PLANNING UNDER TIME PRESSURE

BY

Ethan Burns

BS of Computer Science, University of New Hampshire, 2006
MS of Computer Science, University of New Hampshire, 2008

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, 2013

ALL RIGHTS RESERVED

©2013

Ethan Burns

This dissertation has been examined and approved.

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

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

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

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

Shlomo Zilberstein,
Professor of Computer Science
University of Massachusetts Amherst

Date

DEDICATION

To my wife.

ACKNOWLEDGMENTS

Many people helped me during this undertaking. This section is about them.

My advisor, Wheeler Ruml, taught me so much of what was required for my dissertation. He taught me how to do research. He taught me to look for the good ideas among what often appear to be so many bad ones. He taught me to keep pushing forward even when I was long past the point where I wanted to give up. He taught me how to write, and so much more.

My family was very supportive; especially my wife, Cassandra Burns. She gave me encouragement, she was patient during long and late nights—especially paper deadlines. She didn't allow me to give up.

The work presented in Chapter 2 was done in conjunction with Sofia Lemons and Rong Zhou. Rong also hosted my internship at the Palo Alto Research Center during the summer of 2011, which resulted in Chapter 3. While the latest incarnation is quite different, a previous version of Bugsy, from Chapter 5, was proposed by Ruml and Do (2007). Scott Kiesel provided a lot of help, support, ideas, and coffee for the work in Chapter 6. Steve McCoy co-wrote `mid`, the game that inspired the platform domain from Chapters 5&6.

Everyone from Kingsbury W236, particularly the CS social crew. We had lots of insightful discussions, and even more nonsensical ones. The latter were what made some of the hardest times and latest nights bearable. Many from the group were also coauthors on works not presented here, and I am grateful for all of their help and ideas on those works too.

Much of my work was supported in part by NSF (grant 0812141), the DARPA CSSG program (grant D11AP00242), and the University of New Hampshire Dissertation Year Fellowship.

TABLE OF CONTENTS

DEDICATION iv

ACKNOWLEDGMENTS v

LIST OF TABLES ix

LIST OF FIGURES xiii

ABSTRACT xiv

Chapter 1 INTRODUCTION **1**

1.1 Heuristic Search 2

1.2 Categories of Search 4

1.3 Dissertation Outline 8

Chapter 2 PARALLEL HEURISTIC SEARCH **11**

2.1 Introduction 11

2.2 Previous Approaches 13

2.3 Parallel Best-NBlock-First (PBNF) 23

2.4 Empirical Evaluation: Optimal Search 31

2.5 Bounded Suboptimal Search 50

2.6 Empirical Evaluation: Bounded Suboptimal Search 55

2.7 Anytime Search 62

2.8 Empirical Evaluation: Anytime Search 65

2.9 Discussion 70

2.10 Conclusions 73

Chapter 3 PARALLEL MODEL CHECKING **83**

3.1 Introduction 83

3.2 Depth-first Versus Breadth-first Search 85

3.3 Hash-distributed Breadth-first Search 86
3.4 Parallel Structured Duplicate Detection 90
3.5 Experimental Results 95
3.6 Discussion 103
3.7 Conclusion and Future Work 104

Chapter 4 PREDICTING SEARCH PERFORMANCE 106

4.1 Introduction 106
4.2 Previous Work 108
4.3 Incremental Models of Search Trees 110
4.4 IDA*$_{IM}$ 117
4.5 Empirical Evaluation 122
4.6 Discussion 133
4.7 Conclusion 134

Chapter 5 PLANNING BEFORE EXECUTING 135

5.1 Introduction 135
5.2 Background 137
5.3 Previous Work 139
5.4 Off-line Bound Selection 143
5.5 Best-first Utility-guided Search 145
5.6 Experimental Evaluation 150
5.7 Related Work 168
5.8 Conclusion 172

Chapter 6 CONCURRENT PLANNING AND EXECUTION 174

6.1 Introduction 174
6.2 Previous Work 175
6.3 Goal Achievement Time 179
6.4 Traditional Real-time Algorithms 181

6.5 Metareasoning Real-time Algorithms 196
6.6 Discussion and Conclusion 211

Chapter 7 CONCLUSION **213**

Chapter A PSEUDO-CODE FOR SAFE PBNF **216**

Chapter B TLA$^+$ MODEL: HOT *N*BLOCKS **219**

Chapter C HISTOGRAMS **221**

Chapter D ANYTIME POLICY ESTIMATION **223**

Bibliography **227**

LIST OF TABLES

1-1 Search Objectives. 5

1-2 Search Settings. 7

2-1 Wall time on STRIPS planning problems. 44

2-2 Grid Pathfinding: Average speedup over serial weighted A* for various numbers of threads. 56

2-3 15-puzzle: Average speedup over serial weighted A* for various numbers of threads. 57

2-4 Average speedup over serial optimistic search for various numbers of threads. 57

2-5 Speed-up over serial weighted A* on STRIPS planning problems for various weights. 61

2-6 Speed-up of anytime search to optimality over serial AwA* on STRIPS planning using various weights. 69

2-7 Speed-up of anytime search to optimality over PBNF on STRIPS planning problems using various weights. 70

LIST OF FIGURES

2-1 A simple abstraction. Self-loops have been eliminated. 19

2-2 Two disjoint duplicate detection scopes. 20

2-3 A sketch of basic PBNF search, showing locking. 24

2-4 PBNF locking behavior vs minimum expansions on grid pathfinding with 62,500 $n$blocks. Each line represents a different number of threads. . . . 32

2-5 PBNF abstraction size: 5000x5000 grid pathfinding, 32 minimum expansions. 35

2-6 PRA* synchronization: 5000x5000 grids and easy sliding tile instances. . 75

2-7 PRA* abstraction: 5000x5000 grids and easy sliding tile instances. . . . 76

2-8 Simple parallel algorithms on unit cost, four-way 2000x1200 grid pathfinding. 77

2-9 Speedup results on grid pathfinding and the sliding tile puzzle. 78

2-10 Comparison of wall clock time for Safe PBNF versus AHDA* on the sliding tile puzzle. 79

2-11 Cumulative normalized f value counts for nodes expanded with eight threads on unit-cost four-way grid pathfinding (left) and the 15-puzzle (right). 79

2-12 Mean CPU time per open list operation. 80

2-13 Per-thread ratio of coordination time to wall time on unit-cost four-way pathfinding (top) and the 15-puzzle (bottom). 80

2-14 wPBNF speedup over wA* as a function of problem difficulty. 81

2-15 Raw data profiles (top) and lower hull profiles (bottom) for AwA* (left), AwPBNF (center), and ARA* (right). Grid unit-cost four-way pathfinding. 81

2-16 Grid unit-cost four-way pathfinding lower hull anytime profiles. 82

2-17 Korf's 100 15-puzzles lower hull anytime profiles. 82

3-1 A graph along with one of its possible abstractions (left) and two disjoint duplicate detection scopes of this graph (right) 92
3-2 Memory usage of PSDD, HD-BFS, AHD-BFS and BFS. 97
3-3 States expanded and memory used by PSDD, HD-BFS and BFS. 99
3-4 Parallel speedup for PSDD, HD-BFS, AHD-BFS, and parallel depth-first search. 100
3-5 Wall-clock seconds for PSDD, HD-BFS, AHD-BFS, parallel depth-first search, and serial breadth-first search. 105

4-1 Geometric versus non-geometric growth. 107
4-2 Histogram pruning. 115
4-3 Pseudo code for the simulation procedure used to estimate the f distribution. 116
4-4 Off-line training accuracy when predicting node expansions. 122
4-5 Off-line training accuracy when predicting the harder instance. 124
4-6 Unit tiles: growth rates and number of instances solved. 126
4-7 Square root tiles: growth rates and number of instances solved. 127
4-8 Vacuum maze: growth rates and number of instances solved. 128
4-9 Uniform Tree: growth rates and number of instances solved. 130
4-10 Different histogram sizes (a), IDA*, IDA*$_{CR}$, and IDA*$_{IM}$ growth rates (b) and number of instances solved (c) on the pancake problem. 131

5-1 Pseudo-code for BUGSY. 150
5-2 A screenshot of the platform path-finding domain (left), and a zoomed-out image of a single instance (right). The knight must find a path from its starting location, through a maze, to the door (on the right-side in the left image, and just above the center in the right image). 153

5-3 The visibility navigation instance for the platform domain's heuristic. The visibility path between the initial state and the goal state is drawn in red. . . . 155
5-4 Comparison of the optimal stopping policy and the learned stopping policy. 158
5-5 Comparison of the optimal stopping policy (continued). . . . 159
5-6 BUGSY: Resorting the open list (circles) vs not (boxes). . . . 161
5-7 BUGSY: Heuristic corrections. . . . 162
5-8 Comparison of techniques. . . . 164
5-9 Comparison of techniques (continued). . . . 165
5-10 Grid path-finding on a video game map. . . . 166
5-11 Nodes expanded, planning time, and execution time. . . . 167

6-1 LSS-LRTA*: multi-step, single-step, and dynamic lookahead. . . . 184
6-2 Example of heuristic error and f layers. . . . 185
6-3 f-layered lookahead. . . . 186
6-4 A standard heuristic and its error. . . . 187
6-5 An updated heuristic and its error. . . . 189
6-6 Using an updated heuristic and accounting for heuristic error. . . . 189
6-7 LSS-LRTA*: f-based lookahead and $\hat{f}$-based lookahead. . . . 191
6-8 Comparison of real-time techniques. . . . 192
6-9 Comparison with off-line techniques. . . . 194
6-10 Solution costs for dynamic $\hat{f}$ and single-step f. . . . 195
6-11 Comparison with DTA*. . . . 199
6-12 DTA* with and without extra learning. . . . 201
6-13 Single versus multiple local search spaces. . . . 204
6-14 Estimated f distributions for best action, α, and the second best action, β. 208
6-15 Comparison of Real-time searches, Ms. A*, and BUGSY. . . . 210

D-1 Pseudocode for profile estimation. . . . 225

D-2 Three different policies: (a) prefers cheaper solutions at any expense ($w_f = 1, w_t = 0$), (b) attempts to trade some search time for some solution cost ($w_f = 0.6, w_t = 1$), and (c) prefers to have any solution as fast as possible ($w_f = 0, w_t = 1$). 226

ABSTRACT

PLANNING UNDER TIME PRESSURE

by

Ethan Burns

University of New Hampshire, May, 2013

Heuristic search is a technique used pervasively in artificial intelligence and automated planning. Often an agent is given a task that it would like to solve as quickly as possible. It must allocate its time between planning the actions to achieve the task and actually executing them. We call this problem *planning under time pressure*. Most popular heuristic search algorithms are ill-suited for this setting, as they either search a lot to find short plans or search a little and find long plans. The thesis of this dissertation is: when under time pressure, an automated agent should explicitly attempt to minimize the sum of planning and execution times, not just one or just the other.

This dissertation makes four contributions. First we present new algorithms that use modern multi-core CPUs to decrease planning time without increasing execution. Second, we introduce a new model for predicting the performance of iterative-deepening search. The model is as accurate as previous offline techniques when using less training data, but can also be used online to reduce the overhead of iterative-deepening search, resulting in faster planning. Third we show offline planning algorithms that directly attempt to minimize the sum of planning and execution times. And, fourth we consider algorithms that plan online in parallel with execution. Both offline and online algorithms account for a user-specified preference between search and execution, and can greatly outperform the standard utility-oblivious techniques. By addressing the problem of planning under time pressure, these contributions demonstrate that heuristic search is no longer restricted to optimizing solution cost, obviating the need to choose between slow search times and expensive solutions.

CHAPTER 1

INTRODUCTION

Heuristic search is a technique used pervasively in the fields of artificial intelligence, automated planning and operations research to solve a wide range of problems from planning military deployments to planning tasks for a robot cleaning a messy kitchen. An automated agent can use heuristic search to construct a plan that, when executed, will achieve a desired task. The search algorithm explores different sequences of actions, looking for a sequence that will lead it to a desired goal state. In many situations, an agent is given a task that it would like to solve as quickly as possible. The agent must allocate its time between searching for the actions that will achieve the task and actually executing them. We call this problem *planning under time pressure*.

Most classic heuristic search techniques attempt to find short plans that will execute quickly. For example, the most well-known heuristic search algorithm, A* (Hart, Nilsson, & Raphael, 1968), finds optimal plans that have the minimum execution time. (In general A* optimizes any cost metric, but we are concerned with time so, throughout this work, unless otherwise stated, we assume that cost is given in units of time.) Due to the exponential memory requirements, it is often impractical or intractable to find optimal plans. Simon (1982) suggests that we discard the notion of so called *substantive rationality*, which focuses on optimality of solutions, and instead find techniques that *procedurally rational*—those that account for the limited nature of our computing environments and find *satisficing* solutions instead of optimal ones. Taking this idea to an extreme, other search techniques disregard execution time entirely and attempt to find any legal plan as quickly as possible. Greedy best-first search (Doran & Michie, 1966), will quickly find unboundedly suboptimal plans which are often very time consuming to execute. Algorithms such as weighted A* (Pohl,

1970) or more recently Explicit Estimation Search (EES, Thayer & Ruml, 2011), try to find a balance between optimality and unbounded suboptimality by returning solutions that are guaranteed to be within a user-specified factor of optimal. It is not clear, however, how to choose a suboptimality factor to properly trade execution time for planning time.

When time is of the essence, it is undesirable to spend more time looking for a shorter plan than the amount of time saved by the decreased plan length. Likewise, it is undesirable to spend more time executing a longer plan than the amount of time saved by the decreased planning time. As such, new techniques are required to explicitly solve the problem of planning under time pressure. The thesis of my dissertation is: when under time pressure, an automated agent should explicitly attempt to minimize the sum of planning and execution times, not just one or just the other.

In this dissertation I make four main contributions. First, I present methods to make optimal search faster, allowing it to be a more attractive approach for problems that need solutions very quickly. Second, I introduce a new technique for predicting search effort that can be used to both estimate problem difficulty and for on-line control of search. Third, I present an algorithm for a utility-based objective function, different from cost optimization, that allows one to specify their desired trade off between search time and execution cost. And fourth, I show how concurrent planning and execution can be successfully used to reduce goal achievement time. All of these techniques address the problem *planning under time pressure*, where an agent not only cares about the time required to execute a plan but also the time needed to find the plan in the first place.

1.1 Heuristic Search

In this section we will review heuristic search, introducing some of the terminology used throughout the rest of this dissertation.

At its core, heuristic search is a technique for finding a path in a graph. One of the most common applications of heuristic search is automated planning. In automated planning, each node in the search graph represents the state of the world, and each edge represents an

action that an agent may perform to change the world from one state to another. Because nodes usually represent states of the world, we often refer to the search graph as the *state space graph*, or just the state space. Heuristic search algorithms can solve planning problems by finding a path in the graph from one state of the world to a goal state. An agent can then execute the actions labeling each edge of the path to arrive at the desired world state.

In many planning problems the state space is very large—much too large to fit in the memory of a modern computer. However, for most problems of practical interest, only a small portion of the state space is required to find a solution path. Instead of representing the graph explicitly, e.g. by using an adjacency list or an adjacency matrix, heuristic search implementations tend to represent the state space graph implicitly, constructing the portions that are needed on-the-fly. This is done by using a function called *expand*. The expand function takes a search node as an argument and returns all of its immediate successors in the search graph. When expand has been evaluated on a node we say that it was *expanded* and that each of its successors were *generated*. The expand function effectively represents the entire search graph, as the graph can be fully instantiated by expanding all generated nodes.

Dijkstra's algorithm is probably the most popular algorithm for finding a path in a graph. It exhaustively considers all paths from the start node with length less than that of the shortest path length to a goal node. By doing so, it can guarantee that its eventual solution is the shortest path from the start to the goal. Dijkstra's algorithm can be modified slightly to handle graphs with weighted edges. This modified algorithm is called *Uniform Cost* search. Uniform Cost search uses an exhaustive procedure, similar to Dijkstra's algorithm, to find the *cheapest* path from the start to the goal. Both Dijkstra's algorithm and Uniform Cost search are called *uninformed* search algorithms because they use only the structure of the graph to find their solutions, they don't incorporate domain knowledge.

Heuristic search algorithms are quite similar to their uninformed cousins; however, by using a little extra information a heuristic search can find optimal paths with much less search effort. The extra information is given in the form of a function h, called the *heuristic*.

For any node in the search graph, the heuristic function returns an estimate of the cost-to-go from that node to a goal node. The most well-known heuristic search algorithm is A* (Hart et al., 1968). A* considers nodes in the search graph in order of the estimated solution cost going through the nodes, not just the cost to reach them. We call this value f. It is the sum of the cost to reach the node, notated g, plus the heuristic cost-to-go estimate for the node, h. If the heuristic function always returns a lower bound on the cost-to-go then it is called *admissible*. With an admissible heuristic A* returns cost-optimal solutions.

In addition to being admissible, a heuristic may satisfy a stronger property called *consistency*. A consistent heuristic is an admissible heuristic that does not decrease between a node and one of its descendents by more than the cheapest path between them. This is often written as $h(n) \geq c^*(n, m) + h(m)$, where n and m are two nodes, $c^*(n, m)$ is the cheapest path cost from n to m, and h is the heuristic. With a consistent heuristic A* is *optimally efficient* in that it expands the fewest nodes, short of tie-breaking, required to prove that its solution is optimal (Dechter & Pearl, 1988). A consequence of the optimal efficiency of A* is that one cannot hope to do better. So, if A* is intractable for a given problem then one must usually give up on optimality.

1.2 Categories of Search

Even though optimal search is often too costly, the generality of heuristic search makes it an extremely attractive technique for many problems. There has been a rather large number of heuristic search algorithms proposed to date. Some algorithms trade more search time for reduced memory usage, some trade more solution cost for reduced search time. Other algorithms bring heuristic search to entirely new areas such as real-time or multi-objective settings. In this section, we review some of the different objective functions and problems settings for heuristic search.

1. Solution Cost
   (a) Minimum cost ("optimal")
   (b) Cost < factor of optimal ("bounded suboptimal")
   (c) Cost < absolute value ("cost-bounded")
   (d) Minimum |cost − target| ("target value")
   (e) Cost < optimal + absolute value ("cost over optimal")
2. Search time
   (a) Minimum time ("greedy")
   (b) Time < absolute value ("contract")
3. Search time and Solution cost
   (a) Linear combination ("goal achievement")
   (b) Within a cost threshold or a time threshold, whichever happens first.
4. Multiple cost values per edge
   (a) Cost and duration: minimum cost that arrives on time (computed offline)
   (b) Cost and probability: minimum cost achieving goal within a bounded probability
   (c) Find a non-dominated frontier ("multi-objective")

Table 1-1: Search Objectives.

1.2.1 Objectives

Many search algorithms have been proposed to optimize a wide range of criteria. Table 1-1 gives a list of the search objectives described in this subsection.

Solution cost. The classic search objective is to optimize solution cost. This objective probably contains the largest number of algorithms and, in fact, has five sub-categories. All algorithms in this category have one thing in common: they focus on optimizing solution cost.

The most basic way to deal with solution cost is to minimize it. *Optimal* algorithms (item 1a in Table 1-1), such as A* and IDA* (Korf, 1985) find optimal solutions. Another common way to optimize solution cost is by finding a solution that is guaranteed to be within

a factor of the optimal cost (item 1b). This sub-category contains weighted A* (wA*, Pohl, 1970), Explicit Estimation Search (EES, Thayer & Ruml, 2011) and many more. We call these *bounded suboptimal* algorithms; they tend to find solutions much faster than optimal algorithms as they are not restricted to optimal solutions. *Cost-bounded* algorithms (item 1c, Thayer, Stern, Felner, & Ruml, 2012a) find solutions that are less costly than a given limit. There are also *target-value* search algorithms (item 1d, Kuhn, Schmidt, Price, Zhou, & Do, 2008; Schmidt, Kuhn, Price, de Kleer, & Zhou, 2009) that find a solution with a cost that is as close to a target value as possible. Finally some algorithms attempt to find solutions within an absolute constant of optimal (item 1e).

Search time. For many search problems it is hard to find any solution, never mind a cheap one. In this case, an unbounded suboptimal algorithm (item 2a) like greedy best-first search (Doran & Michie, 1966) may be used. *Greedy* algorithms are used when one wants to minimize search time and pay no heed to solution cost. If there is a limit on the amount of time available for search, a *contract search* algorithm (item 2b) will try to find a good solution while being cognizant of the deadline (Aine, Chakrabarti, & Kumar, 2010; Dionne, Thayer, & Ruml, 2011).

Search time and cost combination. An interesting category of objectives are those that combine both search time and cost. The combination may be a linear relation (item 3a, Ruml & Do, 2007), or perhaps specified as a threshold for both cost and time with the goal of returning a solution that within either threshold, whichever is reached first (item 3b). An example of this is a situation where any solution that costs less than $30 is acceptable, however, a solution is required within 1 hour. Since planning under time pressure is concerned with finding good solutions as quickly as possible, this category of objective functions, specifically linear combinations of cost and time, will be investigated further in Chapter 5.

1. Single Goal
   (a) Find a complete plan
   (b) Begin execution before planing completes
      i. Need next action within a bounded time ("real-time search")
      ii. Next action may be emitted when ready
2. Additional goals arrive after planning has started
   (a) Must complete plan before execution can start
   (b) Complete plans for some goals can execute before complete plans for other goals have been found ("reactive search")
   (c) Can start executing actions before a complete plan for any goal is found

Table 1-2: Search Settings.

Multi-objective. Some problems have multiple different costs associated with each action. For example, Kiesel, Burns, Wilt, and Ruml (2011) present the Waypoint Allocation and Motion Planning (WAMP) problem where search is performed in a space where each action has both a cost and time. In WAMP, the goal is to find a minimum cost path that arrives no later than a given time (item 4a). Other domains have a cost and probability where the goal is to find a cheap path that meets a specified upper bound on the probability of success (item 4b). *Multi-objective* problems are ones where the goal is to minimize all objectives. Multi-objective search algorithms find an entire frontier containing all non-dominated solutions, allowing the user to choose from among this set (item4c, Stewart & Chelsea C. White, 1991).

1.2.2 Settings

In addition to optimizing different objectives, heuristic search has been used in different problem settings too. In this subsection, we discuss the division of these settings into two categories: *single goal* and *multiple goal* with many sub-categories as shown in Table 1-2.

Single Goal

The first category of search settings is the traditional single goal setting where the algorithm is presented with one single goal and it must find just one plan to achieve this goal.

Sequential versus concurrent planning and execution. Traditionally, heuristic search algorithms are run before any execution takes place (item 1a in Table 1-2). An alternative to this approach is to allow search and execution to be interleaved or to take place concurrently (Benton, Do, & Ruml, 2007). Real-time search algorithms (item 1(b)i) such as Real-time A* (RTA*, Korf, 1990) or Local Search Space Learning Real-time A* (LSS-LRTA*, Koenig & Sun, 2009) achieve this by doing a fixed amount of lookahead search before emitting a plan prefix. Other algorithms, such as Decision-theoretic A* (DTA*, Russell & Wefald, 1991), perform a possibly unbounded amount of search between emitting actions. DTA* attempts to determine the correct amount of look-ahead based on a decision theoretic analysis of the search problem (item 1(b)ii). Chapter 6 focuses on algorithms in setting 1b.

Multiple Goals

The second category of search settings contains those algorithms that solve for multiple goals. Instead of solving a single problem, these techniques run continually, solving for different goals as they arrive. Techniques in this category range from making a complete plan for multiple goals before any execution begins (item 2a), making a complete plan for an individual goal before executing (item 2b, Nebel & Koehler, 1995; Fox, Gerevini, Long, & Serina, 2006; Ruml, Do, Zhou, & Fromherz, 2011), or allowing execution to occur concurrent with all planning (item 2c, Burns, Benton, Ruml, Do, & Yoon, 2012a).

1.3 Dissertation Outline

The next four chapters focus on single-goal off-line heuristic search (Table 1-2 item 1a). Chapter 2 is about parallel heuristic search for multicore CPUs. The main contribution of this chapter is the Parallel Best Nblock First search algorithm (PBNF, Burns, Lemons,

Ruml, & Zhou, 2010; Burns, Lemons, Zhou, & Ruml, 2009b). PBNF utilizes the parallelism inherent in modern hardware to decrease the time required to find optimal solutions to search problems. By decreasing search time without increasing the plan cost, PBNF is able to decrease the sum of planning and execution times, the goal of planning under time pressure. PBNF is compared to a variety of alternative techniques for parallelizing heuristic search, and experiments show that it is both faster and gives better parallel speedup on the domains tested. In this chapter, we also see how PBNF and other parallel search algorithms can be extended to support bounded suboptimal solutions (Table 1-1 item 1b) and anytime search (Burns, Lemons, Ruml, & Zhou, 2009a).

In Chapter 3, parallel search is extended from the domain of automated planning to the related field of model checking. We see how the PSDD algorithm (Zhou & Hansen, 2007), which uses the same technology underlying the PBNF, can verify formal systems faster and with less memory than alternative approaches (Burns & Zhou, 2012). This demonstrates the generality of the methods from Chapter 2 by applying them, successfully, to a different problem setting.

Chapter 4 shows a new technique for predicting the performance of iterative-deepening A* (IDA*, Korf, 1985), a memory-efficient, optimal heuristic search algorithm. The new *incremental model* presented in Chapter 3 is able to predict as accurately as the current state-of-the-art model for the 15-puzzle when trained offline, but, unlike the previous techniques, the incremental model can be trained online during search and can handle real-valued heuristic estimates (Burns & Ruml, 2013). We show that this model can be used to control an IDA* search by using information learned on completed iterations to determine a bound to use in the subsequent iteration. While IDA* guidance using the new model tends to be expensive in terms of CPU time, the gain in accuracy allows the search to remain robust.

Chapter 5 introduces search on a new search objective: minimizing a linear combination of search time and execution time (Table 1-1 item 3a). This chapter makes four main contributions. First, we see how to combine anytime heuristic search with dynamic

programming-based execution monitoring (Hansen & Zilberstein, 2001). To the best of our knowledge, we are the first to apply this monitoring technique to anytime heuristic search. Next, we see how to create a very simple portfolio-based method that uses automatic parameter selection for bounded suboptimal search to estimate the parameter setting that best optimizes the combination of search and execution time. Then the BUGSY algorithm is introduced. BUGSY optimizes a linear combination of search time and execution cost. Unlike the previous techniques, BUGSY does not require any offline training. We evaluate these approaches, and the results show two surprising things. First, the simple portfolio-based technique tends to give the best results on many of the domains tested. Second, BUGSY, an algorithm using only online estimation, is competitive with the offline techniques, and is the algorithm of choice when a representative set of training instances is unavailable.This demonstrates that heuristic search is no longer restricted to optimizing solely solution cost, freeing a user from the choice of either slow search times or expensive solutions.

Chapter 6 focuses on minimizing the sum of search and execution time when planning is allowed to happen concurrently with execution (Table 1-2 item 1b). Of the settings considered by this work, this is the most challenging since actions cannot be retracted after they are executed. As we will see, one of the benefits of this setting is that search and execution times may overlap, thus committing to a longer action earlier can increase the available time for further deliberation.

Chapter 7 concludes.

CHAPTER 2

PARALLEL HEURISTIC SEARCH

2.1 Introduction

The goal of planning under time pressure is to decrease the sum of both the planning and execution times. Often there is a tradeoff: more search yields less execution and less search yields more execution. This chapter attempts to bypass the standard tradeoff by using modern multicore CPUs to decrease search time without increasing execution time. By doing so, the sum of the search and execution times is therefore decreased.

It is widely anticipated that future microprocessors will not have faster clock rates, but instead more computing cores per chip. Tasks for which there do not exist effective parallel algorithms will suffer a slowdown relative to total system performance. In artificial intelligence, heuristic search is a fundamental and widely-used problem solving framework. In this chapter, we compare different approaches for parallelizing best-first search, a popular method underlying algorithms such as Dijkstra's algorithm and A* (Hart et al., 1968).

In best-first search, two sets of nodes are maintained: *open* and *closed*. Open contains the search frontier: nodes that have been generated but not yet expanded. In A*, open nodes are sorted by their f value, the estimated lowest cost for a solution path going through that node. Open is typically implemented using a priority queue. Closed contains all previously generated nodes, allowing the search to detect states that can be reached via multiple paths in the search space and avoid expanding them multiple times. The closed list is typically implemented as a hash table. One central challenge in parallelizing best-first search is avoiding contention between threads when accessing the open and closed lists. We look at a variety of methods for parallelizing best-first search, focusing on algorithms which

are based on two techniques: *parallel structured duplicate detection* and *parallel retracting A**.

Parallel structured duplicate detection (PSDD) was originally developed by Zhou and Hansen (2007) for parallel breadth-first search, in order to reduce contention on shared data structures by allowing threads to enjoy periods of synchronization-free search. PSDD requires the user to supply an abstraction function that maps multiple states, called an $n$block, to a single abstract state. We present a new algorithm based on PSDD called Parallel Best-NBlock-First (PBNF[1]). Unlike PSDD, PBNF extends easily to domains with non-uniform and non-integer move costs and inadmissible heuristics. Using PBNF in an infinite search space can give rise to livelock, where threads continue to search but a goal is never expanded. We will discuss how this condition can be avoided in PBNF using a method we call *hot nblocks*, as well as our use of bounded model checking to test its effectiveness. In addition, we provide a proof of correctness for the PBNF framework, showing its liveness and completeness in the general case.

Parallel retracting A* (PRA*) was created by Evett, Hendler, Mahanti, and Nau (1995). PRA* distributes the search space among threads by using a hash of a node's state. In PRA*, duplicate detection is performed locally; communication with peers is only required to transfer generated search-nodes to their home processor. PRA* is sensitive to the choice of hashing function used to distribute the search space. We show a new hashing function, based on the same state space abstraction used in PSDD, that can give PRA* significantly better performance in some domains. Additionally, we show that the communication cost incurred in a naïve implementation of PRA* can be prohibitively expensive. Kishimoto, Fukunaga, and Botea (2009) present a method that helps to alleviate the cost of communication in PRA* by using asynchronous message passing primitives.

We evaluate PRA* (and its variants), PBNF and other algorithms empirically using dual quad-core Intel machines. We study their behavior on three popular search domains:

[1]Peanut Butter 'N' (marshmallow) Fluff, also known as a fluffernutter, is a well-known children's sandwich in the USA.

STRIPS planning, grid pathfinding, and the venerable sliding tile puzzle. Our empirical results show that the simplest parallel search algorithms are easily outperformed by a serial A* search even when they are run with eight threads. The results also indicate that adding abstraction to the PRA* algorithm can give a larger increase in performance than simply using asynchronous communication, although using both of these modifications together may outperform either one used on its own. Overall, the PBNF algorithm often gives the best performance.

In addition to finding optimal solutions, we show how to adapt several of the algorithms to bounded suboptimal search, quickly finding w-admissible solutions (with cost within a factor of w of optimal). We provide new pruning criteria for parallel suboptimal search and prove that algorithms using them retain w-admissibility. Our results show that, for sufficiently difficult problems, parallel search may significantly outperform serial weighted A* search. We also found that the advantage of parallel suboptimal search increases with problem difficulty.

Finally, we demonstrate how some parallel searches, such as PBNF and PRA*, lead naturally to effective anytime algorithms. We also evaluate other obvious parallel anytime search strategies such as running multiple weighted A* searches in parallel with different weights. We show that the parallel anytime searches are able to find better solutions faster than their serial counterparts and they are also able to converge more quickly on optimal solutions.

2.2 Previous Approaches

There has been much previous work in parallel search. We will briefly summarize selected proposals before turning to the foundation of our work, the PRA* and PSDD algorithms.

2.2.1 Depth- and Breadth-first Approaches

Early work on parallel heuristic search investigated approaches based on depth-first search. Two examples are distributed tree search (Ferguson & Korf, 1988), and parallel window

search (Powley & Korf, 1991).

Distributed tree search begins with a single thread, which is given the initial state to expand. Each time a node is generated an unused thread is assigned to the node. The threads are allocated down the tree in a depth-first manner until there are no more free threads to assign. When this occurs, each thread will continue searching its own children with a depth-first search. When the solution for a subtree is found it is passed up the tree to the parent thread and the child thread becomes free to be re-allocated elsewhere in the tree. Parent threads go to sleep while their children search, only waking once the children terminate, passing solutions upward to their parents recursively. Because it does not keep a closed list, depth-first search cannot detect duplicate states and does not give good search performance on domains with many duplicate states, such as grid pathfinding and some planning domains.

Parallel window search parallelizes the iterative deepening A* (IDA*, see Korf, 1985) algorithm. In parallel window search, each thread is assigned a cost-bound and will perform a cost-bounded depth-first search of the search space. The problem with this approach is that IDA* will spend at least half of its search time on the final iteration and since every iteration is still performed in only a single thread, the search will be limited by the speed of a single thread. In addition, non-uniform costs can foil iterative deepening, because there may not be a good way to choose new upper-bounds that give the search a geometric growth.

Holzmann and Bošnački (2007) have been able to successfully parallelize depth-first search for model checking. The authors are able to demonstrate that their technique that distributes nodes based on search depth was able to achieve near linear speedup in the domain of model checking. Other research has used graphics processing units (GPUs) to parallelize breadth-first search for use in two-player games (Edelkamp & Sulewski, 2010). In the following sections we describe algorithms with the intent of parallelizing best-first search.

2.2.2 Simple Parallel Best-first Search

The simplest approach to parallel best-first search is to have open and closed lists that are shared among all threads (Kumar, Ramesh, & Rao, 1988). To maintain consistency of these data structures, mutual exclusion locks (mutexes) need to be used to ensure that a single thread accesses the data structure at a time. We call this search "parallel A*." Since each node that is expanded is taken from the open list and each node that is generated is looked up in the closed list by every thread, this approach requires a lot of synchronization overhead to ensure the consistency of its data structures. As we see in Section 2.4.3, this naïve approach performs worse than serial A*.

There has been much work on designing complex data structures that retain correctness under concurrent access. The idea behind these special *wait-free* data structures is that many threads can use portions of the data structure concurrently without interfering with one another. Most of these approaches use a special *compare-and-swap* primitive to ensure that, while modifying the structure, it does not get modified by another thread. We implemented a simple parallel A* search that we call lock-free parallel A* in which all threads access a single shared, concurrent priority queue and concurrent hash table for the open and closed lists, respectively. We implemented the concurrent priority queue data structure of Sundell and Tsigas (2005). For the closed list, we used a concurrent hash table which is implemented as an array of buckets, each of which is a concurrent ordered list as developed by Harris (2001). These lock-free data structures used to implement LPA* require a special lock-free memory manager that uses reference counting and a *compare-and-swap* based stack to implement a free list (Valois, 1995). We will see that, even with these sophisticated structures, a straightforward parallel implementation of A* does not give competitive performance.

One way of avoiding contention altogether is to allow one thread to handle synchronization of the work done by the other threads. K-Best-First Search (Felner, Kraus, & Korf, 2003) expands the best k nodes at once, each of which can be handled by a different thread. In our implementation, a master thread takes the k best nodes from open and gives one to

each worker. The workers expand their nodes and the master checks the children for duplicates and inserts them into the open list. This allows open and closed to be used without locking, however, in order to adhere to a strict k-best-first ordering this approach requires the master thread to wait for all workers to finish their expansions before handing out new nodes. In the domains used in this chapter, where node expansion is not particularly slow, we show that this method does not scale well.

One way to reduce contention during search is to access the closed list less frequently. A technique called *delayed duplicate detection* (DDD, Korf, 2003), originally developed for external-memory search, can be used to temporarily delay access to the a closed list. While several variations have been proposed, the basic principle behind DDD is that generated nodes are added to a single list until a certain condition is met (for example, a depth level is fully expanded, or some maximum list size is reached Stern and Dill (1998)) Once this condition has been met, the list is sorted to draw duplicate nodes together. All nodes in the list are then checked against the closed list, with only the best version being kept and inserted onto the open list. The initial DDD algorithm used a breadth-first frontier search and therefore only the previous depth-layer was required for duplicate detection. A parallel version was later presented by Niewiadomski, Amaral, and Holte (2006a), which split each depth layer into sections and maintained separate input and output lists for each. These were later merged in order to perform the usual sorting and duplicate detection methods. This large synchronization step, however, will incur costs similar to KBFS. It also depends upon an expensive workload distribution scheme to ensure that all processors have work to do, decreasing the bottleneck effect of nodes being distributed unevenly, but further increasing the algorithm's overhead. A later parallel best-first frontier search based on DDD was presented (Niewiadomski, Amaral, & Holte, 2006b), but incurs even further overhead by requiring synchronization between all threads to maintain a strict best-first ordering.

Jabbar and Edelkamp (2006) present an algorithm called parallel external A* (PEA*) that uses distributed computing nodes and external memory to perform a best-first search.

PEA* splits the search space into a set of "buckets" that each contain nodes with the same g and h values. The algorithm performs a best-first search by exploring all the buckets with the lowest f value beginning with the one with the lowest g. A master node manages requests to distribute portions of the current bucket to various processing nodes so that expanding a single bucket can be performed in parallel. To avoid contention, PEA* relies on the operating system to synchronize access to files that are shared among all of the nodes. Jabbar and Edelkamp used the PEA* algorithm to parallelize a model-checker and achieved almost linear speedup. While partitioning on g and h works on some domains it is not general if few nodes have the same g and h values. This tends to be the case in domains with real-valued edge costs. We now turn our attention to two algorithms that will reappear throughout the rest of this chapter: PRA* and PSDD.

2.2.3 Parallel Retracting A*

PRA* (Evett et al., 1995) attempts to avoid contention by assigning separate open and closed lists to each thread. A hash of the state representation is used to assign nodes to the appropriate thread when they are generated. (Full PRA* also includes a retraction scheme that reduces memory use in exchange for increased computation time; we do not consider that feature in this work.) The choice of hash function influences the performance of the algorithm, since it determines the way that work is distributed. Note that with standard PRA*, any thread may communicate with any of its peers, so each thread needs a synchronized message queue to which peers can add nodes. In a multicore setting, this is implemented by requiring a thread to take a lock on the message queue. Typically, this requires a thread that is sending (or receiving) a message to wait until the operation is complete before it can continue searching. While this is less of a bottleneck than having a single global, shared open list, we will see below that it can still be expensive. It is also interesting to note that PRA* and the variants mentioned below practice a type of delayed duplicate detection, because they store duplicates temporarily before checking them against a thread-local closed list and possibly inserting them into the open list.

Improvements

Kishimoto et al. (2009) note that the original PRA* implementation can be improved by removing the synchronization requirement on the message queues between nodes. Instead, they use the asynchronous send and receive functionality from the MPI message passing library (Snir & Otto, 1998) to implement an asynchronous version of PRA* that they call Hash Distributed A* (HDA*). HDA* distributes nodes using a hash function in the same way as PRA*, except the sending and receiving of nodes happens asynchronously. This means that threads are free to continue searching while nodes which are being communicated between peers are in transit.

In contact with the authors of HDA*, we have created an implementation of HDA* for multicore machines that does not have the extra overhead of message passing for asynchronous communication between threads in a shared memory setting. Also, our implementation of HDA* allows us to make a fair comparison between algorithms by sharing common data structures such as priority queues and hash tables.

In our implementation, each HDA* thread is given a single queue for incoming nodes and one outgoing queue for each peer thread. These queues are implemented as dynamically sized arrays of pointers to search nodes. When generating nodes, a thread performs a non-blocking call to acquire the lock[2] for the appropriate peer's incoming queue, acquiring the lock if it is available and immediately returning failure if it is busy, rather than waiting. If the lock is acquired then a simple pointer copy transfers the search node to the neighboring thread. If the non-blocking call fails the nodes are placed in the outgoing queue for the peer. This operation does not require a lock because the outgoing queue is local to the current thread. After a certain number of expansions, the thread attempts to flush the outgoing queues, but it is never forced to wait on a lock to send nodes. It also attempts to consume its incoming queue and only waits on the lock if its open list is empty, because in that case it has no other work to do. Using this simple and efficient implementation, we confirmed

[2]One such non-blocking call is the `pthread_mutex_trylock` function of the POSIX standard.

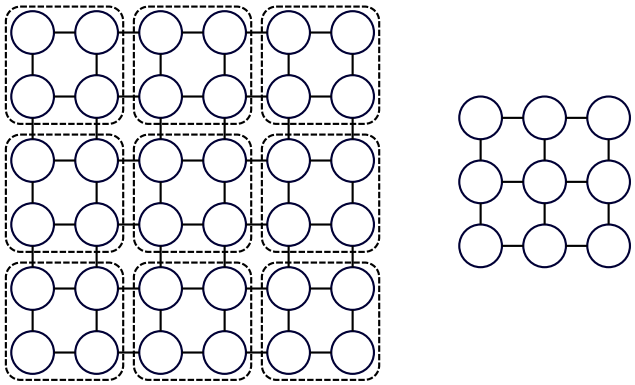


Figure 2-1: A simple abstraction. Self-loops have been eliminated.

the results of Kishimoto et al. (2009) that show that the asynchronous version of PRA* (called HDA*) outperforms the standard synchronous version. Full results are presented in Section 2.4.

PRA* and HDA* use a simple representation-based node hashing scheme that is the same one, for example, used to look up nodes in closed lists. We present two new variants, APRA* and AHDA*, that make use of state space abstraction to distribute search nodes among the processors. Instead of assigning nodes to each thread, each thread is assigned a set of blocks of the search space where each block corresponds to a state in the abstract space. The intuition behind this approach is that the children of a single node will be assigned to a small subset of all of the remote threads and, in fact, can often be assigned back to the expanding thread itself. This reduces the number of edges in the communication graph among threads during search, reducing the chances for thread contention. Abstract states are distributed evenly among all threads by using a modulus operator in the hope that open nodes will always be available to each thread. We discuss this type of abstraction in greater detail in the following section.

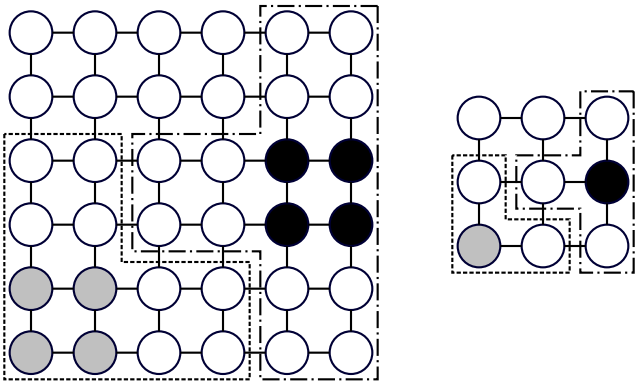


Figure 2-2: Two disjoint duplicate detection scopes.

2.2.4 Parallel Structured Duplicate Detection

PSDD is the major previously-proposed alternative to PRA*. The intention of PSDD is to avoid the need to lock on every node generation and to avoid explicitly passing individual nodes between threads. It builds on the idea of structured duplicate detection (SDD), which was originally developed for external memory search (Zhou & Hansen, 2004). SDD uses a *homomorphic abstraction function*, a many-to-one mapping from states in the original search space to states in an abstract space. The abstract node to which a state is mapped is called its *image*. An $n$block is the set of nodes in the state space that have the same image in the abstract space. The abstraction function creates an *abstract graph* that is homomorphic to the state space graph: if two states are successors in the state space, then their images are successors in the abstract graph. Figure 2-1 shows a state space graph (left) consisting of 36 nodes and an abstract graph (right) which consists of nine nodes. Each node in the abstract graph represents a grouping of four nodes, called an $n$block, in the original state space, shown by the dotted lines in the state space graph on the left.

Each $n$block has an open and closed list. To avoid contention, a thread will acquire exclusive access to an $n$block. Additionally, the thread acquires exclusive access to the $n$blocks that correspond to the successors in the abstract graph of the $n$block that it is searching. For each $n$block we call the set of $n$blocks that are its successors in the abstract

graph the its *duplicate detection scope*. This is because these are the only abstract nodes to which access is required in order to perform perfect duplicate detection when expanding nodes from the given $n$block. If a thread expands a node n in $n$block b the children of n must fall within b or one of the $n$blocks that are successors of b in the abstract graph. Threads can determine whether or not new states generated from expanding n are duplicates by simply checking the closed lists of $n$blocks in the duplicate detection scope. This does not require synchronization because the thread has exclusive access to this set of $n$blocks.

In PSDD, the abstract graph is used to find $n$blocks whose duplicate detection scopes are disjoint. These $n$blocks can be searched in parallel without any locking during node expansions. Figure 2-2 shows two disjoint duplicate detection scopes delineated by dashed lines with different patterns. An $n$block that is not in use by any thread and whose duplicate detection scope is also not in use is considered to be *free*. A free $n$block is available for a thread to acquire it for searching. Free $n$blocks are found by explicitly tracking, for each $n$block b, $\sigma(b)$, the number of $n$blocks among b's successors that are in use by another thread. An $n$block b can only be acquired when $\sigma(b) = 0$.

The advantage of PSDD is that it only requires a single lock, the one controlling manipulation of the abstract graph, and the lock only needs to be acquired by threads when finding a new free $n$block to search. This means that threads do not need to synchronize while expanding nodes, their most common operation.

Zhou and Hansen (2007) used PSDD to parallelize breadth-first heuristic search (Zhou & Hansen, 2006). In this algorithm, each $n$block has two lists of open nodes. One list contains open nodes at the current search depth and the other contains nodes at the next search depth. In each thread, only the nodes at the current search depth in an acquired $n$block are expanded. The children that are generated are put in the open list for the next depth in the $n$block to which they map (which will be in the duplicate detection scope of the $n$block being searched) as long as they are not duplicates. When the current $n$block has no more nodes at the current depth, it is swapped for a free $n$block that does have open nodes at this depth. If no more $n$blocks have open nodes at the current depth, all threads

synchronize and then progress together to the next depth. An admissible heuristic is used to prune nodes that fall on or above the current solution upper bound.

Improvements

While PSDD can be viewed as a general framework for parallel search, in our terminology, PSDD refers to an instance of SDD in a parallel setting that uses layer-based synchronization and breadth-first search. In this subsection, we present two algorithms that use the PSDD framework and attempt to improve on the PSDD algorithm in specific ways.

As implemented by Zhou and Hansen (2007), the PSDD algorithm uses the heuristic estimate of a node only for pruning; this is only effective if a tight upper bound is already available. To cope with situations where a good bound is not available, we have implemented a novel algorithm using the PSDD framework that uses iterative deepening (IDPSDD) to increase the bound. As we report below, this approach is not effective in domains such as grid pathfinding that do not have a geometrically increasing number of nodes within successive f bounds.

Another drawback of PSDD is that breadth-first search cannot guarantee optimality in domains where operators have differing costs. In anticipation of these problems, Zhou and Hansen (2004) suggest two possible extensions to their work, best-first search and a speculative best-first layering approach that allows for larger layers in the cases where there are few nodes (or $n$blocks) with the same f value. To our knowledge, we are the first to implement and test these algorithms.

Best-first PSDD (BFPSDD) uses f value layers instead of depth layers. This means that all nodes that are expanded in a given layer have the same (lowest) f value. BFPSDD provides a best-first search order, but may incur excessive synchronization overhead if there are few nodes in each f layer. To ameliorate this, we loosen the best-first ordering by enforcing that at least m nodes are expanded before abandoning a non-empty $n$block. (Zhou & Hansen, 2007 credit Edelkamp & Schrödl, 2000 with this idea.) Also, when populating the list of free $n$blocks for each layer, all of the $n$blocks that have nodes with the current

layer's f value are used or a minimum of k $n$blocks are added where k is four times the number of threads. (This value for k gave better performance than other values tested.) This allows us to add additional $n$blocks to small layers in order to amortize the cost of synchronization. In addition, we tried an alternative implementation of BFPSDD that used a range of f values for each layer. A parameter Δf was used to proscribe the width (in f values) of each layer of search. This implementation did not perform as well and we do not present results for it. With either of these enhancements, threads may expand nodes with f values greater than that of the current layer. Because the first solution found may not be optimal, search continues until all remaining nodes are pruned by the incumbent solution.

Having surveyed the existing approaches to parallel best-first search, we now present a new approach which comprises the main algorithmic contribution of this chapter.

2.3 Parallel Best-NBlock-First (PBNF)

In an ideal scenario, all threads would be busy expanding $n$blocks that contain nodes with the lowest f values. To approximate this, we combine PSDD's duplicate detection scopes with an idea from the Localized A* algorithm of Edelkamp and Schrödl (2000). Localized A*, which was designed to improve the locality of external memory search, maintains sets of nodes that reside on the same memory page. The decision of which set to process next is made with the help of a heap of sets ordered by the minimum f value in each set. By maintaining a heap of free $n$blocks ordered on each $n$blocks best f value, we can approximate our ideal parallel search. We call this algorithm Parallel Best-NBlock-First (PBNF) search.

In PBNF, threads use the heap of free $n$blocks to acquire the free $n$block with the best open node—the open node with the lowest f value. A thread will search its acquired $n$block as long as it contains nodes that are better than those of the $n$block at the front of the heap. If the f values of the open nodes in the acquired $n$block become greater than those of the best free $n$block, then the thread will release its current $n$block in an attempt to acquire the better one. There is no layer synchronization, so threads do not need to wait unless there are no free $n$blocks. The first solution found may be suboptimal, so search must continue

```
1. while there is an nblock with open nodes
2.    lock; b ← best free nblock; unlock
3.    while b is no worse than the best free nblock or we've done fewer than min expansions
4.       m ← best open node in b
5.       if f(m) ≥ f(incumbent), prune all open nodes in b
6.       else if m is a goal
7.          if f(m) < f(incumbent)
8.             lock; incumbent ← m; unlock
9.       else for each child c of m
10.         if c is not on the closed list of its nblock
11.            insert c in the open list of the appropriate nblock
```

Figure 2-3: A sketch of basic PBNF search, showing locking.

until all open nodes have f values worse than the incumbent solution. Figure 2-3 shows high-level pseudo-code for the algorithm.

Because PBNF is designed to tolerate a search order that is only approximately best-first, we have freedom to introduce optimizations that reduce overhead. It is possible that an $n$block has only a small number of nodes that are better than the best free $n$block, so PBNF avoids excessive switching by requiring a minimum number of expansions before an $n$block can be exchanged. Due to the minimum expansion requirement it is possible that the nodes expanded by a thread are arbitrarily worse than the frontier node with the minimum f. We refer to these expansions as *speculative*. This can be viewed as trading off node quality for reduced contention on the abstract graph. Section 2.4.1 shows the results of an experiment that evaluates this trade off.

Our implementation also attempts to reduce the time a thread is forced to wait on a lock by using non-blocking operations to acquire the lock whenever possible. Rather than sleeping if a lock cannot be acquired, a non-blocking lock operation (such as `pthread_mutex_trylock`) will immediately return failure. This allows a thread to continue expanding its current $n$block if the lock is busy. Both of these optimizations can introduce additional speculative expansions that would not have been performed in a serial best-first search.

2.3.1 Livelock

The greedy free-for-all order in which PBNF threads acquire free $n$blocks can lead to livelock in domains with infinite state spaces. Because threads can always acquire new $n$blocks without waiting for all open nodes in a layer to be expanded, it is possible that the $n$block containing the goal will never become free. This is because we have no assurance that all $n$blocks in its duplicate detection scope will ever be unused at the same time. For example, consider a situation where threads are constantly releasing and acquiring $n$blocks that prevent the goal $n$block from becoming free. To fix this, we have developed a method called *hot $n$blocks* where threads altruistically release their $n$block if they are interfering with a better $n$block. We call this enhanced algorithm *Safe PBNF*.

We use the term *the interference scope of b* to refer to the set of $n$blocks that, if acquired, would prevent b from being free. The interference scope includes not only b's successors in the abstract graph, but their predecessors too. In Safe PBNF, whenever a thread checks the heap of free $n$blocks to determine if it should release its current $n$block, it also ensures that its acquired $n$block is better than any of those that it interferes with. If the current $n$block interferes with a better one, it flags that $n$block as *hot*. Any thread that finds itself blocking a hot $n$block will release its $n$block in an attempt to free the hot one. For each $n$block b we define $\sigma_h(b)$ to be the number of hot $n$blocks that b is in the interference scope of. If $\sigma_h(b) \neq 0$, b is removed from the heap of free $n$blocks. This ensures that a thread will not acquire an $n$block that would prevent a hot $n$block from becoming free.

Consider, for example, an abstract graph containing four $n$blocks connected in a linear fashion: $A \leftrightarrow B \leftrightarrow C$. A possible execution of PBNF can alternate between a thread expanding from $n$blocks A and C. If this situation arises then $n$blocks B will never be considered free. If the only goals are located in $n$block B then, in an infinite search space there may be a livelock. With the "Safe" variant of PBNF, however, when expanding from either A or C a thread will make sure to check the f value of the best open node in $n$block B periodically. If the best node in B is seen to be better than the nodes in A or C then B will be flagged as "hot" and both $n$blocks A and C will no longer be eligible for expansion

until after $n$block B has been acquired.

More formally, let $\mathcal{N}$ be the set of all $n$blocks, $Predecessors(x)$ and $Successors(x)$ be the sets of predecessors and successors in the abstract graph of $n$block x, $\mathcal{H}$ be the set of all hot $n$blocks, $IntScope(b) = \{l \in \mathcal{N} : \exists x \in Successors(b) : l \in Predecessors(x)\}$ be the interference scope of an $n$block b and $x \prec y$ be a partial order over the $n$blocks where $x \prec y$ iff the minimum f value over all of the open nodes in x is lower than that of y. There are three cases to consider when attempting to set an $n$block b to hot with an undirected abstract graph:

1. $\mathcal{H} \cap IntScope(b) = \{\} \wedge \mathcal{H} \cap \{x \in \mathcal{N} : b \in IntScope(x)\} = \{\}$; none of the $n$blocks b interferes with or that interfere with b are hot, so b can be set to hot.

2. $\exists x \in \mathcal{H} : x \in IntScope(b) \wedge x \prec b$; b is interfered with by a better $n$block that is already hot, so b must not be set to hot.

3. $\exists x \in \mathcal{H} : x \in IntScope(b) \wedge b \prec x$; b is interfered with by an $n$block x that is worse than b and x is already hot. x must be un-flagged as hot (updating σ_h values appropriately) and in its place b is set to hot.

Directed abstract graphs have two additional cases:

4. $\exists x \in \mathcal{H} : b \in IntScope(x) \wedge b \prec x$; b is interfering with an $n$block x and b is better than x so un-flag x as hot and set b to hot.

5. $\exists x \in \mathcal{H} : b \in IntScope(x) \wedge x \prec b$; b is interfering with an $n$block x and x is better than b so do not set b to hot.

This scheme ensures that there are never two hot $n$blocks interfering with one another and that the $n$block that is set to hot is the best $n$block in its interference scope. As we verify below, this approach guarantees the property that if an $n$block is flagged as hot it will eventually become free. Full pseudo-code for Safe PBNF is given in Appendix A.

2.3.2 Correctness of PBNF

Given the complexity of parallel shared-memory algorithms, it can be reassuring to have proofs of correctness. In this subsection we will verify that PBNF exhibits various desirable properties:

Soundness

Soundness holds trivially because no solution is returned that does not pass the goal test.

Deadlock

There is only one lock in PBNF and the thread that currently holds it never attempts to acquire it a second time, so deadlock cannot arise.

Livelock

Because the interaction between the different threads of PBNF can be quite complex, we modeled the system using the TLA$^+$ (Lamport, 2002) specification language. Using the TLC model checker (Yu, Manolios, & Lamport, 1999) we were able to demonstrate a sequence of states that can give rise to a livelock in plain PBNF. Using a similar model we were unable to find an example of livelock in Safe PBNF when using up to three threads and 12 $n$blocks in an undirected ring-shaped abstract graph and up to three threads and eight $n$blocks in a directed graph.

In our model the state of the system is represented with four variables: *state*, *acquired*, *isHot* and *Succs*. The *state* variable contains the current action that each thread is performing (either *search* or *nextblock*). The *acquired* variable is a function from each thread to the ID of its acquired $n$block or the value *None* if it currently does not have an $n$block. The variable *isHot* is a function from $n$blocks to either `TRUE` or `FALSE` depending on whether or not the given $n$block is flagged as hot. Finally, the *Succs* variable gives the set of successors of each $n$block, defining the abstract graph.

The model has two actions: *doSearch* and *doNextBlock*. The *doSearch* action models

the search stage performed by a PBNF thread. Since we were interested in determining if there is a livelock, this action abstracts away most of the search procedure and merely models that the thread may choose a valid $n$block to flag as hot. After setting an $n$block to hot, the thread changes its state so that the next time it is selected to perform an action it will try to acquire a new $n$block. *doNextBlock* models a thread choosing its next $n$block if there is one available. After a thread acquires an $n$block (if one was free) it sets its state so that the next time it is selected to perform an action it will search.

The TLA$^+$ source of the model is located in Appendix B.

Formal proof: In addition to model checking, the TLA$^+$ specification language is designed to allow for formal proofs of properties. This allows properties to be proved for an unbounded space. Using our model we have completed a formal proof that a hot $n$block will eventually become free regardless of the number of threads or the abstract graph. We present here an English summary. First, we need a helpful lemma:

Lemma 1 *If an $n$block n is hot, there is at least one other $n$block in its interference scope that is in use. Also, n is not interfering with any other hot $n$blocks.*

Proof: Initially no $n$blocks are hot. This can change only while a thread searches or when it releases an $n$block. During a search, a thread can only set n to hot if it has acquired an $n$block m that is in the interference scope of n. Additionally, a thread may only set n to hot if it does not create any interference with another hot $n$block. During a release, if n is hot, either the final acquired $n$block in its interference scope is released and n is no longer hot, or n still has at least one busy $n$block in its interference scope. □

Now we are ready for the key theorem:

Theorem 1 *If an $n$block n becomes hot, it will eventually be added to the free list and will no longer be hot.*

Proof: We will show that the number of acquired $n$blocks in the interference scope of a hot $n$block n is strictly decreasing. Therefore, n will eventually become free.

Assume an $n$block n is hot. By Lemma 1, there is a thread p that has an $n$block in the

interference scope of n, and n is not interfering with or interfered by any other hot $n$blocks. Assume that a thread q does not have an $n$block in the interference scope of n. There are four cases:

1. p searches its $n$block. p does not acquire a new $n$block and therefore the number of $n$blocks preventing n from becoming free does not increase. If p sets an $n$block m to hot, m is not in the interference scope of n by Lemma 1. p will release its $n$block after it sees that n is hot (see case 2).

2. p releases its $n$block and acquires a new $n$block m from the free list. The number of acquired $n$blocks in the interference scope of n decreases by one as p releases its $n$block. Since m, the new $n$block acquired by p, was on the free list, it is not in the interference scope of n.

3. q searches its $n$block. q does not acquire a new $n$block and therefore the number of $n$blocks preventing n from becoming free does not increase. If q sets an $n$block m to hot, m is not in the interference scope of n by Lemma 1.

4. q releases its $n$block (if it had one) and acquires a new $n$block m from the free list. Since m, the new $n$block acquired by q, was on the free list, it is not in the interference scope of n and the number of $n$blocks preventing n from becoming free does not increase.

□

We can now prove the progress property that we really care about:

Theorem 2 *A node n with minimum f value will eventually be expanded.*

Proof: We consider n's $n$block. There are three cases:

1. The $n$block is being expanded. Because n has minimum f, it will be at the front of *open* and will be expanded.

2. The $n$block is free. Because it holds the node with minimum f value, it will be at the front of the free list and selected next for expansion, reducing to case 1.

3. The $n$block is not on the free list because it is in the interference scope of another $n$block that is currently being expanded. When the thread expanding that $n$block checks its interference scope, it will mark the better $n$block as hot. By Theorem 1, we will eventually reach case 2.

□

Completeness

This follows easily from liveness:

Corollary 1 *If the heuristic is admissible or the search space is finite, a goal will be returned if one is reachable.*

Proof: If the heuristic is admissible, we inherit the completeness of serial A* (Nilsson, 1980) by Theorem 2. Nodes are only re-expanded if their g value has improved, and this can happen only a finite number of times, so a finite number of expansions will suffice to exhaust the search space. □

Optimality

Because PBNF's expansion order is not strictly best-first, it operates like an anytime algorithm, and its optimality follows the same argument as that for algorithms such as Anytime A* (Hansen & Zhou, 2007).

Theorem 3 *PBNF will only return optimal solutions.*

Proof: After finding an incumbent solution, the search continues to expand nodes until the minimum f value among all frontier nodes is greater than or equal to the incumbent solution cost. This means that the search will only terminate with the optimal solution. □

Before discussing how to adapt PBNF to suboptimal and anytime search, we first evaluate its performance on optimal problem solving.

2.4 Empirical Evaluation: Optimal Search

We have implemented and tested the parallel heuristic search algorithms described above on three different benchmark domains: grid pathfinding, the sliding tile puzzle, and STRIPS planning. We will discuss each domain in turn. With the exception of the planning domain, the algorithms were programmed in C++ using the POSIX threading library and run on dual quad-core Intel Xeon E5320 1.86GHz processors with 16Gb RAM. For the planning results the algorithms were written independently in C from the pseudo code in Appendix A. This gives us additional confidence in the correctness of the pseudo code and our performance claims. The planning experiments were run on dual quad-core Intel Xeon X5450 3.0GHz processors limited to roughly 2GB of RAM. All open lists and free lists were implemented as binary heaps except in PSDD and IDPSDD which used a queue giving them less overhead since they do not require access to minimum valued elements. All closed lists were implemented as hash tables. PRA* and APRA* used queues for incoming nodes, and a hash table was used to detect duplicates in both open and closed. For grids and sliding tiles, we used the jemalloc library (Evans, 2006), a special multi-thread-aware malloc implementation, instead of the standard glibc (version 2.7) malloc, because we found that the latter scaled poorly above 6 threads. We configured jemalloc to use 32 memory arenas per CPU. In planning, a custom memory manager was used which is also thread-aware and uses a memory pool for each thread.

On grids and sliding tiles abstractions were hand-coded and, $n$block data structures were created lazily, so only the visited part of abstract graph was instantiated. The time taken to create the abstraction is accounted for in all of the wall time measurements for these two domains. In STRIPS planning the abstractions were created automatically and the creation times for the abstractions are reported separately as described in Section 2.4.5.

2.4.1 Tuning PBNF

In this section we present results for a set of experiments that we designed to test the behavior of PBNF as some of its parameters are changed. We study the effects of the

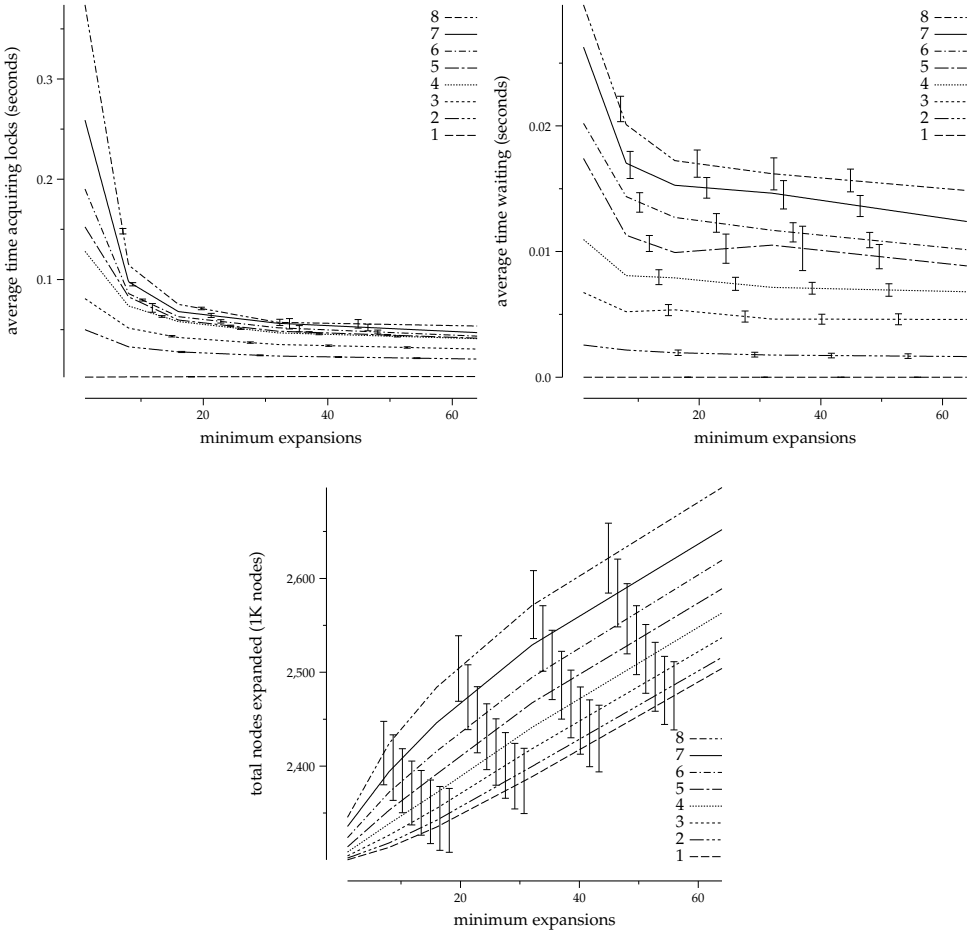


Figure 2-4: PBNF locking behavior vs minimum expansions on grid pathfinding with 62,500 $n$blocks. Each line represents a different number of threads.

two important parameters of the PBNF algorithm: minimum expansions required before switching to search a new $n$block and the size of the abstraction. This study used twenty 5000x5000 four-connected grid pathfinding instances with unit-cost moves where each cell has a 0.35 probability of being an obstacle. The heuristic used was the Manhattan distance to the goal location. Error bars in the plots show 95% confidence intervals and the legends are sorted by the mean of the dependent variable in each plot.

In the PBNF algorithm, each thread must perform a minimum number of expansions before it is able to acquire a new $n$block for searching. Requiring more expansions between

switches is expected to reduce the contention on the $n$block graph's lock but could increase the total number of expanded nodes. We created an instrumented version of the PBNF algorithm that tracks the time that the threads have spent trying to acquire the lock and the amount of time that threads have spent waiting for a free $n$block. We fixed the size of the abstraction to 62,500 $n$blocks and varied the number of threads (from 1 to 8) and minimum expansions (1, 8, 16, 32 and 64 minimum expansions).

The upper left panel in Figure 2-4 shows the average amount of *CPU time* in seconds that each thread spent waiting to acquire the lock (y-axis) as the minimum expansions parameter was increased (x-axis). Each line in this plot represents a different number of threads. We can see that the configuration which used the most amount of time trying to acquire the lock was with eight threads and one minimum expansion. As the number of threads decreased, there was less contention on the lock as there were fewer threads to take it. As the number of minimum required expansions increased the contention was also reduced. Around eight minimum expansions the benefit of increasing the value further seemed to greatly diminish.

The upper right panel of Figure 2-4 shows the results for the CPU time spent waiting for a free $n$block (y-axis) as minimum expansions was increased (x-axis). This is different than the amount of time waiting on the lock because, in this case, the thread successfully acquired the lock but then found that there were no free $n$blocks available to search. We can see that the configuration with eight threads and one minimum expansion caused the longest amount of time waiting for a free $n$block. As the number of threads decreased and as the required number of minimum expansions increased the wait time decreased. The amount of time spent waiting, however, seems fairly insignificant because it is an order of magnitude smaller than the lock time, indicating that PBNF was able to keep threads busy searching. Again, we see that around eight minimum expansions the benefit of increasing seemed to diminish.

The final panel, on the bottom in Figure 2-4, shows the total number of nodes expanded (y-axis, which is in thousands of nodes) as minimum expansions was increased. Increasing

the minimum number of expansions that a thread must make before switching to an $n$block with better nodes caused the search algorithm to explore more of the space that may not have been covered by a strict best-first search. As more of these "speculative" expansions were performed the total number of nodes encountered during the search increased. We can also see that adding threads increased the number of expanded nodes too.

From the results of this experiment it appears that requiring more than eight expansions before switching $n$blocks had a decreasing benefit with respect to locking and waiting time. In our non-instrumented implementation of PBNF we found that slightly greater values for the minimum expansion parameter lead to the best total wall times. For each domain below we use the value that gave the best total wall time in the non-instrumented PBNF implementation.

Since PBNF uses abstraction to decompose a search space it is also important to understand the effect of abstraction size on search performance. Our hypothesis was that using too few abstract states would lead to only a small number of free $n$blocks therefore making threads spend a lot of time waiting for an $n$block to become free. On the other hand, if there are too many abstract states then there will be too few nodes in each $n$block. If this happens, threads will perform only a small amount of work before exhausting the open nodes in their $n$block and being forced to switch to a new portion of the search space. Each time a thread must switch $n$blocks the contention on the lock is increased. Figure 2-5 shows the results of an experiment that was performed to verify this theory. In each plot we have fixed the minimum expansions parameter to 32 (which gave the best total wall time on grid pathfinding) and varied the number of threads (from 1 to 8) and the size of the abstraction (10,000, 62,500 and 250,000 $n$blocks).

The upper left panel of Figure 2-5 shows a plot of the amount of CPU seconds spent trying to acquire the lock (y-axis) versus the size of the abstraction (x-axis). As expected, when the abstraction had very few nodes there was little time spent waiting on the lock, but as the size of the abstraction grew and the number of threads increased the amount of time spent locking increased. At eight threads with 250,000 $n$blocks over 1 second of

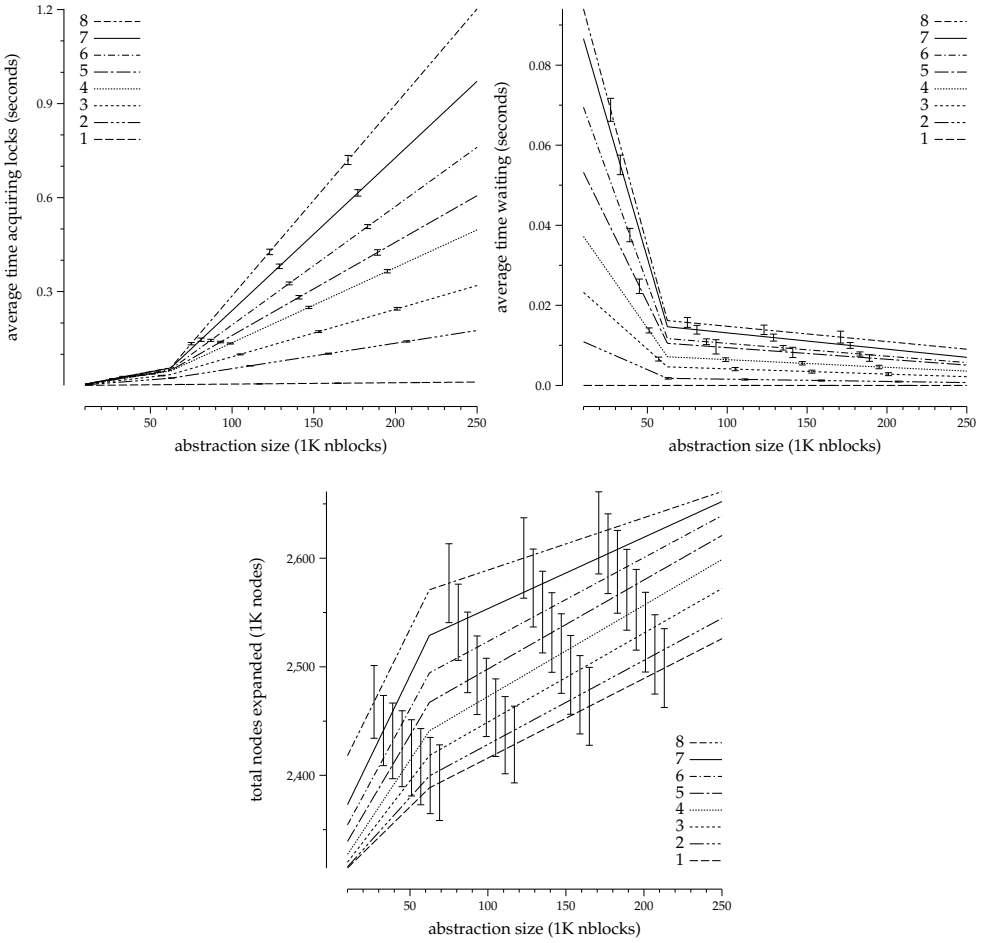


Figure 2-5: PBNF abstraction size: 5000x5000 grid pathfinding, 32 minimum expansions.

CPU time was spent waiting to acquire the lock. We suspect that this is because threads exhausted all open nodes in their $n$blocks and were forced to take the lock to acquire a new portion of the search space.

The upper right panel of Figure 2-5 shows the amount of time that threads spent waiting for an $n$block to become free after having successfully acquired the lock only to find that no $n$blocks are available. Again, as we suspected, the amount of time that threads wait for a free $n$block decreases as the abstraction size is increased. The more available $n$blocks, the more disjoint portions of the search space will be available. As with our experiments for minimum expansions, the amount of time spent waiting seems to be relatively insignificant

compared to the time spent acquiring locks.

The bottom panel in Figure 2-5 shows that the number of nodes that were expanded increased as the size of the abstraction was increased. For abstractions with mode nodes the algorithm expanded more nodes. This is because each time a thread switches to a new $n$block it is forced to perform at least the minimum number of expansions, so more switches means more forced expansions.

2.4.2 Tuning PRA*

We now turn to looking at the performance impact on PRA* of abstraction and asynchronous communication. First, we compare PRA* with and without asynchronous communication. Results from a set of experiments on twenty 5000x5000 grid pathfinding and a set of 250 random 15-puzzle instances that were solvable by A* in 3 million expansions are shown in Figure 2-6. The line labeled *sync. (PRA*)* used synchronous communication, *async. sends* used synchronous receives and asynchronous sends, *async. receives* used synchronous sends and asynchronous receives, and *async. (HDA*)* used asynchronous communication for both sends and receives. As before, the legend is sorted by the mean performance and the error bars represent the 95% confidence intervals on the mean. The vertical lines in the plots for the life cost grid pathfinding domains show that these configurations were unable to solve instances within the 180 second time limit.

The combination of both asynchronous sends and receives provided the best performance. We can also see from these plots that making sends asynchronous provided more of a benefit than making receives asynchronous. This is because, without asynchronous sends, each node that is generated will stop the generating thread in order to communicate. Even if communication is batched, each send may be required to go to a separate neighbor and therefore a single send operation may be required per-generation. For receives, the worst case is that the receiving thread must stop at each expansion to receive the next batch nodes. Since the branching factor in a typical search space is approximately a constant value greater than one, there will be approximately a constant factor more send commu-

nications as there are receive communications in the worst case. Therefore, making sends asynchronous reduces the communication cost more than receives.

Figure 2-7 shows the results of an experiment that compares PRA* using abstraction to distribute nodes among the threads versus PRA* with asynchronous communication. The lines are labeled as follows: *sync. (PRA*)* used only synchronous communication, *async. (HDA*)* used only asynchronous communication and *sync. with abst. (APRA*)* used only synchronous communication and used abstraction to distribute nodes among the threads and *async. and abst. (AHDA*)* used a combination of asynchronous communication and abstraction. Again, the vertical lines in the plots for the life cost grid pathfinding domains show that these configurations were unable to solve instances within the 180 second time limit.

It is clear from these plots that the configurations of PRA* that used abstraction gave better performance than PRA* without abstraction in the grid pathfinding domain. The reason for this is because the abstraction in grid pathfinding will often assign successors of a node being expanded back to the thread that generated them. When this happens no communication is required and the nodes can simply be checked against the local closed list and placed on the local open list if they are not duplicates. With abstraction, the only time that communication will be required is when a node on the "edge" of an abstract state is expanded. In this case, *some* of the children will map into a different abstract state and communication will be required. This experiment also shows that the benefits of abstraction were greater than the benefits of asynchronous communication in the grid pathfinding problems. We see the same trends on the sliding tile instances, however they are not quite as pronounced—the confidence intervals often overlap.

Overall, it appears that the combination of PRA* with both abstraction for distributing nodes among the different threads and using asynchronous communication gave the best performance. In the following section we show the results of a comparison between this variant of PRA*, the Safe PBNF algorithm and the best-first variant of PSDD.

2.4.3 Grid Pathfinding

In this section, we evaluate the parallel algorithms on the grid pathfinding domain. The goal of this domain is to navigate through a grid from an initial location to a goal location while avoiding obstacles. We used two cost models (discussed below) and both four-way and eight-way movement. On the four-way grids, cells were blocked with a probability of 0.35 and on the eight-way grids cells were blocked with a probability of 0.45. The abstraction function that was used maps blocks of adjacent cells to the same abstract state, forming a coarser abstract grid overlaid on the original space. The heuristic was the Manhattan distance to the goal location. The hash values for states (which are used to distribute nodes in PRA* and HDA*) are computed as: $x \cdot y_{max} + y$ of the state location. This gives a minimum perfect hash value for each state. For this domain we were able to tune the size of the abstraction and our results show execution with the best abstraction size for each algorithm where it is relevant.

Four-Way Unit Cost

In the unit-cost model, each move has the same cost: one.

Less Promising Algorithms Figure 2-8, shows a performance comparison between algorithms that, on average, were slower than serial A*. These algorithms were tested on 20 unit-cost four-way movement 1200x2000 grids with the start location in the bottom left corner and the goal location in the bottom right. The x-axis shows the number of threads used to solve each instance and the y-axis shows the mean wall-clock time in seconds. The error bars give a 95% confidence interval on the mean wall-clock time and the legend is sorted by the mean performance.

From this figure we can see that PSDD gave the worst average solution times. We suspect that this was because the lack of a tight upper bound which PSDD uses for pruning. We see that A* with a shared lock-free open and closed list (LPA*) took, on average, the second longest amount of time to solve these problems. LPA*'s performance improved

up to 5 threads and then started to drop off as more threads were added. The overhead of the special lock-free memory manager along with the fact that access to the lock-free data structures may require back-offs and retries could account for the poor performance compared to serial A*. The next algorithm, going down from the top in the legend, is KBFS which slowly increased in performance as more threads were added however it was not able to beat serial A*. A simple parallel A* implementation (PA*) using locks on the open and closed lists performed worse as threads were added until about four where it started to give a very slow performance increase matching that of KBFS. The PRA* algorithm using a simple state representation based hashing function gave the best performance in this graph but it was fairly erratic as the number of threads changed, sometimes increasing and sometimes decreasing. At 6 and 8 threads, PRA* was faster than serial A*.

We have also implemented the IDPSDD algorithm which tries to find the upper bound for a PSDD search using iterative deepening, but the results are not shown on the grid pathfinding domains. The non-geometric growth in the number of states when increasing the cost bound leads to very poor performance with iterative deepening on grid pathfinding. Due to the poor performance of the above algorithms, we do not show their results in the remaining grid, tiles or planning domains (with the exception of PSDD which makes a reappearance in the STRIPS planning evaluation of Section 2.4.5, where we supply it with an upper bound).

More Promising Algorithms The upper left plot in Figure 2-9 shows the performance of algorithms on unit-cost four-way grid pathfinding problems. The y-axis represents the speedup over serial A* and the x-axis shows the number of threads in use for each data point. Error bars indicate 95% confidence intervals on the mean over 20 different instances. Algorithms in the legend are ordered by their average performance. The line labeled "Perfect speedup" shows a perfect linear speedup where each additional thread increases the performance linearly.

A more practical reference point for speedup is shown by the "Achievable speedup" line. On a perfect machine with n processors, running with n cores should take time that

decreases linearly with n. On a real machine, however, there are hardware considerations such as memory bus contention that prevent this n-fold speedup. To estimate this overhead for our machines, we ran sets of n independent A* searches in parallel for $1 \leq n \leq 8$ and calculated the total time for each set to finish. On a perfect machine all of these sets would take the same time as the set with $n = 1$. We compute the "Achievable speedup" with the ratio of the actual completion times to the time for the set with $n = 1$. At t threads given the completion times for the sets, $\langle C_1, C_2, ..., C_n \rangle$, $achievable\_speedup(t) = \frac{t \cdot C_1}{C_t}$.

The upper left panel shows a comparison between AHDA* (PRA* with asynchronous communication and abstraction), BFPSDD and Safe PBNF algorithm on the larger (5000x5000) unit-cost four-way problems. Safe PBNF was superior to any of the other algorithms, with steadily decreasing solution times as threads were added and an average speedup over serial A* of more than 6x when using eight threads. AHDA* had less stable performance, sometimes giving a sharp speedup increase and sometimes giving a decrease as more threads were added. At seven threads, where AHDA* gave its best performance, it was able to reach 6x speedup over serial A* search. The BFPSDD algorithm solved problems faster as more threads were added however it was not as competitive as PBNF and AHDA* giving no more than 3x speedup over serial A* with eight threads.

Four-Way Life Cost

Moves in the life cost model have a cost of the row number of the state where the move was performed—moves at the top of the grid are free, moves at the bottom cost 4999 (Ruml & Do, 2007). This differentiates between the shortest and cheapest paths which has been shown to be a very important distinction (Richter & Westphal, 2010; Cushing, Bentor, & Kambhampati, 2010). The left center plot in Figure 2-9 shows these results in the same format as for the unit-cost variant – number of threads on the x axis and speedup over serial A* on the y axis. On average, Safe PBNF gave better speedup than AHDA*, however AHDA* outperformed PBNF at six and seven threads. At eight threads, however, APRA* did not perform better than at seven threads. Both of these algorithms achieve

speedups that are very close to the "Achievable speedup" for this domain. Again BFPSDD gave the worst performance increase as more threads were added reaching just under 3x speedup.

Eight-Way Unit Cost

In eight-way movement path planning problems, horizontal and vertical moves have cost 1, but diagonal movements cost $\sqrt{2}$. These real-valued costs make the domain different from the previous two path planning domains. The upper right panel of Figure 2-9 shows number of threads on the x axis and speedup over serial A* on the y axis for the unit cost eight-way movement domain. We see that Safe PBNF gave the best average performance reaching just under 6x speedup at eight threads. AHDA* did not outperform Safe PBNF on average, however it was able to achieve a just over 6x speedup over serial A* at seven threads. Again however, we see that AHDA* did not give very stable performance increases with more threads. BFPSDD improved as threads were added out to eight but it never reached more than 3x speedup.

Eight-Way Life Cost

This model combines the eight-way movement and the life cost models; it tends to be the most difficult path planning domain presented in our results. The right center panel of Figure 2-9 shows threads on the x axis and speedup over serial A* on the y axis. AHDA* gave the best average speedup over serial A* search, peaking just under 6x speedup at seven threads. Although it outperformed Safe PBNF on average at eight threads AHDA* has a sharp decrease in performance reaching down to almost 5x speedup where Safe PBNF had around 6x speedup over serial A*. BFPSDD again peaks at just under 3x speedup at eight threads.

2.4.4 Sliding Tile Puzzle

The sliding tile puzzle is a common domain for benchmarking heuristic search algorithms. For these results, we use 250 randomly generated 15-puzzles that serial A* was able to solve within 3 million expansions.

The abstraction used for the sliding tile puzzles ignores the numbers on a set of tiles. For example, the results shown for Safe PBNF in the bottom panel of Figure 2-9 use an abstraction that looks at the position of the blank, one and two tiles. This abstraction gives 3360 $n$blocks. In order for AHDA* to get the maximum amount of expansions that map back to the expanding thread (as described above for grids), its abstraction uses the one, two and three tile. Since the position of the blank is ignored, any state generation that does not move the one, two or three tiles will generate a child into the same $n$block as the parent therefore requiring no communication. The heuristic that was used in all algorithms was the Manhattan distance heuristic. The hash value used for tiles states was a perfect hash value based on the techniques presented by Korf and Schultze (2005).

The bottom panel of Figure 2-9 shows the results for AHDA*, and Safe PBNF on these sliding tiles puzzle instances. The plot has the number of threads on the x axis and the speedup over serial A* on the y axis. Safe PBNF had the best mean performance but there was overlap in the confidence intervals with AHDA*. BFPSDD was unable to show a speedup over serial A* and its performance was not shown in this plot.

Because sliding tile puzzles vary so much in difficulty, in this domain we also did a paired-difference test, shown in Figure 2-10. The data used for Figure 2-10 was collected on the same set of runs as shown in the bottom panel of Figure 2-9. The y-axis in this figure, however, is the average, over all instances, of the time that AHDA* took on that instance minus the time that Safe PBNF took. This paired test gives a more powerful view of the algorithms' relative performance. Values greater than 0.0 represent instances where Safe PBNF was faster than AHDA* and values lower than 0.0 represent those instances where AHDA* was faster. The error bars show the 95% confidence interval on the mean. We can clearly see that the Safe PBNF algorithm was significantly faster than AHDA* across all

numbers of threads from 1 to 8.

2.4.5 STRIPS Planning

In addition to the path planning and sliding tiles domains, the algorithms were also embedded into a domain-independent, optimal, sequential STRIPS planner. In contrast to the previous two domains where node expansion is very quick and therefore it is difficult to achieve good parallel speedup, node expansion in STRIPS planning is relatively slow. The planner used in these experiments uses regression and the max-pair admissible heuristic of Haslum and Geffner (2000). The abstraction function used in this domain is generated dynamically on a per-problem basis and, following Zhou and Hansen (2007), this time was not taken into account in the solution times presented for these algorithms. The abstraction function is generated by greedily searching in the space of all possible abstraction functions (Zhou & Hansen, 2006). Because the algorithm needs to evaluate one candidate abstraction for each of the unselected state variables, it can be trivially parallelized by having multiple threads work on different candidate abstractions.

Table 2-1 presents the results for A*, AHDA*, PBNF, Safe PBNF, PSDD (given an optimal upper bound for pruning and using divide-and-conquer solution reconstruction), APRA* and BFPSDD. The values of each cell are the total wall time in seconds taken to solve each instance. A value of 'M' indicates that the program ran out of memory. The best result on each problem and results within 10% of the best are marked in **bold**. Generally, all of the parallel algorithms were able to solve the instances faster as they were allowed more threads. All of the parallel algorithms were able to solve instances much faster than serial A* at seven threads. The PBNF algorithm (either PBNF or Safe PBNF) gave the best solution times in all but three domains. Interestingly, while plain PBNF was often a little faster than the safe version, it failed to solve two of the problems. This may be due to livelock, although it could also simply be because the hot $n$blocks fix forces Safe PNBF to follow a different search order than PBNF. AHDA* tended to give the second-best solution times, followed by PSDD which was given the optimal solution cost up-front for pruning.

	A*	AHDA*				PBNF			
threads	1	1	3	5	7	1	3	5	7
logistics-6	2.30	1.44	0.70	0.48	**0.40**	1.27	0.72	0.58	0.53
blocks-14	5.19	7.13	5.07	2.25	**2.13**	6.28	3.76	2.70	2.63
gripper-7	117.78	59.51	33.95	15.97	12.69	39.66	16.43	10.92	**8.57**
satellite-6	130.85	95.50	33.59	24.11	18.24	68.14	34.15	20.84	16.57
elevator-12	335.74	206.16	96.82	67.68	57.10	156.64	56.25	34.84	**26.72**
freecell-3	199.06	147.96	93.55	38.24	**27.37**	185.68	64.06	44.05	36.08
depots-7	M	299.66	126.34	50.97	39.10	M	M	M	M
driverlog-11	M	315.51	85.17	51.28	48.91	M	M	M	M
gripper-8	M	532.51	239.22	97.61	76.34	229.88	95.63	60.87	**48.32**

	SafePBNF				PSDD			
threads	1	3	5	7	1	3	5	7
logistics-6	1.17	0.64	0.56	0.62	1.20	0.78	0.68	0.64
blocks-14	6.21	2.69	**2.20**	**2.02**	6.36	3.57	2.96	2.87
gripper-7	39.58	16.87	11.23	**9.21**	65.74	29.37	21.88	19.19
satellite-6	77.02	24.09	17.29	**13.67**	61.53	23.56	16.71	**13.26**
elevator-12	150.39	53.45	34.23	**27.02**	162.76	62.68	43.34	36.66
freecell-3	127.07	47.10	38.07	37.02	126.31	53.76	45.47	43.71
depots-7	156.36	63.04	42.91	**34.66**	159.98	73.00	57.65	54.70
driverlog-11	154.15	59.98	38.84	**31.22**	155.93	63.20	41.85	**34.02**
gripper-8	235.46	98.21	63.65	**51.50**	387.81	172.01	120.79	105.54

	APRA*				BFPSDD				Abst.
threads	1	3	5	7	1	3	5	7	1
logistics-6	1.44	0.75	1.09	0.81	2.11	1.06	0.79	0.71	0.42
blocks-14	7.37	5.30	3.26	2.92	7.78	4.32	3.87	3.40	7.9
gripper-7	62.61	43.13	37.62	26.78	41.56	18.02	12.21	10.20	0.8
satellite-6	95.11	42.85	67.38	52.82	62.01	24.06	20.43	**13.54**	1
elevator-12	215.19	243.24	211.45	169.92	151.50	58.52	40.95	32.48	0.7
freecell-3	153.71	122.00	63.47	37.94	131.30	57.14	47.74	45.07	17
depots-7	319.48	138.30	67.24	49.58	167.24	66.89	48.32	42.68	3.6
driverlog-11	334.28	99.37	89.73	104.87	152.08	61.63	42.81	34.70	9.7
gripper-8	569.26	351.87	236.93	166.19	243.44	101.11	70.84	59.18	1.1

Table 2-1: Wall time on STRIPS planning problems.

BFPSDD was often better than APRA*,

The column, labeled "Abst." shows the time that was taken by the parallel algorithms to serially generate the abstraction function. Even with the abstraction generation time added on to the solution times all of the parallel algorithms outperform A* at seven threads, except in the block-14 domain where the time taken to generate the abstraction actually was longer than the time A* took to solve the problem.

2.4.6 Understanding Search Performance

We have seen that the PBNF algorithm tends to have better performance than the AHDA* algorithm for optimal search. In this section we show the results of a set of experiments that attempts to determine which factors allow PBNF to perform better in these domains. We considered three hypotheses. First, PBNF may achieve better performance because it expands fewer nodes with f values greater than the optimal solution cost. Second, PBNF may achieve better search performance because it tends to have many fewer nodes on each priority queue than AHDA*. Finally, PBNF may achieve better search performance because it spends less time coordinating between threads. In the following subsections we show the results of experiments that we performed to test our three hypotheses. The results of these experiments agree with the first two hypotheses, however, it appears that the third hypothesis does not hold and, in fact, PBNF occasionally spends more time coordinating between threads than AHDA*.

Node Quality

Because both PBNF and AHDA* merely approximate a best-first order, they may expand some nodes that have f values greater than the optimal solution cost. When a thread expands a node with an f value greater than the optimal solution cost its effort was a waste, because the only nodes that must be expanded when searching for an optimal solution are those with f values less than the optimal cost. In addition to this, both search algorithms may re-expand nodes for which a lower cost path has been found. If this happens work was

wasted during the first sub-optimal expansion of the node.

Threads in PBNF are able to choose which $n$block to expand based on the quality of nodes in the free $n$blocks. In AHDA*, however, a thread must expand only those nodes that are assigned to it. We hypothesized that PBNF may expand fewer nodes with f values that are greater than the optimal solution cost because the threads have more control over the quality of the nodes that they choose to expand.

We collected the f value of each node expanded by both PBNF and AHDA*. Figure 2-11 shows cumulative counts for the f values of nodes expanded by both PBNF and AHDA* on the same set of unit-cost four-way 5000x5000 grid pathfinding instances as were used in Section 2.4.3 (right) and on the 15-puzzle instances used in Section 2.4.4 (left). In both plots, the x axis shows the f value of expanded nodes as a factor of the optimal solution cost for the given instance. The y axis shows the cumulative count of nodes expanded up to the given normalized f over the set of instances. By looking at y-location of the right-most tip of each line we can find the total number of nodes expanded by each algorithm summed over all instances.

On the left panel of Figure 2-11 we can see that both algorithms tended to expand only a very small number of nodes with f values that were greater than the optimal solution cost on the grid pathfinding domain. The AHDA* algorithm expanded more nodes in total on this set of instances. Both PBNF and AHDA* must expand all of the nodes below the optimal solution cost. Because of this, the only way that AHDA* can have a greater number of expansions for nodes below a factor of 1 is if it re-expanded nodes. It appears that AHDA* re-expanded more nodes than PBNF and this seems to account for the fact that AHDA* expanded more nodes in total.

The right half of Figure 2-11 shows the results on the 15-puzzle. We see that, again, AHDA* expanded more nodes in total than PBNF. In this domain the algorithms expanded approximately the same number of nodes with f values less than the optimal solution cost. We can also see from this plot that AHDA* expanded many more nodes that had f values greater than or equal to the optimal solution cost. In summary, PBNF expanded fewer

nodes and better quality nodes than AHDA* in both the grid pathfinding and sliding tiles domains. We speculate that this may happen because in PBNF the threads are allowed to choose which portion of the space they search and they choose it based on low f value. In AHDA* the threads must search the nodes that map to them and these nodes may not be very good.

Open List Sizes

We have found that, since PBNF breaks up the search space into many different $n$blocks, it tends to have data structures with many fewer entries than AHDA*, which breaks up the search space based on the number of threads. Since we are interested general-purpose algorithms that can handle domains with real-valued costs (like eight-way grid pathfinding) both PBNF and AHDA* use binary heaps to implement their open lists. PBNF has one heap per $n$block (that is one per abstract state) whereas AHDA* has one heap per thread. Because the number of $n$blocks is greater than the number of threads AHDA* will have many more nodes than PBNF in each of its heaps. This causes the heap operations in AHDA* to take longer than the heap operations in PBNF.

The cost of operations on large heaps has been shown to greatly impact overall performance of an algorithm (Dai & Hansen, 2007). In order to determine the extent to which large heaps effect the performance of AHDA* we added timers to all of the heap operations for both algorithms. Figure 2-12 shows the mean CPU time for a single open list operation for unit-cost four-way grid pathfinding domain and for the 15-puzzle. The boxes show the second and third quartiles with a line drawn across at the median. The whiskers show the extremes of the data except that data points residing beyond the first and third quartile by more than 1.5 times the inter-quartile range are signified by a circle. The shaded rectangle shows the 95% confidence interval on the mean. We can see that, in both cases, AHDA* tended to spend more time performing heap operations than PBNF which typically spent nearly no time per heap operation. Heap operations must be performed once for each node that is expanded and may be required on each node generation. Even though these times

are in the tens of microseconds the frequency of these operations can be very high during a single search.

Finally, as is described by Hansen and Zhou (2007), the reduction in open list sizes can also explain the good single thread performance that PBNF experiences on STRIPS planning (see Table 2-1). Hansen and Zhou point out that, although A* is optimally efficient in terms of node expansions, it is not necessarily optimal with respect to wall time. They found that the benefit of managing smaller open lists enabled the Anytime weighted A* algorithm to outperform A* in wall time even though it expanded more nodes when converging to the optimal solution. As we describe in Section 2.9, this good single thread performance may also be caused by speculative expansions and pruning.

Coordination Overhead

Our third hypothesis was that the amount of time that each algorithm spent on "coordination overhead" might differ. Both parallel algorithms must spend some of their time accessing data structures shared among multiple threads. This can cause overhead in two places. The first place where coordination overhead can be seen is in the synchronization of access to shared data structures. PBNF has two modes of locking the $n$block graph. First, if a thread has ownership of an $n$block with open nodes that remain to be expanded then it will use `try_lock` because there is work that could be done if it fails to acquire the lock. Otherwise, if there are no nodes that the thread could expand then it attempt to acquire the lock on the $n$block graph using the normal operation that blocks on failure. AHDA* will use a `try_lock` on its receive queue at each expansion where it has nodes on this queue and on its open list. In our implementation AHDA* will only use the blocking lock operation when a thread has no nodes remaining to expand but has nodes remaining in its send or receive buffers.

The second place where overhead may be incurred is when threads have no nodes to expand. In PBNF this occurs when a thread exhausts its current $n$block and there are no free $n$blocks to acquire. The thread must wait until a new $n$block becomes free. In AHDA*

if no open nodes map to a thread then it may have no nodes to expand. In this situation the thread will busy-wait until a node arrives on its receive queue. In either situation, locking or waiting, there is time that is wasted because threads are not actively searching the space.

When evaluating coordination overhead, we combine the amount of time spent waiting on a lock and the amount of time waiting without any nodes to expand. Figure 2-13 shows the per-thread coordination times for locks, waiting and the sum of the two normalized to the total wall time. Unlike the previous set of boxplots, individual data points residing at the extremes are not signified by circles in order to improve readability. The "Locks" column of this plot shows the distribution of times spent by each thread waiting on a lock, the "Wait" column shows the distribution of times that threads spent waiting without any nodes available to expand and the "Sum" column shows the distribution of the sum of the mean lock and wait times.

The left side of Figure 2-13 shows the results for grid pathfinding. From the "Locks" column we see that threads in AHDA* spent almost no time acquiring locks. This is expected because AHDA* uses asynchronous communication. It appears that the amount of time that threads in PBNF spent acquiring locks was significantly greater than that of AHDA*. The "Wait" column of this plot shows that both PBNF and AHDA* appeared to have threads spend nearly the same amount of time waiting without any nodes to expand. Finally, the "Sum" column shows that the threads in PBNF spent more time overall coordinating between threads.

The bottom half of Figure 2-13 shows the coordination overhead for the 15-puzzle domain. Again, we see that threads in AHDA* spent almost no time acquiring a lock. Individual threads in PBNF, however, tended to spend a larger fraction of their time waiting on locks in the sliding tiles domain than in grid pathfinding. In the "Wait" column of this figure we can see that AHDA* spent more time than PBNF without any nodes to expand. Finally, we see that, over all, PBNF spent more time coordinating between threads than AHDA*.

Overall our experiments have verified that our first two hypotheses that PBNF expanded

better quality nodes than AHDA* and that it spent less time performing priority queue operations than AHDA*. We also found that our third hypothesis did not hold and that threads in PBNF tended to have more coordination overhead that AHDA* but this seems to be out-weighed by the other two factors.

2.4.7 Summary

In this section we have shown the results of an empirical evaluation of optimal parallel best-first search algorithms. We have shown that several simple parallel algorithms can actually be slower than a serial A* search even when offered more computing power. Additionally we showed empirical results for a set of algorithms that make good use of parallelism and do outperform serial A*. Overall the Safe PBNF algorithm gave the best and most consistent performance of this latter set of algorithms. Our AHDA* variant of PRA* had the second fastest mean performance in all domains.

We have also shown that using abstraction in a PRA* style search to distribute nodes among the different threads can give a significant boost in speed by reducing the amount of communication. This modification to PRA* appears to be a lot more helpful than simply using asynchronous communication. Using both of these improvements in conjunction (AHDA*), yields a competitive algorithm that has the additional feature of not relying on shared memory.

Finally, we performed a set of experiments in an attempt to explain why Safe PBNF tended to give better search performance than AHDA*. Our experiments looked at three factors: node quality, open list sizes and thread-coordination overhead. We concluded that PBNF is faster because it expands fewer nodes with suboptimal f values and it takes less time to perform priority queue operations.

2.5 Bounded Suboptimal Search

Sometimes it is acceptable or even preferable to search for a solution that is not optimal. Suboptimal solutions can often be found much more quickly and with lower memory require-

ments than optimal solutions. In this section we show how to create bounded-suboptimal variants of some of the best optimal parallel search algorithms.

Weighted A* (Pohl, 1970), a variant of A* that orders its search on $f'(n) = g(n) + w \cdot h(n)$, with $w > 1$, is probably the most popular suboptimal search. It guarantees that, for an admissible heuristic h and a weight w, the solution returned will be w-admissible (within a w factor of the optimal solution cost, Davis, Bramanti-Gregor, & Wang, 1988).

It is possible to modify AHDA*, BFPSDD, and PBNF to use weights to find suboptimal solutions, we call these algorithms wAHDA*, wBFPSDD and wPBNF. Just as in optimal search, parallelism implies that a strict f' search order will not be followed. The proof of weighted A*'s w-optimality depends crucially on following a strict f' order, and for our parallel variants we must prove the quality of our solution by either exploring or pruning all nodes. Thus finding effective pruning rules can be important for performance. We will assume throughout that h is admissible.

2.5.1 Pruning Poor Nodes

Let s be the current incumbent solution and w be the suboptimality bound. A node n can clearly be pruned if $f(n) \geq g(s)$. But according to the following theorem, we only need to retain n if it is on the optimal path to a solution that is a factor of w better than s. This is a much stronger rule.

Theorem 4 *We can prune a node n if $w \cdot f(n) \geq g(s)$ without sacrificing w-admissibility.*

Proof: If the incumbent is w-admissible, we can safely prune any node, so we consider the case where $g(s) > w \cdot g(opt)$, where *opt* is an optimal goal. Note that without pruning, there always exists a node p in some open list (or being generated) that is on the best path to *opt*. Let f^* be the cost of an optimal solution. By the admissibility of h and the definition of p, $w \cdot f(p) \leq w \cdot f^*(p) = w \cdot g(opt)$. If the pruning rule discards p, that would imply $g(s) \leq w \cdot f(p)$ and thus $g(s) \leq w \cdot g(opt)$, which contradicts our premise. Therefore, an open node leading to an optimal solution will not be pruned if the incumbent is not

w-admissible. A search that does not terminate until open is empty will not terminate until the incumbent is w-admissible or it is replaced by an optimal solution. □

We make explicit a useful corollary:

Corollary 2 *We can prune a node n if $f'(n) \geq g(s)$ without sacrificing w-admissibility.*

Proof: Clearly $w \cdot f(n) \geq f'(n)$, so Theorem 4 applies. □

With this corollary, we can use a pruning shortcut: when the open list is sorted on increasing f' and the node at the front has $f' \geq g(s)$, we can prune the entire open list.

2.5.2 Pruning Duplicate Nodes

When searching with an inconsistent heuristic, as in weighted A*, it is possible for the search to find a better path to an already-expanded state. Likhachev, Gordon, and Thrun (2003b) noted that, provided that the underlying heuristic function h is consistent, weighted A* will still return a w-admissible solution if these duplicate states are pruned during search. This ensures that each state is expanded at most once during the search. Unfortunately, their proof depends on expanding in exactly best-first order, which is violated by several of the parallel search algorithms we consider here. However, we can still prove that some duplicates can be dropped. Consider the expansion of a node n that re-generates a duplicate state d that has already been expanded. We propose the following weak duplicate dropping criterion: the new copy of d can be pruned if the old $g(d) \leq g(n) + w \cdot c^*(n, d)$, where $c^*(n, d)$ is the optimal cost from node n to node d.

Theorem 5 *Even if the weak dropping rule is applied, there will always be a node p from an optimal solution path on* open *such that $g(p) \leq w \cdot g^*(p)$.*

Proof: We proceed by induction over iterations of search. The theorem clearly holds after expansion of the initial state. For the induction step, we note that node p is only removed from *open* when it is expanded. If its child p_i that lies along the optimal path is added to *open*, the theorem holds. The only way it won't be added is if there exists a previous duplicate copy p'_i and the pruning rule holds, i.e., $g(p'_i) \leq g(p_{i-1}) + w \cdot c^*(p_{i-1}, p_i)$. By

the inductive hypothesis, $g(p_{i-1}) \leq w \cdot g^*(p_{i-1})$, and by definition $g^*(p_{i-1}) + c^*(p_{i-1}, p_i) = g^*(p_i)$, so we have $g(p'_i) \leq w \cdot g^*(p'_i)$. □

Note that the use of this technique prohibits using the global minimum f value as a lower bound on the optimal solution's cost, because g values can now be inflated by up to a factor of w. However, if s is the incumbent and we search until the global minimum f' value is $\geq g(s)$, as in a serial weighted A* search, then w-admissibility is assured:

Corollary 3 *If the minimum f' value is $\geq g(s)$, where s is the incumbent, then we have $g(s) \leq w \cdot g^*(\text{opt})$*

Proof: Recall node p from Theorem 5. $g(s) \leq f'(p) = g(p) + w \cdot h(p) \leq w \cdot (g^*(p) + h(p)) \leq w \cdot g^*(opt)$. □

It remains an empirical question whether pruning on this rather weak criterion will lead to better performance in practice. Our results indicate that it does provide an advantage in the grid pathfinding domain. Results are presented in Section 2.6.1. It should be noted that, while extra pruning can preserve w-admissibility, it may result in solutions of lower quality than those resulting from search without pruning.

2.5.3 Optimistic Search

Korf (1993) showed that weighted A* typically returns solutions that are better than the bound, w, would suggest. To take advantage of this, Thayer and Ruml (2008) use an optimistic approach to bounded suboptimal search that works in two stages: aggressive search using a weight that is greater than the desired optimality bound to find an incumbent solution and then a cleanup phase to prove that the incumbent is indeed within the bound. The intuition behind this approach is that wA* can find a solution within a very tight bound (much tighter than $w \cdot g(opt)$), then the search can continue looking at nodes in f order until the bound can be proved. Thayer and Ruml show that, indeed, this approach can surpass the speed of wA* for a given optimality bound. We have implemented an optimistic version of PBNF (oPBNF).

One of the requirements of oPBNF is that it must have access to the minimum f value

over all nodes in order to prove the bound on the incumbent solution. For the aggressive search stage, the open lists and the heap of free $n$blocks are sorted on f' instead of f so a couple of additions need to be made. First, each $n$block has an additional priority queue containing the open search nodes sorted on f. We call this queue $open_f$. The open_f queue is simply maintained by adding and removing nodes as nodes are added and removed from the f' ordered open list of each $n$block. Second, a priority queue, called $\min_f$, of all of the $n$blocks is maintained, sorted on the lowest f value in each $n$block at the time of its last release. $\min_f$ is used to track a lower bound on the minimum f value over all nodes. This is accomplished by lazily updating $\min_f$ only when an $n$block is released by a thread. When a thread releases an $n$block, it sifts the released $n$block and its successors to their new positions in the $\min_f$ queue. These are the only $n$blocks whose minimum f values could have been changed by the releasing thread. Since the global minimum f value over all nodes is strictly increasing (assuming a consistent heuristic) we have the guarantee that the f value at the front of the $\min_f$ queue is strictly increasing and is a lower bound on the global minimum f value at any given time. Using this lower bound, we are able to prove whether or not an incumbent solution is properly bounded.

oPBNF needs to decide when to switch between the aggressive search phase and the cleanup phase of optimistic search. As originally proposed, optimistic search performs aggressive search until the first incumbent is found then it switches between cleanup (when $f'(n) \geq g(s)$, where n is the best node based on f' and s is the incumbent solution) and aggressive search (when $f'(n) < g(s)$) to hedge against the case when the current incumbent is not within the bound. In oPBNF, we were left with a choice: switch between aggressive search and cleanup on a global basis or on a per-$n$block basis. We choose to switch on a per-$n$block basis under the assumption that some threads could be cleaning up areas of the search space with low f values while other threads look for better solutions in areas of the search space with low f' values. In oPBNF, when deciding if one $n$block is better than another (when deciding to switch or to set an $n$block to hot), the choice is no longer based solely on the best f' value of the given $n$block, but instead it is based on the f' value first,

then the f value to break ties of if the best f' value is out of the bound of the incumbent. When acquiring a new $n$block, a thread takes either the free $n$block with the best f' value or best f value depending on which $n$block is better (where the notion of better is described in the previous sentence). Finally, when expanding nodes, a thread selects aggressive search or cleanup based on the same criteria as standard optimistic search for the nodes within the acquired $n$block.

2.6 Empirical Evaluation: Bounded Suboptimal Search

We implemented and tested weighted versions of the parallel search algorithms discussed above: wAHDA*, wAPRA*, wBFPSDD, wPBNF and oPBNF. All algorithms prune nodes based on the $w \cdot f$ criterion presented in Theorem 4 and prune entire open lists on f' as in Corollary 2. Search terminates when all nodes have been pruned by the incumbent solution. Our experiments were run on the same three benchmark domains as for optimal search: grid pathfinding, the sliding tile puzzle, and STRIPS planning.

2.6.1 Grid Pathfinding

Results presented in Table 2-2 show the performance of the parallel search algorithms in terms of speedup over serial weighted A* on grid pathfinding problems. Duplicate states that have already been expanded are dropped in the serial wA* algorithm, as discussed by Likhachev et al. (2003b).

The rows of this table show the number of threads and different algorithms whereas the columns are the weights used for various domains. Each entry shows the mean speedup over serial weighted A*. We performed a Wilcoxon signed-rank test to determine which mean values were significantly different; elements that are in **bold** represent values that were not significantly different ($p < 0.05$) from the best mean value in the given column. In general, the parallel algorithms show increased speedup as threads are added for low weights, and decreased speedup as the weight is increased.

In unit-cost four-way movement grids, for weights of 1.1, and 1.2 the wPBNF algorithm

	threads	weight: Unit Four-way Grids 1.1	1.2	1.4	1.8	Unit Eight-way Grids 1.1	1.2	1.4	1.8	Life Four-way Grids 1.1	1.2	1.4	1.8
wPBNF	1	0.98	0.91	0.51	0.73	0.93	1.37	**0.73**	0.74	0.65	0.66	0.84	0.67
	2	1.74	1.65	1.07	0.87	1.65	1.82	0.57	0.66	1.15	1.17	1.59	0.39
	3	2.47	2.33	1.62	0.89	2.36	1.77	0.55	0.61	1.65	1.67	2.32	0.39
	4	3.12	2.92	2.13	0.90	2.97	1.72	0.53	0.58	2.08	2.10	2.96	0.49
	5	3.76	3.52	2.48	0.91	3.55	1.67	0.52	0.56	2.53	2.55	3.63	1.49
	6	4.30	3.99	2.80	0.89	4.04	1.61	0.50	0.54	2.94	2.95	4.20	1.64
	7	4.78	4.40	**3.01**	0.88	4.40	1.55	0.49	0.51	3.31	3.33	4.63	2.12
	8	**5.09**	**4.66**	**3.11**	0.87	**4.70**	1.49	0.45	0.46	**3.61**	**3.64**	**5.11**	1.06
wBFPSDD	1	0.82	0.84	0.96	0.94	0.87	0.79	0.43	0.33	0.52	0.53	0.58	0.60
	2	1.26	1.26	1.45	0.91	1.37	1.10	0.43	0.35	0.83	0.83	0.92	0.76
	3	1.65	1.65	1.90	0.84	1.80	1.22	0.41	0.33	1.10	1.09	1.26	0.84
	4	1.93	1.92	2.09	0.79	2.13	1.25	0.42	0.33	1.29	1.29	1.48	0.89
	5	2.24	2.24	2.36	0.75	2.47	1.31	0.39	0.32	1.53	1.51	1.61	**0.93**
	6	2.51	2.51	**2.58**	0.71	2.74	1.21	0.36	0.30	1.73	1.72	1.78	0.93
	7	2.73	2.69	2.63	0.67	2.94	1.26	0.34	0.29	1.91	1.89	1.94	0.91
	8	2.91	2.84	**2.68**	0.63	3.10	1.23	0.32	0.26	2.06	2.03	2.10	0.85
wAHDA*	1	0.87	0.79	0.32	0.56	0.79	1.10	0.66	0.76	0.56	0.55	0.71	0.22
	2	1.35	1.17	0.63	0.84	1.04	1.99	0.62	0.61	0.88	0.86	1.29	0.32
	3	1.90	1.69	1.30	**1.30**	2.08	**2.93**	0.64	0.62	1.09	1.39	1.86	0.56
	4	2.04	2.10	1.57	**1.30**	2.48	**2.84**	0.56	0.54	1.60	1.64	2.24	0.56
	5	1.77	2.08	1.79	0.97	2.49	2.52	0.42	0.41	1.88	1.92	2.58	0.41
	6	3.23	3.03	2.18	**1.33**	3.73	**2.83**	0.49	0.45	2.15	2.17	3.02	**1.50**
	7	3.91	3.78	2.56	**1.30**	4.45	**2.89**	0.45	0.41	2.39	2.41	3.50	**1.07**
	8	3.79	3.64	**3.02**	1.13	4.39	2.58	0.37	0.38	2.38	2.42	3.55	**4.16**
wAPRA*	1	0.88	0.81	0.32	0.56	0.80	1.11	0.67	**0.77**	0.56	0.56	0.72	0.23
	2	0.51	0.44	0.22	0.36	0.35	0.69	0.31	0.28	0.35	0.34	0.46	0.12
	3	0.36	0.32	0.20	0.26	0.41	0.65	0.23	0.22	0.23	0.26	0.32	0.10
	4	0.50	0.44	0.30	0.41	0.43	0.73	0.22	0.19	0.42	0.43	0.55	0.16
	5	0.55	0.56	0.39	0.48	0.49	0.87	0.23	0.19	0.54	0.56	0.67	0.20
	6	0.52	0.49	0.31	0.30	0.50	0.65	0.16	0.14	0.39	0.39	0.49	0.13
	7	0.73	0.67	0.40	0.36	0.62	0.73	0.17	0.14	0.49	0.49	0.65	0.18
	8	1.09	1.07	0.82	0.77	0.89	1.38	0.28	0.22	1.00	0.98	1.22	0.42

Table 2-2: Grid Pathfinding: Average speedup over serial weighted A* for various numbers of threads.

threads	wPBNF				wBFPSDD			
	1.4	1.7	2.0	3.0	1.4	1.7	2.0	3.0
1	0.68	0.44	0.38	0.69	0.65	0.61	0.44	0.35
2	1.35	0.81	1.00	0.63	0.87	0.74	0.49	0.43
3	1.48	0.97	0.85	0.56	1.05	0.72	0.63	0.46
4	1.70	1.20	0.93	0.60	1.09	1.00	0.57	0.45
5	2.04	1.38	0.97	0.74	1.27	0.97	0.65	0.40
6	2.16	1.30	**1.19**	0.67	1.33	1.17	0.61	0.39
7	**2.55**	**1.46**	1.04	0.62	1.49	1.10	0.59	0.34
8	**2.71**	**1.71**	1.10	0.60	1.53	1.08	0.62	0.33

threads	wAHDA*				wAPRA*			
	1.4	1.7	2.0	3.0	1.4	1.7	2.0	3.0
1	0.61	0.60	0.59	0.54	0.61	0.59	0.59	0.54
2	1.18	1.11	1.32	**0.78**	1.18	1.08	1.36	0.78
3	1.53	**1.30**	1.40	0.73	1.45	1.25	1.32	0.78
4	1.91	**1.57**	1.55	0.74	1.77	**1.50**	1.36	0.62
5	**2.33**	1.70	1.27	0.66	2.32	1.62	1.26	0.64
6	**2.28**	**1.72**	1.24	0.52	2.18	1.54	**1.83**	0.47
7	2.71	1.50	1.03	0.44	2.63	1.40	1.09	0.43
8	**2.70**	**1.51**	1.24	0.44	**2.34**	**1.61**	1.22	0.41

Table 2-3: 15-puzzle: Average speedup over serial weighted A* for various numbers of threads.

	Unit Four-way Grids				Unit Eight-way Grids				250 easy 15-puzzles			
threads	1.1	1.2	1.4	1.8	1.1	1.2	1.4	1.8	1.4	1.7	2.0	3.0
1	0.54	0.99	0.74	**0.47**	0.74	**0.76**	**0.09**	**0.05**	0.56	0.58	0.77	0.60
2	0.99	2.00	1.05	0.45	1.26	0.71	0.09	0.05	0.85	**1.07**	**0.83**	**0.72**
3	1.40	2.89	1.19	0.45	1.64	0.70	0.09	0.05	1.06	**0.94**	**0.79**	**0.80**
4	1.76	3.62	1.26	0.44	1.90	0.69	0.09	0.05	1.01	0.82	**0.93**	**0.69**
5	2.11	4.29	1.33	0.43	2.09	0.68	0.08	0.05	**1.20**	**1.21**	**0.97**	**0.74**
6	2.43	4.84	**1.35**	0.44	2.21	0.68	0.08	0.05	**1.32**	**0.83**	**0.99**	0.67
7	2.70	5.44	**1.37**	0.43	2.29	0.67	0.08	0.04	**1.14**	**0.93**	**0.88**	0.71
8	**2.97**	**6.01**	**1.39**	0.42	**2.30**	0.67	0.08	0.04	**1.33**	0.87	0.81	0.64

oPBNF

Table 2-4: Average speedup over serial optimistic search for various numbers of threads.

was the fastest of all of the algorithms tested reaching over five times the speed of wA* at a weight of 1.1 at and over 4.5x at a weight of 1.2 . At a weight of 1.4 wPBNF, wBFPSDD and wAHDA* did not show a significant difference in performance at 8 threads. wAHDA* had the best speed up of all algorithms at a weight of 1.8. wAPRA* never gave the best performance in this domain.

In eight-way movement grids wPBNF gave the best performance for a weight of 1.1 and 1.4, although in the latter case this best performance was a decrease over the speed of wA* and it was achieved at 1 thread. wAHDA* was the fastest when the weight was 1.2, however, this did not scale as expected when the number of threads was increased. Finally wAPRA* gave the least performance decrease over weighted A* at a weight of 1.8 with 1 thread. In this case, all algorithms were slower than serial weighted A* but wAPRA* gave the closest performance to the serial search. wBFPSDD never gave the best performance in this domain.

In the life-cost domain wPBNF outperformed all other algorithms for weights 1.1, 1.2 and 1.4. At weight 1.8, wPBNF's performance quickly dropped, however and wAHDA* had the best results with more than a 4x speedup over wA*, although the performance appears to have been very inconsistent as it is not significantly different from much lower speedup values for the same weight. wAPRA* never gave the best performance in this domain.

Overall, we see that wPBNF often had the best speedup results at eight threads and for weights less than 1.8. wAHDA*, however, gave the best performance at a weight of 1.8 across all grid pathfinding domains. wBFPSDD often gave speedup over serial weighted A*, however it was not quite as competitive as wPBNF or wAHDA*. wAPRA* was only very rarely able to outperform the serial search.

Table 2-4 shows the results for the optimistic variant of the PBNF algorithm (oPBNF). Each cell in this table shows the mean speedup of oPBNF over serial optimistic search. Once again, the **bold** cells entries that are not significantly different from the best value in the column. For unit-cost four-way pathfinding problems oPBNF gave a performance increase over optimistic search for two or more threads and for all weights less than 1.8. At

a weight of 1.2, oPBNF tended to give the best speedup, this may be because optimistic search performed poorly at this particular weight. In unit-cost eight-way pathfinding, we see that oPBNF performs comparably to the unit-cost domain for a weight of 1.1, however, at all higher weights the algorithm is slower than serial optimistic search.

2.6.2 Sliding Tile Puzzles

For the sliding tiles domain, we used the standard Korf 100 15-puzzles (Korf, 1985). Results are presented in Table 2-3. wPBNF, wAHDA* and wAPRA* tended to give comparable performance in the sliding tile puzzle domain each having values that are not significantly different for weights of 1.4 and 1.7. At a weight of 3.0, wAHDA* gave the least performance decrease over weighted A* at 2 threads.

The right-most column of Table 2-4 shows the results for optimistic PBNF on 250 15-puzzle instances that were solvable by A* in fewer than 3 million expansions. oPBNF gave its best performance at a weight of 1.4. For weights greater than 1.4 oPBNF was unable to outperform its serial counterpart. For greater weights oPBNF tended to perform better with smaller numbers of threads.

One trend that can be seen in both the sliding tiles domain and the grid pathfinding domain is that the speedup of the parallel algorithms over serial suboptimal search decreases as the weight is increased. We suspect that the decrease in relative performance is due to the problems becoming sufficiently easy (in terms of node expansions) that the overhead for parallelism becomes harmful to overall search. In problems that require many node expansions the cost of parallelism (additional expansions, spawning threads, synchronization – albeit small, waiting for threads to complete, etc.) is amortized by the search effort. In problems that require only a small number of expansions, however, this overhead accounts for more of the total search time and a serial algorithm could potentially be faster.

To confirm our understanding of the effect of problem size on speedup, Figure 2-14 shows a comparison of wPBNF to weighted A* on all of the 100 Korf 15-puzzle instances using eight threads. Each point represents a run on one instance at a particular weight, the y-axis

represents wPBNF speedup relative to serial wA*, and the x-axis represents the number of nodes expanded by wA*. Different glyphs represents different weight values used for both wPBNF and wA*. The figure shows that, while wPBNF did not outperform wA* on easier problems, the benefits of wPBNF over wA* increased as problem difficulty increased. The speed gain for the instances that were run at a weight of 1.4 (the lowest weight tested) leveled off just under 10 times faster than wA*. This is because the machine has eight cores. There are a few instances that seem to have speedup greater than 10x. These can be explained by the speculative expansions that wPBNF performs which may find a bounded solution faster than weighted A* due to the pruning of more nodes with f' values equal to that of the resulting solution. The poor behavior of wPBNF for easy problems is most likely due to the overhead described above. This effect of problem difficulty means that wPBNF outperformed wA* more often at low weights, where the problems required more expansions, and less often at higher weights, where the problems were completed more quickly.

2.6.3 STRIPS Planning

Table 2-5 shows the performance of the parallel search algorithms on STRIPS planning problems, again in terms of speedup versus serial weighted A*. In this table columns represent various weights and the rows represent different planning problems with two and seven threads. **Bold** values represent table entries that are within 10% of the the best performance for the given domain. All algorithms had better speedup at seven threads than at two. wPBNF gave the best speedup for the most number of domains followed by wAHDA* which was the fastest for three of the domains at seven threads. At two threads there were a couple of domains (satellite-6 and freecell-3) where wBFPSDD gave the most speedup, however it never did at seven threads. wAPRA* was always slower than the three remaining algorithms. On one problem, freecell-3, serial weighted A* performs much worse as the weight increases. Interestingly, wPBNF and wBFPSDD do not show this pathology, and thus record speedups of up to 1,700 times.

		wAPRA*				wAHDA*			
		1.5	2	3	5	1.5	2	3	5
2 threads	logistics-8	0.99	1.02	0.59	1.37	1.25	1.11	0.80	1.51
	blocks-16	1.29	0.88	4.12	0.30	1.52	1.09	**4.86**	0.38
	gripper-7	0.76	0.76	0.77	0.77	1.36	1.35	1.33	1.30
	satellite-6	0.68	0.93	0.70	0.75	1.15	1.09	1.28	1.44
	elevator-12	0.65	0.72	0.71	0.77	1.16	1.20	1.27	1.22
	freecell-3	1.03	1.00	1.78	1.61	1.49	1.20	7.56	1.40
	depots-13	0.73	1.25	0.97	1.08	0.92	1.29	0.96	1.09
	driverlog-11	0.91	0.79	0.94	0.93	**1.30**	0.97	0.96	0.93
	gripper-8	0.63	0.61	0.62	0.62	1.14	1.16	1.15	1.16
7 threads	logistics-8	3.19	3.10	3.26	2.58	4.59	4.60	3.61	2.58
	blocks-16	3.04	1.37	1.08	0.37	**3.60**	1.62	0.56	0.32
	gripper-7	1.71	1.74	1.73	1.82	3.71	3.66	3.74	3.83
	satellite-6	1.11	1.01	1.29	1.44	3.22	3.57	3.05	3.60
	elevator-12	0.94	0.97	1.04	1.02	2.77	2.88	2.98	3.03
	freecell-3	3.09	7.99	2.67	2.93	4.77	2.71	48.66	4.77
	depots-13	2.38	5.36	1.13	1.17	2.98	**6.09**	1.22	1.17
	driverlog-11	1.90	1.25	0.93	0.92	**3.52**	1.48	0.95	0.92
	gripper-8	1.70	1.68	1.68	1.74	3.71	3.63	3.67	4.00

		wPBNF				wBFPSDD			
		1.5	2	3	5	1.5	2	3	5
2 threads	logistics-8	2.68	2.27	**4.06**	1.00	1.86	2.12	1.14	0.15
	blocks-16	0.93	0.54	0.48	1.32	0.34	0.19	0.16	0.32
	gripper-7	**2.01**	**1.99**	**1.99**	**2.02**	**1.91**	**1.89**	**1.86**	**1.84**
	satellite-6	2.02	1.53	5.90	3.04	1.71	2.22	**7.50**	2.80
	elevator-12	**2.02**	**2.08**	**2.21**	**2.15**	1.76	1.76	1.81	**2.18**
	freecell-3	2.06	0.84	8.11	10.69	1.42	0.54	16.88	**55.75**
	depots-13	2.70	**4.49**	0.82	0.81	1.48	1.58	0.18	0.14
	driverlog-11	0.85	0.19	0.69	0.62	0.85	0.11	0.19	0.21
	gripper-8	**2.06**	**2.04**	**2.08**	**2.07**	**2.00**	**1.96**	**1.97**	**1.98**
7 threads	logistics-8	**7.10**	**6.88**	1.91	0.46	3.17	3.59	0.62	0.10
	blocks-16	2.87	0.70	0.37	1.26	0.49	0.22	0.11	0.32
	gripper-7	**5.67**	5.09	5.07	**5.18**	4.33	4.28	4.14	4.05
	satellite-6	4.42	2.85	2.68	**5.89**	3.13	2.31	3.01	1.05
	elevator-12	6.32	6.31	**6.60**	**7.10**	3.68	3.78	4.04	3.95
	freecell-3	7.01	2.31	131.12	**1,721.33**	2.12	0.70	44.49	137.19
	depots-13	3.12	1.80	0.87	0.88	1.88	1.87	0.15	0.12
	driverlog-11	1.72	0.43	0.67	0.42	1.26	0.21	0.30	0.23
	gripper-8	**5.85**	5.31	**5.40**	**5.44**	4.62	4.55	4.55	4.51

Table 2-5: Speed-up over serial weighted A* on STRIPS planning problems for various weights.

2.6.4 Summary

In this section, we have seen that bounded suboptimal variants of the parallel searches can give better performance than their serial progenitors. We have also shown that, on the sliding tile puzzle, parallel search gives more of an advantage over serial search as problem difficulty increases and we suspect that this result holds for other domains too. We suspect that this is because the overhead of using parallelism is not amortized by search time for very easy problems.

2.7 Anytime Search

A popular alternative to bounded suboptimal search is anytime search, in which a highly suboptimal solution is returned quickly and then improved solutions are returned over time until the algorithm is terminated (or the incumbent solution is proved to be optimal). The two most popular anytime heuristic search algorithms are Anytime weighted A* (AwA*) (Hansen & Zhou, 2007) and anytime repairing A* (ARA*) (Likhachev, Gordon, & Thrun, 2003a). In AwA* a weighted A* search is allowed to continue after finding its first solution, pruning when the unweighted $f(n) \geq g(s)$ where s is an incumbent solution and n is a node being considered for expansion. ARA* uses a weighted search where the weight is lowered when a solution meeting the current suboptimality bound has been found and a special *INCONS* list is kept that allows the search to expand a node at most once during the search at each weight.

In this section we present anytime versions of the best performing parallel searches from our previous sections. We used the PBNF framework to implement Anytime weighted PBNF (AwPBNF) and Anytime Repairing PBNF (ARPBNF). We use the PRA* framework to create anytime weighted AHDA* (AwAHDA*). We also show the performance of a very simple algorithm that runs parallel weighted A* searches with differing weights. In the planning domain, we have implemented anytime weighted BFPSDD (AwBFPSDD) for comparison as well.

Because our parallel searches inherently continue searching after their first solutions are found, they serve very naturally as anytime algorithms in the style of Anytime weighted A*. The main difference between the standard, optimal versions of these algorithms and their anytime variants is that the anytime versions sort all open lists and the heap of free $n$blocks on $f'(n) = g(n) + w \cdot h(n)$. In fact, in both cases the optimal search is a degenerate case of the anytime search where $w = 1$. This approach (simply using $w > 1$) is used to implement all algorithms except for ARPBNF and multi-weighted A*.

Next, we will discuss the details of the ARPBNF algorithm. Following that, we introduce a new parallel anytime algorithm called multi-weighted A*. Finally, we show the results of a set of comparisons that we performed on the anytime algorithms discussed in these sections.

2.7.1 Anytime Repairing PBNF

ARPBNF is a parallel anytime search algorithm based on ARA* (Likhachev et al., 2003a). In ARPBNF, open lists and the heap of $n$blocks are sorted on f' as in AwPBNF, but instead of merely continuing the search until the incumbent is proved optimal, ARPBNF uses a weight schedule. Each time an incumbent is found, the weight on the heuristic value is lowered by a specified amount, all open lists are resorted and the search continues. On the final iteration, the weight will be 1.0 and the optimal solution will be found.

The following procedure is used to resort the $n$blocks in parallel between incumbent solutions:

1. The thread calling for a resort (the one that found a goal) becomes the leader by taking the lock on the $n$block graph and setting the *resort flag*. (If the flag has already been set, then another thread is already the leader and the current thread becomes a worker). After the flag is set the leader thread releases the lock on the $n$block graph and waits for all $n$blocks to have σ values of zero (no $n$blocks are acquired).

2. Threads check the *resort flag* each expansion, if it is set then threads release their $n$blocks and become worker threads and wait for the leader to set the *start flag*.

3. Once all $n$blocks have $\sigma = 0$, the leader re-takes the lock on the $n$block graph and ensures that all σ values are still zero (if not, then it releases the lock and retries). The leader sets the global weight value to the next weight on the weight schedule and populates a lock-free queue with all $n$blocks. Once the queue has been populated, the leader sets the *start flag*.

4. All threads greedily dequeue $n$blocks and resort them until the queue is empty.

5. When all $n$blocks have been resorted, the leader thread clears the *resort flag* and the *start flag* and releases the lock on the $n$block graph. All threads will now acquire new $n$blocks and the search will continue.

We modeled this procedure in TLA$^+$ and showed it to be live-lock and dead-lock free for up to 4 threads and 5 nblocks by the use of the TLC model checker (Yu et al., 1999). This model is very simple so we do not include it in an appendix.

2.7.2 Multi-weighted A*

In this section we introduce a new and simple parallel anytime algorithm called multi-weighted A*. The PBNF and PRA* frameworks for parallelizing anytime algorithms can be thought of as one end on a spectrum of parallel anytime algorithms. In PBNF and PRA* all threads are working on finding a single solution of a given quality; on the opposite end of the spectrum each thread would be working to find its own solution. To compare to an algorithm at that end of the spectrum we implemented an algorithm we call multi-weighted A* that allocates its available threads to their own weighted A* searches. The thread that finishes first will generally be the thread that was searching at the greatest weight and therefore the solution will be of the worst quality. The next thread to finish will have the next greatest weight, and so on. The final thread to complete will generally be searching at a weight of 1.0, performing a standard A* search, and will return the optimal solution.

The algorithm is given a schedule of weighs in decreasing order. The largest weights in the schedule are distributed among the available threads. The threads begin searching

using wA* with their given weight values. When a thread finds a new solution that is better than the current one, it updates the incumbent that is shared between all threads to allow for pruning. When a thread finds a better incumbent solution, it will be w-admissible with respect to the weight the thread was searching with. If a thread finishes (either finding a solution or pruning its entire open list), it takes the highest unclaimed weight from the schedule and starts a fresh search using that weight. If there are no weights left in the schedule, the thread terminates. When all threads have terminated, the search is complete. If the final weight in the schedule is 1.0, then the last solution found will be optimal.

One of the benefits of multi-weighted A* is that it is a very simple algorithm to implement. However, as we will see below, it doesn't benefit much from added parallelism. A reason for this may be because, when the weight schedule is exhausted (a thread is searching with the lowest weight, 1.0) threads that complete their searches will sit idle until the entire search terminates. Since the final weight will take the longest, this may be a majority of the search time. A more dynamic schedule could be used to keep threads busy until the optimal solution is found. One could also attempt to use more threads at once by using some multi-threaded search at each weight, such as wPBNF or wAHDA*. We leave these extensions for future work.

2.8 Empirical Evaluation: Anytime Search

The implementation and empirical setup was similar to that used for suboptimal search. For ARA*, ARPBNF and Multi-wA* we considered four different weight schedules: {7.4, 4.2, 2.6, 1.9, 1.5, 1.3, 1.1, 1}, {4.2, 2.6, 1.9, 1.5, 1.3, 1.1, 1.05, 1}, {3, 2.8, ..., 1.2, 1}, {5, 4.8, ..., 1.2, 1}. For AwA* and the other anytime parallel algorithms we consider weights of: 1.1, 1.2, 1.4, 1.8 and 3.4 for grid pathfinding and 1.4, 1.7, 2.0, 3.0 and 5.0 for the sliding tiles domain. To fully evaluate anytime algorithms, it is necessary to consider their performance profile, i.e., the expected solution quality as a function of time. While this can be easily plotted, it ignores the fact that the anytime algorithms considered in this chapter all have a free parameter, namely the weight or schedule of weights used to accelerate the

search. In order to compare algorithms, we make the assumption that, in any particular application, the user will attempt to find the parameter setting giving good performance for the time scale they are interested in. Under this assumption, we can plot the performance of each anytime algorithm by computing, at each time point, the best performance that was achieved by any of the parameter settings tried for that algorithm – that is minimum solution cost over all parameter settings for a given algorithm up to the given time point. We refer to this concept as the *lower hull* of the profiles, because it takes the minimum over the profiles for each parameter setting.

The top row of Figure 2-15 shows an example of the raw data for three algorithms on our 5000x5000 unit-cost four-way grid pathfinding problems. The y-axis of these plots is the solution quality as a factor of optimal and the x-axis is the wall clock time relative to the amount of time A* took to find an optimal solution. The bottom row of this figure shows the lower hull for the respective data displayed above. By comparing the two images on the left that display the data for the AwA* algorithm, one can see that the three big "steps" in the lower hull plot is where a different weight is used in the hull because it has found a better solution for the same time bound. The center panel in Figure 2-15 shows that the AwPBNF algorithm gives a similar performance to AwA*, however it is often faster. This is not surprising since AwPBNF is based on the AwA* approach and it is running at eight threads instead of one. The final panel in Figure 2-15 shows ARA*, which uses weight schedules instead of a single weight.

Figures 2-16 and 2-17 present the lower hulls of both serial and parallel algorithms on grid pathfinding and the sliding tile puzzle. In each panel, the y-axis represents solution cost as a factor of the optimal cost. In Figure 2-16 the x-axis represents wall time relative to the amount of time that serial A* took to find an optimal solution. This allows for a comparison between the anytime algorithms and standard serial A*. Since A* is not able to solve all of Korf's 100 15-puzzle instances on this machine, the x-axis in Figure 2-17 is the absolute wall time in seconds. Both serial and parallel algorithms are plotted. The profiles start when the algorithm first returns a solution and ends when the algorithm has

proved optimality or after a 180 second cutoff (since Multi-wA* can consume memory more quickly than the other algorithms, we gave it a 120 second cutoff on the sliding tile puzzle to prevent thrashing).

2.8.1 Four-Way Unit Cost Grids

Figure 2-16 shows the anytime performance for unit cost four-way movement grid pathfinding problems. AwAHDA* and AwPBNF found the best solutions quicker than the other algorithms. Both of these algorithms improved in the amount of time taken to find better solutions as more threads were added. AwPBNF converged more quickly as more threads were added. Even at two threads AwPBNF was the first algorithm to converge on the optimal solution in 60% of the time of serial A*. The next two algorithms are Multi-wA* and anytime repairing PBNF (ARPBNF). Multi-wA* converged more quickly as threads were added, but its performance on finding intermediate solutions did not change too much for different numbers of threads. ARPBNF, on the other hand, took longer to find good solutions for low thread counts, but as threads were added it started to perform better, eventually matching Multi wA* at eight threads. Both of these algorithms improved the solution quality more steadily than AwPBNF and AwAHDA* which had large jumps in their lower hulls. Each of these jumps corresponds to the hull switching to a different weight value (compare with the raw data for AwPBNF in Figure 2-15). All of the parallel algorithms found good solutions faster than serial AwA* and serial ARA*. Some parallel algorithms, however, took longer to prove optimality than AwA* in this domain.

2.8.2 Sliding Tile Puzzles

Figure 2-17 presents lower hulls for the anytime algorithms on Korf's 100 instances of the 15-puzzle. In this figure, the x-axes show the total wall clock time in seconds. These times are not normalized to A* because it is not able to solve all of the instances. In these panels, we see that AwAHDA* tended to find good solutions faster than all other algorithms. AwA* and AwPBNF performed very similarly at two threads and as the number of threads

increased AwPBNF begun to find better solutions faster than AwA*. ARPBNF took longer to find good solutions than AwPBNF and AwAHDA* but it was able to find better solutions faster than its serial counterpart. The simple Multi wA* algorithm performed the worst of the parallel algorithms. Increasing the number of threads used in Multi-wA* did not seem to increase the solution quality. ARA* gave the worst performance in this domain; its profile curve can be seen at the very top of these three panels.

2.8.3 STRIPS Planning

Table 2-6 shows the speedup of the parallel anytime algorithms over serial anytime A*. All algorithms were run until an optimal solution was proved. (For a weight of 5, AwA* ran out of memory on blocks-14, so our speedup values at that weight for that instance are lower bounds.) The **bold** entries in the table represent values that are within 10% of the best performance for the given domain. For all algorithms, speedup over serial generally increased with more threads and a higher weight. PBNF gave the fastest performance for all except two domains (blocks-14 and freecell-3). In these two domains the AwAHDA* gave the best performance by at least a factor of 10x over AwPBNF.

Hansen and Zhou (2007) show that AwA* can lead to speedup over A* for some weight values in certain domains. Finding a suboptimal solution quickly allows f pruning that keeps the open list short and quick to manipulate, resulting in faster performance even though AwA* expands more nodes than A*. We found a similar phenomenon in the corresponding parallel case. Table 2-7 shows speedup over unweighted optimal PBNF when using various weights for the anytime algorithms. A significant fraction of the values are greater than 1, representing a speedup when using the anytime algorithm instead of the standard optimal parallel search. In general, speedup seems more variable as the weight increases. For a weight of 1.5, AwPBNF always provides a speedup.

		AwAPRA*				AwAHDA*			
		1.5	2	3	5	1.5	2	3	5
2 threads	logistics-6	1.09	1.06	1.40	1.40	1.23	1.21	1.59	1.66
	blocks-14	1.36	7.76	56.41	>90.16	1.62	9.90	63.60	**>110.16**
	gripper-7	0.78	0.77	0.76	0.75	1.35	1.33	1.32	1.33
	satellite-6	0.77	0.78	0.78	0.76	1.26	1.23	1.24	1.23
	elevator-12	0.64	0.67	0.69	0.70	1.20	1.19	1.16	1.17
	freecell-3	1.37	1.43	4.61	1.37	1.66	1.68	**5.65**	1.95
	depots-7	1.24	1.30	1.30	2.68	1.51	1.51	1.50	3.18
	driverlog-11	1.15	1.19	1.11	1.20	1.50	1.55	1.46	1.54
	gripper-8	0.61	0.62	0.62	0.62	1.16	1.11	1.14	1.11
7 threads	logistics-6	1.45	1.43	1.81	1.81	2.87	2.81	3.65	3.74
	blocks-14	2.54	15.63	98.52	>177.08	3.30	19.91	132.97	**>231.45**
	gripper-7	1.77	1.68	1.71	1.73	3.75	3.69	3.61	3.67
	satellite-6	1.22	1.22	1.26	1.26	3.56	3.46	3.51	3.50
	elevator-12	0.93	0.93	0.95	0.94	2.77	2.75	2.79	2.77
	freecell-3	3.64	3.75	11.59	4.44	5.00	4.97	16.36	**21.57**
	depots-7	3.60	3.64	3.65	7.60	4.41	4.42	4.40	9.25
	driverlog-11	3.04	3.20	3.05	3.17	4.74	4.82	4.66	4.87

		AwPBNF				AwBFPSDD			
		1.5	2	3	5	1.5	2	3	5
2 threads	logistics-6	1.06	1.35	**1.94**	**1.98**	0.68	0.91	0.91	0.56
	blocks-14	1.91	1.99	13.22	>22.36	1.02	1.18	7.71	>11.92
	gripper-7	**2.05**	**1.96**	**1.99**	**1.95**	**1.94**	**1.89**	**1.94**	1.82
	satellite-6	1.58	**1.96**	**1.98**	**1.91**	**1.85**	**1.87**	1.49	**1.80**
	elevator-12	**2.01**	**2.07**	**2.13**	**2.07**	1.74	1.74	1.75	1.69
	freecell-3	1.93	1.06	2.78	**6.23**	1.45	1.46	1.97	3.08
	depots-7	1.94	2.00	2.01	**4.10**	1.44	1.45	1.32	2.40
	driverlog-11	**1.95**	**2.10**	**1.99**	0.77	1.73	1.78	1.59	1.41
	gripper-8	**2.04**	**2.05**	**2.09**	**2.06**	**2.01**	**2.00**	**1.98**	**1.96**
7 threads	logistics-6	2.04	2.46	**4.19**	**4.21**	1.02	1.35	1.37	0.92
	blocks-14	3.72	22.37	25.69	>7.20	1.60	1.96	12.10	>19.94
	gripper-7	**5.61**	**5.05**	5.03	**5.06**	4.30	4.24	4.16	3.96
	satellite-6	**5.96**	4.66	5.74	4.70	4.10	3.54	4.16	3.88
	elevator-12	**6.18**	**6.03**	**6.20**	**6.05**	3.71	3.74	3.73	3.38
	freecell-3	3.54	1.50	15.32	11.46	1.78	1.82	2.59	4.14
	depots-7	5.74	5.52	5.48	**10.84**	2.02	1.96	1.92	3.68
	driverlog-11	**5.78**	**5.83**	**5.73**	2.18	2.58	2.86	2.57	2.34

Table 2-6: Speed-up of anytime search to optimality over serial AwA* on STRIPS planning using various weights.

		AwPBNF				AwBFPSDD				AwAPRA*			
		1.5	2	3	5	1.5	2	3	5	1.5	2	3	5
7 Threads	logistics-6	1.48	1.84	2.36	2.27	0.68	0.93	0.71	0.54	1.12	1.08	1.08	0.98
	blocks-14	1.24	1.22	0.21	0.03	0.87	0.18	0.16	0.16	1.46	1.46	1.42	0.94
	gripper-7	1.07	0.99	0.99	1.00	0.93	0.95	0.93	0.92	0.99	1.03	1.01	0.99
	satellite-6	1.10	0.87	1.08	0.88	0.88	0.77	0.91	0.90	0.99	1.00	1.01	1.02
	elevator-12	1.06	1.04	1.04	1.03	0.77	0.78	0.76	0.73	1.02	1.00	1.00	1.00
	freecell-3	1.05	0.44	0.99	0.29	0.64	0.64	0.20	0.14	1.13	1.16	0.82	0.10
	depots-7	1.20	1.15	1.15	1.08	0.54	0.53	0.52	0.49	M	M	M	M
	driverlog-11	1.16	1.15	1.19	0.43	0.53	0.58	0.54	0.50	M	M	M	M
	gripper-8	1.06	0.99	0.99	1.00	0.99	0.98	0.99	0.97	M	M	M	M

Table 2-7: Speed-up of anytime search to optimality over PBNF on STRIPS planning problems using various weights.

2.8.4 Summary

In this part of the chapter we have shown how to create some new parallel anytime search algorithms based on the frameworks introduced in the previous sections. We have also created a new parallel anytime algorithm that simply runs many weighted A* searches with differing weights. In our experiments, we have seen that AwPBNF and AwAHDA* found higher quality solutions faster than other algorithms and that they both showed improved performance as more threads were added. Additionally, ARPBNF, a parallel algorithm that is based on ARA*, improved with more threads and tended to give a smoother increase in solution quality than the former two algorithms, although it did not find solutions quite as quickly and it was unable to converge on the optimal solution in the sliding tiles domain within the given time limit. Running multiple weighted A* searches did not give solutions faster as the number of threads increased, and its convergence performance was mixed.

2.9 Discussion

We have explored a set of best-first search algorithms that exploit the parallel capabilities of modern CPUs. First we looked at parallel optimal search with (Safe) PBNF, several variants of PRA* and a set of simpler previously proposed algorithms. Overall, Safe PBNF gave the best performance for optimal search. By reducing search time without increasing execution cost, PBNF addresses the problem of planning under time pressure. Next we created a

set of bounded-suboptimal search algorithms based on PBNF, the successful variants of PRA*, and the BFPSDD algorithm. PBNF and PRA* with asynchronous communication and abstraction (AHDA*) gave the best performance over all, with PBNF doing slightly better on the average. In addition, we showed some results that suggest that bounded-suboptimal PBNF has more of an advantage over serial weighted A* search as problem difficulty increases. Finally we converted PBNF and PRA* into anytime algorithms and compared them with some serial anytime algorithms and a new algorithm called multi-weighted A*. We found that anytime weighted PBNF and the anytime variant of AHDA* gave the best anytime performance and were occasionally able to find solutions faster than their non-anytime counterparts.

Our results show that PBNF outperforms PSDD. We believe that this is because of the lack of layer-based synchronization and a better utilization of heuristic cost-to-go information. The fact that BFPSDD got better as its f layers were widened is suggestive evidence. Another less obvious reason why PBNF may perform better is because a best-first search can have a larger frontier size than the breadth-first heuristic search used by PSDD. This larger frontier size will tend to create more $n$blocks containing open search nodes. There will be more disjoint duplicate detection scopes with nodes in their open lists and, therefore, more potential for increased parallelism.

Some of our results show that, even for a single thread, PBNF can outperform a serial A* search (see Table 2-1). This may be attributed in part to the speculative behavior of the PBNF algorithm. Since PBNF uses a minimum number of expansions before testing if it should switch to an $n$block with better f values, it will search some sub-optimal nodes that A* would not search. In order to get optimal solutions, PBNF acts as an anytime algorithm; it stores incumbent solutions and prunes until it can prove that it has an optimal solution. Zhou and Hansen show that this approach has the ability to perform better than A* (Hansen & Zhou, 2007) because of upper bound pruning, which reduces the number of expansions of nodes with an f value that is equal to the optimal solution cost and can reduce the number of open nodes, increasing the speed of operations on the open list. PBNF may also give

good single thread performance because it breaks up the search frontier into many small open lists (one for each $n$block). Because of this, each of the priority queue operations that PBNF performs can be on much smaller queues than A*, which uses one big single queue (see Section 2.4.6).

2.9.1 Possible Extensions

While the basic guideline for creating a good abstractions in SDD (and PBNF) is to minimize the connectivity between abstract states, there are other aspects of abstraction that could be explored. For instance, discovering which features are good to include or abstract away may be helpful to users of PBNF. Too much focus on one feature could cause good nodes to be too focused in a small subset of $n$blocks (Zhou & Hansen, 2011). Likewise, size of the abstraction could be examined in more detail. Although we always use a constant abstraction size in our current work for simplicity it seems likely that abstraction size should change when number of threads changes or perhaps even based on features of the domain or problem instance. If a guideline could be devised, such as a ratio between number of $n$blocks to threads or h value of the start state, a problem-adaptive abstraction size would be much simpler in real world use. Additionally, edge partitioning (Zhou, Schmidt, Hansen, Do, & Uckun, 2010) could allow us to reduce connectivity of the abstraction used by PBNF, but further study will be necessary to discover the full impact of this technique on PBNF's behavior.

Some possible future extensions to PBNF include adaptive minimum expansion values, use of external memory, and extension to a distributed setting. Our preliminary work on adapting minimum expansion values indicated that simply increasing or decreasing based on lock failures and successes had either neutral or negative effect on performance. One reason for this may be because the minimum expansions parameter adds speculation.

It may be possible to combine PBNF with PRA* in a distributed memory setting. This algorithm may use a technique based on PRA* to distribute portions of the search space among different nodes on a cluster of work stations while using a multicore search such as

PBNF on each node.

An additional technique that was not explored in this chapter is running multicore search algorithms with more threads than there are available cores. This technique has been used to improve the performance of parallel delayed duplicate detection (Korf, 1993; Korf & Schultze, 2005) which is heavily I/O intensive. Using this approach, when one thread is blocked on I/O another thread can make use of the newly available processing core. Even without disk I/O this technique may be useful if threads spend a lot of time waiting to acquire locks.

2.10 Conclusions

In this chapter we have investigated algorithms for best-first search on multicore machines. We have shown that a set of previously proposed algorithms for parallel best-first search can be much slower than running A* serially. We have presented a novel hashing function for PRA* that takes advantage of the locality of a search space and gives superior performance. Additionally, we have verified results presented by Kishimoto et al. (2009) that using asynchronous communication in PRA* allows it to perform better than using synchronous communication. We presented a new algorithm, PBNF, that approximates a best-first search ordering while trying to keep all threads busy. We proved the correctness of the PBNF search framework and used it to derive new suboptimal and anytime algorithms.

We have performed a comprehensive empirical comparison with optimal, suboptimal and anytime variations of parallel best-first search algorithms. Our results demonstrate that using a good abstraction to distribute nodes in PRA* can be more beneficial than asynchronous communication, but that these two techniques can be used together (yielding AHDA*). We also found that the original breadth-first PSDD algorithm does not give competitive behavior without a tight upper bound for pruning. We implemented a novel extension to PSDD, BFPSDD, that gives reasonable performance on all domains we tested. Our experiments, however, demonstrate that the new PBNF and AHDA* algorithms outperformed all of the other algorithms. PBNF performs best for optimal and bounded-

suboptimal search and both PBNF and AHDA* gave competitive anytime performance.

In the next chapter, we evaluate the core techniques used in PBNF on a new type of search problem: model checking.

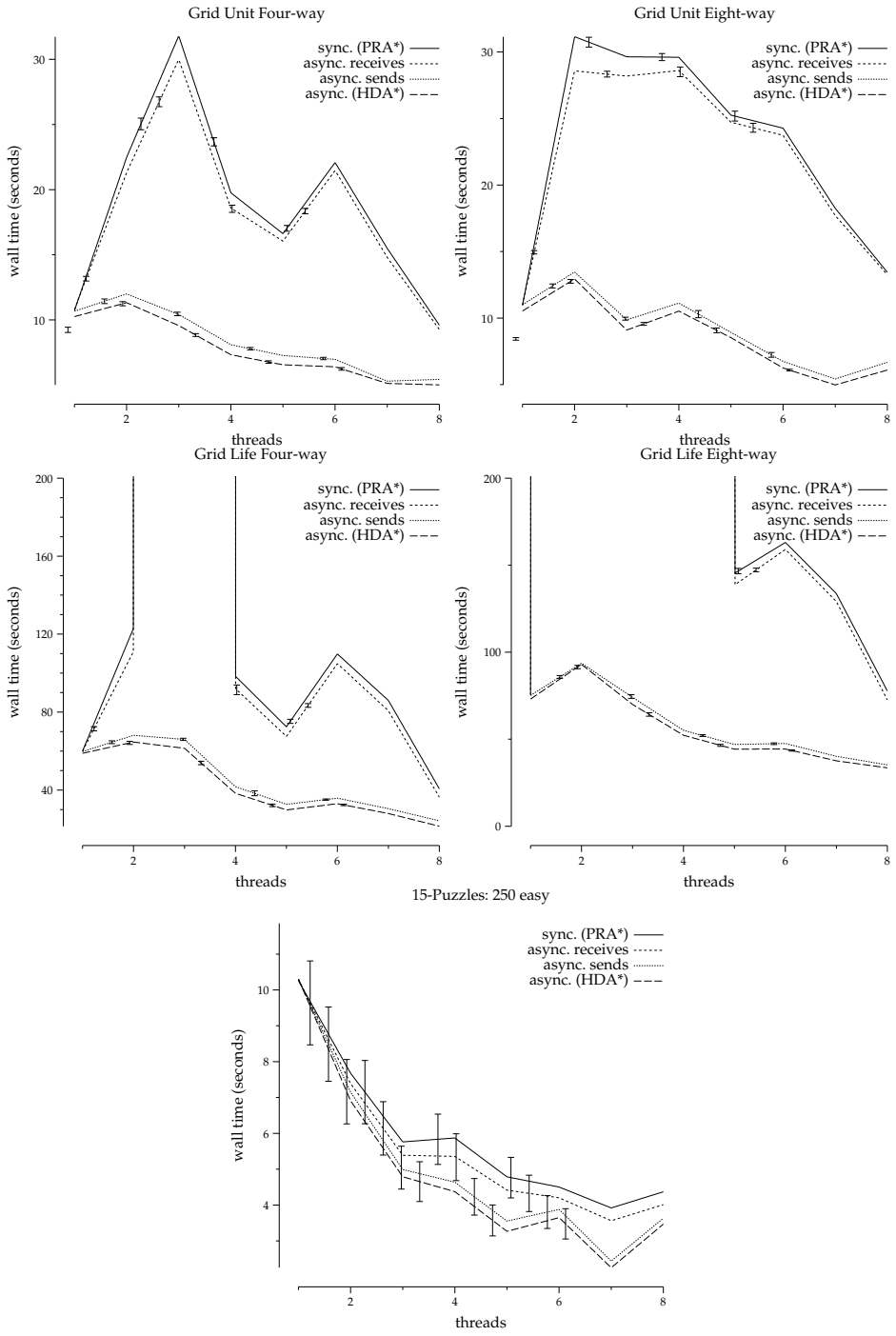


Figure 2-6: PRA* synchronization: 5000x5000 grids and easy sliding tile instances.

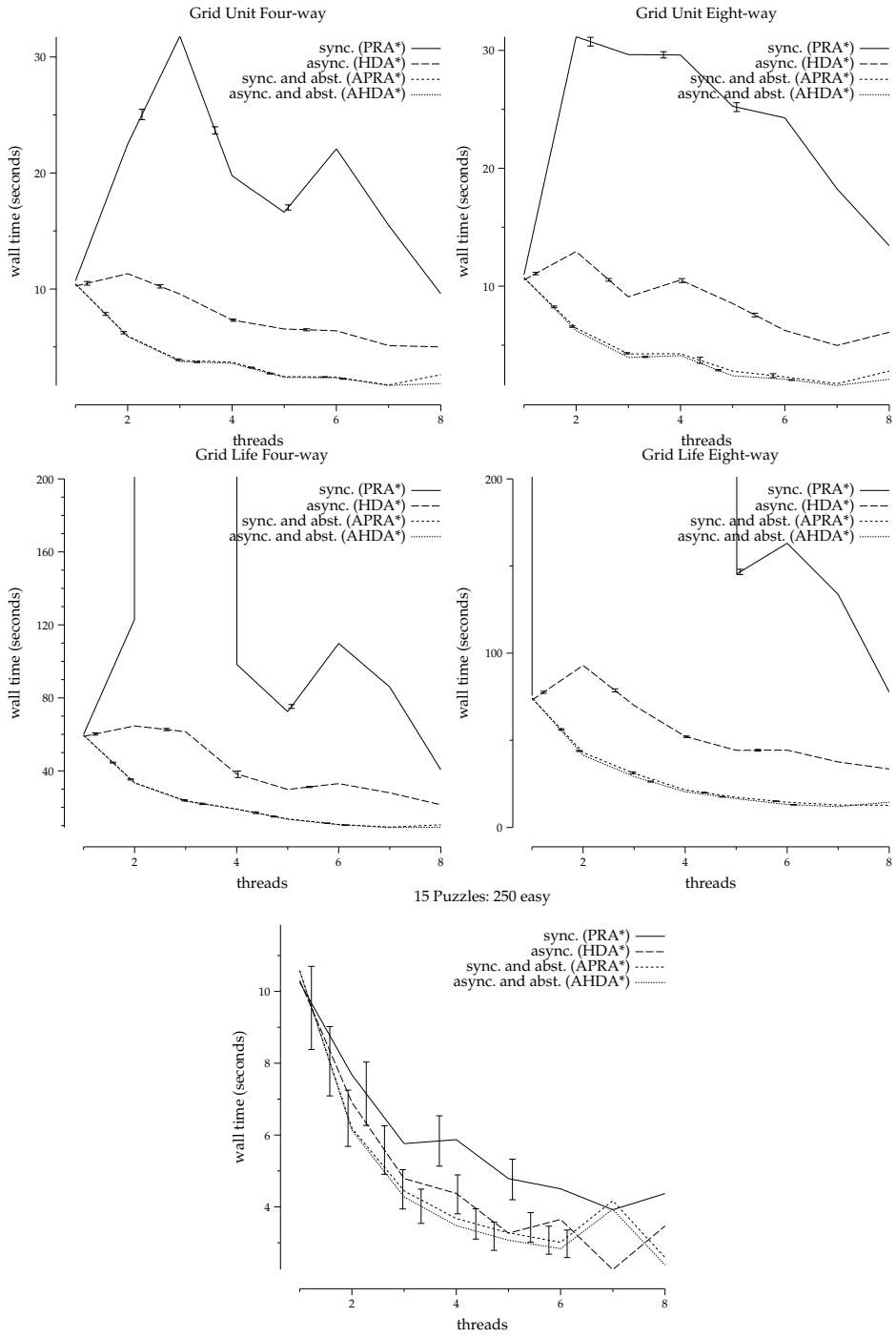


Figure 2-7: PRA* abstraction: 5000x5000 grids and easy sliding tile instances.

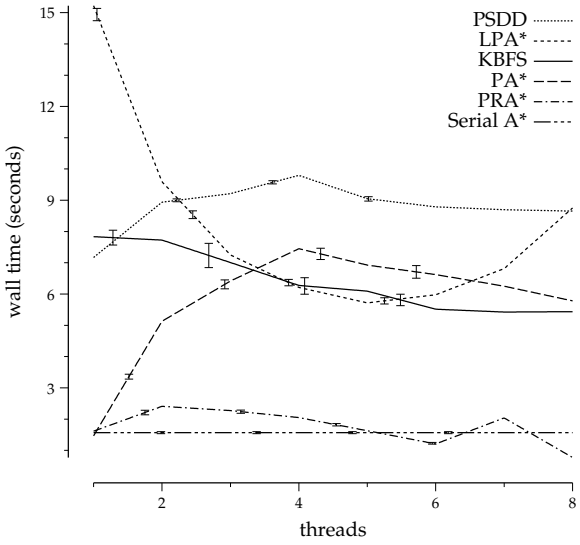


Figure 2-8: Simple parallel algorithms on unit cost, four-way 2000x1200 grid pathfinding.

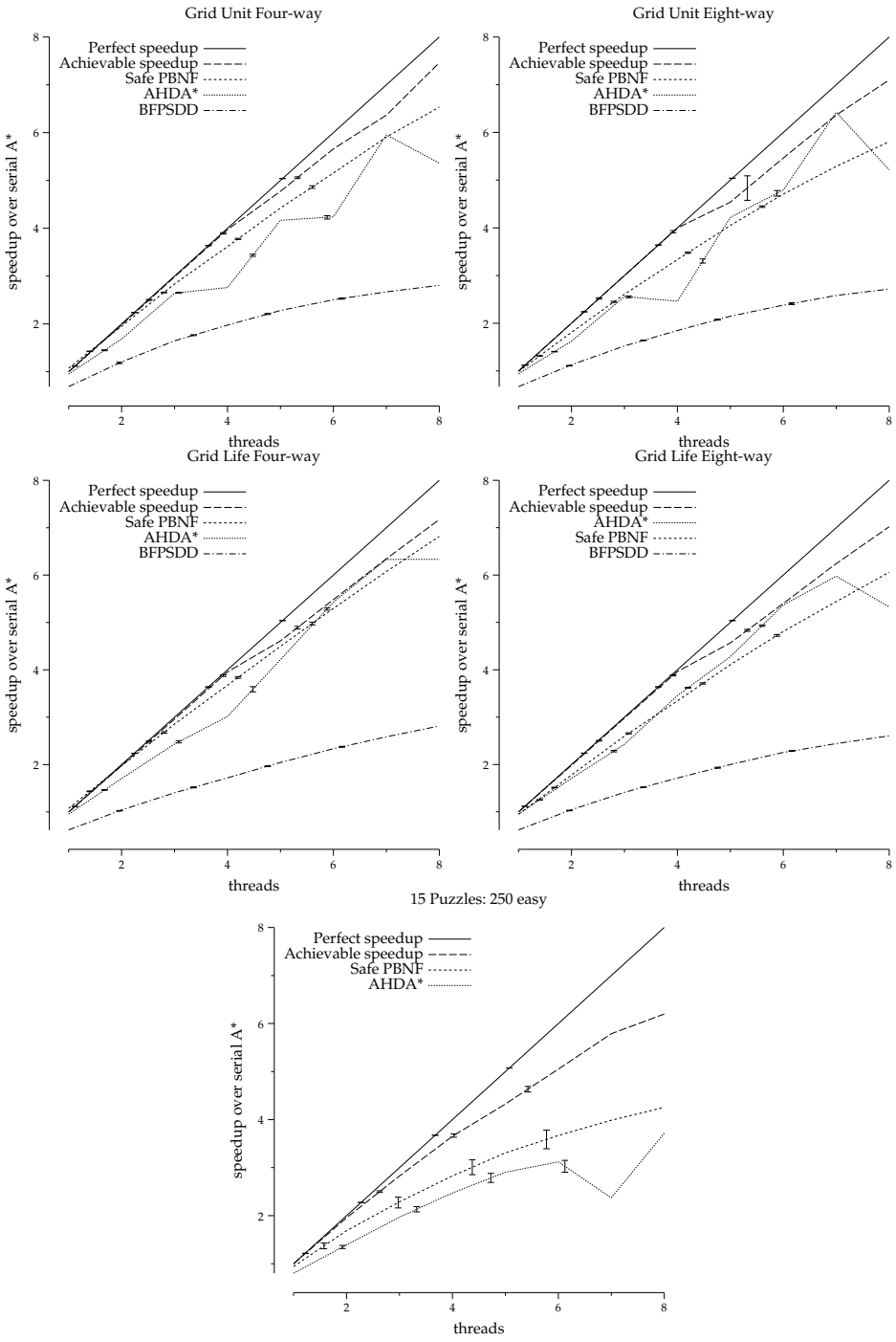


Figure 2-9: Speedup results on grid pathfinding and the sliding tile puzzle.

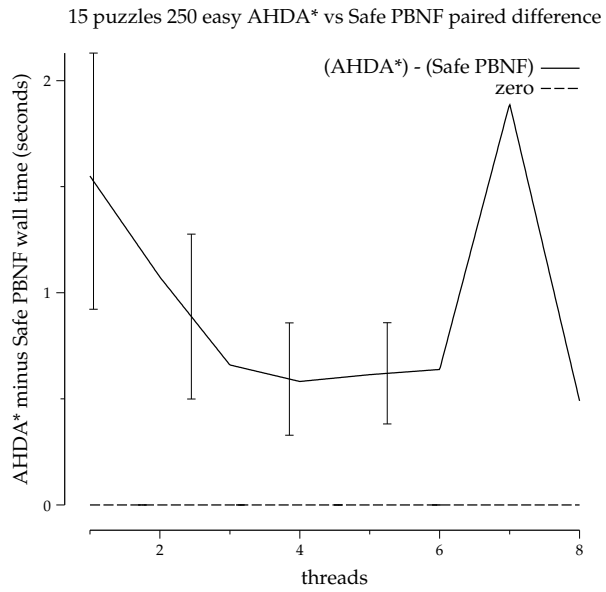


Figure 2-10: Comparison of wall clock time for Safe PBNF versus AHDA* on the sliding tile puzzle.

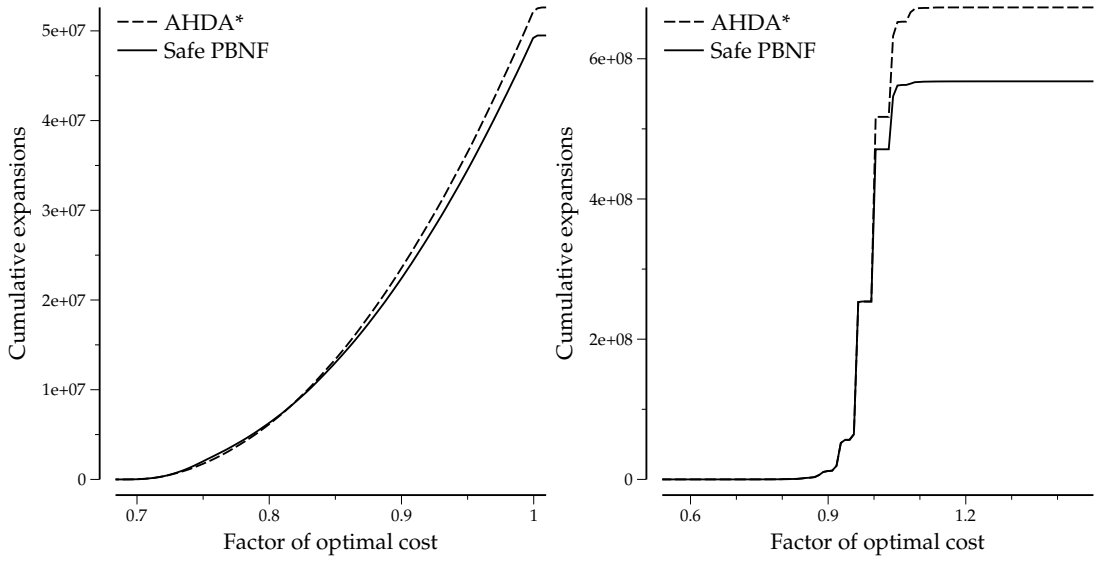


Figure 2-11: Cumulative normalized f value counts for nodes expanded with eight threads on unit-cost four-way grid pathfinding (left) and the 15-puzzle (right).

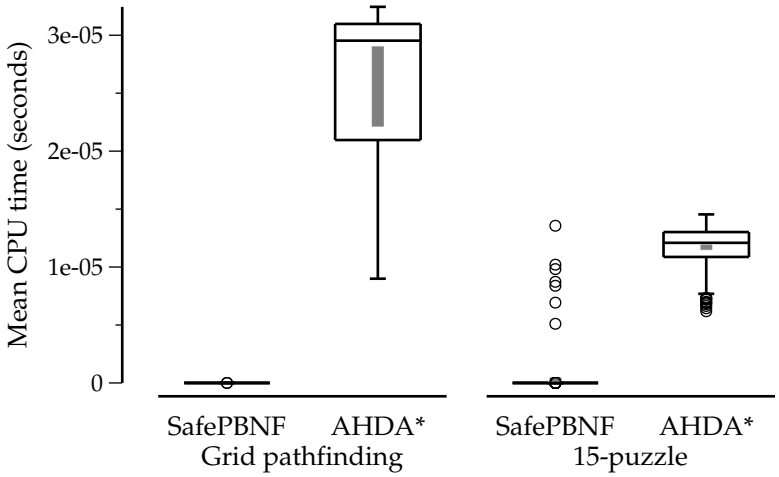


Figure 2-12: Mean CPU time per open list operation.

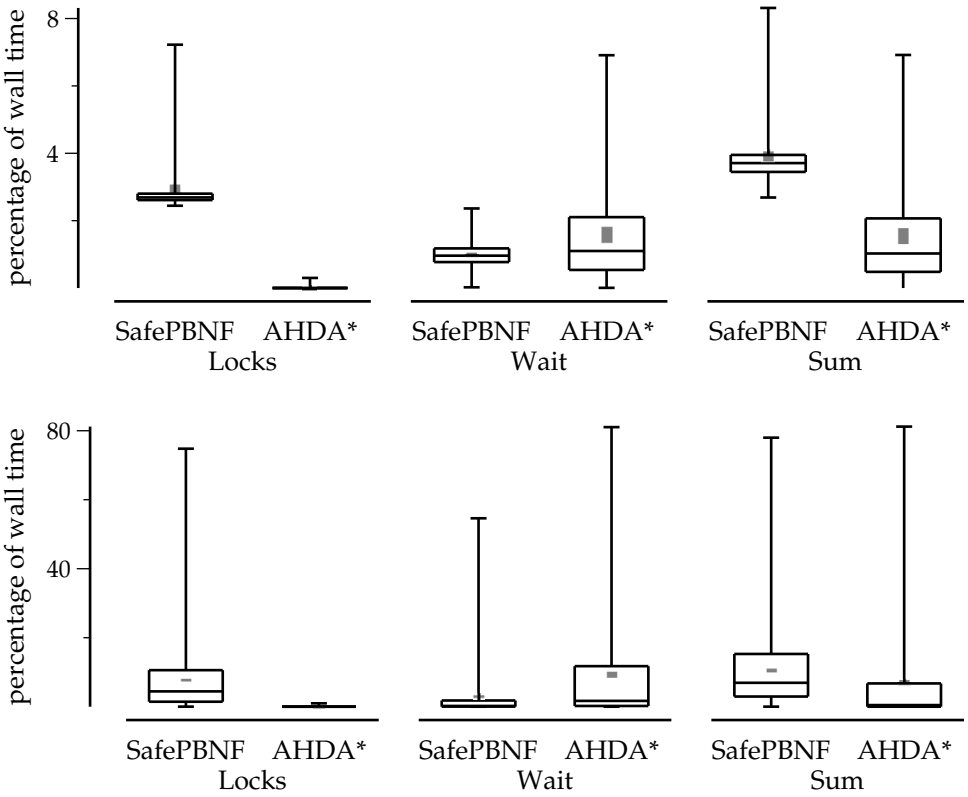


Figure 2-13: Per-thread ratio of coordination time to wall time on unit-cost four-way pathfinding (top) and the 15-puzzle (bottom).

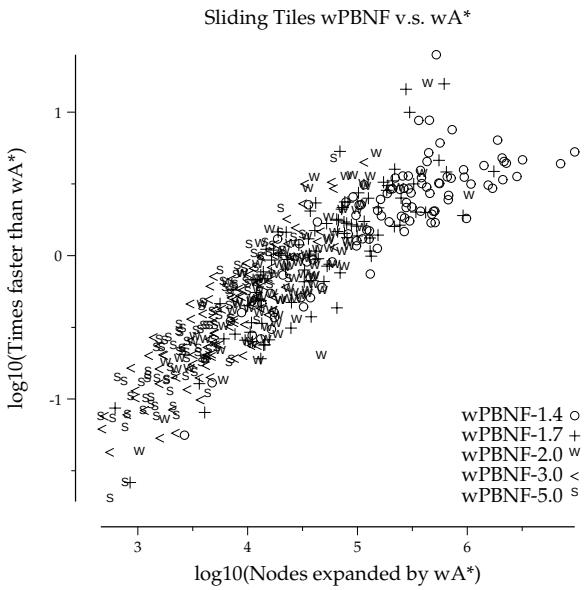


Figure 2-14: wPBNF speedup over wA* as a function of problem difficulty.

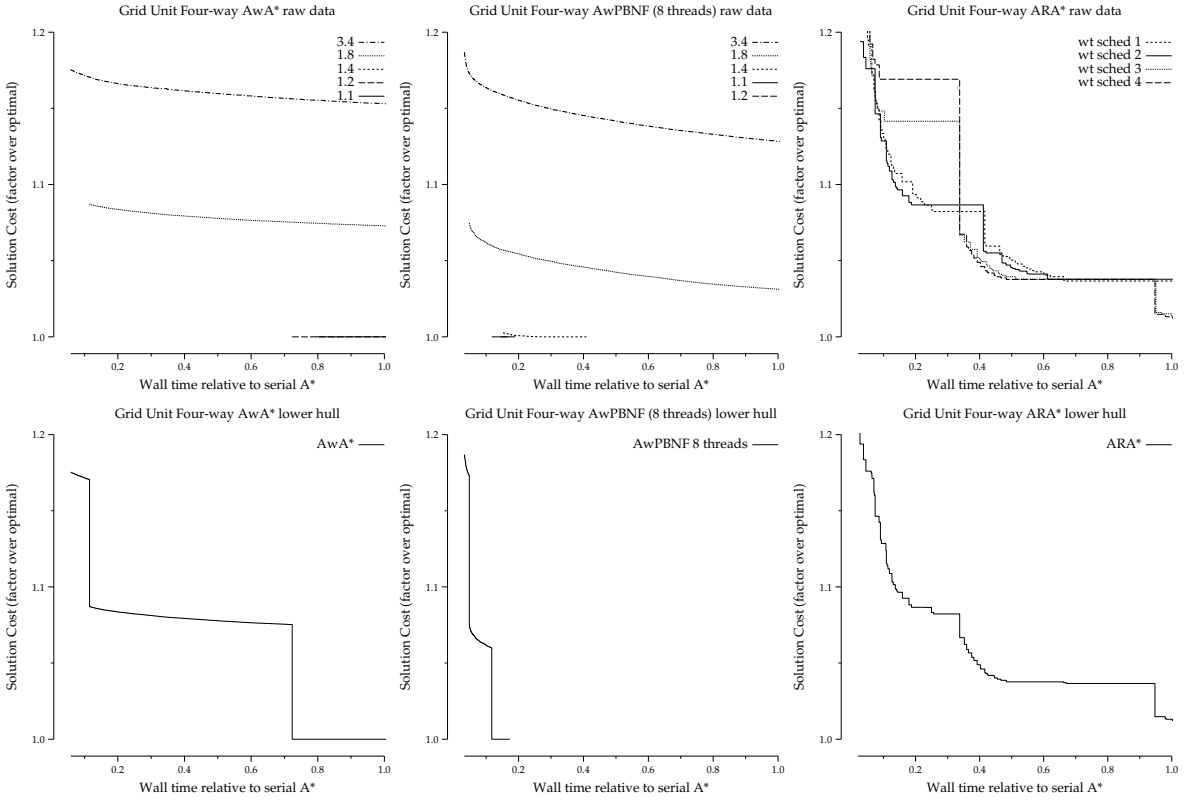


Figure 2-15: Raw data profiles (top) and lower hull profiles (bottom) for AwA* (left), AwPBNF (center), and ARA* (right). Grid unit-cost four-way pathfinding.

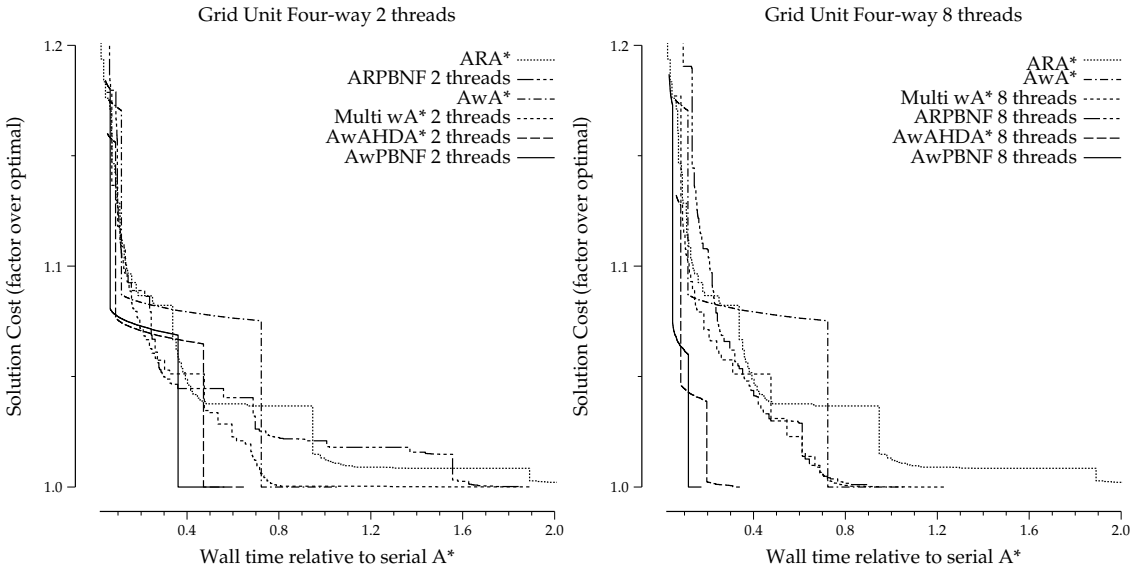


Figure 2-16: Grid unit-cost four-way pathfinding lower hull anytime profiles.

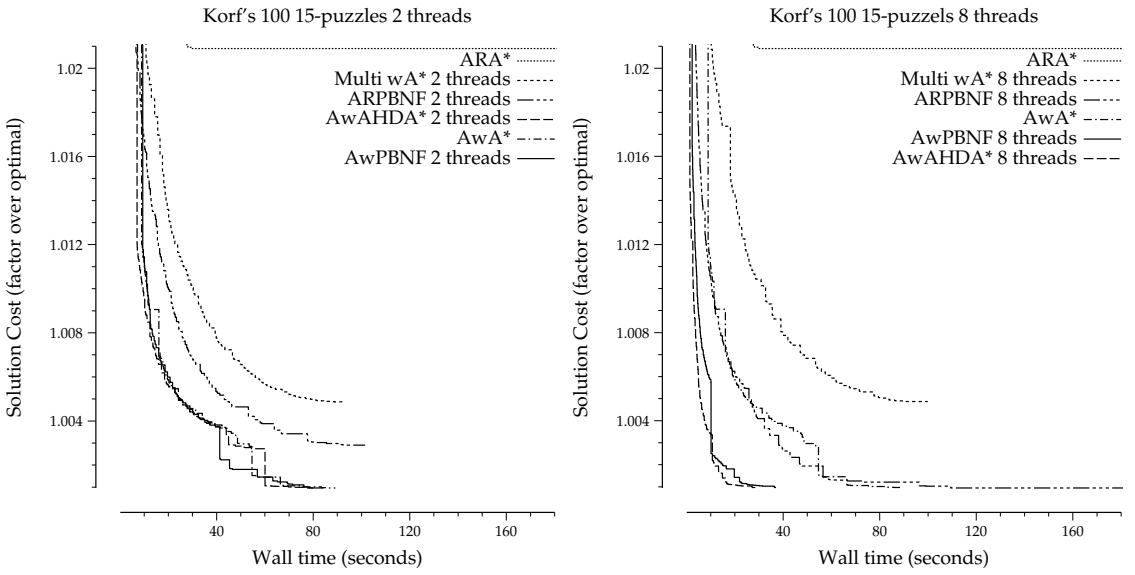


Figure 2-17: Korf's 100 15-puzzles lower hull anytime profiles.

CHAPTER 3

PARALLEL MODEL CHECKING

3.1 Introduction

Model checking is a fundamental tool used in the creation and verification of asynchronous and distributed systems. Since the actions performed by each component of such a system may be interleaved in many ways, there can be a large number of configurations of the system as a whole. Given an abstract model of a system, a model checker can enumerate all reachable configurations of the model in order to aid in verification of its correctness. During enumeration, the model checker can ensure that the model does not exhibit any invalid behaviors or reach any invalid states. If such an error is found then a trace of the actions leading to it can be reported back to the user. This trace information is invaluable when creating and debugging a new system. Additionally, if the model checker is unable to find any invalid behaviors then it is evidence that the system is in fact correct.

To enumerate the all possible states of an asynchronous system, many popular model checkers treat the configuration space as an implicitly defined graph where nodes correspond to system states and edges are the possible transitions of each component. A path through this graph gives one possible interleaving of the actions that the system may perform. Once the graph is defined, an exhaustive search algorithm can then explore all reachable states of the system looking for ones that violate certain properties. As is typical with implicit graphs, however, there can be a very large number of nodes causing the search to take a prohibitive amount of time or memory.

The model checking community, the heuristic search and automated planning communities have all been quite successful in developing new search frameworks that take advantage of modern multi-core processors. These frameworks have enabled them to improve the

performance of their algorithms and have also been shown to be successful at offloading a significant portion of the memory requirement of a large graph search to external storage devices such as hard disks. However, some of the most successful techniques used by the heuristic search and planning communities have yet to be tested for model checking. Because of their success on other types of search problems, we would like to compare these approaches to those commonly used to parallelize search in model checking.

We have implemented two techniques for parallelizing breadth-first search in the Spin model checker (Holzmann, 2004). The first technique is similar to the PRA* (Evett et al., 1995) and HDA* (Kishimoto et al., 2009) algorithms presented in Chapter 2. The hash distribution technique is a common approach for parallel model checking (Stern & Dill, 1997). We call our implementation of this algorithm hash-distributed breadth-first search (HD-BFS). The second method is called parallel structured duplicate detection (PSDD) Zhou & Hansen, 2007, and it uses a similar framework to PBNF.

As we will see, PSDD has a few major advantages over hash-distributed search for model checking. First, HD-BFS uses delayed duplicate detection (Korf, 2008) and must store duplicate search nodes temporarily while they are being communicated between threads. PSDD is able to detect duplicate states immediately after they are generated thus abolishing the need to use extra memory in order to store them. Second, PSDD is able to preserve Spin's ability to perform partial-order reduction—a technique used by model checkers to decrease the size of the search space. This means that, when using multiple threads, PSDD is often able to search a significantly smaller space than both HD-BFS and Spin's built-in multi-core depth-first search, both of which must be more conservative when performing partial-order reduction. Overall, the results of our experiments demonstrate that PSDD is faster and able to achieve greater parallel speedup than both HD-BFS and Spin's state-of-the-art multi-core depth-first search.

In addition to improving the performance of breadth-first search, we show some preliminary results demonstrating that PSDD can also successfully reduce the memory requirements of model checking by making use of external storage devices. In one experiment

PSDD is able to reduce the memory requirement of the search by over 500% when using a hard disk to supplement internal memory.

3.2 Depth-first Versus Breadth-first Search

Two of the most well-known graph search algorithms are depth-first search and breadth-first search. Depth-first search generates the successors of nodes in the graph in deepest-first order. This means that the most recently generated node will be the next node that is expanded. Breadth-first search, on the other hand, expands nodes in shallowest-first order. Spin uses depth-first search by default as it is able to check both safety properties (typically used to verify that something undesirable will not happen) and liveness properties (typically used to verify that something desirable will eventually happen) whereas Spin's breadth-first search algorithm is only able to verify safety properties.

Breadth-first search for model checking is guaranteed to find shortest counter-examples if the model violates a safety property. This is significant because, many important properties of an asynchronous system are safety properties and when debugging a system one must understand the counter-example provided by the model checker in order to determine why the system is not behaving as desired. Depth-first search pays no heed to the number of steps used to reach a node in the state space and therefore may produce a counter example that is many steps longer than necessary. These long traces can be extremely hard to interpret as they will contain a lot of transitions that are not necessary to produce the faulty behavior. To put this in perspective, on one model we have observed that depth-first search finds a deadlock and provides a trace consisting of 9,992 steps where breadth-first search finds a trace for the deadlock with the smallest number of possible steps: 42.

While breadth-first search cannot be used directly to verify liveness properties, there has been work on efficient translations of liveness checking problems into safety checking problems, which can subsequently be verified by breadth-first reachability analysis (Biere, Artho, & Schuppan, 2002; Schuppan & Biere, 2004). Given depth-first search's inherently sequential nature (Reif, 1985), checking liveness property using a breadth-first, instead of

depth-first, strategy can better leverage the latest multi-core processors for greater parallel speedups.

3.3 Hash-distributed Breadth-first Search

In Chapter 2, we discussed the difficulties in parallelizing best-first search algorithms such as breadth-first search[1] and we saw that many naïve implementations of parallel search actually perform worse than their serial counterparts.

In order to successfully search a graph in parallel the graph should be divided in a way that each thread performing the search can operate on an independent portion of the graph. A simple way to achieve this is to divide the nodes of the graph statically using a hash function; as each new node is generated, its hash value is computed and it is distributed to the thread with the thread ID equal to the hash value modulo the number of threads. If a node is generated multiple times, each duplicate will be assigned to the same thread so duplicate detection can be performed locally within each thread. This framework is called hash-distributed search and was originally proposed as a method for parallelizing the A* algorithm (Evett et al., 1995) and was later discovered by Stern and Dill (Stern & Dill, 1997) in the context of model checking and then by Kishimoto et al. (2009) who called the algorithm hash-distributed A* (HDA*) and applied it to automated planning problems.

We have implemented a hash-distributed breadth-first algorithm, based on HDA*. We call this algorithm hash-distributed breadth-first search (HD-BFS). HD-BFS works in layers by expanding the nodes at a given depth from the root in parallel until all nodes at the current depth have been expanded. When a depth layer has been completely expanded, all threads proceed synchronously to the next depth and begin searching there.

Each HD-BFS thread uses a pair of queues to represent the search frontier. One queue, called the *current queue*, contains all nodes assigned to the thread that are at the current search depth. The second queue, called the *next queue*, contains all nodes assigned to

[1]Breadth-first search can be viewed as a special best-first search where all edges have unit cost.

the current thread that are at the next search depth. Each thread also has a hash table containing all nodes that it has previously expanded. This table is used to prevent the search from expanding the same nodes multiple times. Note that, because all duplicates of a search node will be assigned to the same thread by the hash function, no node resides in more than a single hash table.

When searching, each thread expands the nodes from its current queue one-at-a-time. When a successor node is assigned by the hash function to a different thread than the one that generated it, it must be sent there using inter-thread communication. Otherwise, when a successor node is assigned back to the same thread that generated it, it is immediately checked for membership in the local hash table to determine if it is a duplicate and if it is not a duplicate then it is added to the next queue for the local thread; no communication is required. Our implementation of HD-BFS uses the same communication scheme described for PRA* in the previous chapter to send nodes between threads asynchronously using shared-memory queues.

After receiving a new node sent from a different thread, the receiving thread checks to see if the node is a duplicate by testing it for membership in its local hash table. If the node is not a duplicate then it is placed on the thread's next queue. This is the appropriate queue because all threads are expanding nodes at the same depth from the root and therefore any generated node resides at the next depth regardless of which thread generated it.

If all threads have empty current queues and no nodes are in transit between threads, then the current depth layer has been completely expanded. When this happens, all threads synchronously swap their next queue with their current queue and begin searching nodes at the next depth. If all current queues are still empty after swapping to the next depth then the search space has been exhausted and the algorithm terminates.

3.3.1 Disadvantages

We have found that there are two major disadvantage to hash-distributed search when applied to model checking. The first is that hash-distributed search delays the detection

of duplicate nodes when they are communicated between threads. When nodes are sent to another thread they are placed on the receiving queue for that thread and sit there until they are eventually received and checked against the receiving thread's hash table. This delayed detection of duplicate nodes can cause the search to require more memory as the duplicates reside in the receiving queue instead of immediately having their memory freed for reuse. As we will see, the extra memory overhead created by delaying duplicate detection can be quite substantial.

The second disadvantage of hash-distributed search is that it must be conservative when applying partial-order reduction (Holzmann & Peled, 1994), a technique used in model checking to reduce the size of the search graph. When expanding a node while using partial-order reduction, only a subset of the successors are considered and the rest are discarded. While performing breadth-first search with partial-order reduction, Spin uses a test called the *Q proviso* (Bošnački & Holzmann, 2005) to prevent reduction in cases where completeness can not be ensured. When generating the successors of a node, the Q proviso tests if any of the successors already resides on or is placed on the breadth-first search queue. If the Q proviso is satisfied then the reduction can take place, otherwise the full expansion must happen. Bošnački and Holzmann (2005) proved that this simple test allows breadth-first search to remain complete under partial-order reduction when searching for safety property violations and deadlocks.

With hash-distributed search, the successors of a node may not be assigned to the expanding thread. When this happens, the expanding thread does not have the ability to test if the successors are on or end up on the queue because this queue is owned by a different thread. To preserve completeness, HD-BFS must be conservative and assume that all nodes that are sent to different threads do *not* pass the Q proviso. This reduces the chances of successfully performing partial-order reduction because, in order to reduce, a thread must generate a successor that is assigned to itself and also passes the Q proviso. As we will show in our experimental results, with a greater number of threads the chance that successors will not be assigned to the expanding thread increases, so as the number of

threads increases the size of the search space will increase too. Because of this, HD-BFS using multiple threads can actually perform worse than a serial breadth-first search because the former must search a significantly larger space to guarantee completeness.

3.3.2 Abstraction-based Hashing

Both of the previous issues with hash-distributed search stem from the fact that the hash function used to assign nodes to threads is designed to uniformly distribute the nodes. This is beneficial from a load balancing perspective, however, it means that it is uncommon for the successors of a node to be assigned to the thread that generated them. In the previous chapter, we saw a novel modification to hash-distributed search that helped alleviate this issue at the cost of possibly decreasing load balancing: instead of using a hash function that distributes the nodes uniformly, a homomorphic abstraction function can be used to distribute the nodes in a more structured fashion (recall APRA* and AHDA*). Each thread is responsible for a set of nodes in an abstract representation of the search graph. When a node is generated, its abstract representation is computed and it is assigned to the thread responsible for this abstract node.

The advantage of this approach, when using a carefully created abstraction, is that the successors of a search node will tend to be assigned back to the same thread that generated them. This means that the need for communication is reduced as newly generated nodes can often be handled locally. The disadvantage is that the search load may not be evenly balanced among the threads. Previously, we saw that, in practice, using an abstraction instead of a uniformly distributed hash function greatly increased the performance of PRA* and HDA* on puzzle solving and planning problems.

For model checking, fewer communications mean fewer duplicate nodes that reside in memory. It also means that there are more chances to perform partial-order reduction. As we will see, this approach can greatly reduce the memory requirements and the size of the search space explored by hash-distributed search. Unfortunately, because the nodes are no longer distributed uniformly among the threads, this abstraction-based implementation of

HD-BFS (which we call AHD-BFS) gives very brittle performance for different numbers of threads. We suspect that the nodes tend to be distributed unevenly causing some threads to be very busy and some threads to starve for work. This behavior hinders the ability of the search to fully exploit the available parallelism.

3.4 Parallel Structured Duplicate Detection

Instead of assigning nodes to threads *a priori* by using a hash function, Zhou *et al.* (Zhou & Hansen, 2007) developed a framework called *Parallel Structured Duplicate Detection* (PSDD) that allows threads to dynamically divide the search effort. Like PBNF, PSDD uses a homomorphic abstraction to map nodes in the search graph to nodes in an abstract representation of the search graph. We review the structured duplicate detection technique more formally here.

Given a search graph and a homomorphic abstraction function, an *abstract graph* is constructed as follows.

1. The set of nodes, called *abstract nodes*, in the abstract graph corresponds to the set of abstract states.

2. An abstract node y' is a successor of an abstract node y if and only if there exist two states x' and x, such that

    a. x' is a successor of x, and

    b. y' and y are images of x' and x, respectively.

The abstract graph is used during search to locate portions of the search space that are disjoint. More formally, let abstract node $y = \phi(x)$ be the image of state x under a homomorphic abstraction function $\phi(\cdot)$ and let $succ(y)$ be the set of abstract successor nodes of y in the abstract graph.

Definition 1 *The* duplicate-detection scope *of a state x under a homomorphic abstraction function $\phi(\cdot)$ corresponds to the union of sets of stored nodes that map to an abstract node*

y' *such that* $y' \in succ(\phi(x))$*, that is,*

$$\bigcup_{y' \in succ(\phi(x))} \phi^{-1}(y')$$

where $\phi^{-1}(y')$ *is the set of stored nodes that are pre-images of* y'*.*

Proposition 1 *The duplicate-detection scope of a node contains all stored duplicates of the successors of the node.*

Definition 2 *The duplicate-detection scopes of states* x_1 *and* x_2 *are* disjoint *under a homomorphic abstraction function* $\phi(\cdot)$*, if and only if the set of abstract successors of* x_1*'s image is disjoint from the set of abstract successors of* x_2*'s image in the abstract graph, that is,* $succ(\phi(x_1)) \cap succ(\phi(x_2)) = \emptyset$*.*

Proposition 2 *Two states cannot share a common successor if their duplicate-detection scopes are disjoint.*

Proposition 2 provides an important guarantee that a parallel model checker can leverage to reduce the amount of synchronization needed in parallel graph search. In particular, multiple threads can search disjoint portions of the graph, which correspond to disjoint duplicate-detection scopes, without the need for communication. Unlike HD-BFS, duplicate states are detected in PSDD as soon as they are generated.

As with HD-BFS, the search proceeds in layers. Each node in the abstract graph has two queues, one for the current depth-layer and one for the next. These queues contain the frontier nodes of the search graph that map to the given abstract node. Each abstract node also has a hash table containing all of the previously expanded search nodes that map to it.

Threads acquire access to expand all of the search nodes at the current depth for a single abstract node at a time. Because the abstraction is homomorphic, the successors of a search node will either map to the same abstract node or to one of the successors in the abstract graph. By claiming exclusive access to an abstract node and its successors, a thread can expand from the abstract node and perform immediate duplicate detection on the generated successors using only the data structures to which it has exclusive access.

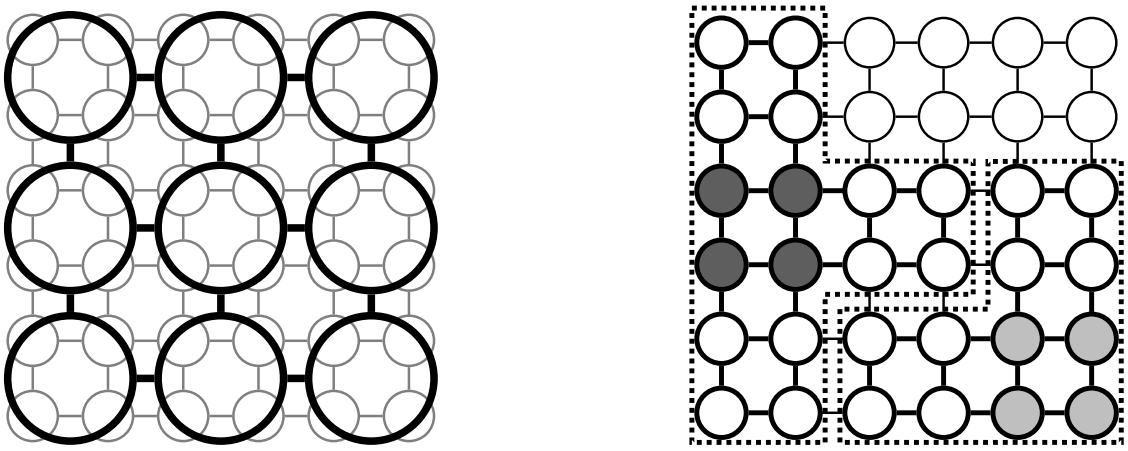


Figure 3-1: A graph along with one of its possible abstractions (left) and two disjoint duplicate detection scopes of this graph (right)

We call the set of nodes corresponding to an abstract node and its successors a *duplicated detection scope* (see Definition 1) or just a *scope* for short.

The left image in Figure 3-1 shows an example graph in light gray with a possible abstraction of the graph drawn in dark black on top of it. This abstraction groups together sets of four nodes. There is an edge in the abstract graph between each pair of abstract nodes for which there exists a pair of nodes in the underlying graph that are connected by an edge and whose images correspond to each respective abstract state. The right image in Figure 3-1 shows two duplicate detection scopes in this graph, each defined by the gray nodes and surrounded by a dashed line. Both duplicate detection scopes consist of the gray nodes and all nodes that map to the successors of their image in the abstract graph. When expanding any of the gray nodes, all successors will correspond to a node that resides in the same duplicate detection scope.

To perform parallel search, each thread will use the abstract graph to locate a duplicate detection scope that does not overlap the scopes being used by other threads. Given Proposition 2, these *disjoint duplicate detection scopes* may be searched in parallel without requiring communication. With this scheme, the only time that threads must synchronize is when multiple threads require access to the abstract graph at the same time. Only a single

mutex is required to serialize access to the abstract graph and operations on the abstract graph tend to be quick.

The two duplicate detection scopes shown on the right half of Figure 3-1 are disjoint as they do not share any nodes.

When a thread completes the expansion of all open search nodes mapping to its current abstract node, it can release its duplicate detection scope, marking all abstract nodes in the scope as free to be re-acquired. Then the thread can try to acquire a new scope to search. If there are no free scopes with open search nodes at the current depth then the thread attempting to acquire a new scope must wait until another thread finishes expanding and releases its abstract nodes. This wait time can be reduced by using a finer-grained abstraction with sufficiently many disjoint duplicate detection scopes. In practice, we find that abstractions can typically be made large enough that wait times are insignificant (cf 2.4.1).

Eventually, as open search nodes become exhausted in the current depth-layer, there will be only a single thread actively searching as the other threads wait for abstract nodes to become free. When the final non-waiting thread releases its duplicate detection scope and finds that there are no free scopes with open nodes it will progress the search to the next depth layer. To do this, the current and next queues for each abstract node are swapped, all abstract nodes with open nodes in their new current layer are marked as free, waiting threads are woken up and the search resumes. If the new depth-layer contains no open search nodes then the search space has been exhausted and the threads can terminate.

PSDD provides at least two advantages over hash-distributed search: 1) there may be less synchronization between threads in PSDD because threads only need to synchronize access to the abstract graph when releasing and acquiring a new duplicate detection scope and 2) duplicates can be checked immediately instead of using extra memory to store duplicate nodes before they can be checked against the hash table.

PSDD provides an additional benefit when applied to model checking: it does not need to be conservative when performing partial-order reduction. Recall that HD-BFS did not have access to test if successor nodes reside on the breadth-first queue when the successors

were not assigned to the expanding thread. In PSDD, however, the expanding thread has exclusive access to the data structures for the duplicate detection scope of the abstract node from which it is expanding. This means that PSDD is able to test if the successors that it is generating pass the *Q proviso* and therefore it does not need to be conservative when performing partial-order reduction. As we will see, this gives PSDD a major advantage over both HD-BFS and Spin's multi-core depth-first search on many models.

3.4.1 Abstraction for Model Checking

PSDD requires an abstract representation of the state-space graph in order to exploit the local structure of the space. Since the state space is not explicitly represented in memory, this abstraction must be a function that can be computed on each node. In Spin, each state in the search space consists of the set of processes whose executions are being modeled. Each process is represented by a finite automaton which has a current state and a set of transitions. The abstraction that we used in our implementation of PSDD is: given any state, consider only the process type and the automaton state of a fixed subset of the process IDs. All other information is projected away. For example, consider a state with seven processes numbered 0–6. One possible abstraction is to consider only the automaton states of the first two process IDs. This effectively ignores process IDs 2–6, leading to a much smaller set of abstract nodes.

We use the transitions of the finite automaton to determine the predecessor and successor relations in the abstract graph. Because only the state of a single component automaton will transition between a node and its successors[2], the successors in the abstract graph are all of the possible single transitions of the process IDs that have not been removed in the abstraction. For efficiency, we generate the abstract graph lazily as needed during the search. This provides the benefit of only instantiating the portions of the graph that are actually used and it also constructs the graph in parallel with the execution of the search

[2]For Spin, this is not strictly true when using *never claims*. Our implementation requires all never claims to be projected away.

instead of doing it serially as a pre-processing step.

3.5 Experimental Results

In this section we present the results of a set of experiments that we performed to evaluate the two methods of parallelizing breadth-first search. In addition, we compare to Spin's built-in multi-core depth-first search where applicable. The machine used in our experiments has two 3.33GHz Xeon 5680 processors, each having six cores, and 96GB of RAM.

3.5.1 Multi-core Depth-first Search

Spin comes, by default, with a state-of-the-art multi-core depth-first search algorithm (Holzmann & Bošnački, 2007). The algorithm connects each of the threads performing the search in a ring. Nodes may be passed from one thread to another around the ring in a single direction. Each thread is then responsible for expanding all of the nodes that fall within a particular depth interval. When the successors of a node fall outside of an interval assigned to the current thread, the newly generated successors must be passed to the neighboring thread along the ring using a shared memory queue. This neighboring thread may then receive the nodes from the queue and begin expanding them.

Using this technique, Holzmann and Bošnački (2007) were able to achieve speedups of just over 1.6x at two threads on a set of benchmark models and almost perfect linear speedup for two threads on a reference model that provided a set of tunable parameters. In their results, however, they show that this technique must be conservative when doing partial-order reduction. So, as with HD-BFS, the performance of multi-core depth-first search can actually be worse than serial search when partial-order reduction is used.

In the following experiments, we compare to Spin's multi-core depth-first search on models which do not contain safety property violations, because it finds these violations via suboptimal paths. Since breadth-first search is constrained to only return optimal length traces and it must perform significantly more work, rendering the comparison unfair. On models without property violations, however, all algorithms must exhaust the search space

and therefore will do a comparable amount of search.

Spin provides many parameters that may be tweaked to tune the search performance for different models. We compiled the multi-core depth-first with the

```
-DFULL_TRAIL -DSAFETY -DMEMLIM=64000
```

options on all models. For each individual model we also used any additional parameter settings that were recommended by Spin after running with the default parameter set.

3.5.2 Effect of Delaying Duplicate Detection

To compare the effects of the immediate duplicate detection of PSDD with the delayed duplicated detection of HD-BFS we looked at the memory usage of the two algorithms. Our hypothesis was that HD-BFS would require more memory in order to store duplicate search nodes during communication before they can be checked against the hash table by the receiving thread. The model that we choose for this experiment is a model of the dining philosophers problem with 10 philosophers. The model is constructed to avoid the classic deadlock situation and therefore the entire search space will be exhausted by the search algorithms. This removes the effects of tie-breaking that may be encountered when searching a model that contains an error. Also, with this model, the same number of states are expanded by all algorithms regardless of whether or not partial-order reduction is used and therefore we can conclude that any difference in memory usage must be attributed to immediate detection of duplicate nodes or lack thereof.

Figure 3-2 shows the memory usage reported by Spin for the 10 philosophers problem. The x axis gives the number of threads from 1–12 and the y axis shows the number of gigabytes used to complete the search. Each line represents the mean of five runs at each thread count and the error bars (which are so tight that they are hardly even visible in this plot) show 95% confidence intervals on the mean. Breadth-first search only uses a single thread but we have extended the line for its single threaded performance across the x axis to ease comparison.

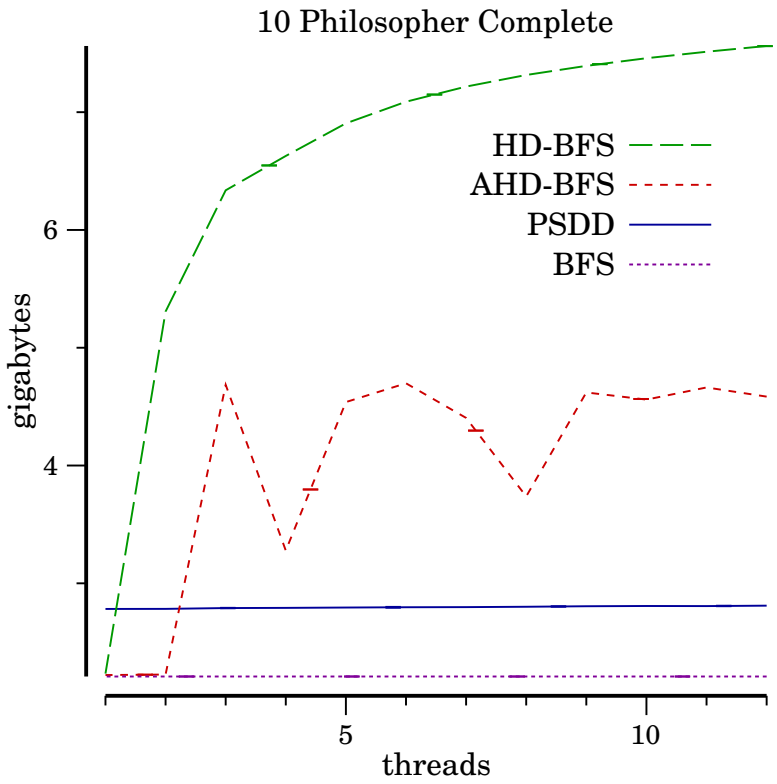


Figure 3-2: Memory usage of PSDD, HD-BFS, AHD-BFS and BFS.

From this figure, we can see that breadth-first search and PSDD both used less than 3 gigabytes of memory. The memory usage for PSDD remained nearly constant in the number of threads that performed the search. HD-BFS, however, required significantly more memory on this model when run with more than a single thread. The amount of memory required by HD-BFS increased sharply for up to six threads where it begun to even out. As mentioned above, this can be attributed to the fact that HD-BFS was required to store duplicate nodes in memory during communication instead of detecting them immediately. Due to the reduction in inter-thread communication, AHD-BFS required less memory than HD-BFS, however it still required more memory than breadth-first search and PSDD for more than a two threads.

In addition to the results shown here, we have observed that HD-BFS required a lot more memory on all of the models that we have used in our experiments. Presumably,

this is because of duplicate nodes, however, for other models the conservative partial-order reduction may also be a factor as we will see next.

3.5.3 Effect of Conservative Partial-Order Reduction

To evaluate the performance degradation that hash-distributed search and Spin's multi-core depth-first search suffer from due to conservative partial-order reduction we performed an experiment using a model of the semaphore implementation from the *Plan 9 from Bell Labs* operating system (Plan 9, Pike, Presotto, Dorward, Flandrena, Thompson, Trickey, & Winterbottom, 1995)[3]. The model is of particular interest because, unlike the philosopher model used in the previous experiments, the semaphore model was taken from a real-world model checking problem. Partial-order reduction is able to reduce the size of the state space of this model by approximately a factor of three, so failure to perform the full reduction has a significant impact on performance.

Figure 3-3 shows the number of states expanded (left) and the amount of memory used (right) by PSDD, HD-BFS, breadth-first search and Spin's multi-core depth-first search on the semaphore model with four separate processes contending for the semaphore. The format of the plot is the same as that of Figure 3-2. We can see that breadth-first search expanded the fewest nodes and used the least amount of memory in order to exhaust the configuration space of this model. PSDD expanded only slightly more nodes than breadth-first search and approximately the same amount of memory. The reason that PSDD and breadth-first search expanded slightly different numbers of nodes is that they may expand nodes within the same depth layer in a different order. This difference in tie-breaking can have a small effect on the partial-order reduction by slightly increasing or decreasing the number of nodes that must be expanded.

With a single thread, HD-BFS expanded about the same number of nodes and used about the same amount of memory as breadth-first. As the number of threads was increased, however, the number of expansions and memory requirement of HD-BFS rapidly increased.

[3]The model was available from `http://swtch.com/spin/`

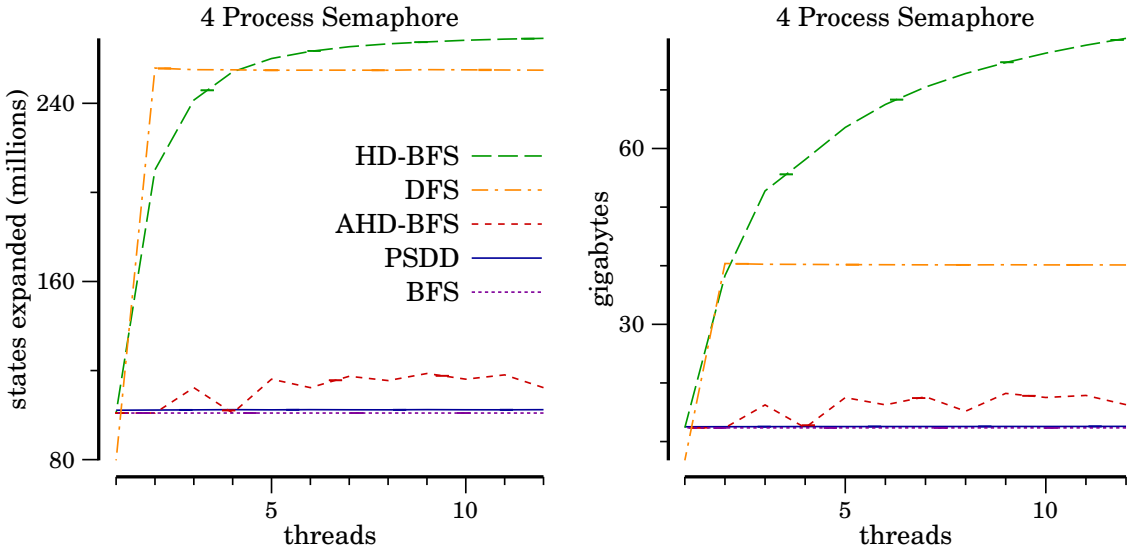


Figure 3-3: States expanded and memory used by PSDD, HD-BFS and BFS.

HD-BFS required almost 80GB of memory when run with 12 threads. The reason for the steep increase is that HD-BFS required more communications as the nodes were divided up between more threads. Each time a node is communicated the search conservatively assumed that it could not perform partial-order reduction and therefore many redundant paths were explored that were not pursued by the other two algorithms. The plot also shows this same effect happens with Spin's multi-core depth-first search. The depth-first search suffers from the same conservative partial-order reduction as HD-BFS and for more than a single thread it expanded many more states than PSDD and breadth-first search.

3.5.4 Overall Performance

Next we show the overall performance in terms of parallel speedup and wall-clock time for the different algorithms on four models. For PSDD and AHD-BFS, which both require an abstraction, we choose the fixed subset of processor IDs used in the projection experimentally. For each model we ran the algorithms using a small set of hand-chosen process ID sequences from 0–n and 1–n for small values of n (up to 7). The sequence that gave the best performance for each model was used in the following comparisons. We believe that

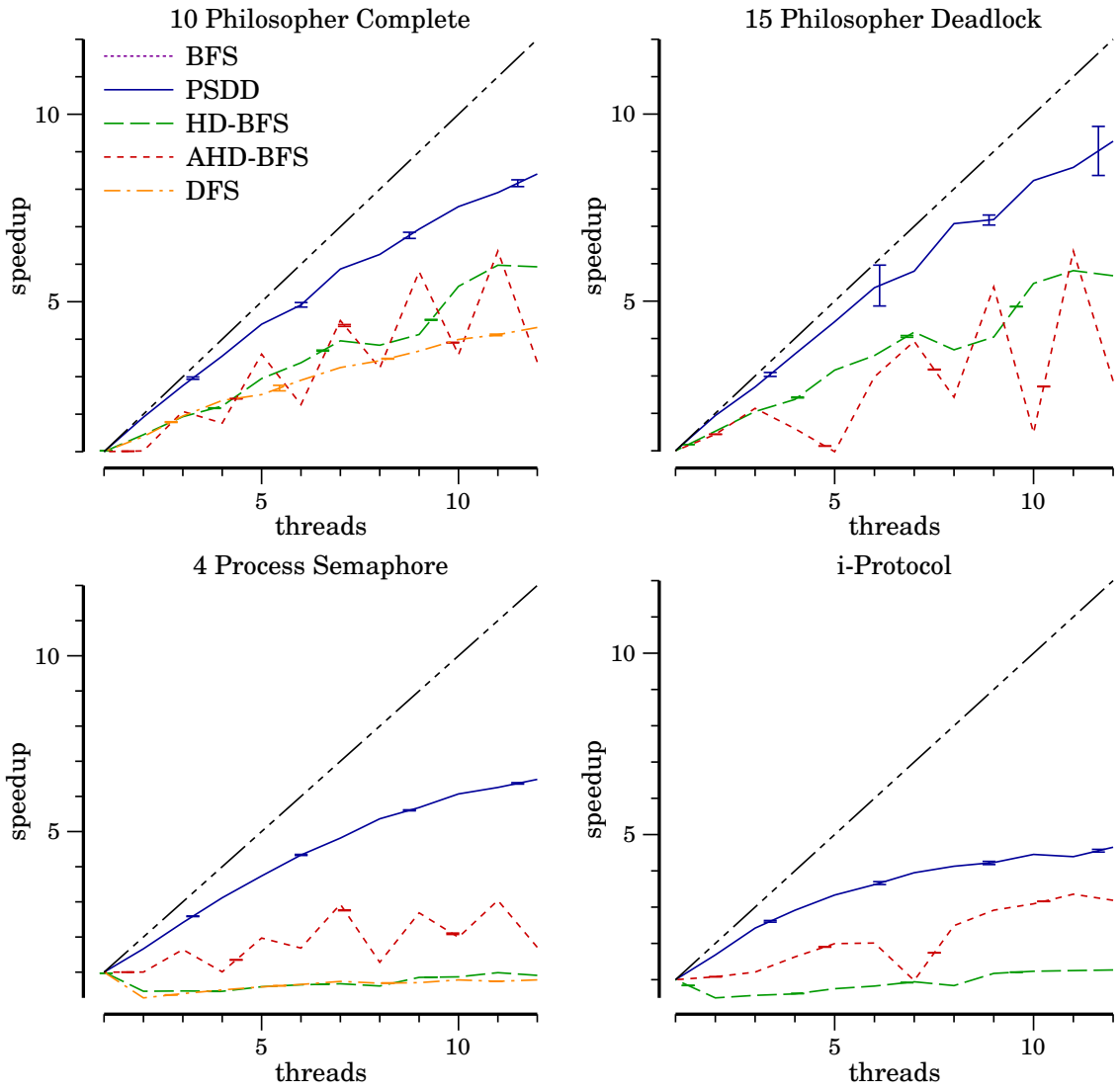


Figure 3-4: Parallel speedup for PSDD, HD-BFS, AHD-BFS, and parallel depth-first search.

the good performance exhibited by PSDD in the following results when using such a simple abstraction is strong evidence that finding a good abstraction for PSDD is not a difficult task.

Figure 3-4 shows the parallel speedup and Figure 3-5 shows the total wall-clock time that the algorithms required to search four different models using 1–12 threads. As in the previous plots, each line shows the mean performance across five runs with error bars giving the 95% confidence intervals. The x axis show the number of threads used from 1–12 and in Figure 3-5 the performance of breadth-first search is drawn across the x axis of each

plot even though it was only run serially. The models used were the dinning philosopher problem with 10 philosophers and no deadlock, the dining philosophers problem with 15 philosophers and a deadlock which is reachable in 42 steps, the Plan 9 semaphore with 4 contending processes, and the 0-level abstraction of the GNU i-Protocol model from Dong *et al.* (Dong, Du, Holzmann, & Smolka, 2003)[4] which contains a live-lock that is reachable in 72 steps, modified to avoid rendezvous as Spin complains that, on this model, rendezvous do not maintain completeness with breadth-first search. Spin's multi-core depth-first search algorithm is not shown on the 15 philosopher model or the i-Protocol model because they both exhibit errors for which depth-first search does not find shortest counter-examples and therefore does not perform a comparable amount of search.

Figure 3-4 shows the parallel speedup of PSDD, HD-BFS, AHD-BFS and depth-first search, computed as the single-threaded time divided by the time required for the number of threads given on the x axis. Speedup is perhaps one of the most important metrics when comparing parallel algorithms as it is indicative of how well the algorithm will perform as the parallelism increases. The diagonal line in each of the speedup plots shows perfect linear speedup which is typically unachievable in practice, however, it can provide a useful reference. The closer that the performance of an algorithm is to the diagonal line, the closer that its performance is to a perfect linear speedup. We can see from these figures that PSDD came the closest to linear speedup on all of these models; it always provided better speedup than the other parallel algorithms.

Figure 3-5 shows the wall-clock time, that is the actual time in seconds, required by each algorithm for the four models. We can see from this figure that for greater than three threads, PSDD was able to solve all of these models more quickly than the other algorithms. On the two "real-world" models, the semaphore and i-Protocol models, HD-BFS actually required more time than serial breadth-first search when using more than a single thread. This is because its conservative use of partial-order reduction caused it to search a much larger graph (c.f., Figure 3-3). Spin's multi-core depth-first search also suffered from this

[4]Available from `http://www.cs.sunysb.edu/~lmc/iproto/`

same issue, however, it seems to have made better use of parallelism and with greater than four threads it was faster than serial breadth-first search on the semaphore model. Finally, we can see that AHD-BFS gives very erratic performance across different numbers of threads. We attribute this to poor load balancing among the threads due to use of the abstraction instead of uniform node distribution.

3.5.5 External-Memory PSDD

Our results have demonstrated that PSDD requires less memory on model checking problems than hash-distributed search and it gives better parallel speedup and faster search times than both hash-distributed search and Spin's multi-core depth-first search. PSDD is also able to act as an external-memory search algorithm where external storage such as a hard disk is used to supplement core memory. In fact, the PSDD framework was originally developed by Zhou *et al.* for external-memory search (Zhou & Hansen, 2004). External-memory PSDD (external PSDD for short, Zhou & Hansen, 2007) works just like PSDD, however, when an abstract node is not in use by one of the threads, it can be pushed off to external storage. This reduces the memory usage of the search algorithm from that of the entire search graph to just the size of the duplicate detection scopes acquired by each thread.

As a preliminary experiment, we implemented external PSDD in Spin and used it to solve the deadlock-free 10 philosophers model. We ran on a machine with eight cores and four disks configured in a RAID 0 array. A limitation of our setup was that I/O operations were serialized via a single disk controller, therefore when using all eight cores external PSDD did not benefit from parallelism. When using a single thread, standard PSDD used an average of 233 seconds to complete its search and external PSDD required 1,764 seconds on average (both times had very little variance). With a more sophisticated machine, external PSDD will show improved performance when using parallelism, for example, Zhou and Hansen (2007) show performance improvements for up to four threads with external PSDD for automated planning. Even given this limitation with our experimental setup, the real

benefit of external PSDD is still realized: external PSDD was able to reduce the memory usage of search from 2.5 Gigabytes with standard PSDD down to around 0.5 gigabytes when using a single thread. This is a 500% reduction in the memory requirement of the search. In many cases this reduction in the memory requirement is much more important than reducing the search time because it is easier to wait longer for the search to complete, however, it may not be possible to add more memory. So, the memory requirement is often the limiting factor determining whether or not a model can be validated with a model checker.

3.6 Discussion

In a preliminary experiment we have seen that external-memory PSDD is able to reduce the memory requirement of search by a substantial amount. The penalty for external-memory PSDD, however, is that it can take a lot longer than serial search as it has to access hard disk storage. We suspect that the performance of external PSDD can be increased substantially by using multiple RAID arrays in order to exploit parallelism.

In our current implementation, external PSDD uses more memory when run with more than a single thread as each thread must have its own duplicate detection scope in RAM. With eight cores, external PSDD used around the same amount of memory as standard PSDD which does not use hard disk storage at all. A new technique called edge partitioning (Zhou et al., 2010) may be able to fix this problem. Edge partitioning reduces the size of a duplicate detection scope to be only those search nodes that map to a single abstract node. This can be a very significant reduction that will enable external PSDD to use multiple threads while still having a very small memory footprint.

Until now, we have not discussed, in detail, how the chosen abstraction effects the performance of PSDD. For our experiments, the abstraction was selected by evaluating a small set of different abstractions on each model and choosing the one that gave the best performance. As we saw in Section 2.4.1 with PBNF, if the abstract graph is too small or is too strongly connected then PSDD can suffer as it will be unable to find a sufficient number

of disjoint scopes to search in parallel. We have found that the simple abstractions used in our experiments have provided a sufficient amount of parallelism. Recent work, however, has shown that PSDD can greatly benefit from a dynamic search space partitioning that changes the abstraction during search (Zhou & Hansen, 2011). By using dynamic partitioning, the algorithm would be able to select an abstraction that is more balanced, reducing the peak memory requirement of external search, and less connected, increasing its ability to exploit parallelism.

3.7 Conclusion and Future Work

We have compared two techniques for parallelizing the breadth-first search algorithm used to find deadlocks and safety property violations in model checking. Our results showed that Parallel Structured Duplicate Detection provides benefits over both hash-distributed search and Spin's multi-core depth-first search because it gives better parallel speedup and it requires significantly less memory. We have also demonstrated that external PSDD can reduce the memory requirements of model checking even further by taking advantage of cheap secondary storage such as hard disks. As CPU performance relies more on parallelism than raw clock speed, the techniques presented in this chapter enable model checking to better exploit the full capabilities of modern hardware.

Partial-order reduction is a widely used technique for tackling the state-space explosion problem found in model checking. However, combining it with parallelization techniques has been a challenge in the past. In this chapter, we show that not only PSDD is effective for parallel reachability analysis, but it also preserves the full power of Spin's partial-order reduction algorithm. As for future work, we will apply PSDD to other model checkers to show its generality and effectiveness in speeding up search with full partial-order reduction.

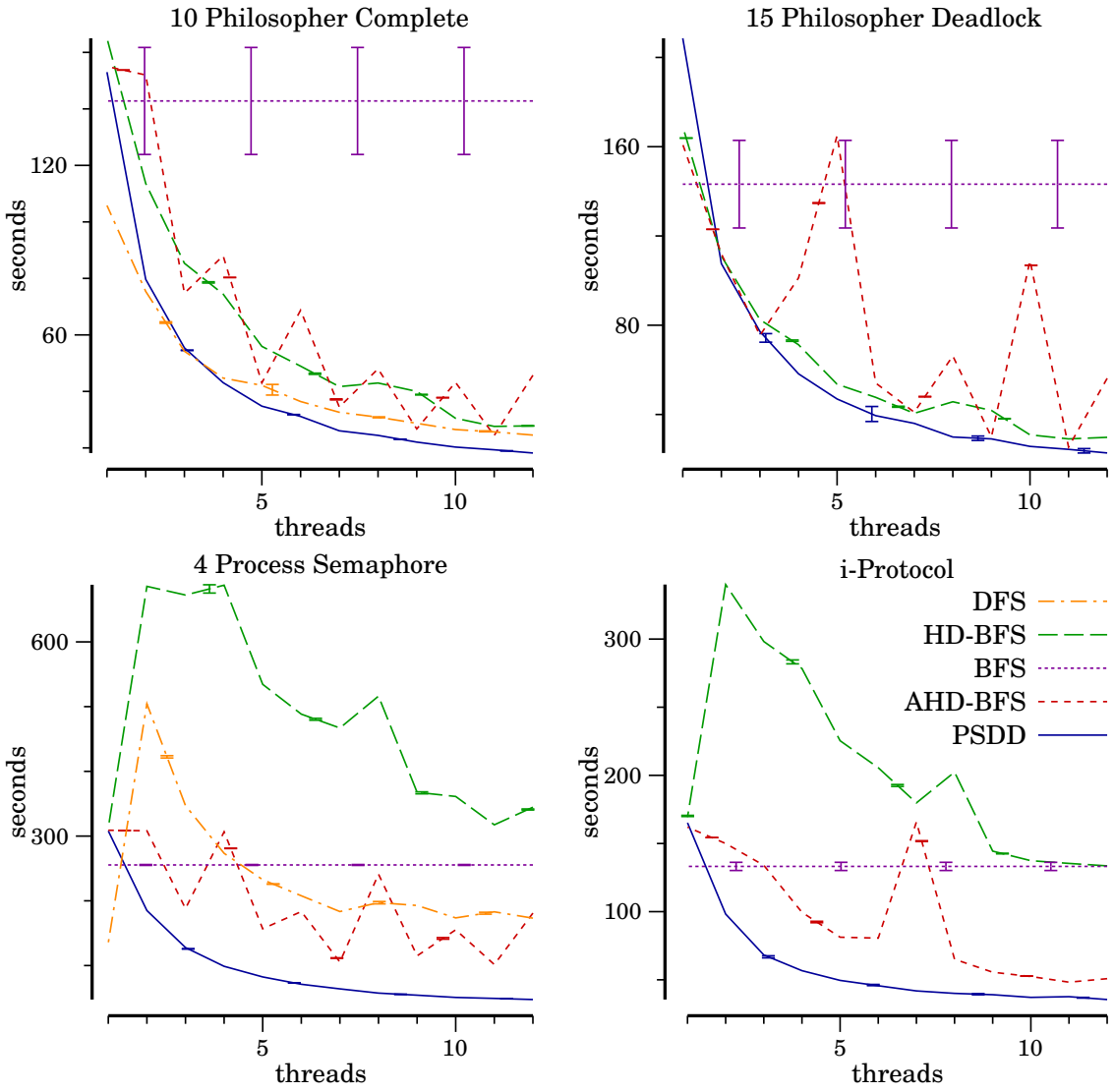


Figure 3-5: Wall-clock seconds for PSDD, HD-BFS, AHD-BFS, parallel depth-first search, and serial breadth-first search.

CHAPTER 4

PREDICTING SEARCH PERFORMANCE

4.1 Introduction

One major drawback of best-first search algorithms like as A* (Hart et al., 1968), is that they store every node that they generate. This means that for difficult problems in which many nodes must be generated, A* runs out of memory. If optimal solutions are required, iterative deepening A* (IDA*, Korf, 1985) can often be used instead. IDA* performs a series of depth-first searches where each search expands all nodes whose estimated solution cost falls within a given bound. As with A*, the solution cost of a node n is estimated using the value $f(n)$. After every iteration that fails to expand a goal, the bound is increased to the minimum f value of any node that was generated but not previously expanded. Because the heuristic estimator is defined to be a lower-bound on the true cost-to-go and because the bound is increased by the minimum amount, any solution found by IDA* is guaranteed to be optimal. Also, since IDA* uses depth-first search at its core, it only uses an amount of memory that is linear in the maximum search depth. Unfortunately, it performs poorly on domains with few nodes per f layer, because in that situation it will re-expand many interior nodes in order to expand only a very small number of new frontier nodes on each iteration.

One reason why IDA* performs well on classic academic benchmarks like the sliding tiles puzzle and Rubik's cube is that both of these domains have a geometrically increasing number of nodes in successive f layers. This means that each iteration of IDA* will re-expand not only all of the nodes of the previous iterations but also a significant number of new nodes that were previously out-of-bounds. Sarkar, Chakrabarti, Ghose, and Sarkar (1991) show that, in a domain with this geometric growth, IDA* will expand $\mathcal{O}(n)$ nodes

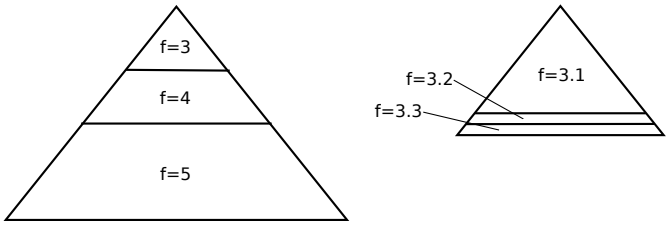


Figure 4-1: Geometric versus non-geometric growth.

where n is the number of nodes expanded by A* (the minimum required to prove a solution is optimal, short of tie-breaking nodes with cost equal to the optimal solution cost). They also show, however, that in a domain that does not exhibit geometric growth, IDA* may expand as many as $\mathcal{O}(n^2)$ nodes.

Figure 4-1 shows the two different behaviors graphically. The diagram on the left shows a tree with three f layers, each of an integer value and each successive layer adding a substantial portion of the tree, such that successive iterations of IDA* will each expand many new nodes that were not expanded previously. The right diagram in Fig 4-1, on the other hand, shows a tree with real-valued f layers. Each layer contains only a very small number of nodes, so IDA* will spend a majority of its time re-expanding nodes that it has expanded previously. Because domains with real-valued edge costs tend to have many distinct f values, they fall within this later category in which IDA* performs poorly.

The main contribution of this chapter is a new type of model that can be used to estimate the number of nodes expanded in an iteration of IDA*. As we will discuss in detail below, while the previous state-of-the-art approach to estimating search effort (Zahavi, Felner, Burch, & Holte, 2010) is able to predict the number of expansion with surprising accuracy in several domains, it has two drawbacks: 1) it requires a large amount of off-line training to learn the distribution of heuristic values and 2) it does not extend easily to domains with real-valued heuristic estimates. Our new model, which we call an *incremental model*, is able to predict as accurately as the current state-of-the-art model for the 15-puzzle when trained off-line. Unlike the previous approaches, however, our incremental model can also handle domains with real-valued heuristic estimates. Furthermore, our model may be trained on-line during a search. We show that our model can be used to control an IDA* search

by using information learned on completed iterations to determine a bound to use in the subsequent iteration. Our results show that our new model accurately predicts IDA* search effort. While IDA* guidance using our model tends to be expensive in terms of CPU time, the gain in accuracy allows the search to remain robust. Unlike the other IDA* variants which occasionally give very poor performance, IDA* using an incremental model is the only IDA* variant that can perform well over all of the domains used in our experiments. It also represents the first on-line use of detailed tree size prediction models to guide search.

4.2 Previous Work

Korf, Reid, and Edelkamp (2001) give a formula (henceforth abbreviated *KRE*) for predicting the number of nodes IDA* will expand with a given heuristic when searching to a given cost threshold. The KRE formula requires two components: the distribution of heuristic values in the search space and a function for predicting the number of nodes at a given depth in the brute-force search tree. They showed that off-line random sampling can be used to learn a sufficient estimate of the heuristic distribution. For their experiments, a sample size of ten billion states was used to estimate the distribution of the Manhattan distance heuristic on the 15-puzzle. Additionally, they demonstrate that a set of recurrence relations, based on a feature called the *type* of a node, can be used to compute the number of nodes at a given depth in the brute-force search tree for the tiles puzzle and Rubik's cube. The results of the KRE formula using these two techniques gave remarkably accurate predictions when averaged over a large number of initial states for each domain.

Zahavi et al. (2010) provide a further refinement to the KRE formula called Conditional Distribution Prediction (CDP). The CDP formula replaces the heuristic distribution in the KRE formula with one that is conditioned on a set of features, such as the heuristic value and type of the parent and grandparent of each node. This extra information enables CDP to make predictions for individual initial states and to extend to domains with inconsistent heuristics. Using CDP, Zahavi et al. show that substantially more accurate predictions can be made on the sliding tiles puzzle and Rubik's cube given different initial states with the

same heuristic value.

While the KRE and CDP formulas are able to give accurate predictions, their main drawback is that they require copious amounts of off-line training to estimate the heuristic distribution in a state space. Not only does this type of training take an excessive amount of time but it also does not make use of any instance-specific information. In addition, the implementation of these formulas as specified by Zahavi et al. (2010) assumes that the heuristic estimates have integer values so that they can be used to index into a large multi-dimensional array. Many domains have real-valued edge costs and therefore these techniques are not directly applicable in those domains.

4.2.1 Controlling Iterative Search

The problem of node re-expansion in IDA* in domains with many distinct f values is well known and has been explored in past work. Vempaty, Kumar, and Korf (1991) present an algorithm called DFS* that is similar to IDA*, however, it increases the cost bound between iterations more liberally. DFS* performs branch-and-bound on its final iteration so that it can find a provably optimal solution. While the authors describe a sampling approach to estimate the bound increase between iterations, in their experiments, the bound is simply increased by doubling.

Wah and Shang (1995) present a set of three linear regression models to control an IDA* search. Unfortunately, this technique requires intimate knowledge of the growth properties of f in a domain for which it will be used. In many settings, such as domain-independent planning, this knowledge is not available in advance.

IDA* with Controlled Re-expansion (IDA*$_{CR}$, Sarkar et al., 1991) uses a more liberal bound increase as in DFS*, however to determine its next bound, it uses a simple model. During an iteration of search, the model tracks the number of nodes that have each out-of-bound f value in a fixed-size histogram. Histograms are an appropriate choice over alternative techniques because fixed-size histograms provide constant-time operations whereas other methods, such as kernel density estimation, take linear time in the number of

samples (which, in our case corresponds to the number of generated search nodes). When an iteration is complete, the histogram is used to estimate the f value that will double the number of nodes in the next iteration. The remainder of the search proceeds as in DFS*, by increasing the bound and performing branch-and-bound on the final iteration to guarantee optimality.

While IDA*$_{CR}$ is simple, the model that it uses to estimate search effort relies upon two assumptions about the search space to achieve good performance. The first is that the number of nodes that are generated outside of the bound must be at least as large as the number of nodes that were expanded. If there are an insufficient number of pruned nodes, IDA*$_{CR}$ sets the bound to the greatest pruned f value that it has seen. This value may be too small to significantly advance the search. The second assumption is that none of the children of the pruned frontier nodes of one iteration will fall within the bound on the next iteration. If this happens, then the next iteration may be much larger than twice the size of the previous. As we will see, this can cause the search to overshoot the optimal solution cost on its final iteration, giving rise to excessive search effort.

4.3 Incremental Models of Search Trees

To estimate the number of nodes that IDA* will expand when searching to a given cost threshold, one would ideally know the distribution of all of the f values in the search space. Assuming a consistent heuristic[1], all nodes with f values within the cost threshold will be expanded, so by using the f distribution one could simply find the bound for which the number of nodes with smaller f values matches the desired count. Our new incremental model estimates this distribution.

We will estimate the distribution of f values in two steps. In the first step, we learn a model of how the f values are changing from nodes to their offspring. In the second step,

[1]A heuristic is consistent when the change in the h value between a node and its successor is no greater than the cost of the edge between the nodes. If the heuristic is not consistent then a procedure called pathmax (Méro, 1984) can be used to make it consistent locally along each path traversed by the search.

we extrapolate from the model of change in f values to estimate the overall distribution of all f values. This means that our incremental model manipulates two main distributions: we call the first one the Δf distribution and the second one the f distribution. In the next section, we describe the Δf distribution and give two techniques for learning it: one off-line and one on-line. Then, in Section 4.3.2, we describe how the Δf distribution can be used to estimate the distribution of f values in the search space.

4.3.1 The Δf Distribution

The goal of learning the Δf distribution is to predict how the f values in the search space change between nodes and their offspring. The advantage of storing these Δf increment values instead of storing the f values themselves is that it enables our model to extrapolate to portions of the search space for which it has no training data, a necessity when using the model on-line or with few training samples. We will use the information from the Δf distribution to build an estimate of the distribution of f values over the search nodes.

The CDP technique of Zahavi et al. (2010) learns a conditional distribution of the heuristic value and node type of a child node c, conditioned on the node type and heuristic estimate of the parent node p, notated $P(h(c), t(c)|h(p), t(p))$. As described by Zahavi et al., this requires indexing into a multi-dimensional array according to $h(p)$ and so the heuristic estimate must be an integer value. Our incremental model also learns a conditional distribution, however in order to handle real-valued heuristic estimates, it uses the integer valued search-space-steps-to-go estimate d of a node instead of its cost-to-go lower bound, h. In unit-cost domains, d will often be the same as h, however in domains with real-valued edge costs they will differ. d is typically easy to compute while computing h (Thayer & Ruml, 2009). For example, it is often sufficient to use the same procedure as for the heuristic but with a cost of 1 for each action. The distribution that is learned by the incremental model is $P(\Delta f(c), t(c), \Delta d(c)|d(p), t(p))$, that is, the distribution over the change in f value between a parent and child, the child node type and the change in d estimate between a parent and child, given the distance estimate of the parent and the type of the parent node.

The only non-integer term used by the incremental model is $\Delta f(c)$. Our implementation uses a large multi-dimensional array of fixed-sized histograms (see Appendix C) over $\Delta f(c)$ values. Each of the integer-valued features is used to index into the array, resulting in a histogram of the $\Delta f(c)$ values. By storing counts, the model can also estimate the branching factor of the search space by dividing the total number of nodes with a given d and t by the total number of their offspring. This branching factor will be used below to estimate the number of successors of a node when estimating the f distribution.

Zahavi et al. (2010) found that it is often important to take into account information about the grandparent of a node for the distributions used in CDP; we also found this to be the case for the incremental model. To accomplish this, we combine the node types of the parent and grandparent into a single type. For example, on the 15-puzzle, if the parent state has the blank in the center and it was generated by a state with the blank on the side, then the parent type would be a *side–center* node.

Learning Off-line.

We can learn an incremental Δf model off-line using the same method as KRE and CDP. A large number of random states from a domain are sampled, and the children (or grandchildren) of each sampled state are generated. The change in distance estimate $\Delta d(c) = d(c) - d(p)$, node type $t(c)$ of the child node, node type $t(p)$ of the parent node, and the distance estimate $d(p)$ of the parent node are computed and a count of 1 is then added to the appropriate histogram for the (possibly real-valued) change in f, $\Delta f(c) = f(c) - f(p)$, between parent and child.

Learning On-line.

An incremental Δf model can also be learned on-line during search. Each time a node is generated, the $\Delta d(c)$, $t(c)$, $t(p)$ and $d(p)$ values are computed for the parent node p and child node c and a count of 1 is added to the corresponding histogram for $\Delta f(c)$, as in the off-line case. In addition, when learning a Δf model on-line, the *depth* of the parent node

in the search tree is also known. We have found that this feature greatly improves accuracy in some domains (such as the vacuum domain described below) and so we always add it as a conditioning feature when learning an incremental model on-line.

Learning a Back-off Model.

Due to data sparsity, and because the Δf model will be used to extrapolate information about the search space for which it may not have any training data, a back-off version of the model may be needed. A back-off model is one that is conditioned on fewer features than the original model. When querying the model, if there is no training data for a given set of features, the more general back-off model is consulted instead. When learning a model on-line, because the model is learned on instance-specific data, we found that it was only necessary to learn a back-off model that ignores the depth feature. When training off-line, however, we learn a series of two back-off models, first eliminating the parent node distance estimate and then eliminating both the parent distance and type.

4.3.2 The f Distribution

Our incremental model predicts a bound that will result in expanding the desired number of nodes for a given start state by estimating the distribution of f values of the nodes in the search space. To accomplish this, the model uses the estimated f value distribution of one search depth in concert with the Δf model to generate an estimate of the f value distribution for the next depth. By beginning with the root node, which has a known f value, our procedure simulates the expansions of each depth layer to incrementally compute estimates of the f value distribution at the next layer. The accumulation of these depth-based f value distributions is an estimation of the f value distribution over the search space.

To increase accuracy, the estimated distribution of f values at each depth is conditioned on node type t and distance estimate d. We begin our simulation with a model of depth 0, which is simply a count of 1 for $f = f(root)$, $t = t(root)$ and $d = d(root)$. Next, the Δf

model is used to find a distribution over Δf, t and Δd values for the offspring of the nodes at each combination of t and d values at the current depth. Recall that our incremental model stores $P(\Delta f(c), t(c), \Delta d(c) | d(p), t(p))$. Because we store Δ values, we can compute $d(c) = d(p) + \Delta d(c)$ and $f(c) = f(p) + \Delta f(c)$ for each child c with a parent p even if we have not seen nodes at this depth or f value before. This gives us an estimate of the number of nodes with each $\langle f, t, d \rangle$ combination at the next depth of the search.

Because the Δf values may be real numbers, they are stored as histograms by our Δf model. In order to add $f(p)$ and $\Delta f(c)$, we use a procedure called additive convolution (Ruml, 2002; Rose, Burns, & Ruml, 2011). The convolution of two histograms ω_x and ω_y, where ω_x and ω_y are functions from values to weights, is a histogram ω_z, in which the count for a resulting f value k is the count at a parent f value i times the count of Δf at value $k - i$, summed over all possible parent values i. More formally,

$$\omega_z(k) = \sum_{i \in Domain(\omega_x)} \omega_x(i) \cdot \omega_y(k - i) \tag{4.1}$$

By convolving the f distribution of a set of nodes with the distribution of the change in f values between these nodes and their offspring, we get the f distribution of the offspring.

Because the maximum depth of a shortest-path search tree is typically unknown, our simulation must use a special criterion to determine when to stop. With a consistent heuristic the f values of nodes will be non-decreasing along a path (Pearl, 1984). This means that the Δf values in the model will always be non-negative, and the f values generated during the simulation will never decrease between layers. As soon as the accumulated f distribution has a weight that is greater than or equal to our desired node count, the maximum f value in the histogram can be fixed as an upper bound; selecting a greater f value can only give more nodes than desired. As the simulation proceeds and the weight in our accumulation histogram increases, we re-estimate the f value that gives our desired count. This upper bound will continue to decrease, and the simulation will estimate fewer and fewer new nodes within the bound at each depth. When the expected number of new nodes is smaller than some ϵ the simulation can stop. In our experiments we use $\epsilon = 10^{-3}$. Additionally, because the d value of a node can never be negative, we can prune all nodes

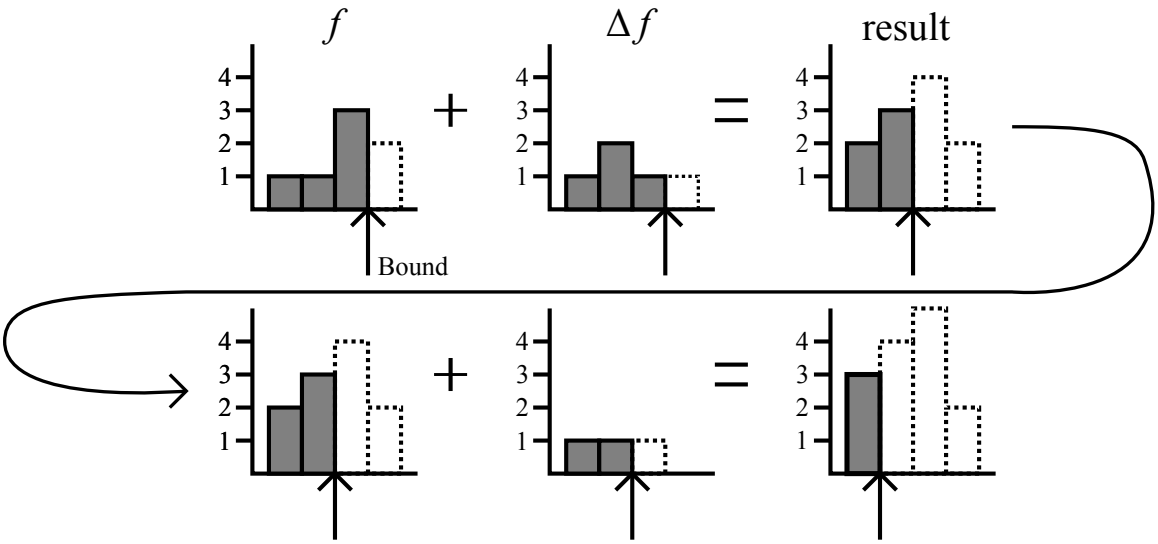


Figure 4-2: Histogram pruning.

that would be generated with $d < 0$.[2]

Example 1 *Figure 4-2 shows a graphical example of histogram pruning. In this example, the desired number of nodes is 3. In the top row, the left-most histogram shows an accumulation histogram with a total weight of 7 (1 from each of the left two bars, 3 from the 3rd bar and 2 from the right-most bar). An arrow is drawn showing that the pruning bound is just after the 3rd bar; this is the first bar at which the total weight (aggregating from the left) surpasses the desired amount. The right-most histogram in this row shows the result of adding new nodes to this accumulation. Because the height of each bar has increased, the pruning bound has shifted to the left: only the first two bars are required to get our desired count. The next row shows a subsequent addition to this accumulation. As the bound moves further and further to the left, the total non-pruned weight in each of the histograms being added to the accumulation (the distribution of new nodes introduced at the current depth) will continue to decrease. When the count in one of these histograms is finally less than ϵ, the simulation will stop.*

Figure 4-3 shows the pseudo-code for the procedure that uses the Δf distribution to

[2]This could also be used to estimate the solution cost.

```
SIMULATE(bound, desired, depth, accum, nodes)
 1. nodes' = SIMEXPAND(depth, nodes)
 2. accum' = add(accum, nodes − nodes')
 3. bound' = find_bound(accum', bound, desired)
 4. if weight_left_of(bound', nodes − nodes') > ε
 5.    depth' = depth + 1
 6.    SIMULATE(bound', desired, depth', accum', nodes')
 7. else return accum'

SIMEXPAND(depth, nodes)
 8. nodes' = new 2d histogram array
 9. for each t and d with weight(nodes[t, d]) > 0 do
10.    fs = nodes[t, d]
11.    SIMGEN(depth, t, d, fs, nodes')
12. return nodes'

SIMGEN(depth, t, d, fs, nodes')
13. for each type t' and Δd
14.    Δfs = delta_f_model[t', Δd, d, t]
15.    if weight(Δfs) > 0 then
16.       d' = d + Δd
17.       fs' = convolve(fs, Δfs)
18.       nodes'[t', d'] = add(nodes'[t', d'], fs')
19. done
```

Figure 4-3: Pseudo code for the simulation procedure used to estimate the f distribution.

estimate the f distribution. The entry point is the recursive function SIMULATE, which has the following parameters: the cost bound, desired number of nodes, the current depth, a histogram that contains the accumulated distribution of f values so far and a 2-dimensional array of histograms that stores the conditional distribution of f values among the nodes at the current depth. SIMULATE begins by simulating the expansion of the nodes at the current depth (line 1). The result of this is the conditional distribution of f values for the nodes generated as offspring at the next depth. These f values are accumulated into a histogram of all f values seen by the simulation thus far (line 2). An upper bound is determined (line 3), and if more than ϵ new nodes are expected to be in the next depth then the simulation continues recursively (lines 4–6), otherwise the accumulation of all f values is returned as the final result.

The SIM-EXPAND function is used to build the conditional distribution of the f values for the offspring of the nodes at the current simulation depth. For each node type t and distance

estimate d for which there exist nodes at the current depth, the SIM-GEN function is called to estimate the conditional f distribution of their offspring (lines 9–11). SIM-GEN uses the Δf distribution (line 14) to compute the number of nodes with each f value generated from parents with the specified combination of type and distance-estimate. Because this distribution is over Δf, t and Δd, we have all of the information that is needed to construct the conditional f distribution for the offspring (lines 16–18).

We have shown how to learn Δf, and use it to estimate the f distribution in a search tree. Now we turn to how an incremental model can be used to predict and control the performance of an IDA* search.

4.4 IDA*$_{IM}$

As we will see in Section 4.5.1, the incremental model can be used off-line for such tasks as distinguishing the more difficult of two search problems, a task that can be useful for automatically finding good heuristics (Haslum, Botea, Helmert, Bonte, & Koenig, 2007). One of the main features of the incremental model, however, is its ability to learn on-line, during search. In this section, we introduce IDA*$_{IM}$, a variant of IDA* that uses the incremental model to control how it increases bounds between iterations.

Like IDA* and IDA*$_{CR}$, IDA*$_{IM}$ uses a cost-bounded depth-first search. During each iteration, the Δf portion of an incremental model is learned as described in Section 4.3.1. When an iteration completes without finding a goal, the simulation procedure from Section 4.3.2 estimates a new bound that is expected to yield twice the number of expansions as the previous iteration. Given a good prediction, IDA*$_{IM}$ will increase the bound enough so that it does not degenerate into the $\mathcal{O}(n^2)$ worst-case of IDA*, and by a small enough amount that it does not greatly over-shoot the optimal cost and perform an extremely large final iteration.

With this approach, the goal may be found on an iteration where IDA*$_{IM}$ used a bound greater than the optimal cost. Like DFS* and IDA*$_{CR}$, IDA*$_{IM}$ uses branch-and-bound to complete its final iteration, expanding all nodes with f less than the current incumbent

solution cost. If a cheaper goal is found, it becomes the new incumbent, and when all nodes with f less that the incumbent cost have been expanded, the search terminates with the optimal solution.

4.4.1 Implementation Details

Due to limited precision in the histograms and inaccuracies in the model, we found that IDA*$_{IM}$ occasionally estimates a bound for the subsequent iteration that is too low, i.e., the same or smaller than the previous bound. This tends to happen either very early in the search when the model has been trained on very few expansions or very late in the search when the histogram's precision becomes overly restrictive. In the latter case, the histogram size can be increased to increase its accuracy. In the former case, we track the minimum out-of-bound f value, just as IDA*, and use it as the minimum value for the next bound, ensuring that the search continues to progress.

Each iteration of IDA* search will expand a superset of the nodes expanded during the previous iteration. Instead of learning a new Δf model from scratch on each depth-first search, we learn one model and simply update it whenever a new node is encountered that was not generated on a previous iteration. We accomplish this by tracking the bound used in the previous iteration and only updating the model when expanding a node that would have been pruned by the previous bound. Additionally, the search spaces for many domains form graphs instead of trees. In domains with many cycles, it is beneficial to perform cycle checking—backtracking when the current node is one of the ancestors along the current path. In domains where this optimization is appropriate, our implementation uses a hash table of all of the nodes along the current path to locate and prune cycles. In order for our model to take this extra pruning into account, we only train the model on the successors of a node that do not create cycles.

As an iterative deepening search progresses, some of the shallower depths become *completely expanded*: no nodes are pruned at that depth or any shallower depth. All of the children of nodes in a completely expanded depth are *completely generated*. When learning

the Δf distribution on-line, our incremental model has the exact *depth*, d and f values for all of the layers that have been completely generated. To speedup computation and improve accuracy, we "warm start" the simulation by initializing it with the perfect information for completed layers and begin it at the deepest completed depth instead of beginning at the root node. In some domains, such as the sliding tile domain, there are very few completed depth layers and warm starting has little effect. In other domains, such as the vacuum maze domains described in Section 4.5.2, many depth layers are completed during search and warm starting is extremely beneficial.

4.4.2 Theoretical Evaluation

In this section, we evaluate the theoretical properties of IDA*$_{IM}$.

Theorem 6 *IDA*$^*_{IM}$ *is sound.*

Proof: IDA*$_{IM}$ never returns a solution that did not pass the goal test, therefore it will never return a non-solution. □

Theorem 7 *Given an admissible heuristic, IDA*$^*_{IM}$ *is complete: if a solution exists then IDA*$^*_{IM}$ *will find it.*

Proof: Suppose that there is a solution, it must have some cost, say c. IDA*$_{IM}$ always increases the cost bound between iterations, because it sets each subsequent bound to no less than the minimum out-of-bound f value. So, c will eventually be within the bound, and since the heuristic never overestimates, every node on the path to the goal will also be within the bound and the goal will be expanded. □

To prove the optimality of IDA*$_{IM}$, we use two helpful lemmata.

Lemma 2 *There always exists a deepest node along each optimal path to the goal that has been generated via an optimal path—it has an optimal g value.*

Proof: The proof is by induction on node expansions. To begin, the root is the first node along all optimal paths and is surely generated. At any point in the search, we have a

deepest node expanded on each optimal path by the inductive hypothesis, and each of its successors is either: 1) not on an optimal path to a goal, 2) on an optimal path to a goal but has been generated via a suboptimal path or 3) is on an optimal path to some goal and was generated via an optimal path. In all three cases either the previous deepest node remains the deepest (1 and 2) or the newly generated node becomes the new deepest node on the path (3). Finally, the search never continues down a path after it expands a goal node. □

Lemma 3 *With a consistent heuristic (or one made consistent along each path from the root using pathmax) an iteration of IDA*$^*_{IM}$ *expands all nodes with f less than or equal to the current bound.*

Proof: For sake of contradiction, suppose that IDA*$_{IM}$ completes an iteration with bound b but does not expand a node n with $f(n) \leq b$. In order for n not to be expanded it must have an ancestor m for which $f(m) > b$. However, $b \geq f(n) \geq f(m)$ since consistency implies that f strictly increases along each path (Pearl, 1984). □

Theorem 8 *Given an admissible heuristic, IDA*$^*_{IM}$ *is optimal: if a solutions exists then IDA*$^*_{IM}$ *will find the cheapest solution.*

Proof: For the sake of contradiction, suppose that IDA*$_{IM}$ returns a suboptimal solution sol. Heuristic admissibility implies $h(\cdot) = 0$ for all goal nodes, so by the suboptimality of sol, $g(opt) < g(sol)$, where opt is an optimal solution. In order for sol to have been expanded, the final iteration of IDA*$_{IM}$ must have used some cost bound $b \geq g(sol)$. Due to Lemma 2, there must be some node, say p, that is the deepest node along an optimal path to opt that was expanded along an optimal path. When p was expanded along its optimal path, the next node on this optimal path, q, must have been generated but not expanded. At this time, $f(q) \leq f(opt) = g(opt) < g(sol) \leq b$ due to heuristic consistency, and so q was within the bound and thus also expanded (Lemma 3): a contradiction. □

Lastly, we prove the correctness of the incremental model in an idealized setting.

Theorem 9 *Given an accurate Δf model, the admissible heuristic $h(\cdot) = 0$, and a uniform tree, i.e., a tree with a uniform branching factor and edge costs, additive convolution produces the correct f distribution for the next depth of the tree.*

Proof: Given a uniform tree, let $N(d)$ be the number of nodes and $\langle n_0, ..., n_{N(d)-1} \rangle$ be the nodes at depth d in the tree. Since the tree is uniform, each node $n_i, 0 \leq i < N(d)$ has children $\langle n'_{i,0}, ..., n'_{i,b-1} \rangle$ with costs c_j for $0 \leq j < b$ that are only dependent on the edge. Let $[p(\cdot)]$ be an indicator that has the value 1 when the predicate $p(\cdot)$ is true and 0 when $p(\cdot)$ is false (Graham, Knuth, & Patashnik, 1998), and let $delta_d(\cdot)$ be the Δf distribution for level d, i.e., it gives the number of successors of a node at level d with the given change in f. Since $h(\cdot) = 0$, $f(n'_{i,j}) - f(n_i) = c_j, 0 \leq i < N(d), 0 \leq j < b$. So, the number of nodes with f value of x is given by the equation:

$$\begin{aligned}
count_{d+1}(x) &= \sum_{i=0}^{N(d)-1} \sum_{j=0}^{b-1} [f(n_i) = x - c_j] \\
&= \sum_{j=0}^{b-1} \sum_{i=0}^{N(d)-1} [f(n_i) = x - c_j] \\
&= \sum_{j=0}^{b-1} count_d(x - c_j) \\
&= \sum_{\Delta f} \sum_{j=0}^{b-1} [c_j = \Delta f] \cdot count_d(x - \Delta f) \\
&= \sum_{\Delta f} delta_d(\Delta f) \cdot count_d(x - \Delta f)
\end{aligned}$$

This last formula is exactly the one computed by convolving the Δf and f distributions at depth d (c.f. Equation 4.1). □

We have presented a new variant of IDA* with attractive properties including the ability to make correct predictions in some ideal situations. We do not, however, have any guarantees about the accuracy of the model in the case of non-uniform search spaces; determining its usefulness in these cases is a matter for empirical study. In the next section, we assess the performance of our new algorithm in practice, and we show empirically that the model can give accurate predictions in a variety of domains.

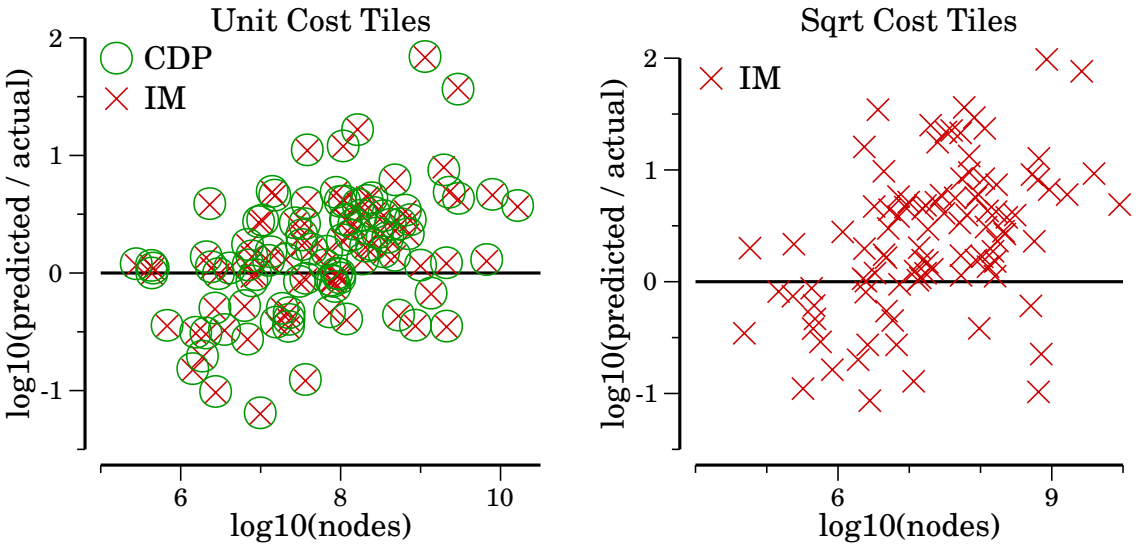


Figure 4-4: Off-line training accuracy when predicting node expansions.

4.5 Empirical Evaluation

In the following sections we show an empirical study of our new model and some of the related previous approaches. We begin by evaluating the accuracy of the incremental model when trained off-line. We then show the accuracy of the incremental model when used on-line to control an IDA* search.

4.5.1 Off-line Learning

We evaluate the quality of the predictions given by the incremental model when using off-line training by comparing the predictions of the model with the true node expansion counts. For each problem instance, the optimal solution cost is used as the cost bound. Because both CDP and the incremental model estimate all of the nodes within a cost bound, the true values are computed by running a full depth-first search of the tree bounded by the optimal solution cost. This search is equivalent to the final iteration of IDA*, assuming that the algorithm finds the goal node after having expanded all other nodes that fall within the cost bound.

Estimation Accuracy.

We trained both CDP (Zahavi et al., 2010) and an incremental model off-line on ten billion random 15-puzzle states using the Manhattan distance heuristic. We then compared the predictions given by each model to the true number of nodes within the optimal-solution-cost bound for each of the standard 100 15-puzzle instances due to Korf (1985). The left plot of Figure 4-4 shows the results of this experiment. The x axis is on a log scale; it shows the actual number of nodes within the cost bound. The y axis is also on a log scale; it shows the ratio of the estimated number of nodes to the actual number of nodes, we call this metric the *estimation factor*. The closer that the estimation factor is to one (recall that $\log_{10} 1 = 0$) the more accurate the estimation was. The median estimation factor for the incremental model was 1.435 and the median estimation factor for CDP was 1.465 on this set of instances. From the plot we can see that, on each instance, the incremental model gave estimations that were nearly equivalent to those given by CDP, the current state-of-the-art predictor for this domain.

To demonstrate our incremental model's ability to make predictions in domains with real-valued edge costs and with real-valued heuristic estimates, we created a modified version of the 15-puzzle where each move costs the square root of the tile number that is being moved. We call this problem the square root tiles puzzle and for the heuristic we use a modified version of the Manhattan distance heuristic that takes into account the cost of each individual tile.

As presented by Zahavi et al. (2010), CDP is not able to make predictions on this domain because of the real-valued heuristic estimates. The right plot in Figure 4-4 shows the estimation factor for the predictions given by the incremental model trained off-line on fifty billion random square root tiles states. The same 100 puzzle states were used. Again, both axes are on a log scale. The median estimation factor on this set of puzzles was 2.807.

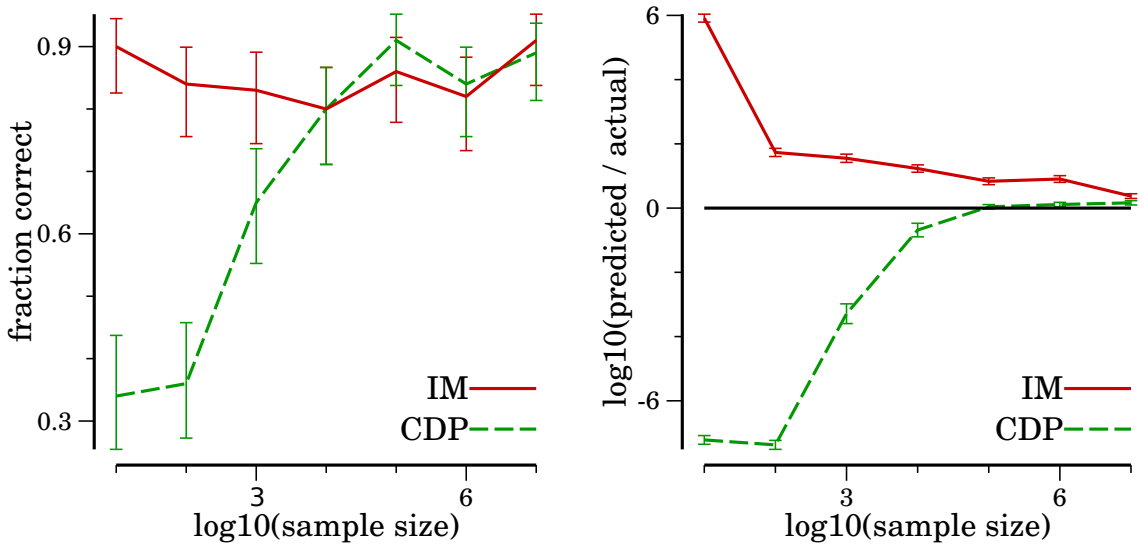


Figure 4-5: Off-line training accuracy when predicting the harder instance.

Small Sample Sizes.

Haslum et al. (2007) use a technique loosely based on the KRE formula to select between different heuristics for domain independent planning. When given a choice between two heuristic lower bound functions, we would like to select the heuristic that will expand fewer nodes. Using KRE (or CDP) to estimate node expansions requires a very large off-line sample of the heuristic distribution to achieve accurate predictions, which is not achievable in applications such as Haslum et al.'s. Since the incremental model uses Δ values and a back-off model, however, it is able to make useful predictions with very little training data. To demonstrate this, we created 100 random pairs of instances from Korf's set of 15-puzzles. We used both CDP and the incremental model to estimate the number of expansions required by each instance when given its optimal solution cost. We rated the performance of each model based on the fraction of pairs for which it was able to correctly determine the more difficult of the two instances.

The left plot in Figure 4-5 shows the fraction of pairs that were ordered correctly by each model for various sample sizes. Error bars represent 95% confidence intervals on the mean. We can see from this plot that the incremental model was able to achieve much higher accuracy when ordering the instances with as few as ten training samples. CDP

required 10,000 training samples or more to achieve comparable accuracy. The right plot in this figure shows the $\log_{10}$ estimation factor of the estimates made by each model. While CDP achieved higher quality estimates when given 10,000 or more training instances, the incremental model was able to make much more accurate predictions when trained on only 10, 100 or 1,000 samples.

4.5.2 On-line Learning

In this section, we evaluate the incremental model when trained and used on-line during an IDA* search. As described in Section 4.4, the IDA*$_{IM}$ algorithm sets the bound for the next iteration by consulting the incremental model to find a bound that is predicted to double the number of node expansions from that of the previous iteration. As we will see, because the model is trained on the exact instance for which it will be predicting, the estimations tend to be more accurate than the off-line estimations, even with a much smaller training set. In the following subsections, we evaluate the incremental model by comparing IDA*$_{IM}$ to the original IDA* (Korf, 1985) and IDA*$_{CR}$ (Sarkar et al., 1991).

Sliding Tiles.

The unit-cost sliding tiles puzzle is a domain where standard IDA* search works very well. The minimum cost increase between iterations is two and this leads to a geometric increase in the number of nodes between subsequent iterations.

The left panel of Figure 4-6 shows the median *growth factor*, the relative size of one iteration compared to the next, on the y axis, for IDA* , IDA*$_{CR}$ and IDA*$_{IM}$. Ideally, all algorithms would have a median growth factor of two. All three of the lines for the algorithms are drawn directly on top of one another in this plot. While both IDA*$_{CR}$ and IDA*$_{IM}$ attempted to double the work done by subsequent iterations, all algorithms still achieved no less than 5x growth. This is because, due to the coarse granularity of f values in this domain, no threshold can actually achieve the target growth factor. However, the median estimation factor of the incremental model over all iterations in all instances was

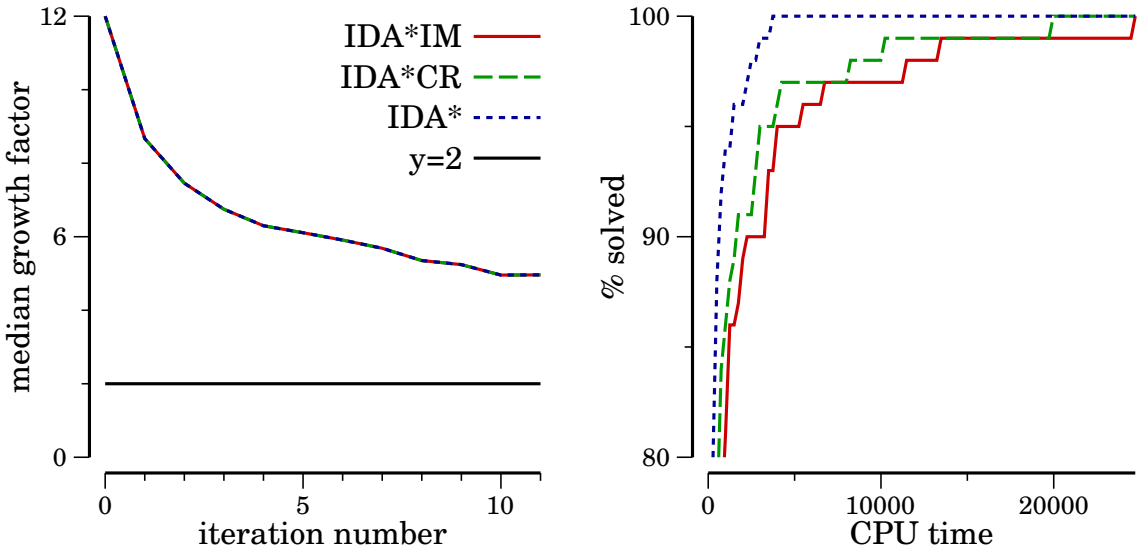


Figure 4-6: Unit tiles: growth rates and number of instances solved.

1.029. This is very close to the optimal estimation factor of one. So, while granularity of f values made doubling impossible, the incremental model still predicted the amount of work with great accuracy. The right panel shows the percentage of instances solved within the time given on the x axis. Because IDA*$_{IM}$ and IDA*$_{CR}$ must use branch-and-bound on the final iteration of search they are unable to outperform IDA* in this domain. It should also be noted that, because IDA*$_{IM}$ and IDA*$_{CR}$ expand the exact same number of nodes on these instances (they use the same bounds for all iterations, and both use branch-and-bound on the final iteration), the difference in their performance, as seen in this plot, also shows the overhead incurred in learning the incremental model.

Square Root Tiles.

While IDA* works well on the classic sliding tile puzzle, a trivial modification exposes its fragility: changing the edge costs. We examined the square root cost variant of the sliding tiles. This domain has many distinct f values, so when IDA* increases the bound to the smallest out-of-bound f value, it will visit a very small number of new nodes with the same f in the next iteration. We do not plot the results for IDA* on this domain because it gave extremely poor performance. IDA* was unable to solve any instances with a one hour

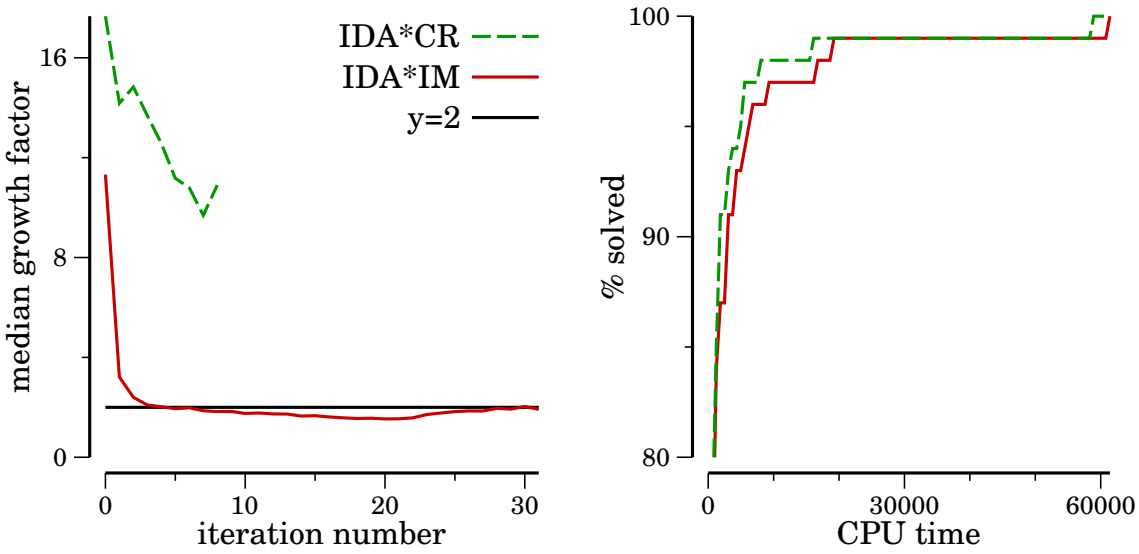


Figure 4-7: Square root tiles: growth rates and number of instances solved.

timeout and at least one instance requires more than a week to solve.

Figure 4-7 presents the results for IDA*$_{IM}$ and IDA*$_{CR}$. Even with the branch-and-bound requirement, IDA*$_{IM}$ and IDA*$_{CR}$ easily outperform IDA* by increasing the bound more liberally between iterations. While IDA*$_{CR}$ gave slightly better performance with respect to CPU time, its model was not able to provide very accurate predictions. The growth factor between iterations for IDA*$_{CR}$ was no smaller than eight times the size of the previous iteration when the goal was to double. The incremental model, however, was able to keep the growth factor very close to doubling. The median estimation factor was 0.871 for the incremental model which is closer to the optimal estimation factor of one than when the model was trained off-line. We conjecture that the model was able to learn information specific to the instance for which it was predicting.

One reason why IDA*$_{CR}$ was able to achieve competitive performance in this domain is because, by increasing the bound very quickly, it was able to skip many iterations of search that IDA*$_{IM}$ performed. IDA*$_{CR}$ performed no more than 10 iterations on any instance in this set whereas IDA*$_{IM}$ performed up to 33 iterations on a single instance. Although the rapid bound increase was beneficial in the square root tiles domain, in a subsequent section we will see that increasing the bound too quickly can severely hinder performance.

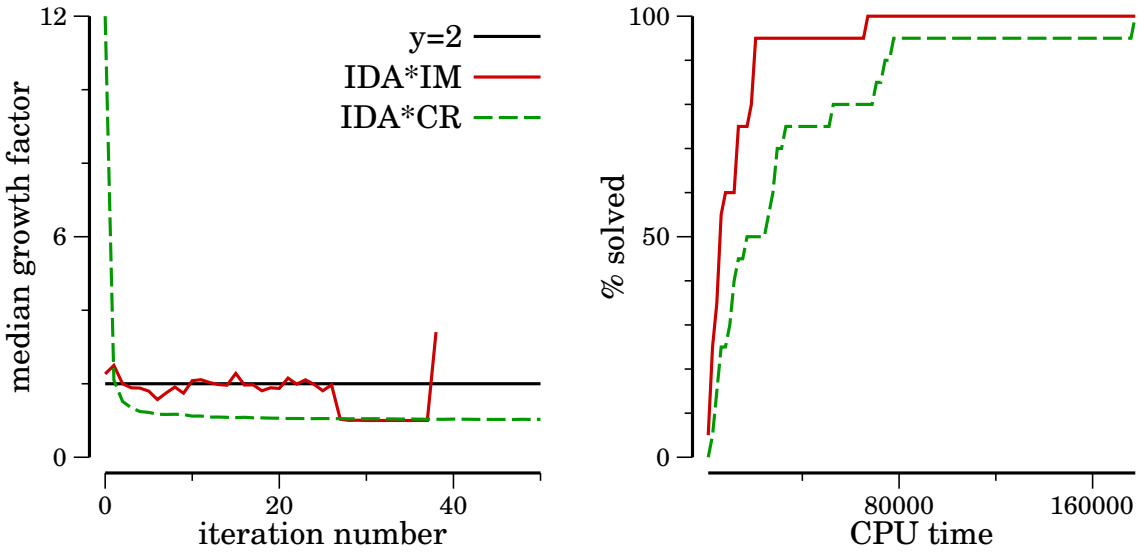


Figure 4-8: Vacuum maze: growth rates and number of instances solved.

Vacuum Maze.

The objective of the vacuum maze domain is to navigate a robot through a maze in order to vacuum up spots of dirt. In our experiments, we used 20 instances of 500x500 mazes that were built with a depth-first search. Long hallways with no branching were then collapsed into single edges with a cost equivalent to the hallway length. Each maze contained 10 pieces of dirt and any state in which all dirt had been vacuumed was a goal. The median number of states per instance was 56 million and the median optimal solution cost was 28,927. The heuristic was the size of the minimum spanning tree of the locations of the dirt and vacuum. The *pathmax* procedure (Méro, 1984) was used to make the f values non-decreasing along a path.

Figure 4-8 shows the median growth factor and number of instances solved by each algorithm for a given amount of time. Again, IDA* is not shown due to its very poor performance in this domain. Because there are many dead ends in each maze, the branching factor in this domain is very close to one. The model used by IDA*$_{CR}$ gave very inaccurate predictions and the algorithm often increased the bound by too small of an increment between iterations. IDA*$_{CR}$ performed up to 386 iterations on a single instance. With

the exception of a dip near iterations 28–38, the incremental model was able to accurately find a bound that doubled the amount of work between iterations. The dip in the growth factors may be attributed to histogram inaccuracy on the later iterations of the search. The median estimation factor of the incremental model was 0.968, which is very close to the perfect factor of one. Because of the poor predictions given by the IDA*$_{CR}$ model, it was not able to solve instances as quickly as IDA*$_{IM}$ on this domain.

While our results demonstrate that the incremental model gave very accurate predictions in the vacuum maze domain, it should be noted that, due to the small branching factor, iterative searches are not ideal for this domain. A simple implementation of Frontier A* (Korf, Zhang, Thayer, & Hohwald, 2005), an A*-like search that elides the use of a closed list, and performs well in domains with low branching factors, was able to solve each instance in this set in no more than 1,887 CPU seconds.

Uniform Trees.

We also designed a simple synthetic domain that illustrates the brittleness of IDA*$_{CR}$. We created a set of trees with 3-way branching where each node has outgoing edges of cost 1, 20 and 100. The goal node lies at a depth of 19 along a random path that is a combination of 1- and 20-cost edge and the heuristic $h = 0$ for all nodes. We have found that the model used by IDA*$_{CR}$ will often increase the bound extremely quickly due to the large 100-cost branches. Because of the extremely large searches created by IDA*$_{CR}$, we use a five hour time limit in this domain.

Figure 4-9 shows the growth factors and number of instances solved in a given amount of time for IDA*$_{IM}$, IDA*$_{CR}$ and IDA*. Again, the incremental model was able to achieve very accurate predictions with a median estimation factor of 0.978. IDA*$_{IM}$ was able to solve ten of twenty instances and IDA* solved eight within the time limit. IDA*$_{IM}$ solved every instance in less time than IDA*. IDA*$_{CR}$ was unable to solve more than two instances within the time limit. It grew the bounds in between iterations extremely quickly, as can be seen in the growth factor plot on the left in Figure 4-9.

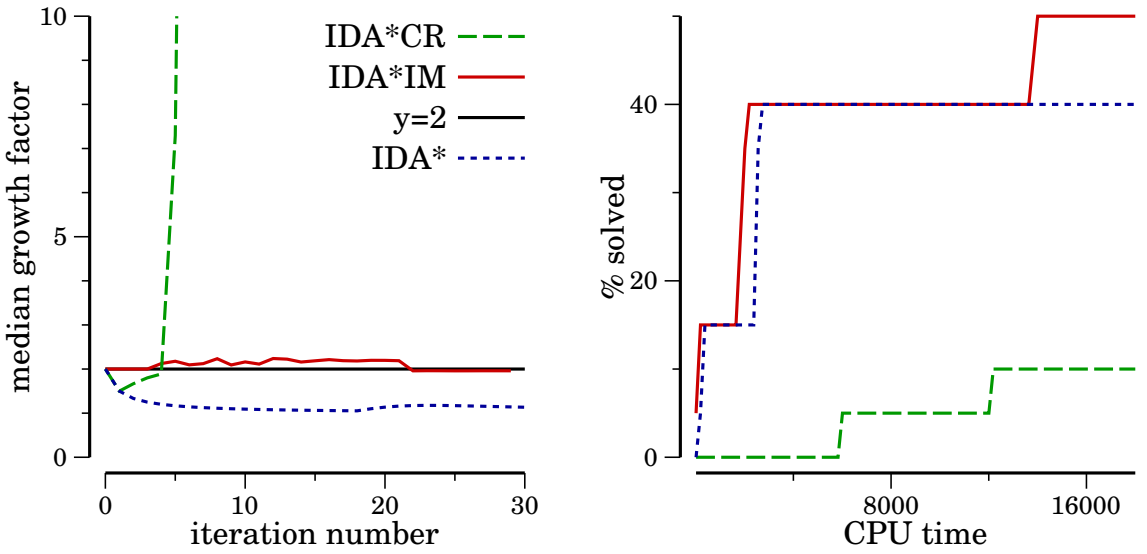


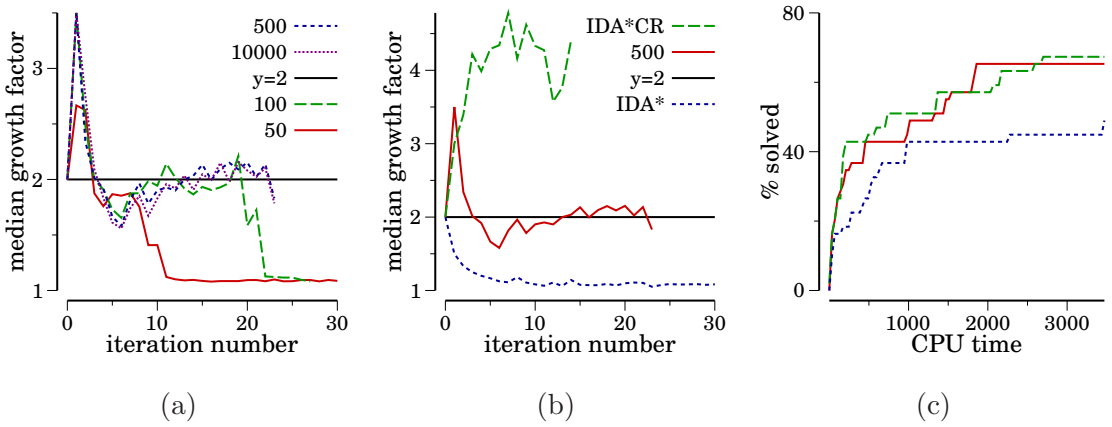
Figure 4-9: Uniform Tree: growth rates and number of instances solved.

Although IDA* tended to have reasonable CPU time performance in this domain, its growth factors were very close to one. The only reason that IDA* achieved reasonable performance is because expansions in this synthetic tree domain required virtually no computation. This would not be observed in a domain where expansion required any reasonable computation.

4.5.3 Effect of Histogram Size

Since the incremental model uses fixed-size histograms to represent distributions, it is useful to know the effect of the histogram size on the performance of the model. In our previous experiments we used a histogram size of 100. One limitation of the incremental model and thus IDA*$_{IM}$ is that its performance may rely on choosing a good histogram size. In this section we look at a domain for which the histogram size of 100 is too small.

In the *pancake problem*, a chef is given a stack of different size pancakes and a spatula. Unfortunately, the pancakes are all out of order, and to make for a beautiful presentation the chef must repeatedly stick his spatula into the stack and flip some of the top pancakes until the entire stack is sorted. At its core, the pancake problem is a permutation puzzle, where the goal is to sort an n-permutation using a sequence of prefix reversals. This domain



(a) (b) (c)

Figure 4-10: Different histogram sizes (a), IDA*, IDA*$_{CR}$, and IDA*$_{IM}$ growth rates (b) and number of instances solved (c) on the pancake problem.

differs from those of the previous sections because it has a fairly large branching factor; an n-pancake problem has a branching factor of $n - 2$ if you disallow flipping the very top pancake (which has no effect on the ordering) and reversing the previous flip.

We ran IDA*$_{IM}$ on a non-unit-cost variant of the pancake problem where each flip costs the sum of the numbers of the pancakes being flipped, simulating the idea that flipping more or larger pancakes costs more than flipping fewer or smaller pancakes. This variant of the problem is challenging for standard IDA* as there are many distinct f values. In this experiment, we used 50 random 12-pancake instances with a 7-pancake pattern database (Culberson & Schaeffer, 1998) for the heuristic. Our initial results showed that IDA*$_{IM}$ gave poor performance on this domain; as the search progressed, the estimations from the incremental model became very inaccurate. We hypothesized that the large branching factor in the pancake problem was adding too many values to the fixed-size histograms leading to inaccuracy in the estimates. To test our hypothesis we ran IDA*$_{IM}$ with different histogram sizes on the pancake puzzle. The growth rates for histogram sizes of 50, 100, 500, and 10,000 are shown in Figure 4-10a. With the histogram size of 50 the accuracy of the predictions from the model, and thus the growth rates of IDA*$_{IM}$ begun to significantly decrease just before 10 iterations. When the histogram size was increased to 100 the prediction accuracy begun to decrease at around 20 iterations. When we increased the histogram size to 500,

the drop off in accuracy went away and the growth factor remained around the correct value of 2 until the very final iterations of search. Finally, we increased the histogram size much further to 10,000, the performance was very similar to the performance of 500, including the slight estimation inaccuracy during the final few iterations.

Figures 4-10b and 4-10c compare IDA*, IDA*$_{CR}$, and IDA*$_{IM}$ with a histogram size of 500 on the pancake problem. As before we can see that IDA* performs poorly as it requires many iterations, each of which is very similar to the previous. IDA*$_{CR}$, once again grows its iterations quite quickly, about quadrupling each subsequent iteration. The incremental model, using sufficiently large histograms, is rather accurate at doubling the size of its iterations, however, its performance is similar to IDA*$_{CR}$ with respect to the number of instances solved in the time limit. We suspect that IDA*$_{IM}$ does not outperform IDA*$_{CR}$ in this domain due to its increased overhead in maintaining larger histograms.

4.5.4 Summary

When trained off-line, the incremental model was able to make predictions on the 15-puzzle domain that were nearly indistinguishable from CDP, the current state-of-the art. In addition, the incremental model was able to estimate the number of node expansions on a real-valued variant of the sliding tiles puzzle where each move costs the square root of the tile number being moved. When presented with pairs of 15-puzzle instances, the incremental model trained with 10 samples was more accurately able to predict which instance would require fewer expansions than CDP when trained with 10,000 samples.

The incremental model made very accurate predictions across all domains when trained on-line and when used to control the bounds for IDA*, our model made for a robust search. While each of the alternative approaches occasionally gave extremely poor performance, IDA* controlled by the incremental model achieved the best performance of the IDA* searches in the vacuum maze and uniform tree domains and was competitive with the best search algorithms for both of the sliding tiles domains and the pancake puzzle. This provides an example of how a flexible tree model can be used in practice.

4.6 Discussion

In search spaces with small branching factors such as the vacuum maze domain, the back-off model seems to have a greater impact on the accuracy of predictions than in search spaces with larger branching factor such as the sliding tiles domains. Because the branching factor in the vacuum maze domain is small, however, the simulation must extrapolate out to great depths (many of which the model has not been trained on) to accumulate the desired number of expansions. The simple back-off model used here merely ignored depth. While this tended to give accurate predictions for the vacuum maze domain, a different model may be required for other domains.

We have looked at using an incremental model for predicting the number of nodes that will be expanded within a given cost bound. Another possible use for the incremental model is estimating the optimal solution cost for a problem. Since the distance-to-go estimate d is only equal to zero for goal nodes, the simulation procedure described in Section 4.3 may be used to determine the smallest f value among all of the nodes with $d = 0$. This f value is an estimate of the optimal solution cost. This would extend the results presented by Lelis, Stern, and Jabbari (Lelis, Stern, & Jabbari Arfaee, 2011) to on-line learning and domains with real valued costs.

There has been much successful work in the area of exploiting instance-specific information for automatic algorithm selection and configuration (Xu, Hutter, Hoos, & Leyton-Brown, 2008; Malitsky, 2012; Arbelaez, Hamadi, & Sebag, 2010). In agreement with these results, we believe that instance-specific information is what allows the incremental model to make accurate predictions when trained online (cf Section 4.5.2, where our results show that the model was more accurate when trained online on each instance than when trained offline). Reactive and autonomous search (Battiti, Brunato, & Mascia, 2008; Hamadi, Monfroy, & Saubion, 2010), in which optimization algorithms attempt to tune themselves online, have been show to be quite effective for constraint programming and combinatorial optimization. IDA$^*_{IM}$ demonstrates that on-line learning can be used to control search for shortest-path problems too.

4.7 Conclusion

In this chapter, we saw a new incremental model for predicting the distribution of solution cost estimates in a search tree. This allowed us to estimate the number of nodes that bounded depth-first search will visit. This new model is comparable to state-of-the-art methods in domains where those methods apply. The three main advantages of the new model are that it works naturally in domains with real-valued heuristic estimates, it is accurate with few training samples, and it can be trained on-line. We saw that the model can lead to more accurate predictions when trained online. Additionally, we saw that the incremental model can be used to control an IDA* search, giving a robust algorithm, IDA*$_{IM}$. Given the prevalence of real-valued costs in real-world problems, on-line incremental models are an important step in broadening the applicability of iterative deepening search.

CHAPTER 5

PLANNING BEFORE EXECUTING

5.1 Introduction

In the previous chapters, we focused on search techniques that optimized cost (item 1 in Table 1-1). The algorithms presented Chapter 2 minimized the sum of planning and execution time by planning faster without increasing solution cost. We saw the generality of these ideas in Chapter 3 by extending them to model checking. Chapter 4, presented a new approach for estimating search effort required by algorithms like IDA* (Korf, 1985). The new incremental model was then used to control optimal search in domains with real-valued costs. In this chapter we turn to a new search objective: minimizing a utility function of search time and solution cost (item 3a in Table 1-1). With such a utility function a user can specify their preference between search time and solution cost, and the algorithm handles the rest. This objective function more directly addresses the problem of planning under time pressure.

We consider utility functions given as a linear combination of search time and solution cost. This is an important form of utility function, for two reasons. First, it is easily elicited from a user if not already explicitly in their application domain. For example, if cost is given in monetary terms, it is usually possible to ask how much time one is willing spend to decrease the solution cost by a certain amount. Second, if the solution cost is given in terms of time (i.e., the cost represents the time required for the agent to execute the solution), then this form of utility function can be used to optimize what we call *goal achievement time*; by weighting search time and execution time equally, a utility-aware search will attempt to minimize the sum of the two, thus attempting to behave such that the agent will achieve its goal as quickly as possible.

Existing techniques for this problem are based on anytime algorithms, a general class of algorithms that emit a stream of solutions of decreasing cost until converging on an optimal one. With sufficient knowledge about the performance profile of an anytime algorithm, which represents the probability that it will decrease its solution cost by a certain amount given its current solution cost and additional search time, it is possible to create a stopping policy that is cognizant of the user's preference for trading solving time for solution cost (Hansen & Zilberstein, 2001; Finkelstein & Markovitch, 2001). To the best of our knowledge, this technique has not previously been applied to heuristic search.

There are two disadvantages to using anytime algorithms to trade off solving time for solution cost. The first is that the profile of the anytime algorithm must be learned off-line on a representative set of training instances. In many settings, such as domain independent planning, the problem set is unknown, so one cannot easily assemble a representative training set. Also, it is often not obvious which parameters of a problem affect performance, and even if an instance generator is available, the problems that it generates may not represent those seen in the real world. The second issue is that, while the stopping policy is aware of the user's preference for time and cost, the underlying anytime algorithm is oblivious and will emit the same stream of solutions regardless of the desired trade off. The policy must simply do the best that it can with the solutions that are found, and the algorithm may waste a lot of time finding many solutions that will simply be discarded.

This chapter presents four main contributions. First, we combine anytime heuristic search with the dynamic programming-based monitoring technique of Hansen and Zilberstein (2001). To the best of our knowledge, we are the first to apply anytime monitoring to anytime heuristic search. Second, we present a very simple portfolio-based method that estimates a good parameter to use for a bounded-suboptimal search algorithm to optimize a given utility function. Third, we present BUGSY, a best-first search algorithm that does not rely on any off-line training, yet accounts for the user's preference between search time and solution cost. Finally, we present a set of experiments comparing the three techniques along with utility-oblivious algorithms such as A* and greedy best-first search.

The results of our experiments reveal two surprises. First, if a representative set of training instances is available, the most effective approach is the very simple technique of selecting a bound to use for a bounded-suboptimal search. Surprisingly, this convincingly dominates anytime algorithms with monitoring in many of our tests. Second, neither BUGSY or anytime search with monitoring dominates the other. BUGSY does not require any off-line training, yet surprisingly, BUGSY can perform as well as a method that uses training data. If a representative problem set is not available, then BUGSY is the algorithm of choice.

5.2 Background

In this section we briefly review suboptimal heuristic search, present some terminology used in the remainder of this chapter, and discuss the type of utility functions we are addressing.

5.2.1 Suboptimal Search

Greedy best-first search is a popular suboptimal search algorithm. It proceeds like A*, but orders its open list only on the heuristic, $h(n)$, with the idea that remaining search effort correlates with remaining solution cost. In other words, it assumes that it will be easier to find a path to the goal from nodes with low h. When strictly attempting to minimize search time, Thayer and Ruml (2009) show that greedy best-first search on a different heuristic, d, can be more effective. Instead of estimating cost-to-go, the d heuristic, called a *distance estimate*, estimates the number of remaining search nodes on the solution path beneath a node. In practice, distance estimates are as readily available as cost-to-go heuristics, and can provide much better performance for greedy best-first search in domains where less cost-to-go is not correlated with less search-to-go. We call this algorithm *Speedy* search, in analogy to greedy search.

While greedy best-first search can find solutions very quickly, there is no bound on the cost of its solutions. Bounded-suboptimal search algorithms remedy this problem. We saw some bounded-suboptimal searches in Section 2.5; we review them here. Weighted A* (Pohl, 1970) is perhaps the most common of these techniques; it proceeds just like A*, but

it orders the open list on $f'(n) = g(n) + w \cdot h(n)$, with $w \geq 1$. The weighting parameter, w, puts more emphasis on the heuristic estimate than the cost of arriving at a node, thus it is greedier and it often finds suboptimal solutions much faster than A* finds optimal ones. In addition, the weight provides a bound on the suboptimality of its solutions; the solutions are no more than w times the cost of the optimal solution (Pohl, 1970). Unlike greedy best-first search, weighted A* lets the user select a weight, allowing it to provide either cheaper solutions or faster solutions depending on their needs.

We refer the reader to Thayer (2012) for a more in-depth study of suboptimal and bounded-suboptimal search algorithms, including many that use d heuristics, which we saw in Chapter 4.

5.2.2 Utility Functions

In the previous chapters we described techniques that find optimal solutions. The previous subsection also described bounded-suboptimal search algorithms and greedy best-first search. Often, none of these are really desired: optimal solutions require an impractical amount of resources, one rarely requires solutions strictly within a given bound of optimal, and unboundedly suboptimal solutions are too costly. In this chapter, we propose optimizing a simple utility function given as a linear combination of search time and solution cost:

$$U(s, t) = -(w_f \cdot g^*(s) + w_t \cdot t) \tag{5.1}$$

where s is a solution, $g^*(s)$ is the cost of the solution (the cost of the empty solution, $g^*(\{\})$, is a user-specified value that defines the utility achieved in the case that the search gives up without returning a solution), t is the time at which the solution is returned, w_f and w_t are user-specified weights used to express a preference for trading-off search time and solution cost: the number of time units that the user is willing to spend to achieve an improvement of one cost unit is w_f/w_t. This quantity is usually easily elicited from users if it is not already explicit in the application domain.

While it's not trivial to explicitly optimize a utility function that includes time, a linear

utility function, such as this, has two benefits. First, it lets us express some very useful utility functions. For example, one can optimize goal achievement time by weighting search time equally with solution makespan. Second, the passage of time decays all utility values at the same rate. This simplification allows us to ignore all time that has passed before the current decision point. We can then express utility values in terms of the utility of each outcome starting at the current moment in time. Without this benefit, the mere passage of time would change the relative ordering between the utilities of different outcomes; we would need to re-compute all utility values at every point in time in order to select the best outcome.

5.3 Previous Work

Next we describe previous techniques for trading-off solver time for solution cost.

5.3.1 Contract Search

Dionne et al. (2011) consider the problem of *contract search*, where a goal must be returned by a hard deadline. Unlike real-time search (Korf, 1990), contract search requires the algorithm to return a complete path, not just the next action. Like optimizing a utility function, contract search must not only be aware of the cost of solutions but also of the amount of time required to find them. While the conventional approaches to contract search is to use anytime algorithms, Dionne et al. (2011) present Deadline-Aware Search (DAS) which considers search time directly.

The basic idea behind DAS is to consider only states that lead to solutions deemed reachable within the deadline. Two different estimates are used to determine this set of nodes: an estimate of the maximum-length solution path that the search can explore before the deadline arrives, called d_{max}, and an estimate of the distance to the solution beneath each search node on the open list, in other words d. States for which $d \leq d_{max}$, are deemed reachable, all other states are "pruned." The search expands non-pruned nodes in best-first order on f, updating its d_{max} and d estimates on-line. If the updates cause all remaining

nodes to be pruned while there is remaining time before the deadline, DAS uses a recovery mechanism to repopulate the open list from the set of pruned nodes and continues searching until the deadline is reached.

As mentioned previously, d estimates are as readily available as cost-to-go heuristics for most domains. This leaves the question of how to estimate d_{max}. Dionne et al. (2011) show that simply using the remaining number of possible expansions, computed via the expansion rate and remaining time, is not appropriate due to a phenomenon that they call *search vacillation*. When a best-first search expands nodes, it typically does not expand straight down a single solution path, instead it considers multiple solution paths at the same time, expanding some nodes from each. When it does this, it is said to be *vacillating* between many different paths, and it may not return to work on a particular path until it has performed many expansions along others. To account for vacillation, Dionne et al. introduce a metric called *expansion delay* that estimates the number of expansions performed by a search between two successive nodes along a single path. They define $d_{max} = \frac{t_{rem} \cdot t_{exp}}{delay}$, where t_{rem} is the time remaining before the deadline, t_{exp} is the average expansion rate, and *delay* is the average expansion delay. They compute the average expansion delay by averaging the difference in the algorithm's expansion count between when each node is expanded and when it was generated.

Dionne et al. (2011) showed experimentally that DAS performs favorably to anytime-based approaches and alternative contract search algorithms, indicating that an approach that directly considers search time may also be beneficial for utility function optimization.

5.3.2 Monitoring Anytime Algorithms

Much previous work in optimizing utility functions of solving time and cost, such as Equation 5.1, has focused on finding stopping policies for *anytime algorithms*. As we saw in Section 2.7, anytime algorithms (Dean & Boddy, 1988) are a general class of algorithms that find not one solution, but a stream of solutions with strictly decreasing cost. They get their name because one can stop an anytime algorithm at any time to get its current best

solution. Anytime algorithms are an attractive candidate for optimizing a utility function; since there is more than just a single solution from which to pick, there is more opportunity to choose a solution with a greater utility than when using an algorithm that just finds a single solution. If we were to know the time at which the algorithm will find each of its solutions and the cost of those solutions, then we could compute their utilities and return the solution that maximizes utility. Unfortunately, it is usually not possible to know what solutions an anytime algorithm will return without running it. Instead, while the algorithm is running, one must continually make the decision: stop now, or keep going?

Deciding when to stop is no easy task, because the utility of a solution depends not only on its cost but also on the time needed to find it. On one hand, stopping early can reduce the amount of time used at the expense of having a more costly solution. On the other hand, if the algorithm continues it may not reduce the solution cost by enough to justify the extra computation time. In this case, the final utility can be worse than it would have been had the algorithm stopped earlier. With a little extra information, however, it can be possible to create a reasonable policy.

Hansen and Zilberstein (2001) present a dynamic programming-based technique for building an optimal stopping policy given the *profile* of an anytime algorithm. They define the profile as a probability distribution over the cost of the solution returned by the algorithm, conditioned on its current solution cost and the additional time it is given to improve its solution: $P(q_j|q_i, \Delta t)$, where q_j and q_i are two possible solution costs and Δt is the additional time. The profile allows for reasoning about how the solution cost may decrease if the algorithm is given more time to improve it.

Hansen and Zilberstein's (2001) technique monitors the progress of the anytime algorithm by evaluating the stopping policy at discrete time intervals. If the algorithm considers stopping every Δt time units, then the utility achievable at time t when the algorithm's current solution costs q_i is:

$$V(q_i, t) = \max_d \begin{cases} U(q_i, t) & \text{if d = stop,} \\ \sum_j P(q_j|q_i, \Delta t)\, V(q_j, t + \Delta t) & \text{if d = continue} \end{cases} \tag{5.2}$$

and the stopping policy is:

$$\pi(q_i, t) = \underset{d}{\operatorname{argmax}} \begin{cases} U(q_i, t) & \text{if d = stop,} \\ \sum_j P(q_j | q_i, \Delta t)\, V(q_j, t + \Delta t) & \text{if d = continue} \end{cases} \tag{5.3}$$

where U is the user-specified utility function and P is the profile of the anytime algorithm. They also show a more sophisticated technique that accounts for the cost of evaluating the policy, however, for the algorithms presented in this chapter, the cost of evaluating the policy consists of a mere array lookup and is essentially free.

Since the profile of an anytime algorithm is usually not known, it must be estimated. It is possible to estimate the profile off-line if one has access to a representative set of training instances. The algorithm is run on each of the training instances and a 3-dimensional histogram is created to represent the conditional probability distribution, $P(q_j | q_i, \Delta t)$, needed to compute the stopping policy (cf. Equation 5.3). Appendix D gives a more detailed description of our implementation of this procedure.

5.3.3 Anytime Heuristic Search

Anytime algorithms are a very general class and there are many anytime algorithms for heuristic search (Likhachev et al., 2003a; Hansen & Zhou, 2007; Richter, Thayer, & Ruml, 2010; van den Berg, Shah, Huang, & Goldberg, 2011; Thayer, Benton, & Helmert, 2012b). In this chapter we use Anytime Repairing A* (ARA*, Likhachev et al., 2003a) since it tended to give the best performance over other approaches according to experiments done by Thayer and Ruml (2010). Recall from Section 2.7 that ARA* executes a series of weighted A* searches, each with a smaller weight than the previous. Since the weight bounds the solution cost, the looser bounds on early iterations tend to find costly solutions quickly. As time passes and the weight decreases, so does solution cost, eventually converging to optimal. ARA* also has special handling for duplicates that are encountered during search that enables it to be more efficient while still guaranteeing a bound on each of its solutions.

Like most anytime heuristic search algorithms, ARA* has parameters. Before running ARA*, the user must select the weight schedule, which is typically comprised of an initial

weight and the amount by which to decrement the weight after each solution is found. The behavior of ARA* varies with different weight schedules. For our experiments, we used an initial weight of 3.0 and a decrement of 0.02. This schedule was used by Likhachev et al. (2003a), and we found that it gave the best performance when compared to several alternative schedules on the domains we considered.

5.4 Off-line Bound Selection

We now turn to the first of the two new methods introduced in this chapter.

In this section, we will present a very simple technique for trading search time for solution cost that is based on bounded-suboptimal search. Recall that bounded-suboptimal search algorithms return solutions that are guaranteed to be within a user-specified factor of the optimal solution cost. In practice, few applications require an actual bound, instead the bound is used as a parameter that can be tweaked to speed-up the search if it is not finding solutions quickly enough. The fact that the bound can trade search time for solution cost makes it a prime candidate for automatic parameter tuning (Rice, 1976). That is exactly what we propose.

As with the anytime methods discussed in the previous section, off-line bound selection requires a representative set of training instances. The instances are used to gather information about how a bounded-suboptimal search trades-off search time for solution cost. The only other requirement is that the user select a set of diverse bounds to try as parameters to the search algorithm. The algorithm is then run on each of the N training instances with each suboptimality bound, creating a list of N pairs for each bound: $sols(b) = \langle (c_1, t_1), ..., (c_N, t_N) \rangle$ where b is the bound passed as a parameter to the algorithm, c_i is the cost of the ith solution and t_i is the time at which the ith solution was found. Given a utility function $U : cost \times time \rightarrow \mathbb{R}$, we can select the bound that gives the greatest expected utility on the training set:

$$bound_U = \underset{b}{\operatorname{argmax}} \left(\frac{1}{|sols(b)|} \cdot \sum_{(c,t) \in sols(b)} U(c,t) \right) \tag{5.4}$$

In our experiments, we select a different weight to use for each utility function from the set 1.1, 1.5, 2, 2.5, 3, 4, 6, and 10. One may be able to reduce the number of weights in the training set by using linear interpolation to estimate the performance of parameters between those used for training. This simple approach can also be extended to select over a portfolio of different algorithms in addition to different bounds. It may be beneficial, for example, to include both A* and Speedy search in the portfolio, as these algorithms will likely be selected if cheap solutions are required or if a solution must be found very quickly. We will see in Section 5.6 that this very simple technique outperforms ARA* using an anytime monitor in our experimental evaluation. In fact, if a representative set of training instances is available then this technique tends to perform better than all other algorithms that we evaluate.

A technique related to that which we present here is the dove-tailing method of Valenzano, Sturtevant, Schaeffer, Buro, and Kishimoto (2010). Their approach is presented as a way of side-stepping the need for parameter tuning by running all parameter settings simultaneously. They found that, with dove-tailing, weighted IDA* (Korf, 1985) was able to return its first solution much more quickly, as the dove-tailing greatly reduced the high variance in solving times for any given weight. They also found that dove-tailing over different operator orderings is effective for IDA*. The main difference between the work by Valenzano et al. and ours is that we have quite different goals. Our concern is not to find the first solution more quickly, but rather to select a setting that we expect will better optimize a user-specified utility function. As such, our approach does not run multiple settings at the same time and instead selects a single parameter to run in a single search. In fact, the approaches could be complementary, as one can imagine using dove-tailing to optimize one parameter of an algorithm for speed and then using our technique to set a different one to select a time/cost trade-off.

5.5 Best-first Utility-guided Search

Anytime search is not cognizant of utility. Monitoring and bound selection require training. In this section, we present BUGSY[1], a utility-aware search algorithm that does not require any off-line training.

5.5.1 Expansion Order

Like A*, BUGSY is a best-first search, however, instead of ordering its open list on f, BUGSY orders its open list on an estimate of the utility of expanding each node. Since utility is dependent on time, the mere passage of time affects the utility values. Recall, however, that we are only concerned with linear utility functions, so all utilities decrease at the same rate. Given this, we ignore all past time and compare the utilities assuming that time begins at the current decision point. While these utility values will not match the utility of the ultimate outcome, they still preserve relative order of the different choices that the agent can make.

To understand BUGSY's ordering function, we will first consider the true utility of each node expansion as computed by an oracle. If we had foreknowledge of a maximum utility outcome, the only purpose of the search algorithm would be to achieve it by expanding the nodes along the path from the initial node. Since our utility function is given as a linear combination of solution cost and search time, the utility value of this outcome can be written in terms of the cost and path length of a (possibly empty) maximum utility outcome, s:

$$U^* = -(w_f \cdot g^*(s) + w_t \cdot d^*(s) \cdot t_{exp}) \tag{5.5}$$

where $g^*(s)$ is the cost of the path s (recall that the cost of the empty path is a user-specified constant), $d^*(s)$ is the number of nodes on s, and t_{exp} is the time required to expand a node.

Given the maximum utility value U^*, the true utility of the outcome resulting from

[1] BUGSY is an acronym for "Best-first Utility-Guided Search—Yes!"

expanding a node is:

$$u^*(n) = \begin{cases} U^* & \text{if } n \text{ leads to a maximum utility outcome} \\ U^* - w_t \cdot t_{exp} & \text{otherwise} \end{cases} \tag{5.6}$$

In other words, the utility we get from expanding a node that leads to a maximum utility outcome is the maximum utility; expanding any other node is simply a waste of time, and has a utility of the maximum utility minus what was lost by performing the unnecessary expansion.

In practice, we do not know the maximum utility, so we must rely on estimates. Bugsy uses two estimates to approximate the maximum utility: first, it estimates the cost of the solution that it will find beneath each node, f (note that f is an estimate, not only because the heuristic is an estimate of the true cost-to-go, but also because the cheapest solution beneath a node may not be the solution of greatest utility), and second it estimates the number of expansions required to find each solution, *exp*. One crude estimate for remaining expansions is d, the distance heuristic that estimates the remaining nodes on the solution path. In reality, Bugsy will experience search vacillation, expanding more nodes than just those along a single solution path. To account for this vacillation, we use the expansion delay technique of Dionne et al. (2011) discussed earlier, and we estimate $exp(n) = delay \cdot d(n)$. That is, we expect each of the remaining $d(n)$ steps to a goal to require *delay* expansions. Bugsy can either choose to expand a node, or it can stop and return the empty solution. So, Bugsy's estimated maximum utility using these values and Equation 5.5 is given by:

$$\hat{U} = \max \left\{ \max_{n \in open} -(w_f \cdot f(n) + w_t \cdot d(n) \cdot delay \cdot t_{exp}), U(\{\}, 0) \right\} \tag{5.7}$$

Once the estimate $\hat{U}$ is found, it would be possible to substitute it into Equation 5.6 to estimate the utility of expanding each node on the open list. Bugsy skips this step, however, because it would only use the estimates to expand one node: the one with the maximum estimated utility. Additionally, instead of computing the maximization in Equation 5.7 from scratch each time it is about to expand a node, Bugsy simply orders its open list on $u(n) = -(w_f \cdot f(n) + w_t \cdot d(n) \cdot delay \cdot t_{exp})$, each iteration popping off the node with the maximum $u(n)$ for expansion.

Heuristic Corrections

Many best-first search algorithms use so called *admissible* heuristic estimates that never overestimate the true cost-to-go. The proof of optimality of A* and the proofs of bounded suboptimality of bounded suboptimal search algorithms rely crucially on the admissibility property of the heuristic. BUGSY does not fixate on optimal solutions and does not guarantee bounded cost. Instead, BUGSY optimizes a utility function for which solution cost is only one of two terms. Since there are no strict cost guarantees, BUGSY is free to drop the admissibility requirement if more informed but inadmissible estimates are available.

Thayer, Dionne, and Ruml (2011) show that inadmissible estimates can provide better performance for bounded suboptimal search. One such technique attempts to correct the heuristic estimates on-line using the average single-step error between the heuristic values of a node and its best child. Thayer et al. show that, while this technique provides good guidance, it is actually less accurate to the true cost-to-go values than the standard admissible heuristics. For BUGSY, this is undesirable, as it does not need good guidance, but proper estimates. Thayer et al. also show that learning the heuristic off-line with linear regression can provide more accurate estimates. Unfortunately, the fact that the heuristic must be trained off-line negates BUGSY's benefit of being an on-line algorithm. It is a matter of empirical evaluation as to whether any of these techniques will provide better performance for BUGSY. In Section 5.6.9, we show that using the standard admissible heuristics often gives the best performance anyway.

5.5.2 Stopping

BUGSY orders its open list by $u(n)$, and stops searching when the maximum estimated utility is less than that of returning the empty solution. While it may be possible to continue searching after the first goal is found, in an anytime fashion, from a utility perspective this is not the correct approach. We prove that here:

Theorem 10 *Assuming the expansion time t_{exp} is constant, h is admissible, and exp never overestimates the expansions to go, at the time that* BUGSY *finds its first solution, s, the*

solutions BUGSY *would find beneath the remaining nodes would result in less utility than immediately returning* s*.*

Proof: Let $\mathcal{T}$ be the current time at which BUGSY found solution s. The utility of returning s is $U(s, \mathcal{T}) = u^*(s) = -(w_f \cdot f^*(s) + w_t \cdot \mathcal{T})$, where $u^*(s)$ is the utility of returning s now, and $f^*(s)$ is the cost of solution s. Note that $u(s) = u^*(s)$ because h is admissible, s is a goal, and therefore $h(s) = 0$, and also *exp* never overestimates the expansions to go and thus $exp(s) = 0$. Also, since s was chosen for expansion $u^*(s) \geq u(n)$ for every node n on the open list.

Let $t(n)$ be the minimum amount of additional time BUGSY requires to find the solution beneath any unexpanded node n. $t(n) \geq t_{exp}$ since BUGSY must at least expand n itself. Since h is admissible, $f(n) \leq f^*(n)$, and since *exp* never overestimates, $exp \cdot t_{exp} \leq t(n)$, thus $u(n) \geq -(w_f \cdot f^*(n) + w_t \cdot (t(n) + \mathcal{T})) = u^*(n)$. So $u^*(s) \geq u(n) \geq u^*(n)$ for all unexpanded nodes n. □

This justifies BUGSY's strategy of returning the first goal node that it selects for expansion.

5.5.3 Resorting

Instead of requiring off-line training as in the previous approaches, BUGSY uses on-line estimates to order nodes on its open list. First, while many analyses regard t_{exp} as as a constant, it can in practice depend on log-time heaps, cache behavior, multiprogramming overhead, etc., so BUGSY estimates t_{exp} as a global average computed during search. Second, BUGSY's expansion delay estimate is calculated as the global average of the difference in expansion count from when each node was generated to when it was expanded; this too must be done on-line. Unfortunately, the on-line estimates may change at each node expansion, and naïvely using the latest estimates to compute the u value for newly generated nodes can lead to poor performance. This is due to the comparisons used to order the open list; instead of fair comparisons based on the estimated utility of each node, the recent and very fresh estimates of new nodes will be compared with the old and possibly more stale

estimates of nodes that have been open for a long time.

To alleviate this problem, our implementation of BUGSY uses two sets of estimates: one stable set used to order the open list, and one ever-changing set maintaining the most recent estimates. At certain points throughout the search, BUGSY copies the most up-to-date estimates into the stable set, recomputes the utility values of all open nodes, and resorts the open list. Our open list is implemented as a binary heap so it can re-establishing the heap property in linear time. Unfortunately, it would still be very expensive to do this at every node expansion, so, instead, we reorder the open list exponentially less frequently as the search progresses—only when the number of expansions is a power of two. We prove that this *logarithmic scheme* only adds a constant amount of overhead per-expansion when amortized over the entire search.

Theorem 11 *In a search space with a finite maximum branching factor, the overhead of reordering the open list on power of two expansions is constant for each expansion when amortized over the search.*

Proof: Let b be the maximum branching factor. The maximum number of nodes that can be on the open list after n expansions is $N(n) = bn - n = n(b-1)$. The total cost of all resorting after n expansions is no more than:

$$\begin{aligned}
\sum_{i=0}^{\lfloor \lg n \rfloor} \mathcal{O}(2^{N(i)}) &= c \sum_{i=0}^{\lfloor \lg n \rfloor} 2^{i(b-1)}, \text{for some } c > 0, \text{ by definition of } \mathcal{O}(2^{N(i)}) \\
&= c2^{b-1} \sum_{i=0}^{\lfloor \lg n \rfloor} 2^i \\
&= c2^{b-1}(2^{\lfloor \lg n \rfloor + 1} - 1), \text{by the identity } \sum_{0 \le i < j} 2^i = 2^{j+1} - 1 \\
&\le c2^{b-1}(2 \cdot 2^{\lg n} - 1) \\
&= c2^{b-1}(2n - 1) \\
&= \mathcal{O}(n)
\end{aligned}$$

□

So, the overhead per-expansion is constant when amortized over all expansions. It is a matter of empirical evaluation to determine if this constant overhead is detrimental.

```
BUGSY(initial, u(·))
 1. open ← {initial}, closed ← {}
 2. do
 3.    n ← remove node from open with highest u(n) value
 4.    if n is a goal, return it
 5.    add n to closed
 6.    for each of n's children c,
 7.       if c is not a goal and u(c) < 0 or an old version of c is in open or closed
 8.          skip c
 9.       else add c to open
10.    if the expansion count is a power of two
11.       re-compute u(n) for all nodes on the open list using the most recent estimates
12.       re-heapify the open list
13. loop to step 3
```

Figure 5-1: Pseudo-code for BUGSY.

5.5.4 Implementation

Figure 5-1 shows high-level pseudo-code for BUGSY. For clarity, the code elides the details of computing $u(n)$ values. The algorithm proceeds like A*, selecting the open node with the highest $u(n)$ for expansion (line 3). If this node is a goal then it is returned as the solution (line 4), otherwise the node is put on the closed list (line 5) and its children are generated. Each new child is put onto the open list (line 9) except duplicate nodes and nodes for which expansion is estimated to have a negative utility; these are discarded (lines 7–8). At each power-of-two expansion, the utility of each open node is re-computed using the latest set of estimates for t_{exp} and expansion delay, and the open list is re-heapified (lines 10–12).

5.6 Experimental Evaluation

All the techniques discussed above involve approximations and estimations that may or may not work well in practice. In this section we present results of an experimental comparison of the techniques to better understand their performance.

5.6.1 Questions Answered

In the following sections, we answer four questions experimentally. First, we would like to ensure that our monitored ARA* algorithm is performing at its best by comparing the profile learned off-line with an oracle. As we will see, the off-line profile, while only an estimate of the true profile of the algorithm, is quite well-informed.

In Section 5.5.3, we proved that resorting only adds a constant overhead per-expansion when amortize over the entire search, however, it is a matter of empirical evaluation to determine whether or not the benefits outweigh this overhead. Our experiments show that resorting with a logarithmic schedule greatly outperforms BUGSY without resorting.

In Section 5.5.1 we pointed out that BUGSY does not require admissible heuristic estimates, and in fact it may perform better with inadmissible, but more accurate heuristics. Next we show how BUGSY performs with admissible heuristics, and with two different types of corrected heuristics. Overall, we conclude that the best configuration is BUGSY with the standard admissible heuristics.

Finally, we compare A*, Speedy search, monitored ARA*, weighted A* with a learned weight, and BUGSY. We find that the simplest approach of learning a good weight for weighted A* gives the best performance. We also find that BUGSY, which doesn't use any off-line training, performs about as well as monitored ARA*, which does use off-line training. Therefore, if training instances are available, we recommend the simple weighted A* approach where the weight is selected based on performance on the training set. If no training instances are available BUGSY is the algorithm of choice.

5.6.2 Domains

In order to verify that our results hold for a variety of different problems, we performed our experiments on four different domains. Some of these domains were also presented in previous chapters, but we review them here.

5.6.3 15-puzzle

The 15-puzzle is one of the most popular benchmark domains for heuristic search algorithms. It consists of a 4-by-4 frame into which 15 tiles have been placed. One slot of the board does not contain a tile, it is called the *blank*. Tiles that are above, below, left of or right of the blank may be slid into the blank slot. The objective of the 15-puzzle is to slide tiles around in order to transform an initially scrambled puzzle into the goal state with the blank in the upper-left corner and the tiles ordered 1–15 going from left to right, top to bottom. This domain is interesting because plans are hard to find, the branching factor is small and varies little from its mean of about 2.13 (Korf et al., 2001), there are few duplicates, and the heuristic is reasonably informed.

In our experiments we use the popular 100 15-puzzle instances from Korf (1985). In plots that include A*, however, we only used the 94 instances solvable by A* in 6GB of memory. The average optimal solution length for these instances was 52.4. For our training set, we generated 1,000 instances using a 1 million step random walk back from the goal position. We used the Manhattan distance heuristic, which sums the vertical and horizontal distance that each tile must move to arrive at its goal position. Our implementation follows the heavily optimized solver presented by Burns, Hatem, Leighton, and Ruml (2012b).

5.6.4 Pancake Puzzle

The pancake puzzle (Dweighter, 1975; Gates & Papadimitriou, 1979) is another permutation puzzle. It consists of a stack of differently sized pancakes numbered 1–N. The pancakes must be presented at a fancy breakfast, so a chef needs to sort the originally unordered stack of pancakes by continually sticking a spatula into the stack and reversing the order of the pancakes above. Said another way, the pancake problem involves sorting a sequence of numbers by using only prefix reversal operations. This simple problem is interesting because it creates a search graph with a large branching factor (the number of pancakes less 1). For our experiments, we used 25 randomly generated 50-pancake puzzle instances, and our training set consisted of 1,000 randomly generated instances. We used the powerful

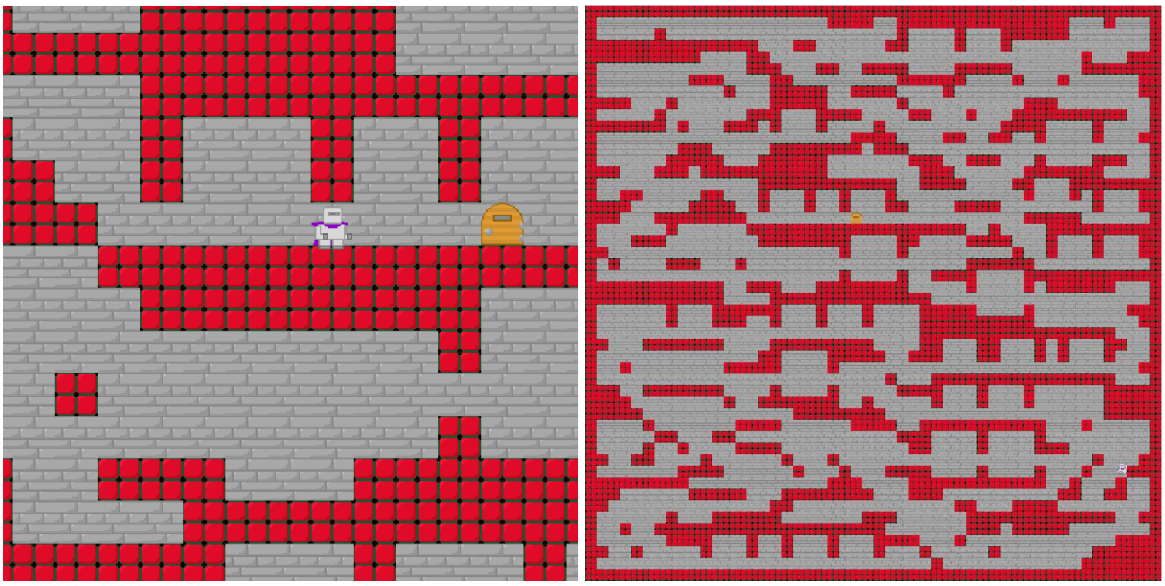


Figure 5-2: A screenshot of the platform path-finding domain (left), and a zoomed-out image of a single instance (right). The knight must find a path from its starting location, through a maze, to the door (on the right-side in the left image, and just above the center in the right image).

GAP heuristic of Helmert (2010).

5.6.5 Platform Path-finding

The platform domain is a path-finding domain of our own creation with dynamics based on a 2-dimensional platform-style video game called `mid`. The left image of Figure 5-2 shows a screenshot from `mid`[2]. The goal is for the knight to traverse a maze from its initial location, jumping from platform to platform, until it reaches the door. `Mid` is an open source game available from `http://code.google.com/p/mid-game`. For our experiments the game physics of the game were ported from C to C++ and were embedded in our C++ search codebase. We generated 1,000 training instances and 100 test instances using the level generator from `mid`. An example instance is shown on the right panel in Figure 5-2. The domain is unit-cost and has a large state space with a well-informed heuristic.

[2]The tile graphics were drawn by Steve McCoy.

The available actions are different combinations of controller keys that may be pressed during a single iteration of the game's main loop: left, right, and jump. Left and right move to the knight in the respective directions (holding both at the same time is never considered by the search domain, as the movements would cancel each other out, leaving the knight in place), and the jump button makes the knight jump, if applicable. The knight can jump to different heights by holding the jump button across multiple actions in a row up to a maximum of 8. The actions are unit cost, so the cost of an entire solution is the number of game loop iterations, called frames, required to execute the path. Each frame corresponds to 50ms of game play.

Each state in the state space contains the x, y position of the knight using double-precision floating point values, the velocity in the y direction (x velocity is not stored as it's determined solely by the left and right actions), the number of remaining actions for which pressing the jump button will add additional height to a jump, and a boolean stating whether or not the knight is currently falling. The knight moves at a speed of 3.25 units per frame in the horizontal direction, it jumps at a speed of 7 units per frame, and to simulate gravity while falling, 0.5 units per frame are added to the knight's downward velocity up to a maximum of 12 units per frame.

Level Generator

The instances used in our experiments were created using the level generator from `mid`, a special maze generator that builds 2-dimensional platform mazes on a grid of blocks. In our experiments we used 50×50 grids. Each block is either open or occluded, and to ensure solvability given the constraints imposed by limited jump height, the generator builds the maze by stitching together pieces from a hand-created portfolio. Each piece consists of a number of blocks that are either free or occluded, and a start and end location for which traversability is ensured within the piece. A piece can be added to the grid at any location for which it fits. A piece fits if it does not occlude a block that belongs to a previously placed piece. The maze is built using a depth-first procedure: a piece is selected at random and

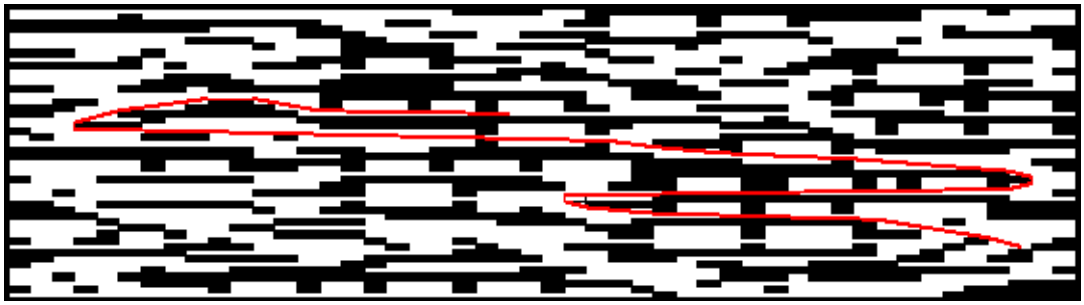


Figure 5-3: The visibility navigation instance for the platform domain's heuristic. The visibility path between the initial state and the goal state is drawn in red.

if it fits in the grid with its start location lined up with the end location of its predecessor then it is placed and the procedure recurs. The number of successors of each node is chosen uniformly from the range 3–9 inclusive, and the procedure backtracks when there are no pieces that fit on the previous block. Once the maze is constructed, blocks that do not belong to any piece are marked as occluded. The right image in Figure 5-2 shows a sample of a level generated by this procedure. The source code for the level generator is available in the `mid` source repository mentioned above.

Heuristic

We developed a heuristic for the platform domain that is based on visibility navigation (Nilsson, 1969). Each maze is pre-processed to convert its grid representation into a set of polygons representing each connected component of occluded cells in the level. The space is then scaled to account for the movement speed of the knight. The knight can fall faster than it can move in the horizontal direction, so the polygons end up squished vertically and stretched horizontally. The visibility navigation problem is then solved in reverse from the four corners of the goal cell to the center of every non-occluded cell of the maze. To maintain admissibility, the cost of each edge in the visibility problem is not the length of the visibility line, but instead is the maximum of the length of the line divided by $\sqrt{2}$ and the X and Y displacements between the end points of the line. This accounts for the fact

that the knight can be moving both horizontally and vertically at the same time, and that moving a distance of $\sqrt{2}$ in the scaled space still takes only a single frame.

During search, the heuristic value of a state is computed in one of two different ways. If the straight-line path from the center of the knight to the goal is not occluded then the maximum of the X and Y distances to the goal scaled down by travel speed is used as the heuristic estimate. Otherwise, the heuristic is the cost of the path in the visibility graph from the center of the cell that constraints the knight's center point minus the maximum of the X and Y distance (in number of frames) of the knight's center point to the center of its cell. Figure 5-3 shows the same map from the right image of Figure 5-2, scaled, broken into polygon components, and with the visibility path between the initial state and the goal state.

5.6.6 Grid Path-finding

Our final domain was grid path-finding. This is a very popular domain in both video games and robotics, as such it has garnered much attention in the heuristic search community. In our experiments, we used 5,000x5,000 grids with both four-way and eight-way connectivity and uniform obstacle distributions. For four-way connected grids, each cell was blocked with a probability of 0.35, and for eight-way connected grids cells were blocked with a probability 0.45. We also consider two different cost models, the standard *unit cost* model in which horizontal and vertical moves cost 1 and diagonal moves cost $\sqrt{2}$. The other is called the *life cost* model, where each move has a cost equal to the row number from which the move took place, causing cells toward the top of the grid to be preferred. With the life cost model, short direct solutions can be found quickly, however they will be relatively expensive, while a least-cost solution involves many annoying economizing steps (Ruml & Do, 2007). This model can be viewed as an instantiation of the popular belief that time is money. For each combination of movement model and cost model, we generated 25 test instances and 1,000 training instances. Finally, we used the Manhattan distance heuristic for four-connected grids and the octile distance heuristic for eight-connected grids. For the

life cost model our heuristics also took into account the fact that moving toward the top of the grid then back down may be cheaper than a direct path.

5.6.7 Anytime Profile Accuracy

We want to ensure that our implementation works well and training instance sets are representative enough that monitored ARA* can perform at its best. In this subsection we evaluate the accuracy of the stopping policies created using the estimated anytime profiles by comparing them to an oracle. Since the stopping policy is only guaranteed to be optimal for the true algorithm profile, it is a matter of empirical study to determine whether or not the estimated profile will lead to a good policy.

To estimate the profile used by the monitored version of ARA*, we ran ARA* with a 6GB memory limit or until convergence on 1,000 separate test instances for each domain. Next, we created a histogram by discretizing the costs and times of each of the solutions into 10,000 bins (100×100). We experimented with different utility functions by varying the ratio w_f/w_t in Equation 5.1. Small values of w_f/w_t give a preference to finding solutions more quickly, whereas large values prefer finding cheaper solutions. In the case of the platform game, for example, this can be viewed as a way to change the speed at which the agent moves: a slow agent might benefit from more planning in order to find a shorter path, but a fast agent can execute a path quickly, and may prefer to find any feasible solution as fast as possible.

Figures 5-4 and 5-5 show the results of this experiment. The box plots represent the distribution of utility values found by ARA* using the estimated stopping policy, given as the factor of the oracle's utility. Since the utility values are negative, so larger factors represent smaller (more negative) utilities and thus a worse outcome. The boxes surround the center of the data, the whiskers extend to the extremes, and circles show values that are 1.5× the inter-quartile range outside of the box. The center line of each box shows the median, and the gray rectangles show the 95% confidence interval on the means. Each box represents a different w_f/w_t as shown on the x axis. There is a reference line drawn across

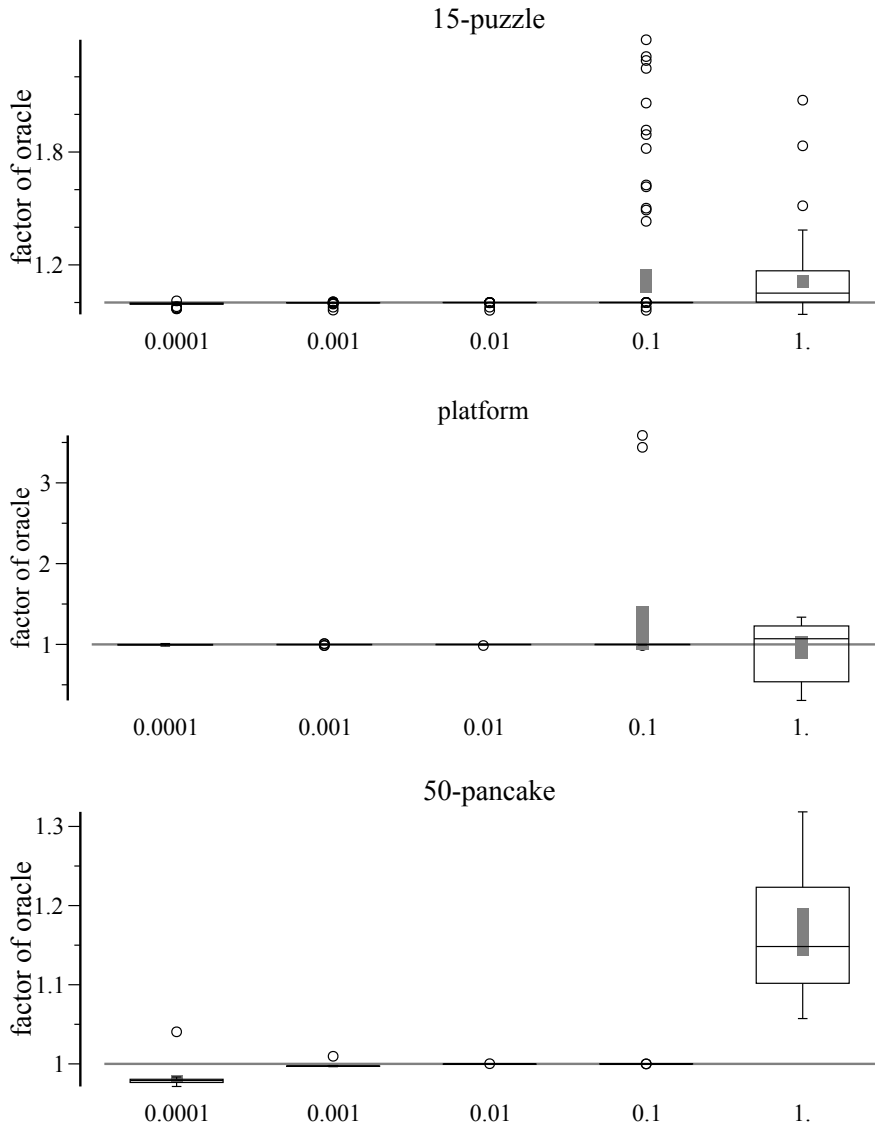


Figure 5-4: Comparison of the optimal stopping policy and the learned stopping policy.

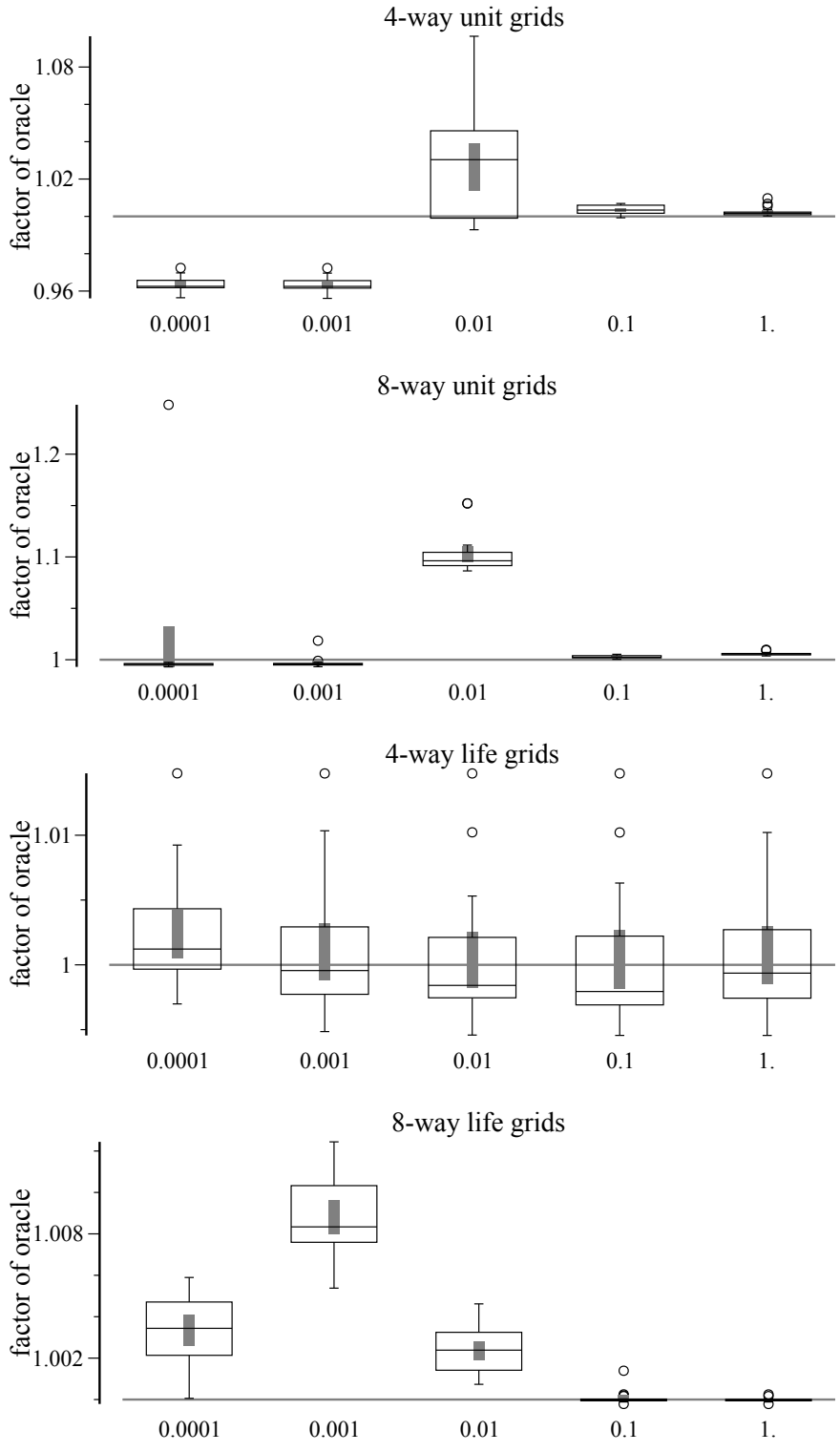


Figure 5-5: Comparison of the optimal stopping policy (continued).

at $y = 1$ (the point where the oracle and estimated policy performed equally as well), and in many cases the boxes are so narrow that they are indistinguishable from this line.

Some points in these figures lie very slightly below the $y = 1$ line, indicating instances where the oracle performed worse than the estimated policy. This is possible due to the variance in solving times. In our experiment, the ARA* runs used to compute the oracle's utilities occasionally found solutions more slowly than the ARA* runs using the estimated stopping policy. In other words, it is caused by the non-determinism inherent in a utility function that depends on solving time. As is obvious in the figure, these instances are quite rare and usually happened for small values of w_f/w_t , where miniscule time differences may matter a lot.

From these results, we conclude that our monitored ARA* implementation performs quite well, as the stopping policy often stopped on the best solution available from those emitted by the underlying anytime algorithm.

5.6.8 To Resort or Not to Resort?

In Section 5.5.3 we proved that resorting BUGSY's open list on power of two expansions only added a constant overhead per-expansion when amortized over the search. It is a matter of empirical evaluation to determine whether or not this overhead is worth the effort.

Figure 5-6 shows the utility achieved by BUGSY both with and without resorting. The x axes show the w_f/w_t ratio determining the preference for solution cost and search time on a $\log_{10}$ scale. As with the previous plots, smaller values indicate a preference for faster search times and larger values indicate a preference for cheaper solutions. The y axes show the factor of the utility achieved by the best technique on each instance, again on a $\log_{10}$ scale. A y value of $\log_{10}1 = 0$ indicates the best utility achieved by any technique on a given instance; values greater than zero indicate less utility. Points show the mean value over all test instances with error bars giving the 95% confidence intervals. From these plots, we can see that resorting the open list led to significant improvements in all domains. On the pancake puzzle, BUGSY without resorting was unable to solve any of the instances within

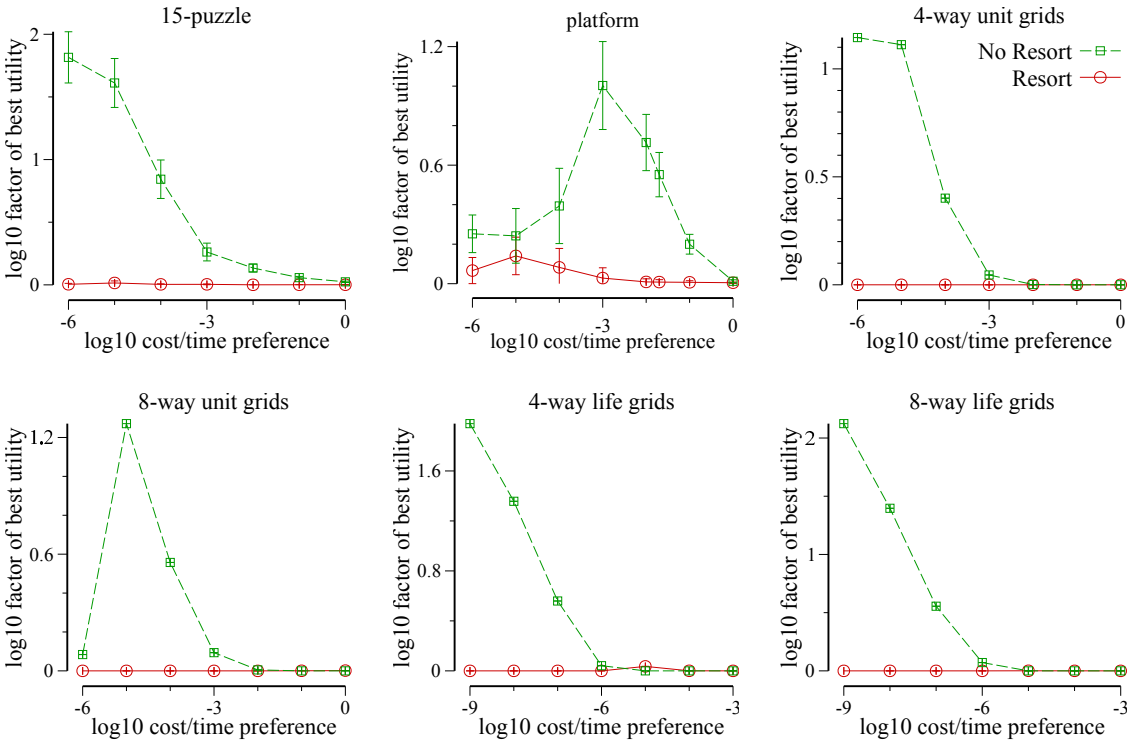


Figure 5-6: BUGSY: Resorting the open list (circles) vs not (boxes).

a 6GB memory limit. In our remaining experiments, we always enable resorting on an exponential schedule.

5.6.9 Heuristic Corrections

In Section 5.5.1, we mentioned that BUGSY does not require admissible heuristic estimates, as it provides no guarantees on solution cost. In this section we compare BUGSY using the standard admissible heuristics to BUGSY using both on-line and off-line corrected heuristics. Our on-line heuristic correction used a global average of the single-step heuristic error between each node and its best offspring, and our off-line heuristic was a linear combination of h, g, *depth*, and d, for each node. The coefficients for each term in the off-line heuristic were learned by solving a set of training problems and using linear least squares regression.

The comparison is shown in Figure 5-7. The plots are in the same style as Figure 5-6. Typically the on-line correction technique performed worst—some times significantly

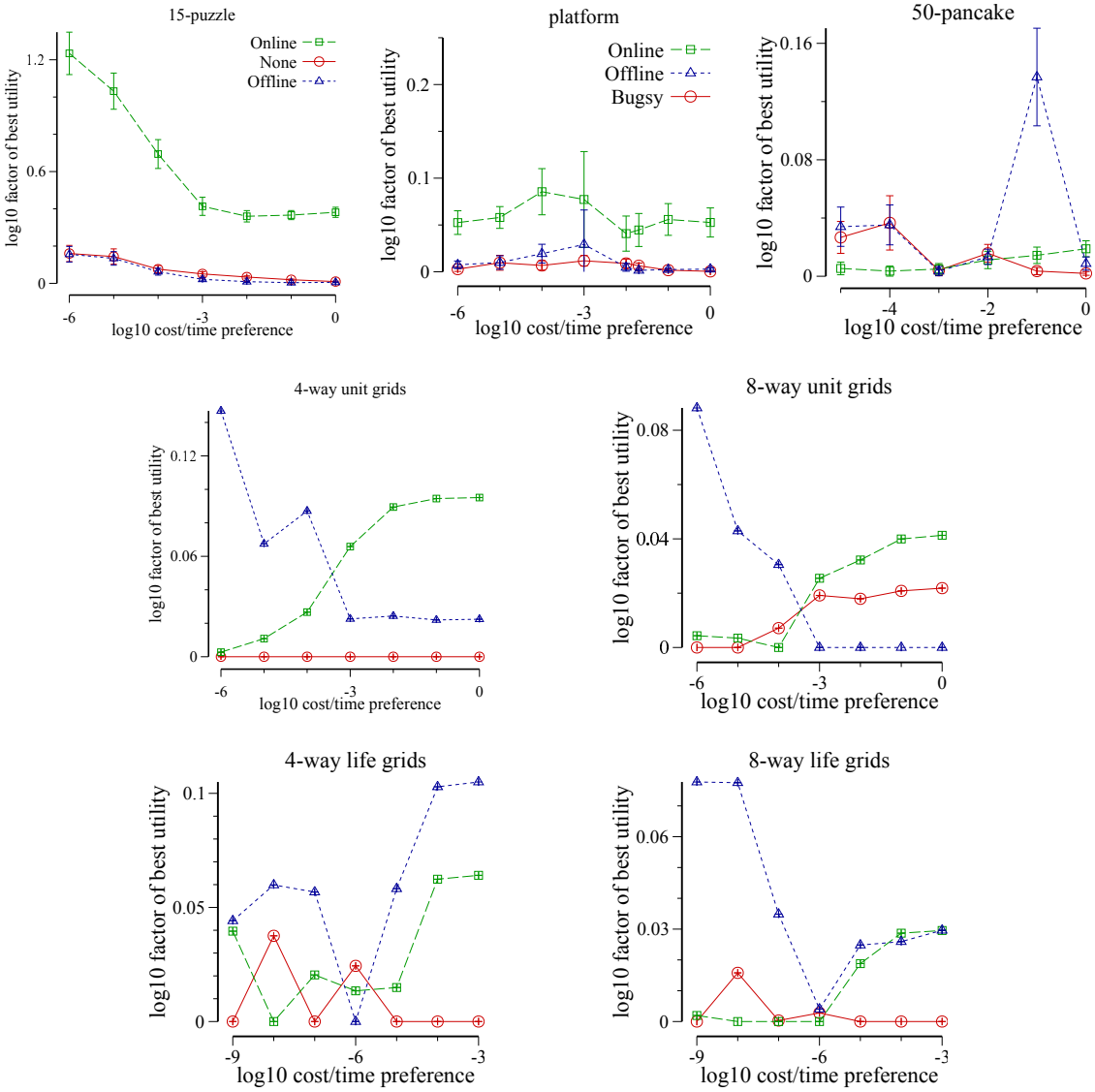


Figure 5-7: BUGSY: Heuristic corrections.

worse—than the other two. We attribute this to its poor accuracy as observed by Thayer et al. (2011). On some problems, such as the 15-puzzle and the 8-way unit-cost grid path-finding, the off-line correction technique performed best, but in general the simple admissible heuristics were the best or were competitive with the best. For the remainder of our experiments, we chose to use the simplest variant without any corrections as it did not require any off-line training (which is one of BUGSY's main benefits), and it was never the worst and was often the best or near the best.

5.6.10 Comparing Techniques

Figures 5-8 and 5-9 show a comparison of the three different techniques for utility-aware search. These plots are larger than the previous plots to improve clarity, because they have more lines. The plots include A*, Speedy search, BUGSY, ARA* with monitoring (ARA*), and weighted A* with the weight chosen automatically for each different utility function from the set 1.1, 1.5, 2, 2.5, 3, 4, 6, and 10 (wA*). As we would expect, when the preference was for cheaper solutions (on the right end of the x axes) A* performed very well and Speedy search performed poorly. As the preference shifted toward desiring solutions more quickly, however, A* begun to perform very poorly as it doggedly stuck to finding the optimal solution, whereas Speedy improved. The utility-cognizant techniques performed much more robustly than A* and Speedy search which do not take the user's preference for search time and solution cost into account at all.

Surprisingly, the simplest of the utility-cognizant methods, weighted A* with an automatically chosen weight, was the technique that achieved the most utility. On all domains, this method nearly dominated all other utility-cognizant approaches, and it was only beat by A* and Speedy at the very extreme ends of the x axes. It should be noted that, if both A* and Speedy search were added to the portfolio in addition to weighted A* with various weights, then this algorithm would have given the best performance even at the extremes.

Of the utility-cognizant techniques, both BUGSY and weighted A* with an automatically selected weight performed the best. BUGSY was better on both the 15-puzzle and the

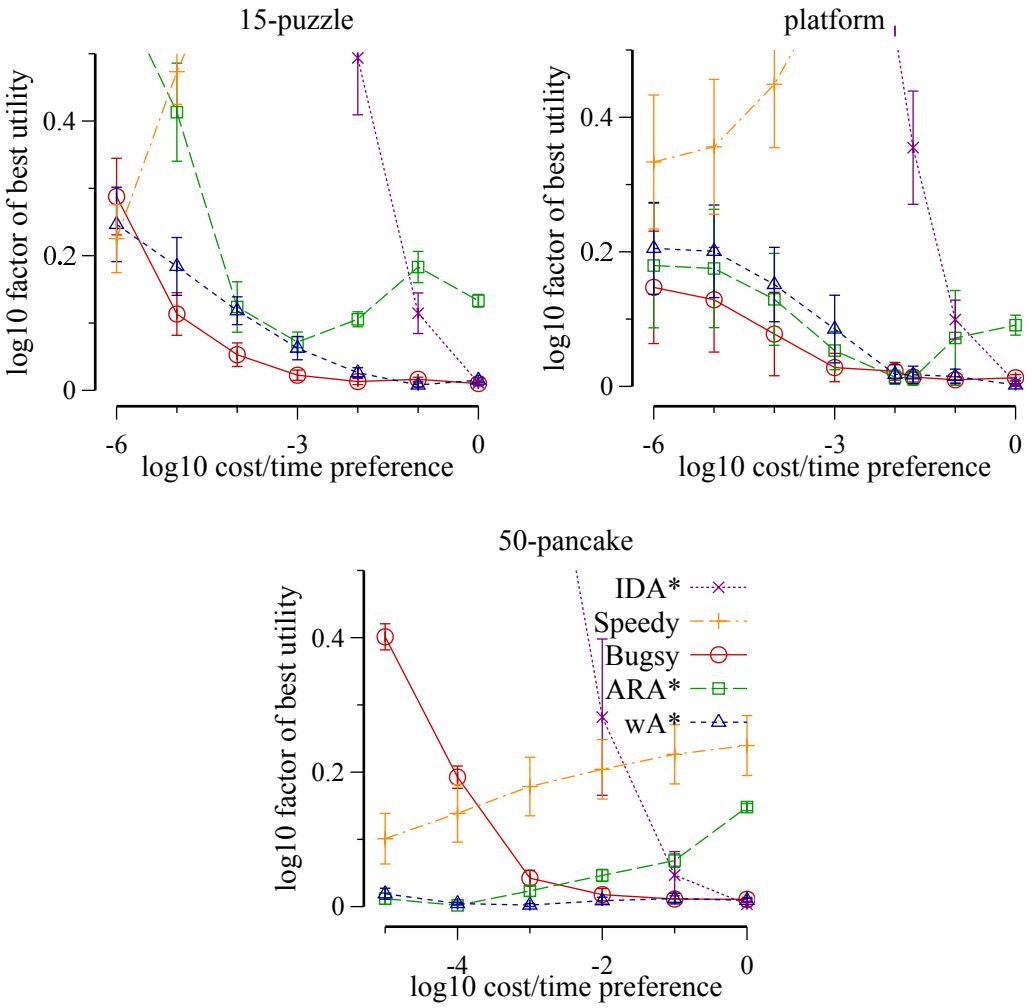


Figure 5-8: Comparison of techniques.

platform domain. On the grid problems, BUGSY and weighted A* had roughly the same performance on the right side of the x axes. On the left side, BUGSY tended to get worse relative to the other utility-cognizant techniques, and ARA* with an anytime monitor was often the best performer.

The utility-cognizant techniques often performed as well as A* when low-cost solutions were preferred. When fast solutions were preferred, these techniques sometimes outperformed Speedy search. This likely indicates that solution cost still played a roll in the final utility on the left-most points in some of the plots. ARA* tended to achieve greater utility than BUGSY when solutions were needed quickly, however, when cheaper solutions were pre-

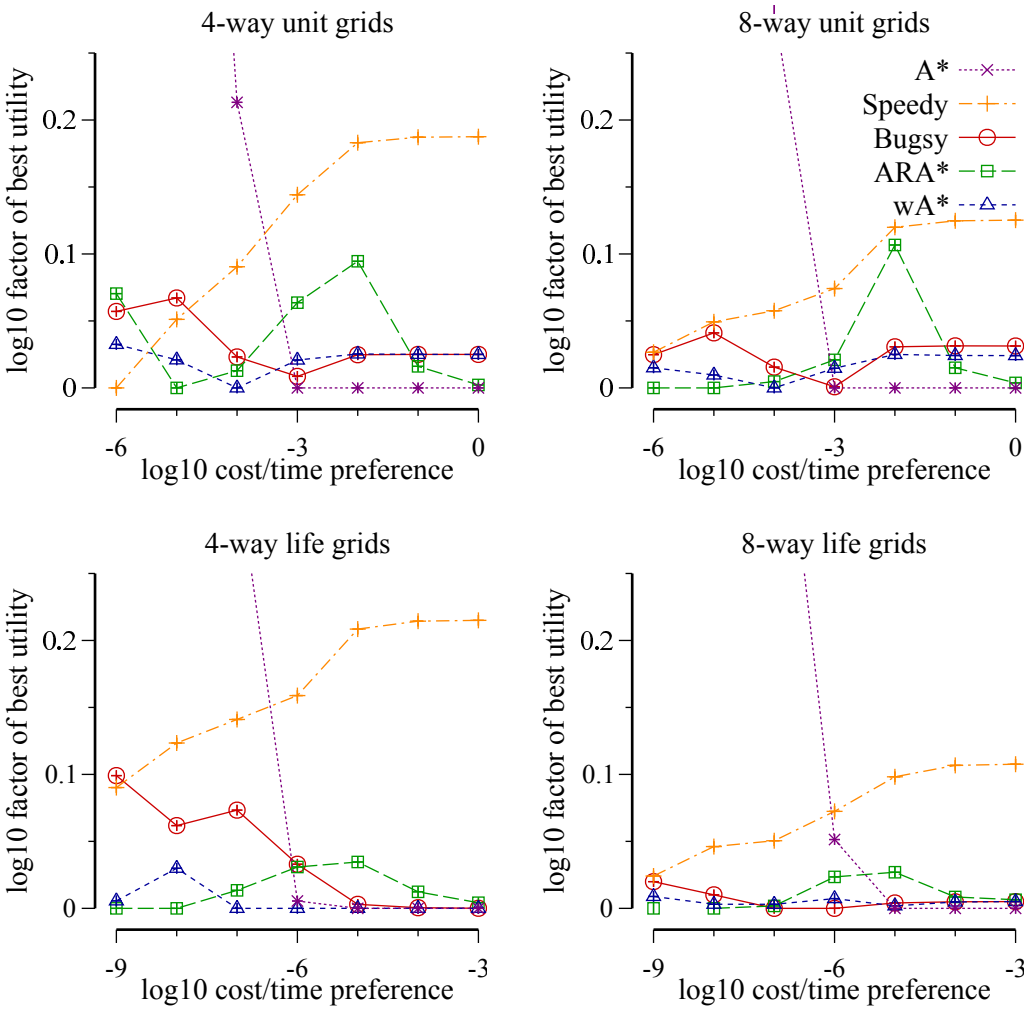


Figure 5-9: Comparison of techniques (continued).

ferred, BUGSY tended to be better than ARA*. On most domains, ARA* seemed to have a spike of low utility for ratios between 0.001, and 1, with the peak appearing between 10^{-6} and 10^{-3} for life-cost grids. This peak approximately coincides with the utility functions for which the estimated profile performed worse than the oracle as shown in Figure 5-4, possibly indicating that more than 1,000 training instances were required for these utility functions.

Overall, the utility-cognizant techniques were able to achieve much greater utility than the utility-oblivious A* and Speedy search algorithms. The results also suggest that our parameter tuning technique can give the best performance if a representative set of training

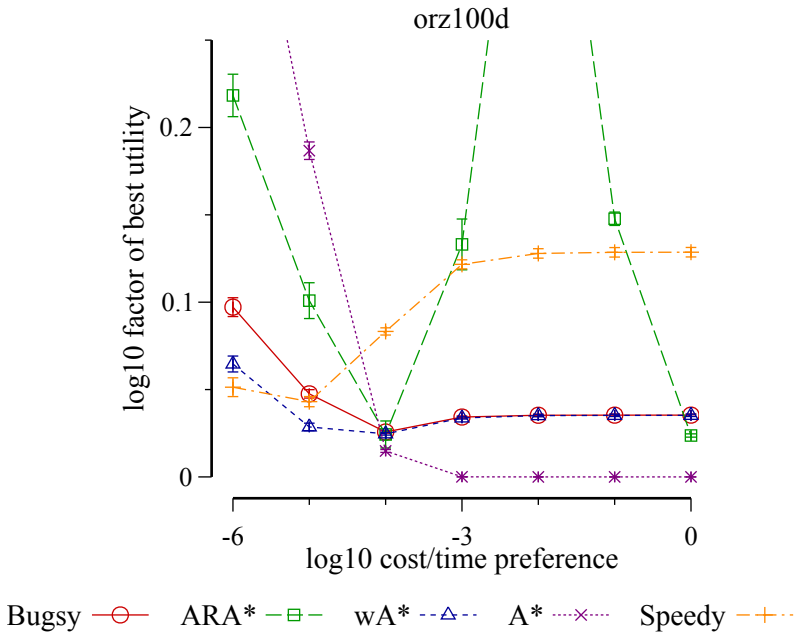


Figure 5-10: Grid path-finding on a video game map.

instances is available. If not, then BUGSY is the algorithm of choice as it performs well and does not require any off-line training.

5.6.11 Limitations

In the previous set of experiments, we saw that the utility-cognizant algorithms outperformed both Speedy search and A* for a wide range of utility functions. In this section, we look at one domain for which this tends not to be the case: video game grid maps.

Video games are one of the main motivations for research in grid path-finding problems. Sturtevant (2012) observed that grid maps created by game designers often exhibit very different properties from maps generated algorithmically. Figure 5-10 shows a comparison of BUGSY, monitored ARA*, weighted A* with an automatically selected weight, Speedy, and A* on the Dragon Age Origins map orz100d from Sturtevant's (2012) benchmark set. This map has a fairly wide-open area at the top, with a more closed-off bottom half containing rooms and hallways. The format of the plot in this figure is the same as those in the

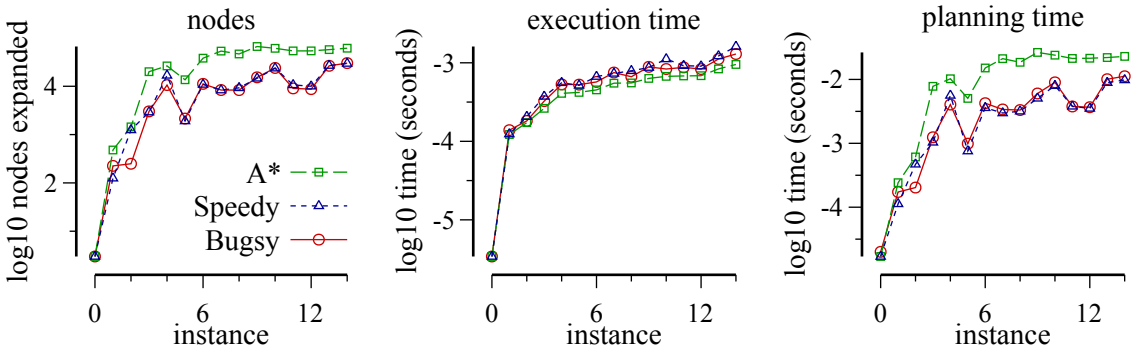


Figure 5-11: Nodes expanded, planning time, and execution time.

previous subsection. As we can see, A* gave the best performance for a large range of utility functions, and BUGSY actually never outperformed Speedy or A* in the entire experiment (neither did ARA*, and wA* only gave the best performance at a single data point). We hypothesized that Bugsy's poor performance was because these problems are very easy to solve, and BUGSY's extra computation overhead, while very small, was more prominent.

To explore this hypothesis, we plotted the performance of BUGSY, A*, and Speedy using a single utility function given by $w_f = 10^{-6}, w_t = 1$. This is the left-most utility function in Figure 5-10, a function for which here Speedy search performed the best, and BUGSY performed poorly. Figure 5-11 shows the number of nodes expanded, the time spent executing, and the time spent planning for a random sample of 15 instances from Sturtevant's (2012) scenario set for the orz100d map. The x axes shows the rank of the instances in this sample ordered by their optimal solution length. The y axes show nodes (for the left-most plot) and time in seconds (for the center and right-most plots) on a $\log_{10}$ scale. As we can see in Figure 5-11 BUGSY both expanded the same number of nodes, and had the same execution times as Speedy. The only difference in performance between these two algorithms is shown in the right-most plot, were we can see that BUGSY required more planning time. Since both BUGSY and Speedy expanded the same number of nodes, this additional time must be due to BUGSY's small amount extra overhead incurred from resorting and computing utility. We conclude that, barring this extra overhead, BUGSY

would have performed as well as the best performer for this utility function. In domains where node expansion and heuristic computation isn't so simplistic, this overhead would be insignificant.

5.6.12 Training Set Homogeneity

In Subsection 5.6.10 we showed that our weighted A* approach outperformed other techniques in all domains, with the notable exception of the platform domain; here, BUGSY was the best. Additionally, compared to other domains, the weighted A* technique performed relatively poorly on video game pathfinding (cf. Figure 5-10 where wA* is outperformed by the utility oblivious approaches at all points except for one). We believe that the poor performance of wA* on the platform domain, and its relatively poor performance on video game path finding is due to inability to create sufficiently homogeneous training sets. To verify this, we looked at the mean and standard deviation in the optimal path lengths for problems in all of our domains. The optimal path length can be viewed as a proxy for problem difficulty, and a high standard deviation in this statistic points to a diverse set of instances—some very easy to solve, and some quite difficult. For both the platform and video game path finding domains, the standard deviation in optimal path length was greater than 50% of the mean; more than twice that of the other domains. This evidence supports our hypothesis that weighted A*'s performance can be greatly hindered in situations where a representative training set is not available.

5.7 Related Work

Because BUGSY uses estimates of its own search time to select whether to terminate or continue, and to select which node to expand, it may be said to be engaging in metareasoning, that is, reasoning about which reasoning action to take. There has been much work on this topic in AI since the late 1980s (Dean & Boddy, 1988) and continuing today (Cox & Raja, 2011).

Dean and Boddy (1988) consider the problem faced by an agent that is trying to respond

to predicted events while under time constraints. Unlike our setting, their concern is with allocating time between prediction and deliberation. To solve this type of time-dependent planning problem, they suggest the use of (and also coined the term) *anytime algorithms*. Unlike the anytime-based techniques discussed previously, which attempt to find a stopping policy to optimize a utility function, Dean and Boddy used anytime algorithms as a means for allowing different allocations of time between predicting and deliberation. Later, Boddy and Dean (1989) show how anytime algorithms and their time-dependent planning framework can be used by a delivery agent that must traverse a set of waypoints on a grid, by allocating time between the ordering of waypoints and the planning used to travel between them. Dean, Kaelbling, Kirman, and Nicholson (1993) also adapt the technique for scheduling deliberation and execution when planning in the face of uncertainty.

Garvey and Lesser (1993) present *design-to-time* methods that advocate using all available time time to find the best possible solution. Unlike anytime approaches that can be interrupted at any time, the design-to-time method requires the time deadline to be given upfront. This way, the algorithm can spend all of its time focusing on finding a single good solution, instead of possibly wasting time finding intermediate results. Design-to-time also differs from *contract* techniques like DAS (Dionne et al., 2011), because in the design-to-time framework there must be a predefined set of solvers with known (or predictable) solution times and costs. The design-to-time method will select an appropriate solver for the problem and deadline, possibly interleaving different solvers if deemed appropriate.

Hansen, Zilberstein, and Danilchenko (1997) show how heuristic search with inadmissible heuristics can be used to make anytime heuristic search algorithms. Like the techniques presented in this chapter, they consider the problem of trading off search effort for solution quality. To this end, they propose one possible optimization function for anytime heuristic search search that attempts to maximize the rate at which the algorithm decreases solution cost. Like the anytime monitoring technique shown in Section 5.3.2, their evaluation function relies on learning the profile of the anytime algorithm offline. In their analysis on the 8-puzzle, they conclude that, while their method had good anytime behavior, there was lit-

tle benefit of using instead of trial-and-error-based hand tuning. This is not surprising given the strong performance demonstrated by offline-tuned weighted A* in our experiments.

More recently, Thayer et al. (2012b) proposed an approach for minimizing the time between solutions in anytime algorithms. They demonstrate that their new state-of-the-art algorithm performs well on a wide variety of domains, and can be more robust than previous approaches. Like BUGSY, their technique relies on using d heuristics to estimate the search effort required to find solutions. However, they only focus on solutions that will require the least amount of effort, and do not optimize for a trade off of search time for solution cost.

Decision-theoretic A* (DTA*, Russell & Wefald, 1991) is a utility-cognizant algorithm for concurrent planning and execution. It is based on ideas from real-time heuristic search (Korf, 1990). Unlike traditional real-time search, where each action is emitted after a fixed amount of search, DTA* decides when to stop searching and emit an action using a decision-theoretic analysis. At any time there is a single best top-level action with the lowest cost estimate. The search emits this action if it is decided that the utility of emitting the action outweighs the utility of further search. DTA* uses an approximation (found by off-line training) of how the solution cost estimate for each top-level action improves with additional search. Using a consistent heuristic, this estimate can only increase (Nilsson, 1980), so DTA* stops searching when it decides that the time required to raise the best action's estimated cost to the point that it is no longer the best action will be more costly than the expected gain from determining that there is a different best action.

Compared to BUGSY, DTA* is relatively myopic because it only considers the cost of search involved in selecting individual actions. DTA* does not consider the additional search required by the solution path to which it commits by choosing an action. Where BUGSY uses both d and expansion delay to reason about the required search effort for the entire path beneath a node, DTA* only reasons about the search required to determine the best action to emit right now.

DTA* was designed for cases where it is acceptable to emit actions before an entire plan has been formulated. In some situations, however, this is not acceptable as it may

lead an agent into a dead-end from which it can no longer reach its goal. Examples of domains with dead-ends include robotics, manufacturing (Ruml et al., 2011), and spacecraft control: exactly those applications involving high value or danger where automation is most worthwhile. In these cases, it is desirable to find an entire plan, guaranteed to reach the goal, before any execution begins.

Hernández, Baier, Uras, and Koenig (2012) introduce a model for evaluating path-finding algorithms for video games, called the *game time model*. The game time model partitions time into uniform intervals, and the agent can execute a single action during each interval. Path planning happens in parallel with execution, and the goal is to move the agent from its start location to its goal location in as few time intervals as possible, minimizing goal achievement time, the same objective that we discuss in Section 5.1. The game time model is a special case of the utility functions considered in this chapter, where solution cost is given in units of time.

One big difference between our work and that of Hernández et al. (2012), is that they focus on real-time heuristic search algorithms and allow search and execution to happen in parallel. The parallelism provides two benefits: first, it may be possible to reduce the number of intervals used by allowing planning and execution to happen at the same time, and second, the agent can start moving toward its goal right away—a necessary property for video games. As we mentioned before, however, in the face of dead ends it is prudent to have a complete plan before any execution begins. Real-time heuristic searches also make decisions based on very local information and consequently find more costly solutions. In their results, Hernández et al. report that their best approach solves problems on initially-known grid maps in about the same number of time intervals as A*. In our results, the utility-cognizant techniques outperformed A* for utility functions that were not heavily focused on solution cost.

Other related work about using metareasoning about time to control combinatorial search has been done in the area of constraint satisfaction problems (CSPs), and boolean satisfiability (SAT). Tolpin and Shimony (2011) use rational metareasoning to decide when

to compute value ordering heuristics in a CSP solver. The focus of the work was on value ordering heuristics that gave solution count estimates; the solver only bothered to compute the heuristic at decision points where it was deemed worthwhile. Their experiments demonstrate the new metareasoning variant outperformed both the variant that always computed the heuristic and one that computed the heuristic randomly. Horvitz, Ruan, Gomes, Kautz, Selman, and Chickering (2001) apply Bayesian structure learning to CSPS and SAT problems. They consider the problem of quasi-group completion, and unlike Tolpin and Shimony (2011) who use on-line metareasoning to control search, they use off-line Bayesian learning over a set of hand-selected variables to predict whether instances will be long or short running.

There has been a lot of work on attempting to estimate the size of search trees off-line (Burns & Ruml, 2013; Knuth, 1975; Chen, 1992; Kilby, Slaney, Thiébaux, & Walsh, 2006; Korf et al., 2001; Zohavi, Felner, Burch, & Holte, 2010). We discussed these techniques in detail in Chapter 4. This is clearly a related topic, as it is concerned with estimating search effort before an entire search has been performed. One may imagine leveraging such a technique to predict search time in an algorithm like BUGSY. Unfortunately, these estimation methods can be rather costly in terms of computation time, so they are not suitable as an estimator that is needed at every single node generation. Another possibility is to use off-line estimations to find parameters of that affect the performance of search on a given domain. This knowledge could be helpful for creating the representative training sets used by algorithms like weighted A* and anytime monitoring, which require off-line training.

5.8 Conclusion

We have investigated utility-cognizant search algorithms that take into account a user-specified preference trading off search time and solution cost. We presented three different techniques for addressing this problem. The first method was based on previous work in the area of learning stopping policies for anytime algorithms. To the best of our knowledge,

we are the first to demonstrate these techniques in the area of heuristic search. The second method was a novel use of algorithm selection for bounded-suboptimal search that chooses the correct weight to use for weighted A* on a given utility function. Finally, we presented the BUGSY algorithm, the only technique of the three that does not require off-line training.

We performed an empirical study of these techniques, investigating the effect of their parameters on performance, and comparing the different techniques to each other. Surprisingly, the simplest technique of learning a weight for weighted A* was able to achieve the greatest utility, outperforming the conventional anytime monitoring approach. Also, surprisingly, BUGSY, an algorithm that does not use any off-line training, performs just as well as ARA* with a stopping policy learned on thousands of off-line training instances. If representative set of training instances is not available then BUGSY is the algorithm of choice. Overall, the utility-cognizant methods outperformed both A* and Speedy search for a wide range of utility functions. This demonstrates that heuristic search is no longer restricted to optimizing solely solution cost, freeing a user from the choice of either slow search times or expensive solutions.

CHAPTER 6

CONCURRENT PLANNING AND EXECUTION

6.1 Introduction

It is often desirable to allow for concurrent or interleaved planning and execution. This is especially true in applications such as robotics and video games where an agent is expected to begin acting right when it is given a task, and likely before it has finished planning. The parallelism inherent in this setting can also be exploited by an algorithm that is attempting to reduce the sum of planning and execution times. For example, it may be beneficial to begin an action before thoroughly exploring plans that extend from it; while executing that action there can be an abundance of time for further planning before the next action must be returned. As a more concrete example, consider cooking: you may start boiling a pot of water before you have even decided what meal to make because most meals require boiling water and water requires a lot of time to boil (de Pomiane, 1930).

In this chapter, we focus on real-time heuristic search algorithms (Korf, 1990). However, unlike much previous work on this topic, we do not assume that planning and acting are interleaved; instead, we allow them to happen in parallel. We consider a few different techniques, based on previously proposed real-time heuristic search algorithms, that can reduce overall goal achievement time. We evaluate these new approaches in the context of minimizing goal achievement time, and show that occasionally they perform better than the offline techniques from Chapter 5. Then we consider approach that use metareasoning. We evaluate the decision-theoretic A* algorithm, showing that it is easily outperformed by simpler real-time searches, and we discuss some reasons why this may be the case. Then we describe the sketch of a new algorithm called Ms. A*. Ms. A* accounts for uncertainty in the current cost-to-goal estimate for different actions available to the agent. It also

takes advantage of *identity actions*, a feature of a real-time problem that allows one to adjust the amount of planning performed before an action is emitted without violating real-time constraints in some states. Like DTA*, Ms. A* is outperformed by simpler real-time methods, but it introduces important ideas for metareasoning in real-time search.

6.2 Previous Work

There has been much work in the area of real-time search since it was initially proposed by Korf (1990). In this section we will review some of the most popular real-time search algorithms.

6.2.1 Learning Real-time A*

In real-time search (Korf, 1990), an agent is allowed a fixed amount of time to plan before it must perform each action. Typical real-time search algorithms are considered *agent-centered search* algorithms, because the agent performs a bounded amount of *lookahead* search rooted at its current state before acting. Since the size of each lookahead search is bounded, the agent can respect the real-time constraints by restricting its lookahead to be completed by the time that the real-time limit has been reached. In his seminal paper, Korf (1990) presented the Learning Real-time A* (LRTA*) algorithm, a complete, agent-centered, real-time search algorithm. To select the next action to perform, LRTA* uses the action costs and an estimate of the cost-to-goal, or heuristic value, for the states resulting from applying each of its current applicable actions; it chooses to execute the action that has the lowest estimated cost-to-goal.

LRTA* estimates the heuristic value for states in two different ways. First, if a state has never been visited before then it uses a depth-bounded depth-first lookahead search. The estimated cost of the state is the minimum f value among all leaves of the lookahead search. The second way that it estimates cost is via learning. Each time LRTA* performs an action, it learns an updated heuristic for its current state. If such a state is encountered again, the learned estimate is used instead of search. Korf proved that as long as the previous

state's heuristic estimate is increased after each move by an amount bounded from below by some ϵ,[1] then the agent will never get into an infinite cycle, and the algorithm will be complete. In the original algorithm, the second best action's heuristic is used to update the cost estimate of the current state before the agent moves.

Local Search Space Learning Real-time A* (LSS-LRTA*, Koenig & Sun, 2009) is one of the most popular real-time search algorithms. LSS-LRTA* has two big advantages over the original LRTA*: it has much less variance in lookahead times, and it does significantly more learning. LRTA* can have a large variance in its lookahead times because, even with the same depth limit, different searches can expand very different numbers of nodes due to pruning. Instead of using bounded depth-first search beneath each successor state, LSS-LRTA* uses a single A* search rooted at the agent's current state. The A* search is limited by the number of nodes that it will expand, so there is significantly less variance in lookahead times. Also, the original LRTA* only learns updated heuristics for states that the agent has visited; LSS-LRTA* learns updated heuristics for every state expanded in each lookahead search. This is accomplished using Dijkstra's algorithm to propagate more accurate heuristic values from the fringe of the lookahead search back to the interior before the agent moves. Koenig and Sun showed that LSS-LRTA* can find much cheaper solutions than LRTA* and that it is competitive with a state-of-the-art search for pathfinding, D*Lite (Koenig & Likhachev, 2002).

Another major difference between LRTA* and LSS-LRTA* is how the agent moves. In LRTA*, after each lookahead, the agent moves by performing a single action; in LSS-LRTA* the agent moves all of the way to the node on the fringe of its lookahead search with the lowest f value. This has the result that agent performs many fewer searches to arrive at a goal. When search and execution are allowed to happen in parallel, the movement method of LSS-LRTA* can actually be detrimental to performance, as the agent performs many actions with little planning. In Subsection 6.4.1, we will see that if a fixed lookahead is used it is better to take only single steps after each lookahead, as more planning is put into each

[1] Russell and Wefald (1991) recognized the necessity of an ϵ lower bound

step. We will also see that the multi-step approach can be better if the lookahead size is allowed to vary based on the length of the previously executed action sequence.

6.2.2 Time-bounded A*

Time-bounded A* (TBA*, Björnsson, Bulitko, & Sturtevant, 2009) is a non-agent-centered, real-time, search algorithm. Instead of performing a bounded amount of lookahead search from the agent's *current* state, TBA* maintains a single A* search from the agent's *initial* state to the goal state. During each iteration, a fixed number of expansions are done on this single search. When these expansions are complete the agent moves one action along the path from the initial state to the node on the A* frontier with the lowest f value. Since the agent may have already moved away from the initial state during previous iterations, and because A* vacillates between many different paths, the agent's current state may not be along the current best path. If this occurs, the agent must backtrack toward the initial state until it is on the current best path. While backtracking, the agent is guaranteed to intersect any path from the initial state, because initial state is on all such paths, and, in the worst case, the agent can backtrack all of the way to the initial state. This procedure repeats until a path is found from the start to the goal state, in which case, the agent will backtrack to this path (if it is not already on it) and it will move to the goal. In their experiments, Björnsson et al. showed that, on grid pathfinding benchmarks, TBA* requires fewer iterations to find the same quality paths as other real-time algorithms such as LRTA*.

6.2.3 Game Time Model

Instead of considering only solution cost, Hernández et al. (2012) introduce the *game time model* for evaluating search algorithms. In the game time model, time is divided into uniform intervals. During each interval, an agent has three choices: it can spend the time interval searching, it can spend the time interval executing an action, or it can spend the the time interval doing both search and execution in parallel. The objective of the game time model is for the agent to move from the initial state to a goal state using the fewest time intervals.

The advantage of the game time model over the previous approach of comparing algorithms on their solution cost is that it allows for comparisons between real-time algorithms that search and execute in the same time step and off-line algorithms like A* that search first and execute only after all search has completed. Off-line algorithms often find lower-cost paths (A* finds optimal paths), but they do not allow search and execution to take place during the same time interval; they always search for a full solution before any execution begins. Real-time algorithms tend to find higher-cost paths, but they can be more efficient by allowing search and execution to occur in parallel. Experimentally, Hernández et al. show that, of the different real-time search algorithms tested, TBA* tends to better optimize the game time objective: it uses about the same number of time intervals as A*.

In their evaluation of algorithms using the game time model, Hernández et al. (2012) only considered utility-oblivious algorithms. As we saw in Chapter 5, when optimizing for goal achievement time, utility-aware techniques can perform much better. In their experiments on 512×512 grid pathfinding problems, Hernández et al. found that TBA* performed about as well as A* in terms of the number of time intervals until a goal was reached. In Chapter 5, we saw that Bugsy, auto-tuned weighted A*, and ARA* with monitoring all greatly outperformed A* for utility functions that consider not only execution cost but search time too.

6.2.4 Decision-theoretic A*

Decision-theoretic A* (DTA*, Russell & Wefald, 1991) is a utility-aware agent-centered search algorithm. DTA* handles concurrent planning and execution by deciding when to stop searching and execute each individual action using a decision theoretic analysis. It weighs the the cost of search against the benefit using a user-specified preference given as a parameter r: the ratio between the cost of a node generation to the cost of executing an action. This form of preference can easily be expressed as a utility function in the form described in Section 5.2.2. Unlike previous algorithms discussed in this section, DTA* is not a real-time algorithm; it may search all of the way to the goal before executing any

actions if that appears to be the best course of action.

To decide whether or not to search more, DTA* considers the search cost required to displace the current incumbent action. At any point during a lookahead search, one of the top-level actions will have the lowest cost-to-goal estimate. This action is the incumbent solution; it would be used for execution if the lookahead were to be stopped. Russell and Wefald argue that the benefit of searching is to decide whether the action that currently appears to be the best is, in fact, the best, or if another action should be executed instead. With a consistent heuristic, additional search will only increase the backed up heuristic estimates for top level actions (Nilsson, 1980). The only way to show that the action that currently has the lowest cost estimate is not the best action is to search beneath it until its backed up value is greater than that of the second-best action. DTA* chooses when to stop performing lookahead search by deciding whether or not the cost of the search required to raise the best action's estimated will be more than the expected gain from determining that a different action is better. The difficulty in this approach is estimating how additional search will change the cost of the best action. DTA* learns this information by using offline training to estimate the distribution in cost estimate increases given different depths of lookahead search.

Related to the work on DTA*, Horvitz and Rutledge (1991) present a system called Protos that they used to experiment with with metareasoning procedures for belief network inference. Like DTA*, Protos used myopic metareasoning to estimate the expected value of computation and to determine whether it should spend more time deliberating or whether it should act right away. This provides an example of how ideas used for metareasoning in search can also find application in other domains as well.

6.3 Goal Achievement Time

Chapter 5 considered algorithms that optimize utility functions defined in terms of planning time and execution time. To briefly review, a utility function is of the form $U(s,t) = -(w_f \cdot g^*(s) + w_t \cdot t)$, where s is a solution, $g^*(s)$ is its cost, t is the search time required to

find it and w_f and w_t are the user-defined weights specifying the preference relating search time and solution cost. In this chapter, we consider at a specific instantiation of this idea: goal achievement time (occasionally referred to as GAT). By definition, search time, t, is given in units of time. When the solution cost, $g^*(s)$ is also expressed in units of time then an agent is said to be minimizing the goal achievement time, that is, the time from when the agent is given a planning problem until the time that the problem is solved. This includes both the time spent planning the actions that will be executed and the time required to execute them—precisely the objective of planning under time pressure.

When t and $g^*(s)$ both express time, the parameters w_f and w_t effectively convert these two terms to equivalent units. For example, if search time is given in seconds and action cost is given in tenths of seconds, then $w_t = 1$ and $w_f = 0.1$ will express the utility, $U(s, t)$, in seconds. This generalizes the game time model of Hernández et al. (2012) where time is discretized into fixed-size intervals. Instead, our utility-based approach allows for real-valued, continuous, time. This is useful, because many domains do not have unit-cost actions. One example is eight-way grid pathfinding where diagonal moves cost $\sqrt{2}$—the Euclidean distance between the centers of diagonally adjacent 1x1 grid cells. Assuming that search time is given in seconds, if an agent is traveling with a constant velocity of v units-distance-per-second then we can optimize goal achievement seconds by simply setting $w_f = 1/v$ and $w_t = 1$.

Throughout this chapter, we will assume that search time is given in seconds, that $w_t = 1$, and that user-specified w_f converts the time units of action cost to seconds. This means that all of our utility values are given in seconds, and small values of w_f correspond to fast executing actions, and large values of w_f correspond to slower actions. This convention is useful for two reasons. First, it makes it possible to compute the amount of time that will elapse during the current executing trajectory of a real-time agent by simply multiplying the trajectory cost by w_f. And, second, it allows us to consider different search and execution trade offs in our experiments as was done in Chapter 5. When w_f is large, solutions will require more time to execute relative to the cost of search, so utility-cognizant algorithms

will prefer to plan more and execute less. When w_f is small, actions execute quickly and planning more may not be worth the time.

When computing goal achievement time in the case of parallel planning and execution, it is important to differentiate between planning that occurs on its own and planning that occurs in parallel with execution. If one were to simply sum the total planning and execution times, then the time spent planning and executing in parallel would be double counted. Most of the algorithms in this chapter (the only exception being DTA*) do all of their planning in parallel with execution except for the planning required to find the very first action. To computing goal achievement time for these algorithms, we simply add this short amount of initial planning time to the execution time.

6.4 Traditional Real-time Algorithms

Much previous work on real-time heuristic search has assumed that planning and acting are interleaved (Koenig & Sun, 2009). Instead, we explicitly consider the situation where planning and acting can happen in parallel. To handle domains with dynamics, where the state of the world is changing in a deterministic way as the agent is planning, we maintain the real-time constraint that the agent must have the next action ready by the time that the currently executing action is completed. In this section we look at two ways to improve the state-of-the-art, real-time, search algorithm LSS-LRTA*, when planning and execution can take place in parallel.

6.4.1 Single-move or Multi-move

An important step in a real-time search algorithm is selecting how to move the agent before the next phase of planning begins. In the original LRTA* algorithm (Korf, 1990), the agent moved a single step by selecting the action with the lowest cost-to-goal estimate. A new approach that has become quite common, is based on that of LSS-LRTA* (Koenig & Sun, 2009): the agent moves all of the way to the frontier node on the local search space with the lowest f value. At first, it may seem beneficial to commit to more than a single action at a

time, but, as we will see, this approach actually performs worse than just executing a single action when using a fixed amount of lookahead. This is because the agent must have each action ready by the time its currently executing action is complete, so if the next action is not ready then the agent cannot search for longer than the time of a single action execution. If the agent then moves all of the way to the frontier after planning for the duration of one action, then each action executed after the first does not happen in parallel with planning. This is less efficient, and leads to higher cost plans.

We consider two different solutions to this issue. The first is to execute single actions like LRTA*. If all of the agents actions have the same duration, then except for the time spent to find the first action, no time is wasted executing without also planning. Unfortunately, if the agent has actions with different durations, then the fixed lookahead size must be set conservatively to the minimum duration of all of the agent's available actions. The second approach we consider is to use a dynamic lookahead strategy. With a dynamic lookahead the agent selects the amount of lookahead search to perform based on the duration of its currently executing trajectory (recall that we can find this easily by multiplying the trajectory cost by w_f). If the agent commits to executing multiple actions, or if the agent's actions have differing durations, then it simply adjusts its lookahead to fill the entire execution time.

Both approaches require offline training to determine the speed at which the agent can perform lookahead search. In the first case, it is necessary to know the maximum lookahead size that the agent can search during the minimum action execution time. This can be found by simply running the search algorithm with different fixed lookahead settings on a representative set of training instances and recording the per-step search times. In the second case, with a dynamic lookahead, the agent must learn a function mapping durations to acceptable lookahead sizes. When the agent commits to a trajectory that requires time t to execute, then it must use this function to find $l(t)$, the maximum lookahead size that the agent can search in time t. Note that, because the data structures used during search often have non-linear-time operations, this function may not be linear. It is possible to

create a conservative approximation of $l(t)$ by running an algorithm on a representative set of training instances with a large variety of fixed lookahead sizes. The approximation of $l(t)$ selects the largest lookahead size that always completed within time t.

For our experiments, we used the platform domain, the 15-puzzle, four-way and eight-way grid pathfinding. All of these domains were described in greater detail in previous chapters. For all domains, we used 25 test instances. We used the offline training techniques described above to learn the amount of time required to perform different amounts of lookahead search. For the offline training, we generated 100 test instances fore each domain. The lookahead values used were 250, 500, 750, 800, 900, 1000, 2000, 4000, 5000, 6000, 7000, 8000, 10000, 12000, 16000, 32000, 48000, 64000, 96000, 128000, 192000, 256000, and 512000. For algorithms that use a fixed-size lookahead, the lookahead value was selected by choosing the largest lookahead size for which the maximum step time on the training instances was within a single action duration. If none of the lookahead values were fast enough to fit within a single action time, for a given w_f, then no data is reported.

Figure 6-1 shows a comparison of these different techniques for the LSS-LRTA* algorithm on the four search domains. The y axes in each plot show the goal achievement time as a factor of the optimal GAT. Optimal goal achievement time was computed using the time to execute the solution found by A*, but not counting the planning time required to find the solution. On the plot for the platform domain, the y axis is shown on a $\log_{10}$ scale. As in Chapter 5, we consider a variety of different utility functions, or w_f values; these are shown on the x axes on a $\log_{10}$ scale. As stated above, each w_f represents a different way of converting action costs to time. Smaller values for w_f represent an agent that can move relatively quickly, so spending a lot of time planning to make small decreases in solution cost may not be worth the time. For larger values, the agent moves more slowly, and it may be worth planning more to execute cheaper paths. Each point on the plot shows the mean goal achievement time over the 25 test instances, and error bars show the 95% confidence interval.

From these plots, it is clear that the multi-step approach is significantly worse than both

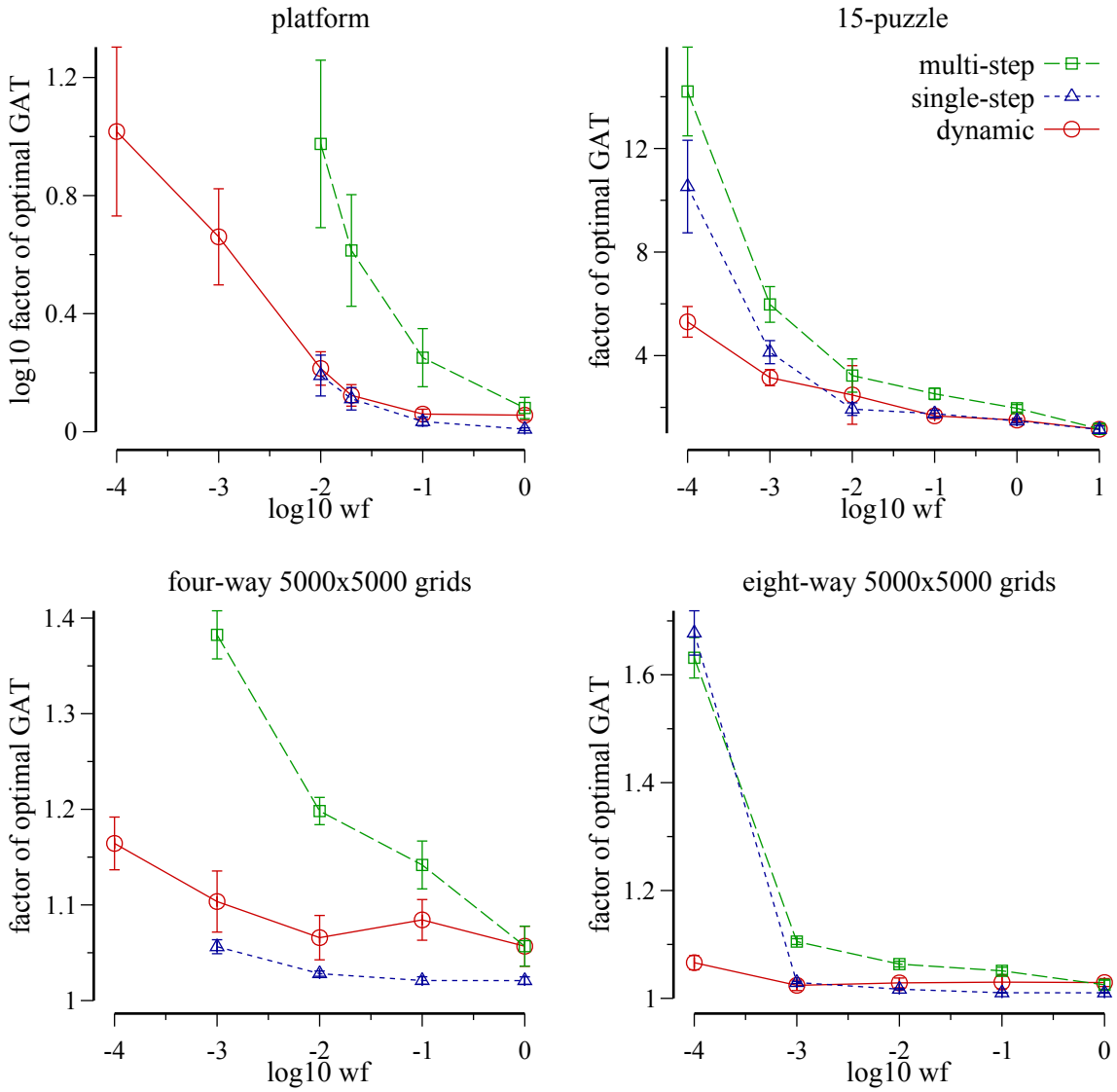


Figure 6-1: LSS-LRTA*: multi-step, single-step, and dynamic lookahead.

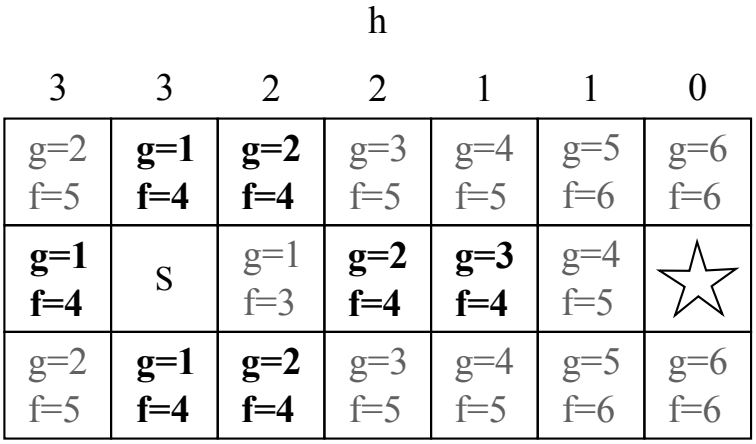


Figure 6-2: Example of heuristic error and f layers.

the single-step and the dynamic lookahead variants. This is likely because the multi-step technique commits to many actions with only a little bit of planning—the same amount of planning that the single-step variant uses to commit to just one action. In both the platform and 15-puzzle domains, there is a lot of overlap in the confidence intervals between the single step and dynamic lookahead techniques. However, on grid pathfinding executing just one action at a time performed significantly better.

6.4.2 Selecting Where To Move

In standard LSS-LRTA* the lookahead search is A*-based, so nodes are expanded in f order, and after searching the agent moves to the node on the open list that has the lowest f value. In this subsection, we see why this choice can be problematic, and we see how it can be remedied.

f-based lookahead doesn't account for heuristic error. The admissible heuristics used to compute f is, by definition, low-biased, so f will optimistically underestimate the true solution cost through each node. Because of this heuristic error, not all nodes with the same f value will actually lead toward the goal node. Figure 6-2 shows an example using a simple grid pathfinding problem. In the figure, the agent is located in the cell labelled 'S' and the goal node is denoted by the star. The admissible h values are the same for each column

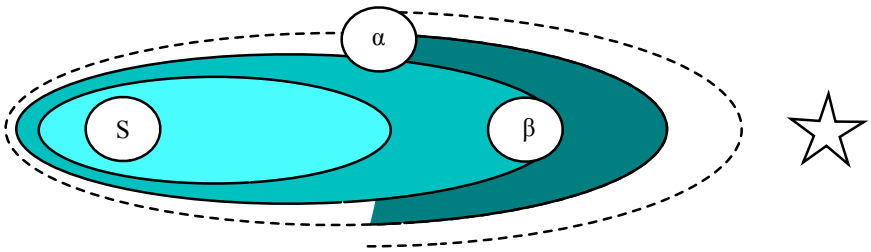


Figure 6-3: f-layered lookahead.

of the grid; they are listed across the top of the columns. The g and f values are shown in each cell. Cells with $f = 4$ are bold, and the rest are light gray. We can see that nodes with equivalent f values form elliptical rings around the start node. We call these rings of nodes with the same f value f *layers*. While some nodes in an f layer are closer to the goal node, there are many nodes in each layer that are not—some nodes in an f layer will be exactly away from the goal. In this simple problem, the optimal solution is to move the agent right until the goal is reached, however, of the 7 nodes with $f = 4$, only 2 nodes are along this optimal path; the other nodes are not, but they have the same f value because of the heuristic error. If the agent were to move to a random node with $f = 4$, chances are it will not be following the optimal path to the goal.

One way to alleviate this problem is to use a second criterion for breaking ties among nodes with the same f value. A common tie breaker is to favor nodes with lower h values as, according to the heuristic, these nodes will be closer to the goal. We can see, in Figure 6-2 that among all nodes in the $f = 4$ layer, the one with the lowest h value ($h = 1$) is actually along the optimal path. In LSS-LRTA* this tie breaking is insufficient, because when LSS-LRTA* stops its lookahead it may not have generated all of the nodes in the largest f layer. If the node with $h = 1$ was not generated then, even with tie breaking, the agent will be led astray.

These incomplete f layers cause other problems too. Recall that, in LSS-LRTA* the agent moves to the node at the front of the open list. If low-h tie breaking is used to order the expansions of the local search space, then the best nodes in the first f layer on the open

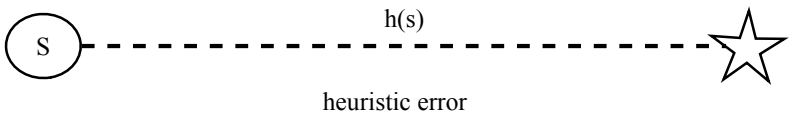


Figure 6-4: A standard heuristic and its error.

list will actually be expanded first and will not be on the open list when it comes time for the agent to move. Figure 6-3 shows the problem diagrammatically. As before, the agent is at the node labelled 'S' and the goal is denoted by the star. Each ellipse represents a different f layer, the shaded portions show closed nodes, darker shading denotes nodes with larger f values, and the dotted lines surround nodes on the open list. As we can see, the closed nodes with the largest f values cap the tip of the second-largest f layer. This is consistent with low-h tie breaking where the first open nodes to be expanded and added to the closed list will be those that have the lowest h. These are the nodes on the portion of the f layer that are nearest to the goal. If the agent moves to the node on its open list with the lowest f value, tie breaking on low h, then it will select node α. This node does not lead up instead of right, the direction of the goal!

We have demonstrated two problems: 1) because of heuristic error, f layers will contain a large number of nodes, many of which do not lead toward the goal, and 2) even with good tie breaking, LSS-LRTA* often misses the good nodes because it only considers partial f layers when deciding where to move. Next we present two solutions.

The first is quite simple. When choosing where to move, select the lowest h value on the completely expanded f layer with the largest f value, not the next node on open. In Figure 6-3, this corresponds to the node labelled β. We call this the "complete" technique, as it considers only completely expanded f layers instead of partially expanded, incomplete layers.

The second technique explicitly accounts for heuristic error and orders the search and moves to a node not on f, but on a less-biased metric: $\hat{f}$ (Thayer et al., 2011). With a consistent and admissible heuristic, the error in the heuristic estimate can only decrease

along a path toward the goal node Russell and Wefald (1991), and the heuristic error of the goal node is zero. Following Thayer et al. (2011), we make the simplifying assumption that the error in the heuristic is distributed evenly among each of the actions on the path from a node to the goal. Figure 6-4 shows the default heuristic value for a node S. It's error is accrued over the distance from S to the goal.

Instead of using the lower-bound estimate f we would prefer to use an unbiased estimate that accounts for and attempts to correct heuristic error. We call this estimate $\hat{f}$. Like f, $\hat{f}$ attempts to estimate the solution cost through a node in the search space. Unlike f, $\hat{f}$ is not biased—it is not a lower bound. $\hat{f}$ is computed similarly to f, however, it attempts to correct for the heuristic error by adding in an additional term:

$$\hat{f}(n) = \overbrace{g(n) + h(n)}^{f} + \overbrace{d(n) \cdot \epsilon}^{\text{error}}$$

where ϵ is the single-step error in the heuristic, and the additional term $d(n) \cdot \epsilon$ corrects the error by adding ϵ back to the cost estimate for each of the $d(n)$ steps remaining from n to the goal.

As we saw in Chapter 5, d estimates are readily available for many domains; they tend to be just as easy to compute as heuristic estimates. To estimate ϵ, the single-step heuristic error, we use an average the difference in the f values between each expanded node and its best child. This difference accounts for the amount of heuristic error due to the single step between a parent node and its child. With a perfect heuristic, one with no error, the f values of a parent node and its best child would be equal; some of the f will simply have moved from h into g:

$$\begin{aligned} f(\mathit{parent}) &= f(\mathit{child}), \text{ so} \\ h(\mathit{parent}) &= h(\mathit{child}) + c(\mathit{parent}, \mathit{child}), \text{ and} \\ g(\mathit{parent}) &= g(\mathit{child}) - c(\mathit{parent}, \mathit{child}) \end{aligned}$$

Since g is known exactly, as is the cost of the edge $c(\mathit{parent}, \mathit{child})$, with an imperfect heuristic, any difference between $f(\mathit{child})$ and $f(\mathit{parent})$ must be caused by error in the

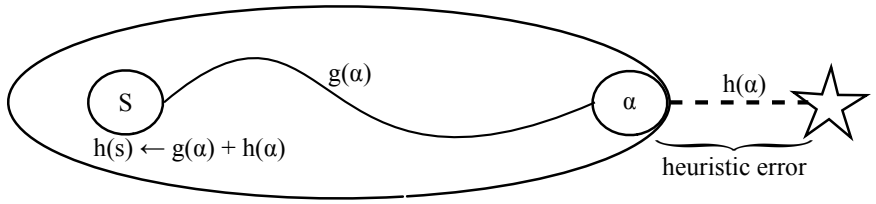


Figure 6-5: An updated heuristic and its error.

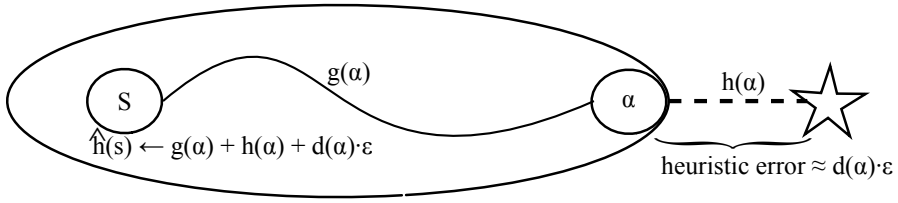


Figure 6-6: Using an updated heuristic and accounting for heuristic error.

heuristic over this step (Thayer et al., 2011). Averaging these differences gives us our estimate ϵ.

In real-time searches like LSS-LRTA*, the heuristic values of nodes that are expanded during a lookahead search are updated each time the agent moves. The updated heuristics are more accurate then the originals because they are based on the heuristic values of a node that is closer to the goal, and thus have less heuristic error. This is shown in Figure 6-5. Here, we can see that α is the node on the fringe from which the start state, S, inherits its updated heuristic value. Since $g(\alpha)$, the cost from S to α, is known exactly, the error in the backed up heuristic now comes entirely from the steps between α and the goal. Since α is closer to the goal, the error is less than than the error of the original heuristic value for S.

When computing $\hat{f}(S)$ in a real-time search, it is necessary to account for the fact that error in the updated heuristic comes from the node α. To do this, we track $d(\alpha)$ for each node with an updated h value, and we use it to compute $\hat{f}$. This is demonstrated by Figure 6-6. The updated heuristic, when accounting for heuristic error, is $\hat{h}(s) = g(\alpha) + h(\alpha) + d(\alpha) \cdot \epsilon$, where $g(\alpha) + h(\alpha)$ is the standard heuristic backup (cf Figure 6-5). Our new technique uses $\hat{f}$ to order expansions during the lookahead search in LSS-LRTA*, and it moves to the next

node on the open list with the lowest $\hat{f}$ value.

Figure 6-7 shows a comparison of the three techniques: the standard *incomplete* f-layer method of LSS-LRTA*, the *complete* f-layer method, and the approach that uses $\hat{f}$ (*fhat*). To better demonstrate the problem with the standard approach, all plots show results for the multi-step movement model that commits to a entire path from the current state to the fringe of the local search space after each lookahead. The style of these plots is the same as for Figure 6-1. In this figure, we can see that the variant of LSS-LRTA* that moved to the best node in the latest completely expanded f layer (labelled *complete*) nearly always had a better goal achievement time than the standard approach of moving to the next node on the open list. This was notably not true for eight-way grid pathfinding; a domain that has real-valued costs, which, as we saw in Chapter 4, leads to very small f layers, and thus is helped less by improved tie breaking. In addition, we see that the $\hat{f}$ method often performed as well as or better than the complete technique. This is especially true in the eight-way grid domain. We suspect that $\hat{f}$ does so much better on eight-way grids because, of all of these domains, it is the only one for which $d(n) \neq h(n)$, so the $\hat{f}$ method is using more information on eight-way grids than the other techniques.

6.4.3 Comparison of Real-time Techniques

We have shown modifications to the state-of-the-art, real-time, heuristic search algorithm LSS-LRTA*. Section 6.4.1 looked at different techniques for setting the lookahead size with respect to the number of executed actions. We saw that it was better to either execute a single action after each lookahead or to use a dynamically-sized lookahead than the standard approach of a fixed-size lookahead and committing to multiple actions. Then, Section 6.4.2 showed that an f-based lookahead can lead to poor action selection in some cases. We saw that using $\hat{f}$ instead of f was quite beneficial. Next, we compare the best combinations of these different real-time heuristic search methods.

Figure 6-8 shows the results of a comparison between the four combinations of single-step versus dynamic lookahead and f-based versus $\hat{f}$-based node ordering. On both the

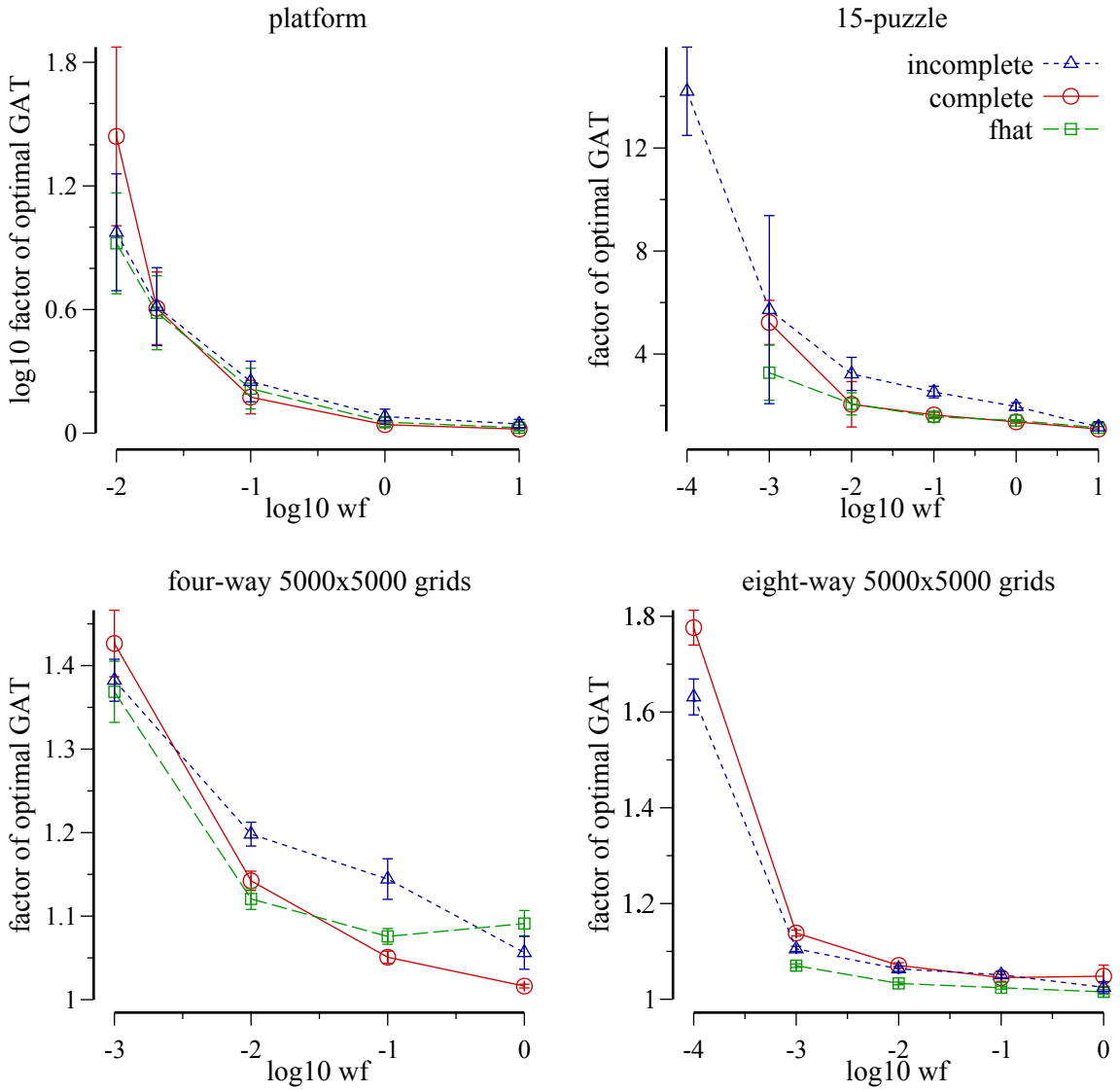


Figure 6-7: LSS-LRTA*: f-based lookahead and $\hat{f}$-based lookahead.

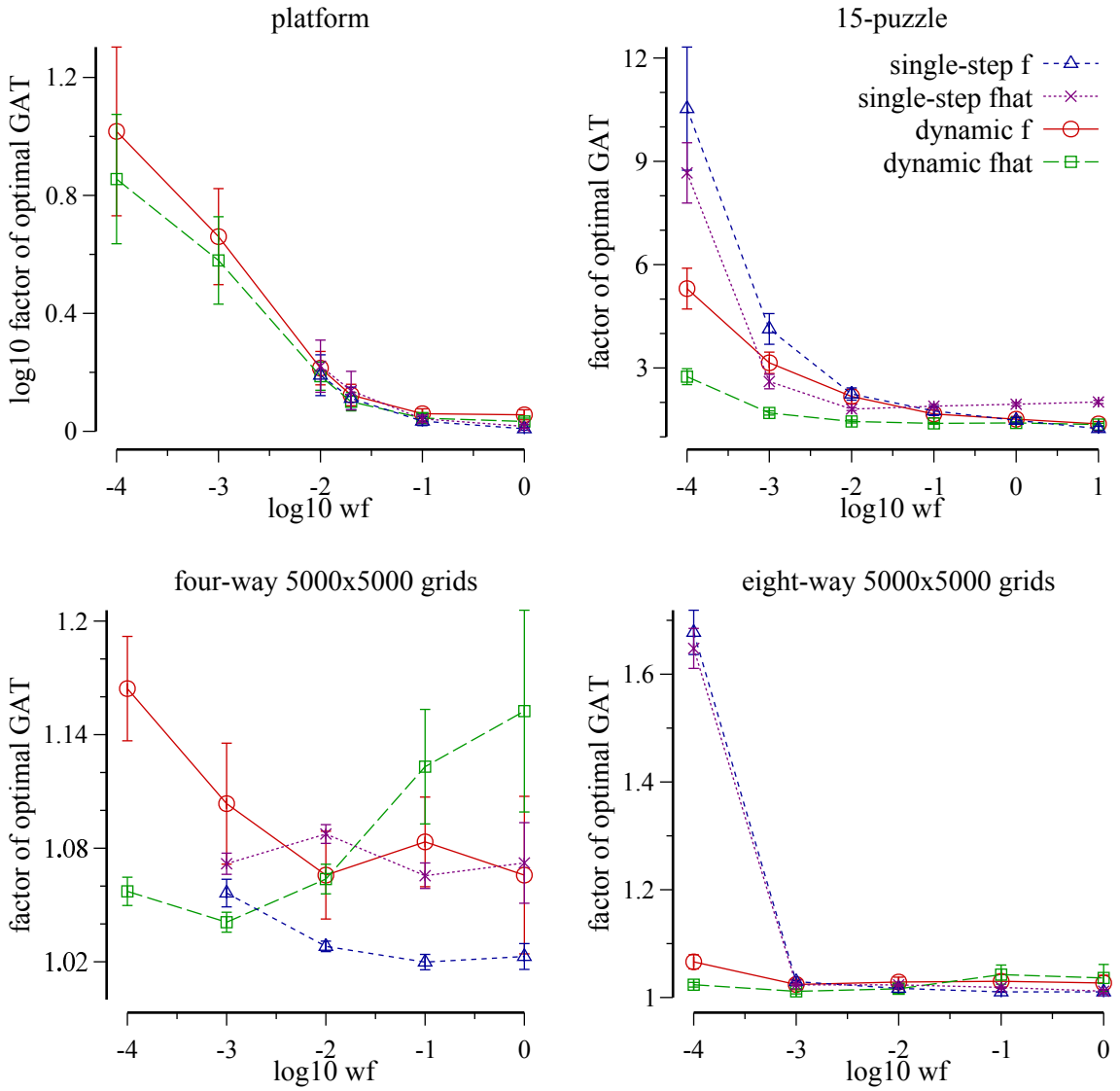


Figure 6-8: Comparison of real-time techniques.

platform and the 15-puzzle domains, $\hat{f}$ with dynamic lookahead tended to give the best goal achievements times in portions of the plots where all algorithms did not have significant overlap (i.e., everywhere except for the right-half of the plot for the platform domain). On the grid pathfinding problems, the $\hat{f}$ method with dynamic lookahead had the best goal achievement times for fast moving agents (smaller values of w_f), but single-step f-based lookahead was the best for slower agents (larger values of w_f). On grid pathfinding problems, when the agent was allowed to perform very large $\hat{f}$-ordered lookahead values, as occurs when the agent is slow and thus has a lot of time to search, it tended to find rather high-cost solutions. This did not happen on the platform domain or the 15-puzzle. $\hat{f}$ always found low-cost solutions compared to the single-step f-ordered technique. These results can be seen in Figure 6-10, in the next subsection.

From this experiment, we conclude that two techniques, one using a dynamic lookahead with $\hat{f}$ and the other performing single actions with the standard f-based ordering, were the best.

6.4.4 Comparison With Off-line Techniques

Chapter 5 presented techniques for planning under time pressure that found entire paths to the goal before executing any actions. The hypothesis of this chapter is that by allowing planning and execution to take place simultaneously it should be possible to improve over these previous offline techniques. In this subsection, we compare the best-performing variants of LSS-LRTA* to A*, Speedy search, and Bugsy. Figure 6-9 shows the results of this comparison with the factor of optimal goal achievement time on the y axis, and Figure 6-10 shows the same algorithms with the solution costs on the y axis. Surprisingly, Bugsy, a completely offline approach, seems to perform quite well when compared to the new real-time searches from this section. On both the platform and 15-puzzle domains, Bugsy dominates the real-time approaches, though the dynamic $\hat{f}$ algorithm performs nearly as well. This was likely because Bugsy found very low-cost (near-optimal) solutions on these domains, as is seen by the top two plots in Figure 6-10.

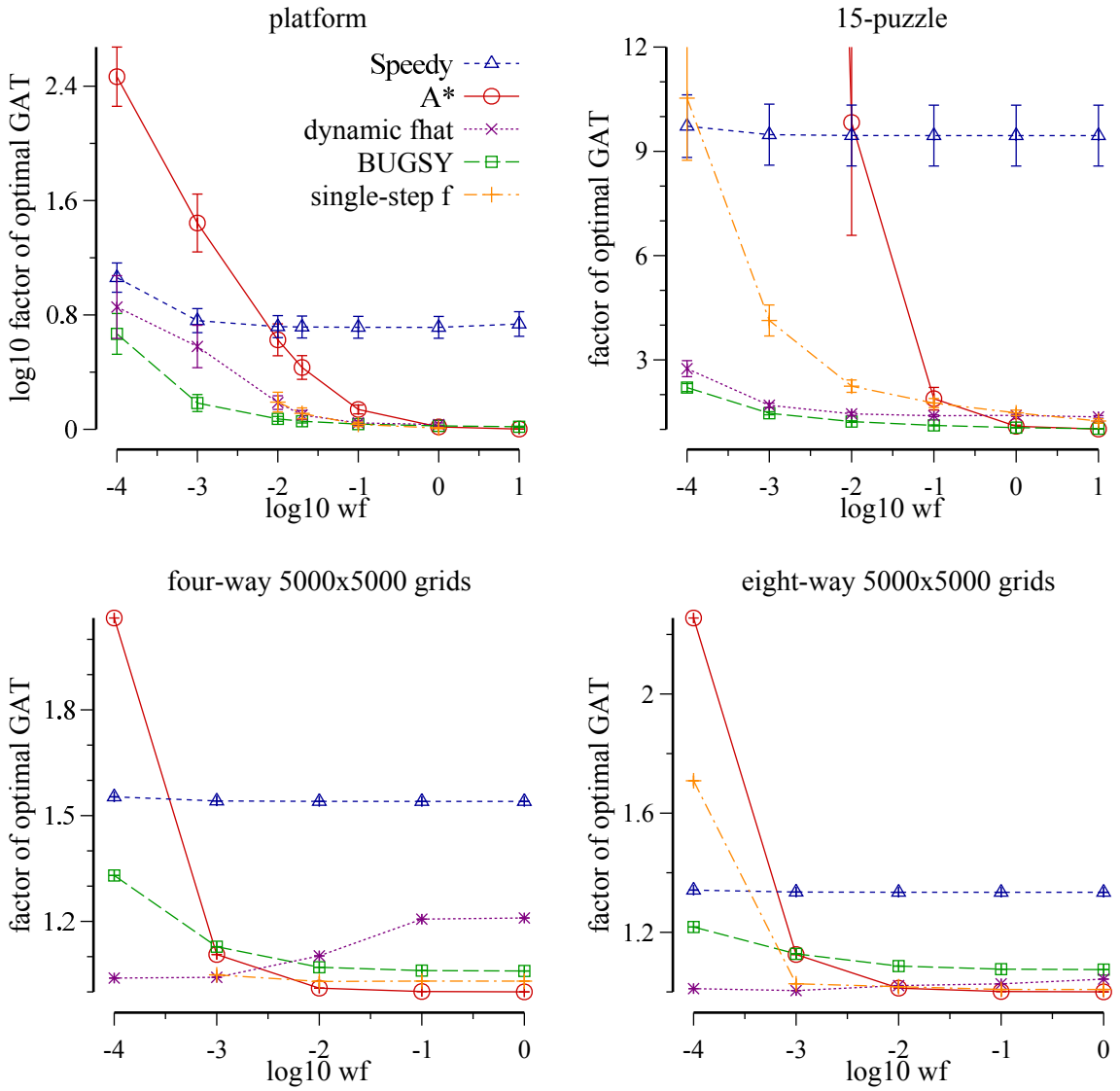


Figure 6-9: Comparison with off-line techniques.

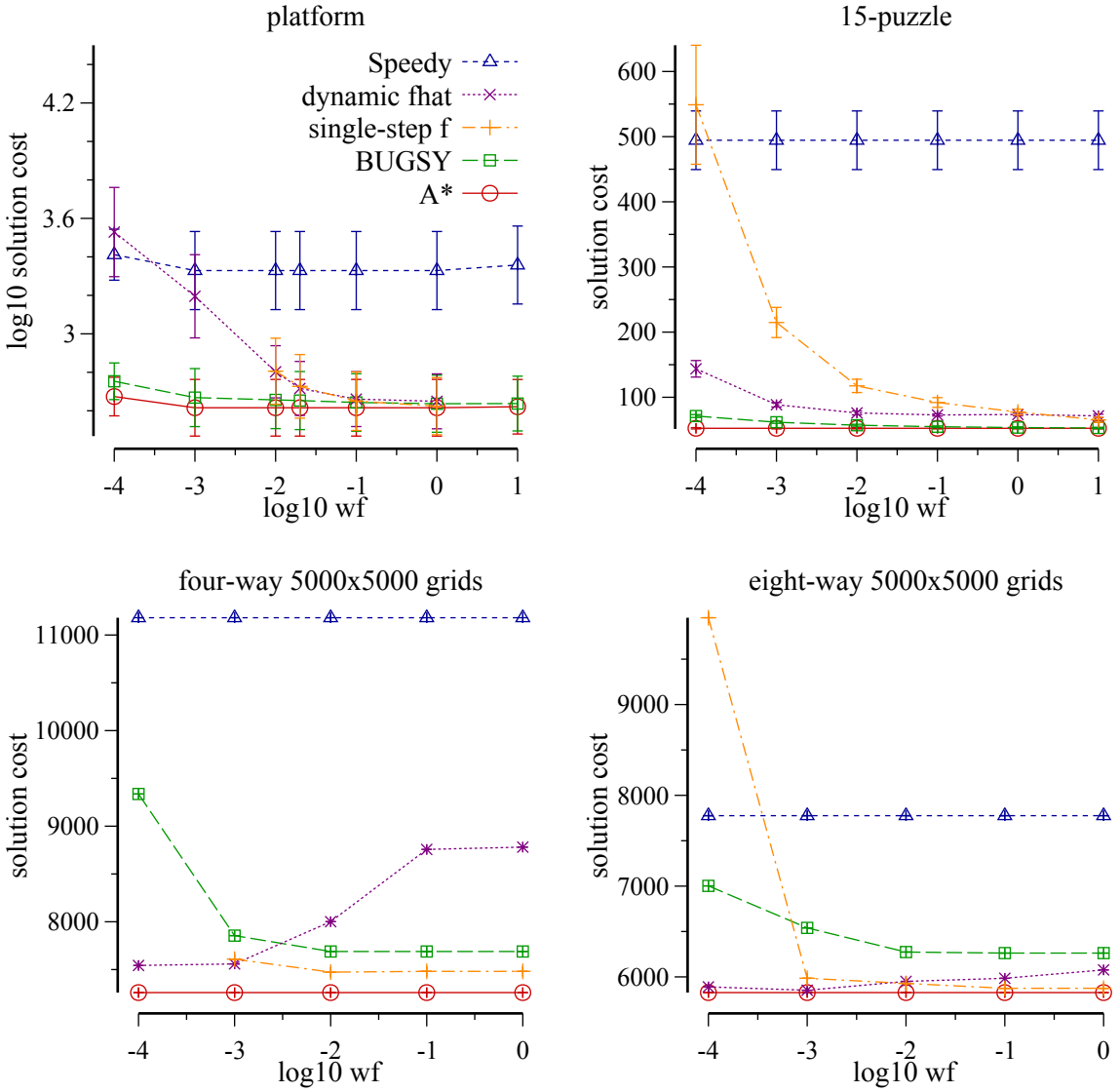


Figure 6-10: Solution costs for dynamic $\hat{f}$ and single-step f.

On the grid pathfinding problems, A* was the best technique for about half of the w_f values tested. This is because A* was able to find solutions to these problems fairly quickly, and its solutions were optimal. For small values of w_f, where the agent moved relatively quickly and execution time mattered less than planning time, A* performed very poorly and both single-step f and dynamic $\hat{f}$ gave the best performance. Looking at the solution costs shown in Figure 6-10, we can see that, on grid problems, the real-time searches found lower cost solutions than BUGSY. Their required planning times also remained low, as the only time they plan without also executing is during their very first lookahead search used to find first action to execute.

6.4.5 Summary

This section considered real-time search in the context of minimizing goal achievement time. LSS-LRTA* is one of the most-popular, state-of-the-art, real-time search algorithms. We saw how the standard LSS-LRTA* algorithm can be improved upon in two different ways: 1) by performing single-steps instead of multiple actions per-lookahead or by using a dynamic lookahead, and 2) by using $\hat{f}$ to explicitly *unbias* the heuristic estimates, and to account for heuristic error. Experimentally, we saw that in the platform domain and the 15-puzzle, BUGSY an offline technique gave the best goal achievement times . In grid pathfinding problems, for cases where the agent has only very little time for planning, the dynamic $\hat{f}$ approach was the best at minimizing goal achievement time. In the next section we turn to real-time algorithms that perform *metareasoning* to decide whether they should continue searching more at the current decision point or whether they should stop searching and act.

6.5 Metareasoning Real-time Algorithms

All of the techniques presented in the previous section used all available time performing lookahead search. For the single- and multi-step approaches, this was a fixed number of expansions selected to take approximately the amount of time searching as is required to execute a single action. For the dynamic lookahead approach, the amount of search varied,

but it was chosen such that it would occupy the entire amount of time that the agent would be executing its current trajectory. In many cases, it may be useful to either stop searching before the deadline has been reached, or, if possible, to search beyond the deadline. If the next action is obvious then there is no point in doing more search instead the time could be spent at the next decision point. On the other hard, if the decision is non-obvious then it may be desirable to search more to ensure that the correct choice is made.

Metareasoning is the process of thinking about thinking. When applied to heuristic search, metareasoning involves solving very simple reasoning tasks to decide what action a search algorithm will perform next. For example, in a real-time search setting, metareasoning could be used to decide between the two actions: search and execute. In this section we look at two different algorithms for metareasoning in real-time search. The first is the DTA* algorithm (Russell & Wefald, 1991) discussed briefly in Section 6.2.4. The second algorithm is a new algorithm called Ms. A*,[2] it is a modification to the $\hat{f}$ variant of LSS-LRTA* that decides when to search more and where to move based on the belief distribution over the f values of the actions at each decision point. We will see that, unfortunately, both DTA* and Ms. A* are outperformed by the simpler approaches from the previous section.

6.5.1 Decision-theoretic A*

DTA* was the inspiration for much of the work in this chapter. However, we found that it actually does not perform very well compared to modern real-time heuristic search algorithms. This is surprising because DTA* is utility-aware whereas algorithms like LSS-LRTA* are not. Also, because DTA* is not a real-time algorithm it is less constrained than LSS-LRTA*, which should provide it with an additional advantage. As described in Section 6.2.4, DTA* decides whether or not to search by considering the cost of search and the potential for additional search to show that the current best action is not actually the

[2]Ms. A* does not stand for anything; it's simply a proper name. The predecessor of Ms. A* was an algorithm called Metareasoning A*, which was shortened to Mr. A*. Mr. A* didn't work well enough to even warrant discussion.

best. If it estimates that the benefit of search is high, DTA* may search all of the way to the goal before executing any actions. It never considers a real-time bound. We would expect that this gives it an advantage because it can decide to search more when presented with a difficult decision, where LSS-LRTA*, or another real-time search algorithm, may be forced to act. In this subsection, we will see how DTA* performs compared to modern real-time heuristic search algorithms, what features of modern real-time search allow them to have better performance, and how to fix DTA* to be competitive with state-of-the-art approaches.

First we compare DTA* to A*, Speedy search, and the best two real-time search variants from the previous section. Because DTA* performed poorly, this experiment used smaller instances than the previous ones. For the platform domain we used 25x25 mazes instead of 50x50, and on grid pathfinding we used 100x100 grids instead of 5000x5000. DTA* was able to solve 15-puzzle instances,—this was one of the domains originally used by Russell and Wefald (1991)—so we kept the standard instances for that domain. Since DTA* is not real-time, and it may plan for longer than the duration of its currently executing action, computing its goal achievement time requires determining for which time periods it is both planning and executing and for which it is only planning. To simplify this procedure, we made the optimistic assumption that DTA* always emitted its next action before its currently executing action completed—we pretended that it was real-time. The goal achievement time that is reported for DTA* is its execution time plus the time required for it to emit its first action. This is optimistic, because if DTA* ever "missed" a deadline then it got extra planning time for free. It should also be noted that, since it is not real-time, DTA* is not strictly applicable on the platform domain which has dynamics. If DTA* used more than the current execution time in this domain, then the agent may actually have moved (due to gravity) before the next action is given. This could have been solved by bounding DTA*'s searches to the time of the current execution, but as we will see, even with this advantage DTA* was still not able to outperform the utility-oblivious methods presented in the previous section.

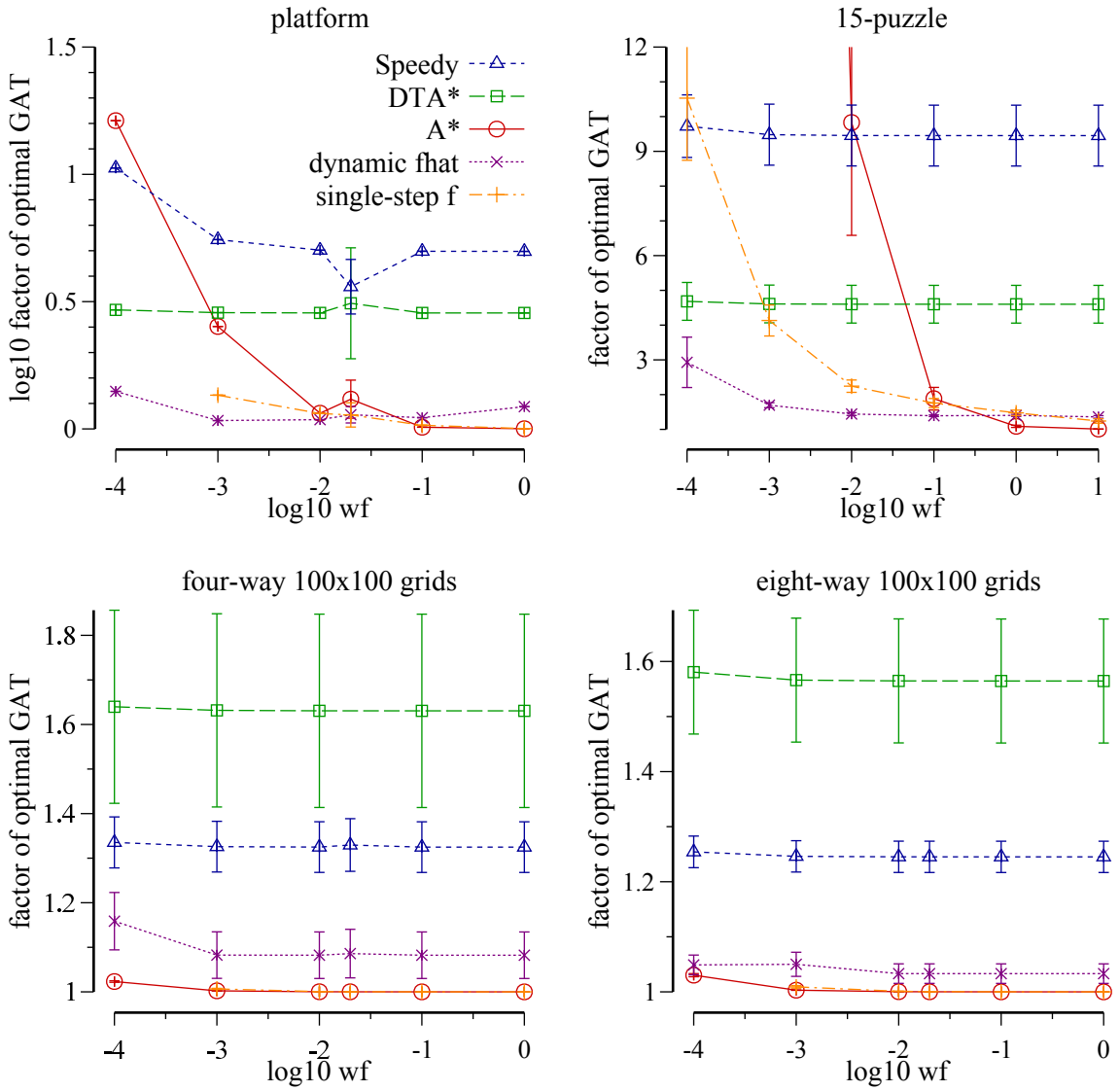


Figure 6-11: Comparison with DTA*.

The results of this comparison are shown in Figure 6-11. In all domains, both single-step LSS-LRTA*, and the variant of LSS-LRTA* using $\hat{f}$ and dynamic lookahead were able to reach the goal faster than DTA*. (Note that, in the plot for the grid pathfinding problems the line for single-step LSS-LRTA* is difficult to see because it is drawn directly on top of A*.) In the platform and 15-puzzle domains, DTA* was able to provide better goal achievement times than A* and Speedy search for some values of w_f. In grid pathfinding, however, both A* and single-step LSS-LRTA* gave the best goal achievement times. A* was able to very quickly find the optimal solution on these easier 100x100 grid instances. In all cases, DTA* performed worse than the utility-oblivious, real-time search methods presented in the previous section.

Two big differences between DTA* and state-of-the-art real-time search algorithms are discussed next.

Learning

Recall that one of the major advantages of LSS-LRTA* over previous algorithms, such as LRTA*, is that it learns updated heuristic estimates for all nodes expanded during its lookahead, not just one node. This additional information allows LSS-LRTA* to more escape heuristic local minima—portions of the state space that have lower heuristic values than their neighbors, but that do not actually lead to the goal. To an agent-centered search, local minima appear to be promising, so the agent will move into them even though they do not actually lead to a goal. In order to escape a heuristic minimum an agent-centered search must update the heuristic values for all states in the minimum until in recognizes that going back into the minimum is more costly than leaving it. By doing more learning, LSS-LRTA* does not wander around in the minima; it can escape relatively quickly, finding a much cheaper solution. DTA*, on the other hand, only learns an updated heuristic estimate for one node at each step: the node from which it just moved. This means that DTA* ignores a significant amount of this extra information from its lookahead search, and may, consequently, require significantly more actions to escape heuristic local minima.

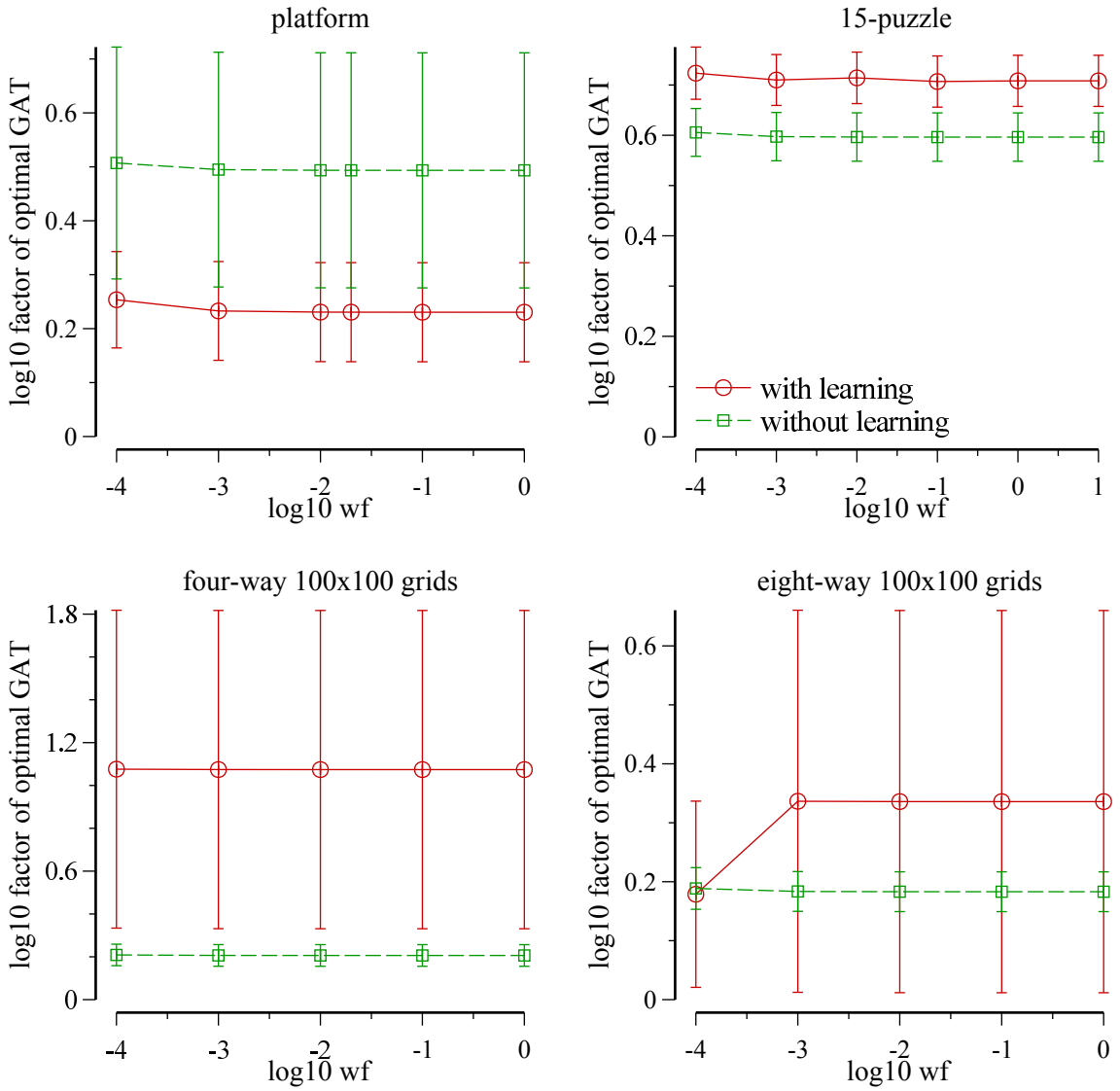


Figure 6-12: DTA* with and without extra learning.

One seemingly obvious solution is to modify DTA* so that it learns updated heuristic estimates for all nodes that it expands too. As shown in Figure 6-12, however, this can actually make the algorithm perform worse! These plots show the same smaller instances as Figure 6-11, however, all y axes are on a $\log_{10}$ scale. As we can see, the extra learning only improved DTA*'s performance in the platform domain. For all other domains, it performed worse. On the grid pathfinding problems, the variance in goal achievement times were also very high when using the extra learning. On the 15-puzzle and four-way grids the extra learning was significantly worse. On eight-way grids there is a lot of overlap in the confidence intervals, but the learning performs worse on the average.

One reason why DTA* may actually do better without the extra learning is because the additional heuristic learning may conflict with the learning that DTA* already uses to estimate the value of search. Recall that DTA* estimates the value of search by using training data gathered offline on a representative set of problem instances. For different lookahead depths and initial heuristic values, DTA* learns a distribution over the increase in backed up heuristic estimates. This information is then used to estimate whether or not the value of further search will outweigh its cost. Since the training data is conditioned on the default, un-updated, heuristic estimates, it is almost guaranteed that changing the heuristic values of all expanded nodes will invalidate the offline training; it will no longer be representative of the values actually encountered during search.

Multiple Local Search Spaces

Another reason why DTA* may not perform as well as variants of LSS-LRTA* is because it uses multiple local searches, one for each action, instead of a single local search space. Much of the design of DTA* assumes that the search spaces beneath each top-level action are disjoint. This assumption is true for trees, but most planning problems are actually graphs, which violate this property. By using separate search frontiers for each action in a graph, DTA* duplicates much of its search effort, expanding similar sets of nodes beneath each action.

Unfortunately, it is not clear how to modify DTA* to use a single search frontier. A fundamental aspect of DTA* is that it compares top-level actions by considering the amount of search required to raise the cost-to-goal estimate of the best action to be greater than that of the second-best action. Crucial to this, is the fact that reasoning can be done over the search frontiers beneath each action individually—something that cannot be done without separate search frontiers. DTA* cannot be made to use a single frontier.

To factor out only the disadvantage of using multiple local search spaces, we implemented a variant of LSS-LRTA* that uses a different local search space (LSS) beneath each action. We compare this multiple local search space variant to the original single local search space version in Figure 6-13. The four plots in this figure show both single- and multi-LSS versions of the original multi-step LSS-LRTA* and the single-step LSS-LRTA*. We can see that, in all domains, the single local search space gave shorter goal achievement times than the corresponding version that uses a different local search space beneath each action. We conclude that it is better to use a single local search space if possible, as multiple local search spaces has an inherent disadvantage when searching on graphs.

Summary

In this subsection we looked at DTA*, a seemingly promising utility-cognizant, agent-centered search algorithm. Unfortunately, in our experiments, we found that DTA* was worse than state-of-the-art, utility-oblivious, real-time search algorithms. We discussed two reasons why these methods may outperform DTA*: 1) the newer methods learn updated heuristic estimates for significantly more nodes during each lookahead, and 2) DTA* has an innate disadvantage since it requires multiple local search spaces. Another reason why DTA* may not perform well on the domains tested is because they all have relatively small branching factors. In their evaluation Russell and Wefald (1991) claim that DTA*'s performance is very sensitive to the branching factor. In fact, on the 15-puzzle they used the hand-tuned value of 1.2 for the branching factor instead of the true asymptotic branching factor of 2.1304 (Korf et al., 2001). In our experiments we used the mean branching factor

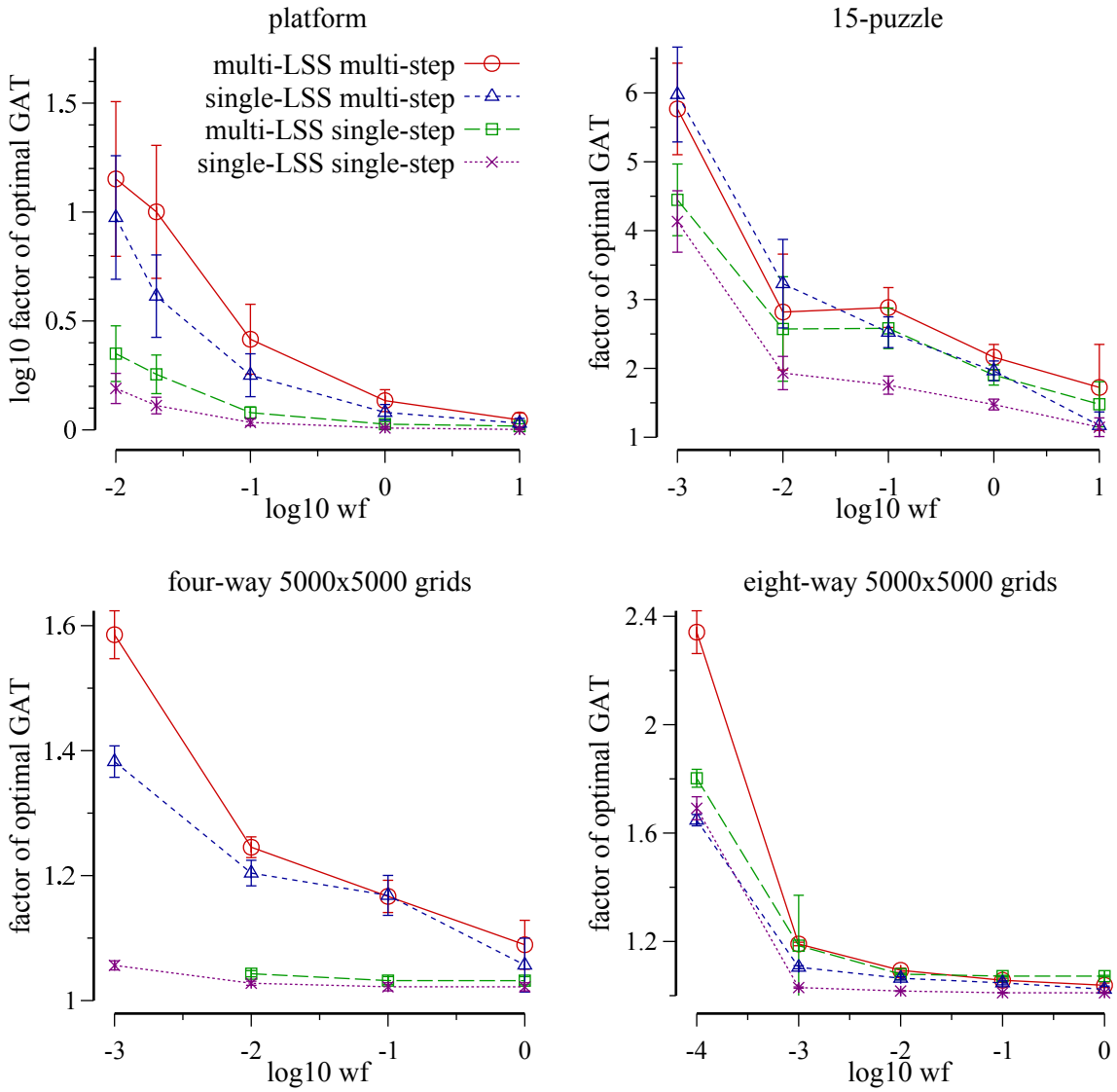


Figure 6-13: Single versus multiple local search spaces.

found during the offline training.

6.5.2 Ms. A*

In real-time search, an agent has a fixed amount of time during which it can plan, and when the time is up it must emit an action to be executed. Such constraints can arise in video games where the agent's planning time is constrained by frame rate requirements. Real-time constraints also exist in areas with dynamics, such as robotics. For a robot cruising down a hallway it is necessary for the the next action to be known before the current action completes, as it is dynamically infeasible for the robot to go from full-speed to a complete stop in order to finish planning its next move. In the previous subsection, we saw DTA*, a utility-cognizant algorithm that does not necessarily respect real-time constraints. This subsection considers a new algorithm called Ms. A*, which practices a simple form of metareasoning, that allows it to choose when to stop searching and how many actions to commit to before searching again. Ms. A* obeys real-time constraints, but it also has the ability to ignore the constraints for certain states where they are unnecessary. While the ideas behind Ms. A* are promising, like DTA*, it is unable to outperform the simpler techniques from the previous section.

Identity Actions

In certain domains, real-time constraints are required for some states but not for others. Recall the robot moving down the hallway. While the robot is driving, it must know its next action before its current action completes. Otherwise, if it does not have an action to execute then it will still be moving but without purpose. This is dangerous, because the robot may be heading directly toward a wall, or a cliff, or another dangerous obstacle. If the robot is at a complete stop, however, then it can plan for as long as it would like; it won't hit a wall because it is not moving (and we assume that all obstacles are static). In this case, if it decides not to execute any actions it will simply remain in its current state until it decides otherwise.

Situations like this also arise in video games. For example, consider a platform-style game where a player must jump from platform to platform to traverse a maze (e.g., the type of game represented by the platform domain). While in the air the agent must contend with gravity; it must plan its next action in time for the next frame of the game when its state will change and it will have fallen. If the agent is standing on a platform, however, then it can plan for as long as it likes; during each subsequent frame it can simply do nothing and it will remain in the same state. We call these "do nothing" operations *identity actions*, actions which, when executed, leave the agent in the same state. If the agent is in a state where an identity action is available then it can spend as much time as it deems necessary to plan its next move. When the real-time bound is reached, the agent can simply emit the identity action and continue planning. In states without an identity action, such as those where the agent is subject to dynamics, the agent must emit actions before the time expires, because if it chooses to do nothing the its state will change and it will need to begin planning from scratch at its new current state.

Current real-time heuristic search algorithms do not take identity actions into account. Instead, they expand a fixed number of nodes, learn updated heuristic values, move, and repeat. If the selected move is an identity action then they will still begin their subsequent searching from scratch expanding many of the same nodes over again. Instead, an algorithm that is cognizant of identity actions can, instead, reuse all of its previous expansions in order to lookahead deeper into the search space. When an identity action is available, the question becomes: when should the lookahead search cease and an action be emitted? As with DTA*, Ms. A* answers this question using metareasoning.

Belief About f^*

Ms. A* is inspired by the two techniques of interval estimation (Kaelbling, 1993) and Hoeffding races (Maron & Moore, 1994). It uses similar methods to both decide how much to search and where to move. In these approaches, simple intervals, such as the 95% confidence intervals, are used to define a likely range for an unknown value. The goal is to

find the best value—be it the success probability for an action, as in interval estimation, or the learning model with the least error, as in Hoeffding races. When deciding among a set of choices, the intervals are consulted to determine whether or not one choice should be selected now, or if more sampling is required before a sufficiently informed decision can be made. When the intervals overlap there is less certainty about which choice is truly better, and more sampling is required to reduce the uncertainty. When the intervals to not overlap, the choice represented by the better interval (the one with the higher probability of success, or the least expected error, etc.) can be used.

Unfortunately, it is not clear how to find a 95% confidence interval on our estimate of cost-to-goal in a search. Instead, Ms. A* uses f and $\hat{f}$ as its interval. Recall from Section 6.4.2 that $\hat{f}(n) = g(n) + h(n) + d(\alpha) \cdot \epsilon$, where α is the node from which the backed up heuristic value $h(n)$ was derived, and the last term accounts for heuristic error in the backed up estimate. Notice that $\hat{f}$ can also be defined in terms of f: $\hat{f}(n) = f(n) + d(\alpha) \cdot \epsilon$, and the difference between $\hat{f}$ and f is simply the estimated heuristic error. This justifies our choice of the bounds on the interval: if the difference between $\hat{f}$ and f is large then there was a lot of error, and if the difference is small then there is little error, and we are more certain in our estimate.

Another way to view this is as a belief distribution over the true cost-to-goal. If we consider the single-step heuristic error to be a random variable with an expected value of ϵ, then $\hat{f}(n)$ is the mean of our belief distribution over the true solution cost of n. We denote the true solution cost of n by $f^*(n)$. Note also that, since f is a lower bound, the distribution is truncated at one end by f.

Ms. A*considers three possible cases for the current belief distributions over f^* values, as shown in Figure 6-14. In all cases, α denotes the action with the lowest expected f^*, and β denotes the action with the second-lowest expected f^*. α is the action that would be selected for execution if search were to stop at this decision point—we don't expect any other action to be cheaper. But, what must be considered is the uncertainty in the distributions over f^* values. If there is little variance and the mean of the distribution for α

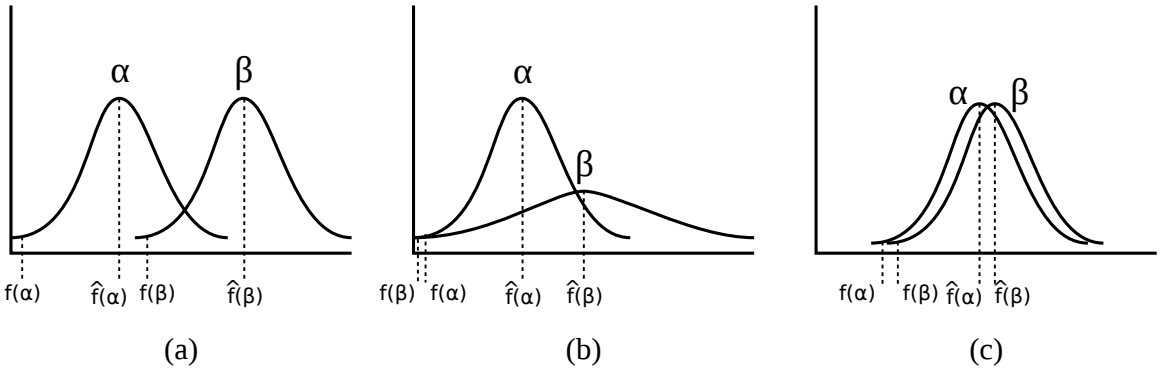


Figure 6-14: Estimated f distributions for best action, α, and the second best action, β.

is lower than that of β then search will provide little benefit, and the agent should execute α. This corresponds to Figure 6-14(a). On the other hand, if the distribution for β has a lot of variance—if the agent is unsure of its estimate—then it should search more to decrease the uncertainty. This corresponds to Figure 6-14(b). Lastly, if there is little variance in the distributions for both α and β and if they have quite similar means, then we would expect both α and β to have approximately the same outcome, and the agent might as well just execute α now without further search. This is shown Figure 6-14(c).

Ms. A* proceeds as LSS-LRTA* on $\hat{f}$; however, when a lookahead search is completed, it uses the cases shown in Figure 6-14 to determine the decision point for the next lookahead search. This is done by considering each possible decision point along the path to the best frontier node of the lookahead search. At each of these branches, Ms. A* considers the difference between the backed up $\hat{f}$ and f values for the best action, α, (where "best action" means the one with the lowest $\hat{f}$) and the second-best actions, β. The policy is: if $f(\beta) < f(\alpha)$ then search, otherwise perform action α. This condition will only be true for case (b) in Figure 6-14—precisely where more search is desired.

We discussed the importance of identity actions. States with identity actions allow a search algorithm to make a less constrained decision about how much search should be done. Without an identity action, the real-time constraint is enforced, and the agent must choose an action before the time limit is up. But, with an identity action, the agent can

simply choose to emit the identity action, stay in the same state, and continue searching deeper. Ms. A* takes advantage of identity actions. If the current state has an identity action available, then Ms. A* considers performing more search at the current decision point by comparing the f values of the current best and second-best actions as described above. If the uncertainty is great then Ms. A* will execute the identity action and search more at its current state. On the other hand, if there is no identity action the Ms. A* instead considers only future decision points starting from the state resulting from the application of its current best action. In this manner, it is real-time only when necessary.

Evaluation

Figure 6-15 shows the performance of Ms. A* compared to the utility-oblivious real-time algorithms and BUGSY. The plots show Ms. A* with two different lookahead techniques. One, labelled *Ms. A**, uses the greatest lookahead value that was able to complete a single search within the time of a single action before it considers moving. This method, ideally, should allow the algorithm to vary between the single-step and multi-step approaches depending on whether or not more search is deemed necessary at each step toward the frontier. The second variant, labelled *dynamic Ms. A**, uses a dynamic lookahead based on the number of actions committed on the previous movement. In both cases, Ms. A* exploited identity actions, so if its current state had an identity action then it considered staying still and searching deeper at its current decision point. In grid pathfinding and the 15-puzzle there are no dynamics, so every state had an identity action available. The platform domain has dynamics as the agent is affected by the force of gravity while falling. States where the agent is standing firmly on a platform had identity actions, allowing Ms. A* to search as long as necessary. States where the agent was in the air did not have an identity action, so Ms. A* needed to have its next action ready before the next frame of the game caused the agent to fall.

Overall, the performance of Ms. A* was not as good as expected. The dynamic Ms. A* algorithm tended to match the performance of dynamic $\hat{f}$ LSS-LRTA* on all domains

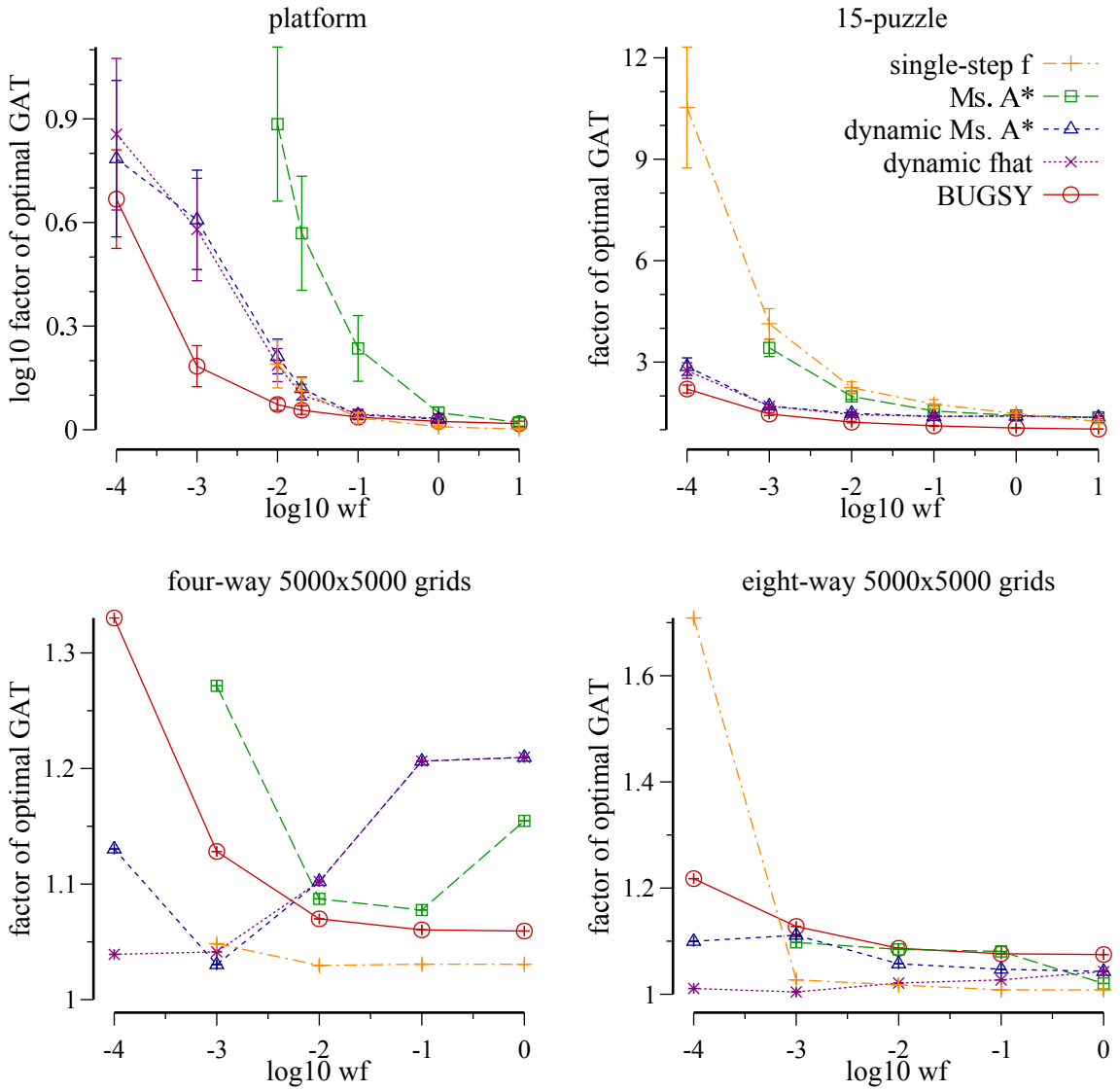


Figure 6-15: Comparison of Real-time searches, Ms. A*, and BUGSY.

except for eight-way grid pathfinding where it performed significantly worse. Ms. A* without dynamic lookahead seemed to have worse performance than single-step f-based LSS-LRTA*, except on the 15-puzzle where it was slightly better. We briefly discuss some possible reasons for its poor performance in the next section.

6.6 Discussion and Conclusion

We saw that there are a variety of ways to decide how long the agent should plan before committing to action execution in real-time search. The standard method of planning for a fixed amount of time before committing to an action trajectory that leads all of the way to the fringe of the lookahead search actually performs significantly worse than the alternatives. Simply committing to a single action after each lookahead works well when doing A*-based lookahead using f. On the other hand, when using $\hat{f}$, a metric that corrects for the estimated heuristic error, the dynamic lookahead approach of committing to multiple actions and searching for longer accordingly, worked best. When compared to offline techniques, these real-time methods worked well on grid pathfinding problems with fast moving agents—where planning time mattered more with respect to execution time. On the platform and 15-puzzle domains, however, Bugsy was the best performer; it was able to find low cost solutions very quickly, and provided the lowest goal achievement times of the algorithms tested.

While, surprisingly, we found that offline methods tended to better minimize goal achievement time than approaches that allowed planning and execution to happen in parallel, the latter methods are still quite important. In many cases offline algorithms are simply not applicable. For example, in the platform domain, the agent always started with its feet planted firmly on a platform—a state with an identity action. Instead, if the agent were to start in the air, then offline techniques would not work, as the agent would be subject to dynamics from the get-go.

We then considered algorithms that perform metareasoning. DTA* has been an inspiration for much of this work. Unfortunately, we saw in our experiments that DTA* was

worse than utility-oblivious techniques. We looked at two different reasons why DTA* was disadvantaged. First, it learns updated heuristic estimates only for one node each time it moves, whereas state-of-the-art real-time searches learn updated values for every expanded node. Second, DTA* relies heavily on multiple local search spaces, one for each action. On graphs, these different spaces will contain many of the same nodes, leading to significant duplication of search effort.

We then presented Ms. A*. In our experiments, Ms. A* did not perform as well as expected; its performance was essentially dominated by the utility-oblivious algorithms. Because the dynamic version of Ms. A* so nearly matched dynamic $\hat{f}$, it seems likely that Ms. A*, like dynamic $\hat{f}$, always decided to act and never stopped to search until it reached the frontier. This may indicate that the intervals we defined were too tight and rarely overlapped. It may be possible for future work in this direction to find a better definition of the intervals—perhaps intervals based on confidence bounds instead of f and $\hat{f}$, but it seems unlikely that confidence bounds on cost-to-go can be found efficiently.

In this chapter we saw techniques that allow for parallel search and execution. Unlike the offline techniques presented in previous chapters, the methods in this chapter allow an agent to begin executing actions before an entire path to the goal is found. When minimizing goal achievement time parallel planning and execution can be a major advantage as the agent is not charged for the entirety of its planning time separately from the time required to execute the plan. Instead, the only planning time accounted for is that which takes place while not executing in parallel. Real-time heuristic search algorithms are a natural fit for parallel planning and executing, because they must execute actions with only a bounded amount of planning time. Real-time search techniques allow planning and execution to take place in parallel, as the agent can plan its next action while its current action is executing. While the metareasoning approaches that we discussed did not work extremely well in practice, we believe that ideas such as identity actions and heuristic uncertainty will be important for a successful utility-aware, real-time search algorithm.

CHAPTER 7

CONCLUSION

Heuristic search is the fundamental technique underlying most modern automated planning systems. Unfortunately, due to time and memory limitations, it is usually impractical to solve planning problems optimally. On the other hand, greedy approaches, which can often find solutions rather quickly, can be too suboptimal, finding plans that are excessively costly. In many cases one simply cares about achieving the goal as quickly as possible regardless of whether the time is spent planning or executing. This thesis has focused on the topic of *planning under time pressure* in which the objective is to minimize the sum of planning time and execution time, not just one or just the other. We have seen many ways in which this problem can be approached, including minimizing planning time without increasing execution cost, predicting future search effort, optimizing a utility function given as a linear combination of search time and execution cost, and allowing planning and execution to take place in parallel.

In Chapter 2 we were introduced to the topic of parallel heuristic search. We saw how many simple approaches to parallelizing heuristic search fail, as they either duplicate too much search effort or spend too much time synchronizing between threads. The PBNF algorithm was introduced; PBNF uses a homomorphic abstraction function to partition the search graph. In PBNF, threads use the small, abstract representation of the graph to find disjoint portions of the space to search in parallel. This technique only requires synchronization on the $n$block graph data structure, and provides threads periods of communication-free search. The parallel search algorithms were also converted to bounded-suboptimal search and anytime search algorithms. We saw, experimentally, that PBNF-based techniques were faster and scaled better than previous state-of-the-art approaches to parallelizing heuristic

search.

To demonstrate the generality of the technology underlying PBNF, Chapter 3 showed how these methods could be applied to model checking. We saw that the PSDD algorithm, which uses the same state space partitioning technique as PBNF, was able to outperform hash-distributed search in terms of memory, time, and scalability when implemented in the Spin model checker, a state-of-the-art model checker widely used for software verification in industry. PSDD detects duplicate states immediately and allows for full partial order reduction, so it often required substantially less memory that hash distributed search. We also saw how PSDD can use external memory, such as hard disks, to reduce the memory requirement of model checking even further. These results are especially significant because memory is usually the limiting factor for model checking problems.

Planning under time pressure is concerned with planing time, and this is usually difficult to know in advance. Chapter 4 presented a new *incremental model* that can be used to predict the number of expansions that the IDA* search algorithm will perform for a given bound. We saw that, when trained offline on unit-cost 15-puzzle problems, the incremental model was able to make predictions with accuracy that was competitive with the current state-of-the-art predictor on that domain while using fewer samples. In addition, the incremental model can make predictions on problems with real-valued costs, where the previous techniques cannot. Then, we saw how the incremental model can be trained online and used to control an IDA* search. While IDA* and IDA*$_{CR}$ failed outright on some problems, IDA*$_{IM}$ gave robust performance on all domains tested.

In addition to optimizing cost, heuristic search has been applied to a variety of different objective functions (cf Table 1-1). Chapter 5 focused on algorithms that optimized utility functions given as linear combinations of search time and solution cost. Such functions can directly address the objective of planning under time pressure. The main contributions of this chapter were four-fold. First, we saw how automated parameter selection could be used with bounded-suboptimal search to find a setting for the bound, specific to a given utility function, to best optimize for utility. Second, to the best of our knowledge, we were the

first to combine anytime heuristic search with anytime monitoring. Anytime search finds a stream of solutions of decreasing cost, and the monitoring policy decides which solution to return for a given utility function. Third we saw the BUGSY algorithm. Unlike the previous two techniques, BUGSY requires no offline training data, yet surprisingly, it is still very competitive. Fourth, we evaluated these three techniques on a variety of different benchmarks. Overall, the simple parameter turning technique gave the best performance, but on some domains, such as the 15-puzzle or the platform domain, BUGSY is the algorithm of choice. If no training data is available then BUGSY is always the algorithm of choice.

In previous chapters algorithms found complete plans, and planning took place before any execution was allowed to begin. In Chapter 6 we considered methods where planning may happen in parallel with execution. We saw two techniques for improving the state-of-the-art, real-time, search algorithm LSS-LRTA*. For utility functions that prefer shorter planning times, these new techniques were able to outperform BUGSY in terms of goal achievement time. On some domains and for utility functions that valued low solution costs, however, BUGSY tended to be better. We also looked at two agent-centered search algorithms that use metareasoning: DTA* and Ms. A*. While neither algorithm performed extremely well, the ideas that were discussed will be important for a successful utility-aware, real-time search.

The overarching contribution of this dissertation is the recognition of and explicit consideration for the trade off between planning time and solution cost. When time is of the essence, it is undesirable to spend more time looking for a shorter plan than the amount of time saved by the decreased plan length; and, it is undesirable to spend more time executing a longer plan than the amount of time saved by the decreased planning time. When under time pressure, an automated agent should explicitly attempt to minimize the sum of planning and execution times, not just one or just the other.

Appendix A

PSEUDO-CODE FOR SAFE PBNF

In the following pseudo code there are three global structures. The first is a pointer to the current incumbent solution, *incumbent*, the second is a *done* flag that is set to true when a thread recognizes that the search is complete and the third is the $n$block graph. The $n$block graph structure contains the list of free $n$blocks, *freelist* along with the σ and σ_h values for each $n$block. For simplicity, this code uses a single lock to access either structure. Each thread also has a local *exp* count. The *best* function on a set of $n$blocks results in the $n$block containing the open node with the lowest f value.

```
SEARCH(INITIAL NODE)
1.   insert initial node into open
2.   for each p ∈ processors, THREADSEARCH()
3.   while threads are still running, wait()
4.   return incumbent

THREADSEARCH()
1.   b ← NULL
2.   while ¬done
3.      b ← NEXTNBLOCK(b)
4.      exp ← 0
5.      while ¬SHOULDSWITCH(b, exp)
6.         m ← best open node in b
7.         if m > incumbent then prune m
8.         if m is a goal then
9.            if m < incumbent then
10.              lock; incumbent ← m; unlock
11.        else if m is not a duplicate then
12.           children ← expand(m)
13.           for each child ∈ children
14.              insert child into open of appropriate nblock
15.        exp ← exp + 1
```

```
ShouldSwitch(b, exp)
1.  if b is empty then return true
2.  if exp < min-expansions then return false
3.  exp ← 0
4.  if best(freelist) < b or best(interferenceScope(b)) < b then
5.    if best(interferenceScope(b)) < best(freelist) then
6.      SetHot(best(interferenceScope(b)))
7.    return true
8.  lock
9.  for each b′ ∈ interferenceScope(b)
10.   if hot(b′) then SetCold(b′)
11. unlock
12. return false

SetHot(b)
1.  lock
2.  if ¬hot(b) and σ(b) > 0
3.    and ¬∃i ∈ interferenceScope(b) : i < b ∧ hot(i) then
4.      hot(b) ← true
5.      for each m′ ∈ interferenceScope(b)
6.        if hot(m′) then SetCold(m′)
7.        if σ(m′) = 0 and σ_h(m′) = 0
8.          and m′ is not empty then
9.            freelist ← freelist \ {m′}
10.       σ_h(m′) ← σ_h(m′) + 1
11. unlock

SetCold(b)
1.  hot(b) ← false
2.  for each m′ ∈ interferenceScope(b)
3.    σ_h(m′) ← σ_h(m′) − 1
4.    if σ(m′) = 0 and σ_h(m′) = 0 and m′ is not empty then
5.      if hot(m′) then
6.        SetCold(m′)
7.        freelist ← freelist ∪ {m′}
8.      wake all sleeping threads

Release(b)
1.  for each b′ ∈ interferenceScope(b)
2.    σ(b′) ← σ(b′) − 1
3.    if σ(b′) = 0 and σ_h(b′) = 0 and b′ is not empty then
4.      if hot(b′) then
5.        SetCold(b′)
6.        freelist ← freelist ∪ {b′}
7.      wake all sleeping threads
```

NextNblock(b)

1. if b has no open nodes or b was just set to hot then lock
2. else if *trylock()* fails then return b
3. if $b \neq$ NULL then
4. $\quad$ *bestScope* $\leftarrow$ *best*(*interferenceScope*(b))
5. $\quad$ if $b <$ *bestScope* and $b <$ *best*(*freelist*) then
6. $\quad\quad$ unlock; return b
7. $\quad$ Release(b)
8. if ($\forall l \in$ *nblocks* : $\sigma(l) = 0$) and *freelist* is empty then
9. $\quad$ *done* $\leftarrow$ true
10. $\quad$ wake all sleeping threads
11. while *freelist* is empty and $\neg$*done*, sleep
12. if *done* then $n \leftarrow$ NULL
13. else
14. $\quad$ $m \leftarrow$ *best*(*freelist*)
15. $\quad$ for each $b' \in$ *interferenceScope*(m)
16. $\quad\quad$ $\sigma(b') \leftarrow \sigma(b') + 1$
17. unlock
18. return m

Appendix B

TLA$^+$ MODEL: HOT NBLOCKS

This is model used to show that Safe PBNF is live-lock free. Refer to Section 2.3.2.

MODULE *HotNblocks*

EXTENDS *FiniteSets*, *Naturals*

CONSTANTS *nnblocks*, *nprocs*, *search*, *nextblock*, *none*

VARIABLES *state*, *acquired*, *isHot*, *Succs*

$Vars \triangleq \langle state, acquired, isHot, Succs\rangle$

$States \triangleq \{search, nextblock\}$

$Nblocks \triangleq 0 \,..\, nnblocks - 1$

$Procs \triangleq 0 \,..\, nprocs - 1$

ASSUME $nnblocks \geq nprocs \land nprocs > 0 \land nnblocks > 1 \land none \notin Nblocks \land Cardinality(States) = 2$

$Preds(x) \triangleq \{y \in Nblocks : x \in Succs[y]\}$ Set of predecessors to *Nblock* x

$IntScope(x) \triangleq Preds(x) \cup \text{UNION } \{Preds(y) : y \in Succs[x]\}$ The interference scope of x

$IntBy(x) \triangleq \{y \in Nblocks : x \in IntScope(y)\}$ Set of *Nblocks* which x interferes.

$Busy(A) \triangleq A \cup \text{UNION } \{Succs[x] : x \in A\}$ Set of *Nblocks* which are busy given the set of acquired nblocks

$Overlap(x,\, A) \triangleq A \cap IntScope(x)$ Set of *Busy Nblocks* overlapping the successors of x

$Hot(A) \triangleq \{x \in Nblocks : isHot[x] \land Overlap(x,\, A) \neq \{\}\}$ Set of all hot nblocks given the set of acquired nblocks

$HotInterference(A) \triangleq \text{UNION } \{IntScope(x) : x \in Hot(A)\}$ Set of *Nblocks* in interference scopes of hot nblocks

$Free(A) \triangleq \{x \in Nblocks : Overlap(x,\, A) = \{\} \land x \notin HotInterference(A)\}$ Free *Nblocks*

$Acquired \triangleq \{acquired[x] : x \in Procs\} \setminus \{none\}$ Set of *Nblocks* which are currently acquired

$OverlapAmt(x) \triangleq Cardinality(Overlap(x,\, Acquired))$ The number of nblocks overlapping x.

$doNextBlock(x) \triangleq$
- $\land$ UNCHANGED $\langle Succs\rangle$
- $\land\ state[x] = nextblock \land acquired[x] = none \Rightarrow Free(Acquired) \neq \{\}$
- $\land$ IF $Free(Acquired \setminus \{acquired[x]\}) \neq \{\}$ THEN
  - $\land \exists\, y \in Free(Acquired \setminus \{acquired[x]\}) : acquired' = [acquired \text{ EXCEPT } ![x] = y]$
  - $\land\ state' = [state \text{ EXCEPT } ![x] = search]$
  - $\land\ isHot' = [y \in Nblocks \mapsto \text{IF } y \in Free(Acquired \setminus \{acquired[x]\})$ THEN FALSE ELSE $isHot[y]]$
- ELSE
  - $\land\ acquired' = [acquired \text{ EXCEPT } ![x] = none]$
  - $\land\ isHot' = [y \in Nblocks \mapsto \text{IF } y \in Free(Acquired')$ THEN FALSE ELSE $isHot[y]]$
  - $\land$ UNCHANGED $\langle state\rangle$

$doSearch(x) \triangleq$
- $\land$ UNCHANGED $\langle acquired, Succs\rangle$
- $\land\ state[x] = search \land state' = [state \text{ EXCEPT } ![x] = nextblock]$
- $\land\ \lor$ UNCHANGED $\langle isHot\rangle$
- $\lor \exists\, y \in IntBy(acquired[x]) :\ \land \neg isHot[y]$
  - $\land\ IntScope(y) \cap Hot(Acquired) = \{\}$
  - $\land\ y \notin HotInterference(Acquired)$
  - $\land\ isHot' = [isHot \text{ EXCEPT } ![y] = \text{TRUE}]$

$Init \triangleq$
- $\land\ state = [x \in Procs \mapsto nextblock] \land acquired = [x \in Procs \mapsto none]$
- $\land\ isHot = [x \in Nblocks \mapsto \text{FALSE}]$

This is a basic graph where each nblock is connected to its neighbors forming a loop.

- $\land\ Succs = [x \in Nblocks \mapsto \text{IF } x = 0 \text{ THEN } \{nnblocks - 1,\, x + 1\}$ ELSE IF $x = nnblocks - 1$ THEN $\{0,\, x - 1\}$ ELSE $\{x - 1,\, x + 1\}]$

$Next \triangleq \exists\, x \in Procs : (doNextBlock(x) \lor doSearch(x))$

$Fairness \triangleq \forall\, x \in Procs : \text{WF}_{Vars}(doNextBlock(x) \lor doSearch(x))$

$Prog \triangleq Init \land \Box[Next]_{Vars} \land Fairness$

$HotNblocks \triangleq \forall\, x \in Nblocks : isHot[x] \leadsto \neg isHot[x]$ The property to prove

Appendix C

HISTOGRAMS

The incremental model makes use of fixed-size histograms to represent distributions over f and Δf values. To maintain as much accuracy as possible, our implementation uses two types of histograms: an exact *points* histogram for representing a small number of data points and an approximate *bins* histogram for representing a large number of data points. A points histogram is a list of the data points representing the distribution. Points histograms are completely accurate as they represents exactly every value in their distribution. When the number of data points added to the distribution reaches the histogram's size limit, it is converted to a bins histogram. A bins histogram is an array of fixed-width bins, where each entry is a floating point value that represents the amount of mass in the distribution for the range of values represented by the corresponding bin. A bins histogram is approximate but has the advantage of using a constant amount of memory to store and a constant amount of computation to manipulate. The size of the histogram represents the maximum number of points allowed before converting to bins and once converted to bins the size specifies the number of bins.

Our histogram implementation supports several operations: *add_mass* adds a point mass to the histogram; *add* returns a histogram representing the sum of a list of histograms; *convolve* returns a histogram that is the convolution two histograms as described in Section 4.3.2; *weight_left_of* returns the amount of mass in the histogram to the left of a given value; *total_weight* returns the total weight of the histogram; *normalize* normalizes the distribution such that the total weight is equal to a given value; and *prune_weight_right* prunes the mass in the histogram to the right of the given value.

When adding mass to a points histogram the new values are added as additional points. For

a bins histogram, the additional mass is 'sprinkled' proportionally across the bins containing values for which mass is being added. If mass is added to a value that extends beyond the domain of a bins histogram then the bins of the histogram are recomputed by increasing the bin width by integral multiple of the current width. When the bin width is increased in this manner, adjacent bins are merged and the newly emptied bins are free to accommodate the new values. When convolving two histograms, the result is a bins histogram if either of the operands are a bins histogram, or if they are both points histograms but the domain of the result has more values than the histogram size.

Appendix D

ANYTIME POLICY ESTIMATION

It can be challenging to write algorithms that rely on off-line training data. If the algorithm behaves unexpectedly, then it is unclear if there is a bug in the implementation, a bug in the off-line learning procedure, or if the training set is merely insufficiently representative. In this appendix, we describe how we implemented and verified our procedure for estimating an anytime profile.

Figure D-1 shows the pseudocode for building a profile, not given by Hansen and Zilberstein (2001). The algorithm accepts a set of solution streams as input, one stream for each solved instance, then proceeds in two steps. The first step is the COUNT-SOLUTIONS function that counts the number of times each solution cost was improved upon. The function iterates through each solution (line 5), computes the bin of a histogram into which its cost value falls (line 7), then for each subsequent solution a count is added to *qqtcounts* for each time step for which the first solution improved to the second solution. In addition, the number of total improvements for each solution and time bin are counted in the *qtcounts* array. The

$$\text{costbin}(q) = \left\lfloor \frac{q - q_{min}}{(q_{max} - q_{min})/ncost} \right\rfloor$$

and

$$\text{timebin}(\Delta t) = \left\lfloor \frac{\Delta t - \Delta t_{min}}{(\Delta t_{max} - \Delta t_{min})/ntime} \right\rfloor$$

functions bin cost and time values respectively by returning an integer for the corresponding index in the histogram.

The second step is the PROBABILITIES function that converts the counts computed in the first step into normalized probability values. This is achieved by dividing the number of Δt steps for which a solution of cost q_i improved to a solution of cost q_j (*qqtcounts*) by

the total number of steps for which a solution of cost q_i was improved (*qtcounts*, and lines 28). The probability values are "smoothed" by adding half of the smallest probability to each bin representing a solution cost improvement. This step removes zero-probabilities, allowing improvement to be considered. Finally, the probabilities are normalized so that the probability of all non-decreasing-cost solutions for each current cost and time step sum to one (lines 31–37). Once the profile is computed, it is saved to disk for later use when computing the stopping policy.

We found that it was extremely useful to have a simple way to validate our policies while debugging our implementation. One option is to create a stopping policy, run ARA* with monitoring on a handful of instances and with a handful of utility functions to verify that it gives the expected behavior. Unfortunately, this approach was rather cumbersome and prone to error as it only evaluated the policy on the small number of instances that we were willing to run by hand. Instead, we chose to validate our implementation by plotting the polices generated from the training data on different utility functions. By plotting the extreme policies that only care about solution cost and search time, along with some intermediate policies that trade off the two, it was much simpler to debug our code.

Figure D-2 shows some of the plots created for the platform domain. Each plot has cost on the y axis and time on the x axis. Green circles represent inputs for which the policy says to keep searching, and red crosses represent inputs for which the policy says to stop searching and return the solution. As expected, the policy always continues when the goal is to minimize solution cost and always stops when the goal is to minimize search time (cf. the left-most and right-most plots in Figure D-2 respectively). The center plot shows that we also successfully found policies that trade search time for solution cost by only stopping once the cost is sufficiently low. Finally, in the left-most plot, the bottom-most and right-most sides of the policy always stop as our implementation chose to stop when there was no training data available to estimate the profile for the given input values.

```
PROFILE(streams)
 1. qtcounts, qqtcounts ← COUNT-SOLUTIONS(streams)
 2. return PROBABILITIES(qtcounts, qqtcounts)

COUNT-SOLUTIONS(streams)
 3. qtcounts ← new int[ncost][ntime] // Initialized to zero.
 4. qqtcounts ← new int[ncost][ncost][ntime] // Initialized to zero.
 5. for s in streams
 6.   for i from 1 to |s|
 7.     q_i ← costbin(s[i].cost)
 8.     q_cur ← q_i, Δt_cur ← 0
 9.     // Count cost at each time increment after solution i.
10.     for j from i + 1 to |s|
11.       q_next ← costbin(s[j].cost)
12.       Δt_next ← timebin(s[j].time − s[i].time)
13.       // Current solution cost up to the time of solution j.
14.       for Δt = Δt_cur to Δt_next − 1
15.         increment qtcounts[q_i][Δt]
16.         increment qqtcounts[q_cur][q_i][Δt]
17.       q_cur ← q_next, Δt_cur ← Δt_next
18.     // Last solution cost up to the final time.
19.     for Δt = Δt_cur to ntime
20.       increment qtcounts[q_i][Δt]
21.       increment qqtcounts[q_cur][q_i][Δt]
22. return qtcounts, qqtcounts

PROBABILITIES(qtcounts, qqtcounts)
23. probs ← new float[ncost][ncost][ntime]
24. for q_i from 1 to ncost
25.   for Δt from 1 to ntime
26.     if qtcounts[q_i][Δt] = 0 then continue
27.     for q_j from 1 to ncost
28.       probs[q_j][q_i][Δt] < −qqtcounts[q_j][q_i][Δt]/qtcounts[q_i][Δt]
29. Smoothing: add half of smallest probability to all elements of probs with improving solution cost.
30. // Normalize.
31. for q_i from 1 to ncost
32.   for Δt from 1 to ntime
33.     sum ← 0
34.     for q_j from 1 to ncost
35.       sum ← sum + probs[q_j][q_i][Δt]
36.     for q_j from 1 to ncost
37.       probs[q_j][q_i][Δt] ← probs[q_j][q_i][Δt]/sum
38. return probs
```

Figure D-1: Pseudocode for profile estimation.

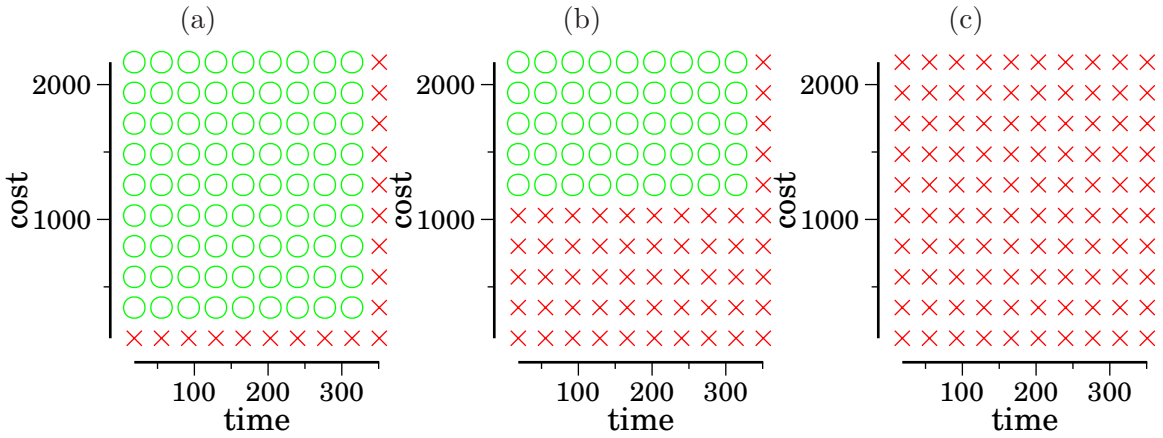


Figure D-2: Three different policies: (a) prefers cheaper solutions at any expense ($w_f = 1, w_t = 0$), (b) attempts to trade some search time for some solution cost ($w_f = 0.6, w_t = 1$), and (c) prefers to have any solution as fast as possible ($w_f = 0, w_t = 1$).

Bibliography

Aine, S., Chakrabarti, P., & Kumar, R. (2010). Heuristic search under contract. *Computational Intelligence*, *26*(4), 386–419.

Arbelaez, A., Hamadi, Y., & Sebag, M. (2010). Continuous search in constraint programming. In *Twenty-second IEEE International Conference on Tools with Artificial Intelligence (ICTAI-10)*, Vol. 1, pp. 53–60.

Battiti, R., Brunato, M., & Mascia, F. (2008). *Reactive search and intelligent optimization*, Vol. 45. Springer.

Benton, J., Do, M., & Ruml, W. (2007). A simple testbed for on-line planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS-07) Workshop on Moving Planning and Scheduling Systems into the Real World*.

Biere, A., Artho, C., & Schuppan, V. (2002). Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, *66*(2), 160–177.

Björnsson, Y., Bulitko, V., & Sturtevant, N. (2009). TBA*: time-bounded A*. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09)*, pp. 431–436.

Boddy, M., & Dean, T. (1989). Solving time-dependent planning problems,. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Vol. 2, pp. 979–984.

Bošnački, D., & Holzmann, G. (2005). Improving spins partial-order reduction for breadth-first search. In Godefroid, P. (Ed.), *Model Checking Software*, Vol. 3639 of *Lecture Notes in Computer Science*, pp. 902–902. Springer Berlin / Heidelberg.

Burns, E., Benton, J., Ruml, W., Do, M., & Yoon, S. (2012a). Anticipatory on-line planning. In *Proceedings of the Twenty-second International Conference on Automated Planning and Scheduling (ICAPS-12)*.

Burns, E., Hatem, M., Leighton, M. J., & Ruml, W. (2012b). Implementing fast heuristic search code. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS-12)*.

Burns, E., Lemons, S., Ruml, W., & Zhou, R. (2009a). Suboptimal and anytime heuristic search on multi-core machines. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS-09)*.

Burns, E., Lemons, S., Zhou, R., & Ruml, W. (2009b). Best-first heuristic search for multi-core machines. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-09)*.

Burns, E., Lemons, S., Ruml, W., & Zhou, R. (2010). Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, *39*, 689–743.

Burns, E., & Ruml, W. (2013). Iterative-deepening search with on-line tree size prediction. *Annals of Mathematics and Artificial Intelligence*, *S68*.

Burns, E., & Zhou, R. (2012). Parallel model checking using abstraction. In *Proceedings of the Nineteenth International SPIN Workshop on Model Checking of Software (SPIN-12)*, pp. 172–190.

Chen, P. C. (1992). Heuristic sampling: a method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, *21*(2), 295–315.

Cox, M. T., & Raja, A. (2011). *Metareasoning: Thinking about thinking*. MIT Press.

Culberson, J. C., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Cushing, W., Bentor, J., & Kambhampati, S. (2010). Cost based search considered harmful. In *The Third International Symposium on Combinatorial Search (SOCS-10)*.

Dai, P., & Hansen, E. A. (2007). Prioritizing bellman backups without a priority queue. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*.

Davis, H. W., Bramanti-Gregor, A., & Wang, J. (1988). The advantages of using depth and breadth components in heuristic search. In *Methodologies for Intelligent Systems 3*, pp. 19–28.

de Pomiane, E. (1930). *French Cooking in Ten Minutes: Adapting to the Rhythm of Modern Life*. North Point Press. Translated by Philip Hyman and Mary Hyman.

Dean, T., & Boddy, M. (1988). An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pp. 49–54.

Dean, T., Kaelbling, L. P., Kirman, J., & Nicholson, A. (1993). Planning with deadlines in stochastic domains. In *Proceedings of the eleventh national conference on Artificial intelligence*, Vol. 574, p. 579. Washington, DC.

Dechter, R., & Pearl, J. (1988). The optimality of A*. In Kanal, L., & Kumar, V. (Eds.), *Search in Artificial Intelligence*, pp. 166–199. Springer-Verlag.

Dionne, A. J., Thayer, J. T., & Ruml, W. (2011). Deadline-aware search using on-line measures of behavior. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search (SoCS-11)*.

Dong, Y., Du, X., Holzmann, G., & Smolka, S. (2003). Fighting livelock in the GNU i-protocol: a case study in explicit-state model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, *4*(4), 505–528.

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

Dweighter, H. (1975). Elementary problem E2569. *American Mathematical Monthly*, *82*(10), 1010.

Edelkamp, S., & Schrödl, S. (2000). Localizing A*. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pp. 885–890. AAAI Press.

Edelkamp, S., & Sulewski, D. (2010). GPU exploration of two-player games with perfect hash functions. In *The Third International Symposium on Combinatorial Search (SOCS-10)*.

Evans, J. (2006). A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of BSDCan 2006*.

Evett, M., Hendler, J., Mahanti, A., & Nau, D. (1995). PRA* - massively-parallel heuristic-search. *Journal of Parallel and Distributed Computing*, *25*(2), 133–143.

Felner, A., Kraus, S., & Korf, R. (2003). KBFS: K-best-first search. *Annals of Mathematics and Artificial Intelligence*, *39*(1-2), 19–39.

Ferguson, C., & Korf, R. E. (1988). Distributed tree search and its applications to alpha-beta pruning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*.

Finkelstein, L., & Markovitch, S. (2001). Optimal schedules for monitoring anytime algorithms. *Artificial Intelligence*, *126*(1), 63–108.

Fox, M., Gerevini, A., Long, D., & Serina, I. (2006). Plan stability: Replanning versus plan repair. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-06)*, pp. 212–221.

Garvey, A. J., & Lesser, V. R. (1993). Design-to-time real-time scheduling. *Systems, Man and Cybernetics, IEEE Transactions on*, *23*(6), 1491–1502.

Gates, W. H., & Papadimitriou, C. H. (1979). Bounds for sorting by prefix reversal. *Discrete Mathematics*, *27*(1), 47–57.

Graham, R. L., Knuth, D. E., & Patashnik, O. (1998). *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley.

Hamadi, Y., Monfroy, E., & Saubion, F. (2010). What is autonomous search?. *Hybrid Optimization*, *45*, 357–391.

Hansen, E. A., & Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research*, *28*(1), 267–297.

Hansen, E. A., & Zilberstein, S. (2001). Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, *126*, 139–157.

Hansen, E. A., Zilberstein, S., & Danilchenko, V. A. (1997). Anytime heuristic search: First results. Tech. rep., University Massachusetts, Amherst.

Harris, T. L. (2001). A pragmatic implementation of non-blocking linked-lists. In *Lecture Notes in Computer Science*, Vol. 2180/2001, pp. 300–314. Springer Berlin / Heidelberg.

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

Haslum, P., Botea, A., Helmert, M., Bonte, B., & Koenig, S. (2007). Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*.

Haslum, P., & Geffner, H. (2000). Admissible heuristics for optimal planning. In *Proceedings of the Fifth Internationas Conference on Artificial Intelligence Planning and Scheduling Systems (AIPS-00)*, pp. 140–149.

Helmert, M. (2010). Landmark heuristics for the pancake problem. In *Proceedings of the Third Symposium on Combinatorial Search (SoCS-10)*.

Hernández, C., Baier, J., Uras, T., & Koenig, S. (2012). Time-bounded adaptive A*. In *Proceedings of the Eleventh International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-12)*.

Holzmann, G. J. (2004). *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley.

Holzmann, G. J., & Bošnački, D. (2007). The design of a multicore extension of the spin model checker. *IEEE Transactions on Software Engineering*, *33*(10), 659–674.

Holzmann, G. J., & Peled, D. (1994). An improvement in formal verification. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques (FORTE-94)*.

Horvitz, E., Ruan, Y., Gomes, C., Kautz, H., Selman, B., & Chickering, M. (2001). A Bayesian approach to tackling hard computational problems. In *Proceetings of the Seventeenth Conference on Uncertainty in Artificial Intelligence (UAI-01)*.

Horvitz, E., & Rutledge, G. (1991). Time-dependent utility and action under uncertainty. In *Proceedings of the Seventh Conference on Uncertainty in Artificial Intelligence*, pp. 151–158. Morgan Kaufmann Publishers Inc.

Jabbar, S., & Edelkamp, S. (2006). Parallel external directed model checking with linear I/O. In Emerson, E., & Namjoshi, K. (Eds.), *Verification, Model Checking, and Abstract Interpretation*, Vol. 3855 of *Lecture Notes in Computer Science*, pp. 237–251. Springer Berlin / Heidelberg.

Kaelbling, L. P. (1993). *Learning in embedded systems*. The MIT Press.

Kiesel, S., Burns, E., Wilt, C., & Ruml, W. (2011). Integrating vehicle routing and motion planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-11)*.

Kilby, P., Slaney, J., Thiébaux, S., & Walsh, T. (2006). Estimating search tree size. In *Proceedings of the twenty-first national conference on artificial intelligence (AAAI-06)*.

Kishimoto, A., Fukunaga, A., & Botea, A. (2009). Scalable, parallel best-first search for optimal sequential planning. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*.

Knuth, D. E. (1975). Estimating the efficiency of backtrack programs. *Mathematics of computation*, *29*(129), 121–136.

Koenig, S., & Sun, X. (2009). Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, *18*(3), 313–341.

Koenig, S., & Likhachev, M. (2002). D* lite. In *Proceedings of the eighteenth national conference on artificial intelligence (AAAI-02)*, pp. 476–483. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Korf, R. (2008). Linear-time disk-based implicit graph search. *Journal of the ACM*, *35*(6).

Korf, R. E. (1985). Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 1034–1036.

Korf, R. E. (1990). Real-time heuristic search. *Artificial intelligence*, *42*(2-3), 189–211.

Korf, R. E. (1993). Linear-space best-first search. *Artificial Intelligence*, *62*(1), 41–78.

Korf, R. E. (2003). Delayed duplicate detection: extended abstract. In *Proceedings of the Eighteenth International Joint Conference on Articial Intelligence (IJCAI-03)*, pp. 1539–1541.

Korf, R. E., Reid, M., & Edelkamp, S. (2001). Time complexity of iterative-deepening-A*. *Artificial Intelligence*, *129*(1), 199–218.

Korf, R. E., & Schultze, P. (2005). Large-scale parallel breadth-first search. In *Proceedings of the Twentieth National Conference on Articial Intelligence (AAAI-05)*, pp. 1380–1385.

Korf, R. E., Zhang, W., Thayer, I., & Hohwald, H. (2005). Frontier search. *Journal of the ACM*, *52*(5), 715–748.

Kuhn, L., Schmidt, T., Price, B., Zhou, R., & Do, M. (2008). Heuristic search for target-value path problem. In *First International Symposium on Search Techniques in Artificial Intelligence and Robotics*.

Kumar, V., Ramesh, K., & Rao, V. N. (1988). Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pp. 122–127.

Lamport, L. (2002). *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

Lelis, L., Stern, R., & Jabbari Arfaee, S. (2011). Predicting solution cost with conditional probabilities. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search (SoCS-11)*.

Likhachev, M., Gordon, G., & Thrun, S. (2003a). ARA*: Anytime a* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems (NIPS-03)*, *16*.

Likhachev, M., Gordon, G., & Thrun, S. (2003b). ARA*: Formal analysis. Tech. rep. CMU-CS-03-148, Carnegie Mellon University School of Computer Science.

Malitsky, Y. (2012). *Instance-Specific Algorithm Configuration*. Ph.D. thesis, Brown University.

Maron, O., & Moore, A. W. (1994). Hoeffding races: Accelerating model selection search for classification and function approximation. In *Proceedings of Advances in Neural Information Processing Systems (NIPS-04)*.

Méro, L. (1984). A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, *23*(1), 13–27.

Nebel, B., & Koehler, J. (1995). Plan reuse versus plan generation: A theoretical and empircal analysis. *Artificial Intelligence*, *76*, 427–454.

Niewiadomski, R., Amaral, J., & Holte, R. (2006a). A parallel external-memory frontier breadth-first traversal algorithm for clusters of workstations. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP-06)*, pp. 531–538.

Niewiadomski, R., Amaral, J. N., & Holte, R. C. (2006b). Sequential and parallel algorithms for frontier A* with delayed duplicate detection. In *Proceedings of the 21st national conference on Artificial intelligence (AAAI-06)*, pp. 1039–1044. AAAI Press.

Nilsson, N. J. (1969). A mobile automaton: an application of artificial intelligence techniques. In *Proceedings of the First International Joint Conference on Artificial intelligence (IJCAI-69)*, pp. 509–520.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Tioga Publishing Co.

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

Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., & Winterbottom, P. (1995). Plan 9 from Bell Labs. *Computing Systems*, *8*(3), 221–254.

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

Powley, C., & Korf, R. E. (1991). Single-agent parallel window search. *IEEE Transactions Pattern Analysis Machine Intelligence*, *13*(5), 466–477.

Reif, J. H. (1985). Depth-first search is inherently sequential. *Information Processing Letters*, *20*(5), 229–234.

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

Richter, S., Thayer, J. T., & Ruml, W. (2010). The joy of forgetting: Faster anytime search via restarting. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10)*, pp. 137–144.

Richter, S., & Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, *39*.

Rose, K., Burns, E., & Ruml, W. (2011). Best-first search for bounded-depth trees. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search (SoCS-11)*.

Ruml, W. (2002). *Adaptive Tree Search*. Ph.D. thesis, Harvard University.

Ruml, W., Do, M., Zhou, R., & Fromherz, M. P. (2011). On-line planning and scheduling: An application to controlling modular printers. *Journal of Artificial Intelligence Research*, *40*(1), 415–468.

Ruml, W., & Do, M. B. (2007). Best-first utility-guided search. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pp. 2378–2384.

Russell, S., & Wefald, E. (1991). *Do the right thing: studies in limited rationality*. The MIT Press.

Sarkar, U., Chakrabarti, P., Ghose, S., & Sarkar, S. D. (1991). Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence*, *50*, 207–221.

Schmidt, T., Kuhn, L., Price, B., de Kleer, J., & Zhou, R. (2009). A depth-first approach to target-value search. In *Proceedings of the Second Symposium on Combinatorial Search (SoCS-09)*.

Schuppan, V., & Biere, A. (2004). Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, *5*(2-3), 185–204.

Simon, H. A. (1982). From substantive to procedural rationality. *Models of bounded rationality: behavioral economics and business organization*, *2*, 424–443.

Snir, M., & Otto, S. (1998). *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA.

Stern, U., & Dill, D. (1997). Parallelizing the murϕ verifier. In *Computer Aided Verification*, pp. 256–267. Springer.

Stern, U., & Dill, D. L. (1998). Using magnetic disk instead of main memory in the mur ϕ verifier. In *Computer Aided Verification*, pp. 172–183. Springer.

Stewart, B. S., & Chelsea C. White, I. (1991). Multiobjective A*. *Journal of the ACM*, *38*(4).

Sturtevant, N. (2012). Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, *4*(2), 144 – 148.

Sundell, H., & Tsigas, P. (2005). Fast and lock-free concurrent priority queues for multi-thread systems. *Parallel and Distributed Processing Symposium, International*, *65*(5), 609–627.

Thayer, J. (2012). *Faster Optimal and Suboptimal Heuristic Search*. Ph.D. thesis, University of New Hampshire.

Thayer, J., Stern, R., Felner, A., & Ruml, W. (2012a). Faster bounded-cost search using inadmissible estimates. In *Proceedings of the Twenty-second International Conference on Automated Planning and Scheduling (ICAPS-12)*.

Thayer, J. T., Benton, J., & Helmert, M. (2012b). Better parameter-free anytime search by minimizing time between solutions. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS-12)*.

Thayer, J. T., Dionne, A., & Ruml, W. (2011). Learning inadmissible heuristics during search. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-11)*.

Thayer, J. T., & Ruml, W. (2008). Faster than weighted A*: An optimistic approach to bounded suboptimal search. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-08)*.

Thayer, J. T., & Ruml, W. (2009). Using distance estimates in heuristic search. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*.

Thayer, J. T., & Ruml, W. (2010). Anytime heuristic search: Frameworks and algorithms. In *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS-10)*.

Thayer, J. T., & Ruml, W. (2011). Bounded suboptimal search: A direct approach using inadmissible estimates. In *IJCAI*, pp. 674–679.

Tolpin, D., & Shimony, S. E. (2011). Rational deployment of CSP heuristics. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI-11)*.

Valenzano, R. A., Sturtevant, N., Schaeffer, J., Buro, K., & Kishimoto, A. (2010). 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 (ICAPS-10)*.

Valois, J. D. (1995). *Lock-Free Data Structures*. Ph.D. thesis, Rensselaer Polytechnic Institute.

van den Berg, J., Shah, R., Huang, A., & Goldberg, K. (2011). ANA*: Anytime nonparametric A*. In *Proceedings of Twenty-fifth AAAI Conference on Artificial Intelligence (AAAI-11)*.

Vempaty, N. R., Kumar, V., & Korf, R. E. (1991). Depth-first vs best-first search. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pp. 434–440.

Wah, B. W., & Shang, Y. (1995). Comparison and evaluation of a class of IDA* algorithms. *International Journal on Artificial Intelligence Tools*, *3*(4), 493–523.

Xu, L., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2008). SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, *32*(1), 565–606.

Yu, Y., Manolios, P., & Lamport, L. (1999). Model checking TLA^+ specifications. In *Correct Hardware Design and Verification Methods*, pp. 54–66. Springer Berlin / Heidlberg.

Zahavi, U., Felner, A., Burch, N., & Holte, R. C. (2010). Predicting the performance of IDA* using conditional distributions. *Journal of Artificial Intelligence Research*, *37*, 41–83.

Zhou, R., & Hansen, E. (2006). Domain-independent structured duplicate detection. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, pp. 1082–1087.

Zhou, R., & Hansen, E. A. (2004). Structured duplicate detection in external-memory graph search. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*.

Zhou, R., & Hansen, E. A. (2006). Breadth-first heuristic search. *Artificial Intelligence*, *170*(4–5), 385–408.

Zhou, R., & Hansen, E. A. (2007). Parallel structured duplicate detection. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*.

Zhou, R., & Hansen, E. A. (2011). Dynamic state-space partitioning in external-memory graph search. In *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS-11)*.

Zhou, R., Schmidt, T., Hansen, E. A., Do, M. B., & Uckun, S. (2010). Edge partitioning in parallel structured duplicate detection. In *The 2010 International Symposium on Combinatorial Search (SOCS-10)*.

Zohavi, U., Felner, A., Burch, N., & Holte, R. (2010). Predicting the performance of IDA* using conditional distributions. *Journal of Artificial Intelligence Research*, *37*(1), 41–84.