Adaptive Tree Search

A thesis presented

by

Wheeler Ruml

to

The Division of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

May, 2002

To my family.

Abstract

Combinatorial optimization and constraint satisfaction problems are ubiquitous in computer science, arising in areas as diverse as resource allocation, automated design, planning, and logical inference. Finding optimal solutions to such problems often entails searching an intractably large tree of possibilities. Problems beyond the reach of exhaustive enumeration require methods that exploit problem-specific heuristic knowledge to find the best solution possible within the time available. Previous algorithms typically follow a fixed search order, making inefficient use of heuristic information. For a new problem, it is not always clear which predetermined search order will be most appropriate.

I propose an adaptive approach in which the search order is adjusted according to heuristic information that becomes available only during the search. By adapting to the current problem, this approach eliminates the need for pilot experiments and enables the use of good search orders that would be too complicated to program explicitly.

To demonstrate the feasibility of the approach, I first present a simple but incomplete technique, *adaptive probing*. Empirical results demonstrate that the method can effectively adapt on-line, surpassing existing methods on several synthetic benchmarks. I then introduce a general framework for complete adaptive tree search, *best-leaf-first search*, and show how previous work can be viewed as special cases of this technique. Incorporating different sources of information into the framework leads to different search algorithms. Five dif-

ferent instantiations are tested empirically on challenging combinatorial optimization and constraint satisfaction benchmarks, in many cases yielding the best results achieved to date. Best-leaf-first search can be understood as an extension of traditional heuristic shortest-path algorithms to combinatorial optimization and constraint satisfaction. By extending heuristic search to these new domains, I unite these previously separate problems.

Contents

List of Figures

List of Tables

Acknowledgments

Stuart Shieber and Joe Marks, with help from Tom Ngo, showed me how research could be serious fun. Thanks guys! You changed my life.

Stuart has been an excellent research advisor, always willing to let me explore and yet always happy to offer insightful suggestions. His clear thinking, good humor, high standards, and intellectual curiosity will forever be a model for me. Barbara Grosz showed me firsthand what it means to care about teaching and was always willing to share her hard-won wisdom. Her dedication to community-building, from the AI Research Group to the international level, has been inspirational. Yu-Chi Ho generously welcomed me to meetings of his research group and kindly agreed to serve on my thesis committee.

The Harvard AI Research Group has been a wonderfully friendly and supportive environment. Avi Pfeffer suffered through many conversations about work that became Chapter 3 and offered key suggestions. I hope to someday be able to explain things as calmly and clearly as he can. I have been blessed with fantastic officemates. Rebecca Hwa was a burst of sunshine. Her high standards for civilized life remain inspirational. Luke Hunsberger was a rock of integrity—talking with him always left me better connected to reality. Mookie Wilson provided welcome comic relief. The burning intellect that is Ken Shan left delightful and refreshing sparks all over the office. I miss Joshua Goodman's incisive, unpretentious analyses. I wish Kobi Gal peace and prosperity. David Parkes, Jill Nickerson, Lillian Lee, Stan Chen, Ellie Baker, Kathy Ryall, Tamara Babaian, Tim Rauenbusch, Dave Sullivan, Emir Kapanci, and Marco Carbone also enriched my time at Harvard, and I learned from each of them. While not official members of AIRG, Rocco Servedio, Joe Marks, Ric Crabbe, Christian Lindig, Norman Ramsey, and Harry Lewis offered friendship and support that was much appreciated.

While his influence is not directly present in this work, Alfonso Caramazza left an indelible mark. It was an honor to work with him. Michele Miozzo's patience was invaluable, as was his friendship. Josh Tenenbaum also humored me good-naturedly when I indulged my weakness for cognitive science.

Other important inspirations who deserve mention here include Julia Child, Harry Olivar, Brian Kernighan, and John A. and Janette H. Wheeler. I would also like to thank Harvard University, my pleasant home for so many years.

This thesis is dedicated to my family, especially Kate, Mom, Dad, Anton, Frances, and Fred. Without you, none of this would have happened. Thanks for encouraging me to do whatever I wanted and for your unwavering support along the way.

This research was supported in part by NSF grants CDA-94-01024 and IRI-9618848 and by DARPA contract F30602-99-C-0167 through a subcontract with SRI International. Portions of Chapter 3 were described in the *Proceedings of IJCAI-01* (Ruml, 2001a), the *Proceedings of the IJCAI-01 Workshop on Stochastic Search* (Ruml, 2001b), and the *Pro-*

ceedings of the 2001 AAAI Fall Symposium on Using Uncertainty Within Computation (Ruml, 2001c).

Chapter 1

Introduction

This thesis proposes a new approach to solving combinatorial search problems. In guises such as mazes, puzzles, and games, such problems have fascinated people for millennia (Herodotus, 440 BC, 148). They present a simplified version of everyday decision-making, capturing the essential fact that the best choice for one decision often depends on the choices that are made for the other decisions. In abstract formulations such as Hamiltonian path problems, bin packing, or propositional satisfiability, such problems have long been at the core of computer science (Euler, 1759; Dantzig, Fulkerson, and Johnson, 1954). We can think of a combinatorial problem as a fixed set of decision variables, each of which must be assigned a value selected from a set of discrete alternatives. For instance, the variables might represent the actions to take at each step in a puzzle and their possible values would represent the choices available at the corresponding step.

Unfortunately, many combinatorial problems are NP-hard and the only known methods for solving them optimally require enumerating all possible assignments to the variables (Garey and Johnson, 1991). One can conceptualize these methods as traversing a tree that compactly represents the possibilities. As shown in Figure 1.1, each internal node in the

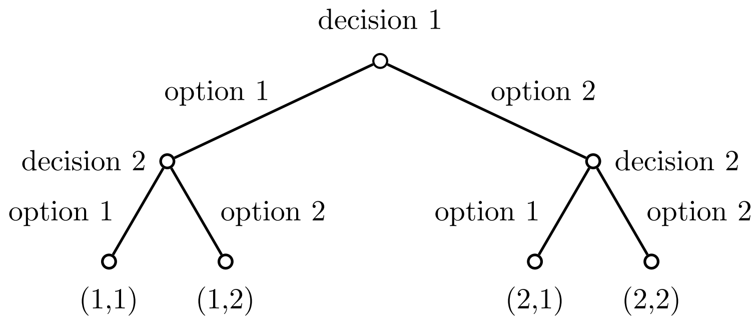

Figure 1.1: A tree representation of alternatives in a small combinatorial problem.

tree represents a decision variable in the problem and each branch represents a possible value for that variable. The leaves of the tree represent candidate solutions. Clearly, the computation time required to exhaustively search such a tree for the optimal solution is exponential in the size of the problem. For large problems or in real-time applications, our only hope is to find the best solution possible within a given time bound. This thesis introduces a new rational framework for approaching this vexing yet ubiquitous problem and demonstrates that algorithms derived within the framework can surpass existing ad hoc methods.

The central idea is to use problem-specific information that arises during the exploration of the tree to adapt the search to the specific problem being solved. A tree search algorithm can be viewed as an agent gathering information as the exploration proceeds, making inferences about where good solutions might lie, and selecting appropriate actions to reach them. Any prior information one might have about the nature of the problem can be easily incorporated into this learning process. One might assume that this rational approach to search would have impractical overhead. However, as we will see, an adaptive search algorithm for combinatorial optimization need not be complex to be efficient and effective. Furthermore, by pursuing the idea of the search algorithm as an agent with ex-

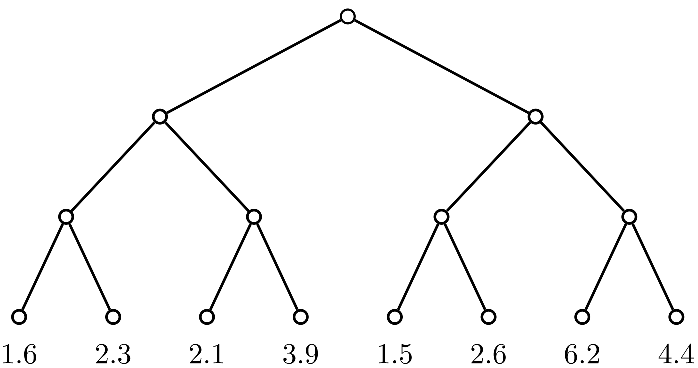

Figure 1.2: A tree representing a combinatorial optimization problem.

plicit knowledge, we will show how combinatorial tree search can be elegantly unified with traditional work on shortest-path problems.

1.1 Types of Search Problems

Combinatorial tree search problems arise in any situation in which the best course of action is not immediately evident and one may need to return to a previously visited state. These states form the internal nodes in the tree. There are four main kinds of combinatorial search problems: combinatorial optimization problems, shortest-path problems, adversarial search problems, and improvement search problems. In this thesis, we will mainly be concerned with combinatorial optimization. A brief review will make these distinctions clear.

1.1.1 Combinatorial Optimization

As we mentioned above, a combinatorial optimization problem consists of a fixed set of variables, each of which must be assigned a value drawn from a set of discrete possibilities. For instance, we might be trying to decide which machine to use for manufacturing each component of a product. An optimization problem also specifies an objective function which assigns a numerical value to every possible set of assignments. In our example, each complete manufacturing configuration might have an associated production cost. Solving the problem means finding the minimum cost solution. We will view this as the task of finding the best leaf in the tree of possibilities, as illustrated in Figure 1.2. The tree's depth is bounded by the number of problem variables and its branching factor is bounded by the maximum number of possible values for any variable. This general problem formulation covers an enormous number of problems, including the traveling salesman problem, automated design problems, scheduling problems, combinatorial auctions, and many problems in machine learning.

Much work in operations research considers how to construct pruning rules and cutting planes that use information about the current problem instance to quickly eliminate many candidate solutions from consideration and thus reduce the size of the search tree. For instance, assigning a certain machine to manufacture a certain component might turn out to be so costly that it is immediately clear that no solution that includes this assignment could be better than the best solution found so far. Such techniques can be used as a pre-processing step as well as during the course of the search. Of course, even after such pruning, one is usually still left with a tree to search, so such techniques are orthogonal to our concerns here. For large problems, we will always be reduced to finding the best solution we can in a bounded-depth tree that is too large to enumerate.

For some applications, such as automatically laying out graphics in an interactive interface, time constraints will apply even for a relatively small problem. Similarly, the increasing diversity in computing platforms motivates consideration of *anytime algorithms*, which can provide mediocre solutions quickly and then improve them if given additional computation time. Such software can perform well across multiple platforms, flexibly adapting its performance to the resources at hand. With the current interest in intelligent embedded systems, sophisticated real-time search and optimization algorithms will increasingly find application on computationally limited devices.

Constraint satisfaction problems are very similar to combinatorial optimization problems. One is given a fixed number of variables and each of them can take one of a finite number of values. However, instead of finding the assignment that minimizes the value of an objective function, one merely attempts to find a solution that does not violate any of the given constraints. Matching up sports teams for a season's worth of games is a problem of this sort. The possibilities are discrete and various constraints must be satisfied, such as an equal number of home games and away games, no more than two away games or two home games in a row, and the inclusion of traditional rivalry matches. Many other problems, such as configuration, graph coloring, and propositional inference can also be viewed as constraint satisfaction problems. Sophisticated constraint processing techniques can often be used to help reduce the size of the corresponding search tree, setting certain variables automatically and skipping decisions known to be irrelevant. For this reason, some authors treat constraint satisfaction separately from combinatorial optimization.

Since we are concerned with tree search, rather than with particular problem-specific kinds of constraint processing, we will just treat constraint satisfaction problems as an additional kind of combinatorial optimization problem. The objective function will be a measure of the degree of constraint violation, such as the number of variables that remained to be assigned when the first constraint violation was detected. Any leaf that represents a successful assignment to all the variables will have a lower score than any leaf representing a partial assignment that violates a constraint, so finding the best leaf will return a satisfying solution if one exists. This generic point of view is completely compatible with the use of constraint processing techniques.

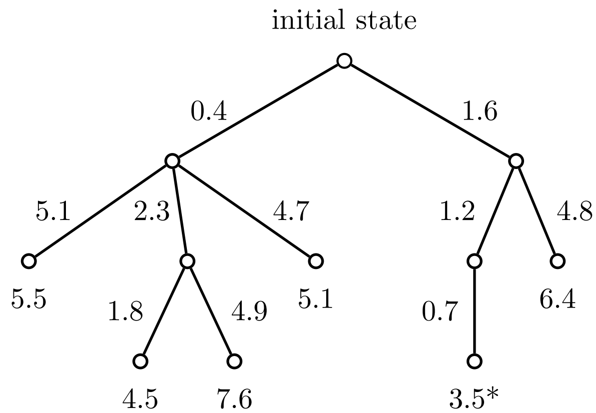

Figure 1.3: A tree representing a shortest-path problem.

1.1.2 Shortest-path Problems

In a shortest-path problem, one attempts to find the cheapest path from a given initial node to any goal node that meets certain criteria. This problem has been the focus of much work in artificial intelligence. It arises in planning, for instance, in which one might want to find the cheapest set of actions that incrementally transform an initial state into a goal state exhibiting some desired properties. Figure 1.3 shows a search tree resulting from a small shortest-path problem. The branches represent the possible actions from each state and they are labeled with their costs. Leaves represent states consistent with the goal conditions. The optimal goal node is labeled with an asterisk (*). As in the problem of combinatorial optimization mentioned above, there are discrete choices at each step. A solution consists of a sequence of choices and can be given a value. Solving the problem means finding the minimum cost solution. But a combinatorial optimization problem involves a fixed and known number of variables, whereas the number of choices that will need to be made in a shortest-path problem is not clear in advance. Because of this, it is difficult to bound the depth of a tree search for a shortest-path problem.

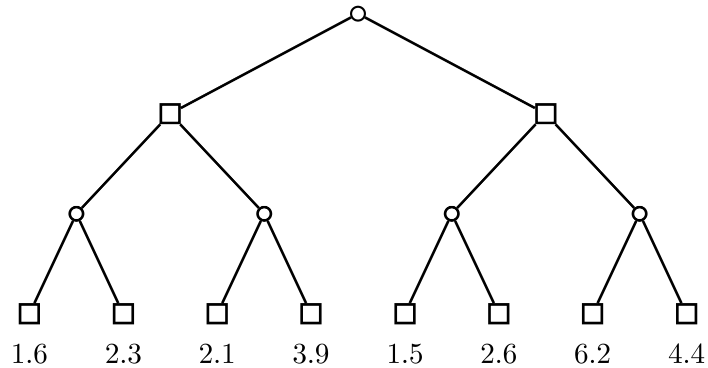

Figure 1.4: A tree representing a game against an adversary.

There are many existing algorithms for shortest-path problems. Some of them require that the entire tree or graph be explicitly represented in the computer at one time. But in many problems, the tree is much too large to represent explicitly and child nodes must be generated dynamically from their parent when needed. The process of generating the children is also known as expanding the parent. We will be assuming henceforth that dynamic tree expansion is necessary and we will be evaluating search algorithms based on the number of nodes they generate. Although this thesis focuses on combinatorial optimization, we will briefly return to shortest-path problems in Section 4.4, where we will see how the two different problems can be approached using a common algorithmic framework.

1.1.3 Adversarial Search

Combinatorial optimization also seems superficially similar to game playing. Games can be modeled as optimization problems in which we must select a sequence of actions so as to minimize (or maximize) a scoring function. In games, however, there is an adversary who can influence the score we can achieve. A game with alternating turns, such as chess,

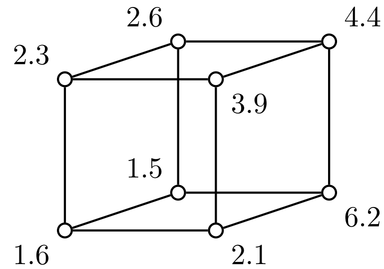

Figure 1.5: A graph representing an improvement-based search.

might be modeled as in Figure 1.4, in which every other level of the tree corresponds to the decisions made by the adversary. In large games like chess, a depth cut-off is usually applied and the scores at the leaves represent an estimate of the strength of the corresponding board position, or the probability of winning from that position, rather than an actual win or loss score from a terminal state of the game.

But merely finding the best leaf in a game search tree is not sufficient, as it was in combinatorial optimization. Even if we take the first action along the path toward the best leaf, our adversary might easily respond with an action that prevents us from reaching that leaf. Instead, one usually seeks to find the top-level decision whose worst case leaf outcome is best, assuming that the adversary is trying to reach the leaves that are poorest for us. This type of multiple-agent strategic search is sufficiently different from the single-agent case that we will not consider it further in this thesis.

1.1.4 Improvement Search

Searching a tree of possibilities is not the only way to attack a combinatorial optimization problem. Another popular method is to construct an initial solution (perhaps even randomly) and then attempt to improve it incrementally by making modifications. Because the changes are typically small, this approach is also called 'local search.' An improvement-based approach leads to a very different type of search problem. In a tree-based approach, many nodes represent partial solutions in which some variables have not yet been assigned values. In an improvement-based approach, one moves from one complete solution to the next, trying to find one with a good score. This can be visualized as traversing a graph in which possible modifications lead to neighboring nodes, each of which is annotated with its cost. Figure 1.5 shows the graph corresponding to the search tree in Figure 1.2. As the figure suggests, the search graph often has the structure of an n-dimensional hypercube, where n is the number of variables in the problem.

One disadvantage of improvement search methods is that they are *incomplete*: there is no guarantee that they will find the optimal solution within a bounded amount of time. In a tree search for combinatorial optimization, one will eventually traverse the entire tree. But when moving through a graph, the high connectivity of the search space makes it difficult to keep a concise record of the states that have and have not been visited. This means that there is no guarantee that every part of it will eventually be explored.

In practice, however, most problems are so large that complete enumeration is infeasible, so the incompleteness of improvement search is not a handicap. Improvement methods are usually simple to implement and they can work very well. In fact, it is not uncommon for researchers to conflate the ideas of improvement search and incomplete search. Of course, the superior performance displayed by improvement search over tree search on many problems may be merely the result of having poor algorithms for incomplete tree search. Many of the results in this thesis will show that the most widely used tree search method, depth-first search, is in fact a terrible choice in many applications.

Further understanding the relative merits of improvement search and tree search is an important direction for future research. The trade-off between the flexible movement afforded by improvement search and the ease with which tree search can incorporate problem-specific information is complex. For the remainder of this thesis, however, we will focus our attention on how best to explore tree-structured spaces. By developing principled methods for tree search under time constraints, this thesis will provide a sound basis for future comparisons with other approaches to search.

1.2 An Adaptive Approach

This thesis approaches the problem of finding good leaves using decision-making concepts from artificial intelligence. The central idea is to adapt during the search process to the particular tree that is being searched. When faced with a new type of optimization problem, or even a new instance from a known type, the optimal search order is not clear *a priori*. So we will view the search algorithm as a rational agent making decisions about which leaf to visit next. New information, such as the costs of the solutions at the visited leaves, becomes available as the algorithm explores the tree. This information is valuable because it can reflect important properties of the current search tree. A rational algorithm would combine this new information with any current beliefs it might have to help infer where to explore next.

Traditional search algorithms for combinatorial optimization, such as depth-first search (DFS), visit leaves in a fixed order, making no use of any information gathered about the nodes visited. Such thoughtless repetition of preprogrammed behavior is an embarrassing caricature of automated problem solving. Although search algorithms are traditionally viewed as lying at the core of an intelligent agent, there is no reason why they should not also be viewed as intelligent agents in their own right. In fact, as we will see, this can lead to both improved algorithms and conceptual advances.

Having an adaptive algorithm considerably lightens the burden for the user, who currently must carry out extensive preliminary experiments to determine which prespecified search order works best for each new problem. Furthermore, because the next action of an adaptive algorithm can be predicated on the complete set of observations to date in a particular tree, an adaptive algorithm can effectively implement a search order that would be very difficult to prespecify.

1.3 Outline

The ability of a search algorithm to adapt intelligently to a particular tree depends on the information that is available during the search. The branching structure of the tree itself typically gives little information—many search trees are uniformly binary branching with a fixed maximum depth. At the very least, however, solution costs are available at leaf nodes. This information is usually computed anyway, to enable the search to remember the best solution seen. After a brief review in Chapter 2 of previous algorithms for tree search, Chapter 3 presents *adaptive probing*, a method for exploiting this leaf cost information during search. We will see that it is possible, using relatively weak assumptions, to efficiently infer the location of good leaves from the positions and costs of the leaves that have already been observed. This is done by learning a model that predicts the costs of leaves based on their location. In a manner reminiscent of reinforcement learning, the model is both learned and exploited during the search. Empirical tests of the algorithm's behavior on both combinatorial optimization and constraint satisfaction problems demonstrate that an adaptive approach can lead to good performance and very robust behavior.

While adaptive probing flexibly adapts to each search tree it encounters, it is not guaranteed to visit every leaf within bounded time. In Chapter 4, we will overcome this incompleteness by proposing a general framework for adaptive tree search called *best-leaf-first search* (BLFS). BLFS follows adaptive probing in using a predictive model of leaf costs to guide search. However, BLFS ensures completeness by using systematic search and an expanding search horizon. BLFS is rational, in the sense that it attempts to maximize its performance based on its current information. As we will discuss, BLFS can be seen as a model-based extension of the iterative deepening A* (IDA*) shortest-path algorithm (Korf, 1985). Their common framework of single-agent rationality provides a clean unification of search for combinatorial optimization and constraint satisfaction with the tradition of heuristic search in AI for shortest-path problems.

BLFS is a general framework that can be specialized according to the particular information that is available in a given application. Often, domain-specific knowledge is available about the likely suitability of each choice at a decision node. These heuristic preferences are usually computed as a numeric score for each child. In Chapter 5, we consider a particular instantiation of BLFS, called *indecision search*, that exploits this information to guide its search. The idea is to backtrack first to those nodes where the children had very similar scores. Intuitively, these are the nodes at which the scoring function was least sure which child was better. The BLFS framework allows this process to be done efficiently. Results on constraint satisfaction problems show indecision search to be superior to existing methods on most classes of difficult problems.

The child scores used in indecision search only become available during the search itself. While these scores are used as input to the leaf cost model to guide the search, the leaf cost model itself remains fixed. In Chapter 6, we investigate an instantiation of the BLFS framework that learns the parameters of the cost model online. This algorithm is essentially a complete and deterministic analogue of the adaptive probing method of Chapter 3. Empirical tests on the combinatorial optimization problem of number partitioning demonstrate that the method yields the best performance known. On a different, less effective formulation of the problem, BLFS is competitive with the best method known.

These results show that an adaptive approach to tree search can be practical, general, and effective. Adaptive algorithms are extremely robust, yielding performance that is either competitive or superior to the best existing techniques in each domain tested. This advance has far-reaching consequences, as combinatorial optimization and constraint solving are ubiquitous problems. In addition, by improving the performance and robustness of optimization in tree-structured search spaces, this approach opens avenues toward quantifying the value of various kinds of heuristic knowledge and comparing the relative advantages of tree-based versus improvement-based problem formulations.

Chapter 2

Tree Search Under Time Constraints

> We briefly review previous work on tree search for combinatorial optimization. Our observations will inform the proposals of subsequent chapters.

We are approaching the problem of combinatorial optimization as a search for the lowest-cost leaf in a tree of bounded depth. Many algorithms have been proposed for this setting and we will briefly review some of them. The algorithms we will consider later can be viewed as generalizing these methods in various ways.

2.1 Greedy Construction

Probably the most common way of dealing with a combinatorial problem is not to search the tree of possibilities at all, but merely to try to construct as good a solution as possible on the first try without backtracking. This is called the *greedy construction* technique because, at every decision, the algorithm takes what seems at that time to be the best choice. These decisions will never be reconsidered, even if the resulting solution is suboptimal. Pseudo-

```
Greedy (node)
If is-leaf(node)
    Return node
else
    Greedy(best-child(node))
```

Figure 2.1: Pseudo-code for greedy construction.

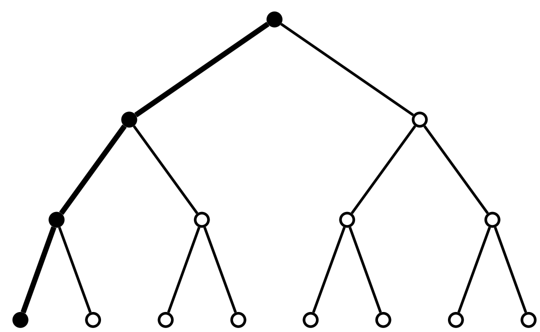

Figure 2.2: The path explored by a greedy construction algorithm.

code for this method is shown in Figure 2.1. A greedy algorithm can be viewed as visiting one leaf in the tree of all possibilities, as shown in Figure 2.2. Note that the tree is drawn such that the left child of each internal node is the one that is seen as more desirable by the greedy algorithm. The leftmost leaf corresponds to the greedy solution. We will continue this convention throughout this thesis.

The greedy algorithm chooses between the available alternatives at each decision point on the basis of some heuristic information. This might be an estimate of the quality or number of solutions available under each child node. For constraint satisfaction problems, where the objective is to find an assignment that violates no constraints, one might use an estimate of the probability that a solution lies below each child. For some problems, a lower bound on solution quality can be calculated and used to order the choices. Often, quantitative heuristic values can be derived as solutions to relaxed versions of the original

```
DFS (node)
1  If is-leaf(node)
2      Visit(node)
3  else
4      For i from 0 to num-children
5          DFS(child(node, i))
```

Figure 2.3: Pseudo-code for depth-first search (DFS).

problem. We will refer to this general kind of child preference information as 'child ordering' or 'node ordering' information.

Of course, the greedy solution is rarely optimal, but much theoretical research has been done to determine exactly how poor it will tend to be for certain problems. For some problems, bounds can be proved on how far from optimal the greedy solution will be. Constructive algorithms with such bounds are known as approximation algorithms. Such algorithms can provide a good starting point for optimization. But if one has extra time available, one might wish to visit other leaves in the hopes of obtaining a better solution.

2.2 Depth-First Search

Every rational algorithm will first explore the path generated by expanding the most preferred child at every decision node.[1] The challenge comes in deciding what to do next. Current algorithms make various assumptions about where to go against the heuristic preference. The most popular backtracking algorithm for exploring a bounded-depth tree is a simple depth-first search (DFS). Pseudo-code for DFS is presented in Figure 2.3. Visiting a leaf, as in step 2 of the pseudo-code, involves computing its cost, checking whether it is

[1] As we will discuss briefly in Section 4.3, active learning may be preferable if both the deadline and the uncertainty of the algorithm's beliefs are known.

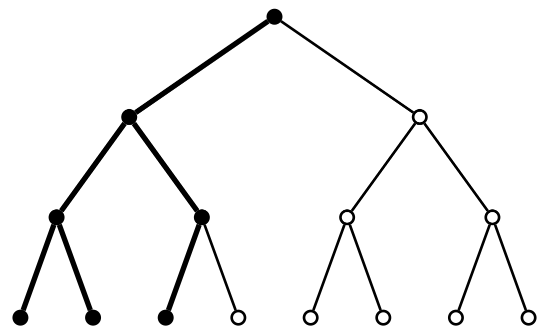

Figure 2.4: The paths explored by DFS after visiting three leaves.

the best leaf seen so far, and exiting if it is recognizably optimal. The backtracking order of depth-first search will visit the second-ranked child of the last internal branching node before reconsidering the choice at the next to last branching node.

If one insists on finding the best leaf in the tree and this leaf cannot necessarily be recognized when it is encountered, then the entire tree must be enumerated. DFS is optimal in this case, because it visits every leaf and generates each internal node only once. The number of internal nodes expanded per leaf visited is at most 1 and even less in non-binary trees. Of course, for most trees, complete enumeration will be out of the question.

DFS can easily take advantage of the same kind of child ordering information as is used by the greedy algorithm. By expanding the children of a node in rank order, DFS will visit first the subtree thought to contain better solutions. As illustrated in Figure 2.4, DFS completely enumerates the subtree below every left child before expanding the right sibling. The convention that the children of a node are ranked left to right in tree diagrams and in decreasing order of desirability according to the heuristic function in pseudo-code will be continued in the remainder of this thesis.

Child ordering information is a form of weak heuristic knowledge that is not necessarily correct. Sometimes, information is also available in the form of provably correct knowledge, such as a lower bound on the cost of any solutions in a particular subtree. In the traveling salesman problem, for instance, the cost of the tour so far plus the cost of the minimum spanning tree of the remaining unvisited cities gives a provably optimistic estimate. In a constraint satisfaction problem, the constraints themselves may be represented in a form that allows early detection of subtrees that cannot contain solutions. Because these stronger forms of knowledge are known to be accurate, they can be used to prune the search tree, eliminating poor regions and reducing its size. This technique is widely used in branch-and-bound algorithms for combinatorial optimization and constraint satisfaction. Of course, even after these strong forms of knowledge have been exploited, one is usually left with a tree to search. Because the weaker forms of knowledge are possibly inaccurate, they cannot be used to prune the tree and hence they can only be exploited to order search in the parts of the tree that are left. In this sense, they are orthogonal to the pruning information. Throughout this thesis, we will assume that any available pruning methods have been applied and focus exclusively on exploiting heuristic knowledge.

DFS enumerates leaves very quickly. But when there is not enough time to enumerate the entire tree, or when the best leaf can be immediately recognized when it is encountered, then it may be advantageous to visit the leaves in a different order. Following Harvey and Ginsberg (1995), we will call each decision at which a non-preferred child is chosen a *discrepancy*. Depth-first search will visit the leaf whose path from the root has all discrepancies below depth i before visiting the leaf with a single discrepancy at depth i. Its search order implicitly represents the assumption that the cost of every leaf whose path includes a discrepancy at the root is greater than the cost of the worst leaf that does not. Equivalently, the penalty for taking a discrepancy at a given depth is assumed to be greater than the cost of taking discrepancies at all deeper depths. This strong assumption is not necessarily

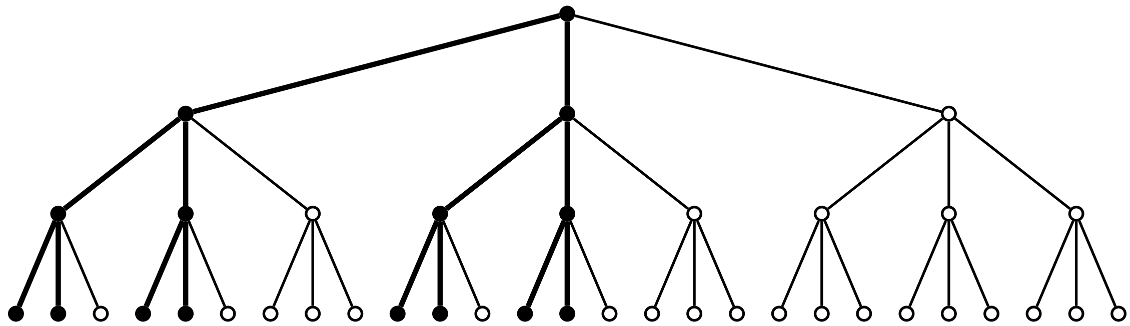

Figure 2.5: The second pass of iterative broadening restricts search to a binary subtree.

correct in a given tree. This will be vividly illustrated in Section 5.5.1 (page 98), where we will see that DFS can exhibit very brittle behavior, performing well for some problem instances and very poorly on others, even when the instances are from the same problem domain. We will now turn our attention to some alternative search methods that rely on different assumptions.

2.3 Iterative Broadening

When the search tree has a high branching factor, the behavior of DFS can seem too stubborn. By trying every child at a given node before backtracking to a previous decision, DFS can become trapped in the lower left portion of the tree. Figure 2.4 illustrates this phenomenon on a small scale. For a large problem, the decision nodes higher in the tree will probably never be revisited by DFS.

Iterative broadening (IB), proposed by Ginsberg and Harvey (1992), attempts to modulate the behavior of DFS. IB works by running a sequence of restricted depth-first searches, each of which considers a larger subset of the tree than the previous. The first iteration treats each node as having only one child: the heuristically preferred one. The second it-

```
IB (node, bound)
If is-leaf(node)
    Visit(node)
else
    For i from 0 to bound
        IB(child(node, i), bound)
```

Figure 2.6: Each iteration of iterative broadening (IB) restricts the effective branching factor of the tree.

eration treats each node as having two children, which will be the two top-ranked children from the original tree. Figure 2.5 gives an example in a ternary tree. In general, iteration k in a tree of depth d visits k^d leaves. Pseudo-code is given in Figure 2.6.

The search order of IB corresponds to the rather loose assumption that the cost of a leaf is proportional to the maximum rank of any child in its path to the root. For a relatively shallow and bushy tree, this may be appropriate. Unfortunately, many trees are dozens or hundreds of levels deep and so even the binary tree explored during the second iteration of IB is too large to enumerate completely. Many problems give rise to trees that are uniformly binary. IB reduces to DFS in such situations. Finer-grained control of the search order is needed. Perhaps because of this flaw, I am not aware of any systems that actually use IB in practice.

2.4 Limited Discrepancy Search

Because DFS and IB will probably never revisit the first decisions they make in large trees, their search orders are implicitly assuming that those decisions were correct. Limited discrepancy search (LDS), introduced by Harvey and Ginsberg (1995), was designed with a different assumption in mind. It assumes that the child ordering function is equally

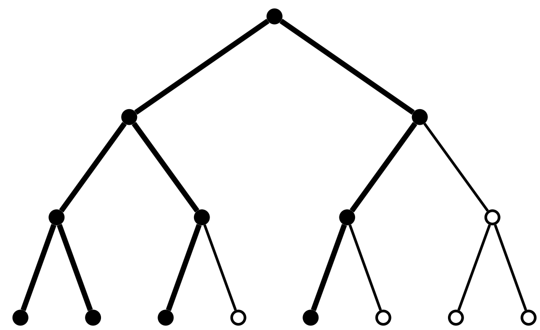

Figure 2.7: The second pass of limited discrepancy search (LDS) visits all leaves with zero or one discrepancies in their path from the root.

```
LDS (node, allowance)
If is-leaf(node)
    Visit(node)
else
    If allowance > 0
        LDS(child(node, 1), allowance − 1)
    LDS(child(node, 0), allowance)
```

Figure 2.8: Each pass in a limited discrepancy search (LDS) visits all leaves whose path from the root contains *allowance* discrepancies. This pseudo-code shows the variant that explores discrepancies at the top of the tree first.

```
ILDS (node, allowance, remaining)
If is-leaf(node)
    Visit(node)
else
    If allowance > 0
        ILDS(child(node, 1), allowance − 1, remaining − 1)
    If remaining > allowance
        ILDS(child(node, 0), allowance, remaining − 1)
```

Figure 2.9: Improved limited discrepancy search (ILDS) only enters subtrees that contain paths with the desired number of discrepancies. This variant explores discrepancies at the top of the tree first.

likely to make mistakes at every level and thus that discrepancies at any depth are equally disadvantageous. LDS visits all leaves with k discrepancies anywhere in their paths before visiting any leaf with $k+1$ discrepancies. The algorithm proceeds in passes, with pass k exploring paths with k or fewer discrepancies and therefore visiting $O(d^k)$ leaves for a tree of depth d. Figure 2.7 shows the leaves visited when k equals one. Note that a leaf in the right subtree from the root is visited on this early pass, but would probably never be visited when running DFS under time constraints. Pseudo-code for a single iteration of LDS is shown in Figure 2.8.

Noting that each iteration of LDS generates a strict superset of the tree explored during the previous iteration, Korf (1996) proposed a modification in which pass k of the algorithm attempts to traverse only those paths containing exactly k discrepancies. This can be done if one knows the maximum depth of the tree, by tracking the remaining depth and making sure that enough decisions are always left to use up the desired number of discrepancies. Pseudo-code for the resulting algorithm, which is called improved limited discrepancy search (ILDS), is shown in Figure 2.9.

ILDS visits $\binom{n}{k}$ leaves on pass k, which leads to a large improvement over LDS when

$O(n)$ passes are performed. When many passes are performed, later iterations of plain LDS will generate more previously-seen nodes than new ones. However, in most applications, n is very large and only the first few passes are ever performed. In the first few passes, both LDS and ILDS must generate $O(n)$ internal nodes to reach each leaf, because only the greedy path above the discrepancy point can be shared across leaves. Thus they have higher overhead per leaf than DFS. As we will see later, experimental results show that the improved search order of ILDS can often overcome this overhead (Korf, 1996; Meseguer and Walsh, 1998), although this depends on the accuracy of the heuristic and the density of recognizably optimal solutions.

Unfortunately, it is not clear how to employ ILDS on a non-binary tree. Some researchers have suggested that all non-preferred children should count as one discrepancy (Korf, 1996; Meseguer and Walsh, 1998), although there is little evidence to suggest that this is preferable to considering less-preferred children to count more. It is also not clear whether to take discrepancies at the top or bottom of the tree first. Such decisions must be made on an ad hoc basis by running pilot experiments.

2.5 Depth-bounded Discrepancy Search

Of course, the basic assumption that discrepancies at each level are equally disadvantageous is itself merely plausible and not necessarily correct. Depth-bounded discrepancy search (DDS), introduced by Walsh (1997), uses a still different assumption: a single discrepancy at depth i is worse than taking discrepancies at all depths shallower than i. Motivated by the idea that node-ordering heuristics are typically more accurate in the later stages of problem-solving, when local information better reflects the remaining subproblem, this

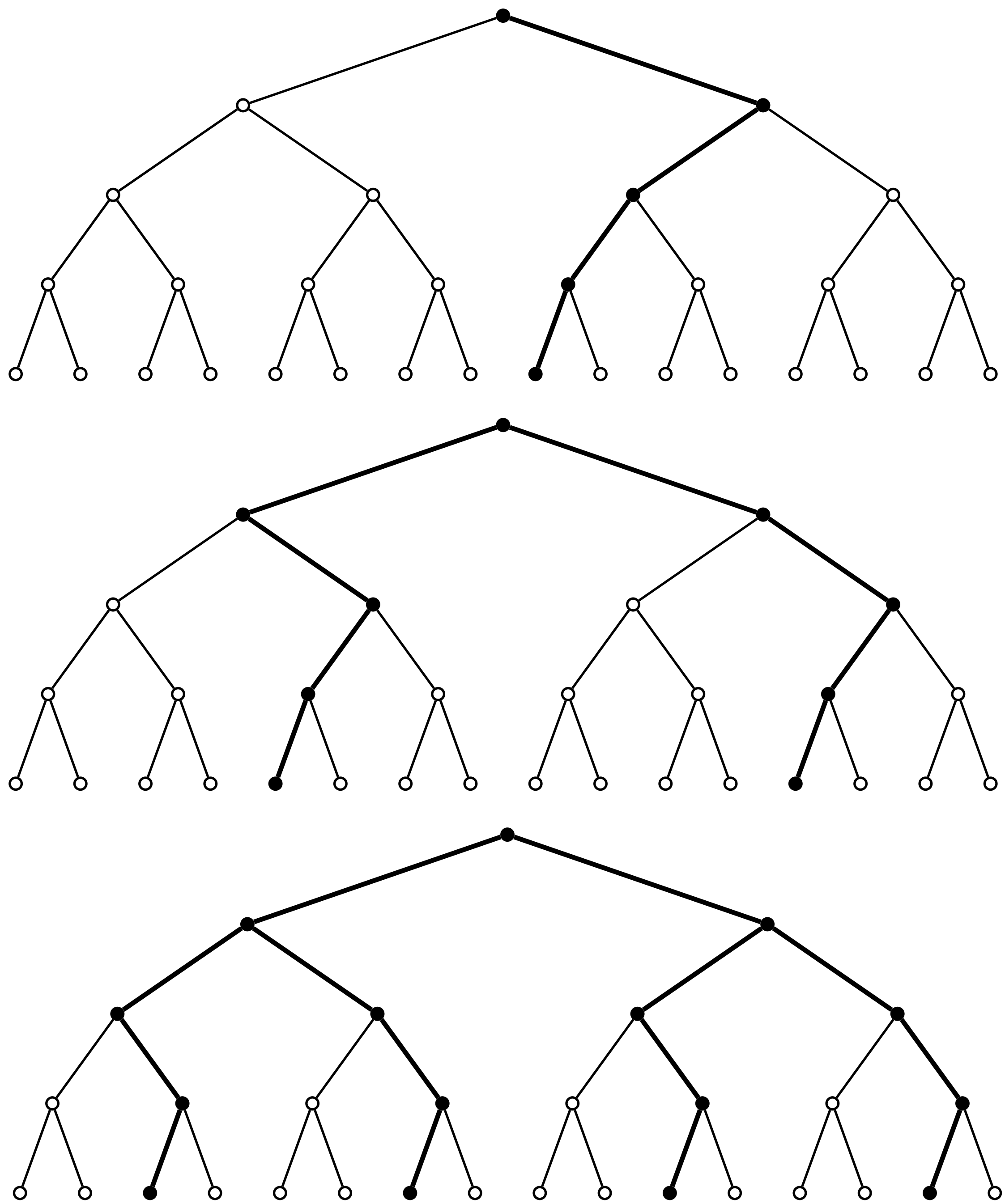

Figure 2.10: The second through fourth iterations of depth-bounded discrepancy search (DDS). The depth bound is 0 for the second iteration (top), then 1 (middle), and then 2 (bottom).

```
DDS (node, remaining)
If is-leaf(node)
    Visit(node)
else
    If remaining < 1
        DDS(child(node, 0), remaining)
    else if remaining = 1
        DDS(child(node, 1), remaining − 1)
    else
        DDS(child(node, 0), remaining − 1)
        DDS(child(node, 1), remaining − 1)
```

Figure 2.11: Depth-bounded discrepancy search (DDS) is greedy below the depth bound.

assumption is directly opposed to the one embodied by depth-first search. The algorithm proceeds in passes, iteratively lowering a depth bound. Above the bound, it behaves like DFS, exploring all possible paths. Below the bound, it always selects the preferred child. In this way, discrepancies are always taken high in the tree.

Note, however, that each pass of DDS includes all the leaves that were visited on the previous pass. In a clever improvement, Walsh prevents duplication by having the algorithm take only non-preferred children at the level of the current depth bound. Because we can be certain that the previous pass, with its depth bound higher in the tree, selected only the preferred child at that level, this ensures that each pass only visits leaves that have not been visited before. Figure 2.10 shows the leaves visited by DDS when the depth bound is zero, one, and two. Pseudo-code for the algorithm is given in Figure 2.11.

DDS can be used with a non-binary tree. In a tree with branching factor b, DDS visits b^k leaves on iteration k. Like ILDS, DDS suffers from $O(n)$ overhead per leaf in the early iterations, although the overhead for DDS is slightly worse because no significant sharing of internal nodes takes place along the paths to one visited leaf and the next. As we will see, experimental results tend to indicate that DDS expands leaves in an excellent order but that this often does not compensate for its overhead (Meseguer and Walsh, 1998). Both ILDS and DDS have recently been added to the commercial ILOG Solver product, and have enabled the solution of previously unsolved scheduling problems (Toby Walsh, personal communication).

2.6 Tunable Techniques

In addition to the methods we have discussed so far, there are other, more complicated tree search techniques that rely on adjustable parameters. Most of these are incomplete stochastic methods, which we will discuss later in Section 3.7. Meseguer (1997), however, has proposed a systematic search technique called interleaved depth-first search (IDFS). The algorithm simulates running multiple DFS processes, each exploring a different subtree. While this algorithm has been shown to be effective on certain CSP problems, it depends on parameters which are not obvious to set, such as the number of simulated processes and the depth at which the tree is divided between them. Chu and Wah (1992) have proposed a similar technique which they call band search. At any given setting of these algorithms' parameters, each of them follows a deterministic ordering.

2.7 Conclusions

Every tree search technique we have discussed either visits leaves in a predetermined order, making strong fixed assumptions about the relative costs of leaves in the tree. Some even require manual tuning of parameters, essentially asking the user to select the most appropriate assumptions from a predefined family. None of the techniques adapts its actions based on its experience in the tree. What we would like is a search method whose cost predictions are based on the current search tree. In the next chapter, we will examine a technique for guiding search based on cost predictions that are learned during the search itself.

Chapter 3

Learning How to Search: Adaptive Probing

We show that a learning approach to tree search can be efficient and effective. The *adaptive probing* algorithm is introduced, which uses leaf costs to infer the costs of choosing the various children at each level of the tree. Empirical results on both combinatorial optimization and constraint satisfaction problems demonstrate that an adaptive approach can lead to good performance and very robust behavior.

We saw in the previous chapter that existing algorithms can be viewed as making strong assumptions about the locations of good leaves. Limited discrepancy search (LDS), for instance, assumes that all discrepancies are equally disadvantageous and that any two discrepancies are worse than any single discrepancy. In this chapter, we will investigate a simple algorithm that incorporates explicit learning in order to avoid strong *a priori* assumptions. In this adaptive approach to tree search, we will use the costs of the leaves we have seen to estimate the actual mean cost of a discrepancy at each level. Simultaneously, we will use these estimates to guide search in the tree. Because the cost predictions are

```
AdaptiveProbing (root)
Initialize model
Loop until time runs out:
    Probe(root)

Probe(node)
If is-leaf(node)
    Visit(node)
    Update model based on node
else
    Choose a child of node using current model
    Probe(child)
```

Figure 3.1: Pseudo-code for adaptive probing.

made dynamically at run-time, they can reflect observed properties of the current search tree. We would thus expect them to lead to better performance than assumptions that are fixed *a priori*. We will see that these expectations are in fact fulfilled. However, the simple algorithm we will consider here has the disadvantage of being incomplete. Later chapters will consider a more elaborate scheme which retains completeness.

3.1 The Algorithm

We will use a very simple framework for the algorithm. To visit a leaf, we will always begin at the root of the tree and select children until we reach a leaf. This lets us avoid having to decide when to return to the root. The choice of child at each node will be guided by the model we have learned so far. Initially, the model has no information about the relative merits of particular children. Starting with no preconceptions, we randomly probe from the root to a leaf. Observing the cost of the solution at that leaf will let us update our model of the actions we took to reach that leaf. By sharpening our action cost estimates based on the leaf costs we observe and choosing children with the probability that they lead to

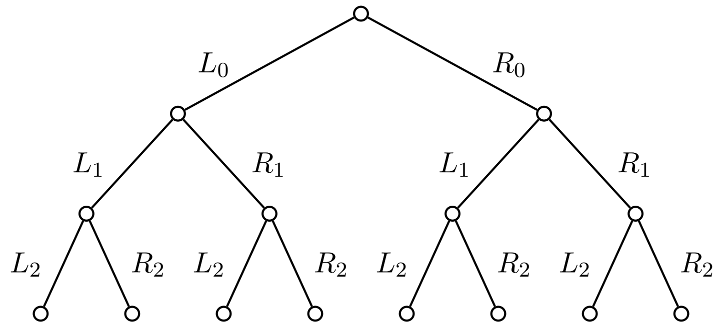

Figure 3.2: The parameters of a separate cost action model for a binary tree of depth three.

solutions with lower cost, we will focus the probing on areas of the tree that seem to contain good leaves. Pseudo-code is given in Figure 3.1.

This stochastic probing approach is incomplete because the method cannot easily keep track of which leaves remain to be visited. Thus, adaptive probing cannot be used to prove the absence of a goal leaf. In addition, it generates the full path from the root to every leaf it visits, incurring overhead proportional to the depth of the tree when compared to depth-first search, which generates roughly one internal node per leaf. However, the problem-specific search order of adaptive probing has the potential to lead to better leaves much faster. Since an inappropriate search order can trap a systematic algorithm into exploring vast numbers of poor leaves, adaptive probing would be useful even if it only avoided such pathological performance on a significant fraction of problems.

3.2 An Additive Cost Model

There are many different ways to instantiate the general algorithm outlined above. We will begin by considering a very simple (yet plausible) model of the search tree. Recall that the model must relate the choice taken at each decision node to the observed cost of the leaf that is eventually reached. We will do this by predicting the cost of every leaf as simply the sum of certain costs, one cost for each choice made along the path from the root. Each cost will depend only on the level at which the choice was made and the rank of the child that was chosen. The model assumes, for instance, that the effect of choosing the second-most-preferred child at level 23 is the same for all decisions at level 23, no matter which choices are made at previous or subsequent levels. We will call this the *separate action cost* model. Figure 3.2 shows an example for a small binary tree. A tree of depth d and branching factor b requires db parameters, one for each action at each level.

The separate action cost model is just a generalization of the assumptions used by DFS and the other search algorithms discussed in Chapter 2. DFS uses an exponential cost assumption in which later children high in the tree have costs greater than the sum of the worst choices at all lower levels. For a tree of depth d and maximum branching factor b, DFS assumes that the cost of child r at level l is rb^{d-l}. In iterative broadening, later children are assumed to be more expensive than taking any combination of lower ranked children. In other words, child r costs roughly d^r. ILDS assumes that each non-preferred child costs the same, no matter what the depth. (One could model slight differences between levels, depending on whether ILDS was implemented to explore discrepancies at the top or bottom of the tree first.) DDS assumes that discrepancies at the bottom of the tree are expensive, essentially assuming a cost of rb^l for child r at level l. And for interleaved depth-first search (IDFS), the assumptions depend on the parameters used but are roughly equivalent to DFS with the additional stipulation that all actions in the first few levels of the tree cost 0. This yields depth-first searching within multiple subtrees.

Although the model generalizes many current tree search algorithms, there are, of course, many orderings it cannot represent. The model generalizes across the breadth of the tree, assuming that all children of a particular rank at a given depth have the same cost no matter how they were reached. The effect of decisions at each level is assumed to be independent from those at other levels. This disallows a model in which taking many poor actions has an effect only slightly worse than taking a few poor actions.

The separate action cost model is simple and can be related directly to observed leaf costs because that is what the model attempts to predict. But recall that the adaptive probing algorithm also needs a method for using the model to decide which child to choose at each decision node. We do not want to always choose the action with the lower estimated cost because the difference between actions might be tiny and our estimates might be inaccurate. Instead, we will use a technique from reinforcement learning called Q-value sampling (Wyatt, 1997; Dearden, Friedman, and Russell, 1998) in which actions are selected with the probability that they are best. To calculate these probabilities in adaptive probing, the model will need a notion of action cost variance. In addition to estimating of the costs of each action, we will estimate the variance of the action costs by assuming that each estimated cost is the mean of a normal distribution, with all actions having the same variance.

3.2.1 Learning the Model

This model is easy to learn during a search. Each probe from the root corresponds to a sequence of actions and results in an observed leaf cost. If a_i is the cost of taking action a at depth i and l_k is the cost of the kth leaf seen, probing three times in a binary tree of depth three might give the following information:

$$\begin{array}{llllll} L_0 & + L_1 & + & & R_2 & = l_0 \\ L_0 & + & R_1 + & L_2 & & = l_1 \\ & R_0 + L_1 & + & L_2 & & = l_2 \end{array}$$

We can then estimate the a_i using a least squares regression algorithm. In the experiments reported below, the Widrow-Hoff procedure was used to estimate the parameters incrementally on-line (Bishop, 1995; Cesa-Bianchi, Long, and Warmuth, 1996).

This simple gradient descent method (also known as LMS, and very similar to the Perceptron) updates each cost according to the error between a prediction of the total leaf cost using the current action estimates, $\hat{l}_k$, and the actual leaf cost, l_k. If d actions were taken, we update each of their estimates by

$$\eta \frac{(l_k - \hat{l}_k)}{d}$$

where η controls the learning rate (or gradient step-size). All results reported below use $\eta = 0.2$, although similar values also worked well. (Values of 1 and 0.01 resulted in reduced performance.) This update requires little additional memory, takes only linear time, adjusts d parameters with every leaf, and often performed as well as an impractical $O(d^3)$ singular value decomposition estimator. It should also be able to track changes in costs as the probing becomes more focused, if necessary. Essentially, though, we regard the target values as fixed. More complex approaches, such as the Kalman filter , are required to deal with time-varying state estimation, which is a slightly different problem.

Because we assume that it is equal for all actions, the variance is also straightforward to estimate. If we assume that the costs of actions at one level are independent from those at another, then the variance we observe in the leaf costs must be the sum of the variances of the costs selected at each level. The only complication is that the variance contributed by each level is influenced by the mean costs of the actions at that level—if the costs are very different, then we will see variance even if each action has none. More formally, if X and Y are independent and normally distributed with common variance σ^2_{XY}, and if W takes its value according to X with probability p and Y with probability $1-p$, then

$$
\begin{aligned}
\sigma^2_W = &\quad E(W^2) - \mu^2_W \\
= &\quad p(\mu^2_X + \sigma^2_{XY}) + (1-p)(\mu^2_Y + \sigma^2_{XY}) - (p\mu_X + (1-p)\mu_Y)^2 \\
= &\quad \sigma^2_{XY} + p\mu^2_X + (1-p)\mu^2_Y - (p\mu_X + (1-p)\ \mu_Y)^2
\end{aligned}
$$

Since we can easily compute p by recording the number of times each action at a particular level is taken, and since the action costs are estimates of the μ_i, we can use this formula to subtract away the effects of the different means. Following our assumption, we can then divide the remaining observed variance by d to distribute it equally among all levels.

3.2.2 Using the Model

Using the model during tree probing is also straightforward. If we are trying to minimize the leaf cost, then for each decision, we want to select the action with the lower expected cost (i.e., the lower mean). As discussed above, we do not always want to select the child with the lower estimated cost. Rather, we merely wish to select each action with the probability that it is truly best. Given that we have estimates of the means and variance of the action costs and we know how many times we have tried each action, we can compute the probability that one mean is lower than another using a standard test for the difference of two sample means. We then choose each action according to the probability that its mean cost is lower. (Preliminary experiments in which an action was chosen according to the probability that a sample from its cost distribution will be less than one from the other's gave worse performance, as expected.)

For deciding between more than two actions, we can just sample from the distributions of the sample means and choose the action whose distribution gave the lowest sampled value. (Note that this is different from sampling from the distributions of the costs themselves.) This gives the desired result without computing any explicit probabilities. To eliminate any chance of the algorithm converging to a single path, the probability of choosing any action is clamped at $0.05^{1/d}$ for a depth d tree, which ensures at least one deviation on 95% of probes.

Now we have a complete adaptive tree probing algorithm. It assumes the search tree was drawn from a simple model of additive discrepancy costs and it learns the parameters of the tree efficiently on-line. Exploitation of this information is balanced with exploration according to the variance in the costs and the number of times each action has been tried. The method extends to trees with large and non-uniform branching factors and depths. The underlying model should be able to express assumptions similar to those built into algorithms as diverse as depth-first search and depth-bounded discrepancy search, as well as many other weightings not captured by current systematic methods.

3.3 Empirical Evaluation

Because the model class we are using can express such a wide range of trees and because a learning algorithm exists for the incremental setting, it seems as if the algorithm is guaranteed to perform well. However, there are several challenges facing the algorithm. Learning could fail for several reasons:

1. Most learning algorithms, including the one used here, have been proved effective only for the case in which samples are drawn at random. But because we are using the model to guide search, the samples are not random at all. After many leaves have been seen, the probing algorithm is likely to find one particular action more likely to have lower cost than another and it will bias the sampling accordingly.

2. Learning is driven entirely by the leaf costs. If the level of noise is high and they are not informative enough, no useful information will be extracted.

3. A useful model must be learned quickly enough that the search can adapt to the current problem instance. Slowly learning a general trend may not be enough to achieve good performance within a reasonable amount of time.

We first investigate the performance of this adaptive probing algorithm using an abstract model of heuristic search. This gives us precise control over the density of good leaves and the accuracy of the heuristic. We will find that adaptive probing outperforms systematic methods on large trees when the node-ordering heuristic is moderately inaccurate, and exhibits better worst-case performance whenever the heuristic is not perfect at the bottom of the tree. To ensure that our conclusions apply to more complex domains, we will also evaluate the algorithm using two NP-complete search problems: the combinatorial optimization problem of number partitioning and the goal-search problem of boolean satisfiability. It performs well on satisfiability and the naive formulation of number partitioning, but when using the powerful Karmarkar-Karp heuristic, it is competitive only for long run-times or when exploiting significant prior knowledge.

3.3.1 An Abstract Tree Model

In this model, introduced by Harvey and Ginsberg (1995) for the analysis of limited discrepancy search, one searches for goal nodes in a binary tree of uniform depth. This is an abstraction of the search trees arising in constraint satisfaction problems. (In combinatorial optimization, each leaf would have a score, rather than either being a goal or not.) Each node either has a goal below it, in which case it is *good*, or does not, in which case it is *bad*. The root is good and bad nodes only have bad children. Goals are distributed according to two parameters: m, which controls goal density, and p, which controls the accuracy of the heuristic. The generation of children from the root can be thought of as a kind of extinction process, whereby goodness is passed on to one or both of the children. The probabilities of the various configurations of parent and children are:

$$\begin{aligned}
&\mathrm{P}(\text{good} \rightarrow \text{good good}) &&= 1 - 2m \\
&\mathrm{P}(\text{good} \rightarrow \text{bad good}) &&= 1 - p \\
&\mathrm{P}(\text{good} \rightarrow \text{good bad}) &&= 2m - (1 - p)
\end{aligned}$$

The expected number of goal nodes is $(2 - 2m)^d$, where d is the depth of the tree.

Following Walsh's (1997) analysis of depth-bounded discrepancy search (DDS), we will estimate the number of leaves that each algorithm must examine before finding a goal using empirical measurements over random trees. Random trees can be generated lazily during the search by deterministically propagating seed values for a random generator down the tree. To provide a leaf cost measure for adaptive probing, we continue the analogy with the constraint satisfaction problems that motivated the model and define the leaf cost to be the number of bad nodes in the path from the root. (If we were able to detect failures before reaching a leaf, this would be the depth remaining below the prune.) The results presented

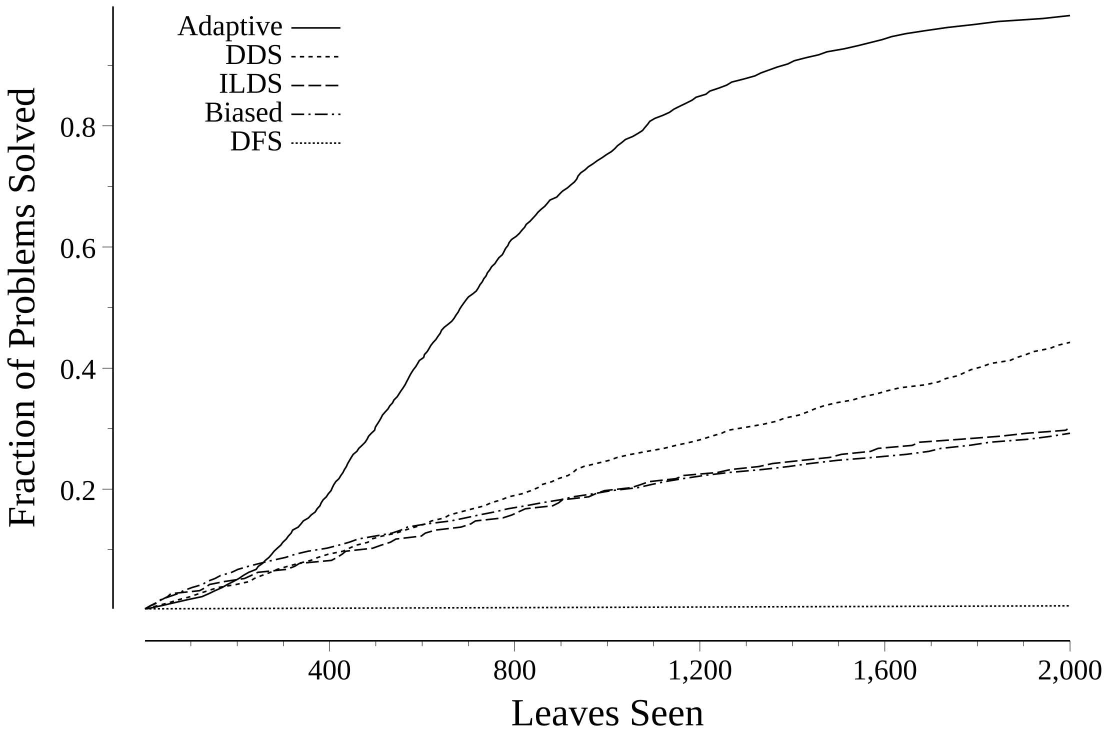

Figure 3.3: Probability of finding a goal in trees of depth 100 with $m = 0.1$ and p linearly varying between 0.9 at the root and 0.95 at the leaves.

below are for trees of depth 100 in which $m = 0.1$. The probability that a random leaf is a goal is 0.000027. By investigating different values of p, we can shift the locations of these goals relative to the paths preferred by the heuristic.

Figure 3.3 shows the performance of DFS, Korf's (1996) improved version of limited discrepancy search (ILDS), DDS, and adaptive probing on 2,000 trees. A heuristic-biased probing algorithm is also shown. This algorithm selects the preferred child with the largest probability that would be allowed during adaptive probing. Following Walsh, we raise the accuracy of the heuristic as depth increases. At the root, $p = 0.9$ which makes the heuristic random, while at the leaves $p = 0.95$ for 75% accuracy. ILDS was modified to incorporate this knowledge and take its discrepancies at the top of the tree first.

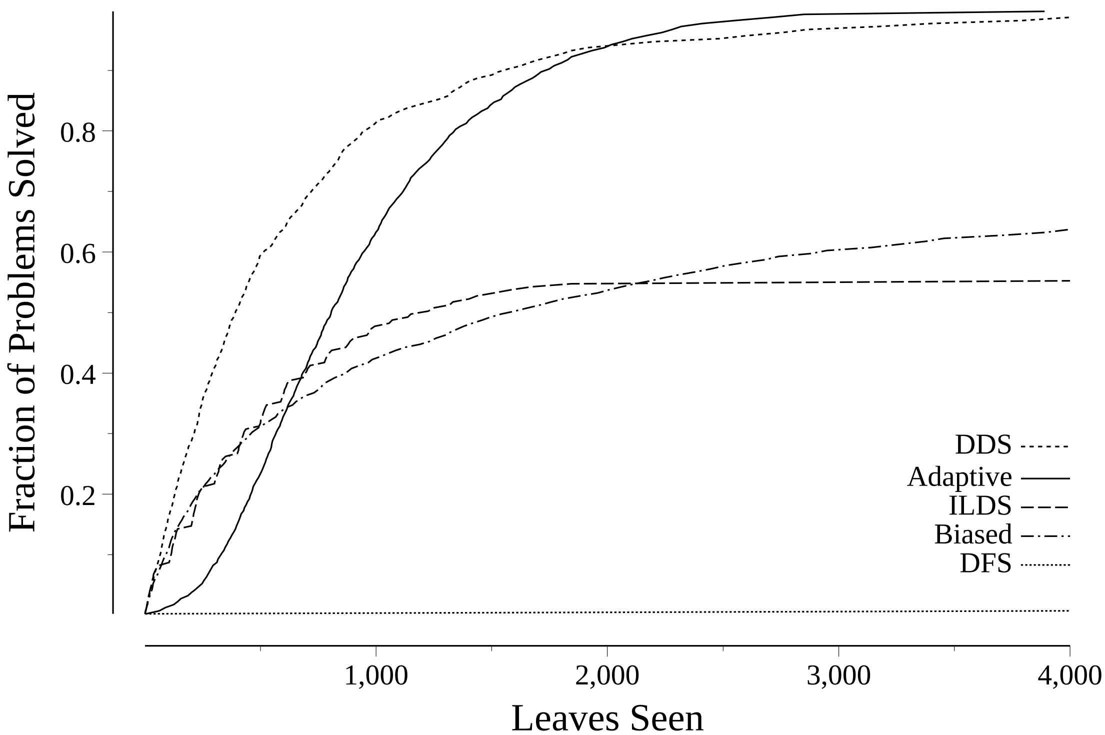

Figure 3.4: Performance on trees of depth 100, $m = 0.1$, and p varying from 0.9 at the root to 0.98 at the leaves.

Adaptive probing quickly learns to search these trees, performing much better than the other algorithms. Even though DDS was designed for this kind of tree, its assumptions are too strong and it only branches at the very top of the tree. ILDS wastes time by branching equally often at the bottom where the heuristic is more accurate. The *ad hoc* biased probing algorithm, which branches at all levels, is competitive with ILDS (and will actually surpass it, given more time) but fails to exploit the structure in the search space. DFS vainly branches at the bottom of the tree, ignorant of the fatal mistake higher in the tree, and solves almost no problems within 2,000 leaves. The superiority of adaptive probing over the heuristic-biased sampling indicates that the tree model is providing a benefit. Adaptive probing's performance cannot be explained simply by its basic stochastic sampling framework.

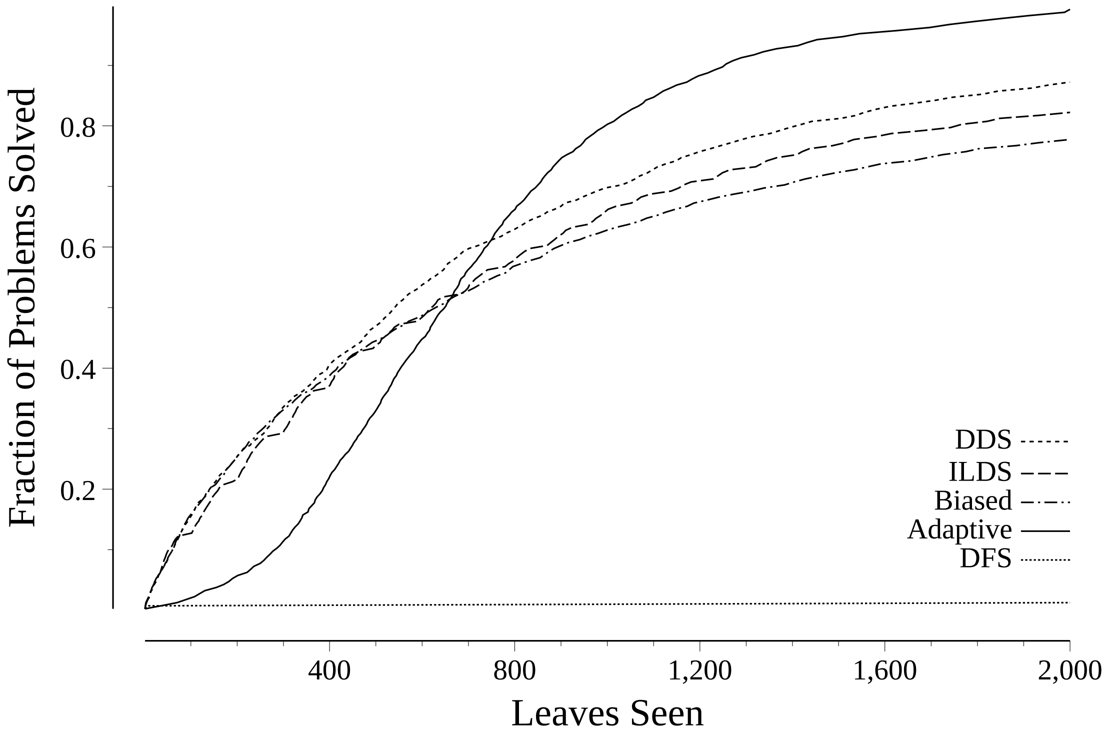

Figure 3.5: Performance on trees of depth 100, $m = 0.1$, and p varying from 0.98 at the root to 0.9 at the leaves.

DDS does better when the heuristic is more accurate, since its steadfast devotion to the preferred child in the middle and bottom of the tree is more often correct. Figure 3.4 shows the algorithms' performance on similar trees in which the heuristic is accurate 90% of the time at the leaves. DDS has better median performance, although adaptive probing exhibits more robust behavior, solving all 2,000 problems within 4,000 leaves. DDS had not solved 1.4% of these problems after 4,000 leaves and did not complete the last one until it had visited almost 15,000 leaves. In this sense, DDS has a heavier tail in its cost distribution than adaptive probing. Similar results were obtained in trees with uniform high p. Adaptive probing avoids entrapment in poor parts of the tree at the expense of an initial adjustment period.

Even with an accurate heuristic, however, the assumptions of DDS can be violated. Figure 3.5 shows what happens in trees in which the heuristic is accurate 95% of the time at the top of the tree and random at the very bottom. DDS still has an advantage over ILDS because a single bad choice can doom an entire subtree, but adaptive probing learns a more appropriate strategy.

To ensure that our insights from experiments with the abstract tree model carry over to other problems, we will now evaluate the algorithms on three additional kinds of search trees.

3.3.2 Boolean Satisfiability

Boolean satisfiability is the problem of determining whether a given formula in propositional logic can ever be true (has any satisfying models) or whether it is self-contradictory, as in $p \wedge \neg p$. It is a fundamental problem in logical inference and, in recent years, it has been used as a target representation for compilers from problems such as planning and graph coloring. It is also used for circuit verification.

Following Walsh (1997), we generated problems according to the random 3-SAT model with 3.5 clauses per variable and filtered out any unsatisfiable problems. All algorithms used unit propagation, selected the variable occurring in the most clauses of minimum size, and preferred the value whose unit propagation left the most variables unassigned. The cost of a leaf was computed as the number of variables unassigned when the empty clause was encountered.

Figure 3.6 shows the percentage of 200-variable problems solved as a function of the number of nodes generated. The distribution for DDS extends beyond the plot to $10^{6.7}$. ILDS, random probing, and adaptive probing solved all problems within 10^6 nodes. Al-

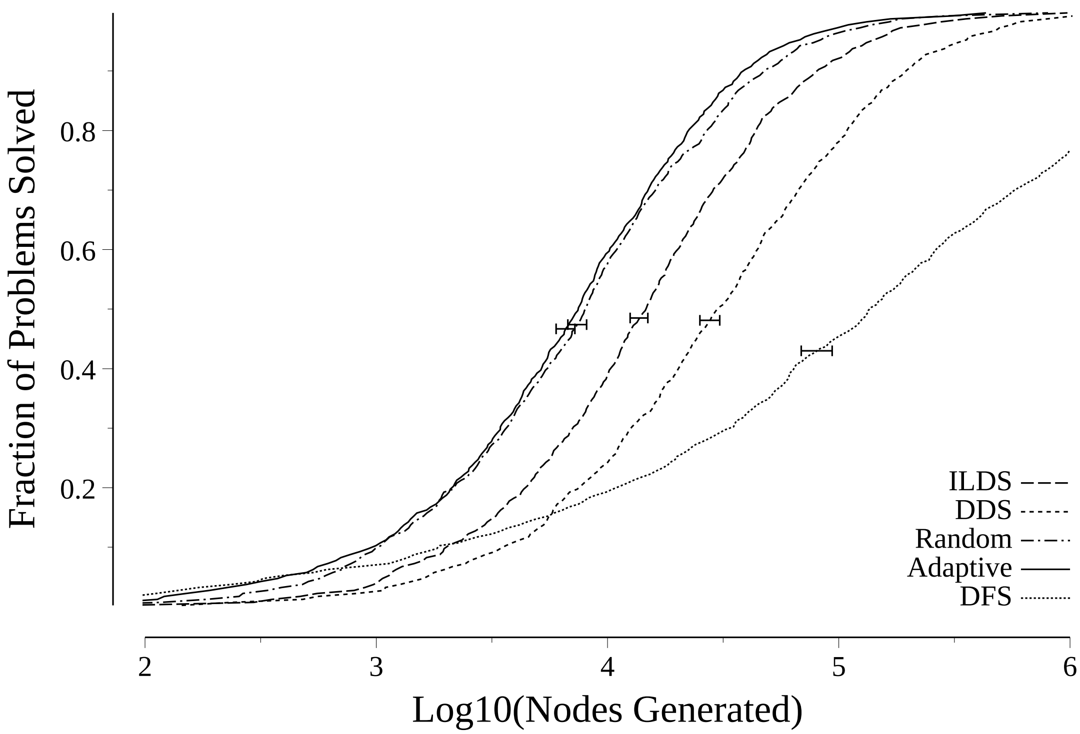

Figure 3.6: Fraction of random 3-satisfiability problems solved. Error bars indicate 95% confidence intervals around the mean over 1000 instances, each with 200 variables and 3.5 clauses per variable. (The DFS and DDS means are lower bounds.)

though Walsh used these problems to argue for the suitability of DDS, he measured leaves seen rather than nodes generated. As we saw in Section 2.5, DDS generates $O(n)$ nodes for every leaf and so performs worse when performance is measured in nodes. Both ILDS and purely random sampling perform significantly better than DDS. (Crawford and Baker (1994) similarly found random sampling effective on scheduling problems that had been converted to satisfiability problems.) DFS performs very poorly. Adaptive probing performs slightly better than random sampling (this is most noticeable at the extremes of the distribution). Although slight, this advantage persisted at all problem sizes we examined (100, 150, 200, and 250 variables).

3.3.3 Number Partitioning

Number partitioning is a simple yet NP-hard combinatorial optimization problem. The objective is to divide a given set of numbers into two disjoint groups such that the difference between the sums of the two groups is as small as possible. One might think of a load balancing problem or dividing a poorly cut pizza between two equally hungry graduate students. Number partitioning is a notoriously difficult problem: it was used by Johnson et al. to evaluate simulated annealing (1991), Korf to evaluate ILDS (1996), and Walsh to evaluate DDS (1997).

When the numbers are chosen uniformly over an interval, the difficulty of the problem depends on the relation between the number of digits in the numbers and the number of numbers. With few digits and many numbers, the probability of a partitioning with a difference of 0 or 1 increases (Karmarkar et al., 1986). This makes the tree search easier, as the search can terminate once such a partitioning is found. To encourage difficult search trees, we can reduce the chance of encountering a perfectly even partitioning by increasing the number of digits in each number. The experiments below use instances with enough digits that, based on the results of Karmarkar et al., we would expect an instance to have a perfect partition with probability 10^{-5}. For n numbers and perfect partition probability p, this is:

$$\left\lceil -\log_{10}\left(\frac{p}{2^n}\sqrt{\frac{n}{24\pi}}\right)\right\rceil$$

For 64 numbers, this is 25 digits; for 128 numbers, 44 digits; and for 256 numbers, 82 digits. These sizes also fall near the hardness peak for number partitioning (Gent and Walsh, 1996), which specifies $log_{10}2^n$ digits for a problem with n numbers.

Common Lisp, which provides arbitrary precision integer arithmetic, was used to implement the algorithms. Results were normalized as if the original numbers had been between 0 and 1. To better approximate a normal distribution, the logarithm of the partition difference was used as the leaf cost.

Two Search Representations

There are two popular ways of representing number partitioning as a tree search problem. The first is a straightforward greedy encoding in which the numbers are sorted in descending order and then each decision places the largest remaining number in a partition, preferring the partition with the currently smaller sum.

A more sophisticated representation for number partitioning was suggested by Korf (1995), based on the heuristic of Karmarkar and Karp (1982). The essential idea is to postpone the assignment of numbers to particular partitions and merely constrain pairs of number to lie in either different bins or the same bin. The decisions of the algorithm build up a simple constraint graph specifying which numbers are in the same or different

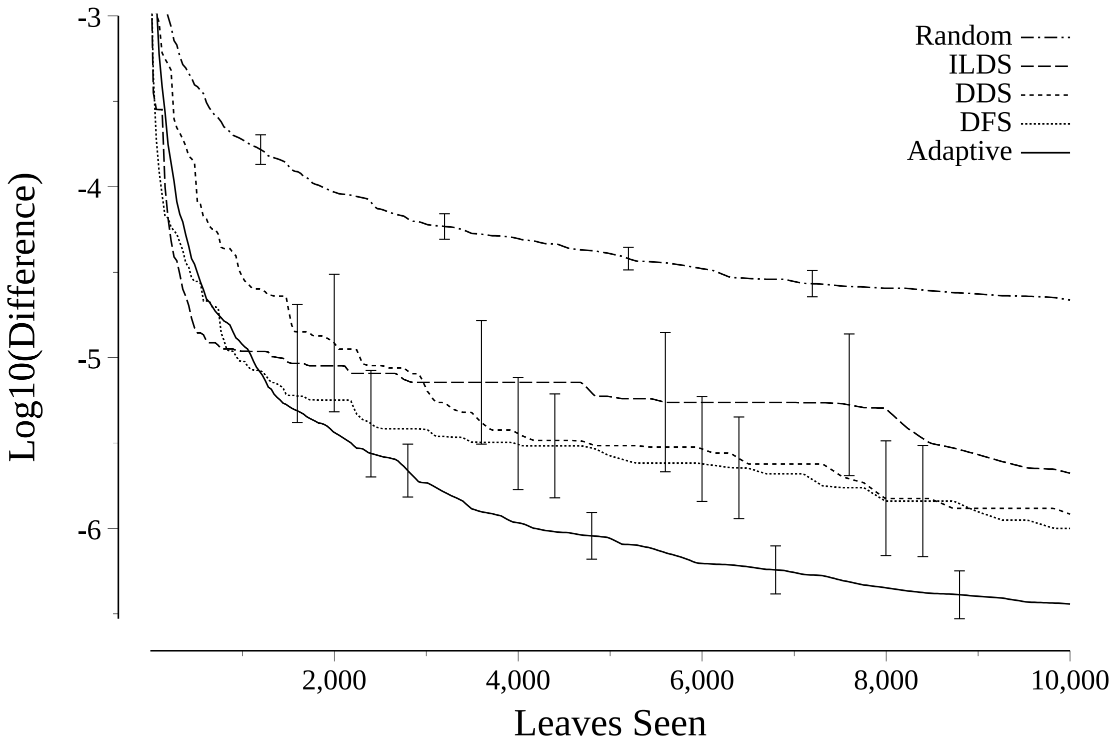

Figure 3.7: Searching the greedy representation of number partitioning. Error bars indicate 95% confidence intervals around the mean over 20 instances, each with 128 44-digit numbers.

bins as other numbers. As with the greedy representation, a sorted list is maintained in decreasing order, but instead of assigning numbers to particular partitions, each decision merely commits to placing the largest two numbers in the same partition or in different partitions. Different partitions are preferred, and the difference between the two numbers is then inserted into the list to be dealt with as any other number. Otherwise, the sum is inserted. When only one number is left, the constraint graph can be two-colored to yield a partitioning. As we will see, this representation creates a very different search space from the greedy heuristic.

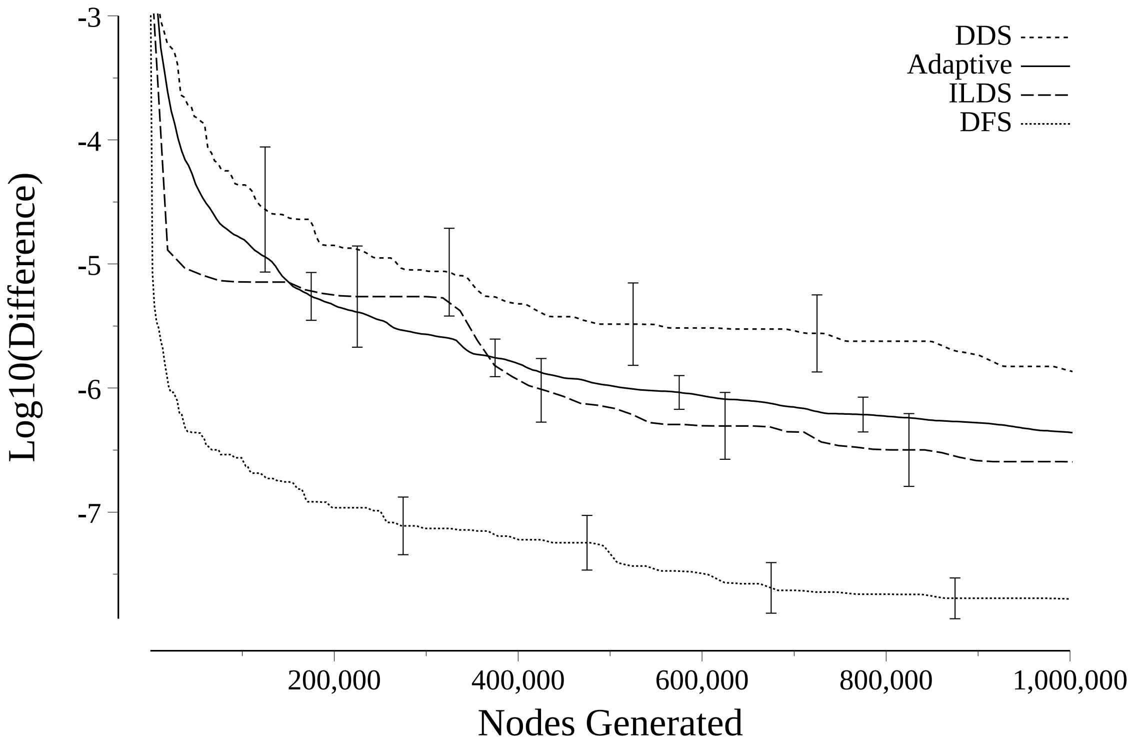

Figure 3.8: Performance on the greedy representation of number partitioning as a function of nodes generated.

Results with the Greedy Representation

We will consider the plain greedy encoding first. Figure 3.7 compares the performance of adaptive tree probing with DFS, ILDS, DDS, and completely random tree probing. To provide a comparison of the algorithms' search orders, the horizontal axis represents the number of leaves seen.

The relative performance of the algorithms indicates that, in this search tree, taking discrepancies in the middle of the tree does not seem to help. ILDS seems to have the worst search order. Adaptive probing starts off poorly, like random sampling, but surpasses all other algorithms after seeing about 1,000 leaves. It successfully learns an informative model of the tree and explores the leaves in a more productive order than the systematic algorithms.

However, recall that adaptive tree probing suffers the maximum possible overhead per leaf, as it generates each probe from the root. (This implementation did not attempt to reuse initial nodes from the previous probe.) The number of nodes (both internal and leaves) generated by each algorithm should correlate well with running time in problems in which the leaf cost is computed incrementally or in which the node-ordering heuristic is expensive. Figure 3.8 compares the algorithms on the basis of generated search nodes. (To clarify the plot, DFS and ILDS were permitted to visit many more leaves than the other algorithms.) In a demonstration of the importance of overhead, DFS dominates all the other algorithms in this view, and ILDS performs comparably to adaptive probing. DFS reuses almost all of the internal nodes on each leaf's path, generating only those just above the leaves. Since ILDS needs to explore discrepancies at every level of the tree, it will usually need to generate a significant fraction of the path down to each leaf. DDS, which limits its discrepancies to the upper levels of the tree, incurs overhead similar to that of adaptive probing because it never reuses internal nodes in the middle of the tree. ILDS finds better solutions in sudden bursts, corresponding to its exploration of discrepancies at the bottom of the tree. Its performance then plateaus as it takes discrepancies at other levels, until another jump (for instance, around 300,000 nodes). The plateaus of ILDS are more evident than in Figure 3.7 because the leaves with discrepancies at lower levels can be visited using fewer new internal nodes.

On instances using 64 numbers, adaptive probing again dominated DDS, but was clearly surpassed by ILDS. (It performed on par with a version of ILDS that visited discrepancies at the top of the tree before those at the bottom.) This suggests that, in these search trees, the advantage of adaptive probing over ILDS and DDS increases with problem size.

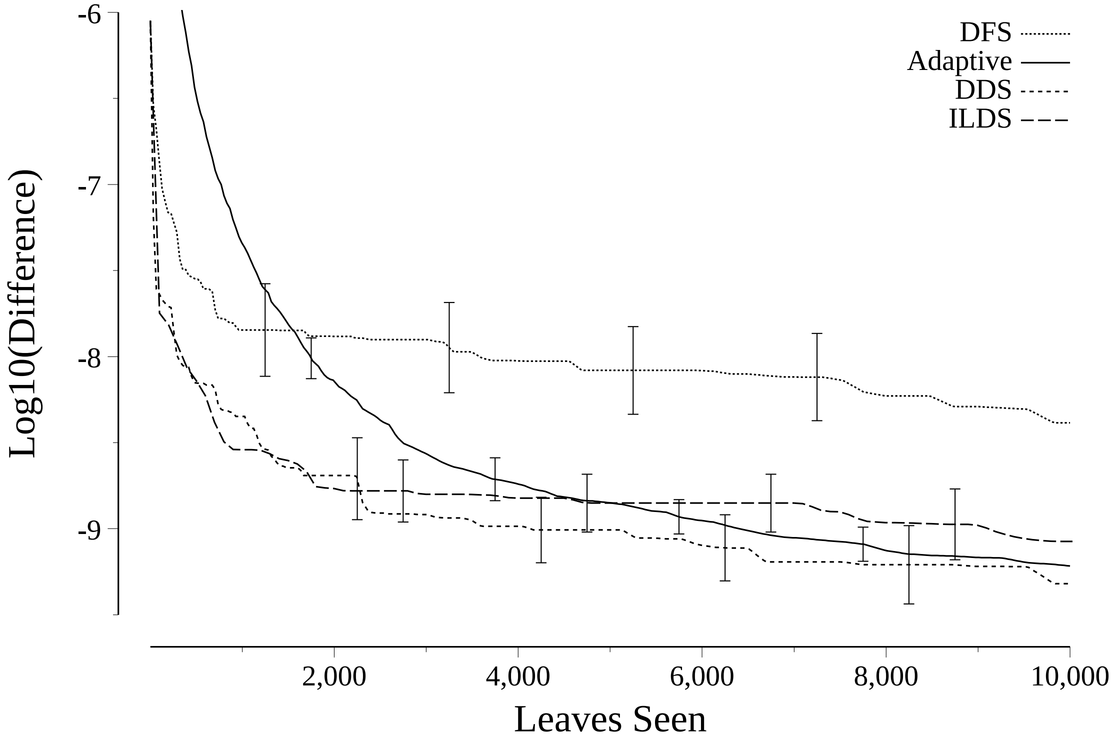

Figure 3.9: Searching the CKK representation of number partitioning. Each instance had 64 25-digit numbers.

Results with the CKK Representation

Figure 3.9 shows the performance of the algorithms in the alternative CKK encoding of the partitioning problem. Performance is shown as a function of leaves seen. DDS has a slight advantage over ILDS, although adaptive probing is eventually able to learn an equally effective search order. DFS and random sampling too often go against the powerful heuristic. As in the greedy representation, however, interior node overhead is an important consideration. Figure 3.10 shows that DDS and adaptive probing are not able to make up their overhead, and results using 128 numbers suggest that these difficulties increase on larger problems. Bedrax-Weiss (1999) argues that the CKK heuristic is extraordinarily effective at capturing relevant information and that little structure remains in the space. These results are consistent with that conclusion, as the uniform and limited discrepancies

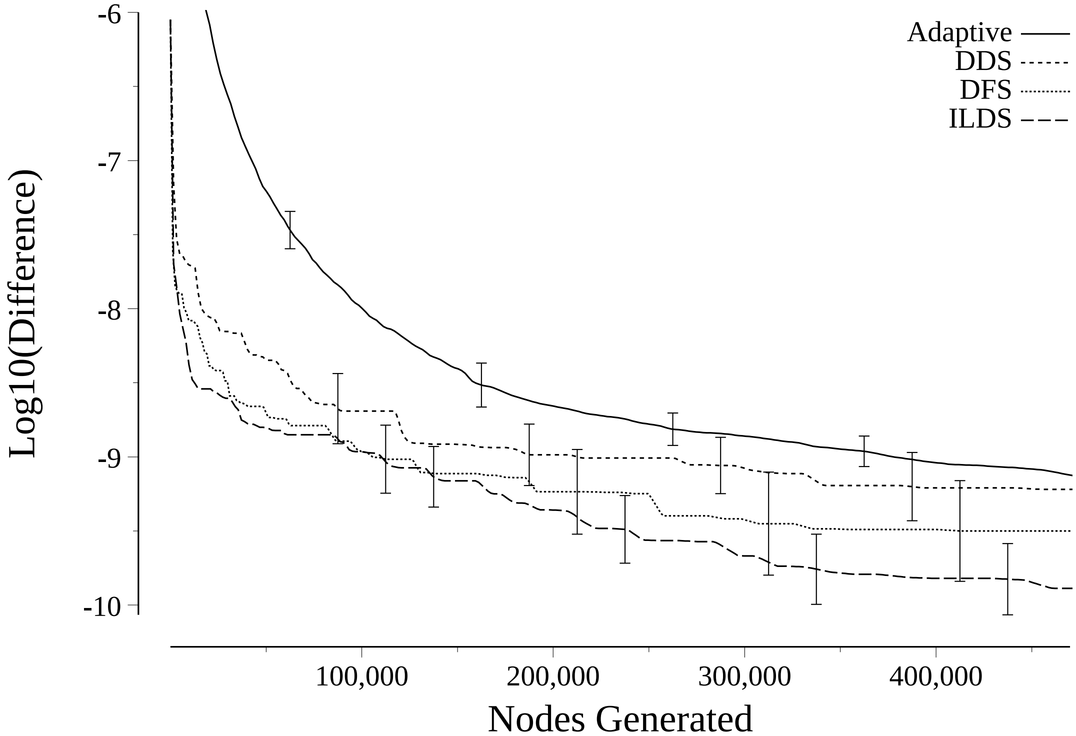

Figure 3.10: Performance on the CKK representation of number partitioning as a function of nodes generated.

of ILDS appear best.

3.3.4 Summary of Results

We have now seen that an adaptive approach to tree search has substantial promise. In each search space we examined, the systematic search algorithms ranked differently in performance. A simple adaptive probing technique used with a straightforward action cost model can adapt its searching behavior to search spaces with different characteristics. It is therefore more robust across different domains than algorithms with fixed assumptions. Experiments in an abstract tree model derived from constraint satisfaction problems showed that adaptive probing could even outperform DDS in the search spaces it was designed for. On boolean satisfiability problems, adaptive probing surpassed all other methods. Results on greedy number partitioning showed that adaptive probing exhibited an excellent search order, although its overhead made its performance only competitive instead of superior. The only disappointment was the method's performance on the CKK representation of number partitioning, on which it exhibited very slow improvement. In the following sections, we will remedy this flaw.

3.4 Using Previous Experience

We have now seen that adaptive probing can successfully adapt to a wide variety of trees and that it is competitive with systematic algorithms except when the heuristic is very accurate. When the heuristic is very often correct, adaptive probing suffers the overhead of having to discover that fact from scratch. In this section, we investigate two methods for remedying this liability. Both are based on the idea of adding our prior knowledge as an element of the search. The first approach is simply to re-use a model that was built during a previous run on a similar problem. This avoids having to specify an initial bias manually, although it requires identifying classes of similar problems. The second is to use a pre-specified probing policy initially, but slowly discount its influence in favor of the learned model. We will empirically evaluate the effectiveness of these approaches on both representations of the number partitioning problem. We will see that the simpler method is more robust, while the combination of policies provides a less reliable advantage.

3.4.1 Reusing Learned Models

With db parameters and a single observed leaf cost per probe, adaptive probing will take at least db probes to estimate the costs of choosing each child. While the ability to adjust to any possible configurations of costs is admirable, it is unlikely that the heuristically preferred child is actually significantly worse than the others. We would like to avoid having to spend the time to learn this, while still maintaining the flexibility to change the model if evidence suggests we have encountered one of these rare situations. Perhaps the simplest way of avoiding a prolonged initial learning period is to begin with an estimated model. More specifically, we can use the costs estimated from a previous run on a similar problem, while resetting the variance to ∞ and the recorded number of counts for each action to 0. This should improve the accuracy of our estimated costs, starting the gradient descent learning procedure in a good part of weight-space and improving our identification of the preferred child as useful, while still allowing the algorithm plenty of latitude to explore and revise the costs as it gradually becomes confident in its estimates and focuses the search.

Number Partitioning: Greedy Space

We will first consider the plain greedy formulation of number partitioning. Figure 3.11 compares the performance of adaptive tree probing with DFS, ILDS, DDS, and completely random tree probing. Adaptive probing was run twice–the first time with its model's costs initialized to zero and the second time with the costs that were estimated by a previous run on a different problem instance. To provide a comparison of the algorithms' search orders, the horizontal axis represents the number of leaves seen.

The figure shows that adaptive probing, in addition to learning to explore a profitable part of the search space, benefits from the prior knowledge. While *tabula rasa* adaptive probing needs to see approximately 1,500 leaves before overtaking the systematic algorithms, the estimates transferred from the previous problem lead the algorithm directly to good solutions.

Although Figure 3.11 shows that adaptive probing with transferred knowledge quickly

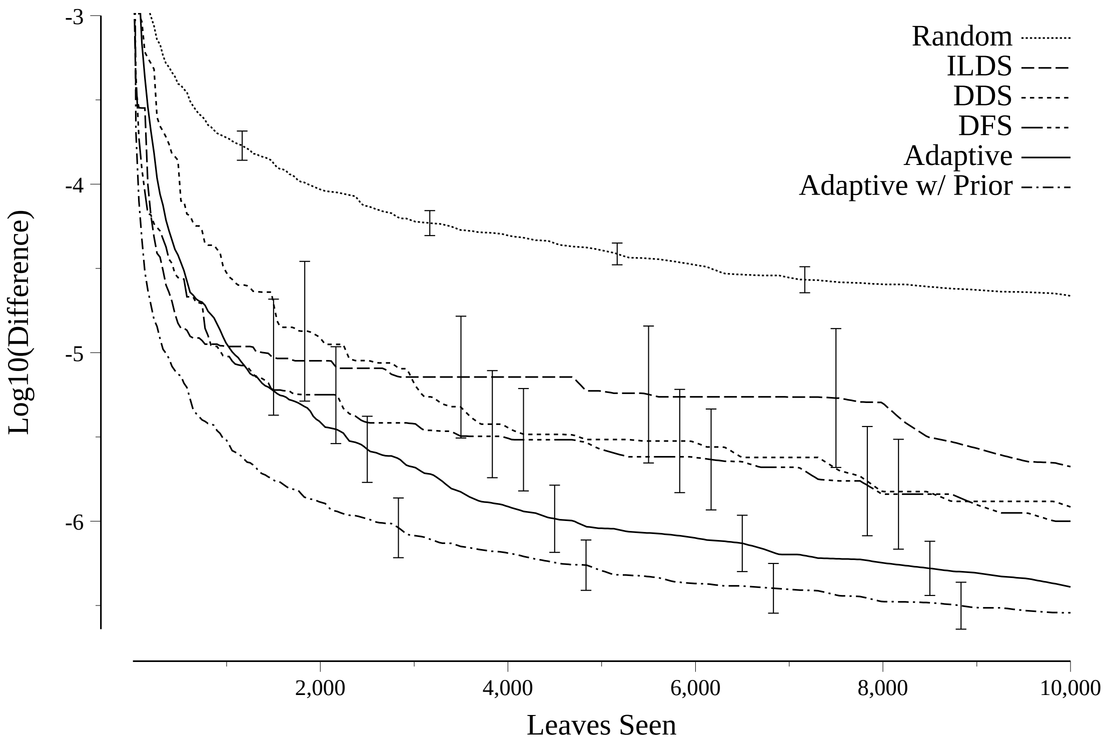

Figure 3.11: Searching the greedy representation of number partitioning. Error bars indicate 95% confidence intervals around the mean over 20 instances, each with 128 44-digit numbers.

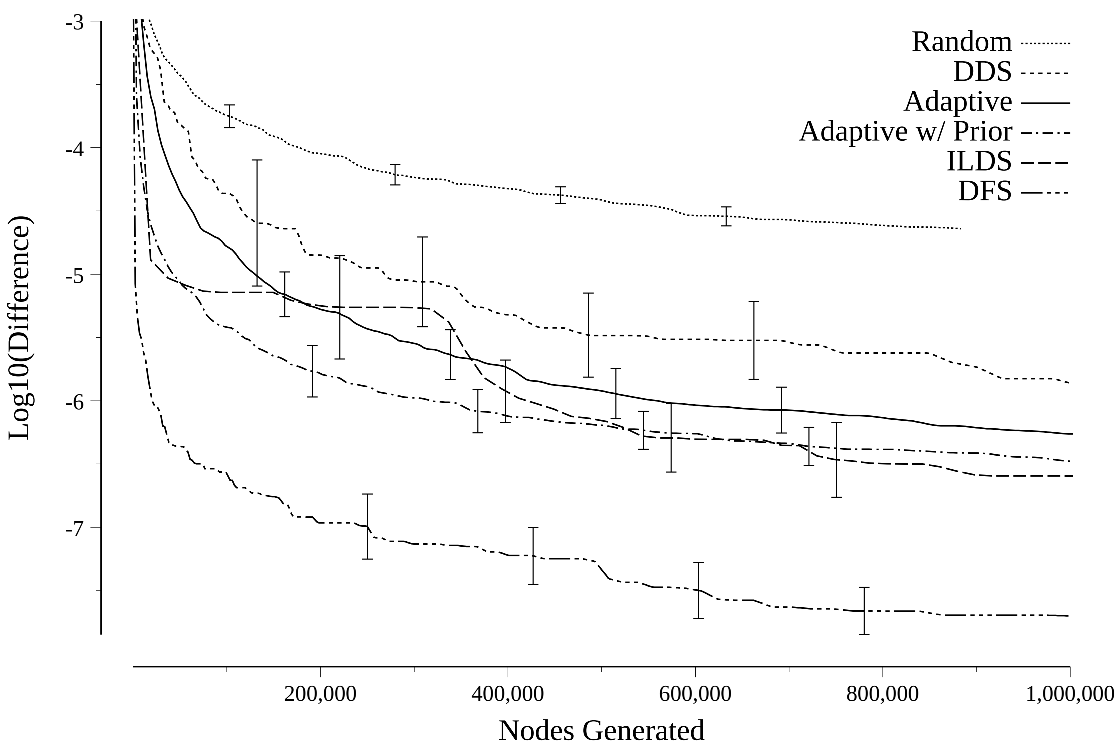

Figure 3.12: Performance on the greedy representation as a function of nodes generated.

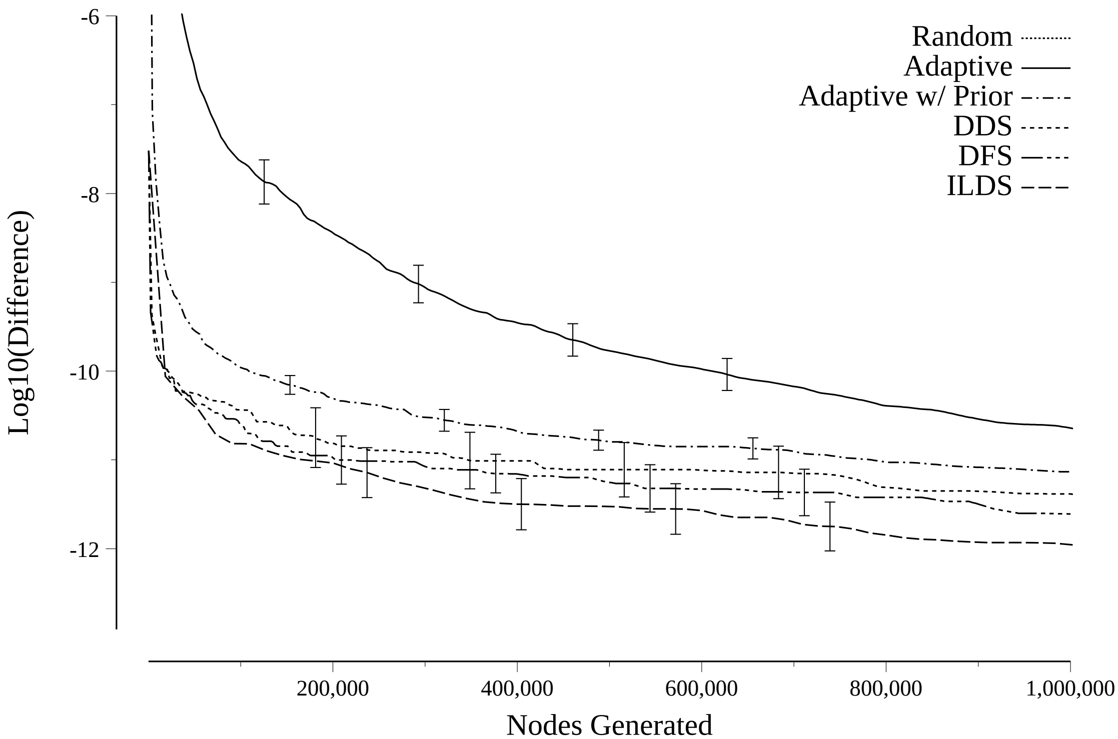

Figure 3.13: Searching the CKK representation of number partitioning problems.

learns a good search order, it ignores the overhead that is inherent in restarting at the root with each probe. Figure 3.12 corrects for this factor, showing performance as a function of the number of nodes (both internal and leaves). Recall that both adaptive probing and DDS suffer the maximum possible overhead compared to DFS, which generates roughly one internal node per leaf. This is reflected in the figure, as DFS finds superior solutions when given the same number of node generations. Although prior knowledge provides a benefit, it is not enough to overcome the inherent overhead of adaptive probing.

Number Partitioning: CKK Space

Now we will see if these general trends hold up when moving to the CKK search space. Figure 3.13 presents the performance of the search algorithms as a function of the number of nodes generated. Random probing would appear off the top of the plot. Ordinary adaptive probing takes a long time to learn that the heuristic is usually accurate everywhere, although it looks as if it may eventually approach the systematic algorithms' performance. When imbued with prior knowledge, adaptive probing quickly approaches DDS (which suffers similar node generation overhead). The benefit of using prior knowledge seems to be greater in this search space than in the greedy one, even though it is the harder one for plain adaptive probing. When the knowledge is harder to acquire, receiving it in advance represents a greater savings.

3.4.2 Blending Search Policies

While reusing an old model is easy and seems remarkably effective, it is only possible if one has the luxury of previous experience with a similar problem. If the previous problem has a very different distribution of leaf costs, the initial bias can be counter-productive. Another method for taking advantage of our *a priori* expectation that the heuristic is beneficial is to behave at first according to that belief, while continuing to learn a fresh model of the tree. We can then gradually reduce the frequency with which we make our decisions according to the prejudiced policy and begin to rely more heavily on our experience in the tree at hand. While this method applies even in the absence of experience with similar problems, it requires a prior judgment on how quickly to make the switch. This is essentially the same problem as deciding how much to trust the initial bias.

In the experiments reported below, we used a multiplicative discounting policy. At the first probe, we use the prior bias with probability 1. After every probe, this probability is multiplied by a constant less than one. In effect, this creates a changing blend of the initial policy and the current model. For an initial policy, we use an algorithm which selects the preferred child with the maximum probability that would be allowed under adaptive

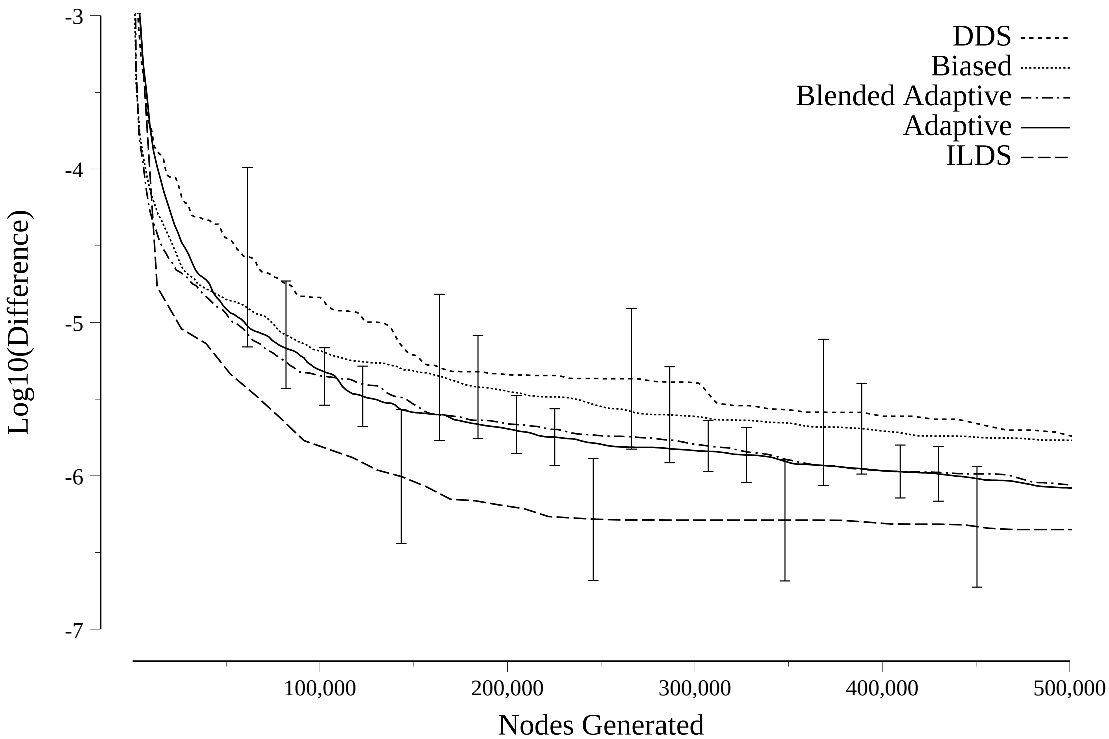

Figure 3.14: Searching the greedy representation of number partitioning instances, each with 64 25-digit numbers.

probing (recall that we clamped the probability of any child as a safeguard against complete convergence of the algorithm). We used a multiplicative constant such that, in a tree of depth d, we are as likely after $15d$ iterations to use the current estimated model as we are to use the prior bias. This constant is $0.5^{1/(15d)}$.

Evaluation

The empirical performance of blending policies was mediocre. Figure 3.14 shows algorithm performance using the greedy representation of number partitioning instances with 64 numbers. Besides plain adaptive probing and the blended policy, we also show the performance of a biased probing algorithm that just uses the initial policy of the blended algorithm. This biased probing algorithm performs on par with DDS in the greedy search space, but

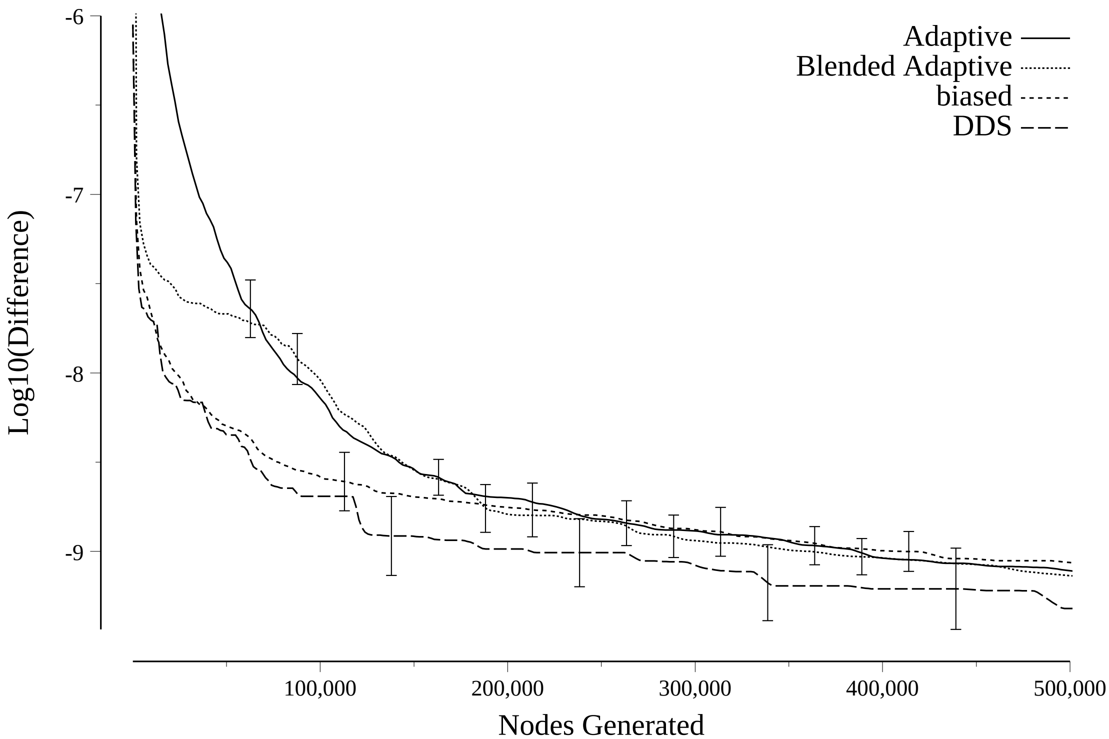

Figure 3.15: Searching the CKK representation of number partitioning instances, each with 64 numbers.

seems to be a little worse than plain adaptive probing. The blended adaptive probing algorithm seems equivalent to the plain. On larger problems, however, the blended policy was eventually surpassed by plain adaptive probing.

In the CKK search space, policy blending seemed to work reasonably well. Figure 3.15 shows the performance of the algorithms on 64-number problems. The blended algorithm follows the biased probing algorithm at first, then switches over to mimic the adaptive one. Unfortunately, the good performance of the initial biased policy seems to provide little benefit to the model learned by the adaptive component. Figure 3.16 shows performance on larger problems. Here, the model learned by the blended algorithm seems to have benefitted from its initial experience, although the algorithm still suffers a significant stagnant period during the transition in which little improvement is seen. Use of a shorter or longer blending

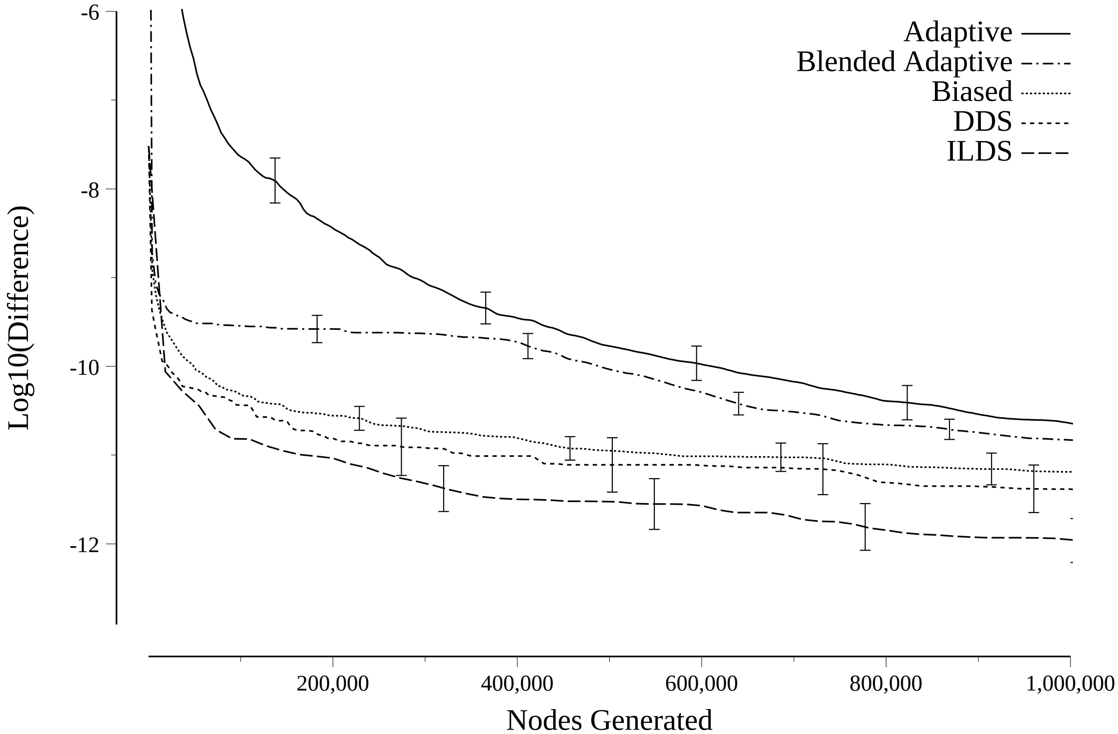

Figure 3.16: Searching the CKK representation of instances with 128 numbers.

time seemed to result in worse performance in preliminary experiments on small problems. Using an abrupt changeover rather than a gradual blending also led to the learning of a poor model. In short, blending policies seems to function more to temporarily override learning than to assist it.

3.4.3 Summary of Results

We investigated two methods for using prior knowledge with an adaptive probing algorithm. The simplest one, merely reusing the action costs estimated on a similar problem, seemed to perform the best. An attempt to blend an *a priori* policy with the learning algorithm gave some improvement, but seems prone to leading to ineffective learning. We conclude that it is better to directly aid the learning process rather than temporarily override it.

3.5 Parametric Action Cost Models

We have seen that aiding learning by using prior experience is effective for improving the performance of adaptive probing. Another way of making learning easier is to simplify and restrict the underlying cost model. Our model has so far allowed action costs to vary arbitrarily much from one depth to another and from one child to another. This seems unlikely to be necessary. We will now try a more restricted model, forcing all action costs for the same child rank to be related by a smooth quadratic function of depth. This restriction means that there will be fewer parameters to learn and that the kinds of trees that can be modeled will be more limited.

We will model the cost of a leaf as the sum of the costs of the actions taken to reach it, as before. If $cost_{k,d}$ is the cost of action k at level d, then

$$leaf = \alpha + \sum_{\mathrm{d}} cost_{k,d}$$

as before (except for the new parameter α, which will be explained below). But now we also have the restriction that

$$cost_{k,d} = a_k + b_k d + c_k d^2$$

where a_k, b_k, and c_k are the coefficients of the quadratic function, indexed by the child rank k. Note that the leaf cost will be linear in the quadratic's parameters. It is these parameters that we now estimate from the leaf costs, using the same on-line technique as before. Instead of db parameters for a tree of depth d with branching factor b, we have $3b + 1$ parameters.

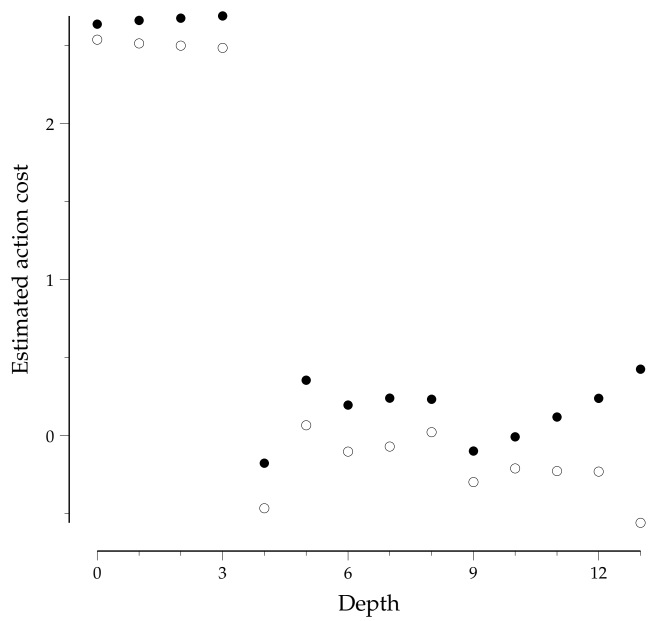

Figure 3.17: The action costs learned for an 18-number partitioning problem using the CKK representation. Filled circles represent the non-preferred actions.

The α in the leaf cost model above is a new additional parameter of the model. This new parameter helps ensure that the other costs are smooth. When leaves lie at different levels of the tree due to pruning, the action costs higher in the tree must be quite a bit larger than those at the bottom, in order that the predicted leaf cost approach the correct order of magnitude quickly, using only levels above the pruning. This can be seen in Figure 3.17. These differences in absolute magnitude between levels do not affect the decisions of adaptive probing because only the actions available at the same level are ever compared. The α parameter can be thought of as the mean solution cost or the cost of the root node and it removes any need for the parameters to be very different to accommodate pruning.

To help keep the values of the parameters to be learned roughly similar in terms of their expected orders of magnitude, the quadratics were expressed not in terms of the usual depth levels, but in terms of the percentage of the maximum depth. This kept the parameter for

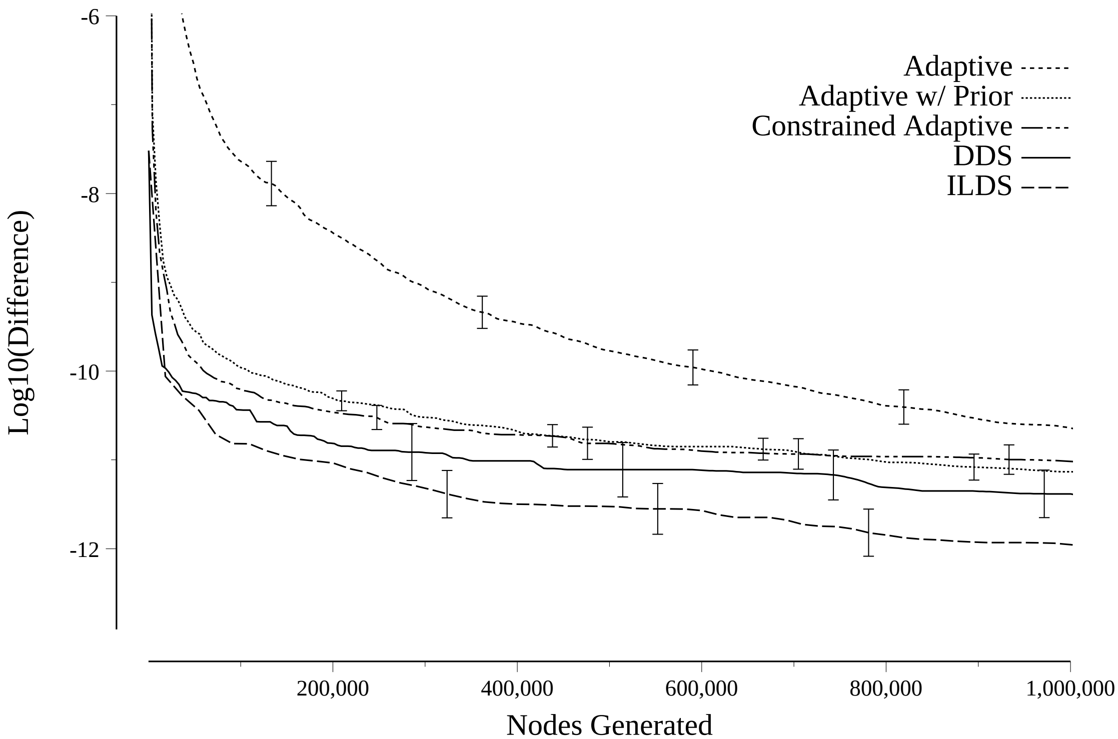

Figure 3.18: Adaptive probing in the CKK space using 128 numbers and a model which constrains action costs to be a quadratic function of depth.

the d term from having to be adjusted very much more finely than that for the d^2 term.

3.5.1 Evaluation

Figure 3.18 shows results of adaptive probing using the quadratic model on the number partitioning problem. As the figure shows, the model is learned very quickly, and is effective in guiding search. The knowledge conveyed by the restricted form of the model gives a performance improvement comparable to reusing a model from a previous run on similar problem, although the quadratic model does seem to suffer for its inflexibility as the run becomes longer. Figure 3.19 shows a model that was learned.

One could also imagine requiring action cost estimates at a particular depth to have a smooth relationship. In the TSP for example, there are $O(n)$ children at each level (one for

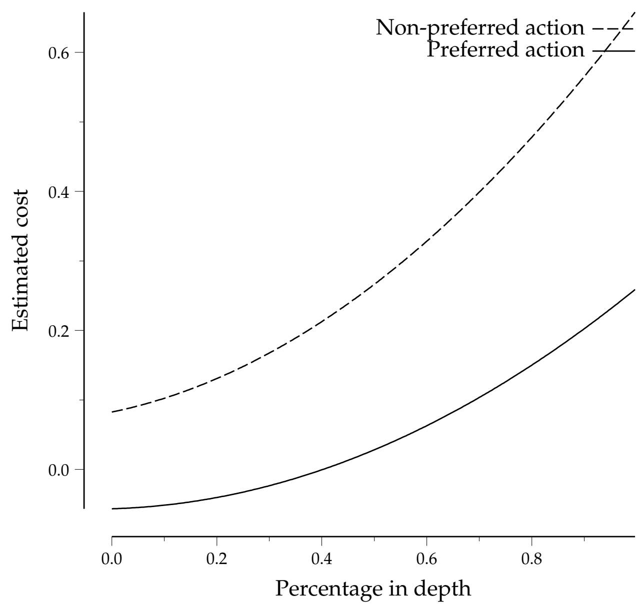

Figure 3.19: A quadratic action cost model learned by adaptive probing for searching the CKK representation.

each unvisited city). Instead of maintaining $O(n)$ costs at each depth, one might maintain a quadratic curve at each depth.

We explored one way to parameterize the action cost model: forcing the costs for each child rank to be quadratic in depth. Different domains might benefit from different parameterizations, however. In the traveling salesman problem, for example, both the depth and the branching factor are $O(n)$ (one child for each unvisited city at each level). It may be useful to constrain the action costs at a particular level to lie in a quadratic relation to each other, when ordered by child rank. (And if the action costs at a particular depth have a known relation to each other, it may be possible to choose among them in less than $O(n)$ time.) One might even imagine constraining the parameters of the quadratics at each level to vary smoothly with depth, perhaps according to an additional quadratic function. In this way, the $O(n^2)$ parameters for the traveling salesman problem could be represented with nine parameters.

3.6 Summary of Results

The adaptive approach to tree search is promising. Using an explicit model of the search space, an algorithm can learn which actions lead to better solutions in the current tree. As expected, we found this approach to be robust both within and across problem domains. As an added bonus, it was easy to incorporate prior experience on similar problems. Even when taking its overhead into account, adaptive probing seemed to perform respectably in every search space. The only space in which it was not the best or near the best was the CKK space for number partitioning, in which the node-ordering heuristic is very accurate. Of course, further work is needed to assess its performance in very different domains, such as those with a high branching factor, and against additional methods, such as interleaved depth-first search (Meseguer, 1997).

3.7 Related Work

Stochastic tree sampling and using learning to improve search performance have each been studied before, although never together in the context of combinatorial optimization. We will briefly review some of the recent work in these areas.

3.7.1 Tree Probing

Crawford and Baker (1994) investigated random tree probing on scheduling problems and found it remarkably effective when compared against both tree-based and iterative improvement satisfiability procedures.

Abramson (1991) used random sampling in two-player game trees to estimate the expected outcome of selecting a given move. He also discussed learning a model off-line to predict outcome from static features of a node. In an optimization context, Juillé and Pollack (1998) used random tree probing as a value choice heuristic during beam search, although no learning was used.

Bresina (1996) used stochastic probing for scheduling, introducing a fixed *ad hoc* bias favoring children preferred by the node-ordering heuristic. One can view adaptive probing as providing a way to estimate that bias on-line, rather than having to specify it beforehand, presumably using trial and error. (However, note that adaptive probing will eventually converge to sampling around a single privileged path once it has enough data.) By making explicit the dependence of the search on the information observed, an adaptive approach makes clear what the search depends on, easing design and debugging. Relieving the user from having to specify parameters also allows the use of a more complex and flexible model.

The Greedy Random Adaptive Search Procedure (GRASP) of Feo and Resende (1995) is, in essence, heuristic-biased stochastic probing with improvement search on each leaf. Adaptive probing provides a principled, relatively parameter-free, way to perform the probing step. By using the cost of the local minimum found by the improvement search as the leaf cost to learn from, adaptive probing would be learning where to disregard the node ordering heuristic in order to produce solutions that were the most useful starting points for the improvement search. (See also the STAGE algorithm, below.)

Finally, one could also interpret aspects of Ant Colony Optimization (ACO) algorithms (Dorigo and Gambardella, 1997), in which 'pheromone' accumulates to represent the information gathered by multiple search trials, as serving as an approximation of adaptive probing. In ACO, each search trial can be viewed as a stochastic probe into a tree that uses a different variable ordering. Much work on ACO has focused on the traveling salesman problem, so this corresponds to starting at a different city in each trial. Statistics are recorded for each edge in the graph on how often it is taken, corresponding to an estimation of the cost of each child at a decision node. By viewing this as adaptive probing, one might be able to avoid the laborious parameter tuning required in current ACO formulations.

3.7.2 Learning to Search

Squeaky-wheel optimization (Joslin and Clements, 1998) adapts during tree search, although it learns a variable ordering for use with a greedy constructive algorithm, rather than learning about the single tree that results from using an ordinary variable choice heuristic. The relative benefits of adapting the variable ordering as opposed to the value ordering seem unclear at present. Adaptive probing is slightly more general, as the squeaky-wheel method requires the user to specify a domain-specific analysis function for identifying variables that should receive increased priority during the next probe.

In the context of short-path algorithms, Korf (1990) introduced the learning real-time A* algorithm (LRTA*), which updates stored heuristic values to improve subsequent search. Nilsson (1998) discusses the use of reinforcement learning techniques such as temporal difference learning to adjusting weights for the components of a heuristic function for shortest-path search, although empirical results are not mentioned.

Adaptive tree probing is similar in spirit to iterative improvement algorithms such as adaptive multi-start (Boese, Kahng, and Muddu, 1994), PBIL (Baluja, 1997), and COMIT (Baluja and Davies, 1998) which explicitly try to represent promising regions in the search space and generate new solutions from that representation. For some problems, however, tree search is more natural and heuristic guidance is more easily expressed over extensions of a partial solution in a constructive algorithm than over changes to a complete solution. Adaptive probing gives one the freedom to pursue incomplete heuristic search in whichever space is most suitable for the problem. It is a promising area of future research to see how the two types of heuristic information might be combined.

Adaptive probing is also related to STAGE (Boyan and Moore, 1998), which attempts to predict promising starting points for hill-climbing given the values of user-specified problem-specific features. The discrepancy cost model requires less of the user, however, since the usual node-ordering function is used as the only problem-specific feature. The tree structure itself can be used to give the geometry for the search space model.

In the previous work on learning for improvement search, there has been some investigation of integrating prior knowledge for transferring problem-solving experience between problems. The X-STAGE algorithm (Boyan and Moore, 2000) is one example. The STAGE algorithm learns a model during search that predicts when an initial solution will yield good results with hill-climbing (or a similar algorithm). This model is used to intelligently restart after the hill-climbing has reached a local maximum by switching temporarily to hill-climbing according to the model's prediction of a solution's potential as a starting place (which might be different from its quality). In X-STAGE, several initial training problems are run and a separate model is learned on each using STAGE. These models are then used to solve a new problem by having each model vote on whether or not to accept a proposed modification to the starting solution. This avoids worrying about having to scale a model to appropriate values for use on a new problem, but does not allow any adaptation to the new problem instance. Zhang and Dietterich (1995) have also done work on learning to control an iterative improvement algorithm, although their method is significantly more complicated.

3.7.3 Decision-theoretic Search

The DTS system of Othar and Hansson (1994) uses learning during search to help allocate effort. Their method learns a function from the value of a heuristic function at a node to the node's probability of being a goal and the expected effort required to explore the node's subtree. It then explores nodes with the greatest expected payoff per unit of effort. In a similar vein, Bedrax-Weiss (1999) proposed weighted discrepancy search, which uses a training set of similar problems to estimate the probability that a node has a goal beneath it, and uses the distribution of these values to derive an optimal searching policy. Adaptive probing is less ambitious and merely estimates action costs rather than goal probability.

Horvitz et al. (2001) use runs on training data to learn a model of running time, and then use this model to derive a restart policy for a randomized backtracking search. This metareasoning approach focuses on the utility of search when trying to solve as many problems as possible from a set under time pressure. It could be used on top of the approach we pursue here, which focuses more on modeling the structure of the search space and using the model to guide search.

3.7.4 Reinforcement Learning

Although adaptive tree probing seems superficially like traditional reinforcement learning, since we are trying to find good actions to yield the best reward, important details differ. Here, we always start in the same state, choose several actions at once, and transition deterministically to a state we have probably never seen before to receive a reward. Rather than learning about sequences of actions through multiple states, our emphasis is on representing the possible action sets compactly to facilitate generalization about the reward of various sets of actions. We assume independence of actions, which collapses the breadth of the tree, and additivity of action costs, which allows learning from leaves. Since each state corresponds to a sequence of actions, we are generalizing over both actions and states.

3.8 Limitations

Adaptive probing is incomplete—even for a small tree, there is no guarantee that all leaves will be visited within a bounded amount of time. And if the entire tree happens to be explored by sheer good luck, there is no way to recognize that happy event. Its stochastic, unsystematic probing means that there is no compact representation for recording the portion of the search space that has been visited—we would have to store all paths explored. Even if we collapse identical prefixes, this could mean storing most of the search tree, thereby using space exponential in the problem size.[1] More generally, it seems as if the ability to immediately alter the search in reaction to new information must preclude an efficient way to completely enumerate the space, as each new piece of information might suggest a new and unrelated direction for subsequent exploration. We will explore this issue further in the next chapter.

Not only is adaptive probing incomplete, but it has no mechanism to promote exploration of the search space. The emphasis of adaptive probing is on narrowing the sampling to the region predicted to contain the best solution. Only the ad hoc limit on the probability that the best action will be selected prevents the algorithm from converging onto the single path that it believes is best. By exclusive emphasis on narrowing the search, complete coverage of the search space is neglected. The algorithm never really tries to explore, but rather merely tries to avoid premature exploitation.

[1]The worst case occurs when, for all nodes whose children are leaves, all but one of the children has been expanded. If all children of a node were expanded, we could mark that node as completely explored.

It should be possible to mitigate this convergence problem by adding a second phase to the algorithm. During the first phase, the algorithm attempts to narrow the search down to a single path. Then, after the probability of the predicted best path reaches a certain threshold, the maximum probability of selecting the best action could be gradually lowered, causing the algorithm to explore paths similar to the predicted best path. Although the algorithm would still be incomplete, its focus of attention would gradually become more diffuse and it would eventually explore the entire search space with high probability.

3.9 Other Possible Extensions

It may be worthwhile to distribute variance unequally among depths. Because the effects of variance sum across levels, it may be possible to simply use a second on-line regression to divide up the mean error. Additional features besides depth might be helpful, perhaps characterizing the path taken so far. This might take the form of conditioning on the previous action.

Adaptive probing can be used for goal search, as we saw with boolean satisfiability, as long as a maximum depth and a measure of progress are available. If a measure of leaf quality is not available, it may be possible to fit the model using many small instances of similar problems (or small versions of the current problem) that can be quickly solved and then to scale up the model to guide probing on the original problem.

In this thesis, we are focusing on searching a tree whose structure has been assumed. In other words, we have been learning when to disregard the heuristic that choose the value for the current variable at each decision node. But the adaptive search paradigm is quite general and could equally well apply to the variable-choice heuristic. In fact, some preliminary experiments on number partitioning have shown that for incomplete search it can sometimes be more useful to search over the choice of variable than the choice of value (Ruml, 2001b). One can even conceive of doubling the depth of the tree, first branching on which variable to choose and then branching on the value to give it. Although this makes the tree very large, an adaptive search algorithm that can quickly discover where it is most useful to branch might be able to take advantage of the additional flexibility to find better solutions.

For some domains, multiple value-ordering heuristics are available and the best one to use depends on the problem. Lagoudakis and Littman (2001) have done work on using training experience to generate a policy off-line for selecting the heuristic based on features of the problem or subproblem under consideration. By using the same techniques as employed in adaptive probing, one could avoid the training phase and learn to select the appropriate heuristic on-line. The cost of a leaf would be its depth and the algorithm would try to learn which heuristics resulted in small subtrees. This on-line approach would also eliminate the need to characterize a suitable class of training instances

3.10 Conclusions

It is a widely held intuition that tree search is only appropriate for complete searches, while local improvement search dominates in hard or poorly understood domains. We have seen how a simple adaptive probing technique can overcome the strong assumptions that are built into traditional systematic tree search procedures. By learning a model of the tree on-line and simultaneously using it to guide search, we have seen how incomplete heuristic search can be effective in a tree-structured search space. When the node-ordering heuristic is very accurate, a suitable learning bias is necessary for acceptable performance in practice. But for problems with unknown character or domains that are less well-understood, the robustness of an adaptive approach makes it superior. Its flexibility raises the possibility that, for difficult and messy problems, incomplete tree search may even be a viable alternative to local improvement algorithms.

In the next chapter, we will extend this initial exploration of adaptive tree search by investigating a remedy to adaptive probing's main limitation: its incompleteness.

Chapter 4

Best-Leaf-First Search

> We introduce the *best-leaf-first search* framework for complete adaptive tree search. The framework relies on a model of leaf costs and it structures the search to visit leaves in order of increasing predicted cost. Later chapters will instantiate the framework using different tree models to derive specialized search algorithms.

In the previous chapter, we saw that an effective model of the distribution of leaf costs could be learned and exploited for search. However, adaptive probing was incomplete. Although this is often not an issue for very large problems, for smaller problems it can be important to guarantee that the best possible solution has been found. In the case of constraint satisfaction problems, for instance, it is often helpful to know that no feasible solution exists. Furthermore, as we will see, structuring the search to achieve completeness can simultaneously provide a principled way to order exploration of the search space.

In this chapter, we will see how a simple and only mildly restrictive framework can be used to structure an adaptive search while achieving completeness. As with adaptive probing, we will encapsulate the algorithm's information about the search tree into a predictive model of leaf costs. The key idea is to visit leaves in a rational order according to the model. Thus we visit first those leaves that, based on the currently available information, are predicted to have lower cost. For this reason, the framework is called best-leaf-first search (BLFS). The general BLFS scheme depends crucially on the model that is used. After discussing the framework itself in this chapter, the following two chapters will explore two particular instantiations of the framework using two different tree models.

4.1 The BLFS Framework

Recall that our goal is to visit the best leaf in the tree. When armed with only heuristic information, the best we can do is to visit leaves in increasing order of their predicted cost. Even if we assume that an accurate model of leaf costs is provided before the search, this search order is difficult to accomplish exactly. Given an expressive model, such as the separate action cost model used in Chapter 3, it is very difficult to enumerate leaf paths in increasing order.[1] Any practical strategy must avoid maintaining a potentially enormous list of backtrack points. We must also be careful to limit the time we spend regenerating portions of the tree we have already seen.

A straightforward solution to all of these difficulties is to merely approximate increasing order by adopting a progressively expanding search horizon. At each iteration, we will visit all leaves whose cost is predicted to fall below some upper bound. At first, we will set the bound very low, so that only leaves that are predicted to have very low cost will be visited. By successively raising the bound, we will visit more and more leaves of progressively increasing predicted cost. Unlike discrepancy search, we cannot increment to the next

[1]Given a set A of n positive integers, create a separate cost model with n levels and two children per level in which all preferred children cost 0 and the other children are assigned unique values from A. If we can answer the question of which path leads to the cheapest leaf whose cost is greater than some positive integer B, then we can solve the subset sum problem, which is NP-complete (Garey and Johnson, 1991, problem SP13), by setting B to be one less than the sum we seek.

```
BLFS(root)
Visit a few leaves
Nodes-desired ← number of nodes visited so far
Loop until time runs out:
    Double nodes-desired
    Estimate cost bound that visits nodes-desired nodes
    Call BLFS-expand(root, bound)
    If entire tree was searched, return

BLFS-expand(node, bound)
If is-leaf(node)
    Visit(node)
else
    For each child of node:
        If best-completion(child) ≤ bound
            BLFS-expand(child, bound)
```

Figure 4.1: Simplified pseudo-code for best-leaf-first search.

integer because child costs may vary widely. If the bound is too small to afford many new children, we risk visiting few new leaves while incurring the overhead of regenerating all nodes visited during the previous pass. But as the increment grows, we lose the desired search order. During a pass, we will visit all affordable leaves in depth-first order, even though we would prefer to visit the most expensive of the affordable leaves last.

Zilberstein, Charpillet, and Chassaing (1999) have shown that doubling the length of successive iterations is the optimal schedule when no information is available about the deadline. If we can arrange the bound such that we visit twice as many nodes at each iteration as were visited during the previous iteration, then the overall overhead of the algorithm will be bounded by 3 and we will visit leaves in an approximation of the optimal order.[2] So we will tracking the number of nodes expanded on every iteration and then

[2] Let the number of nodes visited in the last iteration be n. By definition, this is equal to the number of nodes in the entire tree. In the worst case, the number of nodes visited on the previous iteration of the doubling scheme will be $n-1$. The number of nodes visited in all prior iterations will be $(n-1)/2 + (n-1)/4 + ...$ which sums to about $n-1$. The number of nodes visited by the doubling scheme is the number

estimate the bound needed to visit twice as many during the next. Pseudo-code for the BLFS framework is shown in Figure 4.1.

Because each iteration of BLFS is a depth-first search, BLFS is compatible with all of the sophisticated backtracking enhancements developed to improve depth-first search. These include not only branch-and-bound for combinatorial optimization, but also techniques for constraint satisfaction problems such as backjumping and dynamic backtracking. The cost model just enforces additional pruning.

The predictive model of leaf costs is used in two places: to decide whether to descend into a subtree in step 12, and to estimate the next cost bound in step 5. In the next section, we will explore the role of the model in more detail.

4.2 The Tree Model

BLFS requires that its model of leaf costs support the following two operations:

compute $f(n)$: In order to visit only subtrees containing leaves whose costs fall under the current iteration's bound, we need an estimate of the cost of the best leaf under a given branching node. In other words, we need an estimate of the cost of the best completion of the partial solution represented by that node. We will notate this estimate as $f(n)$ in analogy to terminology used in shortest-path algorithms, as will be explained later in Section 4.4.

estimate cost bound that yields a given number of nodes: In order to limit the overhead of BLFS, the number of nodes must grow exponentially across iterations. The

visited on the last iteration, n, plus the number visited on the previous iteration, $n-1$, plus the number visited on all prior iterations, about $n-1$. So the total number of nodes visited in the worst case is about $3n$.

computation of the cost bound need not be exact, but should preserve the general exponential growth of the search.

In later chapters, we will see how these estimates can be efficiently computed using particular cost models, such as the separate action cost model used by adaptive probing in the previous chapter. But first we will delineate some desirable properties these two computations should have.

4.2.1 Properties of $f(n)$

The function $f(n)$ controls the expansion of nodes during the search (step 12 in Figure 4.1). It can be computed using any information available at n or along its path to the root. (Nodes below n may not have been generated yet so information from the subtree is inaccessible to f.) For every leaf node l, we will define $f(l)$ to be the predicted cost of the solution at that leaf. The most basic property that $f(n)$ should have for the rest of the tree is that it be non-decreasing: its value can become larger as we descend into a tree but not smaller. This is crucial for BLFS to be able to visit all leaves whose predicted cost is within the current cost bound. If the $f(n)$ value were to decrease along a path, then BLFS might prune away that path high in the tree even though deeper down it would have become clear that the subtree does contain a leaf whose cost is within the bound.

While a decreasing $f(n)$ gives incorrect behavior, an increasing $f(n)$ can result in inefficiency. If $f(n)$ increases along a path, then that path might be pruned away at a deeper level in the tree than necessary, possibly wasting time. However, the algorithm will still visit the correct leaves. In fact, no work will have been wasted as long as at least one of the siblings of the pruned node does lead to a leaf that is within the bound.

Putting these two desiderata together, we see that *consistency* is a fundamental property we would like $f(n)$ to obey. That is, every branching node should have at least one child whose f value is equal to that of the parent.[3] This ensures that a node is expanded if and only if it leads to a leaf whose predicted cost is within the cost bound. In other words, $f(n)$ should return the predicted cost of the best leaf below the given node n. Note that consistency is different from accuracy in the sense that the predicted leaf costs might not match the actual solution costs at all. BLFS explores in a manner that makes sense according to its model and the question of how well the model predicts solution quality in a specific tree is a separate issue. But given particular values for the leaves of the tree, consistency tells us what values $f(n)$ should return at the internal nodes.

One easy way to ensure that the estimate is consistent is to represent f as a linear function of separate costs for each action at each level. This is the type of action cost model we saw in Section 3.2 with adaptive probing. The estimated cost from the root to a given node is easy to compute as the sum of the costs of the actions taken, and the estimated best completion is just the sum of the best actions at the remaining levels.

4.2.2 Estimating the Cost Bound

The second operation that the tree model must support, in addition to $f(n)$, is the estimation of a cost bound that will yield a search generating the desired number of nodes. This estimate need only be accurate enough to yield the desired doubling behavior across iterations of search–it can be incorrect by any constant factor.

Even for a relatively simple cost model, this is not a simple task. For the implementations whose results are reported in this thesis, we will avoid this problem by reversing the estimation problem: instead of estimating the bound for a given number of nodes, we will

[3]This is stronger than Pearl's (1984) use of the term, which is equivalent to mere nondecreasing monotonicity.

use the model to estimate the number of nodes that would be visited for a given bound. We will then use a simple one-dimensional bisection-style search over bounds until we find an appropriate one.

This is likely to be more accurate than using analytic approximations to derive a cost bound. We are most interested in extremely small cost bounds, in the tail of the distribution of possible leaf costs. These extreme values are the least likely to be approximated accurately. For the cost models we will be considering, it is relatively straightforward to estimate the number of nodes for a given bound. And if we happen to know the approximate value of the desired bound, that value can be used as the starting point of the bisection search.

We have now seen the two operations that a tree model must support for use with the BLFS framework. First, a model should supply consistent $f(n)$ values representing the predicted cost of the best leaf below a given node. Second, it should supply a function for estimating the number of search nodes that would be generated when using a given cost bound. This estimation should efficiently mimic the BLFS search procedure, anticipating the f values that will be encountered and the pruning that will be done.

4.2.3 On-line Learning

BLFS separates the search framework from the tree model used to guide the search. One advantage of this decoupling is that the model need not be fixed in advance—it can be learned during the search. This interacts well with the iterative behavior of BLFS. The initial model at the beginning of the search process will not be very accurate. But because the early iterations are very short and because each iteration visits strictly more nodes than the previous one, the effects of the poor initial model are limited. As long as we visit enough new nodes to gather new information and refine the model, it will improve and the search will take advantage of these improvements.

One important implementation detail should be noted here: we can increase our chances of visiting exactly the desired number of nodes by based the pruning decisions during each iteration on a separate, static copy of the model that reflects the state it was in when it was used to estimate an appropriate cost bound to use, at the start of the iteration. This concession to efficiency means that the search will take advantage of new information only after the next iteration begins.

4.3 Rational Search

The term ‘rationality’ has been used in so many different ways and in so many different contexts that it is worth taking a moment to clarify in what respects BLFS is rational. BLFS is rational in the sense that, by visiting leaves in order of increasing predicted cost, it is maximizing the expected solution quality. Visiting any leaf other than the predicted best would, on the basis of the algorithm’s current information, result in finding a worse solution. One of our assumptions here is that the error of the tree model is the same for all leaves. More precisely, we assume that the expected bias in the tree model’s predictions is zero for all leaves. This is a benign assumption because if the model were systematically biased, it could be systematically corrected, leaving the resulting corrected model with only unsystematic unbiased errors. So we can say that BLFS is rational because it maximizes quality given its beliefs. More precisely, it is approximately rational, because it must approximate the optimal search order to maintain computational efficiency.

Of course, DFS is also rational in the sense of maximizing quality given current beliefs, because it could be viewed as maximizing quality in the context of fixed beliefs that happen to be implausible. But BLFS is rational in the additional sense that it can react to new information. Depending on the tree model in use, information observed during the search can change the current model in an arbitrarily large way, leading to adaptive behavior. No previous complete search algorithm has this property. We should note that BLFS itself only approximates the ideal of instant adaptation. Because the search is structured in iterations, each of which is guided with a static copy of the model from the previous iteration, there will usually be a lag between changes in the model and changes in the algorithm's behavior.

BLFS is not informed of its time deadline. If one knew that plenty of time remained, and if an estimate of the uncertainty of the cost model were available, one could implement a more sophisticated approach that might select seemingly poor actions in order to best reduce the uncertainty in the model and increase the chance of ultimately selecting the optimal leaf in the future. Such methods have been explored in the context of game tree search (Russell and Wefald, 1991) and reinforcement learning (Dearden, Friedman, and Russell, 1998) and would be an interesting avenue for future extensions of BLFS.

The idea of viewing a search algorithm as a rational agent has been proposed before. Russell and Wefald (1991) discuss work on metareasoning and search, that is to say, reasoning about which search actions (or other work) to perform. Metareasoning applies to situations in which time is a valuable resource, such as real-time applications with time penalties. When time can be valued on the same scale as solution quality, further search may not always be beneficial. Russell and Wefald, as well as Mayer (1994), use past experience to compute preferences for expanding nodes in a shortest-path algorithm. Hansson (1998) conducted a preliminary investigation into similar techniques for optimization trees. This work relies on information already gathered on a corpus of similar problems, rather than learned from the current tree.

In this thesis, we assume that further search is always beneficial, and we merely consider which searching actions might be most profitable to take. Deciding where to search and when to stop searching can be regarded as orthogonal issues. Decision-theoretic metareasoning methods could certainly be applied on top of BLFS to halt the search when the expected improvement is not worth additional time. The tree model should prove helpful in estimating the expected improvement. If even the smallest algorithmic actions, such as node expansion, are expensive, then it may become necessary to weight the benefit to be gained from visiting the leaf with the lowest predicted cost against the time that would be necessary to reach it. But in most domains, one has time to visit thousands of leaves, so we will leave aside issues of node expansion utility and focus on directing the search toward the most promising leaves.

4.4 Relations to Shortest-path Algorithms

The topic of adapting a search order in light of heuristic information uncovered during the search has been explored extensively in artificial intelligence in the context of shortest-path problems. As we recall from Section 1.1.2 (p. 6), the goal in a shortest-path problem is to find the shortest (or cheapest) path from a given initial state to any state that satisfies a given goal test function. These desired destination states are also called goal nodes. Tasks such as planning, puzzle-solving, and navigation can be cast as shortest-path problems in which one wishes to find a cheapest sequence of actions that transforms an initial situation into some desired goal situation.

Often, the general term "heuristic search" is equated with the specific framework of

```
IDA*-expand(node, bound)
If is-goal(node)
    Exit, returning node
else
    For each child of node:
        If f(child) ≤ bound
            IDA*-expand(child, bound)
```

Figure 4.2: Pseudo-code for the inner loop of iterative-deepening A* search (IDA*).

best-first search developed for shortest-path problems (Luger and Stubblefield, 1998, p. 124; Nilsson, 1998, p. 139; Poole, Mackworth, and Goebel, 1998, p. 132). The quintessential example of best-first search in every AI textbook is the exponential-space A* shortest-path algorithm (Hart, Nilsson, and Raphael, 1968). A related algorithm, iterative deepening A* (IDA*), approximates the behavior of A* while taking only linear space (Korf, 1985). As we will discuss below, these shortest-path algorithms are generally not appropriate for searching the bounded-depth trees arising in combinatorial optimization and constant satisfaction (Zhang and Korf, 1993). In many modern AI textbooks, techniques for optimization such as iterative broadening and node ordering for depth-first search are mentioned in the same chapter as heuristic shortest-path algorithms, but they appear together as a grab-bag of techniques for tree search, rather than as cousins related at a fundamental level. A comparison with BLFS exposes these similarities.

Pseudo-code for IDA* is shown in Figure 4.2. There are obvious similarities with BLFS, which was shown in Figure 4.1 on page 73. As in BLFS, IDA* proceeds in passes, each of which is a depth-first search that visits all nodes within a cost bound.[4] Table 4.1 summarizes a comparison of the two algorithms. Both control node expansion according to an evaluation

[4]Because iterative deepening proceeds by increasing a depth bound and BLFS proceeds by increasing a cost bound, a better name for BLFS might have been iterative worsening!

Table 4.1: A comparison of BLFS and IDA*.

	BLFS	IDA*
$f(n)$ semantics	best leaf below n	best path through n
desired $f(n)$ property	consistent	non-overestimating
$f(n)$ non-overestimating	correctness	optimality
$f(n)$ non-underestimating	efficiency	efficiency
$f(n)$ source	from user or learned	$= g(n) + h(n)$
$g(n)$ source	not necessary	from problem
$h(n)$ source	not necessary	from user
additive model	convenient	required
updating bound	estimation	add ϵ
	rational	optimal

function, notated $f(n)$. In BLFS, the $f(n)$ value of a node is computed as the predicted cost of the best leaf below it (i.e., the node's best completion). In IDA*, the $f(n)$ value is a prediction of the cost of the shortest path to the nearest goal that passes through that node. In this way, BLFS can be seen as an extension of IDA* to finding good leafs, rather than good paths.

This difference in the f function's semantics is reflected in how it is computed. In IDA*, the $f(n)$ prediction is decomposed into the path cost, notated $g(n)$, which measures the cost of the shortest path from the root to n, and a heuristic value, notated $h(n)$, which (under)estimates the cost of the shortest path from n to the goal. The h function is supplied by the user. This is a natural decomposition for shortest-path problems, but it does not apply directly to combinatorial optimization problems. BLFS considers $f(n)$ as the predicted cost of the best leaf in the subtree below n. BLFS uses its leaf cost model to estimate the entire $f(n)$ function directly. If one considers that model to be the heuristic information supplied by the user, then BLFS is using the user's heuristic information to estimate f, rather than just h, as is done in shortest-path heuristic search.

IDA* can be used for combinatorial optimization—in many domains, it is possible to construct an f function that yields a lower bound on the quality of any leaf below a given node. However, IDA* is a poor choice for two reasons. First, it visits too many internal nodes. Because most of the f values will be underestimates, many internal nodes will have f values lower than the value of the best leaf. IDA* visits all of these nodes before generating its first leaf. When solving a large problem or when operating under time constraints, this delay is unacceptable. The second problem with IDA* has to do with updating the cost bound. The algorithm updates its bound to the smallest f value that was seen on the previous iteration but that is greater than the current bound. When many nodes have the same f values, this can work well. But in an optimization problem, it is not uncommon for every node to have a slightly different f. This leads IDA* to increase its bound too cautiously, expanding only one new node on each iteration.

BLFS does not suffer those problems. BLFS uses an explicit representation of its predictive model, rather than relying on a black-box function suppled by the user. Being able to choose a simple explicit model leads to two important advantages over IDA*. First, we can choose a model that will give consistent predictions. (As we mentioned in Section 4.2.1, this means that we can ensure that for every node there exists a child whose best descendant will have the same evaluation.) This implies that BLFS is certain to reach leaves on every iteration. It will never expand a node unnecessarily and never overlook a node that has a descendant within the cost bound. To enforce the consistency of f, one can use leaf cost models which are represented as a linear function of action costs. This makes it fast and easy to accurately asses the best descendant's cost. One can view this model in terms of a prefix cost and the cost of the best completion, but unlike with IDA*, this separation into two components is just for convenience.

The second advantage that BLFS enjoys over IDA* is that the cost bound can be updated optimally. Because the predicted costs are generated by a known model, we can choose cost bounds that can be expected to cause twice as many nodes to be visited as on the previous iteration. By approximately doubling the number of nodes visited on each iteration, BLFS limits its overhead to a factor of less than three in the worst-case situation in which the entire tree must be searched. There is no reason that this technique could not also be used with conventional IDA*, although it might be more difficult to achieve acceptable accuracy than in BLFS because one might not have as much information about the internal structure of $g(n)$ and $h(n)$, as they are computed from the problem domain and black-box function supplied by the user rather than from a known model. Wah and Shang (1995) have explored the case in which the parametric form of the relation between cost bound and nodes generated is known beforehand.

These differences between BLFS and IDA* ultimately stem from the fact that BLFS can assume a bound on the depth of the tree. This allows the algorithm to pursue a single line of inquiry into the tree, easily reach a leaf, and potentially update its model. Shortest-path algorithms must be more conservative, pushing uniformly into the tree across its entire breadth. If one probed along a single path into a shortest-path tree, perhaps searching according to h, there is no guarantee that a goal or even a leaf would ever be reached. The problem would be worse if h were consistent, because it would be impossible to use a rise in h to terminate exploration. Using BLFS directly for shortest-path search would likely fail. Examining combinations of BLFS and IDA* or RBFS (Korf, 1993) for approximate shortest-path search would be an interesting direction for future work, however.

4.5 Conclusions

We have now seen a general framework for using heuristic information to search bounded-depth trees. BLFS can be seen as an extension of IDA* that uses an explicit model to ensure efficient search by both guaranteeing consistent node evaluations and allowing appropriate cost bound updating. By separating the search mechanism from the cost model, BLFS can learn on-line while suffering worst-case overhead of a factor of three when the entire tree must be enumerated. In the next two chapters, we will see two different tree models that illustrate how BLFS can be used in practice to solve combinatorial optimization and constraint satisfaction problems.

Chapter 5

BLFS with a Fixed Model: Indecision Search

We present an instantiation of the best-leaf-first search framework called *indecision search*. It uses a cost model based on preference information computed at branching nodes and it backtracks first to those nodes where the choice of best child was least clear. Empirical results show that it provides the best results known for several types of constraint satisfaction problems.

We will now instantiate the general BLFS framework introduced in the previous chapter with a particular cost model. The tree model that is used with BLFS determines how the cost estimates are calculated and how quickly BLFS will find the good leaves in the search tree. As a first test of the BLFS framework, we will investigate a simple model of leaf cost whose parameters do not need to be learned during the search. To adapt the search to the particular tree being explored, we will instead take advantage of the same heuristic scoring information that is used to rank children at each decision node. In the following chapter, we will investigate a tree model whose parameters are estimated on-line.

5.1 Two Tree Models

We will test two very similar tree models. As in the separate action model from Section 3.2, the cost of a leaf will be predicted to be a linear sum of costs for each child rank at each depth, but instead of learning the costs of these actions from the leaf costs, we will assume that the action costs are equal to scores that we will calculate for each child as we go, using a user-supplied node scoring function. Although this may seem odd, the motivation for this set-up derives directly from the idea of heuristic child ordering that we discussed briefly in Section 2.2.

Most heuristic node-ordering functions rank the children of a node based on a numerical score. For example, when solving a traveling salesman problem, one might order the children by the nearest city heuristic, which computes the distance to each unvisited city from the current location. When selecting which nodes to revisit during backtracking, any nodes at which the children all had the same score and were therefore ranked arbitrarily would seem to be much better candidates than those nodes at which the best child had a score that was much better than the score of the second-ranked child. Similarly, we might want to explore several similarly-valued children at one node before ever considering a relatively poor-scoring second child at another node. The spread of heuristic scores can be seen as giving us an indication of the degree of certainty or decisiveness of the heuristic. We can normalize the scores of siblings by subtracting the score of the best child, so that each child is given a value according to how much worse it is than the top-ranked child, according to this node-ordering heuristic.

If we model the cost of a leaf as the sum of these normalized scores, then leaves with low cost are those whose paths involved choices that were either exactly the preferred ones, or else closely ranked by the heuristic. The cost reflects the total amount of 'discrepancy' used at each branching node, and two paths that both have very low cost will differ only at branches where the children were very similarly scored. When using this cost function, BLFS will attempt to backtrack first to those nodes whose children were least differentiated by their heuristic values. In other words, we backtrack by revisiting nodes according to how decisive the heuristic function was in its ranking. Decisions about which the heuristic was less certain are revisited sooner than those involving a large difference in child score. In light of this, we can call this particular specialization of BLFS *indecision search*. It is an adaptive tree search because the backtracking points will be chosen based on the node scores encountered during the search. Unlike traditional systematic tree search algorithms, indecision search does not assume fixed child costs.

We will also test a second, simpler version of indecision search in which the leaf costs are predicted to be the maximum child cost along the path from the root, rather than the sum of all the costs. As we will see, this model turns out to be faster to estimate and, for the benchmarks we investigate, equally effective.

5.2 The Algorithm

Indecision search can be seen as a generalization of discrepancy search in which different non-preferred children have different costs, rather than all counting as one discrepancy. The preferred child costs nothing and the cost of any other child is its difference in heuristic score from the preferred child. If the child scores are $c_0, c_1, \ldots$, then a child i has cost $c_i - c_0$. We want to visit leaves in increasing order according to the sum of the cost along their path to the root. As in general with BLFS, it is difficult to know where these leaves are without

```
Indecision-expand (node, allowance)
If is-leaf(node)
    Visit(node)
else
    Update model of child costs
    Indecision-expand(child(node, 0), allowance)
    For i from 1 to number-of-children(node)−1
        c ← child(node, i)
        If cost(c) ≤ allowance
            Indecision-expand(c, allowance − cost(c))
```

Figure 5.1: Indecision search treats the BLFS cost bound as an allowance that is spent to visit non-preferred children.

exploring the entire tree, so we will use an iteratively increasing cost bound. Since the cost is just the sum of the scores at each level, we can visit only those nodes that are within the bound by considering the bound as a kind of allowance that is spent when we visit a node. Since the preferred child always costs nothing, we can be assured that an affordable child lies below any internal node that we can afford to reach. Pseudo-code for a pass of indecision search is presented in Figure 5.1.

Using the BLFS framework lets us avoid maintaining a potentially enormous list of backtrack points, and limits the time we spend regenerating portions of the tree we have already seen. In order to predict how many nodes will be visited within a certain cost bound, we will need to maintain a learned estimate of the expected distribution of child costs in the tree. After discussing these and other implementation issues, we will evaluate the performance of the resulting algorithm on two constraint satisfaction problems: latin square completion and binary CSPs. The results suggest that indecision search is more efficient than previous search algorithms and that it also avoids the poor worst-case performance of depth-first search. Indecision search is able to successfully exploit the information provided by the heuristic function that is ignored by previous algorithms.

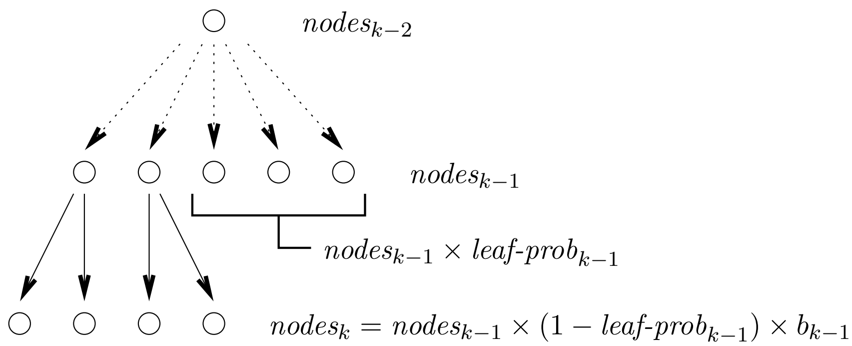

Figure 5.2: The process of estimating the number of nodes at the next level of the tree.

5.3 Estimating the Allowance

Estimating the proper value for the allowance is the most challenging part of the algorithm. Recall that we are finding a good allowance by a reverse process: predicting the number of nodes that will be generated using a given allowance and then searching over possible allowance values.

To predict the number of nodes for a given allowance, we will need to estimate the number of children we will be able to afford at each level, as well as the probability that a node at that level will be a leaf. Figure 5.2 illustrates this computation. If we can afford b_k children on average at level k with the given allowance and a node at that level is a leaf with probability $\textit{leaf-prob}_k$, then the number of nodes at a level, $nodes_k$, can be computed from the percentage of the nodes that are not leaves, times their fertility:

$$
\begin{aligned}
nodes_0 &= 1 \\
nodes_k &= nodes_{k-1} \times (1 - \textit{leaf-prob}_{k-1}) \times b_{k-1}
\end{aligned}
$$

The total number of nodes that we will visit can just be summed over the levels:

$$nodes = \sum_{k=0}^{max\text{-}depth} nodes_k$$

The $leaf\text{-}prob_k$ parameters can be easily estimated during the previous iteration. All that remains is to estimate the b_k for a given initial allowance. To do this, we will statistically model the flow and transformation of the allowance that would take place as the search procedure explored the tree. This requires that we have a model of the costs we would encounter in the tree. Happily, the information needed to construct this model is easy to acquire: we will have seen many of the relevant costs already. During the previous pass, we will note the costs of all the children of the nodes we visit, including the costs of the children we do not expand further. (The scores underlying these costs will have been computed by the heuristic already, in order to find the best child.) We use these costs to construct probability distributions, $p_{c,i}(x)$, over the possible costs x for the c-th ranked child at level i. (These distribution models will be implemented using histograms, as we discuss below.) For a tree of depth d and maximum branching factor b, we will need to form at most $d(b-1)$ distributions, one for each rank at each level, ignoring the preferred child which is always free. This cost model assumes that, for example, the fifth child of a node at level 23 will have the same cost distribution no matter how we reached that node, which is similar to the modeling assumptions that worked so well in Chapter 3.

We will use this cost model to decide how many children we can expect to afford at a given level and, at the same time, to compute a probability distribution over the possible amounts of allowance that will remain for use at the next level. This process is illustrated schematically in Figure 5.3. Starting with a spike distribution $p_{old}(x)$ that has all its mass

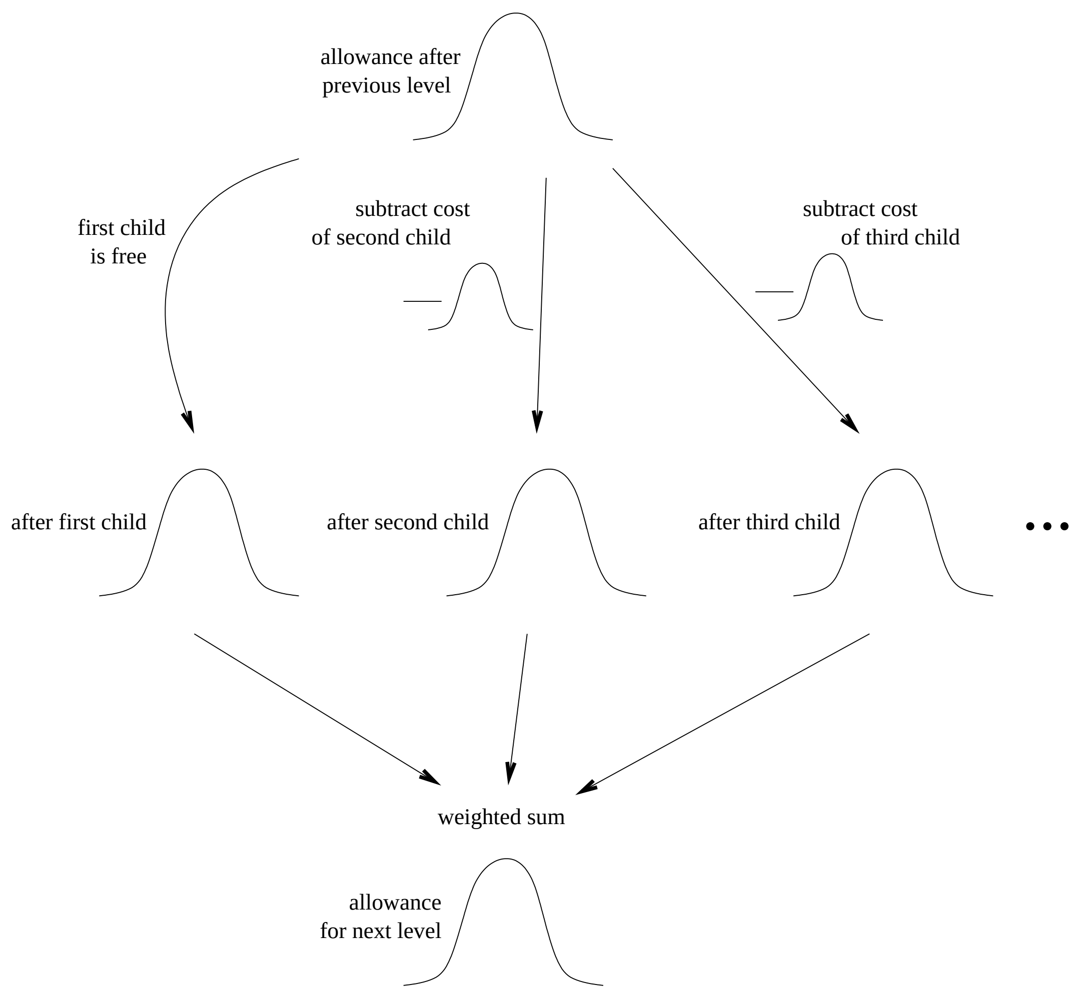

Figure 5.3: The process of estimating the allowance available at the next level.

at the given initial allowance value, we compute the distribution of allowance that is likely to be left after we visit a particular child. The probability of each possible new amount of allowance is just a sum over all the possible ways we could have arrived at that amount by spending allowance on that child, weighting each way by its probability of occurring:

$$p_{new}(x) = \begin{cases} \int (p_{old}(x+y) \times p_{child}(y))dy & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

We call this operation *truncating subtractive convolution*, since it computes the convolution of the two distributions while subtracting the cost from the allowance and squashing the probability of any negative allowance resulting from unaffordable children. The mass of probability that survives the convolution equals the probability that we can afford that child. The sum of the truncated distributions from all the children, plus a copy of the original allowance distribution to represent the free preferred child, forms the distribution of the allowance we can expect to have at the next level. (The sum can be performed over the raw truncated distributions, since we want to weight each one by the probability we can afford the corresponding child, but the resulting allowance distribution should be normalized to sum to one.)

We can now compute the expected number of children we can afford for each level, the b_k, from our estimated allowance distribution and our estimated child cost distributions. The expected number of children hinges on the probability that each possible number of children is the most we can afford, which we write as $p(\mathit{max\ is\ i})$:

$$p(\mathit{max\ is\ i}) \quad = \quad p(\mathit{can\ afford\ i}) \times (1 - p(\mathit{can\ afford\ i+1}))$$

$$b_k = \sum_{i=1}^{max\text{-}num\text{-}children_k} i \times p(max\ is\ i)$$

For a given allowance value and child cost distribution, $p(can\ afford\ i)$ is simply the amount of probability in the child cost distribution that is less than or equal to the allowance. By taking the expectation over the full allowance distribution, we can compute the overall expected b_k. With the b_k in hand, we have everything we need to compute an estimate of the number of leaves we would visit for the given starting allowance.[1]

Unfortunately, the clever trick introduced by Korf (1996) to improve limited discrepancy search does not apply to indecision search. Noting that every leaf from the previous pass was revisited during next pass of discrepancy search, Korf modified the algorithm to prune the preferred child when the discrepancy allowance is equal to the number of levels remaining in the tree. This forces the algorithm to only visit leaves with the desired number of discrepancies. In the case of indecision search, we cannot be sure of the costs of leaves below a given node because we only have a summary distribution. In applications with heuristic child scores whose range varies widely with depth, it might occasionally be possible to predict that all leaves beneath a given node are sufficiently good that they have been visited already. By using conservative estimates in such cases, it may be possible to allow some pruning. This could reduce the overhead of the algorithm, although the maximum benefit is a factor of two speed-up. In general, however, we cannot prune reliably and thus

[1]When recording child costs, we assumed they were independent and recorded them in separate distributions. But when calculating the number of children we can afford, we want to know the conditional probability that we can afford a child *given that we could afford the previous children*. To calculate this properly, we should instead have stored the difference in cost between each child and the previous one, and then derived the allowance remaining after each child using successive subtractive convolutions. Empirically, the two strategies seem to perform similarly (normalizing the non-conditional probabilities of affording each number of children to sum to one). However, the correct incremental approach does not allow a fast implementation of the simplified search strategy described later, so we have presented the version that assumes independence instead.

we must revisit previously seen leaves.

The most complicated part of indecision search is the estimation of the allowance percolating down the levels of the tree. This can be completely avoided if we use the second, simpler indecision search cost model. Instead of modeling a leaf's cost as the sum of the child costs along its path, we can assume that the cost will just be proportional to the maximum child cost along the path. This means that we can modify the search algorithm to visit all children within the same allowance at every level, passing the allowance down unmodified. The same amount of allowance will be available at every level and subtractive convolution is unnecessary. This simple indecision search variant can be viewed as a generalization of iterative broadening, just as indecision search is a generalization of limited discrepancy search. Simple indecision search only needs to estimate the distributions of the child scores and then, when estimating the b_k, find the mass in each distribution lying at or below a given allowance threshold.

5.4 Implementation

Although the basic ideas of indecision search are straightforward, any implementation must confront details such as data structures for probability distributions and search algorithms for cost bounds. We briefly sketch the methods used in the implementation whose results are reported below.

5.4.1 Manipulating Distributions

To implement indecision search, we must learn and convolve distributions. Rather than assuming that they follow a particular parametric form (such as a truncated normal distribution) and attempting to derive closed-form solutions for subtractive convolution, we just use histograms. The bin locations are determined adaptively by the data. As new samples are added to an empty histogram, we record them individually until a fixed size limit is reached (100 in the experiments reported below). At this point, each sample expands to become a bin, reaching halfway to its nearest neighbors. Samples on the ends are expanded symmetrically. When further samples are added, the weights of the appropriate bins are increased. Because we must locate the appropriate bin, the time to add a sample is logarithmic in the size limit of the histogram. We track the largest weight in any bin and when this becomes greater than twice the sum of the weights in any adjacent pair of bins, we split the heaviest bin and merge the smallest adjacent pair. We only need to find the sum of the weights in the smallest adjacent pair when a bin's weight becomes larger than the weight of the heaviest bin.

Subtractive convolution is straightforward, although four cases must be considered depending on whether the distributions involved are represented as individual points or have expanded into bins. When both are points, we can just construct a new histogram and add the appropriately weighted samples to it. This is done in a random order to preserve accuracy if the new histogram converts to bins. When both are bins, we reduce to the points case by treating the bin centers as samples. When one is bins and the other is points, we construct a new truncated bins histogram for each point, incrementally accumulating them into the result. When bin boundaries do not coincide, bins are split. As this may result in too many bins in the result, we then collapse the smallest adjacent pairs as necessary. (This can be made efficient by storing adjacent pair weights in a heap and updating after every merge.) The cost of convolution is bounded by the square of the histogram size limit.

Similarly, four cases must be considered when adding distributions together to computing a new allowance distribution. Adding points to points and bins to bins works as for convolution. When adding bins and points, the points are treated as new samples for the binned distribution unless the total weight of the points is greater than the weight of the bins, in which case the bins are converted to points at their centers and the two point collections are added. This preserves the accuracy of the distribution with the greatest mass. When adding points to bins, we also check to be sure that the histogram has the maximum number of bins, as truncating convolution may have reduced the number of bins. If there is room for an extra bin, we split the bin to which the point is added.

5.4.2 Finding an Appropriate Allowance

Now that we can manipulate cost distributions, we can use them to predict the number of leaves we would see for a given allowance. To find an appropriate allowance given our model of the tree, we use a simple binary search. The allowance for the first pass is always zero. This causes the search to explore all ties after visiting the greedy child. For later iterations, we choose a starting point by increasing the previously used allowance by 20% (or arbitrarily using 1 if the previous iteration was the first). The allowance is increased until more leaves are predicted than we desire to visit. The resulting interval around the correct allowance is then reduced by half until the prediction is sufficiently close (or until seven splits have occurred). Any prediction within 5% of the desired value or greater by less than 50% is deemed sufficient. If we attempt to predict the number of leaves that would be seen with an allowance that is greater than the sum of the largest costs at every level, we recognize that the desired number of nodes may be larger than the size of the current tree and we simply return the attempted allowance immediately. To cope with inaccurate estimates that result in few new leaves being seen, the number of desired leaves is twice the number of leaves seen on the previous pass or twice the number we had wanted to see, whichever is greater.

Since finding the next allowance forms most of the overhead of indecision search, an optimized implementation would retain information across iterations and use interpolation and extrapolation to guess good allowance values and make maximum use of the expensive estimates. Estimates need only be accurate within a constant factor, since the desired number of leaves increases multiplicatively. During early iterations, many histograms will contain exact samples and convolution will be quite accurate.

5.5 Evaluation

We have seen how the underlying idea of backtracking to points of indecision can be turned into a practical algorithm using the BLFS framework. Now we must verify that it is effective in practice and test whether it provides any advantage over discrepancy search algorithms that simply assume fixed child costs. The most obvious candidates for indecision search are constraint satisfaction problems (CSPs), as they are commonly solved using quantitative heuristic node scoring functions. We will evaluate the algorithm's performance on two types of problems: latin squares and binary CSPs. (Additional results on the combinatorial optimization problem of number partitioning will be discussed in Section 6.3.)

5.5.1 Latin Squares

A latin square is an n by n array in which each cell has one of n colors. Each row and each column must contain each color exactly once. Although constructing a latin square from scratch is not difficult, completing a partially-filled latin square can be difficult or impossible. Gomes and Selman (1997) proposed latin square completion as a challenging benchmark problem for constraint satisfaction techniques. They note that, like many real-

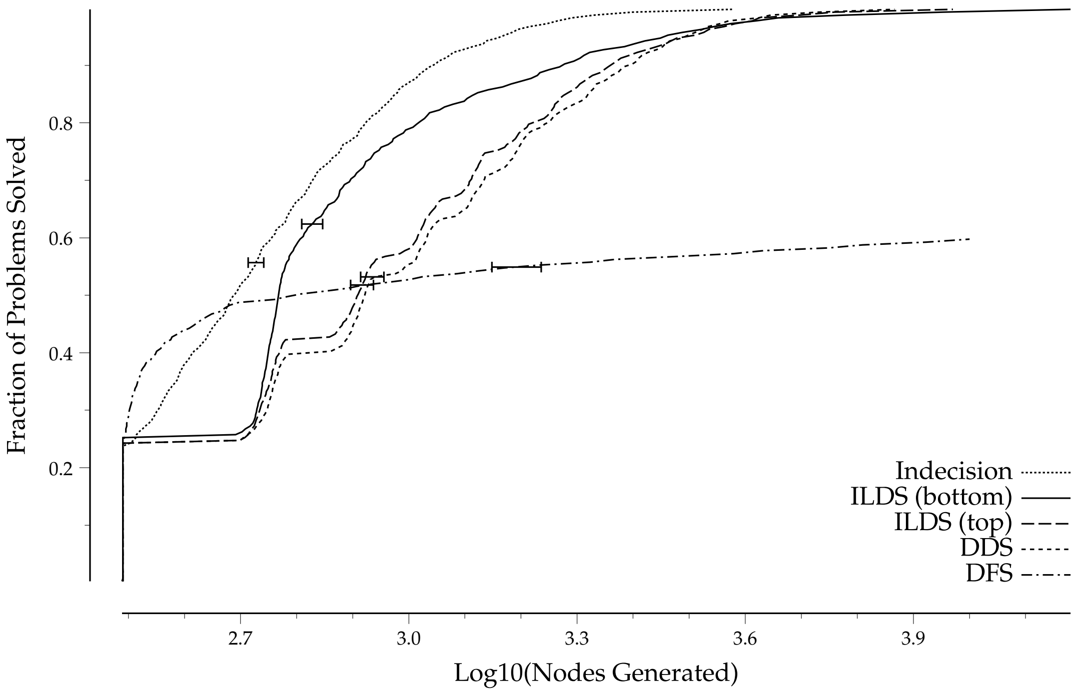

Figure 5.4: Performance on completing 21×21 latin squares that already have 30% of the cells assigned.

world problems, it exhibits both regular structure, due to the row and column constraints, and random elements, due to the preassigned cells.

We used a forward-checking algorithm, choosing variables to assign according to the classic most-constrained variable heuristic of Brélaz (1979). Values were ordered according to the promise heuristic of Geelen (1992), which estimates the number of solutions below each child. For indecision search, the logarithm of the promise was used as the heuristic score of a node. Following Meseguer and Walsh (1998), we used a test set of 1,000 latin squares, each with 30% of the cells assigned, filtering out any unsatisfiable problems. We tested depth-first search (DFS), two version of Korf's improved limited discrepancy search (ILDS), one taking discrepancies at the top first and the other taking them at the bottom first, depth-bounded discrepancy search (DDS), and the plain and simplified versions of indecision search.

The performance of the algorithms is shown in Figure 5.4 in terms of the fraction of problems solved within a given number of node generations. Small horizontal error bars mark 95% confidence intervals around the means. Depth-first search was limited to 10,000 nodes per problem, hence its mean is a lower bound. (In fact, Gomes et al. (2000) have noted that the cost distribution of depth-first search on this problem is heavy-tailed and seems to have essentially infinite mean!)

From the figure, we see that 25% of the problems were solved by visiting a single leaf (the greedy solution). Depth-first search enumerates leaves very efficiently, but soon becomes mired at the bottom of the tree. The discrepancy search algorithms immediately retreat to the root and must visit many nodes before reaching another leaf. Indecision search first explores all ties, which may occur at intermediate levels of tree. As the search progresses, the algorithms biased toward discrepancies at the top seem to be paying a price, as their progress comes in spurts. Indecision search makes more efficient use of its time, exhibiting a smooth performance profile, and it solves all the problems within 4,000 nodes. The simple variant seemed to perform identically to the plain version shown in the figure. Variants that took the most expensive affordable child first (thus branching at the top first) also performed similarly.

To test how important the later stages of indecision search are, we tested a hybrid algorithm that first visits all leaves of cost zero (as in indecision search) and then carries out the same ordering as ILDS, oblivious to the heuristic function. This hybrid algorithm exhibits behavior similar to indecision search for the easiest 70% of problems, but then exhibits the same longer tail as ILDS, taking over 20,000 nodes to solve the last problem. This demonstrates that the later stages of indecision search are an important component of its robustness and that it is not simply exploiting the leaves tied for zero cost.

Table 5.1: The number of nodes generated to solve latin square completion problems, represented by the 95th percentile of the distribution across random instances.

n	DFS	Indec.	ILDS	DDS	Indec. / ILDS
11	7,225	**173**	183	206	.945
13	888,909	**284**	303	357	.937
15	∞	**427**	621	642	.688
17	∞	**621**	1,047	1,176	.593
19	∞	**871**	1,609	1,852	.541
21	∞	**1,339**	2,812	3,077	.476

Similar behavior was observed on smaller instances, although the advantage of indecision search over the discrepancy methods seemed to increase as problems grew larger. Table 5.1 summarizes these experiments by listing the 95th percentile of the distribution of nodes generated by each algorithm. The rightmost column compares the performance of indecision search and its nearest competitor, ILDS. (The superior variant of ILDS, that takes its discrepancies at the bottom of tree first, is the one shown here.)

5.5.2 Binary CSPs

Binary CSPs are those in which all constraints refer to only two variables. This is a canonical form for constraint satisfaction problems, as any problem can be made binary by introducing additional variables. Binary CSPs have received much attention in the literature because it is relatively easy to generate synthetic instances with known properties, allowing researchers to test how algorithmic performance varies with different features of the problems.

Meseguer and Walsh (1998) used binary CSPs to evaluate depth-bounded discrepancy search and interleaved depth-first search, testing on satisfiable problems of the $\langle n, m, p_1, p_2 \rangle$ type. These problems have n variables, each with m possible values. Exactly $p_1 n(n-1)/2$ of the possible pairs of variables are constrained and exactly $p_2 m^2$ of the possible value

Table 5.2: The number of nodes generated to solve 100 instances of binary CSPs in the $\langle 30, 15, .4, p_2 \rangle$ class.

p_2	quantile	DFS	S. Indec.	Indec.	ILDS	DDS
.307	50%	**40**	99	102	60	96
	95%	**241**	391	396	456	424
.320	50%	**100**	258	272	288	282
	95%	1,119	**884**	1,094	1,122	1,115
.333	50%	**520**	878	912	933	2,044
	95%	4,881	**4,501**	4,991	5,862	8,014
.347	50%	**3,187**	4,705	5,511	6,191	16,305
	95%	42,025	**28,294**	33,155	30,996	100,387
.360	50%	**24,214**	49,672	33,324	38,108	141,290
	95%	**103,878**	536,716	612,628	309,848	1,642,806

Table 5.3: The number of nodes generated to solve 100 instances of binary CSPs in the $\langle 50, 12, .2, p_2 \rangle$ class.

p_2	quantile	DFS	S. Indec.	Indec.	ILDS	DDS
.306	50%	**52**	108	118	90	137
	95%	**164**	320	296	358	408
.319	50%	**63**	188	204	237	373
	95%	1,450	**984**	1,098	1,271	1,301
.333	50%	**250**	785	900	1,277	2,478
	95%	**3,156**	3,410	3,942	6,389	12,790
.347	50%	**1,646**	4,173	5,099	4,663	26,277
	95%	**22,852**	28,630	49,051	52,491	187,856
.361	50%	**27,953**	40,454	52,994	83,980	372,064
	95%	**352,788**	387,432	463,774	554,036	3,546,588

combinations are disallowed for each of those pairs. As p_2 increases from 0.25 toward 0.36, the constraints become tighter and the problems become more difficult to solve, exposing differences in performance between the algorithms. We will use the same heuristics we employed above with latin squares.

As with latin squares, there is enormous variance in the number of nodes generated by each algorithm within each set of 100 similar instances. We focus on the upper tail of the distribution because it essentially controls the expected value. We avoid the maximum,

Table 5.4: The number of nodes generated to solve 100 instances of binary CSPs in the $\langle 100, 6, .06, p_2 \rangle$ class.

p_2	quantile	DFS	S. Indec.	Indec.	ILDS	DDS
.306	50%	**102**	146	158	147	152
	95%	**110**	676	858	646	826
.333	50%	**110**	336	514	770	1,490
	95%	31,910	**3,344**	3,527	4,012	11,845
.361	50%	**3,432**	5,896	9,810	19,454	125,488
	95%	208,112	70,664	**62,118**	127,712	2,048,320

as it is subject to sampling error. Tables 5.2 through 5.4 show both the median and the 95th percentile of each distribution. The results obtained for DFS, ILDS, and DDS were consistent with those reported by Meseguer and Walsh, although the algorithms seem to visit slightly more nodes, presumably because our heuristic is less accurate than the one they use (Larrosa and Meseguer, 1995). At very low tightness, problems are easy and DFS is sufficient. DFS always exhibits the best median performance, but as tightness increases the tail of its distribution grows rapidly. Indecision search is either the best or within 25% of the best in every instance class except $\langle 30, 14, .4, .360 \rangle$.[2] DDS seems to fare poorly. Indecision search performs better on these problems than the discrepancy search algorithms. Although its median is not as low as DFS's, it is more robust and tends to have a lower maximum search cost. Overall, the simpler variant seems to perform better than the plain. This is presumably due to its different modeling assumption, in which the cost of a leaf is the maximum cost of any child along its path rather than the sum of all children along the path.

[2]This single poor performance seems to be due to inaccuracies when updating the cost bound—rather than visiting twice as many nodes with each iteration, indecision search visits only a constant number more. In practice, a simple mechanism to detect and correct such systematic mispredictions should be easy to implement.

5.5.3 Time Overhead

Asymptotically, indecision search does not increase the complexity of a search. The additional computation comes in two places: during the search and when updating the cost bound. During the search, two computations must be done. The first is to calculate the $f(n)$ values for a node's children. This is only a small constant number of additional instructions per child. The second operation at each node is to store the observed cost values. For each observed value, this takes time logarithmic in the histogram size (for locating the correct bucket to increment) and so is also constant as problem size increases.

Estimating the next cost bound is more difficult. Although histogram computations are bounded by the square of the histogram size, the remaining allowance must be estimated at each level of the tree, introducing a dependence on problem size. However, this is only a linear dependence (in each of the maximum branching factor and number of variables), and starting the search for a cost bound at a good initial value can reduce the number of estimations required. The search among cost bounds can also be limited to a small constant number of iterations.

An empirical analysis of an optimized implementation in terms of running time would be useful. The main overhead seems to be convolution during leaf estimation, in particular the adding together of the many slightly modified copies of the allowance distribution. On the problem sizes investigated here, plain indecision search currently seems to take longer than limited discrepancy search, although simple indecision search has almost no overhead and runs very quickly.

In addition to optimizing the implementation, it would also be interesting to investigate the sensitivity of the algorithms to accuracy parameters such as the maximum histogram size. Informal experiments on small generic trees indicate that 50 bins give performance equal to 100, which should allow a factor of four speed-up during bound estimation.

5.6 Related Work

Bedrax-Weiss (1999) has proposed an algorithm called weighted discrepancy search, based on similar motivations. Her method is more ambitious and involves estimating the probability that a child's subtree contains an optimal solution. (This requires training data from previous similar problems.) Rather than using the raw heuristic scores, weighted discrepancy search uses probabilities, which can presumably be combined more rationally across levels. The algorithm assumes lognormal distributions of probabilities and attempts to precompute a schedule of probability thresholds that will maximize the probability of finding a goal given a particular time limit (and certain assumptions). Indecision search just uses the difference in heuristic scores, assuming that they are comparable across levels, and attempts to find good allowance values on-line. It would be very interesting to combine the probabilistic framework of weighted discrepancy search with the on-line and non-parametric modeling approach of indecision search. The work of Hansson and Mayer (1994) on learning relations between heuristic scores and search costs may also be applicable. Accumulating enough training data to support accurate probability estimates would appear to be the main hurdle.

Along a similar vein, Ruml, Ginsburg, and Shieber (1999) used training examples to estimate the probability that a subtree (characterized by real-valued features) contains a solution, and then used that data to prune a tree search. In indecision search, the heuristic directly provides the information necessary for backtracking, so no training problems are needed.

Gomes, Selman, and Kautz (1998) and Gomes et al. (2000) have suggested a randomized restarting policy to avoid the poor performance of depth-first search. By randomly reordering children that have very similar heuristic scores, they produce different search trees on different runs. By frequently restarting the search from the beginning, they explore the closely ranked children. (A similar strategy is used in the GRASP procedure (Feo and Resende, 1995).) One can view this technique as an *ad hoc* approximation to indecision search. One would expect indecision search to perform better, as it does not throw away information regarding close-scoring children. It also does not require manual tuning to set a threshold value.

5.7 Possible Extensions

This work can be extended in a number of ways. While we have demonstrated indecision search on constraint satisfaction problems, it should apply naturally to any tree-structured search space that uses a quantitative heuristic. Many combinatorial optimization applications meet these criteria.

More generally, if the number of nodes one will have time to see is given ahead of time, it might be possible to set the allowance early on in the search to yield exactly the number of nodes we have time for. This would avoid regenerating portions of the tree, for an expected speed-up of a factor of two.

5.8 Conclusions

We have introduced a new backtracking algorithm, indecision search, that attempts to revisit first those nodes where the child-ordering heuristic function was least certain of its ranking. The algorithm can be seen as a generalization of limited discrepancy search and a simpler variant of it generalizes iterative broadening. Empirical results on the standard benchmark problems of latin square completion and binary CSPs suggest that indecision search visits fewer nodes than limited discrepancy search and depth-bounded discrepancy search and has more robust worst-case performance than depth-first search. It successfully adapts its behavior to the tree it finds itself in, taking advantage of information in the heuristic values that other algorithms ignore.

Indecision search, while it will repeatedly regenerate nodes, is guaranteed to eventually traverse the entire tree if necessary. (This is because the allowance must increase on every pass.) However, the adaptive probing technique discussed in Chapter 3 was able to learn its costs directly from an objective function on the leaves and did not require a quantitative heuristic function. Although it was incomplete, it could be applied to any bounded-depth tree search problem. In the next chapter, we will see how BLFS can combine the best features of these two methods in a single algorithm.

Chapter 6

BLFS with On-line Learning

We present an instantiation of the best-leaf-first search framework in which the cost model is learned on-line during the search. Using the separate action cost model, this provides a complete and deterministic analogue of the adaptive probing algorithm of Chapter 3. Empirical results show that this algorithm is the best method yet devised for the combinatorial optimization problem of number partitioning.

Adaptive probing buys its flexibility at the price of completeness. But the action cost model that it learns can be considered a form of child preference information, of the kind used by indecision search. The learning of actions costs performed by additive probing can be done during each pass of an indecision search and these costs can be used to guide the search instead of child scores. The result is a version of BLFS that represents a deterministic and complete analogue of adaptive probing. This is a more complex realization of BLFS than indecision search was, because we are now learning the parameters of the leaf cost estimating function during the search itself. (In indecision search, we merely stipulated *a priori* that the cost of a leaf was proportional to the sum of the normalized child scores, rather than grounding the estimates in actual observed leaf values.)

```
BLFS(root)
1  Visit a few leaves
2  Initialize the model
3  Nodes-desired ← number of nodes visited so far
4  Loop until time runs out:
5      Double nodes-desired
6      Estimate cost bound that visits nodes-desired nodes
7      Make static copy of model
8      BLFS-expand(root, bound)
9      If entire tree was searched, return

BLFS-expand(node, bound)
10 If is-leaf(node)
11     Visit(node), updating model with leaf cost
12 else
13     For each child of node:
14         If best-completion(child) ≤ bound
15             BLFS-expand(child, bound)
```

Figure 6.1: Simplified pseudo-code for best-leaf-first search using on-line learning.

6.1 The Tree Model

We will first consider the same type of separate action cost model that we used with adaptive probing, in which each child rank at a particular depth is assumed to correspond to the same cost, and we learn the cost of each rank at each depth, for db parameters overall. (Figure 3.2 on page 30 showed an example.)

Figure 6.1 sketches the pseudo code of BLFS, with additional steps in lines 2, 7, and 11 to account for on-line learning. Recall from our previous discussion of BLFS (Section 4.1) that there are two main operations that must be supported by the tree model:

compute $f(n)$: Given estimated action costs, predict the cost of the best leaf below a node.

This is straightforward for the separate action cost model. As the search descends the tree, one can accumulate the cost of the actions chosen so far. A table of the costs of the best possible completions from each level can be precomputed before each BLFS iteration. Starting from the bottom of the tree, one selects the lowest cost action at each level, accumulating the cost while working back up the tree. This yields, for a node at any depth, the cost of the best possible subsequent sequence of actions. The f value of a child node is then just the sum of the actions taken to the parent plus the cost of the action associated with the child's rank plus the best completion from the next level of the tree.

estimate number of nodes within a cost bound: This estimation is easier than in indecision search because the costs of the children at each level do not depend on observed heuristic scores but are instead fully known and given by the model. To estimate the number of children we will take at nodes at each level, we need to know the various possible sums of action costs we will have experienced up to that level. We can then combine those with the action costs at this level and the cost of the best completion from the following level to determine how many children will be expanded and to derive the cost sums for the next level. We will maintain a histogram of the action costs experienced so far. This is analogous to the allowance distribution used with indecision search, but progressing additively rather than subtractively (recall Figure 5.3). We can initialize it at the root to a spike at zero. To compute the distribution at the next level, we just create several shifted copies of the current distribution, one for each child cost, to represent the path costs at the following level. Each of these distributions is then truncated at a value corresponding to the given cost bound, minus the best possible completion cost. Any child values that, when added to the best completion cost, go over the bound will be pruned by the search and should be discarded during estimation. The probability mass that survives this shifting and truncation represents the expected number of children that will be expanded at this level. To prepare for the next level, the distributions are added together and renormalized to sum to one.

Given these estimates of the expected number of children at each level, we can use the straightforward equations from Section 5.3 to compute the total number of nodes in the resulting search tree.

Given a model which can estimate the number of nodes that will be visited for a given cost bound, we will search over possible values of the bound until we find one which yields the desired number of nodes. In the experiments reported below, we use a simple 'bracket and bisection' approach (Press et al., 1992), although one could certainly imagine using more sophisticated interpolation and learning schemes. To bracket the desired bound, one can either use numerical approximations to ∞ and $-\infty$ or, if the model can easily compute them, the largest and smallest possible predicted costs. Since it is not important to generate exactly the desired number of nodes, the search was terminated when a bound yielded within 10% of the desired number, or more than desired but fewer than 150% more, or after 10 bisections were carried out. The number of nodes in the entire search tree was also estimated, and the largest possible bound was returned if more than the maximum number of nodes was desired.

6.2 Evaluation

Two additional implementation details should be mentioned. The first concerns pruning. As discussed in Section 5.3, the percentage of time that nodes at a given level were leaves and thus did not give rise to children was recorded for each depth individually, and that information was used to refine the estimated tree size for each given cost bound. This information was not taken into account when computing the best completions, however—it was assumed that actions must be chosen until the deepest tree level ever visited was achieved. The alternative, weighting action costs by the probability that a level would be reached, resulted in dramatic failures to reach leaves during early iterations, as the search could not afford to progress beyond the middle of the tree. As an additional measure to alleviate this problem, the search always expanded the best child of every internal node, ensuring that an internal node would never be visited in vain but would always contribute to updating the model.

The other detail concerns the learning rule. In adaptive probing, we used the basic Widrow-Hoff update rule to learn the parameters of the model from the observed leaf costs. In the experiments reported below, we use a slightly more sophisticated algorithm due to Murata et al. (1997), which attempts to adjust the learning rate automatically. Standard parameter settings were used, with no attempt to optimize them for each problem: initial learning rate 0.2, meta-learning rate (α) 0.002, normalization factor (β) $20/(\max ||r||)$, leakiness (δ) 0.05, learning rate clamped between 0.001 and 1.9. (In informal tests using random probing, this procedure seemed to give slightly better learning than plain Widrow-Hoff or the K1 method proposed by Sutton (1992), although it is not clear if it made a difference in the search results reported below.) To aid learning, we forced the learned costs at each level of the tree to be very mildly increasing with child rank. In other words, we assumed that the heuristic ordering function, while not necessarily very helpful, was not deceptive. This was implemented by performing isotonic regression at each level in the model before the start of each iteration.

We tested this BLFS version of adaptive probing on the number partitioning problem.

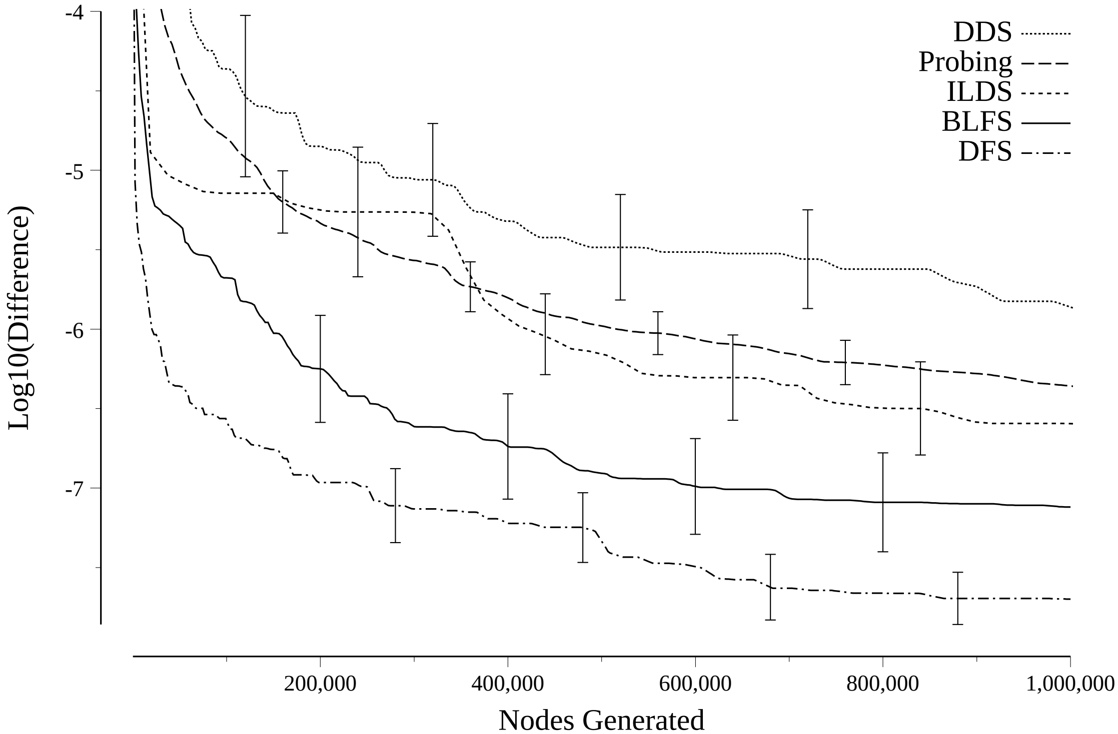

Figure 6.2: Greedy partitioning of 128 numbers

In the experiments reported below, the model was initialized by probing into the tree 10 times, choosing a random child at every decision node.

6.2.1 Greedy Number Partitioning

The number partitioning problem was introduced in Section 3.3.3 (page 43). There are two popular formulations of the problem as a tree search. The first is the straightforward greedy encoding. Figures 6.2 and 6.3 compare the performance of BLFS with DFS, ILDS, DDS, and the adaptive probing algorithm of Chapter 3, which guides search using a similar learned cost model but is stochastic and incomplete. As usual, error bars in the figures indicate 95% confidence intervals around the mean. Although BLFS does not surpass DFS in this search space, it does seem to consistently track DFS as the problem size increases, unlike ILDS and DDS, whose solution quality actually decreases on the larger problems.

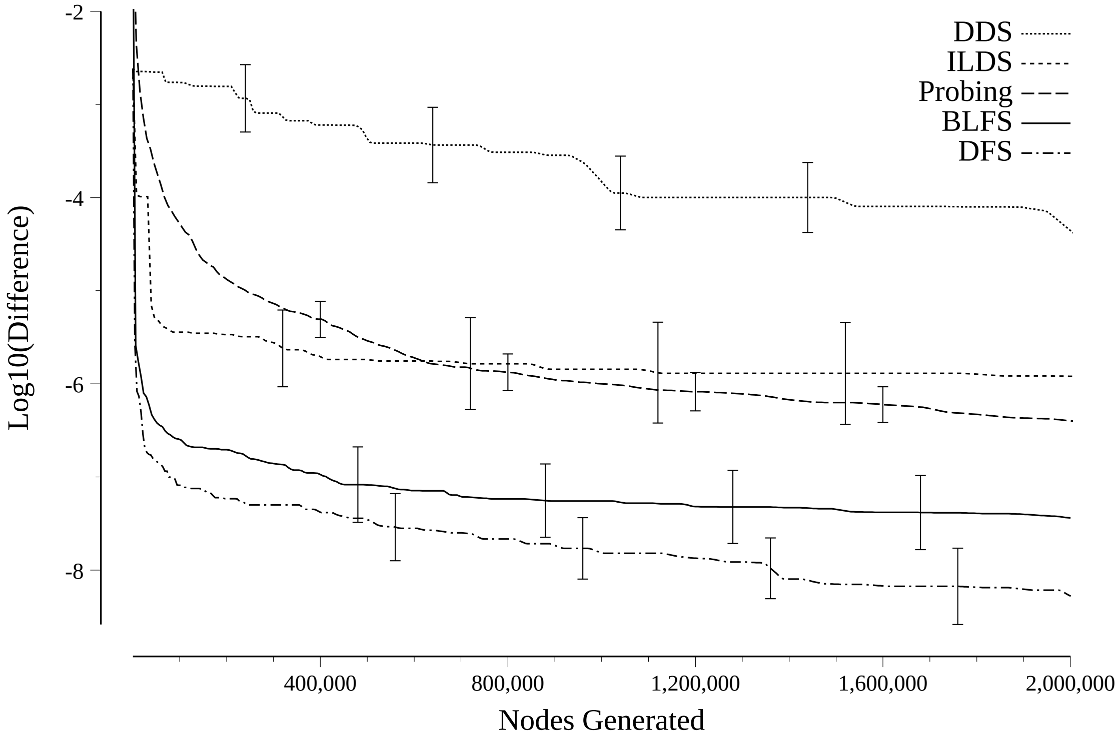

Figure 6.3: Greedy partitioning of 256 numbers

6.2.2 CKK Number Partitioning

The second search space for number partitioning is the CKK representation, due to Korf (1995). Figure 6.4 and 6.5 compare the performance of BLFS with DFS, ILDS, and DDS. (Adaptive probing takes too long to learn to follow the powerful heuristic in this space and would be off the top of both plots.) As in the greedy search space, BLFS successfully adapts and tracks the performance of the best non-adaptive algorithm. In the greedy space, this was DFS, while in the CKK space ILDS outperforms DFS. In fact, BLFS surpasses the performance of ILDS as the problems get larger (Figure 6.5). For larger number partitioning problems, BLFS in the CKK representation yields the best performance known.

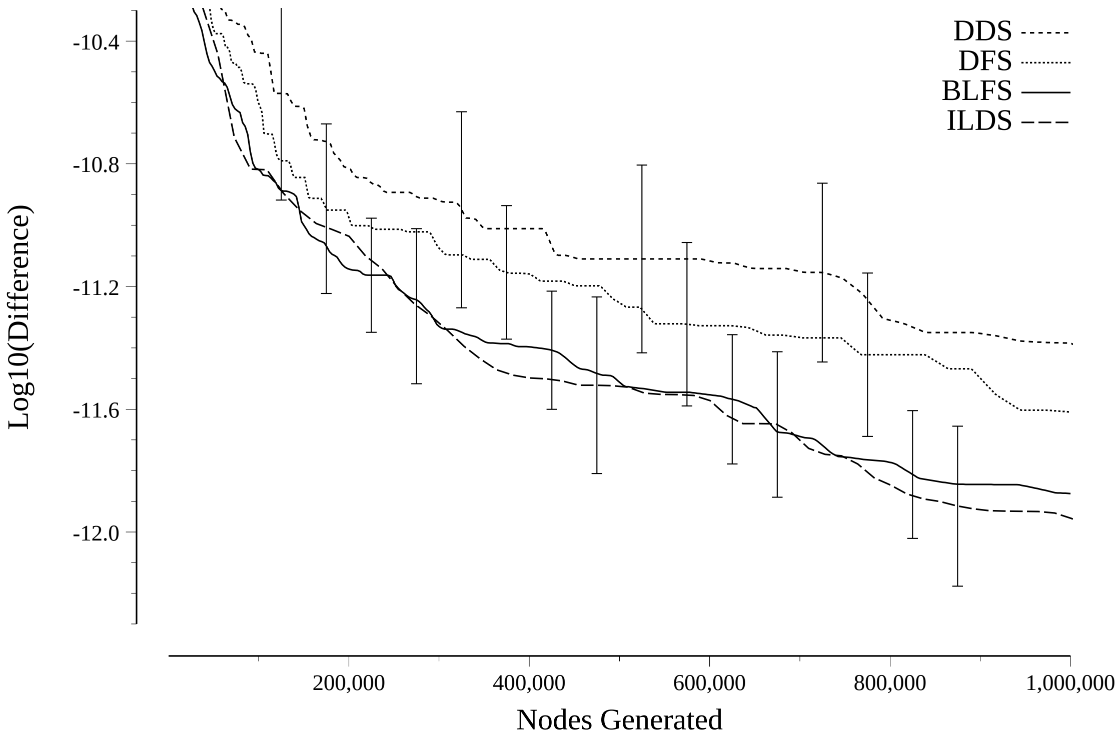

Figure 6.4: CKK representation for partitioning 128 numbers

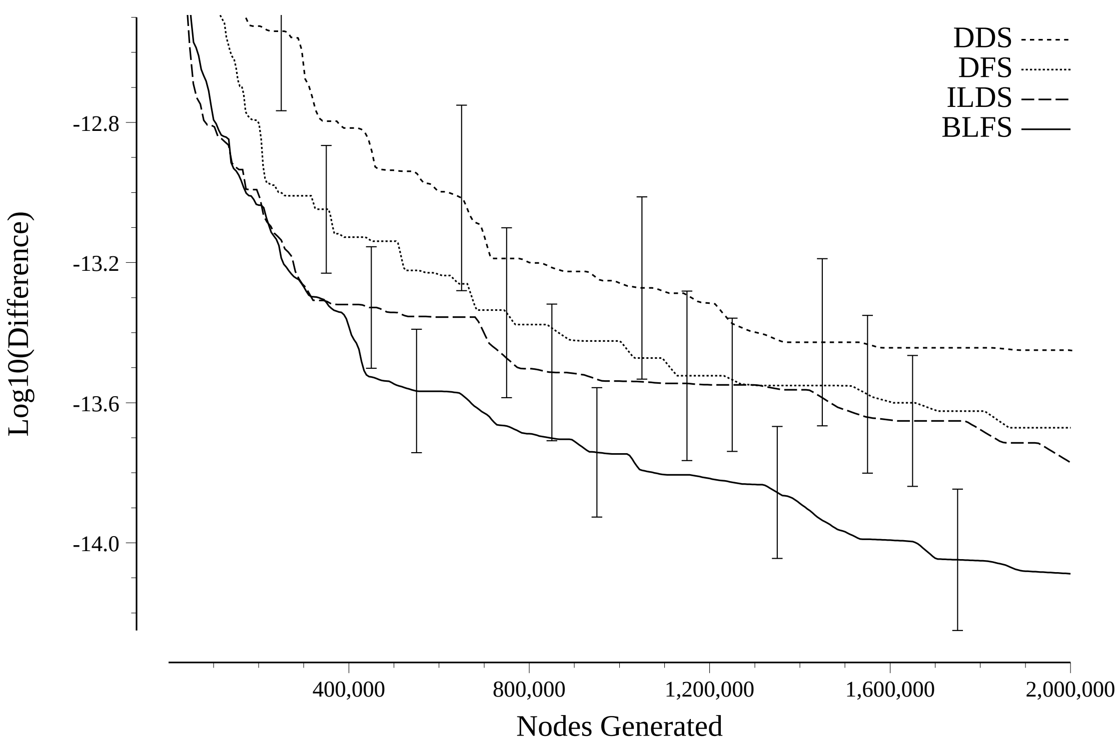

Figure 6.5: CKK representation for partitioning 256 numbers

6.2.3 Time Overhead

Using BLFS with on-line learning does not change the time complexity of the search process. Guiding the search requires only a constant-time table look-up to assess action costs. Updating the model at a leaf is linear in the number of parameters, which is linear in the number of problem variables. This is comparable to recording the best solution seen so far. Empirically, the largest source of overhead is in updating the cost bound, just as it was with indecision search. Propagating the distribution of allowance values down the tree is linear in the number of parameters, because the histograms are bounded by a constant, but can take significant time. The cost bound is updated a logarithmic number of times, so for long runs this overhead will be negligible, but for the prototype implementation and run lengths reported here the overhead consumed 20–30% of the total search time.

6.3 Integrating Multiple Sources of Information

So far, our models have exploited either heuristic child scores or leaf costs, but not both. Child scores provide local information, as they are usually computed using information pertaining only to the part of the problem under immediate consideration. For instance, in a constraint satisfaction problem, the value that is involved in the fewest active constraints might be chosen without regard for its influence in later stages of the problem. In the traveling salesman problem, the next city to visit might be selected according to the nearest-neighbor heuristic. Leaf costs provide global information, as they depend on all of the problem variables. They are the only source of guidance in many improvement-based algorithms such as hill-climbing and simulated annealing. In this section, we will show how the cost model of BLFS provides a convenient way to combine the two forms of information.

We will use a simple extension of the model used in indecision search in Chapter 5. In that model, the heuristic scores for each child were normalized by subtracting the score of the best child to produce a cost for choosing that child. If the child scores were $c_0, c_1, \ldots$, then child i had cost $c_i - c_0$. The cost of a leaf was either the sum of the costs of the nodes along its path from the root or the maximum of these costs. We now extend the sum model to include a separate weighting coefficient at each level of the tree. These weights will be estimated during the search using the observed leaf costs, using the same on-line regression method we used earlier in this chapter. The weights allow us to relax the assumption that heuristic score differences are strictly comparable across levels of the tree. We will also include an additional parameter into the model to serve as a constant term in the weighted sum.

We can compare the performance of BLFS using the new weighted sum cost model with its performance using the plain unweighted sum of normalized costs to see how helpful the leaf cost information is in improving the model. We will also see BLFS with the cost model explored earlier in this chapter, using only the leaf cost information to learn costs for each child rank at each depth. This is different than the new model that uses heuristic scores for two reasons: 1. the preferred child is not always free, and 2. the cost of a child at a particular level is constant and does not depend on its heuristic score.

6.3.1 Evaluation

To allow easy comparison with the methods we discussed earlier in this chapter, we will evaluate the new model on the problem of number partitioning. In particular, we will use the greedy search space formulation in which each decision places the largest remaining number into one of the partitions. The partition with the currently smaller sum is preferred

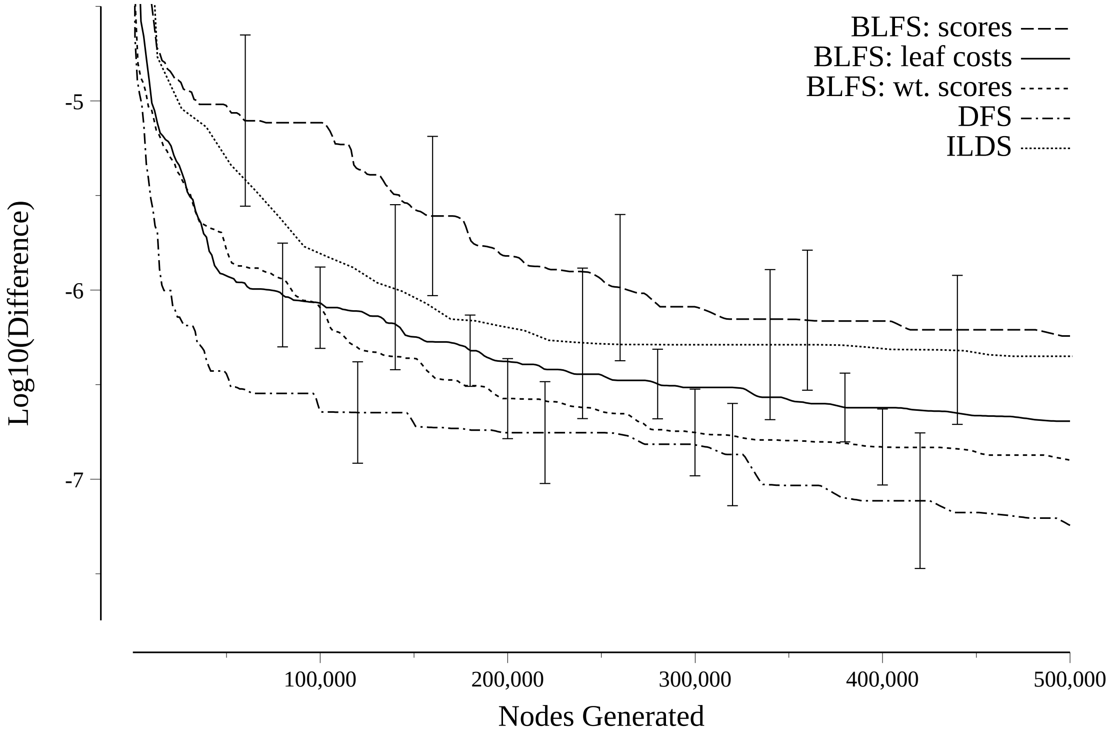

Figure 6.6: Performance on 64-number problems.

and the logarithm of the absolute difference between the partitions is used as the normalized heuristic score of the second child. Clearly, it seems less sensible to place a number in the currently larger partition the greater its sum is than that of its competitor. One might reasonably expect a correlation between the current difference and the cost of the final solution obtained.

Figures 6.6 through 6.8 present the performance of DFS, ILDS, and BLFS using the three cost models we have discussed. 'Scores' refers to the original indecision search model that only uses the heuristic child scores, 'leaf costs' refers to the model that only uses leaf costs, and 'wt. scores' refers to the new model that uses both sources of information in a weighted scores model. The performance of DDS is not shown, as it failed to surpass the performance of random sampling on the 256-number problems. Random sampling is also not shown, as it always performed worse than the remaining algorithms.

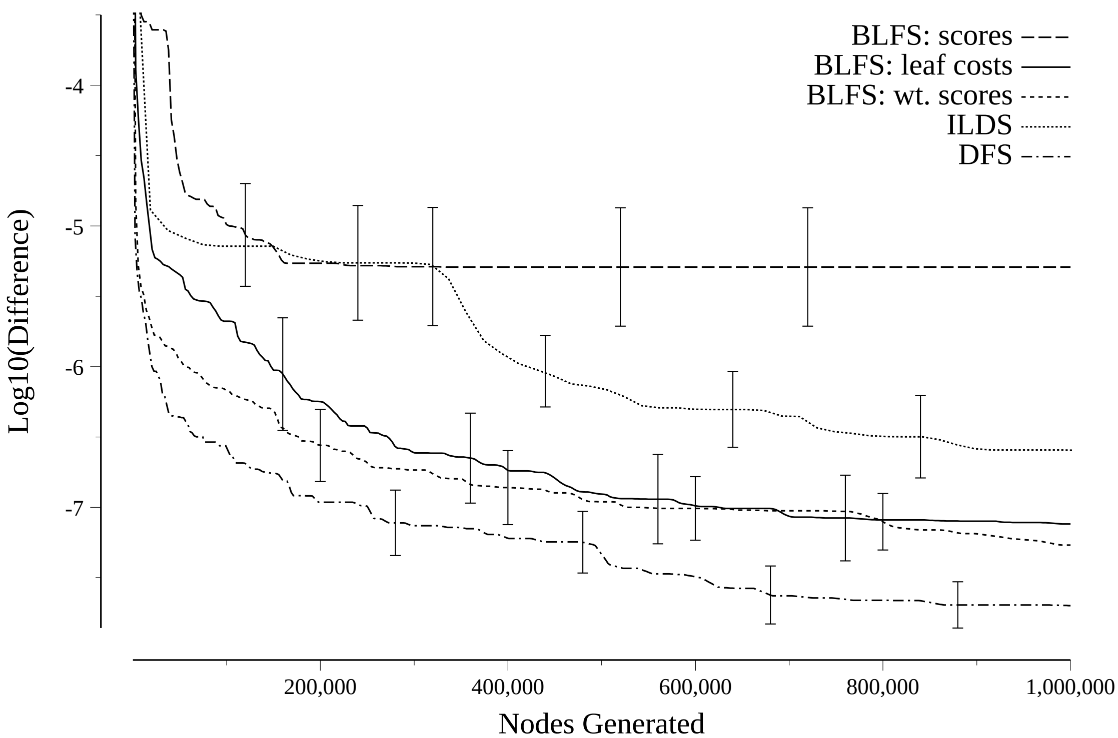

Figure 6.7: Performance on 128-number problems.

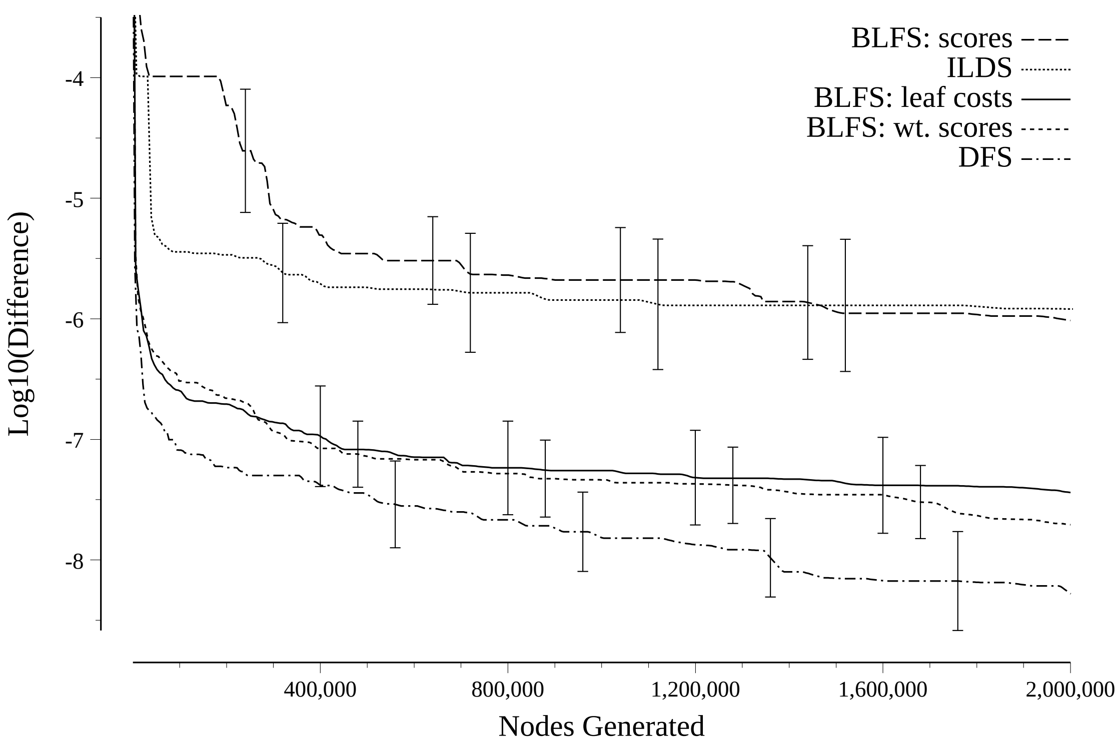

Figure 6.8: Performance on 256-number problems.

The figures show that using the leaf cost information to learn weights for the heuristic scores leads to a large improvement in performance over the unweighted use of the same scores. However, the combined model does not seem to perform significantly better than using leaf costs alone to estimate child costs.

The particular model we examined is just a simple example of the ease with which BLFS allows multiple sources of information to be combined. Other models may lead to improved performance. One obvious avenue for future work would be a model closer to the one that only uses leaf costs. For instance, leaf cost could be predicted as the sum of child costs, each of which is the sum of a weighted heuristic score (with the weight depending quadratically on depth) and a constant (depending on depth and child rank).

6.4 Summary of Results

We have seen in this chapter that BLFS can successfully adapt to different search spaces, even given only observed leaf costs. Different forms of heuristic information can be combined in a principled way in the tree cost model and exploited for search. BLFS is more robust than any other known algorithm, always performing competitively with or better than the best previously known strategy for each of our benchmark domains. For cases in which ILDS was the best method known, BLFS exhibited superior performance. For cases in which DFS is the best method known, BLFS tracked its performance. Unlike DFS, BLFS never exhibited pathological behavior such as taking orders of magnitude more time than other algorithms.

The main drawback of BLFS seems to be its time overhead for bound estimation during short runs. For domains in which many short runs will be performed, each on a similar problem, one way around this problem would be to reuse the cost model and its corresponding bound estimates across problems.

6.5 Possible Extensions

We investigated five different tree models for use with BLFS:

plain indecision: the cost of a leaf is assumed proportional to the sum of the normalized child scores along its path from the root.

simple indecision: the cost of a leaf is assumed proportional to the maximum of the normalized child scores along its path from the root.

separate action costs: as in adaptive probing, each child rank at each depth is a separate cost and the cost of a leaf is predicted as the sum of the costs along its path

quadratic action costs: all the children of a particular rank (e.g., all children ranked second) have action costs which are a quadratic function of depth

weighted heuristic scores: the cost of a leaf is predicted as the weighted sum of the heuristic scores of the nodes along its path, with a separate weight for each level in the tree.

There is no reason why other models could not be tried. For instance, one might suppose that the cost of an action at a given level was a linear function of two variables: the depth and the score of that child. The coefficients on this linear function might be restricted to being a quadratic function of depth. In this way, the quadratic action cost model could be supplemented by child score information. Other possibilities include a multiplicative model, which could easily be implemented by maintaining logarithms, or a model based on taking the maximum cost along a path, as in the simpler variant of indecision search.[1]

By analyzing models that prove successful over multiple domains, it may be possible to design useful new search algorithms. In domains where the on-line costs of adaptation are too high, BLFS may still be useful to help select, diagnose, and tune a non-adaptive method.

Child preferences and leaf costs are the two fundamental types of information available during a tree search. A different type of information that is often exploited in optimization problems is improvement advice, which suggests changes to complete solutions. The process of evaluating and executing such changes amounts to a search in the graph of complete solutions, a process known variously as improvement search, heuristic repair, or local search (recall Section 1.1.4). These search spaces are sufficiently distinct from search trees that we do not consider them in this thesis. It would certainly be interesting to explore how improvement information might be used in conjunction with a tree search. One might view the work on 'squeaky wheel optimization' of Joslin and Clements (1998) as pointing in this direction. In that method, gradient information is used to influence a variable choice heuristic. Neither of these forms of heuristic information is considered in the models presented here.

It should also be straightforward to train multiple models, each of which is slightly more complex than the one before, and guide the search using the simplest model until the next most complex exhibits lower average prediction error across an entire iteration. The simpler model can then be discarded, and the process could continue with the next more complicated model.

[1]It may well be possible to modify Widrow-Hoff to learn a max model.

As we discussed in Section 4.3, it would also be interesting to extend the tree model to explicitly include estimates of its uncertainty. When used with a known deadline, this would allow active learning to reduce model uncertainty, even when the necessary actions are not those that lead to the best solution in the near term.

Chapter 7

Conclusions

Adaptive tree search has enormous potential. This thesis has shown that adaptive methods can be general, requiring no problem-specific information that is not already available; efficient, adding no more than a constant factor to the complexity of a tree search; and effective, solving problems as well or better than current methods and exhibiting much more robust performance.

A simple adaptive probing algorithm demonstrated that it was possible to efficiently learn a model of the distribution of leaf costs in the tree and, at the same time, exploit it for search. Performance on several constraint satisfaction and combinatorial optimization problems showed the algorithm to be exceptionally robust across both different types of problems and different instances of the same type.

Best-leaf-first search (BLFS), a framework for complete adaptive search, uses an explicit model of leaf costs and visits leaves in an efficient approximation of increasing predicted cost. The cost model can be learned on-line during the search, enabling the algorithm to approximate rational exploitation of heuristic information. All previous proposals for complete tree search—including depth-first search, limited discrepancy search, depth-bounded discrepancy search, and iterative broadening—are special cases of BLFS.

We investigated several different cost models for BLFS. The first two were based on the scores assigned by a heuristic node ordering function. They led to excellent performance on latin square completion and all but one class of binary constraint satisfaction problems. The later models were based on on-line learning of action costs or score weights from leaf costs. This approach led to the best results known for the combinatorial optimization problem of number partitioning. BLFS often approached or surpassed the best previous method for each problem class and it never exhibited the pathologically brittle behavior of DFS.

The explicit cost model of BLFS makes it clear that the assumptions made by iterative broadening and the various discrepancy search algorithms are actually very rough approximations of $f(n)$. Similarly, the use of child ordering in traditional depth-first branch-and-bound search is a attempt to exploit the child score as a predictor of the leaf cost. BLFS unites all of these techniques under the same conceptual umbrella, and makes it clearer how one might go about designing more effective ways to leverage problem specific information to improve search order. Adaptive probing and its BLFS analogue demonstrate how feature weights for the f function can be learned on-line, allowing one to toss features into the pot and see if they are actually found to be predictive.

This thesis shows how the application of core ideas from artificial intelligence about exploiting heuristic information can make a significant contribution to problems at the core of operations research. BLFS clarifies the relationship between combinatorial optimization and shortest-path problems. The two problems are not fundamentally different—they can both be solved by the same general approach of single-agent rationality. Only adversarial search is a fundamentally different type of tree search, due to the introduction of a second agent and the need for strategic reasoning.

7.1 Future Directions

Tree search using heuristic information remains a fundamental algorithmic problem in artificial intelligence and other areas of computer science and operations research. The adaptive tree search methods that we have investigated are very general and should be applicable to a wide variety of problems. The robustness of BLFS, as demonstrated on latin square completion for example, may also allow reconsideration of problems previously thought intractable. One interesting direction for future work is dynamic problem domains, in which the underlying optimization problem changes during the search. An adaptive approach should be able to adjust its search order on the fly to handle such disruptions.

This thesis emphasizes the central role that learning can play in a search process. The flexibility of an adaptive approach reduces the chance that the algorithm's assumptions will lead to poor performance. This robustness raises the possibility that a tree search could be reliably used to quickly find near-optimal solutions, a task that has traditionally been left to improvement search algorithms. However, improvement search cannot take advantage of the kinds of heuristic information that are often available during tree search. If robust tree-based and improvement-based methods were both available, researchers would have the freedom to use whichever search paradigm allowed the fullest exploitation of available heuristic domain knowledge. This work takes an important step toward this goal. It remains to be seen, however, if it might be possible to integrate the information that is typically provided to these two types of algorithms. The view we have taken of tree search has emphasized rational inference on the basis of acquired information. By making the assumptions of the search explicit in a model of the tree, it becomes clear how to use the available information to guide search. A similar approach may be helpful for improvement-based algorithms.

References

Abramson, Bruce. 1991. *The Expected-Outcome Model of Two-Player Games*. Pitman.

Baluja, Shumeet. 1997. Genetic algorithms and explicit search statistics. In Michael C. Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems 9*.

Baluja, Shumeet and Scott Davies. 1998. Fast probabilistic modeling for combinatorial optimization. In *Proceedings of AAAI-98*.

Bedrax-Weiss, Tania. 1999. *Optimal Search Protocols*. Ph.D. thesis, University of Oregon, Eugene, August.

Bishop, Christopher M. 1995. *Neural Networks for Pattern Recognition*. Oxford University Press.

Boese, Kenneth D., Andrew B. Kahng, and Sudhakar Muddu. 1994. A new adaptive multi-start technique for combinatorial global optimizations. *Operations Research Letters*, 16:101–113.

Boyan, Justin A. and Andrew W. Moore. 1998. Learning evaluation functions for global optimization and boolean satisfiability. In *Proceedings of AAAI-98*.

Boyan, Justin A. and Andrew W. Moore. 2000. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77–112.

Brélaz, Daniel. 1979. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, April.

Bresina, John L. 1996. Heuristic-biased stochastic sampling. In *Proceedings of AAAI-96*, pages 271–278. AAAI Press/MIT Press.

Cesa-Bianchi, Nicolò, Philip M. Long, and Manfred K. Warmuth. 1996. Worst-case quadratic loss bounds for on-line prediction of linear functions by gradient descent. *IEEE Transactions on Neural Networks*, 7(2):604–619.

Chu, Lon-Chan and Benjamin W. Wah. 1992. Band search: An efficient alternative to guided depth-first search. In *Proceedings of the Fourth International Conference on Tools with Artificial Intelligence*.

Crawford, James M. and Andrew B. Baker. 1994. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of AAAI-94*, pages 1092–1097.

Dantzig, G. B., D. R. Fulkerson, and S. M. Johnson. 1954. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410.

Dearden, Richard, Nir Friedman, and Stuart Russell. 1998. Bayesian q-learning. In *Proceedings of AAAI-98*, pages 761–768.

Dorigo, Marco and Luca Maria Gambardella. 1997. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66.

Euler, Leonhard. 1759. Solution d'une question curieuse qui ne paroit soumise à aucune analyse. *Mem. Acad. Sci. Berlin*, 15:310–337.

Feo, T. A. and M. G. C. Resende. 1995. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133.

Garey, Michael R. and David S. Johnson. 1991. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York.

Geelen, P. A. 1992. Dual viewpoint heuristics for binary constraint satisfaction problems. In B. Neumann, editor, *Proceedings of ECAI-92*, pages 31–35.

Gent, Ian P. and Toby Walsh. 1996. Phase transitions and annealed theories: Number partitioning as a case study. In *Proceedings of ECAI-96*.

Ginsberg, Matthew L. and William D. Harvey. 1992. Iterative broadening. *Artificial Intelligence*, 55:367–383.

Gomes, Carla P. and Bart Selman. 1997. Problem structure in the presence of perturbations. In *Proceedings of AAAI-97*, pages 221–226.

Gomes, Carla P., Bart Selman, Nuno Crato, and Henry Kautz. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100.

Gomes, Carla P., Bart Selman, and Henry Kautz. 1998. Boosting combinatorial search through randomization. In *Proceedings of AAAI-98*.

Hansson, Othar. 1998. *Bayesian Problem-Solving Applied to Scheduling*. Ph.D. thesis, University of California, Berkeley.

Hansson, Othar and Andrew Mayer. 1994. DTS: A decision-theoretic scheduler for space telescope applications. In Monte Zweben and Mark S. Fox, editors, *Intelligent Scheduling*. Morgan Kaufmann, San Francisco, chapter 13, pages 371–388.

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

Harvey, William D. and Matthew L. Ginsberg. 1995. Limited discrepancy search. In *Proceedings of IJCAI-95*, pages 607–613. Morgan Kaufmann.

Herodotus. 440 BC. *Histories*. Book II.

Horvitz, Eric, Yongshao Ruan, Carla Gomes, Henry Kautz, Bart Selman, and Max Chickering. 2001. A bayesian approach to tackling hard computational problems. In *Proceedings of UAI-01*.

Johnson, David S., Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. 1991. Optimization by simulated annealing: An experimental evaluation; Part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, May-June.

Joslin, David E. and David P. Clements. 1998. "Squeaky wheel" optimization. In *Proceedings of AAAI-98*, pages 340–346. MIT Press.

Juillé, Hughes and Jordan B. Pollack. 1998. A sampling-based heuristic for tree search applied to grammar induction. In *Proceedings of AAAI-98*, pages 776–783. MIT Press.

Karmarkar, Narenda and Richard M. Karp. 1982. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, Computer Science Division, University of California, Berkeley.

Karmarkar, Narenda, Richard M. Karp, George S. Lueker, and Andrew M. Odlyzko. 1986. Probabilistic analysis of optimum partitioning. *Journal of Applied Probability*, 23:626–645.

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

Korf, Richard E. 1990. Real-time heuristic search. *Artificial Intelligence*, 42:189–211.

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

Korf, Richard E. 1995. From approximate to optimal solutions: A case study of number partitioning. In *Proceedings of IJCAI-95*.

Korf, Richard E. 1996. Improved limited discrepancy search. In *Proceedings of AAAI-96*, pages 286–291. MIT Press.

Lagoudakis, Michail G. and Michael L. Littman. 2001. Learning to select branching rules in the DPLL procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9, June. LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001).

Larrosa, Javier and Pedro Meseguer. 1995. Optimization-based heuristics for maximal constraint satisfaction. In *Proceedings of CP-95*, pages 103–120.

Luger, George F. and William A. Stubblefield. 1998. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison Wesley Longman, third edition.

Mayer, Andrew Eric. 1994. *Rational Search*. Ph.D. thesis, University of California, Berkeley, December.

Meseguer, Pedro. 1997. Interleaved depth-first search. In *Proceedings of IJCAI-97*, pages 1382–1387.

Meseguer, Pedro and Toby Walsh. 1998. Interleaved and discrepancy based search. In *Proceedings of ECAI-98*.

Murata, Noboru, Klaus-Robert Müller, Andreas Ziehe, and Shun-ichi Amari. 1997. Adaptive on-line learning in changing environments. In Michael Mozer, Michael Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems 9 (NIPS-96)*, pages 599–605. MIT Press.

Nilsson, Nils J. 1998. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, San Francisco, CA.

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

Poole, David, Alan Mackworth, and Randy Goebel. 1998. *Computational Intelligence: A Logical Approach*. Oxford University Press.

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 1992. *Numerical Recipes in C*. Cambridge University Press, second edition.

Ruml, Wheeler. 2001a. Incomplete tree search using adaptive probing. In *Proceedings of IJCAI-01*, pages 235–241.

Ruml, Wheeler. 2001b. Stochastic tree search: Where to put the randomness? In Holger H. Hoos and Thomas G. Stützle, editors, *Proceedings of the IJCAI-01 Workshop on Stochastic Search*, pages 43–47.

Ruml, Wheeler. 2001c. Using prior knowledge with adaptive probing. In Carla Gomes and Toby Walsh, editors, *Proceedings of the 2001 AAAI Fall Symposium on Using Uncertainity Within Computation*, pages 116–120. AAAI Technical Report FS-01-04.

Ruml, Wheeler, Adam Ginsburg, and Stuart M. Shieber. 1999. Speculative pruning for boolean satisfiability. Technical Report 99-02, Harvard University.

Russell, Stuart and Eric Wefald. 1991. *Do the Right Thing: Studies in Limited Rationality*. MIT Press.

Sutton, Richard S. 1992. Gain adaptation beats least squares? In *Proceedings of the Seventh Yale Workshop on Adaptive and Learning Systems*, pages 161–166.

Wah, Benjamin W. and Yi Shang. 1995. Comparison and evaluation of a class of IDA* algorithms. *International Journal on Artificial Intelligence Tools*, 3(4):493–523, October.

Walsh, Toby. 1997. Depth-bounded discrepancy search. In *Proceedings of IJCAI-97*.

Wyatt, Jeremy. 1997. *Exploration and Inference in Learning from Reinforcement*. Ph.D. thesis, University of Edinburgh.

Zhang, Wei and Thomas G. Dietterich. 1995. A reinforcement learning approach to job-shop scheduling. In *Proceedings IJCAI-95*.

Zhang, Weixiong and Richard E. Korf. 1993. Depth-first vs. best-first search: New results. In *Proceedings of AAAI-93*, pages 769–775.

Zilberstein, Shlomo, François Charpillet, and Phillippe Chassaing. 1999. Real-time problem-solving with contract algorithms. In *Proceedings of IJCAI-99*, pages 1008–1013.

This appended page exists solely as a result of formatting bugs. It is not part of the thesis.