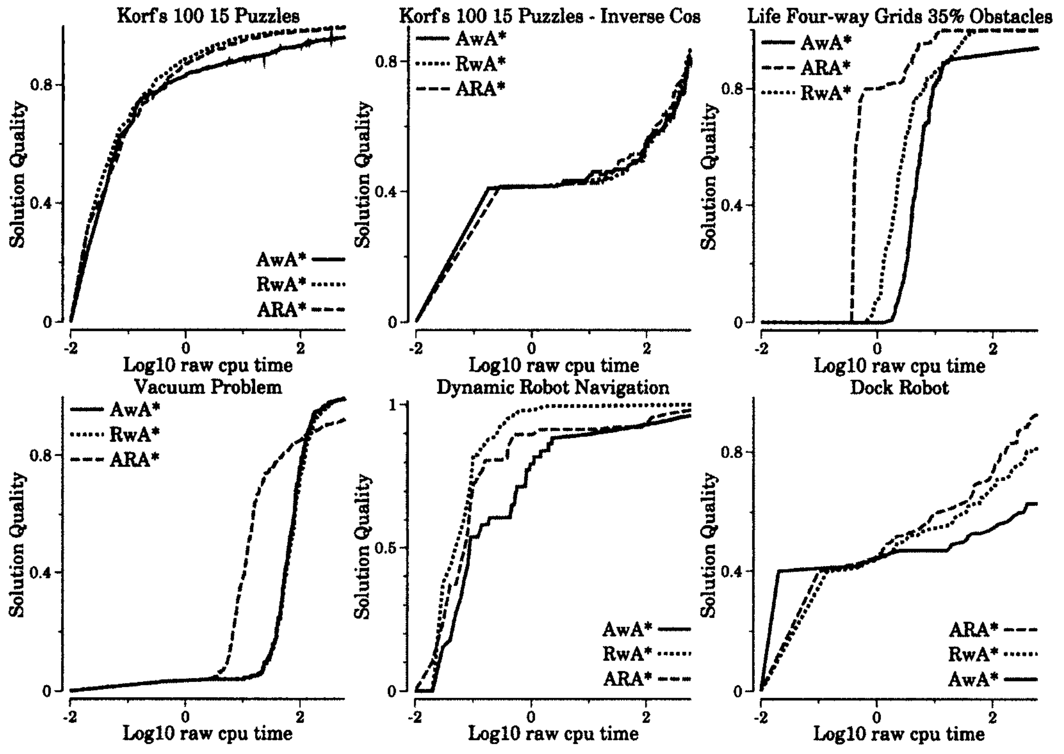


Figure 9-2: Comparison of anytime frameworks

they share information in the form of the incumbent solution, cached heuristic values, and stored paths from the root to states. This way, when a new iteration of search encounters a node previously explored, it must not re-compute the heuristic, an action that may be expensive, and it can replace the current path to the node with a better one found in a previous search iteration.

9.2.4 Comparison of Frameworks

So that we can get a better feel for the relative trade-offs between the various frameworks discussed in this section, we perform an empirical evaluation of weighted A* run in each of them. This is effectively an evaluation of the frameworks as they were proposed on the benchmark domains considered in this thesis. All three frameworks are shown in Figure 9-2 across six benchmark domains. Anytime weighted A*, AwA* in the legend, is shown as the avatar of the continued search framework, anytime repairing A* is used for repairing search and is labeled ARA* in the legend, and restarting weighted A* represents the restarting framework and is labeled RwA* in the legends. The x-axis shows the cutoff time for the algorithm on a log scale, and the y-axis shows the quality of the incumbent solution. We present the mean of the solution quality on the y-axis, and show 95% confidence intervals.

Parameter Settings for Anytime Algorithms

Anytime weighted A*, anytime repairing A*, and restarting weighted A* all require a parameter with which to run. This is actually one of the largest drawbacks of these algorithms. As we saw in Chapter 7, the performance of bounded suboptimal search algorithms can vary between various suboptimality bounds, and the performance does not always improve with looser bounds. This makes setting parameters for anytime search algorithms like those discussed here a challenge.

We use $w = 3$ for anytime weighted A*, and the fixed parameter schedule 5,3,2,1.5,1 for anytime repairing A* and restarting weighted A*. Nothing is sacred about these values. However, they do appear to work well in practice, and they mirror the values used in previous evaluations of these algorithms [47, 67]. For some domains, other settings would have worked better and other settings would have had worse performance. Those reported here work fairly well across the board and provide a realistic view of what one might expect from the various algorithms on the various domains.

Discussion of Results

Perhaps the most interesting thing about the results shown in Figure 9-2 is that no algorithm dominates across all domains. Anytime repairing A* has the best performance, being the dominant algorithm in half of the domains investigated here. In one domain, dynamic robot navigation, restarting weighted A* is the clear choice, and in the inverse tiles puzzle, continued search is clearly preferable. In the original tiles problem, repairing and restarting search have similar performance, so similar that it is difficult to say which approach is best.

The domains where ARA* has the largest advantage over other approaches, life cost grids, dockyard robots and heavy vacuums, roughly in order of the size of ARA*'s advantage, have a common feature: a large number of duplicates. ARA* has a special method for handing duplicate states, so we should expect it to perform better in domains with a large number of duplicate states. Similarly, in domains with very few duplicate states, like the tiles puzzles and dynamic robot navigation, special handling of duplicates only harms the performance of repairing search. It is spending time doing something that is unbeneficial.

In the domains investigated here, it is rare that restarting improves the performance of the algorithms here. This is in contrast to domain independent planning, where restarting can lead to substantial performance improvements by combating a problem called low-h-bias. Low-h-bias is actually exactly the desired outcome of search algorithms like weighted A*. By putting additional emphasis on the cost-to-go estimate, weighted A* prefers nodes with low h-values to other nodes in the search space.

While this is the desired behavior for single-solution settings like bounded suboptimal search, it's not good behavior in anytime search. When finding a solution, we tend to generate several nodes near that solution just because of how state space progression search works. If poor decisions are made early on in the search, algorithms like anytime weighted A* will only reconsider those decisions late in the search order because nodes near the root have very high h-values relative to nodes near the goal. By restarting the search over from the root every iteration, restarting search avoids this problem.

Dynamic robot navigation is the only domain where we see restarting search outperforming other approaches. It is also the only one of our benchmarks where decisions made early on can be argued to be disproportionately important. In these domains, the robot starts from a standstill and must navigate to a given goal location and heading. Note, speed is not considered in the goal state. Since the goal of the problem is to minimize time to solution, the early part of the search is very important. In these early states, the robot gets up to speed, and how quickly it can get up to speed is often determined by decisions made early on in the search. Using high weights causes continued search and repairing search not to reconsider these important early decisions.

To reiterate, unfortunately there does not appear to be a one-size-fits-all model for anytime search frameworks. The appropriate decision lies partly with the domain being searched and partly with the algorithm being used in the framework, as we are about to discuss.

9.3 EES in Frameworks

Much of the previous work in anytime search can be seen as wrapping bounded suboptimal search algorithms in additional functionality, as we just discussed. Previous work focused on weighted A* almost exclusively when extending bounded suboptimal search algorithms. Weighted A* is simple to implement and understand, and until relatively recently there were not consistently better performing algorithms; it was a natural choice at the time those anytime algorithms were published.

Now that we have improved bounded suboptimal search algorithms, it is natural to wonder if we can construct improved anytime search algorithms by using EES instead of weighted A* as the search inside of the previously described anytime search frameworks. Very roughly, the answer to this question is yes, as shown in Figure 9-3.

9.3.1 Benefits of Frameworks

The different frameworks have different things to offer EES. Continued search simply converts EES from a bounded suboptimal search into an anytime search, but this adds substantial utility to the approach as it adapts it for a new setting. However, the restarting and the repairing frameworks add much more to EES.

Previously, EES could not discard duplicates during search without losing its guarantee of bounded suboptimality. When combined in the repairing framework, EES can ignore duplicates on any single iteration, save the final iteration where $w = 1$. This should improve performance on domains with many duplicate states.

Restarting gives EES the potential to reuse learning done on a previous iteration. As we discussed in Chapter 5.2, heuristics learned during search on one problem often transfer

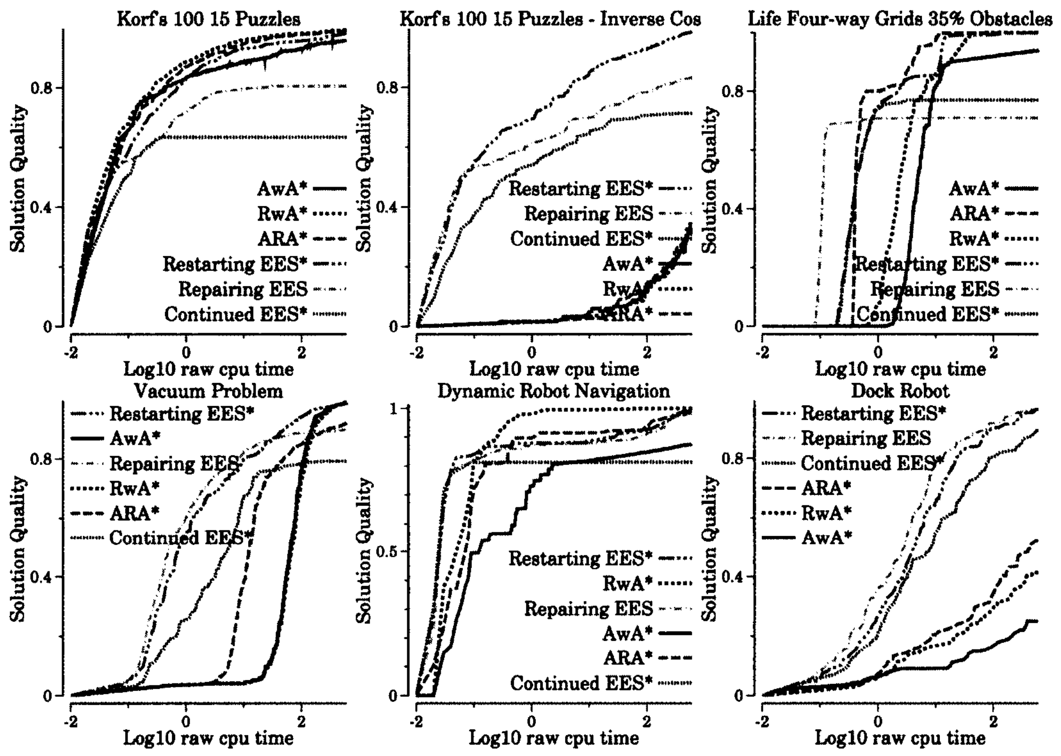


Figure 9-3: EES in anytime search frameworks

well between instances of the same domain. Here though, we are still running search on the *same instance*, so the heuristic should transfer perfectly. If we are using a global error model, we can use the single-step error learned in the previous iteration of search as a base, or just continue to build on what we've already learned as if search hadn't started over again.

9.3.2 Empirical Evaluation

Figure 9-3 shows the relative performance of anytime weighted A* (AwA*), anytime repairing A* (ARA*), and restarting weighted A* (RwA*) compared to explicit estimation search in the same three frameworks, continued EES, repairing EES, and restarting EES. We show the solution quality returned by the algorithms as a function of the cutoff time, shown in log scale on the x-axis.

Again, it is unfortunately the case that no single framework dominates all others. In

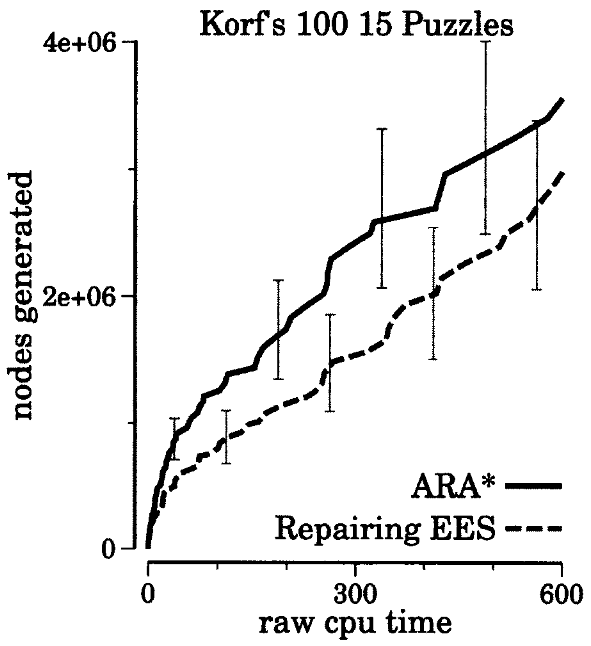


Figure 9-4: Overhead of suboptimal search impacts anytime performance

fact, it is not the case that EES in any particular framework dominates weighted A* in all frameworks. This is in part because no single framework dominates all other frameworks on all domains, but there is another important factor at play: although EES is generally better than previously proposed bounded suboptimal search algorithms, it is not always better as a result of, among many other factors, overhead.

Consider Figure 9-4, where we look at the number of nodes considered by the search, the y-axis, as a function of time, the x-axis, for one of the benchmark domains used in this chapter, the standard fifteen puzzle. We see that over the course of ten minutes, anytime repairing A* is able to examine far many, many more nodes than Repairing EES. In some domains, examining more nodes can lead the less involved techniques to find better solutions by brute force. In others, even though the more deliberative technique considers far fewer nodes, it considers the right ones and thus finds a better solution.

The outcome seems to depend on how informed the inadmissible heuristics are and how low the per-node overhead is for the domain in general. If weighted A* is not able to examine many more nodes and get lucky, its speed will not pay off. In domains like dock robots, EES performs better than other algorithms in the same frameworks because its deliberation pays off and because the other algorithms cannot compensate by simply expanding a staggeringly large number of nodes. In domains like the tiles puzzle, the learned heuristic is less accurate than the base heuristic, and weighted A* can examine hundreds of thousands of nodes a second. This leads to weighted A* in any given framework being better performing in this domain.

As for general trends, while no single framework-based algorithm dominates all other frameworks on all domains, we can see that the algorithms based on EES typically have the best performance. They perform better than other approaches early on, as we see in the inverse tiles puzzle and in the heavy vacuum domain. Further, as time progresses they tend to have the best, or at least competitive, solutions in hand as we see in the same two domains.

9.4 Alternate Approaches

There are anytime searches that are not frameworks for extending bounded suboptimal algorithms into anytime searches. These include beam stack search, BULB, anytime window A*, and branch and bound. Branch and bound performs poorly for all of the benchmarks problems presented here excluding the TSP. The traveling salesman problem is the only domain we examined with a fixed depth. As a result of this fixed depth, depth first approaches like branch and bound can find an incumbent solution quickly, and begin pruning starting the process of converging on an optimal solution. When the safety net of a fixed depth is removed, finding any solution with a depth first search is extremely challenging, and converging on an optimal solution may happen, but it will take a remarkably long time. For example for the 4-connected grid pathfinding problems we considered, A* will solve the problem in less than 2 seconds for all instances we considered, while branch and bound fails to find any solution within the first five minutes. This isn't simply a problem with one domain, it happens in every domain in our evaluation save the TSP. As a result, we omit discussion of it, instead focusing on the more general algorithms which can solve problems of bounded and unbounded depth.

9.4.1 Beam Searches

Beam search is a memory limited search where a set number of nodes at each depth are expanded. The beam is typically some form of 'leaky' priority queue, where the best elements that fit within the size limit are held. When a new element is added to the beam, if the beam is at capacity, the worst element is discarded. Since nodes are discarded before a solution is found, the search is incomplete, but it can be extended into a complete anytime search in several ways.

Beam stack search [80], keeps track of the elements that are discarded from each beam at each depth. Whenever a node is discarded, we make a note of it. When we have exhausted all of the nodes at a certain depth, backtracking begins. When backtracking to a layer, we see if any nodes were discarded. If no nodes were discarded from the beam, we continue backtracking. If some nodes were discarded, we regenerate the beam by re-expanding all of the nodes in the previous beam. This time, rather than only holding on to the best nodes, we hold on to the best nodes that are at least as bad as the best previously discarded node. When repopulating the beam, we still keep track of the best node that is discarded. Eventually, we will exhaust all beams right up to the root layer, at which point we know that the search has returned an optimal solution.

BULB [18] is a blending of limited discrepancy search [33] and beam search that aims to correct the incompleteness of beam search. Limited discrepancy search is a tree based search where we search from the start of the search space towards the leaves but limit the number of times we can choose a node not recommended by the heuristic. Initially, limited discrepancy search will proceed greedily towards a goal, but as the allowed number of discrepancies increases, more of the space is explored until eventually the entire space is considered. It can also be extended to graph search. Rather than maintaining f_{A*}-boundaries as beam stack search, BULB increases the number of discrepancies allowed during an iteration, and eventually it will exhaust the search space. Wilt et al [76] that

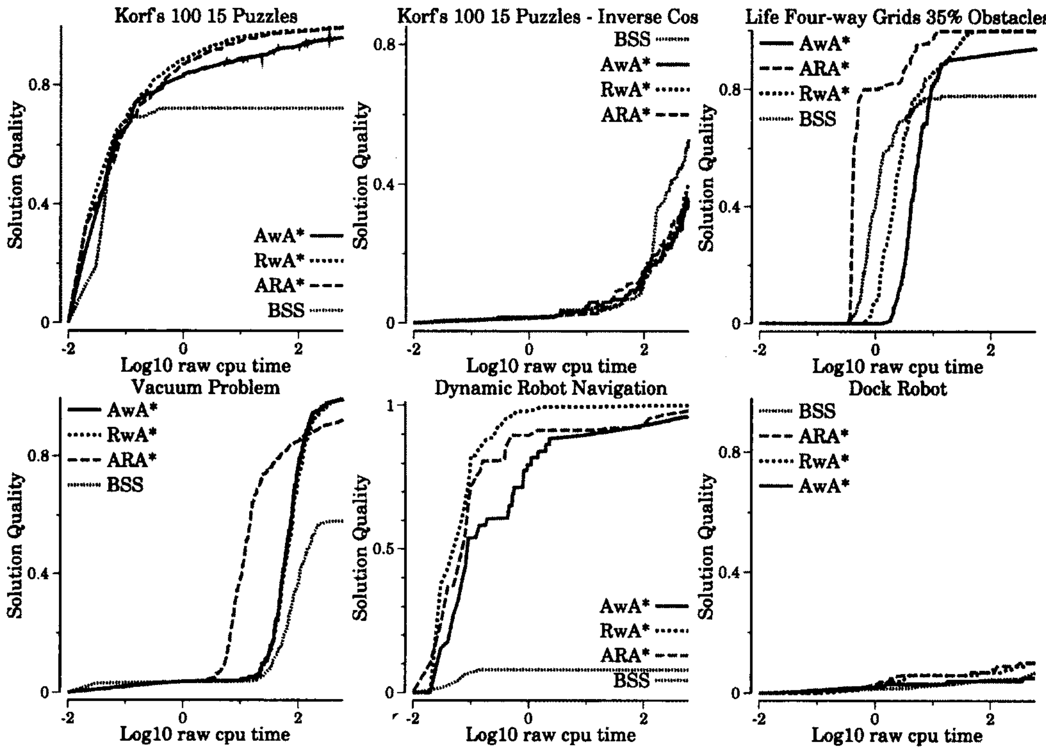


Figure 9-5: Comparison of beam stack search with framework algorithms

beam stack search is consistently better than BULB, so we restrict ourselves to beam stack search in the following evaluation.

Figure 9-5 show a comparison of beam stack search (BSS in the legend) with anytime weighted A*, anytime repairing A*, and restarting weighted A* across six benchmark domains. As before, the x-axis shows the time consumed by the algorithm (on a log scale) and the y-axis shows the mean solution quality as computed in the IPC.

Generally, beam stack search has worse anytime performance than the weighted A* based framework algorithms. There are a few very interesting exceptions. These are the inverse fifteen puzzle and the dock robot puzzle, where beam stack search is better than the framework algorithms, and dynamic robot navigation, where beam stack search is much worse than the framework algorithms. In all three cases, the performance differences are a result of the pruning performed by beam search.

In dynamic robots, there is a disconnect between the heuristic and the goal predicate

```
insert(c, Open, Closed, Suspend)
1.  if $c \notin (Open \cup Suspend \cup Closed)$
2.  then $Open \leftarrow \{c\} \cup Open$
3.  else if $c \in Open \cup Suspend$ & $f(c) <$ previous estimated path cost
4.  then update $c$ in Open & Suspend
5.  else if $c \in Closed$ & $f(c) <$ previous estimated path cost
6.  then replace $c$ in Closed
7.      $Open \leftarrow \{c\} \cup Open$
```

Figure 9-6: Node insertion strategy for window A* and d–Fenestration

that causes many states with $h(n) = 0$ when they are not actually goals. Since we do not break ties on being a goal, beam search will often run circles around the goal state, causing it to perform poorly in this domain. In contrast, the heuristic is apparently often good at identifying a hand full of promising states in inverse tiles and dock robots, and beam search excels in these two domains as a result.

In the inverse tiles puzzle and in dock robots we note that none of the algorithms report high mean quality scores despite quality being relative to the best solution returned by any of the algorithms. What's happening here is that beam search is occasionally returning great solutions to these problems within the time limit, but more frequently it is returning no solution to the problems within the cutoff. The good solutions it does find brings the average for all algorithms down, however not solving many of the problems brings the average quality of beam search down as well.

9.4.2 Window A*

Anytime window A* [1] is an extension of window A* where window A* is run with iteratively increasing window sizes. Window A* is an incomplete search where A* is run on a

```
window A*(Open, Closed, Suspend, BestSol, Depth, WindowSize)
1.  $CurDepth \leftarrow -1$
2.  while $Open \neq \emptyset$
3.      select $n \in Openlist$ with minimum $f$-value
4.      $Closed \leftarrow \{n\} \cup Closed$
5.      if $f(n) \leq g(BestSol)$ then return $BestSol$
7.      else if $Depth(n)\ CurDepth - WindowSize$
8.      then $Closed \leftarrow Closed / \{n\}$
9.          $Suspend \leftarrow \{n\} \cup Suspend$
10.         continue
11      if $Depth(n) > CurDepth$ then $CurDepth \leftarrow Depth(n)$
12.     if $isGoal(n)$
13.     then $BestSol \leftarrow n$
14.         return $BestSol$
15.     else for each successor $c$ of $n$ do
16.        insert(c, Open, Closed, Suspend)
17. return $BestSol$
```

Figure 9-7: Window A*

```
Anytime Window A*($root$)
1.  $Closed \leftarrow \emptyset$
2.  $WindowSize \leftarrow 0$
3.  $Openlist \leftarrow \{root\} \cup Openlist$
4.  $BestSol \leftarrow$ inf
5.  do
6.  $Suspend \leftarrow \emptyset$
7.      $WindowSize \leftarrow WindowSize + 1$
8.      $BestSol \leftarrow WindowA^*(Open, Closed, Suspend, BestSol, Level, WindowSize)$
9.      $Closed \leftarrow Closed \cup Open$
10.     $Open \leftarrow Suspend$
11. while $Suspend \neq \emptyset$
12. return $BestSol$
```

Figure 9-8: Anytime window A*

sliding window of nodes in the search space, instead of on an open list consisting of every node ever generated but not yet expanded. Restricting the comparisons between nodes to nodes a similar distance away from the root makes the comparisons fairer while searching on a restricted set of nodes typically improves the speed with which we can find solutions. Pseudo code for the algorithm is provided in Figures 9-6 9-7 and 9-8. We will now describe each piece in turn.

To understand the behavior of anytime window A*, we need to start by discussing the behavior of window A*. As we previously discussed, the intuition behind window A* is that nodes at different places in the search tree aren't really comparable because the heuristic isn't equally well informed throughout the search. In order to ensure a fairer comparison, window A* restricts its search to a set of nodes at a similar depth. Pseudo code is provided in Figure'9-7.

We can see that the algorithm behave much like A* in line 5. It expands nodes in best-first order as determined by the standard f node evaluation function. However, window A* will only consider a node for consideration if it is within the current window. In line 7, we test to see if the depth of the node is within distance of the deepest node ever expanded. If it is too shallow, the comparison will be too unfair in favor of the shallow node, and we delay the node for expansion until a later time (lines 8 and 9).

If a node is within the window, and it is deeper than any node ever expanded by the search, we will increase the current depth or level of the search (line 11). In this way, the deepest level will always progress forward, forcing window A* to abandon nodes near the root of the search and instead consider nodes further away from the root. Nodes that are placed into the suspend list aren't considered in this iteration of window A*, but might be considered by subsequent calls to window A* if it is used in an anytime framework, as we will now discuss.

Figure 9-8 shows the general layout of the anytime window A* algorithm. Generally, what the algorithm does in call the window A* algorithm shown in Figure 9-7 with progressively larger window sizes. Thus, fewer and fewer nodes will be suspended at each iteration because the window size will be larger, eventually encompassing the entire search tree that A* would have expanded when solving the problem, guaranteeing the optimality.

Rather than starting the search over from the root at each iteration like a restarting search might do, anytime window A* seeds the open list with those nodes that were suspended during the previous iteration. This allows anytime window A* to save effort from previous iterations for use in subsequent searches. It also means that some nodes will be immediately pruned from the next iteration if, for example, the node with the best f-value is very deep in the search space. Search ends when, after an iteration, the suspend list is empty. This signals that all nodes with f-values less than that of the current solution have been explored, and thus the optimal solution is in hand.

d-Fenestration

When we say that window A* assumes nodes at a similar depth are similarly informed, what we mean is that it assume their heuristics are similarly accurate. Large heuristics belong to nodes that are very far away from the goal, and therefore seem more likely to be inaccurate than nodes with small heuristic values. It has been previously noted that the depth of a node does not directly translate into the distance of that node from a goal, even in best first search [66]. We use an estimate of distance to goal, d, to form the window of window A* rather than the node depth, a technique we call d-Fenestration[1].

Using d instead of depth requires a minor change to the algorithm, shown in Figure 9-9. Unlike depth, which grows over the course of a search, d should decrease as new nodes are generated. This may not always be true since d is a heuristic estimate of the distance to a goal for most of the domains in our evaluation. We are interested in the smallest d that the search has ever seen rather than the largest depth. This changes how we determine if a

[1]It's a play on words. Defenestration means to throw someone or something out of a window. The word originates from the Latin "de" meaning down or away from and "fenestra", a window or opening. In this case, we are basing the windowing scheme of window A* on the actions-to-go estimate, d. Hence the name. Seriously, it's very clever and I'd hate for you to miss out on the joke if you bothered to read this far.

```
d–Fenestration(Open, Closed, Suspend, BestSol, WindowSize)
1.   $min_d \leftarrow \infty$
2.   while $Open \neq \emptyset$
3.      select $n \in Openlist$ with minimum $f$-value
4.      $Closed \leftarrow \{n\} \cup Closed$
5.      if $f(n) \leq g(BestSol)$ then return BestSol
7.      else if $d(n) > min_d + WindowSize$
8.      then $Closed \leftarrow Closed / \{n\}$
9.           $Suspend \leftarrow \{n\} \cup Suspend$
10.          continue
11      if $d(n) > min_d$ then $min_d \leftarrow d(n)$
12.     if isGoal(n)
13.     then $BestSol \leftarrow n$
14.          return BestSol
15.     else for each successor $c$ of $n$ do
16.        insert(c, Open, Closed, Suspend)
17.  return BestSol
```

Figure 9-9: *d*–Fenestration

node is within the current window. Nodes are within the window if they have d values that are up to the window size larger than the smallest d we've ever seen, as opposed to up to the window size shallower than the deepest node we have ever seen.

The largest changes to the algorithm are in the direction of the comparisons used to determine if a node is within the current window. Where Window A* looked to see if a node had depth that was not too shallow, d–Fenestration looks to make sure nodes have a similar number of estimated actions-to-go, ie $d(n)$ is not too large relative to min_d and the window size.

It's interesting to note that on the domains where window A* was proposed, the two formulations are equivalent. Window A* was originally proposed for domains where the depth of the solution was known before search began, specifically the 0-1 knapsack problem and the traveling salesman problem. In these domains, there are a fixed number of decisions to be made, and therefor all solutions exist at the same depth. In these settings, nodes at the same depth also have the same d-values, and $d(n) = d^*(n)$ as the depth of solutions is known.[2]

Scaling Windows

Selecting an appropriate window size for the iterations of anytime window A* is key in obtaining reasonable performance. For some domains, such as the knapsack problem, window A* is guaranteed to find a solution for any window size. All nodes have solutions beneath them, so it is impossible for the window to only contain nodes with no solution beneath them. There are also no cycles in the standard encoding, so it is impossible for the algorithm to see nodes it has already generated via a better path, meaning the window can never be exhausted. When these properties do not hold there are many window sizes that find no solution. Typically these are smaller windows, so the question of how to grow the window

[2]This mirrors the relationship of dynamically weighted A* and revised dynamically weighted A*, as we discussed in a previous chapter. Dynamically weighted A* was proposed on a fixed-depth problem, and so the issue of nodes having a differing number of actions-to-go went unaddressed until recently.

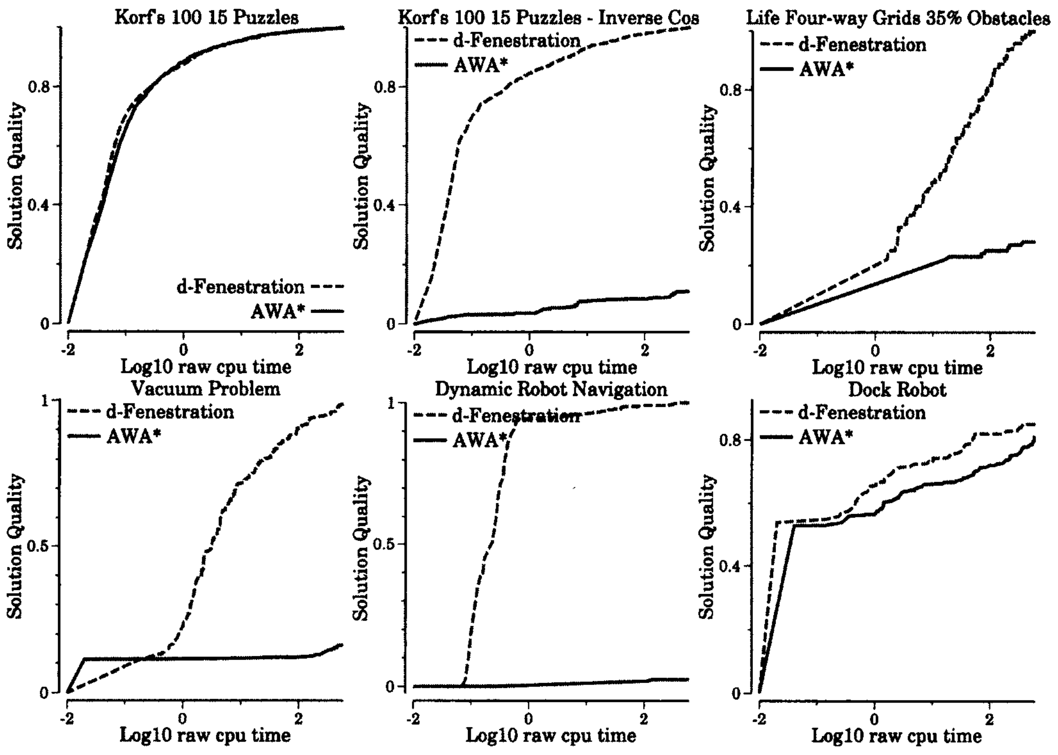


Figure 9-10: Use distance instead of depth in window A*

to the appropriate size naturally arises.

To solve this problem, we grow the window rapidly so long as no solution is found, and become more cautious in growing the window as solutions begin to stream in. We maintain two values, a window step size and a current window size, both initialized to 1. At every iteration, we add the window step size to the current window size to produce a new window. In every iteration where no solution is found, the window step size increases by one, but if we do find a solution, the step size is set back to one. So long as no solution is found, the size of the window continues to grow rapidly until the first solution is encountered. Then, we back off and increase the window size slowly until the solution stream dries up. We also considered using a geometric progression for window step size, but found this was too aggressive in pilot experiments.

Figure 9-11 evaluates the effectiveness of scaling the window size in window based searches using the technique we just described. We perform an evaluation in both the base

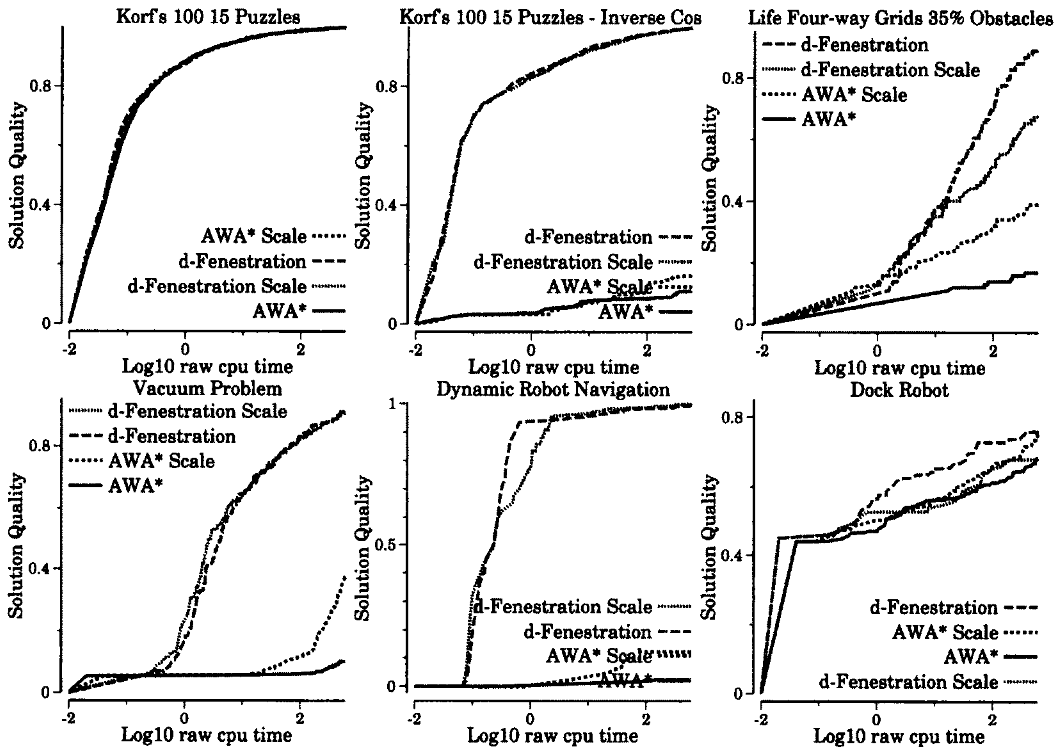


Figure 9-11: Impact of scaling window sizes: anytime window A* and *d*–Fenestration

window A* algorithm as well as the new *d*–Fenestration variant. There is an interesting general trend to be seen here. While window scaling consistently improves the performance of the base window A* search algorithm, it also consistently harms the anytime profile of *d*–Fenestration.

We suspect that the following is happening: Recall that window A* bases the sameness of nodes based on their distance from the root, and so in many cases it is making an incorrect assumption about the proximity of nodes at the same depth being about the same distance away from the goal, as we covered in the discussion of revised dynamically weighted A* in Chapter 7. By increasing the size of the window quickly, window A* with scaling gets to a point where it can make fair comparisons faster. The scaling is likely counteracting the negative effects of the bad assumption. Compare this to *d*–Fenestration, which does not assume that nodes at the same depth are approximately the same distance away from the goal. Thus, increasing the window size too quickly can only cause *d*–Fenestration to do too much work in a given iteration, thus harming its anytime profile.

9.5 A Direct Approach

While we can improve upon the performance of anytime algorithms simply by replacing the bounded suboptimal search algorithm at their core with EES, this is against the philosophy behind EES. When solving a search problem, we should look at exactly what it is we're trying to optimize, and then construct a search algorithm that optimizes what we want to do directly. As we saw in bounded suboptimal search, and as we will soon see in bounded cost search, this tends to lead to a large improvement over algorithms that are not designed specifically to solve the problem at hand.

The first question then is what is the goal of anytime search. Previously we argued that an algorithm that could be interrupted at any time should minimize the time between improvements to the incumbent solution. This reduces the amount of regret, or wasted computation, that the algorithm experiences for any particular cutoff.

The pseudo code in Figure 9-12 presents an algorithm, Anytime EES [63], that is designed to minimize the time between improving incumbent solutions. In line 3 of *AEES* in Figure 9-12 we see AEES and EES have the same definition of best, and thus expand nodes in the same order. *selectNode* pursues the nearest solution estimated to be within the suboptimality bound, provided we can currently prove this node is actually within the bound (line 1 of *selectNode*). Selecting $best_{\widehat{d}}$ is pursuing the next fastest-to-find solution. $best_{\widehat{d}}$ is estimated to both be within bound and have the fewest actions (and thus node expansions) between it and a goal. All other nodes are selected in an effort to make $best_{\widehat{d}}$ a node that could be pursued, either by raising our lower-bound on optimal solution cost or by adding new nodes to the pool from which $best_{\widehat{d}}$ is selected.

EES and AEES differ in what happens when a goal node is encountered (line 5 of *AEES*). EES would simply return the solution. AEES is an anytime search algorithm that must eventually converge on an optimal solution. When AEES finds a goal, it updates the cost of the incumbent solution and lowers the suboptimality bound w before continuing

```
AEES(root)
1.  $open \leftarrow \{root\}$, $cost \leftarrow \infty$, $w \leftarrow \infty$
2.  while $open \neq \{\}$
3.    let $n = selectNode(open, w)$ in
4.      if $f(n) \geq cost$ then continue
5.      else if $goal_p(n)$ then $newIncumbent(n, w, cost, open)$
6.      else $expand(n, open, cost)$
7.        $open \leftarrow open - \{n\}$
8.        for each child $c$ of $n$
9.        if $f(c) < cost$ then $open \leftarrow open \cup \{c\}$
newIncumbent(n, w, cost, open)
1.  if $g(n) < cost$
2.  then let $best_f = \text{argmin}_{n \in open} f(n)$ in
3.    $cost \leftarrow g(n)$
4.    $w \leftarrow \frac{cost}{f(best_f)}$
selectNode(open, w)
1.  if $\widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f)$ then $best_{\widehat{d}}$
2.  else if $\widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ then $best_{\widehat{f}}$
3.  else $best_f$
```

Figure 9-12: Anytime explicit estimation search

search.

Rather than supplying a schedule of suboptimality bounds, we compute one online. During search, we can compute a dynamic bound on the suboptimality of the incumbent solution. Rather than supplying a sequence of suboptimality bounds, we need only compute the dynamic bound when the algorithm needs the next parameter, typically when a new solution is encountered.

In AEES, a dynamic bound can be computed as $\frac{g(incumbent)}{f(best_f)}$. $f(best_f)$ provides a lower bound on the cost of an optimal solution to our problem, and so this equation computes an upper bound on the suboptimality of the current incumbent solution. We use this dynamic bound to set w for the next iteration of AEES. This technique has also been used to augment parameter schedules used by anytime search [35, 21, 67].

While we can construct examples where an algorithm that improves the incumbent solution fastest does not have the best solution in hand for many cutoffs, the empirical evaluation performed in Thayer, Benton, and Helmert [63] and reproduced and extended in part below shows that in practice this rarely happens. In fact, in this evaluation it was with a single exception the case that the algorithm that had the smallest time between improving solutions also tended to have the best solution in hand at any given item.

Figure 9-13 shows the performance of AEES relative to three other state-of-the-art anytime search algorithms: d–Fenestration, Anytime Nonparametric A*, and beam stack search. Anytime Nonparametric A* (ANA*) [74] is a continued search that can be seen as an anytime variant of potential search [61], discussed in the previous chapter. Anytime nonparametric A* expands the node with maximal $e(n) = \frac{G-g(n)}{h(n)}$. This is equivalent to expanding the node with minimal $e'(n) = \frac{h(n)}{G-g(n)}$, where G is the cost of the current incumbent solution, initially ∞.

As before, we note that there is no clearly dominating algorithm for all domains and all potential time cutoffs. We do however see several general trends. It is rare that AEES is the worst performing algorithm for any cutoff, with the one exception being the first second of search on the dynamic robot domain. Similarly, it is rare that beam stack search is not

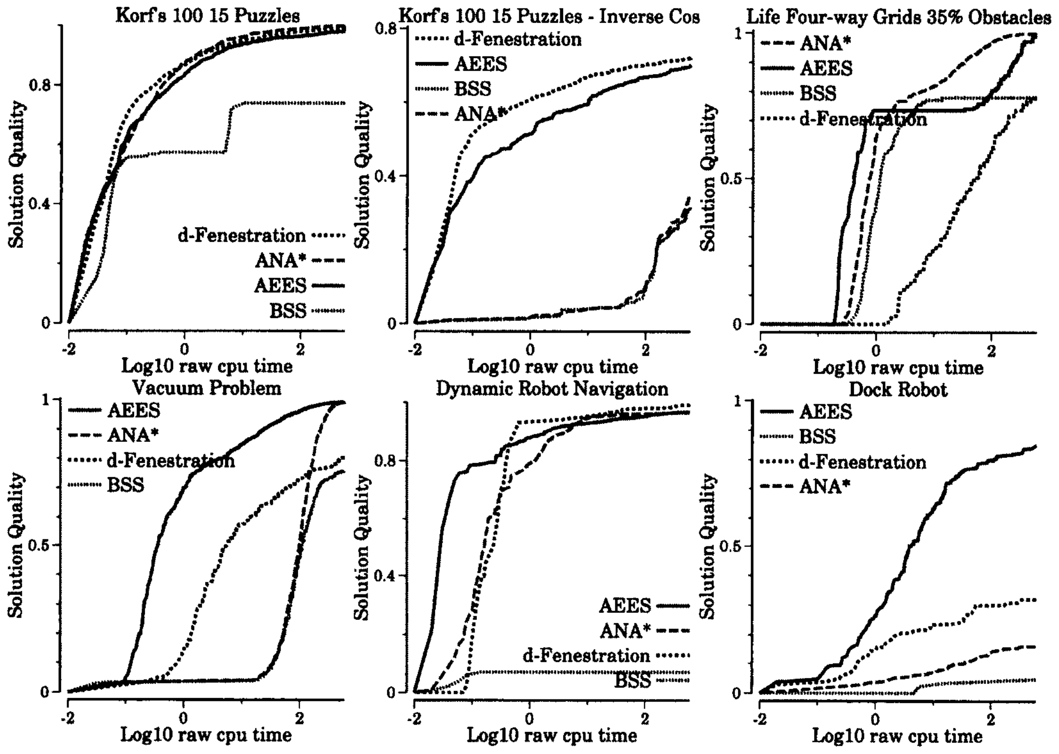


Figure 9-13: AEES versus leading anytime search approaches

the worst performing algorithm, with the exception being in life-cost grid navigation where beam stack search is better than *d*–Fenestration.

In the standard 15 puzzle instances (leftmost panel, top row of Figure 9-13, we see that most of the algorithms have performance that is quite similar to one another. *d*–Fenestration is marginally better than the other algorithms throughout, but not by a substantial amount.

However, when we change the cost-function of the domain, as we do in the inverse tiles problem (center panel of the top row), we see that the performance of the algorithms dramatically changes. Now, AEES and *d*–Fenestration are the only two competitive algorithms, with *d*–Fenestration being the better of the two algorithms. These two algorithms have similar performance because they are the only two approaches under evaluation which take advantage of actions-to-go estimates to guide search. AEES relies on $\widehat{d}$ to select nodes that appear to be close to a goal, while *d*–Fenestration uses d to restrict the comparison of nodes between those nodes that are likely to be similarly informed, ie those nodes that are a similar number of actions away from the goal. AEES sidesteps the issue of similarly informed comparisons by relying on inadmissible heuristics that are not inherently biased based on the distance of a node from the goal.

The rightmost panel of the top row of Figure 9-13 shows the relative performance of the algorithms on the life-cost grid navigation problem. As we've previously discussed, the grid problems are unique among all domains studied in this dissertation in that they have the largest number of duplicate nodes of any domain considered. Thus, algorithms that specially handle duplicate states or that are more likely to reach a node by an optimal path tend to perform better in these domains.

The large number of duplicate states makes it unsurprising that both beam stack search and *d*–Fenestration perform poorly for this domain. Both beam search algorithms and window search algorithms are known to have difficulty in domains with a large number of tight cycles. The relative performance of ANA* and AEES is more difficult to explain. We refer back to Figure 8-5, where we saw that potential search was far more efficient on grid problems than BEES was, in part because it had an expansion order that was more similar to that of A* and thus it re-opened fewer nodes. AEES often expands nodes in order of their proximity to a goal, and this has nothing to do with the cost of that node. Thus it is more likely to expand a node by a suboptimal path, thus requiring a re-expansion, than nonparamteric A*.

In the left and rightmost panels of the bottom row, we see two domains where AEES is far and away better than other state of the art anytime search algorithms. The heavy vacuum domain and the dock yard robot domain have two interesting commonalities: search on an actions-to-go heuristic is often substantially faster than search on a cost-to-go heuristic, and both domains have inconsistent admissible estimates of cost to go. AEES is the only algorithm out of those considered in this evaluation that uses both inadmissible estimates of cost-to-go and actions-to-go. The reason that the inadmissible cost-to-go estimates are interesting in this context is that they are not guaranteed to be consistent, and thus EES, and algorithms based on it, have inadvertently been designed to handle inconsistent heuristics.

Domain / Alg.	AEES	ANA*	d–Fenestration	Beam Stack Search
Tiles	**47**	71	49	*131*
Inv. Tiles	104	*277*	**59**	179
Life	**12**	51	*136*	17
Vacuum	**24**	95	72	*198*
Dyn. Robot	**2**	**2**	110	*200*
Dock	**126**	*379*	357	280

Table 9-1: Average time between solutions in seconds

This is in contrast to many other algorithms that are designed and tested on benchmarks with admissible estimates of cost-to-go.

That leaves the dynamic robot domain for discussion, shown in the center panel of the bottom row of Figure 8-5. This is another domain where anytime explicit estimation search has good, but not dominating, performance. Both anytime nonparametric A* and d–Fenestration have strong performance in this domain as well. We will see in the following evaluation that anytime nonparametric A* and AEES find solutions to this problem with about the same frequency, and this may in part explain their similar performance.

Table 9-1 reports the mean time between solutions for the algorithms shown in Figure 9-13 for the domains used throughout this dissertation. The algorithm with the smallest time between solutions in a given domain has its value **bolded**, while the algorithm with the longest time between improvements in a domain has its value *italicized*. A brief glance at the table will reveal that the algorithm with the smallest time between solutions is often the best performing algorithm. We now discuss this phenomena in more detail.

In the tiles domain, we see in Table 9-1 that AEES has the smallest time between solutions, followed closely by d–Fenestration and LAMA-11. Looking at the results in Figure 9-13, we see that these algorithms are all closely clumped together, and thus have similarly good solutions in had at any given time.

In the inverse tiles problem, we see that d–Fenestration has the smallest time between solutions and it is indeed the best performing algorithm. This observation clearly repeats itself in the heavy vacuum problem and the dock robot domain, although there AEES is the better performing algorithm instead.

In life-cost grids, we see that AEES and beam stack search, and anytime nonparametric A* have the smallest times between solutions. Further, these are the three best-performing algorithms for this domain; early on, AEES has the best performance, but after around a second of computation, nonparametric A* pulls ahead. Beam stack search has comparable performance to these two approaches throughout.

If we look at the table as a whole, we see that the only domain for which AEES doesn't have the smallest delay between improving solutions is the inverse tiles problem. Looking at Figure 9-13, we also see that this is the only domain where another search algorithm has better performance than AEES throughout the entire duration of cutoffs examined. It appears that there is a very strong correlation between small times between improving solutions and good anytime search performance, as we previously hypothesized there would be.

Unfortunately, it is not perfectly clear why this correlation exists. As we previously noted, finding many improving solutions rapidly is not going to cause improved performance if those improvements are very incremental. This can happen, look at the performance of beam stack search in life cost grids, for example, however it appears to be rare. Heuristics, by large, appear to help search, and finding incumbents allows us to prune away unpromising avenues earlier.

9.6 Discussion

Windowing is meant to make the comparison of nodes fairer by restricting the comparison to nodes at about the same depth. The idea here is that nodes a similar distance away from the goal should be similarly informed. This is very much like the idea behind dynamically weighted A*, and it has the same flaw: nodes at the same depth may be radically different distances away from the goal. *d*–Fenestration approaches this problem in the same way that revised dynamically weighted A* tries to approach it, by defining similarity by estimated number of actions to the goal. As we saw, this led to a remarkable improvement in performance.

Why then do we not treat windowing as a general framework, like continued, repairing, and restarting search? Much like low-h bias is unique to algorithms that weight cost-to-go estimates to produce suboptimal search strategies, not all algorithms are prone to making unfair comparisons between nodes. One of the main strengths of inadmissible heuristics is that we can expect them to behave consistently over the entirety of the search space.

We have not yet discussed two alternate ways of controlling the amount of time consumed by a search: search under a known deadline and search with a limited amount of computation per action. Both are fine techniques for limiting the amount of time available to search, and they both nicely line up with a real application: competitions and robotics. Although both areas are interesting, time is finite, and this work doesn't contain any new algorithms in these settings. Possible enhancements and algorithms are discussed in Chapter 10.

9.7 Summary

In this chapter we discussed one particular setting for heuristic search under a time bound, the anytime search setting. We put forth a possible definition for the optimal behavior of anytime search. This definition had been considered, albeit less formally, by other previous authors. We showed that algorithms which optimized this particular performance metric, minimizing wasted search time, also tended to have the best performance in the more classical sense, that is dominant performance. We offered, and have, no explanation for the relationship between these two values other than the intuition that algorithms which make better use of their time often have better solutions in hand when the deadline does arrive.

This chapter covered one search algorithm that is rarely discussed in the literature or deployed in practice: Window-based anytime searches and particularly d–Fenestration. These algorithms are not often employed because they can be quite brittle. When they work well, they work incredibly well, however when they work poorly, they are particularly bad.

Such an assessment ignores an unfortunate truth that came to light several times throughout this chapter and dissertation. There is rarely a best algorithm in heuristic search in general, but this is particularly true of anytime search. There are domains for which beam stack search is the best approach despite performing terribly in many of the domains under evaluation, and this holds for d–fenestration and AEES as well. The question of "Which algorithm performs best in general?" is difficult to answer, and perhaps it is unimportant if we can easily answer "Which algorithm will perform best *here*?".

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

This thesis was separated into two major sections. In the first section, we discussed sources of information not typically considered by optimal heuristic search algorithms: inadmissible estimates of cost and actions-to-go from a node to a goal. The second section of the thesis considered suboptimal search in a variety of settings: bounded suboptimal, bounded cost, and anytime search.

In the first section, we discussed two major techniques for constructing inadmissible sources information. The first involved looking at the domain and constructing inadmissible heuristics based on observations of an expert: the way in which the admissible heuristic is derived, hand-crafted estimators, etc. While such inadmissible heuristics are useful, constructing sources of information by hand doesn't scale well. Thus, we looked at ways of deriving inadmissible heuristics automatically.

Previous work had considered learning heuristics from data written down before any search begins, or from data available in between the solving of multiple instances when solving a large set of instances. We chose to purse the orthogonal approach of learning during the search itself. In order to ensure the technique had the largest possible applicability, we restricted ourselves to information that was ubiquitously available during best-first heuristic search, namely the behavior of the heuristic across a single expansion. By looking at single expansions, we could measure and correct for error in the base heuristic, thus improving it.

The second section of the thesis focused on algorithms for suboptimal search. We talked about three major settings for suboptimal search: bounded suboptimality, bounded cost, and anytime search. In each setting, we proposed a new state of the art algorithm, explicit estimation search, bounded-cost explicit estimation search, and anytime explicit estimation search. These algorithms have two major things in common: they attempt to optimize the goal of the setting directly, and they take advantage of the additional sources of information discussed in the first portion of the thesis.

The first point is the most important. All of the previous work could be modified to take additional information into account, but as we saw in the evaluation, simply taking the information into account did not improve the performance beyond what we could achieve with EES, BEES, and AEES. This is because those three algorithms attempt to solve the problem at hand as directly as possible give the information readily available. "As directly as possible" because we must acknowledge the fact that we are not minimizing time directly; we are searching for short solutions and this approach *tends* to minimize solving time.

Although addressing the problem directly, or nearly directly, is important, we are only able to do so because we rely on the additional sources of information. Thus, their importance cannot be discounted. Without inadmissible estimates of cost and actions-to-go, neither EES nor BEES would have been able to predict which nodes would lie within the desired bound. Without good estimators of actions-to-go, none of the search algorithms discussed here would be able to reason about the proximity of a goal.

The bigger picture of the thesis is that it provides the start of a theory of suboptimal search. In the thesis, we outline three major settings for suboptimal search. We discuss the goal of each area and discuss what ideal performance would be. We then go on to discuss what sorts of information are needed to achieve ideal performance. Finally we consider what available information approximates those sources, and construct state of the art algorithms using this new information.

10.1 Major Lessons

This dissertation covers a lot of ground with respect to the field of suboptimal search, but there are three main points that I feel bear repeating at the end here. These are that suboptimal search is different than optimal search, we should make use of as much information as possible during search, and before working on an algorithm, we should first consider what it is we're optimizing. We now discuss each of these in a bit more detail.

10.1.1 Suboptimal Search is Different

The goals of suboptimal search and optimal search are different. Optimal search seeks to prove that no solution exists better than the one returned; finding the solution itself is almost a secondary consideration. Suboptimal search, on the other hand, is primarily concerned with finding any solution. Proving that solution has certain properties is almost a secondary consideration, especially when we consider difficulty. Proving a solution is optimal is hard even if our heuristics are nearly perfect [24], while proving bounded suboptimality can be easy in certain restricted circumstances [12].

Suboptimal search has different goals than optimal search, and this means we should really be considering sources of information and search strategies that are different from those used by optimal search algorithms. We saw this again and again throughout the dissertation. Techniques like weighted A*, which simply adapt the ideas of optimal search to a suboptimal setting do not work as well as algorithms that are designed explicitly for the suboptimal setting. Their wide adoption is largely the result of their ease of implementation and the amount of time they have existed unopposed.

10.1.2 Use Available Information

The performance of algorithms like EES, BEES, and AEES shows empirically the importance of taking advantage of the information available to the search algorithm. By considering information that was readily available, estimates of actions-to-go and the observable error in the cost and actions-to-go heuristics give, these algorithms are able to out perform the previous sate of the art in their respective areas of suboptimal search.

At first glance, the idea that we wouldn't use information available during search to improve algorithm performance seems ridiculous. However, there are many reasons why it wasn't immediately obvious that we were overlooking information. Many heuristic search papers focus on unit cost benchmarks, and in a unit cost domain there is no difference between cost-to-go and actions-to-go, and therefore no need for an additional set of heuristics. Treating heuristics as sensors and using expansions as observations is an analogy that is not easily made when we forget about the agent in single agent search and instead focus on considering a sequence of potential solutions. It's not surprising that many failed to make this connect before simply because of how we tend to talk about the problems.

While I am very fond of the search algorithms proposed in the second half of the dissertation, I have no doubt that the online learning technique put forth in the first part of the dissertation is the larger contribution. The algorithms will eventually be surpassed by new variants that are able to more directly minimize time under a constrain, probably by considering time directly rather than a proxy like $\widehat{d}$. However, the idea that expansions provide information on the performance of a heuristic that we can leverage to improve heuristics and search performance is, I think, very important.

Suboptimal search is a very large area, but it is only a fraction of state space search, and all state space algorithms can likely benefit from the insight that we can learn from the performance of heuristics during search. The idea is larger than any of EES, BEES, or AEES in the sense of the affected area. Heuristics search algorithms have heuristics by definition, and many of them expand nodes generating successor states from which error can be observed.

10.1.3 Consider What You're Optimizing

Individually, EES, BEES, and AEES are each a nice ideas that contribute to the furtherance of each of their respective areas of suboptimal search. However, they are all bound by a single underlying idea that is more important than any one of the algorithms: we should be mindful of what we are trying to optimize when designing best first search algorithms. All three try to reduce solving time subject to some constraint, be it a relative cost bound, an absolute cost bound, or the cost of the last solution. A large portion of their improvement over the previous state of the art can be attributed to their addressing solving time as directly as possible.

10.2 Future Work

There are several things that are related to the topics discussed in the dissertation that were not explored in sufficient depth. There is also an area relevant to all of the topics in the dissertation that goes largely undiscussed.

10.2.1 Deadline Search

While Chapter 9 addressed one way of controlling the time allotted to a search algorithm, anytime search, it did not consider the setting in which we know the deadline a priori. As we previously touched on, search should make an effort to take advantage of all the information available to it. In this case, the impending deadline is imparting information that should be taken advantage of by search.

Some algorithms have already started looking at this information, for example deadline aware search [14]. There is, however an open question or two relating to deadline search. We don't really have a strong handle on how to take the impending deadline into account. Deadline aware search uses the deadline to try and prune away avenues of search that cannot lead to a solution within the remaining deadline, but this is not obviously the right approach. In fact, it is the most conservative approach imaginable.

This highlights another open problem, how do we estimate if a goal is reachable within the deadline? Deadline aware search uses measurements like vacillation and search velocity to try and estimate the size of the sub-tree that will be expanded form one node on the path to a goal. The effectiveness of these techniques is still unexamined, but even if they were perfect, they don't seem to be answering the right question.

Current techniques for estimating tree size don't consider the quality of solution we're looking for beneath a give node. Search velocity makes no distinction, and vacillation distinguishes between the optimal cost solution and the solution with the fewest actions remaining. This is a step in the right direction, but it is not all the way there. What we really want is a full spectrum, a function of the form "if I am willing to invest X time (or expansions), then I can achieve a solution of cost Y". Given such a function, we could directly optimize the desired goal of search under a deadline. It is unclear how to construct such a heuristic, but tree size estimation of the kind investigated by Korf, Reid, and Edelkampf [34] will likely inform or inspire the approach.

10.2.2 Real-time Search

We also do not consider a second alternative to controlling time during search, the setting of real-time search. In real time search, our goal is to be as certain as possible at the end of the allotted time that we are committing to the correct next action. This is left intentionally ambiguous, as we may be trying to optimize a wide variety of criteria in real-time search: cost, safety, number of actions, etc. The idea is that we want to be sure that we are committing to the right action; the actual metric by which the solution will be measured is a secondary concern.

Current heuristics don't give us the kind of information that we need to solve the problem. They only tell us about cost or distance to go from a node to goal. They don't often tell us much about our certainty in the estimate. That is the more important value here, as we want to be sure we've made the right decision, not estimate what the final value of our solution will be. This desire is a direct result of the interleaved nature of real-time search.

The idea of collapsing confidence intervals is not new to heuristic search. This is the fundamental idea behind algorithms like decision theoretic A* [54] and Monte Carlo tree search approaches like UCT [30]. To the best of my knowledge, neither has been applied directly to real-time search, though both seem like they could be easily adapted. Existing best first search algorithms, like A_ϵ^* [41] could also probably be adapted to this setting. They are well suited to it in that they already consider a set of similar (ie hard to disambiguate between) nodes for search on a secondary criteria.

10.2.3 Dealing with Very Large Problems

In the course of the dissertation, we never bother to discuss very large problems. That isn't to say the problems we look at are particularly small, but they all can be solved in memory using at least one of the algorithms discussed in the dissertation. The largest drawback of best-first heuristic search is that it does not scale well. For a give algorithm and domain, it is almost always possible to specify an input that cannot be solved without needing far more memory than is available on a modern machine. Since the problems we tend to solve with search are often hard in the formal sense, this is unlikely to change.

How to perform suboptimal search on disk or across multiple machines in parallel is an open problem not addressed by this work, with open challenges. The largest is that best first search is, in a very real sense, embarrassingly sequential. Best has a definition that doesn't lend itself well to parallelism. However, as Burns et al [8] point out, we only need to approximate a best first order in search to have the behavior of a best first search and the advantage of parallelism. Hatem et al [23] showed that we can also make use of disk to deal with particularly large spaces in best first search.

10.3 Conclusions

The thesis of this dissertation was that the performance of suboptimal search algorithms can be improved by taking advantage of information that, while widely available, has been overlooked. This information took two major forms: new heuristics and problem statements. The heuristics were either derived from observations about the search space, the performance of heuristics, or both. The larger contribution though, was noticing that suboptimal search differed substantially from optimal search in terms of the desired outcome. This means that simply adapting optimal search techniques, or search techniques from other suboptimal settings, is unlikely to produce ideal performance. Both feed into the foundation of suboptimal search: a formal definition of what we are trying to do, and an analysis of what information is needed to do it.

BIBLIOGRAPHY

[1] Sandip Aine, P.P. Chakrabarti, and Rajeev Kumal. AWA* - a window constrained anytime heuristic search algorithm. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, 2007.

[2] 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 Automated Plannin and Scheduling*, 2010.

[3] Roberto Bisiani. *Encyclopedia of Artificial Intelligence*, volume 2, chapter Beam Search, pages 1467–1468. John Wiley and Sons, 2 edition, 1992.

[4] Mark S. Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings of the Elevent International Joint Conference on Artificial Intelligence*, pages 979–984, 1989.

[5] A Bramanti-Gregor and HW Davis. The statistical learning of accurate heuristics. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1079–1085, 1993.

[6] Anna Bramanti-Gregor and Henry W. Davis. Learning admissible heuristics while solving problems. In *Proceedings of the Twelth International Joint Conference on Artificial Intelligence*, pages 184–189, 1991.

[7] Vadim Bulitko, Yngvi Björnsson, Nathan R. Sturtevat, and Ramon Lawrence. Real-time heuristic search for pathfinding in video games. In *Applied Research in Artificial Intelligence for Computer Games*. Springer, 2011.

[8] Ethan Burns, Seth Lemons, Rong Zhou, and Wheeler Ruml. Best-first heuristic search for multi-core machines. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 2009.

[9] J Christensen and RE Korf. A unified theory of heuristic evaluation functions and its application to learning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 148–152, 1986.

[10] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, second edition, 2001.

[11] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[12] Stephen V. Chenoweth Henry W. Davis. High-performance A* search using rapidly growing heuristics. In *Proceedings of the Twelth International Joint Conference on Artificial Intelligence*, 1991.

[13] Rina Dechter and Judea Pearl. The optimality of A*. In Laveen Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence*, pages 166–199. Springer-Verlag, 1988.

[14] 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.

[15] 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.

[16] Ruediger Ebendt and Rolf Drechsler. Weighted A* search - unifying view and application. *Artificial Intelligence*, 173:1310–1342, 2009.

[17] Michael Fink. Online learning of search heuristics. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, 2007.

[18] David Furcy and Sven Koenig. Limited discrepancy beam search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 125–131, 2005.

[19] M. Ghallab and D.G. Allard. A$_\epsilon$: An efficient near admissible heuristic search algorithm. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 1983.

[20] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice.* Morgan Kaufmann Publishers, 2004.

[21] Eric A. Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297, 2007.

[22] 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.

[23] Matthew Hatem, Ethan Burns, and Wheeler Ruml. Heuristic search for large problems with real costs. In *Proceedings of the Twenty-fifth AAAI Conference on Artificial Intelligence*, 2011.

[24] 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.

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

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

[27] Shahab Jabarri Arfaee, Sandra Zilles, and R.C. Holte. Bootstrap learning of heuristic functions. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, 2010.

[28] Shahab Jabarri Arfaee, Sandra Zilles, and R.C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 2011.

[29] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[30] Levente Kocsis, Csaba Szepesvari, and Jan Willemson. Improved monte-carlo search. 2006.

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

[32] 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, 1985.

[33] Richard E. Korf. Improved limited discrepancy search. In *Proceedings of the Thirteenth AAAI Conference on Artificial Intelligence*, pages 286–291. MIT Press, 1996.

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

[35] 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.

[36] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling*, 2005.

[37] Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. ARA*: Formal analysis. Technical Report CMU-CS-03-148, Carnegie Mellon University School of Computer Science, July 2003.

[38] XuanLong Nguyen and Subbarao Kambhampati. Reviving partial order planning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001.

[39] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, 1971.

[40] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*, pages 172–175. Morgan Kaufmann Publishers, Inc., 1998.

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

[42] 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.

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

[44] Ira Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computation issues in heuristic problem solving. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, pages 12–17, 1973.

[45] Bjoern Reese. AlphA*: An ϵ-admissible heuristic search algorithm. Unpublished, retrieved from http://home1.stofanet.dk/breese/papers.html, 1999.

[46] 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 Plannin and Scheduling*, pages 137–144.

[47] 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.

[48] Silvia Richter and Matthias Westphal. The LAMA planner — Using landmark counting in heuristic search. IPC 2008 short papers, `http://ipc.informatik.uni-freiburg.de/Planners`, 2008.

[49] Silvia Richter, Matthias Westphal, and Malte Helmert. LAMA 2008 and 2011 (planner abstract). IPC 2011 planner abstracts, 2011.

[50] Gabi Röger and Malte Helmert. The more the merrier: Combining heuristic estimators for satisficing planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, pages 246–249, 2010.

[51] Wheeler Ruml and Minh B. Do. Best-first utility-guided search. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 2378–2384, 2007.

[52] Stuart Russell and Peteer Norvig. *Artificial Intelligence: A Modern Approach.* 2003.

[53] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River, New Jersey, second edition, 2003.

[54] Stuart Russell and Eric Wefald. *Do the Right Thing: Studies in Limited Rationality.* MIT Press, 1991.

[55] Aleksander Sadikov and Ivan Bratko. Solving 20x20 puzzles. In *Computer games workshop 2007, Amsterdam, June 15-17, 2007*, pages 157–164, Amsterdam, The Netherlands, The Netherlands, 2007.

[56] Mehdi Samadi, Ariel Felner, and Jonathan Schaeffer. Learning from multiple heuristics. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence*, Summer 2008.

[57] Mehdi Samadi, Maryam Siabani, Ariel Felner, and Robert Holte. Compressing pattern databases with learning. In *Proceedings of the Eighteenth European Conference on Artificial Intelligence*, 2008.

[58] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, July 1959.

[59] Sudeshna Sarkar, P.P. Chakrabarti, and Sujoy Ghose. A framework for learning in search-based systems. In *IEEE Transactions on Knowledge and Data Engfineering*, volume 10, pages 563–575, July/August 1998.

[60] Robert Sedgewick. *Algorithms in C.* Addison-Wesley Professional, Boston, MA, 1992.

[61] Roni T. Stern, Rami Puzis, and Ariel Felner. Potential search: A bounded cost search algorithm. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling*, 2011.

[62] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[63] Jordan T. Thayer, J. Benton, and Malte Helmert. Better parameter-free anytime search by minimizing time between solutions. 2011.

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

[65] Jordan T. Thayer and Wheeler Ruml. Faster than weighted A*: An optimistic approach to bounded suboptimal search. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, Fall 2008.

[66] Jordan T. Thayer and Wheeler Ruml. Using distance estimates in heuristic search. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

[67] Jordan T. Thayer and Wheeler Ruml. Anytime heuristic search: Frameworks and algorithms. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, July 2010.

[68] Jordan T. Thayer and Wheeler Ruml. Finding acceptable solutions faster using inadmissible information. Technical Report 10-01, University of New Hampshire, April 2010.

[69] Jordan T. Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[70] Jordan T. Thayer, Wheeler Ruml, and Ephrat Bitton. Fast and loose in bounded suboptimal heuristic search. In *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR-08)*, 2008.

[71] 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.

[72] Jordan T. Thayer, Roni Stern, Ariel Felner, and Wheeler Ruml. Faster bounded-cost search using inadmissible estimates. In *Proceedings of the Twenty-second International Conference on Automated Planning and Scheduling*, 2012.

[73] Rcihard Valenzano, Nathan Sturtevant, Jonnathan Schaeffer, Karen Buro, and Akihiro Kishimoto. Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, 2010.

[74] Jur van den Berg, Rajat Shah, Arthur Huang, and Kenneth Y. Goldberg. ANA*: Anytime nonparametric A*. In *Proceedings of the Twenty-fifth AAAI Conference on Artificial Intelligence*, pages 105–111, 2011.

[75] 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.

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

[77] Yuehua Xu, Alan Fern, and Sungwook Yoon. Discriminative learning of beam-search heuristics for planning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, 2007.

[78] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. A* with partial expansion for large branching factor problems. In *Proceedings of the Seventeenth AAAI Conference on Artificial Intelligence*, pages 923–929, 2000.

[79] Rong Zhou and Eric A. Hansen. Multiple sequence alignment using Anytime A*. In *Proceedings of the Ninteenth AAAI Conference on Artificial Intelligence*, pages 975–976, 2002.

[80] Rong Zhou and Eric A. Hansen. Beam-stack search: Integrating backtracking with beam search. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling*, 2005.