OPTIMAL SEARCH PROTOCOLS

by

TANIA BEDRAX-WEISS

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

August 1999

"Optimal Search Protocols," a dissertation prepared by Tania Bedrax-Weiss in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

[illegible]

Dr. Matthew Ginsberg, Co-chair of the Examining Committee

[illegible]

Dr. Arthur Farley, Co-chair of the Examining Committee

8-24-99

Date

Committee in charge:	Dr. Matthew Ginsberg, Co-chair
	Dr. Arthur Farley, Co-chair
	Dr. David Etherington
	Dr. Andrzej Proskourowski
	Dr. Richard Koch

Accepted by:

[illegible]

Dean of the Graduate School

An Abstract of the Dissertation of

Tania Bedrax-Weiss for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken August 1999

Title: OPTIMAL SEARCH PROTOCOLS

Approved: [illegible]

Dr. Matthew Ginsberg, Co-chair

[illegible]

Dr. Arthur Farley, Co-chair

Many problems in artificial intelligence involve searching for a set of decisions that eventually lead to some desired state, or goal. The only concern, for the purposes of this dissertation, is to find a solution as fast as possible, that is, any solution is as good as any other. If the problem is structured as a tree and solutions are only found at terminal (leaf) nodes, the optimal protocol will eliminate leaf nodes as fast as possible searching the most promising leaves first. A heuristic, or rule of thumb, usually decides which nodes are more likely to lead to a goal.

Unfortunately, if the distribution of heuristic values is not uniform or if it is dense (as it is in the case where they are real-valued), existing heuristic search techniques perform badly. The best-first family of algorithms may get trapped among many competing lines of decisions and take too long to eliminate leaves, which may prove fatal if the heuristic makes mistakes. Limited discrepancy search LDS, and its current variations, developed on the intuition that heuristics make mistakes, explore leaves in order of increasing number of deviations from the heuristic path. However, it may sometimes be better to explore paths with two or more deviations that are close to the heuristic choice than to explore paths with one clear deviation. In this case, LDS fails to explore leaves in the best order.

We present an optimal algorithm WDS and a general methodology to obtain optimal protocols. We prove that WDS with an optimal cutoff policy is better than LDS and other backtracking alternatives. In formalizing WDS, we present a principled way to obtain optimal policies that minimize the expected cost to solution. We also show that the approach is general enough to obtain optimal policies in cases where there are arbitrary cost functions associated with node expansions.

CURRICULUM VITA

NAME OF AUTHOR: Tania Bedrax-Weiss

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Catholic University of Chile

DEGREES AWARDED:

Doctor of Philosophy in Computer Science, 1999, University of Oregon
Master of Science in Computer Science, 1995, University of Oregon
Master of Engineering Sciences, 1995, Catholic University of Chile
Bachelor of Science in Mathematics, 1993, Catholic University of Chile

AREAS OF SPECIAL INTEREST:

Artificial Intelligence and Combinatorial Optimization

PROFESSIONAL EXPERIENCE:

Research Assistant, Department of Computer and Information Science, University of Oregon, Eugene 1993-99

Student Researcher, Department of Computer Science, Catholic University of Chile, Santiago, Chile 1993–95

Instructor, Faculty of Mathematics, Catholic University of Chile, Santiago, Chile 1993

Research Assistant and Programmer, Department of Mathematics, Catholic University of Chile, Santiago, Chile 1990–93

Teaching Assistant, Department of Mathematics, Catholic University of Chile, Santiago, Chile 1993

Teaching Assistant, Department of Computer Science, Catholic University of Chile, Santiago, Chile 1992

Teaching Assistant, Department of Mathematics, Catholic University of Chile, Santiago, Chile 1991

System Administrator, Department of Computer Science, Catholic University of Chile, Santiago, Chile 1990–91

AWARDS AND HONORS:

Service Award, Department of Computer and Information Science, University of Oregon, Eugene 1997

Residency Scholarship, Engineering School, Catholic University of Chile, Santiago, Chile 1995

Postgraduate Scholarship, National Commission of Scientific and Technological Research, Santiago, Chile, 1993

Scholarship in Support for Graduate Thesis, Catholic University of Chile, Santiago, Chile 1993

Best student award, Faculty of Mathematics, Catholic University of Chile, Santiago, Chile 1993

ACKNOWLEDGEMENTS

I would like to thank the following people: Matt Ginsberg whose honesty and guidance helped me get to where I am today. I will miss Matt's criticism in research and his optimism in life. David Etherington for reminding me how important details are. Art Farley for being a great co-advisor through the years. Andrzej Proskourowski and Dick Koch for their constructive comments on my dissertation. My former officemates, Andrew Baker, Ari Jónsson, and Andrew Parkes, for the countless discussions we had. Dave Clements for reading an early draft. Bart Massey for the thesis macros for LaTeX. Rich Korf for his clear and prompt answers to my questions. Will Harvey, Chip McVey, Brian Drabble, Alan Jaffray, Heidi Dixon, Paul Walser, Hudson Turner, Jimi Crawford, Joe Pemberton, David Joslin, and Laurie Buchanan. You have influenced my life in numerous ways.

Most importantly, I would like to thank my family for allowing me to pursue my dreams. I know it was not easy having me 10,000 kilometers away. Fortunately, frequent long distance calls and email helped bridge the distance, bringing us even closer together. I want my family to know that I love them deeply.

This work was sponsored in part by grants from Air Force Office of Scientific Research (AFOSR), number F49620-92-J-0384, and Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Rome, NY, under agreements numbered F30602-93-C-0031, F30602-95-1-0023, F30602-97-1-0294 and F30602-98-2-0181. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

DEDICATION

To my parents, with love.

TABLE OF CONTENTS

LIST OF FIGURES

Page

Page

Page

LIST OF TABLES

Page

Page

CHAPTER I

INTRODUCTION

The work in this dissertation begins to quantitatively address the question: How to achieve rational behavior, or how to "do the right thing". The Principle of Rational Behavior was initially defined by Good [21] as

> The recommendation always to behave so as to maximize the expected utility per time unit.

Russell and Wefald [51] argue that "behavior" is the result of deliberating and acting. Deliberation takes place when an agent explores different possible lines of decisions before committing to a course of action. Acting occurs when the agent executes the action. This dissertation is exclusively concerned with deliberation. The objective of this work is to find the optimal search protocol. The optimal search protocol explores possible lines of decisions in the order that maximizes the utility of searching for the best possible course of action. The utility will be measured in terms of the cost associated with the exploration of the different decisions. The optimal search protocol maximizes utility by minimizing the cost of the search process.

Think of the search process as building a *search tree* where the root of the tree corresponds to the initial state. A new node is generated by expanding the root node. *Leaf nodes* of the tree correspond to states that do not have successors and *fringe nodes* are nodes that have been explored but have at least one successor, unless the node is a leaf, which has not been explored. If every path from the root to a leaf node has length d, we call d the *depth* of the tree. *Internal nodes* are nodes that are not leaf nodes and are part of paths that have length smaller than d. The length of the path denotes the *level* at which the node can be found. If the number of nodes that results from each node expansion is constant, we call this number the *branching factor* of the tree.

It is important to distinguish between the state space and the search tree. Usually, there are only a finite number of states, but there may be exponentially many paths between the states, so the search tree may have exponentially many nodes. Since there is usually not enough time or space available to search all of the nodes, it is necessary to carefully select the order in which nodes should be expanded.

1.1 Heuristic Search

Uninformed search protocols have no information about the likelihood of a choice leading to a goal. All they can do is discriminate between goal and non-goal nodes. In contrast, *informed search protocols* use a heuristic, or rule of thumb, to prefer choices that seem more likely to lead to a solution. The heuristic is an evaluation function that returns a *signal* representing the "value" of expanding that node. The signal is computed according to the information available to the evaluation function. Heuristics can only provide a guess as to which node appears best. Sometimes, the evaluation function can lead the search astray. We will refer to heuristics that tend to be accurate in their evaluation more often than not as *good heuristics.*

Good heuristics have been found for some constraint satisfaction problems (CSP) [59]. A CSP is a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can take simultaneously. The objective is to assign a value to each variable in order to satisfy all the constraints.

Harvey [24] found that heuristics for solving scheduling problems as constraint satisfaction problems lead the search algorithm straight to a goal most of the time (with probability close to 1). In scheduling, a set of tasks with precedences have to be assigned start times so that no constraints are violated. Other constraints on possible assignments to variables may include ready times for tasks, deadlines, resource requirements and availability, and setup costs.

One way to represent the scheduling problem as a CSP is to consider the problem of assigning start times to tasks directly. In this case, the variables of the CSP are the tasks and the domain is time. Search typically proceeds by identifying one conflict and resolving it before resolving any other conflicts. This formulation usually results in a prohibitively large search space, since there are many possible times at which tasks can be scheduled.

A much better formulation arises from the insight that when scheduling a particular task, different choices for time may fail for the same reason. For instance, assume we choose to schedule task a after task b, despite the fact that a must precede b. We may choose to schedule it immediately after or at any point in time after b finishes. What is important here is not the exact time at which a is scheduled, but rather, that a is scheduled to go *after* b.

The search space can be reduced dramatically by considering only the relative ordering of tasks, rather than absolute start times. Times can easily be assigned later. In this "precedence-based" formulation [56], the search starts with an ordering of the tasks that satisfies all of the precedence constraints. It then proceeds by identifying a pair of tasks that violate some other constraint (resource, deadline, etc.) and imposes a new precedence constraint in order to resolve the conflict. A schedule is found when all constraints are satisfied. Thus, in this formulation of scheduling, and in constraint satisfaction problems in general, all goals are found at the leaves.

Two common types of scheduling problems are job-shop scheduling and resource constrained project scheduling [14]. In job-shop scheduling, a number of jobs consisting of an ordered collection of tasks must be scheduled to run on machines by a set deadline. Each task may only be scheduled on one machine at a time and all tasks have to be scheduled. Ready times and processing times for each task are given. Resource constrained project scheduling is similar to job-shop scheduling except that tasks may require more than a single resource (machine, labor, etc.). We identify and justify the assumptions made throughout this work based on observations about job-shop and resource-constrained project scheduling problems.

1.2 Mistakes

Depth-first search (DFS), explores the heuristic path—the path formed by choosing the heuristically-preferred option at each choice point—first, chronologically backtracking until a solution is found. Since DFS always expands nodes at the deepest level of the search, DFS is the protocol that eliminates leaf nodes the fastest. Thus, DFS is a good candidate for an optimal search protocol. Unfortunately, for many problems, heuristics are more reliable near the bottom of the search tree, after the problem size has been reduced, and tend to make mistakes early in the search, where decisions are often made blindly and thus less reliably.

Mistake nodes were first defined by Harvey [24, 25]. Harvey defines a *bad* node as a node that has no goals in the subtree beneath it. A *good* node is a node that is not bad. A *mistake* node is a bad node whose parent is a good node. Mistakes can occur only if at least one of the choices that are not preferred by the heuristic leads to a goal. If no choices lead to a goal, the mistake must have been made earlier.

The distribution of mistake nodes is critical to good performance of backtracking algorithms. Mistakes made near the top of the tree pose a larger threat to backtracking algorithms than mistakes made near the bottom. The size of a failed subtree is very large if the mistake is made at the top, and small if the mistake is made at the bottom. Large failed subtrees can trap backtracking algorithms.

Although mistakes may be more likely to occur early in the search, mistakes may also tend to occur elsewhere depending on how multiple heuristics used in the problem interact. In solving constraint satisfaction problems, two kinds of decisions need to be made: (1) which variable to choose and (2) how to value the variable. Variable-ordering heuristics may interact with value-ordering heuristics and force mistakes to be distributed uniformly or more heavily towards the bottom of the search tree.

For instance, in precedence-based scheduling, variable-ordering heuristics may suggest which task should be scheduled next and value-ordering heuristics suggest where it should be scheduled. If, after successfully scheduling $k-1$ tasks, the kth task cannot be scheduled without violating the deadline constraints, then a mistake occurs at depth k. Bad variable orderings cause this mistake to repeat in different subtrees. This happens, for example, when task k is long enough that the other tasks necessarily need to be scheduled around it. A good variable ordering will choose to schedule task k first.

In this work, we do not study the interactions between mistakes and different variable-ordering heuristics. There have been extensive studies of good orderings for scheduling problems both in artificial intelligence and in operations research.[1] For the most part, we assume that a sensible ordering exists for the problem at hand and that mistakes are exclusively the result of errors in the signal.

1.3 Limited Discrepancy Search

By convention, if nodes are heuristically ordered in a binary tree, the heuristic choice is always "left" and heuristic violations— points at which the search does not follow the heuristically preferred choice—are always "right". A heuristic violation is called a *discrepancy* [24]. Discrepancy-based approaches search paths in order of increasing number of discrepancies [2, 18, 25, 36, 61, 41].

These approaches were designed in response to the "early mistakes" problem. Limited discrepancy search [25] was the first efficient implementation. LDS first searches the leftmost path, then all paths with at most one right branch, then all paths with at most two right branches, etc., finally exploring all paths with at

[1]Good places to start looking for information on scheduling heuristics include: http://ic-www.arc.nasa.gov/ic/projects/xfr/papers/benchmark-article.html, http://www.cirl.uoregon.edu/aior, and http://mscmga.ms.ic.ac.uk/info.html.

most d right branches. LDS maintains the discrepancy count of the current branch and backtracks whenever this count exceeds the discrepancy limit, or cutoff. Since the discrepancy count never decreases as LDS descends the tree, all leaf nodes with discrepancy count less than or equal to the cutoff value are explored in each iteration. Because LDS exhaustively searches the whole space at d discrepancies, it is guaranteed to find a solution if there is one, or else prove the problem is unsatisfiable. Other variations of the algorithm [36, 61, 41] keep different accounting measures in order to improve the performance of LDS under different conditions.

Unfortunately, discrepancy-based approaches may easily get confused if the distribution of signal strength is not uniform or if it is dense (as in the case when signals are real-valued). If two choices are rated as essentially equal by the heuristic, then this is an indication that both should be explored at roughly the same time. These two choices are either both equally likely to lead to a goal or the heuristic is just uninformed with respect to which choice is more likely to lead to success. In this case, discrepancy-based methods force one of the closely valued choices to be labelled as a discrepancy, unnecessarily delaying exploration of this choice. Success depends on how lucky the method is in labelling the discrepancy.

Although LDS is optimal in terms of the number of "wrong turns", this number is not always correlated with the probability of success of following that path. One path with two discrepancies may be more likely to lead to a goal than another path with only one, depending on the strength of the signal (heuristic value) associated with the discrepancies. The fundamental observation here is that it is not the number of discrepancies that matter, but rather the quality of the discrepancies.

So far, we have established that it is important to eliminate candidate leaf nodes as quickly as possible and that it is important to spend time searching where the heuristic is likely to make mistakes. We have also established that signal strength plays a more significant role than discrepancies in determining where the search effort should be spent.

Best-first search explores nodes so that the ones with best signal are expanded first. Because best-first search methods expand nodes in strict order of decreasing cost, these methods need enough memory to store the fringe in order to determine which node to expand next. This may potentially require an exponential amount of memory.[2] There are more efficient implementations of best-first search which relax the requirement of expanding nodes in strict order of decreasing cost [33, 34, 51] by iterating over a set of cutoffs and exploring nodes within the cutoff in depth-first order. Unfortunately, these methods tend to adjust the cutoff by small amounts between iterations and if all nodes have different values, the cutoff adjustments may not explore enough nodes to justify the extra cost of the iteration.

Eliminating leaf nodes quickly tends to conflict with exploring them in the best order. Even the more efficient best-first search methods may take a long time to reach leaf nodes. Suppose there are several competing lines of best sets of decisions. Best-first search may spend most of its available time jumping from one set to another. This competition causes best-first search to re-explore these paths over and over again, making little progress towards the leaves. Search effort spent without making progress towards the leaves is wasted because many nodes are being re-explored without improving the chances of success of the algorithm. We refer to this waste of search effort without significantly improving the chance

[2]For instance, when the cost is depth, best-first searches nodes in breadth-first order.

of success as *thrashing.* It is important to explore nodes in order of decreasing value but it is also important to avoid thrashing.

1.4 Contributions

The algorithm proposed in this work, weighted discrepancy search (WDS), finds a tradeoff between eliminating leaves and a good ordering. WDS has the properties that (1) eliminates candidate leaf nodes quickly; (2) explores most promising candidates first; and (3) avoids thrashing. The term weighted comes from the fact that, unlike LDS which attaches binary values to decisions (a decision is either a heuristic choice or a discrepancy), WDS transforms the signal into a weight between 0 and 1. WDS is an iterative algorithm: A cutoff policy determines the nodes that are explored in each iteration. Only nodes with weight greater than or equal to the current cutoff are explored in each iteration. Clearly, the performance of WDS will depend on the cutoff policy used.

We propose a methodology to derive optimal cutoff policies. Optimal cutoff policies minimize expected cost to solution. Optimal cutoff policies that maximize probability of success given limited resources can also be obtained using the same formalism. The optimal cutoff policy (OCP) is computed using only the depth and breadth of the search space and the heuristics given as input.

A cutoff policy determines how WDS should distribute its search effort. Since WDS using the OCP allocates its search effort optimally over the expected distribution of problems, we say that WDS using the OCP is an optimal search protocol. It is optimal in the sense that there is no other choice of cutoff policy that will always do better. Of course it is possible that some completely different search strategy will emerge that does better.

The methodology begins by parameterizing the search space given the depth and breadth of the tree and heuristic information. Having characterized the search space of the problem at hand, we proceed to show it is possible to analytically and numerically estimate the number of nodes explored and the probability of success of WDS at particular cutoff values. Thus, given a cutoff policy, we can compute the expected cost to solution. However, since the objective is to determine the optimal cutoff policy, the expression for the expected cost to solution is formulated in terms of an arbitrary cutoff policy. This expression is minimized using the calculus of variations. The minimization results in a differential equation whose solution is the optimal cutoff policy.

In order to solve the differential equation, we organize the information about the number of nodes explored and probability of success at different candidate values into a performance profile. Because there are infinitely many candidate cutoff values, we only collect data for some candidate values selected at random and interpolate the data for the rest of the values. Thus, the performance profile is only an approximation.

The performance profile quantifies the tradeoff between the cost of nodes explored and the probability of success in exploring that many nodes. This information is used in calculating the expected cost to solution and in solving the differential equation that yields the optimal cutoff policy. We show that WDS with the OCP allocates its resources optimally and is expected to find solutions faster than DFS and LDS.

Specific contributions of this work are outlined in the following sections.

1.4.1 Weighted Discrepancy Search

In WDS, the signal strength is transformed into a weight that is used to decide whether or not to delay exploration of the node and its subtree.[3] The weight captures the likelihood that the choice contains a goal in its subtree. Weighted discrepancy search overcomes the limitations of LDS, since paths with stronger signal strength will be explored before paths with weaker signal strength regardless of the discrepancy count. Exactly which paths are explored when is determined by a cutoff policy. Since weights range anywhere between 0 and 1 (1 being most likely to lead to a goal), it is not always clear how to pick a reasonable cutoff policy.

Our contribution begins by showing that, on real problems, weights are more strongly correlated with real value than are discrepancies. We show that WDS is a generalization of LDS, and that WDS can also simulate 1-SAMP[4] and DFS. We show how to compute the costs in special cases and argue that WDS with the optimal cutoff policy will outperform 1-SAMP, DFS, and LDS. The formulation of the expected cost to solution in terms of an arbitrary cutoff policy is completely original. Although WDS is a generalization of LDS, Harvey's cost analysis for LDS cannot be directly generalized because his analysis assumes the policy is known.

1.4.2 Optimal Cutoff Policies

The next major contribution lies in the development of a quantitative approach for obtaining OCPs based on the expression for the expected cost to solu-

[3]The current incarnation of the weighting scheme was derived during several research meetings at CIRL.

[4]In 1-SAMP, the heuristic is followed to the leaf and if the leaf is not a goal node, the process is repeated with different heuristics.

tion. The expected cost to solution is expressed in terms of the number of nodes explored and the probability of success at particular cutoff values. We show how to estimate these quantities analytically and numerically. The numerical estimates are obtained by Monte Carlo sampling. The following section describes our contributions in doing principled simulations.

Our methodology for obtaining optimal cutoff policies is completely new. Given the expression for the expected cost to solution, we use the calculus of variations to find the cutoff policy that minimizes this expression. The result is a differential equation that we solve using the numerical estimates for the unknowns in the equation.

These results led to new results for LDS. Since the expression for the expected cost to solution is expressed in terms of an arbitrary cutoff policy, we explore the effect of skipping some iterations when doing LDS. We show that sometimes it may be more fruitful to skip a few iterations in order to maximize the chances of success.

Most of the results on optimal cutoff policies are obtained using uniform cost functions. A uniform cost function assumes equal cost for every node expansion. This is not true, in general. In many realistic situations, the value of a solution varies inversely with the time taken to find it. A more realistic cost function might assign lower cost to nodes explored before some node bound is reached and higher cost to nodes explored afterward.

In general, cost functions may represent any function of time. Useful cost functions should account for the cost of reasoning or acting, the presence of limited resources (limits on the number of nodes explored), etc. In general, cost functions may account for anything that affects the agent's behavior over time. If the cost function reflects the limited availability of resources, then the problem is said to belong to the field of "limited rationality" [51]. The objective, in this case, is to search so as to maximize the probability of success within the resource limit.

We use a simple step cost function to illustrate how our analysis applies to the case of arbitrary cost functions. In passing, we show that maximizing probability of success with limited resources is analogous to minimizing the expected cost to solution. We also show that our methodology yields an optimal search protocol with non-uniform cost that outperforms the uniform cost optimal search protocol.

We are not aware of any research that attempts to characterize cost functions that arise in practice. This could be due to the fact that no comprehensive approach to dealing with cost functions has been proposed, until now. We hope that this work will prompt researchers to begin study in this direction.

1.4.3 Search Space Characterization

The structure of the search space is a determining factor of the performance of an algorithm. Harvey and Ginsberg [25] define a characterization of the search space in terms of two parameters. One parameter is the heuristic probability p, the probability that following the heuristic to the leaf will lead to a goal. The other parameter is the mistake probability m, the probability that there is no goal underneath a randomly chosen child. They show that p and m determine the number of goals and their distribution in the search space. Thus, these parameters characterize the search space.

In this work, we adopt and extend their characterization. We argue that the weight computed from the heuristic information is a good approximation to the real probability of success of a node. We show how to compute the parameters p and m from the weights.

Given depth, breadth, p, and m, we show how to construct a random-instance generator. If depth and breadth are not known, methods such as those proposed by Knuth [32] or Purdom [48] can be used to estimate the size of the tree. For the purposes of this dissertation, we consider only complete uniform-depth trees.

1.5 Overview

We begin by providing background on Harvey and Ginsberg's work on parameterizing the search space and on limited discrepancy search. Our contributions start in Chapter III with the motivations that led to the development of weighted discrepancy search. This chapter describes WDS, gives an example of how it works, and shows how WDS can simulate other backtracking algorithms using appropriate weight functions and cutoff policies.

Chapter IV describes how to compute the expected cost of WDS in terms of an arbitrary cutoff policy. Because WDS is an extension of LDS, we begin by presenting the expression for the expected cost to solution for LDS in terms of an arbitrary cutoff policy. We show how this expression can be instantiated to simulate DFS and show that the resulting expression coincides with that in [24]. The chapter ends with the generalization of the formulas to the case of WDS.

In Chapter V, we show how to obtain optimal cutoff policies. We begin with a study of the space of possible cutoff policies. Optimal cutoff policies are obtained by minimizing the expression for the expected cost to solution. The minimization results in a differential equation whose solution is the optimal cutoff policy. We use numerical methods to solve the differential equation. We present experimental results that confirm that WDS using the optimal cutoff policy outperforms LDS and DFS. Finally, we show that our analysis also applies when non-uniform cost functions are associated with node expansions.

We show how our formalism works in the domain of number partitioning in Chapter VI. The number partitioning problem [14] is the problem of finding a partition of the given set of integers such that the sum of the integers in one subset equals the sum of the integers in the other subset. We show that, counter to our intuitions, it is hard to obtain good weights from the best heuristics for the problem. We show that WDS using the OCP is within 10% of the best known search algorithm with the best weights we could find by trying out a few different weighting schemes. This may not sound impressive, but the best algorithm when most instances do not have perfect partitions is DFS, and the best algorithm when most instances do have perfect partitions is ILDS [36], an improved version of LDS, that eliminates some of the redundancy in LDS iterations.

Chapter VII describes other approaches to optimal search that relate to ours and Chapter VIII states our conclusions and discusses possible directions for future research.

CHAPTER II

BACKGROUND

This chapter begins by describing the parameters of the search space according to Harvey and Ginsberg [24, 25]. These parameters determine the goal distribution and therefore, they also determine the performance of search algorithms. Then, we describe limited discrepancy search and show how it works in a small example. Finally, we summarize some performance results that show when LDS outperforms other backtracking alternatives.

2.1 Parameterizing the Search Space

Several factors impact search performance. We already mentioned heuristics, but unless heuristics are perfect, they alone are not exclusively responsible for good performance. Easy problems tend to have many solutions, or solutions that are easily found with the heuristics at hand. Hard problems tend to have few solutions or have solutions that cluster away from heuristic choices. Goal density and distribution in combination with heuristics determine search performance.

Before Harvey's work, goals were assumed to be uniformly distributed. Harvey discovered that, at least for scheduling problems, this assumption is wrong and that a much better assumption is that the mistake probability is constant. We will see that if solutions cluster, the mistake probability tends to be constant. There is evidence that solutions tend to cluster in some decision problems [24, 44].

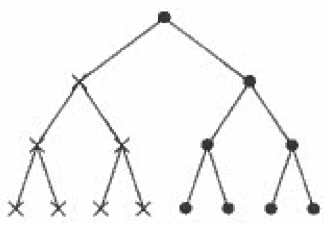


FIGURE 1. Easy problem turns hard for DFS.

2.1.1 Mistakes

Following Harvey [24], we define mistakes and the mistake probability as follows. A *bad* node is a node that does not have any goals in the subtree beneath it. A *good* node is a node that is not bad. A *mistake* node is a bad node whose parent is a good node.

Different search procedures are affected differently by mistakes. Figure 1 illustrates that mistakes can be costly for depth-first search. Good nodes are labeled with dots and bad nodes are labelled with x's. If the first node is a mistake, DFS explores the whole left subtree before finally reaching a goal. Mistakes are costly when they have large subtrees below them.

The *mistake probability*, m_k, is defined as the likelihood that a randomly selected child of a good node at depth k is a mistake node. Harvey [24] studied how this mistake probability changes with depth when goals were randomly distributed. The following summarizes his findings.

The mistake probability m_k is just the ratio of b_{k+1}, the number of bad nodes with good parents at depth $k+1$, to $b_{k+1} + g_{k+1}$, the total number of nodes with good parents at depth $k+1$. That is,

$$m_k = \frac{b_{k+1}}{b_{k+1} + g_{k+1}}. \tag{2.1}$$

In a complete binary tree, each good parent has exactly two children, so the number of bad and good nodes with good parents at depth $k+1$ is twice the number of good nodes with good parents at depth k, or

$$b_{k+1} + g_{k+1} = 2g_k. \tag{2.2}$$

It is easy to see how the number of good nodes grows with depth using (2.1) and (2.2):

$$\frac{g_{k+1}}{g_k} = \frac{2g_{k+1}}{b_{k+1} + g_{k+1}} = \frac{2(g_{k+1} + b_{k+1})}{b_{k+1} + g_{k+1}} - \frac{2b_{k+1}}{b_{k+1} + g_{k+1}} = 2(1 - m_k).$$

The number of good nodes, then, grows as $2(1-m_k)$. Figure 2 shows how m_k grows with depth. The graphs were generated as follows: Assume there are g goal nodes in a tree of depth d. Since goals are uniformly distributed, the probability that a leaf node is a goal is just $p = \frac{g}{2^d}$.

The height of a child's subtree at depth k is $d-k-1$. There are 2^{d-k-1} fringe nodes in that subtree. Given probability p that a fringe node is a goal node, the probability x_k that a child's subtree has no goals, implying that the child is bad is $(1-p)^{2^{d-k-1}}$. This is because the probability that a fringe node is not a goal is $1-p$ and none of the 2^{d-k-1} fringe nodes may be goals. On the other hand, the probability that the node is good is $1 - x_k = 1 - (1-p)^{2^{d-k-1}}$.

If goals are assigned independently, the probability that one child is good and another child is bad is the product of the individual probabilities. We have

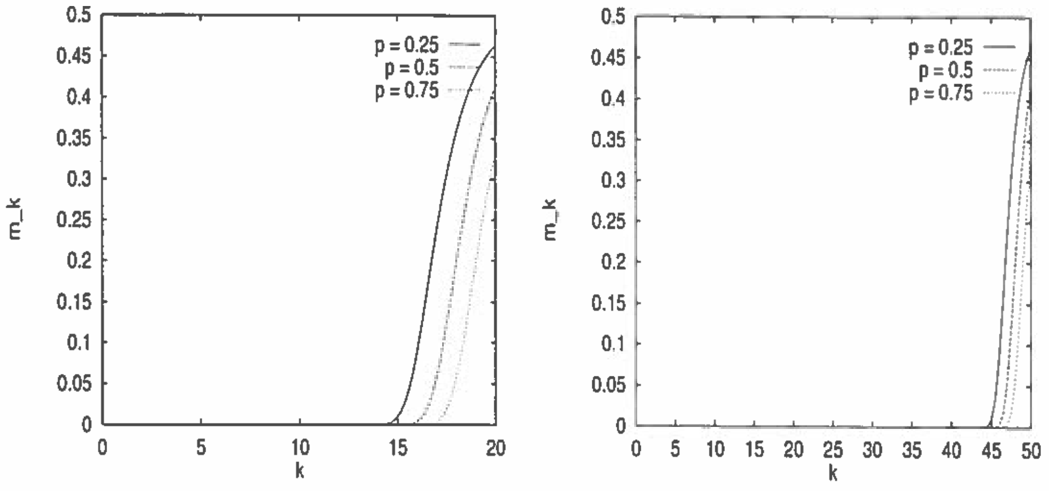


FIGURE 2. Mistake probability with randomly distributed goals.

just established that x_k is the probability that a node is bad. The probability that one node is good and its sibling bad is $x_k(1 - x_k)$ and the probability that both are good is $(1 - x_k)^2$. Since the mistake probability is the ratio of bad nodes with good parents to all nodes with good parents, we just need to calculate each of these quantities. The probability of there being one bad node (with a good parent) is twice the probability that a node is bad and its sibling is good, or $2x_k(1 - x_k)$ (one time the bad node may occur in the left, another time on the right). The probability that there is one bad node or that both nodes are good is $2x_k(1 - x_k) + (1 - x_k)^2 = 1 - x_k^2$. Thus, the mistake probability is $\frac{x_k(1-x_k)}{1-x_k^2}$.

Figure 2 shows m_k for trees with depth 20 and 50 and $p = 0.25$. The graphs show a mistake probability very close to zero for depths less than $d - 5$ increasing to about 0.47 between depths $d - 5$ and d. This behavior is not characteristic of real problems, however.

Studies in real search trees have proven that the mistake probability is actually fairly constant [24]. Harvey found evidence that scheduling problems have large failed subtrees. Trees of depth 40 with 40,000 nodes were found without solutions in the Sadeh suite of job-shop scheduling problems [52]. Harvey also concludes that for scheduling problems with and without heuristics, the assumption that the mistake probability is constant, while not exact, best approximates actual behavior.

The mistake probability is affected by the distribution of goals. If goals are randomly distributed, we saw that the mistake probability tends to be zero until the last few levels of the search tree. Now, imagine that there is a single goal. Figure 3 shows that the mistake probability is constant at all depths if there is a single goal. This is because trees underneath mistake nodes do not affect the mistake probability. For a single instance of a binary tree, either the left branch leads to a good node or the right branch does, so m_k is equal to 0.5. Imagine now that solutions were clustered underneath a single path. Figure 4 shows that the mistake probability is also constant if solutions lie in a single cluster. As long as solutions lie in clusters, it is much safer to assume that the mistake probability is constant than the mistake probability is zero until the last few levels of the tree. The mistake probability is 0.5 because subtrees with no bad nodes do not affect m_k.

Parkes [44] first observed that solutions tend to cluster on random satisfiability problems with at most three variables in each clause. These problems are commonly referred to as random 3-SAT. The parameters of this class of problems are the total number of variables n, and the number of clauses (or constraints) c. The c clauses are generated independently by randomly picking 3 variables and negating each variable with probability 50%. The point where instances change

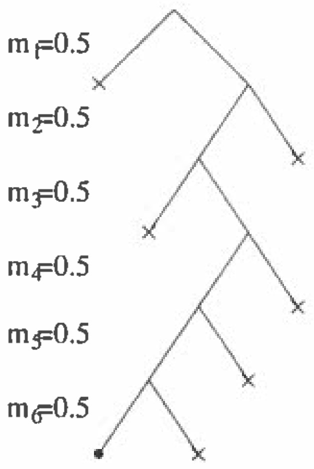


FIGURE 3. Mistake probability is constant if there is a single solution.

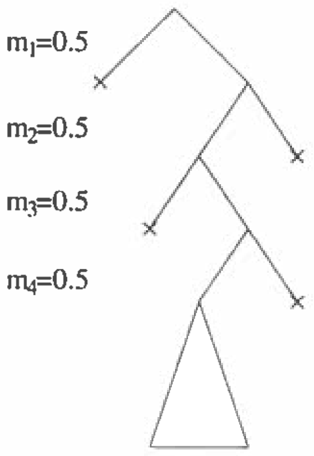


FIGURE 4. Mistake probability is constant if solutions cluster.

from being almost certainly satisfiable to being almost certainly unsatisfiable has been subject of extensive research [8, 43, 62, 31, 55]. This point is interesting because it lies in the middle of a region where instances transition from being generally unsatisfiable to being generally satisfiable and this point tends to be associated with a peak in the search cost in real problems [26].

Problems can be hard to solve because they either have few solutions or because the solutions are clustered away from the heuristic. Parkes found that as we cross into the unsatisfiable phase, a large class of instances emerges where all the models lie in a single exponentially large cluster. These "single cluster" instances are hard to solve.

Besides random 3-SAT and scheduling problems, there are also results that indicate that solutions also tend to cluster in traveling salesman problems (TSP) [6]. In a TSP, a set of cities and their distance matrix is given and the objective is to find the shortest tour so that no city is visited more than once and all cities are covered by the tour. The tour has to start and end in the same city. Boese [6] found evidence of clusters of low cost tours. He called these regions "big valley" regions because they tend to have exponentially many low cost solutions. These regions have also been studied elsewhere [63].

2.1.2 Heuristics

If a choice point has a goal node in the subtree below it, then with probability p its preferred child contains a goal node in its subtree. This probability p is the *heuristic probability*. Together, m and p describe a search space. A random heuristic will have probability $p = 1-m$, since m is the probability that a randomly

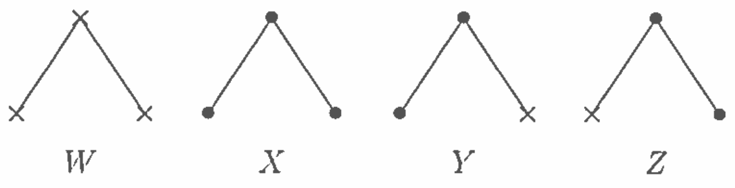


FIGURE 5. The four possibilities for good and bad node distribution.

selected child is a mistake. A heuristic that does better than random selection will have probability $p > 1 - m$.

Harvey found that, for good heuristics [9, 56] on standard job-shop problems [60], 9 out of 10 times the preferred choice would lead to a goal: That translates to a success rate close to 1.0 (0.9^d).

How do p and m determine a search space? For simplicity, we illustrate the concepts using binary trees. The analysis, however, is also valid for trees with larger branching factors.

Figure 5 shows the four possible ways good and bad nodes may be distributed. When building a search tree, each possible configuration can appear with a certain probability. This probability is a function of p and m.

The heuristic probability is the probability that the preferred child of a good node is good. Assuming nodes are heuristically ordered left-to-right, the heuristic probability p equals the probability that X or Y occurs given that W does not occur or

$$\mathrm{pr}(X \vee Y \mid \neg W) = p. \tag{2.3}$$

Since m is the probability a randomly chosen node is a mistake, the probability that only one node is good given its parent is good equals twice the mistake probability, or

$$\mathrm{pr}(Y \vee Z \mid \neg W) = 2m. \tag{2.4}$$

Since given that the parent is good, either one of the children is good or both are,

$$\mathrm{pr}(X \vee Y \vee Z \mid \neg W) = 1. \tag{2.5}$$

If goals are assigned independently, we can conclude that the probability that both children are good is one minus the probability that only one is good, or

$$\mathrm{pr}(X \mid \neg W) = 1 - 2m. \tag{2.6}$$

Since event Z occurs only when the goal is located underneath the right child and this happens whenever it is not underneath the left,

$$\mathrm{pr}(Z \mid \neg W) = 1 - p. \tag{2.7}$$

Finally, from (2.5), (2.6), and (2.7) it follows that

$$\mathrm{pr}(Y \mid \neg W) = p + 2m - 1. \tag{2.8}$$

Notice that the probability that a single child is good is $2m$ and the probability both children are good is $1 - 2m$. It follows that the expected number of goals is determined from m by $(1(2m) + 2(1 - 2m))^d = (2 - 2m)^d$. Additionally, we know that:

$$\begin{aligned} \text{pr(left is good)} + \text{pr(right is good)} &\geq 1 \\ p + 2 - p - 2m &\geq 1 \\ 2 - 2m &\geq 1 \\ 0.5 &\geq m \end{aligned} \tag{2.9}$$

2.1.3 Generating Random Trees

Random trees can be generated inductively using (2.3) through (2.8) given p and m. For instance, Figure 6 shows three possible tree configurations and the corresponding probability of occurrence of each configuration. The probability of occurrence of a particular tree is the product of the probability of occurrence of each of the configurations X, Y, or Z. Let us focus on the first tree. The root node has a good child on the left and a bad child on the right, so the probability of occurrence of that particular configuration is $p + 2m - 1$, according to (2.8). Since the subtree on the right has no good nodes, that subtree does not affect the probabilities. Thus, we consider only the left subtree. The top choice of that subtree has a bad node on the left and a good node on the right. The probability of occurrence of this configuration is $1-p$ according to (2.7). The choice on the left is a mistake so we only consider the choice on the right. That also has a probability of occurrence of $1 - p$. The probability of occurrence of the whole tree is the product of the occurrences of each of its good subtrees, or $(p + 2m - 1)(1 - p)^2$. When generating random trees, each of the trees in the figure will occur with its corresponding probability.

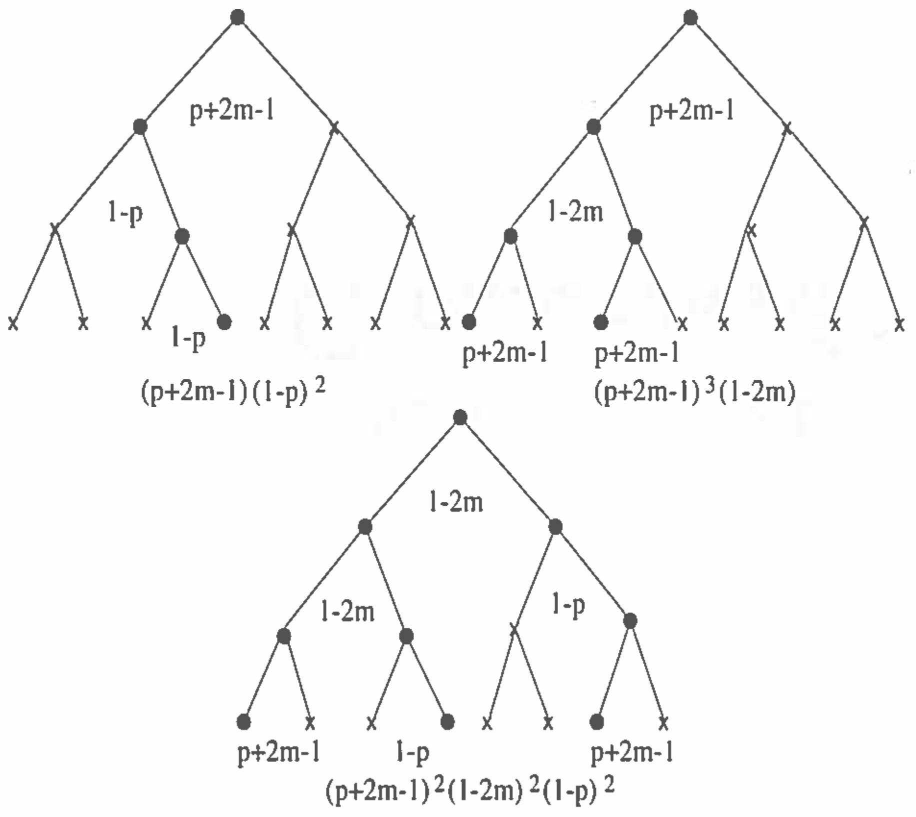


FIGURE 6. Sample trees and their probability of occurrence.

2.1.4 Estimating Parameters of the Search Space

There are several methods to profile search trees for size in terms of depth and branching factor. When confronted with small search trees it suffices to traverse the tree and record average depth and branching factor. However, if the tree is too large, sampling methods [32, 48] must be used.

The parameters p and m can be determined, if trees are small enough, by traversing the whole tree and recording which nodes are good and which are bad at each level. The parameter p is the ratio of configurations where the left node is good to all configurations and m is one half the ratio of configurations with only one good node to all configurations. Unfortunately, trees grow exponentially and it is often impractical to traverse the whole tree in reasonable time. Even if we could, the quantities p and m would indicate properties of the particular tree and not of a set of similar instances.

Harvey [24] concluded that if m is constant, sampling any portion of a search tree is as good as sampling any other portion of the search tree regardless of the path to the root. One way to profile for parameters of the search space is to follow a path to some node at depth k and exhaustively search the subtree below that node. It is important to choose k so that enough samples can be made for the results to be statistically significant. Harvey used bounded backtracking search (BBS) to do his profiling. BBS follows a random path to some node at depth l where l is the backtrack bound and then exhaustively searches the subtree underneath that node. Furthermore, sampling only works when the problems are fairly easy, since subtrees with no goals provide no further information on p and m. Harvey suggests that sampling on successively more relaxed versions of the problem may provide information on how p and m vary across depth. A more relaxed version of the problem can be obtained by imposing less constraints on the variables. This can be achieved, for example, by removing some constraints. If p and m vary predictably with depth across the different versions of the problem, a function may characterize p and m in terms of the number of constraints and the results may be extrapolated to the original problem.

2.2 Limited Discrepancy Search

When carefully tuned heuristics are available, the first logical step in finding a solution to a large search problem is to follow the heuristic path to a leaf. When this fails to lead to a solution, either the heuristics are modified or a search method has to be invoked. Repeatedly modifying the heuristics until they lead to a solution or time and patience are exhausted is known as 1-SAMP. Falling back to a search method is arguably more efficient.

Since solutions are only found at the leaves, the objective is to search as many leaf nodes as possible. The algorithm that explores leaf nodes as quickly as possible is depth-first search. DFS explores the heuristic path first, chronologically backtracking until a solution is found. If the leaf nodes of the search tree are heuristically ordered, DFS will explore them left to right.

If the heuristic makes mistakes, and most of these mistakes are made near the fringe of the search tee, DFS is the algorithm of choice. Unfortunately, there is no reason to expect that mistakes will, in general, be confined to the later decisions. Harvey [24] observed that if a mistake is made early in the search, it can effectively trap DFS, causing it to spend all of its allowed run time exploring the failing subtree, without ever returning to a decision that actually matters. This is known as the "early mistakes" phenomenon [25].

2.2.1 Algorithm

Limited discrepancy search was developed in response to the observations that solutions to search problems typically involve only a small number of heuristic violations, and these violations often occur near the root of the tree [25]. LDS searches nodes in increasing order of discrepancies: Searching the heuristic path first, then all paths with 1 discrepancy, then all paths with 2, and so on, finally searching all paths with d discrepancies.

The notion of searching paths in order of increasing heuristic violations was first proposed by Basin and Walsh in an algorithm called left-first search [2]. They proposed left-first search in order to control term rewriting in theorem proving. This implementation, however, had exponential memory requirements. The large memory requirements were not a problem in theorem proving where trees are small and each node corresponds to a large amount of computation. In contrast, problems we are concerned with have large search trees and we assume each node requires only a small amount of computation.

The LDS algorithm is given in Figure 7. LDS iteratively calls LDS SEARCH with increasing `cutoff`. LDS SEARCH does a depth-first right-to-left traversal of the tree, limiting the number of discrepancies to `cutoff`. Figure 8 shows how node order in LDS compares to nodes order in DFS. It is easy to see that there is always at least one path that leads to a leaf node at any particular cutoff, namely, the one that deviates from the heuristic path as many times as the cutoff allows, but

```
LDS (node, d)
  for cutoff from 0 to d
      result := LDS SEARCH (node, cutoff)
      if result $\neq$ NIL return result
  return NIL

LDS SEARCH (node, cutoff)
  if GOAL (node) return node
  succs := EXPAND (node)
  if NULL (succs) return NIL
  if cutoff = 0 return LDS SEARCH (LEFT(succs), 0)
  else
    result := LDS SEARCH (RIGHT(succs), cutoff−1)
    if result $\neq$ NIL return result
    return LDS SEARCH (LEFT(succs), cutoff)
  return result
```

FIGURE 7. Limited discrepancy search.

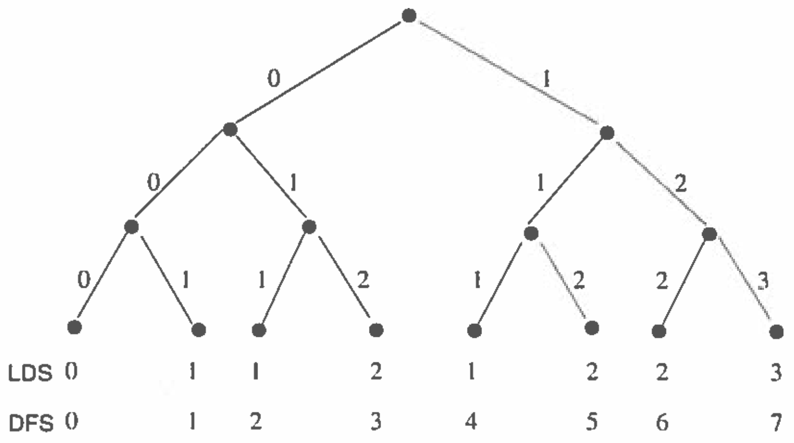


FIGURE 8. Node order in LDS vs. DFS.

follows the heuristic everywhere else.

Although LDS repeats the work of previous iterations, the overhead is negligible. In a tree of depth d, there is one path with zero discrepancies, d paths with one discrepancy and in general $\binom{d}{k}$ paths with k discrepancies (the number of ways of choosing k right branches out of d possible branches). Since each iteration is approximately d times the size of its predecessor, the cost of the last iteration dominates the sum of all previous iterations.[1] LDS explores leaf nodes slightly more slowly than does DFS, but if the heuristic makes early mistakes it explores a better set of leaves first.

2.2.2 Performance Results

Harvey [24] compared the performance of LDS with DFS and ISAMP [38]. ISAMP follows a random path to a leaf and it restarts from the root if it fails to find a goal node. He has shown theoretically and experimentally that LDS outperforms DFS and ISAMP by several orders of magnitude. He showed, theoretically, that it takes ISAMP 29,000 probes to get a probability of success of about 80%, while LDS requires only about 600 nodes if $m = 0.2$ and $p = 0.95$. If the heuristic orders successors randomly, however, then LDS does marginally worse than ISAMP and DFS outperforms both methods by a couple of orders of magnitude. Table 1 shows Harvey's results on a tree of height 30 with different values for p. The heuristic orders nodes randomly if $p = 1 - m$, so if $m = 0.2$, the heuristic is random at $p = 0.8$. Also notice that as the heuristic probability improves, $p > 1 - m$, the

[1]This is true for early iterations only. The cost actually increases by $\frac{d^k}{k!}$. In general, Harvey argues that d is likely to be much larger than k, since in large search spaces LDS only gets to explore a few iterations before we run out of time. Thus, this is not a gross underestimate of the cost of the iterations, in practice.

performance of LDS improves dramatically. For $m = 0.2$ and $p > 0.9$, for instance, LDS is the algorithm of choice. Similar results are obtained for deeper trees. Table 2 shows the results for a tree of height 100 and mistake probability of 0.1. In this case, it takes LDS about 2,000 nodes with $p = 0.975$ to reach a probability of success of 50%, while ISAMP searches 26,096 nodes on average to reach the same probability of success.

TABLE 1. Probability of success after 20 probes or about 600 nodes on a tree of height 30 with $m = 0.2$ and a solution density of 0.001.

Algorithm	p	Success Probability
ISAMP		0.03
DFS		0.35
LDS	0.85	0.12
LDS	0.9	0.4
LDS	0.95	0.85

TABLE 2. Probability of success after 20 probes or about 2,000 nodes on a tree of height 100 with $m = 0.1$ and a solution density of 0.000026.

Algorithm	p	Success Probability
ISAMP		0.01
DFS		0.12
LDS	0.925	0.03
LDS	0.95	0.21
LDS	0.975	0.82

Harvey also shows that combining LDS with bounded backtrack search (BBS) performs better than LDS alone. BBS probes the space randomly but after detecting an inconsistency, instead of returning to the root as ISAMP would, it chronologically backtracks until it reaches some backtrack bound, at which point it returns to the root and restarts. The combined LDS-BBS algorithm backtracks over failed discrepancies that fall within the backtrack bound. Therefore, LDS-BBS does not count discrepancies that fail quickly towards the discrepancy limit. LDS-BBS outperforms LDS because the backtracking helps recover from mistakes made later in the search. Adding a backtrack bound of l costs a factor of 2^l, but since l is usually small, the extra cost is almost negligible.

Harvey ran LDS and LDS-BBS on job-shop problems from a standard OR library.[2] He found that the best OR algorithms at the time [60] performed anywhere between 0.45% and 8.31% worse than optimal. LDS performed as well as the best OR algorithms but LDS-BBS with a lookahead of 4 reduced the percentage worse than optimal by about 1% on at least one problem.

The success of combining LDS with some form of backtracking led us to believe that there may be a more principled way to balance LDS probes with backtracking. In the following chapter we present a new algorithm called weighted discrepancy search that combines the advantages of LDS with the advantages of backtracking.

[2]The problem set is standard in Operations Research and can be found at http://mscmga.ms.ic.ac.uk/info.html.

CHAPTER III

WEIGHTED DISCREPANCY SEARCH

We begin this chapter by presenting a few definitions. Then, we discuss the intuitions behind weighted discrepancy search and justifications for the approach taken. We present WDS and show it working on an example. We end this chapter by showing that WDS can simulate 1-SAMP, LDS, and DFS given appropriate instantiations for the weights and the cutoff policy.

3.1 Definitions

We argued in the previous chapter that good heuristics make the correct decision most of the time. Imagine we are given a good heuristic that makes accurate decisions at the bottom of the search tree but it is almost random at the top. Since the heuristic is good, it will typically not rate the mistake much higher than the good choice, and therefore discrepancies near the top of the tree will often be rated as close choices by the heuristic. Choices further down the tree will typically be viewed as far clearer. We will call a discrepancy *strong* if the heuristic views it as almost as good a choice as its preferred sibling, and *weak* if the heuristic indicates that the choice is clear. LDS explores paths with single weak discrepancies before exploring paths with two or more strong discrepancies, even though the latter paths may well be more likely to contain solutions. Labelling marginally weaker nodes as discrepancies is likely to produce problems for LDS. The number of nodes explored grows roughly as a factor of d for early iterations.

For deep trees, a factor of d is a high price to pay for ignoring the signal strength.

In order to design the most adequate heuristic for a particular problem, the programmer must consider all the information available at the time of the search. Local heuristics produce a signal according to how good a node is with respect to all its siblings. In other words, local heuristics compute signals using information local to each decision. Heuristics used in scheduling measure the impact of deciding a particular ordering for a task versus all other possible orderings for that task. Thus, scheduling heuristics are typically local heuristics. Global heuristics, on the other hand, use information obtained by looking ahead in the search or doing additional reasoning at each decision point. Although global heuristics tend to make less mistakes than local heuristics, they are often impractical because they are very expensive. One would think that with increase in processor speed, global heuristics would replace local heuristics. However, Ginsberg [16] has argued that in at least three cases: game-playing, planning, and scheduling, where AI has made recent progress, human insight has been applied globally to reduce the problem to one in which search with cheap heuristics is effective. Thus, in this dissertation, we will deal exclusively with the issues surrounding local heuristics.

3.2 Effects of Strong Discrepancies in Backtracking.

Backtracking algorithms are brittle in the presence of strong discrepancies. In both DFS and LDS, a choice that is marginally worse will be explored much later in the search than a choice that is marginally better. Imagine the preferred choice is a mistake and the other choice is rated as marginally worse by the heuristic. DFS will unsuccessfully search the whole subtree underneath the mistake node before recovering from the mistake. If that subtree is large, it will waste all of its search effort. LDS will recover more gracefully, but it will still search all paths in that subtree with discrepancy count less than the discrepancy count of the marginally worse choice.

1-SAMP is equally vulnerable. If the probe fails, the programmer will be forced to re-engineer the heuristic even if the heuristic only made a handful of mistakes. Without doing any search though, the programmer has no means of acquiring an intuition about how accurate the heuristic is. Tailoring heuristics to a specific problem is not easy and sometimes requires extensive testing or an extremely well-developed intuition for what is going on. Acquiring this intuition may take longer than searching in the neighborhood of heuristic choices.

Two choices with similarly strong signals should be explored at roughly the same time, since they are both comparably likely to lead to a solution. Because a strong signal indicates that the node is likely to lead to a goal, choices with strong signals should be explored before choices with weaker signals. But, given a signal, how do we determine what is the relative likelihood of goodness of the discrepancy with respect to the preferred choice? Here, by goodness we mean a measure of the likelihood to lead to a goal. Is signal strength alone enough to measure the probability of success of a choice or can we do better?

3.3 Weights

Since signal strength is critical for good performance, we want to study how weights can be assigned to choices in order to improve on the performance of LDS. First, we want to find out if using the signal directly as the weight is likely to

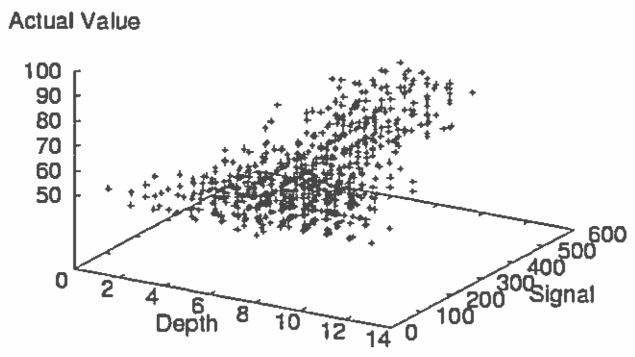


FIGURE 9. Signal strength correlates with actual value across different depths.

improve on LDS. The weights calculated in this way are only as useful as the signal is, so the first thing we need to do is verify that the signal correlates with the actual value of the choice.

In order to study the behavior of the signal, we took a standard job-shop scheduling problem from the literature.[1] In order to study the problem completely, we took several different subsets of the problem and looked at the search trees produced by these subproblems. The heuristics we used were the same heuristics used by Crawford in combination with LDS to successfully solve resource constrained project scheduling problems [10].[2] One thing we found is that the heuristic be-

[1]The problem suite is standard in Operations Research and can be found at http://mscmga.ms.ic.ac.uk/info.html.

[2]We found that these heuristics have also proven successful in producing state-of-the-art solutions to job-shop scheduling problems.

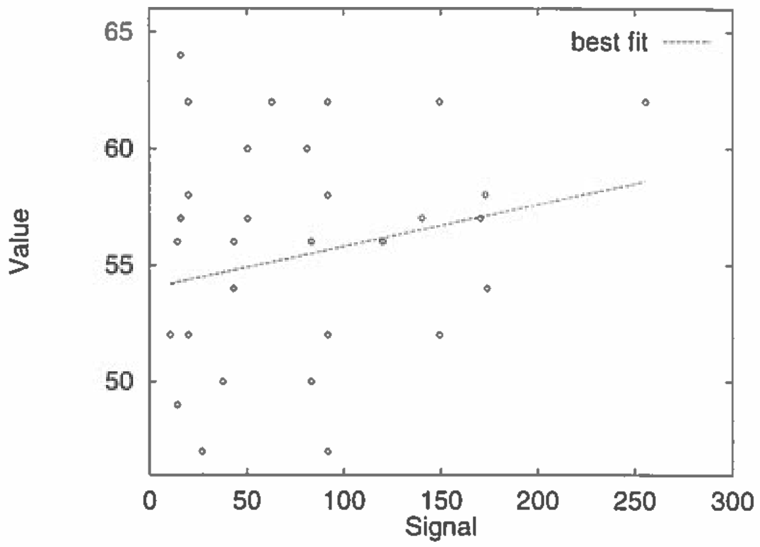


FIGURE 10. Linear correlation of 0.22 between signal and actual value at depth 5.

haved similarly in the different subsets of the same problem. The behavior of the signal of a characteristic subset is shown in Figure 9. The figure shows the behavior of the signal strength across different depths, the "actual" value of a choice—the makespan of the schedule that results from making that choice. The signal corresponds with the expected makespan assuming the rest of the conflicts are resolved. Each data point represents a choice in the search tree. The first thing to notice is that the higher the signal the higher the actual value (represented in the z axis). This means that the signal is correlated with the actual value of the choice. Figure 10 through Figure 13 show the slices at depths 5 through 8 with the linear correlation between signal and value improving from 0.22 to 0.63.

The next thing to notice is that the density of the data is larger at depths between 5 and 10 than near the top or near the bottom of the tree. This is obvious because the number of nodes increases exponentially as depth increases and the

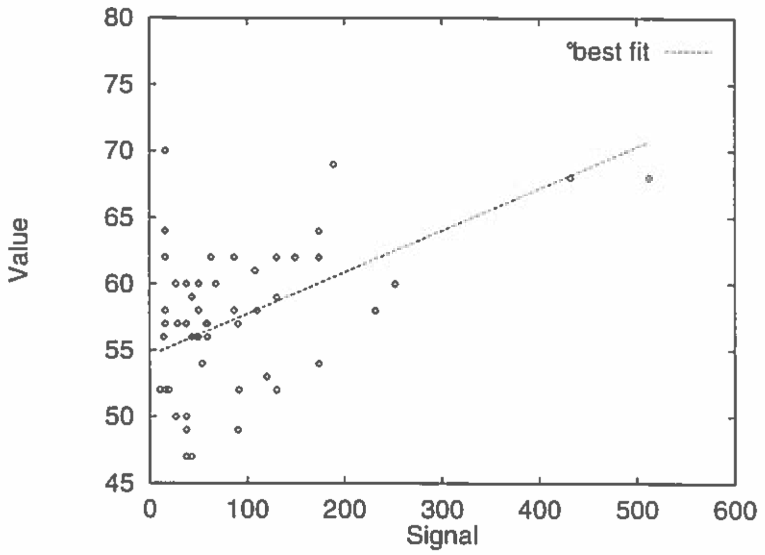


FIGURE 11. Linear correlation of 0.55 between signal and actual value at depth 6.

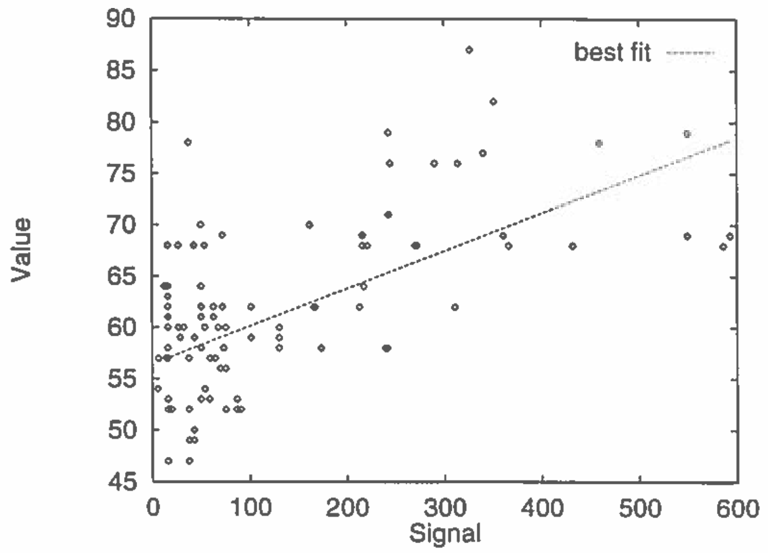


FIGURE 12. Linear correlation of 0.61 between signal and actual value at depth 7.

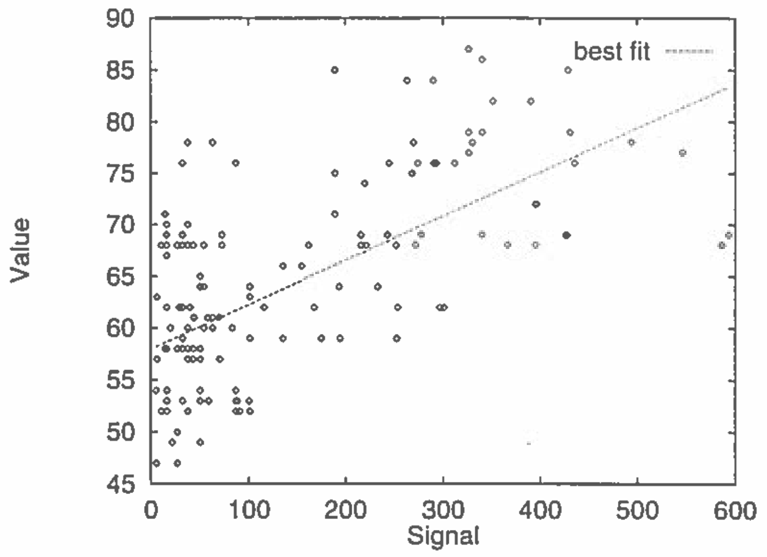


FIGURE 13. Linear correlation of 0.63 between signal and actual value at depth 8.

trees are non-uniform. There are leaf nodes anywhere between depths 8 and 13. What is not clear from the figure is that for particular values of depth and signal strength the points are spread across different values in the z axis (representing the actual values). In other words, the signal is the same for those points even though the actual value is different. We will see that the weighting scheme we propose correlates better than signal does with the actual value.

Typically in constraint satisfaction, future decisions depend on previous ones. As choices are made, the set of possible assignments becomes more constrained and the set of goal nodes reachable from the most recently explored node is a subset of all possible goals. Think of the weight of a node not as a probability measure per se, but as a measure of the likelihood that a goal is found in the subtree beneath it. As choices are made, and the search space is reduced, there are less nodes that remain as candidates for goals. Thus, children are less likely than (or as likely

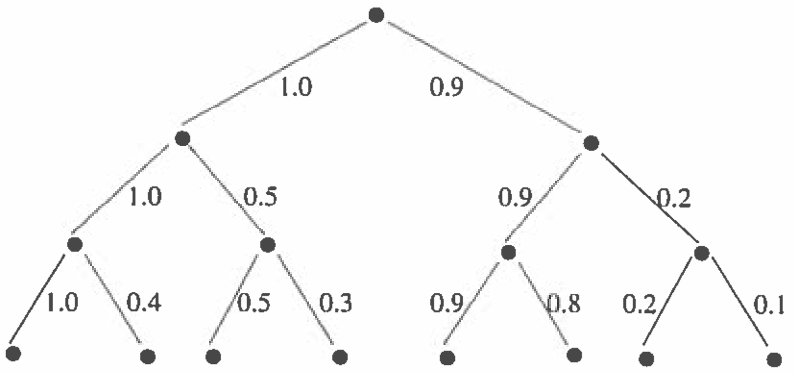


FIGURE 14. How weights are assigned in WDS.

as) their parents to lead to a goal. Do not confuse this likelihood measure with a probability measure that the node is a goal. The probability of success may increase as we descend the tree, in particular if we are heading towards a goal node. But this probability measure can only increase if there is knowledge that the path leads to a goal node. In the presence of uncertainty, all we can assure is that in making choices we have ruled out other nodes that could potentially lead to goals. Thus, we multiply the weights down the tree.

We require weights to reflect the strength of a discrepancy. The heuristic choice should always have the higher weight and the other choices should have a weight that corresponds to how likely they are to have a goal underneath relative to the heuristic choice. One way to obtain "relative" weights is to normalize the signal. Normalizing the heuristic value of both children with respect to the most promising child will achieve two things. First, it will identify the preferred choice and show how much more likely this choice is to lead to a goal versus making the other choice. Second, it will label the preferred child with a 1, but since weights are propagated downward, the preferred child will inherit its parent's weight. This

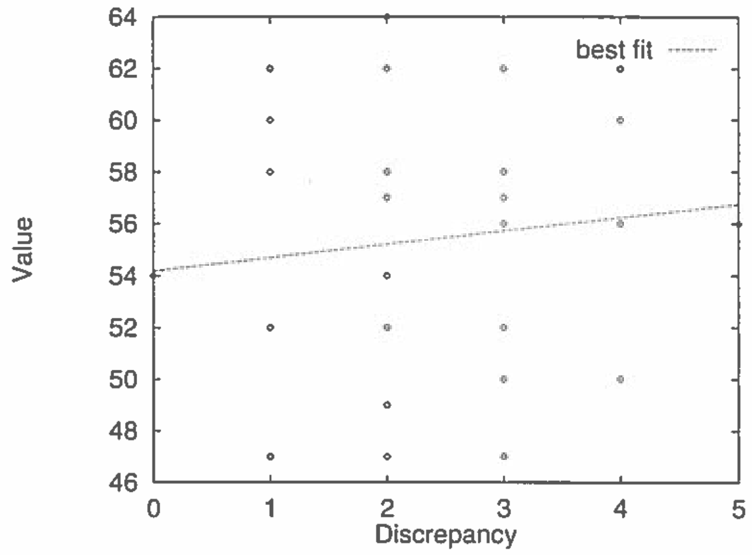


FIGURE 15. Correlation between discrepancy and exact value of a choice is 0.11.

means that each time WDS reaches an interior node, it will reach a leaf node, since there will be a path to a leaf with the same weight.

Two choices with similar signal strength will have similar weights. For instance, in Figure 14, the first two choices are rated as close by the heuristic. The first discrepancy is an example of a strong discrepancy. The other two single discrepancies, at depths 2 and 3, are examples of weak discrepancies.

We have claimed that the weight should correlate better with the exact value of a choice than the signal produced by the heuristic. In order to show that this claim is true, we calculated the correlation between discrepancy count and exact value of a choice for choices in job-shop scheduling problems. Figure 15 shows the correlation between discrepancy count and the exact value for choices at depth 5 for the same job-shop scheduling problem we mentioned at the beginning of this section. The figure shows that the discrepancy count is not always correlated

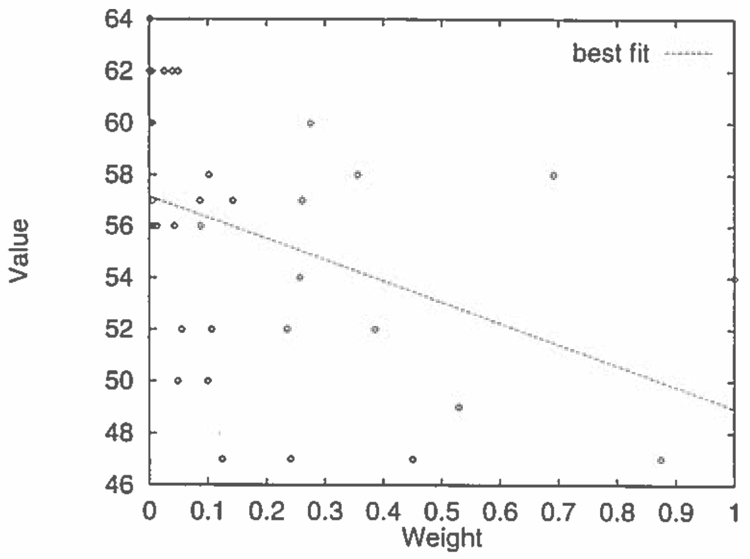


FIGURE 16. Correlation between weight and exact value of a choice is -0.42.

with the exact value. A correlation of 0.11 too weak to be considered other than random. Notice that the slight correlation is positive. This is due to the fact that shorter schedules appear to lie in paths with fewer discrepancies.

On the other hand, Figure 16 shows weight tends to correlate better. This figure shows the weights calculated from the signal for the same problem at depth 5. Observe that weights result in a negative correlation with the exact value. This is due to the fact that in scheduling the better choices have shorter makespan (length of the longest path of jobs). The better the choice, the shorter the makespan, the higher the weight. Thus, weights are negatively correlated with exact value.

Since weights are better correlated with actual value than discrepancy count, searching in order of decreasing weights should improve our chances of success.

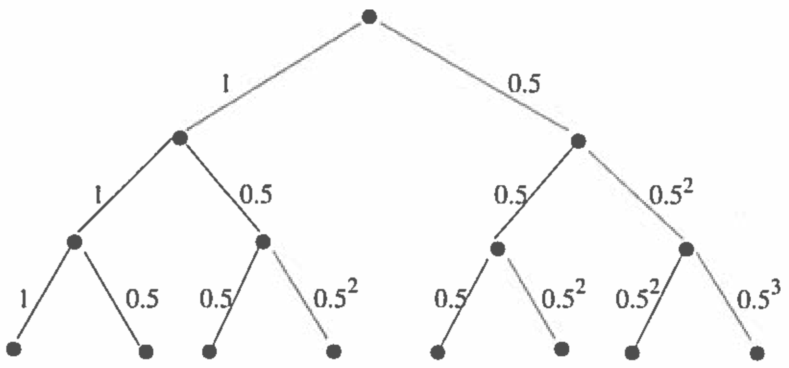


FIGURE 17. How weights would be assigned in LDS.

3.3.1 Weights in LDS

WDS is a generalization of LDS, and weights can be assigned so as to simulate discrepancy counts. Discrepancy counts and weights share the same properties: While in LDS discrepancy counts do not decrease as we descend the tree, in WDS weights do not increase. In LDS the preferred child has the same discrepancy count as the parent, and in WDS it has the same weight. The only difference is in the non-preferred child. In LDS the non-preferred child is labelled as a discrepancy. The discrepancy "weighs" the same regardless of the depth at which it occurs. Thus, it suffices to assign a constant weight to each discrepancy. Assume the normalized signal is 1 for the heuristic choice and w for the other choice. The weight of the non-preferred choice will be w times the weight of the parent. If there is only one other discrepancy in the path to the parent, the node's weight will be w^2. That is, each node is assigned a power of w.

Figure 17 shows how weights are assigned to simulate LDS. In this particular example, the weights are assigned to reflect the fact that the choice on the left is exactly twice more likely as the choice on the right to lead to a goal. Signal strength however, is unlikely to be uniform across depths, and no matter what value we pick for w, it is never going to reflect the true signal distribution. It seems unlikely to be the case that all discrepancies are rated as equal by the heuristic.

Consider building a tree from top-to-bottom using the discrepancy counts as node labels. At depth 1, there is one node at 0 discrepancies and one node at 1 discrepancy. At depth 2, there is one node at 0 discrepancies, two nodes at 1 discrepancy and one node at 2 discrepancies. It is easy to see a pattern emerging. Append the sequence obtained by adding one to $(0,1)$ to the end of the $(0,1)$ sequence. This results in the sequence $(0,1,1,2)$, exactly the discrepancy distribution (or exponent of the weight) for depth 2. The distribution of discrepancy counts of nodes at depth 3 can be obtained in a similar fashion. First we copy the sequence at depth 2 $(0,1,1,2)$. Then we add one to the sequence to obtain $(1,2,2,3)$. Finally, we append the new sequence to the original one to obtain $(0,1,1,2,1,2,2,3)$. Intuitively, if we are building the tree from the bottom up, a new level is formed by copying the tree structure we have constructed so far, adding one to each discrepancy count appearing in the copy and placing the copy to the right of the original structure, and joining both structures by a new root node.

Viewed from the top down, one can identify the number of k discrepancies at depth d by counting the number of ways one can descend the tree by going right k times in at most d descents. This is the same as counting the number of ways one can choose k right branches out of a total of d possible right branches. This quantity is characterized by the binomial coefficients and is commonly written as $\binom{d}{k}$. The binomial coefficients appear in the binomial probability distribution:

$\binom{n}{x}p^x q^{n-x}$, where $p^x q^{n-x}$ indicates the probability of each individual sequence of $\binom{n}{x}$. In a binary tree, the probability that there are $\binom{d}{k}$ discrepancies is 1 for each $0 \leq k \leq d$ and 0 otherwise. Thus, the distribution is characterized by the choose term.

It is well known that the binomial distribution tends to approximate to a normal distribution [22]. This fact is especially true when the n, the top of the binomial coefficient, is very large and when np and nq are both equal to or greater than 10. In the case of a binary tree, this means that if the tree has depth greater than or equal to 10, the number of discrepancies distributes according to the normal distribution, with mean d and standard deviation $\sqrt{d}$.

Although LDS weights themselves do not distribute according to the normal distribution, their logarithms do. This is because the logarithm of weights encodes the discrepancy count of the choice. As seen in the construction above, the only parameter of this distribution is depth. Let us now examine the weight distribution in WDS.

3.3.2 Weights in WDS

So far, we have seen that the signal in scheduling problems tends to correlate better with actual value of a choice as depth increases. We have also argued that weights correlate with the actual value better than signals and discrepancy counts. In this section, we study the weight distribution. In order to study the weight distribution we decided to look at how the weights are distributed in the same job-shop scheduling problems we looked at in Section 3.3. Because we wanted to explore the whole tree and since we established previously that subsets of the same problem behaved similarly, we took one subset of each of the following three different problems:

1. mt06, from Fisher and Thompson [12]. It has 6 jobs that must run on 6 machines.

2. la39, from Lawrence [39]. This problem has 15 jobs that must run on 15 machines.

3. tail15x15, from the Taillard suite of problems [58]. It has 15 jobs that should run on 15 machines.

The descriptions of these problems can be found in Appendix A along with the descriptions of the subsets we took. For the purposes of the following discussion, we will refer to the subset of mt06 as Problem 1. The subset of la39 will be referred to as Problem 2, and the subset of tail15x15 will be Problem 3.

Figure 18 shows the weight distribution at the leaves for Problem 1 and Figure 19 and Figure 20 show distribution of weights at the leaves for Problems 2 and 3, respectively. The figures also show the best fit to the data. The best fits show that the weights also distribute lognormally [1]. The lognormal distribution is the following:

$$f(x) = \alpha(\frac{1}{\sqrt{2\pi}\sigma x}\exp(\frac{-(\log(x)-\mu)^2}{2\sigma^2}))$$

We found that different job-shop scheduling problems distribute lognormally with different values for the parameters of the distribution. The different values, however, seem to depend only on depth.[3] All of the parameters increase as the average depth of the problem increases. These changes can be explained by observing

[3]They probably depend also on breadth, but the problems we studied have branching factor 2.

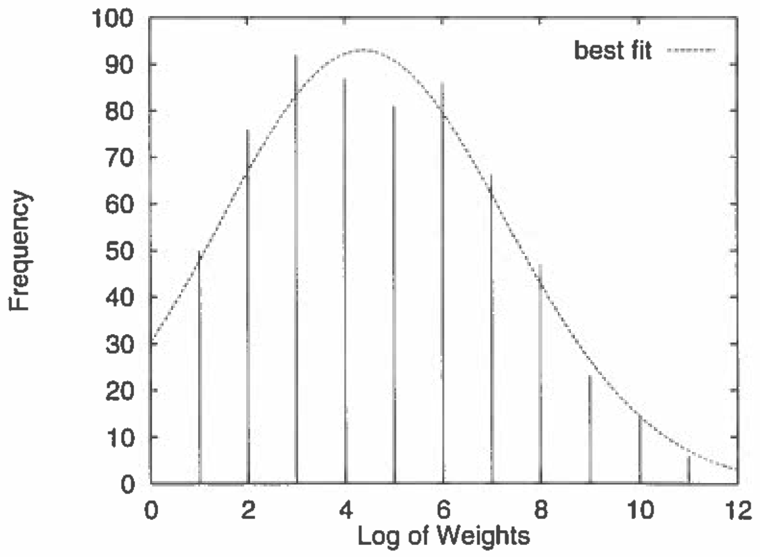


FIGURE 18. Lognormal distribution of weights with $\alpha = 3,820,903$, $\sigma = 2.92$, and $\mu = 12.9$ for Problem 1. This problem has average depth 10.

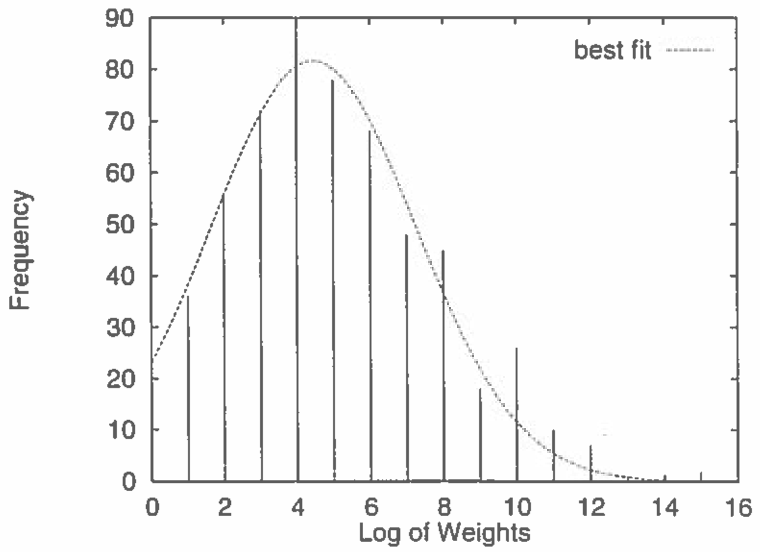


FIGURE 19. Lognormal distribution of weights with $\alpha = 2,524,988$, $\sigma = 2.81$, and $\mu = 12.32$ for Problem 2. This problem has average depth 9.

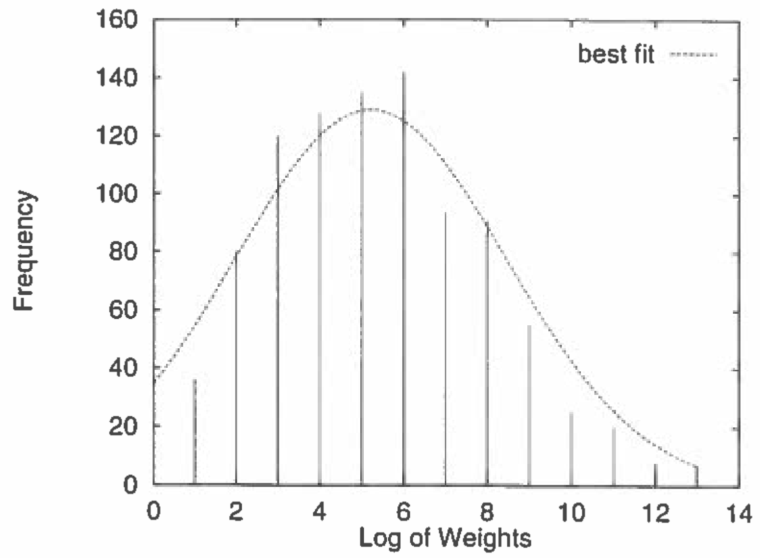


FIGURE 20. Lognormal distribution of weights with $\alpha = 34,123,422$, $\sigma = 3.22$, and $\mu = 15.58$ for Problem 3. This problem has average depth 12.

the distribution. If the distribution is approximated by $f(x)$, the total number of nodes in the tree is approximated by $\int_0^\infty f(x)dx$. As depth increases, so does the number of nodes in the tree. Thus, the fact that the number of nodes increases explains the increase in α. The mean of the distribution, given by $\exp(\mu + \frac{1}{2}\sigma^2)$ also increases with depth. The means for Problems 1, 2, and 3 are 7.44, 4.60, and 30.52, respectively. Hence, we believe it may be possible to extrapolate the weight distribution at large depths from a set of problems at smaller depths. Moreover, the subset problems need not be of the same instance since, as we have seen, the weight distribution seems to be tied to signal and problem class, rather than to a particular instance.

```
WDS (cutoff-policy)
   for cutoff in cutoff-policy do
       result := WDS SEARCH (root, 1.0, cutoff)
       if result $\neq$ NIL return result
   return NIL

WDS SEARCH (node, parent-weight, cutoff)
   if GOAL (node) return node
   succs := EXPAND (node)
   sorted-succs := SORT (succs)
   for child in sorted-succs
       weight := WEIGH (child, parent-weight)
       if weight $\geq$ cutoff
          WDS SEARCH (child, weight, cutoff)
   return NIL
```

FIGURE 21. Weighted discrepancy search.

3.4 Algorithm

In WDS, the weight captures the expectation that the choice contains a goal in the subtree underneath it (independently of its siblings). The only information the algorithm has about a node, at any particular time, is its signal strength, that of its siblings, and the weight of the parent. The weight of the node is calculated by normalizing its signal with respect to the best sibling and multiplying it by the weight of the parent. Weights are highest where the signal is strongest.

WDS is a recursive algorithm that iterates through the given cutoff policy exploring nodes with weight larger than or equal to the current cutoff in each iteration. The WDS algorithm is shown in Figure 21. WDS iteratively calls WDS SEARCH with each successive cutoff. WDS SEARCH does a depth-first traversal of the tree, exploring nodes with higher weight first and only nodes with weight

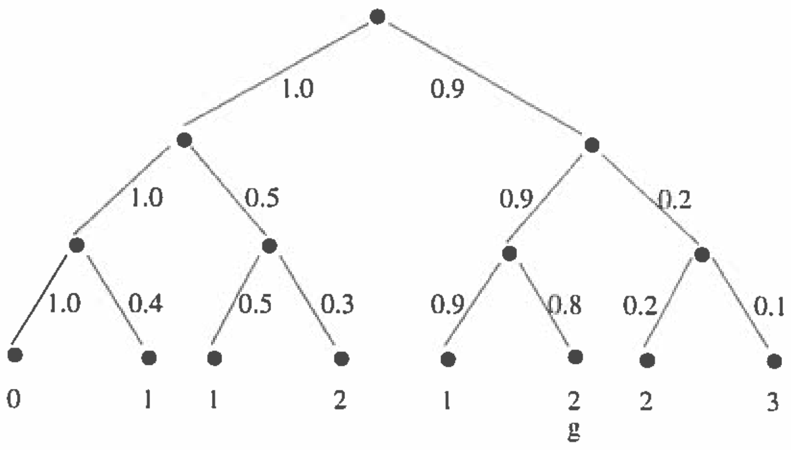


FIGURE 22. Comparing LDS to WDS on a small tree with a single goal.

larger than or equal to `cutoff`. SORT sorts nodes in decreasing order.

The first call to WDS SEARCH is made with a `parent-weight` of 1, so in the case of the heuristic path, every node on the path is valued 1. Since the best child is always assigned the same weight as its parent and search proceeds in depth-first order at least one leaf node is reached every time an interior node is expanded. Weight values do not increase as WDS descends the tree. Thus all leaf nodes with weight greater than or equal to the cutoff value are explored in each iteration. The cutoff policy is a set of decreasing values between 1 and 0, inclusive. WDS will search the whole tree when `cutoff` reaches 0.

3.5 Example

Weighted discrepancy search overcomes the limitations of LDS. Paths with stronger signal strength will be explored sooner than paths with weaker signal strength regardless of the discrepancy count. For instance, in Figure 22 the goal is found by LDS in the third iteration after exploring eleven leaf nodes, but WDS

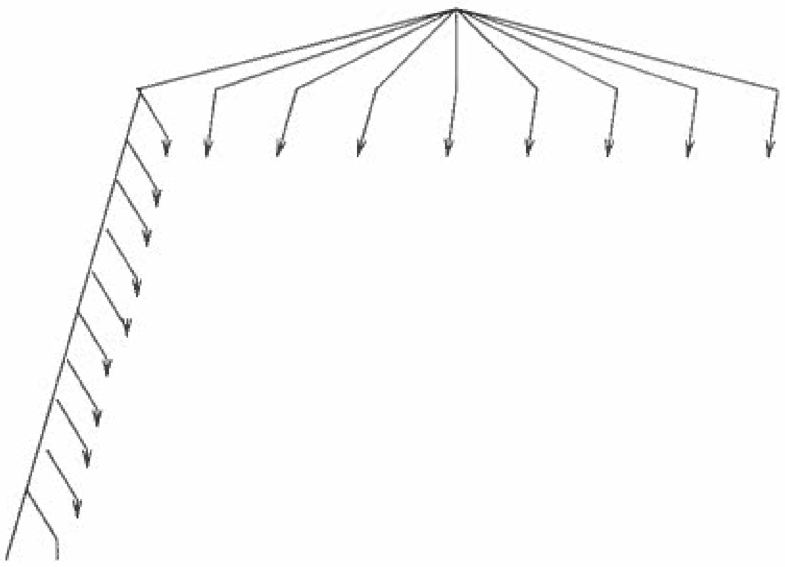


FIGURE 23. LDS explores all single discrepancies even in trees with large branching factors. The arrows represent probes following the heuristic to the leaf.

may reach the goal after only exploring three leaf nodes if it begins with a cutoff larger than 0.5 and smaller than or equal to 0.8.

In search spaces with larger branching factors, LDS will spend all of its allowed run time exploring all discrepancies regardless of whether these discrepancies are likely to lead to a goal or not. Figure 23 shows the branches that would be explored by LDS(1) on a tree with large branching factor on the first level. In trees with large branching factors, LDS may not get to explore paths with greater number of discrepancies but better chances of success. In trees with large branching factors, it is unlikely that all single discrepancies will have a signal that is equally strong. Some discrepancies will have stronger signal than others and we should explore discrepancies with strong signal before exploring discrepancies with weaker signals.

In contrast to LDS, WDS will spend its time exploring the stronger discrepancies regardless of their positioning in the tree. How soon WDS gets to the goal, however, depends on the cutoff policy. In this work, we assume the cutoff policy is chosen before the search begins. In this case, it is possible to choose a cutoff policy where each iteration explores only one new node each time. In the worst case, a policy such as $\{1.0, 0.99, 0.98, 0.97, ..\}$ may be chosen in which case WDS, running on the example in Figure 22, explores the heuristic path ten times before exploring any new paths. Of course, we can easily guarantee at run-time that the policy explores at least one new node each time, by storing the largest weight we have pruned and skipping cutoffs smaller than the current one and larger than this stored weight.

Because WDS assigns real-valued weights to the nodes, it is no longer clear which cutoff values should be part of the policy. In LDS there are only $d+1$ different weights, but in WDS the weight space is continuous. If the cutoff value decreases too rapidly between iterations, WDS will spend time exploring nodes with low weights before it explores more promising nodes. If the cutoff value decreases only slightly, WDS may end up searching very few additional nodes without significantly improving its chance of success.

3.6 Simulation of 1-Samp, Dfs, and Lds

Wds can simulate other algorithms by fixing the weights and the cutoff policy. Wds simulates LDS when the weights are powers of w for some value of w. All nodes in paths with one discrepancy have weight w, all nodes in paths with two discrepancies have weight w^2, etc. The cutoff policy $\{1, w, w^2, \ldots, w^d\}$ corresponds to the $d+1$ iterations of LDS.

WDS can simulate DFS as well. Since WDS explores nodes in a depth-first order within a cutoff, WDS will simulate DFS when the cutoff policy has a single cutoff that is smaller than all weights, for example, $\{0\}$. Similarly, 1-SAMP is simulated by setting the cutoff policy to $\{1\}$. This ensures that the heuristic path is the only one explored.

Since WDS can simulate LDS, DFS, and 1-SAMP, the expected cost of WDS with the optimal cutoff policy will be at most the cost of the best of these on any given search problem. The following chapter presents an average case cost analysis for all of these algorithms.

CHAPTER IV

AVERAGE CASE COST ANALYSIS

We established in the previous chapter that the performance of WDS depends on the cutoff policy used. In this chapter we present an average case cost analysis in terms of an arbitrary cutoff policy. Harvey [24] presents an analysis of LDS and DFS, but his analysis depends on the fact that the corresponding cutoff policies are known. In WDS the cutoff policy is exactly what we are trying to determine. Instead of extending Harvey's results directly, we are forced to find a different formulation. However, we obtain the same results in the cases where the policy is known. We present the analysis for LDS first and then extend the results to cope with WDS.

4.1 A Simple Case: Expected Cost of LDS

The expected cost to solution is given by adding the product of the number of nodes explored by the probability of success of each iteration. The probability of success is given by the probability of occurrence of each possible tree configuration. We already saw in Section 2.1.3 that it is possible to calculate the probability of occurrence of each possible tree configuration. All we have to do is collect all the probabilities for the trees where LDS(c) might succeed for each cutoff c, count the number of nodes LDS(c) will explore before it succeeds in each case, and compute the expected cost.

Figure 24 shows the possible trees for each iteration where LDS might succeed

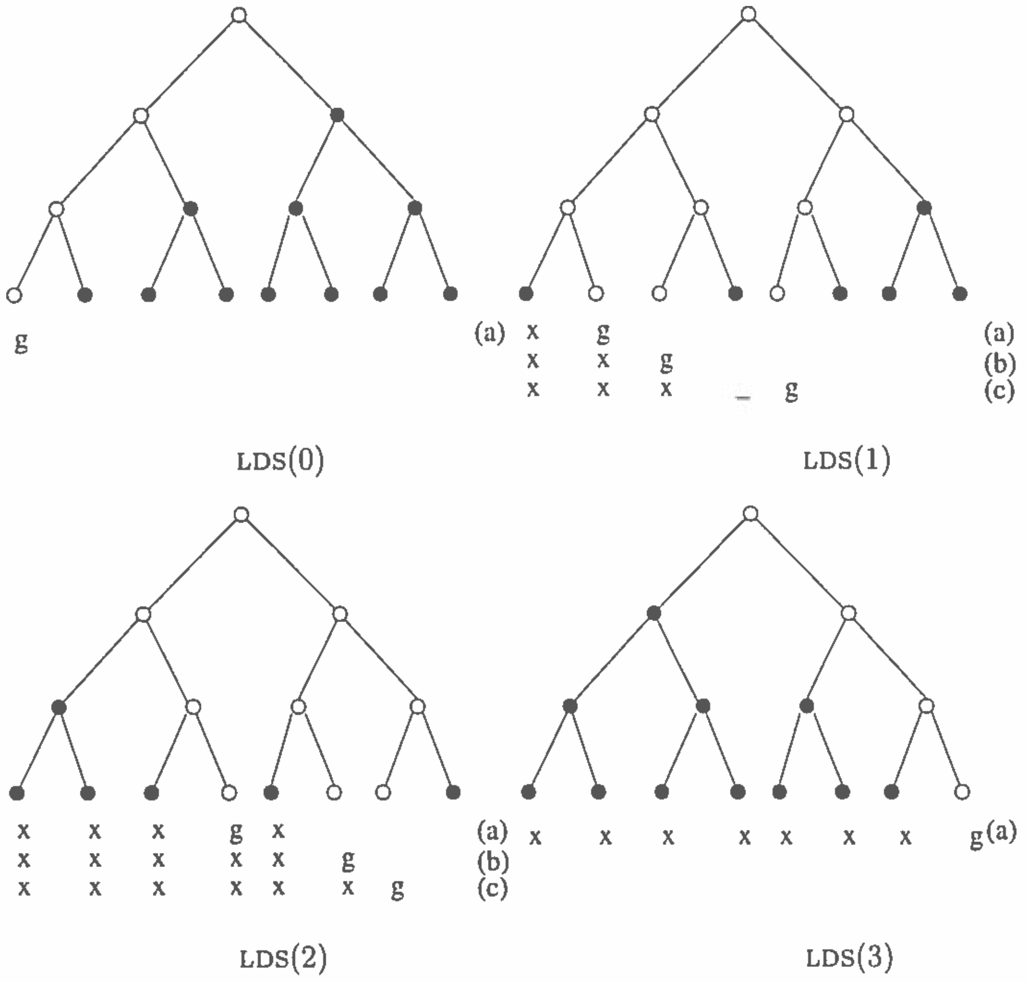


FIGURE 24. Probability that LDS succeeds in each iteration.

TABLE 3. Probability of success for LDS(0) through LDS(3).

LDS(0)	(a)	p^3
LDS(1)	(a)	$p^2(1-p)$
	(b)	$p^2(1-p)$
	(c)	$(1-2m)(1-p)^2p^2 + (1-p)p^2$
LDS(2)	(a)	$(p+2m-1)(1-p)^2 + (1-2m)(1-p)^2(1-p^2)$
	(b)	$p(1-p)^2$
	(c)	$p(1-p)^2$
LDS(3)	(a)	$(1-p)^3$

for a tree of depth three. A dash (-) denotes a don't care, where either a bad node or a good node may occur, an x denotes bad nodes and a g denotes good nodes. The unlabelled subtrees account for all possible subtree configurations and thus, the probability of occurrence is one. These subtrees play no role in the computation of the probability of occurrence of the tree. The nodes explored for the first time in each iteration are denoted by circles. For simplicity, in the following analysis we assume m and p constant, but the same formula applies if m or p are functions. Table 3 shows the corresponding probabilities for the trees in the figure.

Notice that the probability of success of LDS(1) is not $3p^2(1-p)$ as one might casually assume, since LDS(1) explores all paths with two left branches and one right branch. Similarly, the probability of success of LDS(2) is not $3p(1-p)^2$. The difference in the first case is that LDS(1) skips a node on the left subtree, which may or may not contain a goal (represented by the - symbol in Figure 24), and we must account for either case. In the second case, the difference is that we must account for the fact that the success of LDS(2) is conditional on LDS(1) failing on a node on the right subtree.

The expected cost to solution of LDS for a tree of depth 3 is given by the following equation:

$$
\begin{aligned}
EC(3,p,m) = \\
&4p^3 + \\
&9p^2(1-p) + 11p^2(1-p) + 14((1-2m)(1-p)^2p^2 + (1-p)p^2) + \\
&22((p+2m-1)(1-p)^2 + (1-2m)(1-p)^2(1-p^2)) + \\
&26p(1-p)^2 + 28p(1-p)^2 + \\
&43(1-p)^3 \qquad (4.1)
\end{aligned}
$$

For larger depths, one can approximate the expected cost to solution by averaging the probability of success across all nodes in the iteration. For illustration purposes only, let us calculate the approximate cost and see what is the difference between the exact and the approximate cost to solution. The approximate cost for a tree of depth 3 is given by:

$$
\begin{aligned}
AC(3,p,m) = \\
&4p^3 + \\
&(4 + \tfrac{1}{3}22)(2p^2(1-p) + (1-2m)(1-p)^2p^2 + (1-p)p^2) + \\
&(14 + \tfrac{1}{3}34)((p+2m-1)(1-p)^2 + (1-2m)(1-p)^2(1-p^2) + p(1-p)^2) + \\
&43(1-p)^3 \qquad (4.2)
\end{aligned}
$$

Figure 25 shows that the approximate cost (4.2) is a slight underestimate

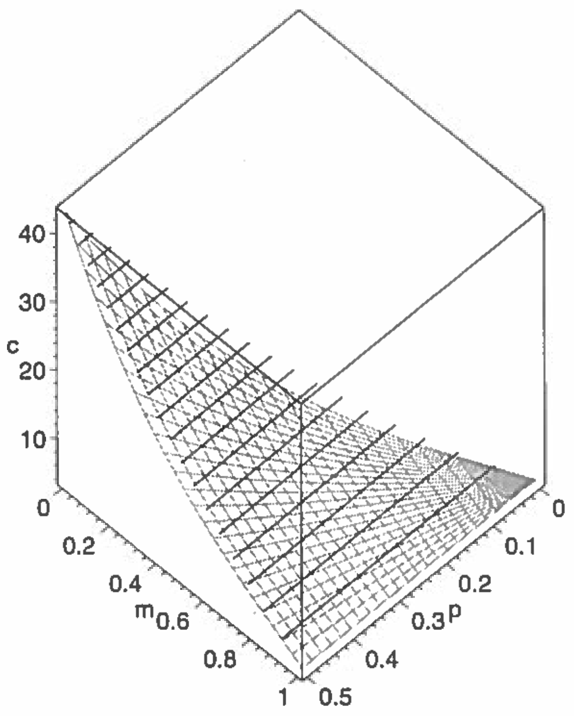


FIGURE 25. Approximate cost underestimates the exact cost for a tree of depth 3.

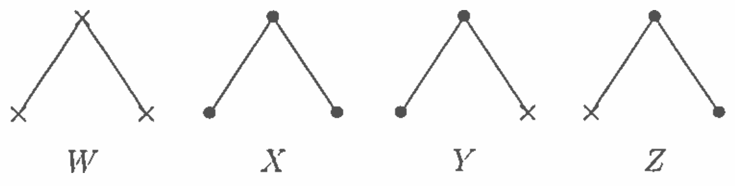


FIGURE 26. The Four Possibilities for Good and Bad Node Distribution.

of the exact cost (4.1). This approximation yields a simple way to calculate the expected cost. We will later see that this approximation is also a good approximation to the expected cost to solution. It also allows us to recursively compute the number of nodes explored and the probability of success. These quantities can be computed efficiently using dynamic programming.

The recursive function for the number of nodes explored in case of failure is:

$$\mathrm{N_{fail}}(d,c) = \begin{cases} 1 & \text{if } d = 0 \\ 1 + \mathrm{N_{fail}}(d-1,c) + \mathrm{N_{fail}}(d-1,c-1) & \text{otherwise} \end{cases} \tag{4.3}$$

It may be instructive to consider building a tree from top to bottom. The quantity d denotes how many levels are left to explore. The expression for the number of nodes in case of failure counts one for each explored node plus the number of nodes in the left child with the same number of discrepancies, but one less level left to explore, plus the number of nodes in the right child with one less allowed discrepancy and one less level left to explore.

Following a similar argument, but also making sure the probabilities of occurrence of each possible goal configuration in Figure 26 are taken into account, we can compute the probability of success for each iteration. Instead of directly computing the probability of failure we compute the probability of failing to fail because it is it is conceptually easier to compute. For instance, event X now represents the case where we fail to fail in both children. The probability of failing to fail is computed by multiplying the probability of occurrence of each configuration (e.g. $\mathrm{pr}(X \mid \neg W)$) by the probability of failing to fail in both the left and right choices (e.g. $\mathrm{P}_{\mathsf{fail}}(d-1,c)\mathrm{P}_{\mathsf{fail}}(d-1,c-1)$).

$$\mathrm{P}_{\mathsf{fail}}(d,c) = \begin{cases} 0 & \text{if } k = d \\ 1 & \text{if } d = 0 \\ \begin{array}{l} \mathrm{pr}(X \mid \neg W)\mathrm{P}_{\mathsf{fail}}(d-1,c)\mathrm{P}_{\mathsf{fail}}(d-1,c-1)+ \\ \mathrm{pr}(Y \mid \neg W)\mathrm{P}_{\mathsf{fail}}(d-1,c) + \mathrm{pr}(Z \mid \neg W)\mathrm{P}_{\mathsf{fail}}(d-1,c-1) \end{array} & \text{otherwise} \end{cases} \tag{4.4}$$

In general, the expected cost to solution for a tree of depth d, where the cost is grouped by iterations, is computed as follows:

$$\langle \textsc{lds} \rangle = \sum_{c=0}^{d} \mathrm{P}_{\mathsf{succ}}(d,c,c-1)\Big(\sum_{j=0}^{c-1} \mathrm{N}_{\mathsf{fail}}(d,j) + \mathrm{N}_{\mathsf{succ}}(d,c)\Big) \tag{4.5}$$

where $\mathrm{P}_{\mathsf{succ}}(d,c,c-1)$ is the probability of success at cutoff c given that we failed at $c-1$, $\mathrm{N}_{\mathsf{fail}}(d,j)$ is the total number of nodes at cutoff j, and $\mathrm{N}_{\mathsf{succ}}(d,c)$ is the number of nodes explored at cutoff c on average when it succeeds.

We show how to compute $\mathrm{P}_{\mathsf{succ}}(d,c,c-1)$ from (4.4) next. $\mathrm{P}_{\mathsf{succ}}(d,c,c-1)$ is the probability that LDS with cutoff c succeeds given that it failed at previous cutoffs. Notice that if LDS failed at cutoff $c-1$ it must have also failed at previous cutoffs $0 \ldots c-2$. This is because the nodes explored at $c-1$ are a superset of nodes explored at previous cutoffs. Thus,

$$\begin{aligned} \mathrm{P}_{\mathrm{succ}}(d,c,c-1) &= \mathrm{P}_{\mathrm{succ}}(d,c) - \mathrm{P}_{\mathrm{succ}}(d,c-1) && (4.6)\\ &= \mathrm{P}_{\mathrm{fail}}(d,c-1) - \mathrm{P}_{\mathrm{fail}}(d,c). && (4.7)\end{aligned}$$

Substituting $\mathrm{P}_{\mathrm{succ}}$ in (4.5) we obtain:

$$\langle \mathrm{LDS} \rangle = \sum_{c=0}^{d} (\mathrm{P}_{\mathrm{fail}}(d,c-1) - \mathrm{P}_{\mathrm{fail}}(d,c))(\sum_{j=0}^{c-1} \mathrm{N}_{\mathrm{fail}}(d,j) + \mathrm{N}_{\mathrm{succ}}(d,c)). \qquad (4.8)$$

$\mathrm{N}_{\mathrm{fail}}(d,c)$ is computed from (4.3) directly. The number of nodes explored in the successful iteration is harder to predict. One may assume, as we did in (4.2), that success is distributed uniformly across the different probes of the iteration, but it is possible to obtain a more accurate estimate by weighing the distribution according to the heuristic probability, p.

$\mathrm{N}_{\mathrm{succ}}(d,c)$ is derived from an analysis of the cost of searching a good node, following analysis by Harvey [24]. Recall that a good node is a node with a goal in the subtree underneath it. A bad node is any node that is not a good node. With heuristic probability p, the cost of searching a good node at depth d with cutoff c can be expressed recursively as:

$$Good(d,k,c,p) = \begin{cases} 1 & \text{if } k = d \\ \begin{array}{l} 1 + pGood(d,k+1,c,p) + \\ (1-p)(Bad(d,k+1,c,p) + Good(d,k+1,c-1,p)) \end{array} & \text{otherwise} \end{cases} \qquad (4.9)$$

where,

$$Bad(d,k,c,p) = \begin{cases} 1 & \text{if } k = d \\ 1 + d - k + \sum_{j=0}^{d-k-1} Bad(d, d-j, c-1, p) & \text{otherwise} \end{cases} \tag{4.10}$$

where k keeps count of the recursion depth and the initial call is made with $Good(d, 0, c, p)$. *Good* basically computes how many nodes are explored if we succeed on the left (which we do with probability p) plus the number of nodes explored if we succeed on the right (and fail on the left with probability $1 - p$).

Expanding the recursions and rearranging terms we arrive at the following simplified expression for the expected cost for the successful iteration.[1]

$$\sum_{k=0}^{d} p^k + (1-p)\sum_{k=0}^{d-1}(d-k)p^k + \sum_{k=1}^{c}\sum_{i_k=0}^{m_1}\sum_{h=1}^{k+1}(1-p)^h C_k(i_1, \ldots, i_k, h) \tag{4.11}$$

where,

$$m_1 = \begin{cases} d-1 & \text{if } k = 1 \\ i_{k-1} - 1 & \text{if } k > 1 \end{cases} \tag{4.12}$$

[1]Section B.1 contains an explanation of how the simplified forms were obtained with the help of Maple (http://www.maplesoft.com), a software for symbolic manipulation.

and where,

$$C_k(i_1,\ldots,i_k,h) = \begin{cases} \displaystyle\sum_{j=d-i_k-k}^{d-k-1} (d-j-k)p^j & \text{if } h > k \\ \displaystyle(i_k+1)\sum_{j=m_2}^{d-i_h-(h+1)} p^j & \text{if } h < k \\ \displaystyle(i_k+1)\sum_{j=m_2}^{d-i_k-(k+1)} p^j + \sum_{j=d-i_k-k}^{d-k} p^j & \text{if } h = k \end{cases} \tag{4.13}$$

where,

$$m_2 = \begin{cases} 0 & \text{if } h = 1 \\ d - i_{k-1} - (k-1) & \text{if } h > 1 \end{cases} \tag{4.14}$$

We derive closed-form solutions for most of the expressions above.

$$\sum_{k=0}^{d} p^k = \frac{1-p^{d+1}}{1-p} \tag{4.15}$$

$$\begin{aligned} \sum_{k=0}^{d}(d-k)p^k &= d\sum_{k=0}^{d} p^k - \sum_{k=0}^{d} kp^k \\ &= \frac{d(1-p^{d+1})}{1-p} - \frac{dp^{d+2}-(d+1)p^{d+1}+p}{(1-p)^2} \\ &= -\frac{p^{d+1}-dp+d-p}{(1-p)^2} \end{aligned} \tag{4.16}$$

These closed-form formulas are enough to get closed-form expressions for the first two terms of the second expression and for the coefficients, C_k.

We can simplify the expressions even further so that the formula in its simplest form is:

$$1 + d + \sum_{k=1}^{c} \sum_{i_k=0}^{m_1} \sum_{h=1}^{k+1} (1-p)^h C_k(i_1, \ldots, i_k, h) \tag{4.17}$$

where,

$$m_1 = \begin{cases} d-1 & \text{if } k = 1 \\ i_{k-1} - 1 & \text{if } k > 1 \end{cases} \tag{4.18}$$

and where,

$$C_k(i_1, \ldots, i_k, h) = \begin{cases} \frac{p^{d-k+1} - i_k p^{d-i_k-k+1} + i_k p^{d-i_k-k} - p^{d-i_k-k+1}}{(1-p)^2} & \text{if } h > k \\ \frac{(i_k+1)(p^{d-i_h-h} - p^{d-i_{k-1}-k+1})}{1-p} & \text{if } h < k \text{ and } h \neq 1 \\ \frac{(i_k+1)(1-p^{d-i_h-h})}{1-p} & \text{if } h < k \text{ and } h = 1 \\ \frac{(i_k+1)p^{d-i_{k-1}-k+1} - i_k p^{d-i_k-k} - p^{d-k+1}}{1-p} & \text{if } h = k \text{ and } h \neq 1 \\ \frac{i_k+1-i_k p^{d-i_k-1} - p^d}{1-p} & \text{if } h = k \text{ and } h = 1 \end{cases} \tag{4.19}$$

Let us try to understand at least where the terms come from. $1+d$ is just the number of nodes explored in the leftmost probe. The term with the nested sums amounts to counting nodes in paths with all the possible combinations of up to c right turns.

For example, let us explore the expression for a tree of depth 2 and cutoffs 0, 1, and 2. For $c = 0$, we notice that the nested sums term is eliminated since the upper limit of the sum is lower than the lower limit of the sum. Then, for $c = 0$, the term is just 3 which is exactly $d + 1$.

For $c = 1$:

$$3 + \sum_{i_1=0}^{1}\sum_{h=1}^{2}(1-p)^h C_1(i_1,h) =$$
$$3 + (1-p)C_1(0,1) + (1-p)^2 C_1(0,2) + (1-p)C_1(1,1) + (1-p)^2 C_1(1,2) =$$
$$3 - p^2 - 2p \tag{4.20}$$

One way to interpret the coefficients is to consider the ones that multiply $1-p$ as quantifying the probability of failing once, first, at depth 1 and then at depth 2. The other coefficients evaluate to 0 in this case.

Finally, for $c = 2$:

$$3 + \sum_{i_1=0}^{1}(\sum_{h=1}^{2}(1-p)^h C_1(i_1,h) + \sum_{i_2=0}^{i_1-1}\sum_{h=1}^{3}(1-p)^h C_2(i_1,i_2,h)) =$$
$$3 - p^2 - 2p + (1-p)C_2(1,0,1) + (1-p)^2 C_2(1,0,2) + (1-p)^3 C_2(1,0,3) =$$
$$3 - p^2 - 2p + (1-p)^2 = 7 - 4p \tag{4.21}$$

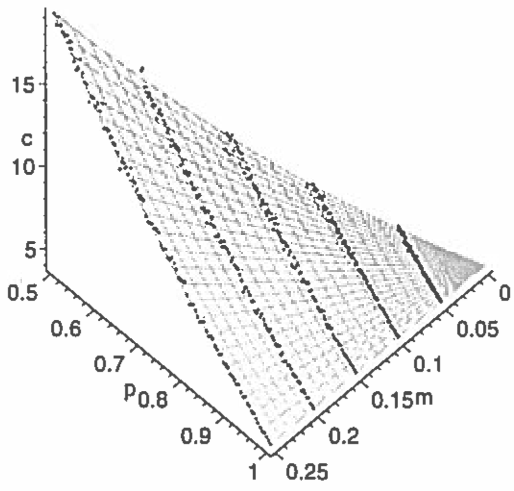
The coefficients of all terms except the one that multiplies $(1-p)^2$ disappear.

Notice that the nested sums result in a polynomial in p. This polynomial depends only on d, c, and p. We can further assume this polynomial is evaluated off-line and results in the term $\mathrm{poly}(d,c,p)$.

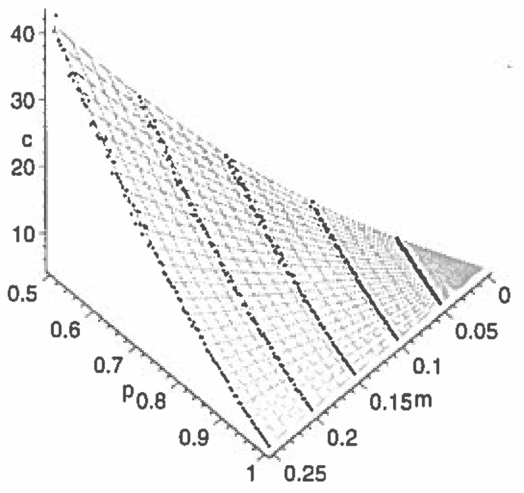
Substituting into the equation we arrive at the following formula for the expected cost for the successful iteration at cutoff c is

$$\mathrm{N}_{\mathrm{succ}}(d,c) = 1 + d + \sum_{k=1}^{c}\mathrm{poly}(d,c,p) \tag{4.22}$$

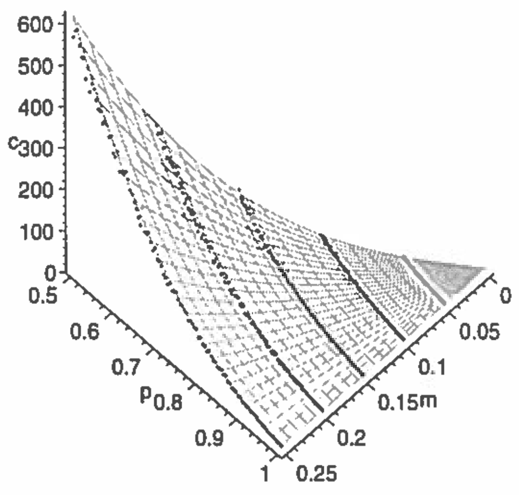
For fixed values of p and m we graphed (4.8) against the data obtained by



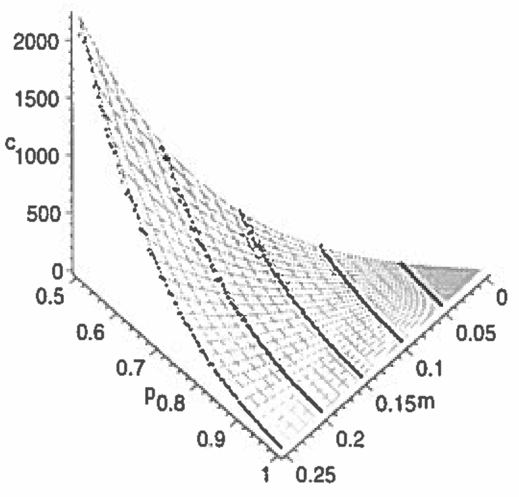
Depth 3



Depth 4



Depth 8



Depth 10

FIGURE 27. Comparing theoretical LDS costs with average over 1,000 runs at small and large depths.

Monte Carlo simulations on trees generated as discussed in Section 2.1.3. Each point in the graph corresponds to the average cost over 1000 trials for each pair of values for p and m. The surface corresponds to the theoretical estimate of the expected cost to solution. The results are displayed in Figure 27. Notice that the theoretical curve for small values of m is always underneath the experimental data. This is an effect of calculating the expected value by iterations instead of calculating the expected cost of each path within each iteration. We emphasize that without knowing the cutoff policy we are unable to determine which paths will be explored in each iteration and that therefore, we are unable to calculate the exact cost. The difference between this approximate cost and the exact cost is minimal, however.

4.2 Expected Cost of DFS

WDS simulates DFS when the cutoff policy is $\{0\}$. LDS explores the whole tree in the last iteration, when $c = d$. If we consider this as the single iteration of the algorithm, then LDS would effectively explore the same nodes as DFS. Thus, the expected cost is the probability of succeeding in this single iteration (which is one since we assume the problem is satisfiable) times the expected number nodes in this successful iteration. The expected cost is given by (4.22) by replacing c with d, since we allow up to d discrepancies in this iteration. Simplifying all expressions yields the expected cost of DFS:

$$\langle \text{DFS} \rangle = 1 + d + (1-p)(2^{d+1} - 2 - d) \tag{4.23}$$

In fact, if we replace d by 2 in (4.23) we obtain $7 - 4p$ which is exactly what we obtained in (4.21). The same result is obtained by Harvey [24] for the last iteration of LDS on a tree of depth 2.

4.3 Expected Cost of WDS

In order to generalize the results of the previous section, we need to change the definition of the cutoff policy. The LDS policy $\{0, 1, \ldots, d\}$ is made up of integral values, but the WDS policy will be made up of values between 0 and 1. A cutoff policy for WDS is a function from iteration numbers to values between 0 and 1, $\mathcal{C} : \mathbb{N} \rightarrow [0, 1]$. Following LDS, we are only interested in strictly monotonic cutoff policies for WDS. Instead of writing $\mathcal{C}(i)$ for a particular cutoff value, we write $\mathcal{C}_i$, since the parameter is an integer and a WDS policy could just as well be denoted by a sequence of cutoff values.

4.3.1 General Expression for the Expected Cost

The general expression for the expected cost to solution for WDS is similar to the expression for LDS (4.5):

$$\langle \text{WDS} \rangle = \sum_{i=1}^{l} \mathrm{P}_{\text{succ}}(\mathcal{C}_i, \mathcal{C}_{i-1})(\sum_{j=1}^{i-1} \mathrm{N}_{\text{fail}}(\mathcal{C}_j) + \mathrm{N}_{\text{succ}}(\mathcal{C}_i)) \tag{4.24}$$

where l denotes the length of the policy $\mathcal{C}$. Notice we left out the parameter d assuming it is fixed for a particular problem. Rearranging, we get

$$\langle \text{WDS} \rangle = \sum_{i=1}^{l} \mathrm{N}_{\text{succ}}(\mathcal{C}_i)\mathrm{P}_{\text{succ}}(\mathcal{C}_i, \mathcal{C}_{i-1}) + \mathrm{N}_{\text{fail}}(\mathcal{C}_i) \sum_{j=i+1}^{l} \mathrm{P}_{\text{succ}}(\mathcal{C}_j, \mathcal{C}_{j-1}).$$

As in LDS, in WDS each iteration explores a superset of the nodes explored in previous iterations, so

$$\begin{aligned} \mathrm{P}_{\mathrm{succ}}(\mathcal{C}_i, \mathcal{C}_{i-1}) &= \mathrm{P}_{\mathrm{succ}}(\mathcal{C}_i) - \mathrm{P}_{\mathrm{succ}}(\mathcal{C}_{i-1}) \\ &= \mathrm{P}_{\mathrm{fail}}(\mathcal{C}_{i-1}) - \mathrm{P}_{\mathrm{fail}}(\mathcal{C}_i) \end{aligned}$$

and the expected cost becomes

$$\begin{aligned} \langle \textsc{wds} \rangle &= \sum_{i=1}^{l} \mathrm{N}_{\mathrm{succ}}(\mathcal{C}_i)(\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_{i-1}) - \mathrm{P}_{\mathrm{fail}}(\mathcal{C}_i)) + \\ &\quad \mathrm{N}_{\mathrm{fail}}(\mathcal{C}_i)(\sum_{j=i+1}^{l} \mathrm{P}_{\mathrm{fail}}(\mathcal{C}_{j-1}) - \mathrm{P}_{\mathrm{fail}}(\mathcal{C}_j)) \\ &= \sum_{i=1}^{l} \mathrm{N}_{\mathrm{succ}}(\mathcal{C}_i)\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_{i-1}) + (\mathrm{N}_{\mathrm{fail}}(\mathcal{C}_i) - \mathrm{N}_{\mathrm{succ}}(\mathcal{C}_i))\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_i). \qquad (4.25) \end{aligned}$$

But, in order to compute the expected cost to solution we need to compute $\mathrm{N}_{\mathrm{succ}}$, $\mathrm{N}_{\mathrm{fail}}$, and $\mathrm{P}_{\mathrm{fail}}$ in terms of a cutoff policy. We are interested in the case where the cutoff policy yields the minimum expected cost to solution. So far, we have assumed that the cost of exploring nodes is uniform and therefore, each node can be assumed to be of unit cost. In the following chapter we show that it is possible to do a similar analysis to find the OCP even if some nodes are cheaper than others. In this case, we will say that the cost of exploring nodes is *non-uniform*. In either case, we will show that the optimal cutoff policy, OCP, will be the one that minimizes (4.25).

CHAPTER V

OPTIMAL CUTOFF POLICIES

There are several ways to obtain the optimal policy from the equation of the expected cost to solution (4.25). One way is to analytically find the cutoff policy that minimizes this equation. Another way is to minimize the equation for the expected cost to solution using numerical methods. Numerical methods usually incur errors and should be used only when the mathematics proves to be too difficult to handle directly. We choose to analytically derive the optimal cutoff policy. However, we will use numerical methods to estimate $\mathrm{N}_{\mathrm{succ}}$, $\mathrm{N}_{\mathrm{fail}}$, and $\mathrm{P}_{\mathrm{fail}}$ and to verify our results.

This chapter begins with a study of the space of candidate policies. The analysis is first done for policies in the particular instantiation that simulates LDS. We determine that exhaustive study for WDS is impractical because the space is too large. We proceed to develop the means to analytically obtain optimal cutoff policies. We finish the chapter with results and discussion on scaling and some useful extensions to our approach.

5.1 General Remarks

In Section 3.4, we determined that cutoff policies start with 1 and end with 0. Since cutoff policies are monotonically decreasing functions, if C is a cutoff policy, $\forall i : \mathrm{N}_{\mathrm{fail}}(C_i) \geq \mathrm{N}_{\mathrm{fail}}(C_{i+1})$.

Notice that two cutoff policies may explore the same nodes in the same order and still be different functions. For instance, $C = \{1, c_1, 0\}$ and $C' = \{1, c_2, 0\}$ such that $c_1 \neq c_2$. Let c_1 and c_2 lie between the same pair of weights (if weights are arranged in decreasing order). WDS with either C or C' will explore the same nodes in the same order.

Two cutoff policies C and C' are said to be *equivalent* if $\forall i \ \exists j : \mathrm{N_{fail}}(C_i) = \mathrm{N_{fail}}(C'_j)$ and $\forall j \ \exists i : \mathrm{N_{fail}}(C'_j) = \mathrm{N_{fail}}(C_i)$. Similar definitions exist for $\mathrm{N_{succ}}$.

An optimal cutoff policy is a function C such that the expected cost of WDS $\langle \text{WDS} \rangle$, is minimized. There is obviously more than one such function since values attained by the function that are smaller than the smallest weight do not change the overall cost. The probability of failure is zero once the policy instructs WDS to explore the whole tree. It is also easy to see that an optimal cutoff policy will be equivalent to some induced function of the weight distribution, where by induced we mean there is an ordered subset of weights such that the subset is equivalent to the optimal cutoff policy. For instance, if the weights are distributed as in the LDS case, the cutoff policy is equal to $\{w^k \mid k \leq d\}$, but any other policy whose cutoff values are interleaved with those of the LDS policy belongs to the same equivalence class since it will instruct LDS to explore the same set of nodes in each iteration.

Since two policies that explore the same nodes in the same order are equivalent, then that the standard LDS policy, $\{0, 1, \ldots, d\}$, is equivalent to $\{1, w, \ldots, w^d\}$, the policy WDS uses when simulating LDS (even in this case, where they are acting on different algorithms). We will use either notation interchangeably throughout the rest of the thesis.

5.1.1 Stability of OCPs

It is important to determine how stable optimal cutoff policies are with respect to changes in the parameters of the search space. An optimal cutoff policy is stable if the expected cost to solution of WDS using the OCP remains near-optimal with respect to small changes in the parameters of the search space. If the parameters of the search space are estimated with error, an unstable optimal cutoff policy will be useless. It is easy to study the stability of policies in the case of LDS, since we know they are formed by integers between 0 and d, where d is the depth of the tree.

5.1.1.1 LDS Policies

A priori, there is no particular reason why LDS should always do iterations $\{0, 1, \ldots, d\}$. Can the expected cost to solution decrease if a different policy is used? How unstable is the optimal cutoff policy for LDS? These are the questions we address in this section.

We will consider only policies which end in d since we want to guarantee the whole space will eventually be searched. We will also consider only finitely discrete and strictly monotonic CPs. The problem of generating all cutoff policies that end with d is equivalent to counting the number of sets of sequences of integers less than d. Since there are d numbers less than d (counting 0), the number of different cutoff policies is 2^d.

Since evaluating all of the policies takes exponential time (there are exponentially many), we opted for hill climbing [50], a widely used method in solving combinatorial optimization problems [30]. Hill climbing starts with a complete configuration or assignment and makes modifications to improve its quality. In our case, the complete configuration is a policy and the objective is to make changes to the policy so as to minimize the cost of LDS running with the policy. In hill climbing, the choice criteria for selecting the modifications, or moves, and the termination criteria are given as input. Hill climbing continually moves in the direction of increasing value. When there is more than one solution to choose from, when there is a tie for instance, one is selected at random.

Imagine the landscape formed by the space of possible solutions. The height of a solution in the landscape corresponds to the value of the evaluation function at that point. The idea is to move around the landscape trying to find the highest peaks (or lowest valleys since we are minimizing cost). Unfortunately, hill climbing has three faults. It may get trapped by local minima, where it will not be able to escape because as formulated, it has no way of making an uphill (or downhill) move. Secondly, hill climbing will wander aimlessly if it searches on a plateau. Finally, hill climbing will zig-zag slowly towards a solution if it is searching along a ridge. In all of these cases, what is missing is some way of making a long jump. Randomly restarting hill climbing after some set timeout is one way of making a long jump. Another way is to add noise. That is, instead of always selecting the best move, with probability p choose the best move and with probability $1 - p$ choose a random move.

We started hill climbing (or descending) from the standard LDS policy. With probability p, we take out the cutoff value that results in the largest decrease in the cost. With probability $1 - p$ we pick a random cutoff and we replace it with another random cutoff. We stop when the policy is equivalent to DFS, or when we have not made progress after twice the number of iterations it took to get to the current best solution. We compare the policies by computing their cost using the equations in Section 4.1. The implementation in Maple of the expected cost calculation may be found in Section B.2. Section B.3 contains the Maple code that implements hill climbing as we have just described.

Table 4 shows the OCPs we found for mistake probability 0.2 and heuristic probability ranging from 0.8 to 0.9. We were curious as to how the different policies would respond to changes in the parameters p and m. We calculated that the expected cost to solution of LDS with OCP(0.2,0.9) at $m = 0.2$ and $p = 0.9$ is 99. If we calculate instead the cost with OCP(0.2,0.8) but with parameters $m = 0.2$ and $p = 0.9$, the cost increases by 12% to 112. A change in p produces a large increase in cost. We wanted to find out if a change in m would also produce a large effect in cost, so we calculated the cost of LDS with OCP(0.2,0.8) at $m = 0.1$ and $p = 0.8$ and obtained a cost of 251. The OCP(0.1,0.8) is $\{1\text{–}5, 10\}$, with a cost of 251, the same cost as OCP(0.2,0.8) with the same parameters. A change in m does not produce a change in cost. Notice that the OCP(0.1,0.8) is identical to the OCP(0.2,0.8) except that it skips cutoff 7. This prompted us to investigate whether all iterations on OCP(0.2,0.8) are actually necessary.

TABLE 4. Results of hill climbing on policy space at depth 10 with mistake probability 0.2 and heuristic probability between 0.8 and 0.9.

OCP(0.2,p)	p
$\{1\text{–}5, 7, 10\}$	0.8–0.83
$\{0\text{–}5, 7, 10\}$	0.84–0.89
$\{0\text{–}5, 6, 8, 10\}$	0.9

In order to determine the impact of each individual cutoff value in a policy, we implemented a neighborhood search algorithm. This algorithm takes one cutoff value at a time away from the policy and evaluates the policy with the remaining cutoffs. It outputs the index of the cutoff value that is skipped and the new cost. The code may be found in Section B.3. We performed this neighborhood search on the OCP(0.2,0.8). Table 5 summarizes the results. Notice that the data suggests that skipping either cutoff 5 or 7 will not produce a change in cost. Skipping both, however, will. If we skip both cutoffs, the policy becomes $\{1\text{–}4, 10\}$ and the cost increases by 7% to 313.

TABLE 5. Neighborhood search around OCP(0.2,0.8) $= \{1\text{–}5, 7, 10\}$ with cost 290 at depth 10.

skip cutoff	1	2	3	4	5	7
cost	344	372	323	295	290	290

TABLE 6. Results of hill climbing on policy space at depth 30 with mistake probability 0.2 and heuristic probability between 0.8 and 0.9.

OCP(0.2,p)	p
$\{2\text{–}14, 16, 18, 23, 30\}$	0.8
$\{2\text{–}14, 16, 18, 22, 30\}$	0.81, 0.82
$\{2\text{–}14, 16, 18, 21, 30\}$	0.83
$\{2\text{–}15, 17, 19, 23, 29, 30\}$	0.84
$\{2\text{–}15, 17, 19, 23, 30\}$	0.85
$\{1\text{–}15, 17, 19, 23, 29, 30\}$	0.86
$\{1\text{–}15, 17, 19, 23, 28, 29, 30\}$	0.87
$\{1\text{–}15, 17, 19, 22, 27, 28, 29, 30\}$	0.88, 0.89
$\{1\text{–}16, 18, 21, 25, 26, 27, 28, 29, 30\}$	0.9

We did the same analysis for depth 30. The resulting optimal cutoff policies may be found in Table 6. We also found that variations in p and m produced increases in cost when the policy remained unchanged. For instance, at depth 30, we calculated the cost ofLDS with OCP(0.2,0.8) at $m = 0.2$ and $p = 0.9$ and obtained a cost of 19,063 versus the 17,557 that we obtain with OCP(0.2,0.9), an increase of 8%. Changing m from 0.2 to 0.1, however, did not produce changes in cost. The OCP(0.1,0.8) is $\{2\text{–}15, 17, 20, 30\}$, which is different from the OCP(0.2,0.8) but they both yield the same cost. It is not uncommon that more than one policy will yield the same cost. What is happening is that there are several iterations (usually past the middle and most expensive iteration) that when replaced by others yield similar costs. There is a difference between iterations that matter, i.e., that if skipped produce a large increase in the cost and iterations that do not matter, i.e., that if skipped the cost remains basically unchanged. The iterations that matter may be determined by a neighborhood search around the optimal cutoff policy. We recorded the results of the neighborhood search in Table 7.

TABLE 7. Neighborhood search around OCP(0.2,0.8) = $\{2\text{–}14, 16, 18, 23, 30\}$ with cost 464,093 at depth 30.

skip cutoff	2	3	4	5	6	7	8	9
cost $\times 10^{-3}$	466	517	644	672	609	543	499	477
skip cutoff	10	11	12	13	14	16	18	23
cost $\times 10^{-3}$	466	465	464	464	464	464	464	464

The neighborhood search suggests that we could skip any of 12 through 23 without affecting the optimal cost. In fact, if we skip all iterations between 12 and 30, the cost is 464,669, suboptimal by 0.1%. This result shows that there are many policies that yield costs near to the optimal cost. This result also shows that the iterations that matter must belong to a near-optimal cutoff policy, but adding a few extra iterations near the end may not affect the cost at all.

TABLE 8. Results of hill climbing on policy space at depth 50 with mistake probability 0.2 and heuristic probability between 0.8 and 0.9.

OCP(0.2,p)	p
$\{2, 4–24, 26, 28, 31, 37, 43–50\}$	0.8
$\{3–23, 25, 27, 30, 34, 42–50\}$	0.81
$\{3–23, 25, 27, 29, 32, 39, 41–50\}$	0.82
$\{3–24, 26, 29, 32, 38, 41–50\}$	0.83
$\{1, 3–24, 26, 28, 31, 36, 40–50\}$	0.84
$\{2–24, 26, 28, 31, 35, 39–50\}$	0.85
$\{2–24, 26, 28, 30, 33, 38–50\}$	0.86
$\{2–24, 26, 28, 30, 34, 39–50\}$	0.87
$\{2–25, 27, 29, 32, 36–50\}$	0.88
$\{1–26, 28, 30, 33, 35–50\}$	0.89
$\{1–25, 27, 29, 31, 34–50\}$	0.9

We also have results at depth 50. Table 8 shows the different optimal cutoff policies for $m = 0.2$ and p ranging between 0.8 and 0.9. Just as we found previously, variations in p produce changes in the expected cost to solution, but variations in m do not. For instance, when we computed the cost of LDS with the OCP(0.2,0.8) at $m = 0.1$ and $p = 0.8$ we obtained a cost of 0.922×10^8. The OCP(0.1,0.8) $= \{3–24, 26, 28, 31, 36, 41–40\}$, however has a cost of 0.920×10^8, roughly the same as the OCP(0.2,0.8). Thus, a change in m does not produce a significant change in cost. We also computed the cost of LDS with OCP(0.8,0.2) with parameters $m = 0.2$ and $p = 0.9$. This produced an expected cost of 0.915×10^7 versus the 0.117×10^7 that is produced by the OCP(0.9,0.2), a dramatic change of 87% in the cost.

We also did a neighborhood search around the OCP(0.2,0.8). The results

TABLE 9. Neighborhood of OCP(0.2,0.8) = $\{2, 4\text{–}24, 26, 28, 31, 37, 43\text{–}50\}$ with cost 297×10^6 at depth 50.

skip cutoff	2	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	26	28	31	37	43	44	45	46	47	48	49
cost $\times 10^{-6}$	297	303	335	391	417	412	390	364	340	322	310	303	300	298	297	297	297	297	297	297	297	297	297	297	297	297	297	297	297	297	297	297	297

of the neighborhood search suggests that only cutoffs 4 through 16 are necessary. As we have seen in experiments at smaller depths, there are many policies that yield costs near to the optimal cost. Iterations that matter in all cases may be summarized by the following rule:

1. Include all cutoffs corresponding to iterations between 0 and $d/3$.

2. Add every other cutoff corresponding to iterations between $d/3$ and $d/2$.

3. Add the cutoff corresponding to the last iteration, d.

We will see that policies constructed according to this rule are “robust” cutoff policies. A policy is robust with respect to changes in p and m if the expected cost to solution remains near-optimal. A policy constructed using this rule will be referred to as a robust cutoff policy, and we will denote it by ROCP. Table 10 compares the costs of the OCP and ROCP at different values of p and $m = 0.2$ for depths 10, 30, 50, and 100. Note that the ROCP gets more robust with depth. At depth 50, the ROCP is within 1% of the OCP at each value of p and at depth

100, the cost of LDS with the ROCP is the same as the optimal cost for each value of p.

TABLE 10. Comparing OCP vs. ROCP costs at $m = 0.2$ and $p = 0.8$–0.9.

Depth	Policy Cost	0.8	0.81	0.82	0.83	0.84	0.85
10	OCP cost	290	264	241	219	199	179
	ROCP cost	299	272	247	223	202	181
30	OCP cost $\times 10^{-3}$	464	347	258	191	139	101
	ROCP cost $\times 10^{-3}$	465	348	259	191	140	101
50	OCP cost $\times 10^{-6}$	297	181	110	65.6	38.7	22.5
	OCP cost $\times 10^{-6}$	297	182	110	65.7	38.7	22.5
100	OCP cost $\times 10^{-12}$	1147	420	150	52.5	17.8	5.89
	ROCP cost $\times 10^{-12}$	1147	420	150	52.5	17.8	5.89
Depth	Policy Cost	0.86	0.87	0.88	0.89	0.9	
10	OCP cost	160	143	127	113	99	
	ROCP cost	162	145	129	114	100	
30	OCP cost $\times 10^{-3}$	73.1	52.0	36.6	25.5	17.5	
	ROCP cost $\times 10^{-3}$	73.1	52.1	36.7	25.5	17.5	
50	OCP cost $\times 10^{-6}$	12.9	7.2	4.0	2.1	1.1	
	ROCP cost $\times 10^{-6}$	12.9	7.2	4.0	2.1	1.1	
100	OCP cost $\times 10^{-12}$	1.89	0.58	0.17	0.5	0.1	
	ROCP cost $\times 10^{-12}$	1.89	0.58	0.17	0.5	0.1	

If the heuristic probability, p is estimated with error, it may sometimes be more useful to obtain a policy that is robust across different values of p than a policy that is optimal for a particular value. This is especially true if the OCPs tend to be unstable.

It makes sense that no iterations after $d/2$ matter because the middle iteration is the most expensive (due to the normal distribution of discrepancy counts) so once LDS explores it, it may as well explore what is left of the search tree since successive iterations will not add many more new nodes. We will see in Section 5.4

that it is possible to quantify the tradeoff between probability of success and number of nodes explored.

5.1.1.2 WDS Policies

We would like to do a similar analysis for WDS. We need to extend the equations in Section 4.1 to account for weights in order to be able to compute the expected cost to solution.

The number of nodes and probability of failure will depend on the weight distribution but, we already saw in Section 3.3.2 that the weight distribution is lognormal and its parameters depend on depth. Let Ω be the cumulative density function for the weights. Then, $\Omega(c) = P(\omega \geq c)$ is the probability that ω is greater than c.

Given a weight distribution, it is possible to determine the expected number of nodes at a particular cutoff value c. We give the general expression for $\mathrm{N}_{\mathrm{succ}}$ in terms of Ω to illustrate how the expressions in Section 4.1 would change to accommodate the weights. Expressions for $\mathrm{N}_{\mathrm{fail}}$ and $\mathrm{P}_{\mathrm{fail}}$ are obtained analogously.

The cost of the successful iteration can be estimated by considering only the cost of nodes whose weight is expected to be greater than or equal to the cutoff, or

$$\mathrm{N}_{\mathrm{succ}}(\Omega, d, c) = 1 + d + \sum_{k=0}^{d} \sum_{i_k=0}^{m_1} \Omega(c) \sum_{h=1}^{k+1} (1-p)^h C(i_1, \ldots, i_k, h) \tag{5.1}$$

where, m_1 and C_k are defined as in Section 4.1.

In Chapter IV we showed that in the specific case of LDS and DFS, the respective instantiations of this formula yield good approximations to the actual number of nodes explored in the successful iteration. We showed in Section 3.3.2

that weights distribute according to the lognormal distribution and that the parameters depend on depth. Hence, if we record the depths at which right turns are made, we should be able to get a value for $\Omega(c)$ for a given c. The code in Section B.1 implements both LDS and WDS. If it is running WDS it records the depths at which right turns were made. If the distribution is available, we should replace the routine that records the depths with a call to the weight distribution function.

The accuracy of (5.1) depends on the accuracy with which we estimate the weight distribution. If it can be estimated precisely, we believe (5.1) will provide a good approximation to N_{succ} in the case of WDS. We have not verified this, however, because the ultimate goal is to obtain optimal cutoff policies for WDS. In the case of LDS, we obtained optimal cutoff policies by evaluating policies using the equations derived in Section 4.1. In the case of WDS, there are infinitely many policies and though hill climbing may work, we have chosen to follow a different path. The following section presents an alternative mechanism for obtaining optimal cutoff policies for WDS.

5.2 Using Calculus of Variations to Derive Optimal Cutoff Policies

One way to determine optimal cutoff policies is to resort to analytical methods. In this section, we show how OCPs can be obtained using calculus of variations.

The technique presented in this section is completely general and has much greater applicability than treated in this section. For the purposes of this section, we assume each node takes constant time to expand. Another way to phrase this assumption is to view it as if there was a cost function associated with node expansions and the cost function assigned uniform cost to all nodes explored. This issue will play a role later on when we discuss extensions of our approach that cope with non-uniform cost.

In Section 4.3 we derived the expression for the expected cost to solution of WDS:

$$\langle \text{WDS} \rangle = \sum_{i=1}^{\infty} \mathrm{N}_{\text{succ}}(\mathcal{C}_i)\mathrm{P}_{\text{fail}}(\mathcal{C}_{i-1}) + (\mathrm{N}_{\text{fail}}(\mathcal{C}_i) - \mathrm{N}_{\text{succ}}(\mathcal{C}_i))\mathrm{P}_{\text{fail}}(\mathcal{C}_i) \tag{5.2}$$

We replaced l with ∞ on the grounds that if the policy instructs WDS to search the whole tree at $\mathcal{C}_l$, the probability of failure is zero for this and subsequent iterations.

For large depths, the space of possible cutoff values is dense, so we can assume the space is continuous.[1]

$$\langle \text{WDS} \rangle = \int_1^{\infty} di \; (\mathrm{N}_{\text{fail}}(\mathcal{C}_i) - \mathrm{N}_{\text{succ}}(\mathcal{C}_i))\mathrm{P}_{\text{fail}}(\mathcal{C}_i) + \mathrm{N}_{\text{succ}}(\mathcal{C}_i)\mathrm{P}_{\text{fail}}(\mathcal{C}_{i-1}) \tag{5.3}$$

Since the expected cost is a functional (function of a function), the calculus of variations [46] can be used to obtain the cutoff policy that minimizes the cost. An extremum of the cost functional will occur when

$$\langle \text{WDS} \rangle_{\mathcal{C}} - \frac{d}{di}\langle \text{WDS} \rangle_{\mathcal{C}'} = 0 \tag{5.4}$$

but, since the expected cost functional depends only on the cutoff policy and not

[1]Even though the space of cutoff policies is continuous, we denote the previous cutoff by $\mathcal{C}_{i-1}$. It does not make sense to denote the previous cutoff by its differential, because the cutoff policy presumably varies discretely.

on its derivative, $\langle \textsc{wds} \rangle_{\mathcal{C}'} = 0$.

$$\begin{aligned}
&\frac{d}{d\mathcal{C}}\langle \textsc{wds} \rangle \\
&= \frac{d}{d\mathcal{C}} \int_1^\infty di\ (\mathrm{N_{fail}}(\mathcal{C}_i) - \mathrm{N_{succ}}(\mathcal{C}_i))\mathrm{P_{fail}}(\mathcal{C}_i) + \mathrm{N_{succ}}(\mathcal{C}_i)\mathrm{P_{fail}}(\mathcal{C}_{i-1}) \\
&= \int_1^\infty di\ (\mathrm{N_{fail}}'(\mathcal{C}_i) - \mathrm{N_{succ}}'(\mathcal{C}_i))\mathrm{P_{fail}}(\mathcal{C}_i) + (\mathrm{N_{fail}}(\mathcal{C}_i) - \mathrm{N_{succ}}(\mathcal{C}_i))\mathrm{P_{fail}}'(\mathcal{C}_i) + \\
&\quad \mathrm{N_{succ}}'(\mathcal{C}_i)\mathrm{P_{fail}}(\mathcal{C}_{i-1}) + \frac{d}{d\mathcal{C}} \int_0^{\infty-1} dk\ \mathrm{N_{succ}}(\mathcal{C}_{k+1})\mathrm{P_{fail}}(\mathcal{C}_k) \\
&= \int_1^\infty di\ (\mathrm{N_{fail}}'(\mathcal{C}_i) - \mathrm{N_{succ}}'(\mathcal{C}_i))\mathrm{P_{fail}}(\mathcal{C}_i) + (\mathrm{N_{fail}}(\mathcal{C}_i) - \mathrm{N_{succ}}(\mathcal{C}_i))\mathrm{P_{fail}}'(\mathcal{C}_i) + \\
&\quad \mathrm{N_{succ}}'(\mathcal{C}_i)\mathrm{P_{fail}}(\mathcal{C}_{i-1}) + \int_0^{\infty-1} dk\ \mathrm{N_{succ}}(\mathcal{C}_{k+1})\mathrm{P_{fail}}'(\mathcal{C}_k) \\
&= \int_1^\infty di\ (\mathrm{N_{fail}}'(\mathcal{C}_i) - \mathrm{N_{succ}}'(\mathcal{C}_i))\mathrm{P_{fail}}(\mathcal{C}_i) + (\mathrm{N_{fail}}(\mathcal{C}_i) - \mathrm{N_{succ}}(\mathcal{C}_i))\mathrm{P_{fail}}'(\mathcal{C}_i) + \\
&\quad \mathrm{N_{succ}}'(\mathcal{C}_i)\mathrm{P_{fail}}(\mathcal{C}_{i-1}) + \mathrm{N_{succ}}(\mathcal{C}_{i+1})\mathrm{P_{fail}}'(\mathcal{C}_i) + \\
&\quad \int_0^1 di\ \mathrm{N_{succ}}(\mathcal{C}_{i+1})\mathrm{P_{fail}}'(\mathcal{C}_i) - \int_{\infty-1}^\infty di\ \mathrm{N_{succ}}(\mathcal{C}_{i+1})\mathrm{P_{fail}}'(\mathcal{C}_i).
\end{aligned}$$

But the last two terms disappear since no nodes are explored before the first cutoff and the probability of failure near infinity is zero because the policy is finite.

Thus (5.4) becomes

$$\int_1^\infty di\ ((\mathrm{N_{fail}}'(\mathcal{C}_i) - \mathrm{N_{succ}}'(\mathcal{C}_i))\mathrm{P_{fail}}(\mathcal{C}_i) + (\mathrm{N_{fail}}(\mathcal{C}_i) - \mathrm{N_{succ}}(\mathcal{C}_i))\mathrm{P_{fail}}'(\mathcal{C}_i) + \\ \mathrm{N_{succ}}'(\mathcal{C}_i)\mathrm{P_{fail}}(\mathcal{C}_{i-1}) + \mathrm{N_{succ}}(\mathcal{C}_{i+1})\mathrm{P_{fail}}'(\mathcal{C}_i)) = 0$$

or

$$(\mathrm{N_{fail}}'(\mathcal{C}_i) - \mathrm{N_{succ}}'(\mathcal{C}_i))\mathrm{P_{fail}}(\mathcal{C}_i) + (\mathrm{N_{fail}}(\mathcal{C}_i) - \mathrm{N_{succ}}(\mathcal{C}_i))\mathrm{P_{fail}}'(\mathcal{C}_i) + \\ \mathrm{N_{succ}}'(\mathcal{C}_i)\mathrm{P_{fail}}(\mathcal{C}_{i-1}) + \mathrm{N_{succ}}(\mathcal{C}_{i+1})\mathrm{P_{fail}}'(\mathcal{C}_i) = 0.$$

Let

$$T(\mathrm{N_{succ}}, \mathrm{N_{succ}}', \mathrm{N_{fail}}, \mathrm{N_{fail}}', \mathrm{P_{fail}}, \mathrm{P_{fail}}', \mathcal{C}_i) =$$
$$(\mathrm{N_{fail}}'(\mathcal{C}_i) - \mathrm{N_{succ}}'(\mathcal{C}_i))\mathrm{P_{fail}}(\mathcal{C}_i) + (\mathrm{N_{fail}}(\mathcal{C}_i) - \mathrm{N_{succ}}(\mathcal{C}_i))\mathrm{P_{fail}}'(\mathcal{C}_i).$$

Substituting for T and solving for $\mathcal{C}_{i+1}$ gives us the optimal cutoff policy:

$$\mathcal{C}_{i+1} = \mathrm{N_{succ}}^{-1}\left(\frac{-\mathrm{N_{succ}}'(\mathcal{C}_i)\mathrm{P_{fail}}(\mathcal{C}_{i-1}) - T(\mathrm{N_{succ}}, \mathrm{N_{succ}}', \mathrm{N_{fail}}, \mathrm{N_{fail}}', \mathrm{P_{fail}}, \mathrm{P_{fail}}', \mathcal{C}_i)}{\mathrm{P_{fail}}'(\mathcal{C}_i)}\right). \tag{5.5}$$

As a boundary condition, we require that $\mathcal{C}_1 = 1$, an extra cost of $d+1$ nodes. The second cutoff can be obtained by setting $\mathrm{P_{fail}}(\mathcal{C}_{i-1}) = 1$ since we have not yet explored any nodes and thus, we are guaranteed to fail. The formula is used as stated to obtain the rest of the cutoff values. The process stops when the cutoff indicates WDS should search the entire tree.

Note that the only assumption we have made is that cutoff values lie in a continuous space. For large depths, this argument holds for LDS as well, and the result could be used to obtain optimal cutoff policies for LDS.

We could use the equations in Section 4.1 to obtain expressions for $\mathrm{N_{succ}}$, $\mathrm{N_{succ}}'$, $\mathrm{N_{fail}}$, $\mathrm{N_{fail}}'$, $\mathrm{P_{fail}}$, and $\mathrm{P_{fail}}'$. Unfortunately, the mathematics gets too involved. A much easier way to estimate these functions is to numerically approximate them via Monte Carlo simulations. In order to do this, we must first specify how to generate random search trees given a weight distribution.

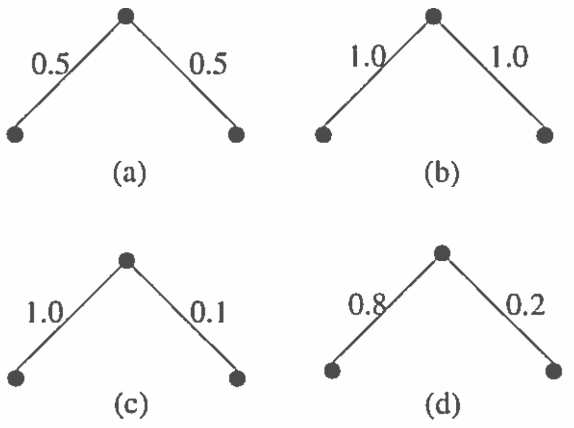


FIGURE 28. How different right nodes compare to their siblings.

5.3 Generating Random Search Trees

In Section 2.1.3 we discussed how to generate random trees given p and m. Here, we are interested in relating p and m to the weights, assuming the weights reflect, in some measure, the probability of success of a node. The goal is to generate random search trees with goals distributed according to a given weight distribution.

Given a weight w_l for the left child and a weight w_r for the right child of a node, we can determine how likely we are to succeed if we follow that choice. For instance, Figure 28 shows how different right nodes compare to their siblings. Figure 28 (a) and (b) show situations where the right node is as likely as the left node to lead to a goal, (showing strong discrepancies). Figure 28 (c) and (d) show weak discrepancies. In particular, Figure 28 (d) shows the case where eight out of ten times the goal will be found underneath the left child and two out of ten times it will be found underneath the right child.

The probability that the left child is good is $p_l = \frac{w_l}{w_l+w_r}$ and the probability that the right child is good is $p_r = \frac{w_r}{w_l+w_r}$. If the heuristic assigns values to siblings independently of each other, then we can assume that the chances of both children being good is just the product of the probability that each child is good. Thus, the probability that both children are good is $p_{lr} = p_l p_r = \frac{w_l w_r}{(w_l+w_r)^2}$.

Following this reasoning we can easily derive the relationship between the weights and p and m. If there is a single goal, we know (from Section 2.1.2) that a single goal occurs with probability $2m$. With probability $p_r 2m$ it occurs in the right. But, the probability that it occurs on the right also is $1-p$, so $1-p = p_r 2m$. Thus,

$$p = 1 - p_r 2m. \tag{5.6}$$

On the other hand, we know that the probability both are goals is given by $1-2m$, but it is also equal to $p_l p_r$, therefore, $1 - 2m = p_l p_r$ which means

$$m = \frac{1 - p_l p_r}{2}. \tag{5.7}$$

Substituting (5.7) in (5.6), we get the general expression for p in terms of only p_l and p_r

$$p = 1 - \frac{p_r(1 - p_l p_r)}{p_l + p_r}. \tag{5.8}$$

For instance, if $w_l = 1.0$ and $w_r = 1.0$, this means that $p_l = 0.5$ and $p_r = 0.5$, which yields $m = 0.375$ and $p = 0.625$. If, on the other hand, $w_l = 0.8$ and $w_r = 0.1$, then $m = 0.45$ and $p = 0.9$.

We can prove the properties of p and m are still valid given these relationships. The properties are the following:

1. $m \leq 0.5$ (from (2.9))

2. if $p = 1 - m$, p orders successors randomly

3. if $p > 1 - m$, p does better than random

4. if $p < 1 - m$, p does worse

5. if $p = 1 - 2m$, p is the worst possible heuristic

These relationships are still valid when p and m are computed from the weights.

First, let us show that Property 1 is still valid. We know that $m = \frac{1-p_l p_r}{2}$, where $p_l \in [0,1]$ and $p_r \in [0,1]$. For the extreme values, if $p_l p_r = 1$, then $m = 0$, and if $p_l p_r = 0$ then $m = 0.5$. Intuitively, if both probabilities are 1 it means that both nodes contain goals and therefore, $m = 0$. If either p_l or p_r or both are zero, then it means that only one or none contain goals and we know that if $m = 0.5$, then the probability that both contain goals is zero. In general, we may plot m versus p_l and p_r and obtain the surface in Figure 29 which confirms that in fact $m \leq 0.5$.

For Property 2, we have that if p orders successors randomly, then $p = 1-m$. If p orders successors randomly, this means that $p_l = p_r$. If $p_l = p_r$, then $m = \frac{1-p_l^2}{2}$ and $p = 1 - \frac{p_l(1-p_l^2)}{2p_l} = 1 - \frac{1-p_l^2}{2} = 1 - m$.

Next, we show that Property 3 is still valid. We know that if the heuristic orders nodes better than random, then $p_l > p_r$. But, if $p_l > p_r$, then $p_l + p_r > 2p_r$, which means that $\frac{p_r}{p_l+p_r} \leq \frac{1}{2}$. Since $p = 1 - \frac{p_r}{p_l+p_r}2m$ and $\frac{p_r}{p_l+p_r} \leq \frac{1}{2}$, then $p \geq 1 - \frac{1}{2}2m = 1 - m$.

Properties 4 and 5 are proven analogously, although they are irrelevant if the heuristic does better than random. Figure 30 confirms that $0 \leq p \leq 1$ when it is derived from p_l and p_r.

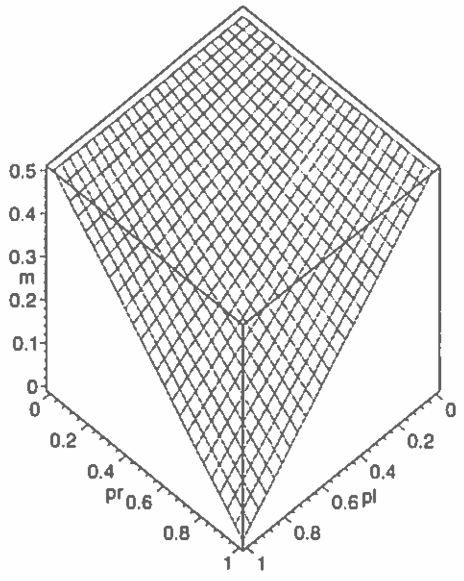


FIGURE 29. Mistake probability as a function of weights.

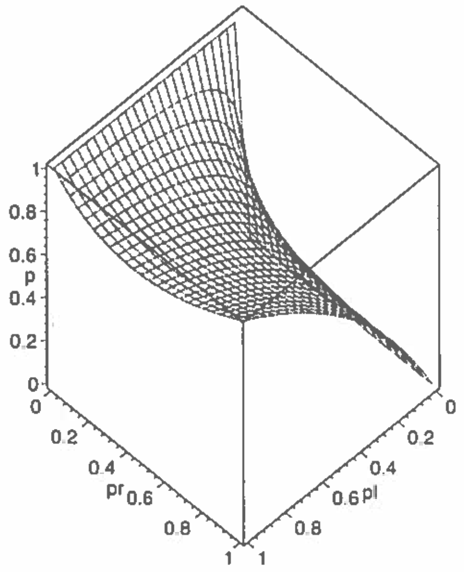


FIGURE 30. Heuristic probability as a function of weights.

This gives us a mechanism to compute p and m given a weight distribution, and we have already seen in Section 2.1.3 that it is possible to construct random trees for given values of p and m.

In order to understand where p_l and p_r come from, we can invert the expressions for p and m. It turns out that the expressions for p_l and p_r in terms of p and m show they are exactly what we would expect them to be:

$$p_r = \sqrt{\frac{(1-2m)(1-p)}{p+2m-1}} \tag{5.9}$$

So p_r is the square root of the probability that there is a goal in both children times the ratio of the probability that there is a goal only on the left child versus the probability that there is a goal only on the right child. p_l is just the ratio of the probability that both children are goals over p_r:

$$p_l = \frac{1-2m}{p_r} = \sqrt{\frac{(1-2m)(p+2m-1)}{1-p}}. \tag{5.10}$$

To summarize, given a particular weight distribution for trees of a fixed depth, we can generate random trees by translating the weights into values for p and m and then use equations (2.3) through (2.8) to assign the goal nodes.

The subject of efficiently building reproducible random trees has been treated in detail elsewhere [37]. An incremental random tree is generated by assigning independent random values to the edges of a tree, and computing the signal of an interior node as well as the exact value of a leaf node. The idea, briefly, is to tag each node in the tree and use the tag to generate the node's value.

Berliner [4] first proposed reseeding the random number generator with some tag particular to the node. He proposed using the breadth-first enumeration of the node in the tree as a seed. He then used the random number so generated to value the node. Any other function of the path to the node may be used in order to seed the random number generator, but the breadth-first index is simple enough to use. Korf [37] uses the breadth-first index in order to generate random instances. Given a random seed used to generate the parent edge value, the child's value is produced by skipping as many random values in the sequence as the difference between the indices of the two nodes in the breadth-first enumeration scheme. Korf shows that it is possible to do this in $\log(n)$ time where n is the total number of nodes generated. He proves this by expanding the recurrence relation that produces the sequence of (pseudo) random numbers.

5.4 Optimal Cutoff Policies for LDS

In order to illustrate that we do not lose anything by sampling, we first generate random trees according to weight distributions for LDS. We estimate N_{succ}, N_{fail}, and P_{fail} in order to compare different cutoff policies and see how the results compare to those obtained in Section 4.1.

For a fixed depth, we generated 10,000 instances for fixed values of the parameters of the search space. We settled on 10,000 trials because separate runs with different random seeds yield results that vary by 1%. We select candidate cutoff values at random between 0 and 1. For each cutoff value, we run WDS simulating LDS on a single iteration with that cutoff value. Note that if the weight used in the simulation of LDS is w and two cutoff values lie between weights w^k and w^{k+1}, both cutoffs will yield the same results. We record the total number of nodes

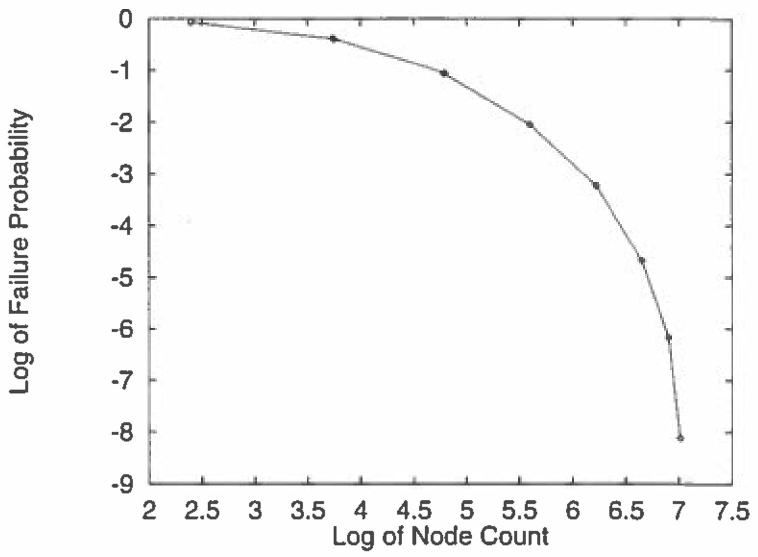


FIGURE 31. Tradeoff between node count and probability of failure at depth 10.

explored, the number of nodes explored in the successful iteration and the cutoff value at which it succeeds. With this information, it is possible to compute N_{succ}, N_{fail}, and P_{fail} for each cutoff value. The number of nodes in the failed iteration is obtained by subtracting the number of nodes in the successful iteration from the total number of nodes explored. The probability of success of each iteration is obtained by counting the number of times out of 10,000 that we succeed in that particular iteration.

Figure 31 shows the logarithm of the number of nodes explored versus the logarithm of the probability of failure for each LDS iteration that fails. Each point on the figure denotes the result of running a single iteration at cutoff values corresponding to an increasing number of discrepancies. The topmost point corresponds to LDS(0), the next point corresponds to LDS(1), etc. until the last point in the figure which corresponds to LDS(7). Since there are no points for LDS(8) through

LDS(10), this means that LDS succeeds before exploring paths with 8 discrepancies in all trials. Notice the shape of the curve. Each data point seems to be equally spaced along the elbow of the curve. This means that a constant factor more new nodes are explored in each successive iteration. Points along the bend of the elbow reflect substantial changes in the probability of success (actual decreases in the probability of failure). Given the best fit to the points in the figure, the bend corresponds to the portion of the function where the second derivative is negative and where the slope changes slowly. The second derivative is positive for points in Figure 31 lying to the left of 3.7, and it is negative elsewhere. Notice that 3.7 corresponds to LDS(1). The slopes between consecutive points in the figure are: -0.02,-0.10,-0.21,-0.36,-0.51,-0.70,-0.89. Notice that the changes in slope between the last three is greater than the change in slope between the first five. The last two slopes correspond to the slope between points corresponding to LDS(5) and LDS(6), and LDS(6) and LDS(7). Thus, LDS(1) through LDS(5) belong to the bend of the elbow.

Iterations that matter (as discussed in Section 5.1.1.1) should be related to the points that have negative slope and represent substantial increases in probability of success. At depth 10, we established that the cutoffs that were essential for good performance of LDS were 1 through 5. These are the same iterations that can be identified in Figure 31 as those that should matter.

Figure 32 shows the tradeoff between node count and failure probability for a tree of depth 15. Notice the same elbow-shaped curve appears. Also notice that LDS succeeds before LDS(10) in all trials, since there are only 10 points each corresponding to LDS(0) through LDS(9). If the analysis at depth 10 carries over

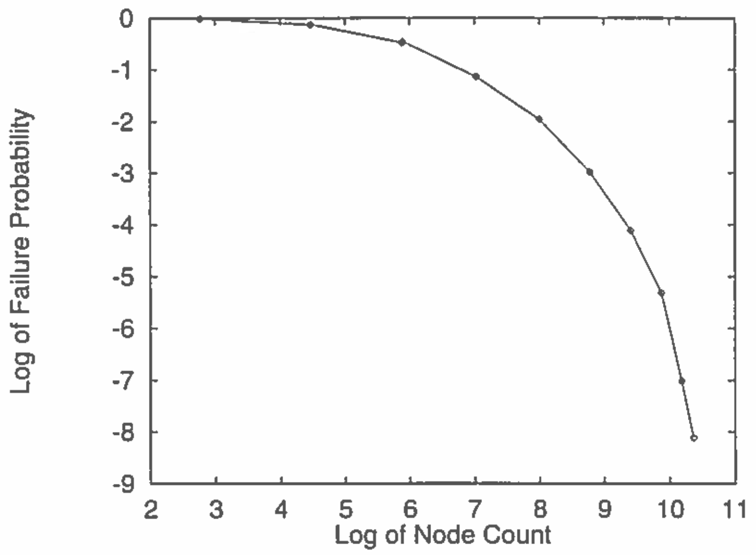


FIGURE 32. Tradeoff between node count and probability of failure at depth 15.

to depth 15, then the iterations that matter would lie along the bend of the elbow. The elbow again is identified by points where the second derivative is negative and where the slope changes slowly. The second derivative is positive for points in Figure 32 lying to the left of 4.1 and negative elsewhere. The point at 4.5 corresponds to LDS(1). The slopes between consecutive points in the figure are: -0.01,-0.03,-0.16,-0.24,-0.33,-0.43,-0.53,-0.68,-0.81. Notice there is almost a constant change in the slope in the middle, but the last two slopes are steeper. Thus, we can say that the points lying in the bend of the elbow are those corresponding to LDS(1) through LDS(7).

We want to verify that this observation is correct at depth 15. We obtained the following optimal cutoff policy via hill climbing, just as we did for depth 10 in Section 5.1.1.1: $\{1\text{–}7, 9, 13, 15\}$ with an expected cost to solution of 2176 nodes. Then we did a neighborhood search to determine if all iterations were necessary.

TABLE 11. Neighborhood search around OCP(0.2,0.8) = $\{1\text{–}7, 9, 13, 15\}$ with cost 2176 at depth 15.

skip cutoff	1	2	3	4	5	6	7	9	13
cost	2243	2749	2928	2524	2270	2192	2181	2177	2176

Table 11 shows the results of the neighborhood search. Notice that the cost is not affected by skipping cutoffs 6 through 13, so skipping one of these does not affect the resulting cost by more than 1%. We still do not know if skipping all of these iterations produces a large increase in cost. Let us examine the cost of the ROCP at depth 15. The ROCP for this depth is $\{1\text{–}5, 7, 15\}$ and the cost of LDS running with this policy is 2204 nodes, an increase of 1% over the OCP. Skipping cutoffs 6, 9, and 13 has no noticeable effect on the cost. This is consistent with our observations of Figure 32.

At depths 10 and 15, the iterations that matter are always smaller than or equal to $\frac{d}{2}$. We arrive at the same conclusion we arrived in Section 5.1.1: The iterations that matter lie between cutoffs 1 and $\frac{d}{2}$. The ROCP seems to be a good approximation to the OCP and it is easy to construct. But, how does the ROCP fare against LDS in actual runs of the algorithms?

We ran LDS with the ROCP and LDS with its standard policy $\{0, 1, \ldots, d\}$ for 10,000 trials on randomly generated trees at depth 10 and depth 15. At depth 10, the average run time was 463 while for LDS it was 478. This represents a savings of 3%. At depth 15, however, the savings increase. LDS with the ROCP has an average cost of 6694 while LDS with the standard policy has an average cost of 7345. This represents a savings of 9%.

We have shown that we can do better by skipping a few iterations. Can we

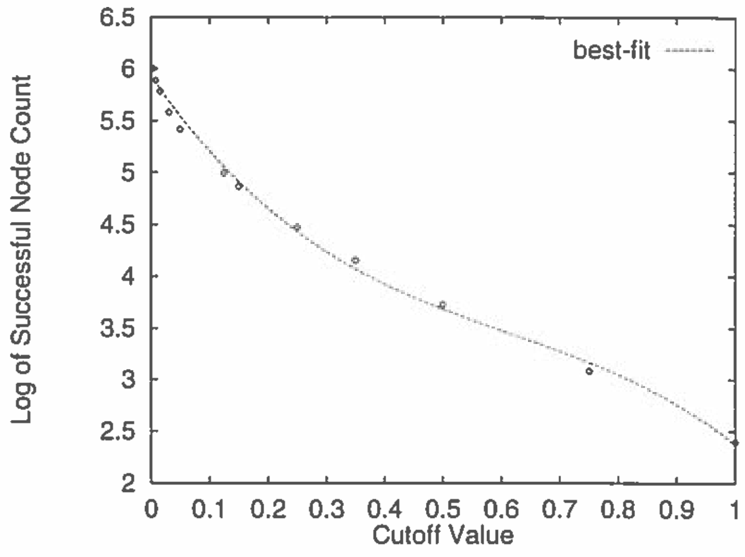


FIGURE 33. Successful node count vs. cutoff value at depth 10.

improve on LDS by finding the optimal cutoff policy for WDS and running WDS using the OCP instead of LDS?

5.5 Optimal Cutoff Policies for WDS

We need to determine the functions N_{succ}, N_{fail}, and P_{fail} in order to use (5.5) to obtain the optimal cutoff policy. As described in the previous section, we select cutoff values at random, run 10,000 trials, and record the data for N_{succ}, N_{fail}, and P_{fail}. Then we find best fits for these functions. We use the fits and their derivatives to compute the OCP directly from (5.5).

Figure 33 shows the number of nodes explored in successful iterations in terms of a cutoff value c at depth 10. The best fit is:

$$s(c) = e^{-5.494c^3+10.124c^2-8.188c+5.931}.$$

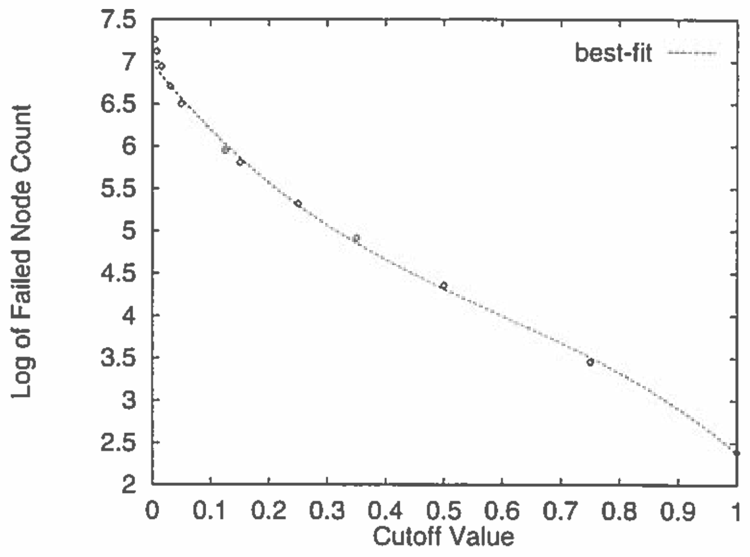


FIGURE 34. Failed node count vs. cutoff value at depth 10.

Figure 34 shows the number of nodes explored in failed iterations in terms of a cutoff value c, at depth 10. It turns out to be easier to make best fits to the logarithm of the data. The best fit is:

$$f(c) = e^{-5.376c^3+9.536c^2-8.756c+6.977}.$$

Figure 35 shows the failure probability as a function of f. The best fit is:

$$\mathrm{P}_{\mathrm{fail}}(c) = e^{-0.105\ln^3(f)+1.184\ln^2(f)-4.574\ln(f)+5.607}.$$

With the fit functions it is now possible to compute the optimal cutoff policy using (5.5), where

$$\mathrm{N}_{\mathrm{succ}}(c) = e^{-5.494c^3+10.124c^2-8.188c+5.931}$$

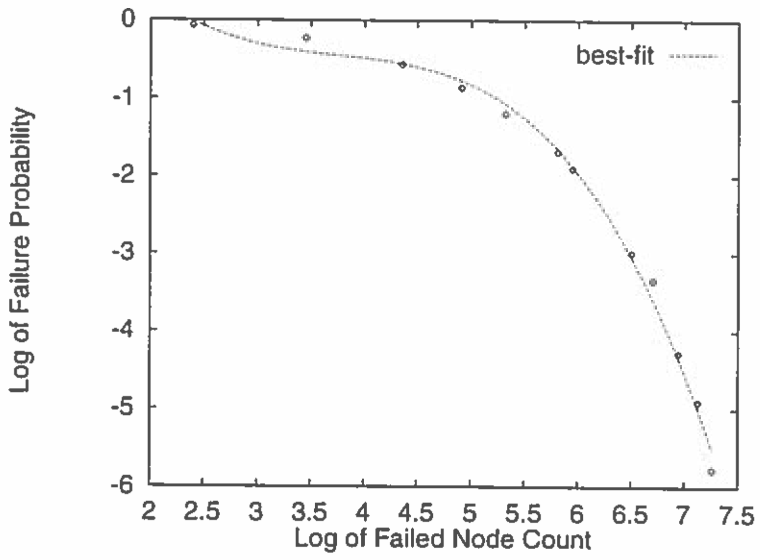


FIGURE 35. Probability of failure vs. failed node count at depth 10.

$$\mathrm{N_{succ}}'(c) = (-16.482c^2 + 20.248c - 8.188)\mathrm{N_{succ}}(c)$$

$$\mathrm{N_{fail}}(c) = e^{-5.376c^3+9.536c^2-8.756c+6.977}$$

$$\mathrm{N_{fail}}'(c) = (-16.128c^2 + 19.072c - 8.756)\mathrm{N_{fail}}(c)$$

$$\mathrm{P_{fail}}(c) = e^{-0.105\ln^3(f(c))+1.184\ln^2(f(c))-4.574\ln(f(c))+5.607}$$

$$\mathrm{P_{fail}}'(c) = (f'(c)/f(c))(-0.315\ln^2(f(c)) + 2.368\ln(f(c)) - 4.574)\mathrm{P_{fail}}(c)$$

Substituting these results in (5.5) yields OCP $= \{1, 0.91, 0.39, 0.04, 0\}$, with an expected cost to solution of 277 nodes. For comparison purposes only, a few other policies were selected at random. The policies and their expected costs appear in Table 12. According to the data in the table, WDS with the OCP should outperform other policies by 5% and 18%.

Table 13 shows the results of running WDS with these policies on randomly

TABLE 12. Theoretical comparison of OCP vs. other policies at depth 10.

Cutoff Policy	$\langle\textsc{wds}\rangle$	Savings
$\{1, 0.91, 0.39, 0.04, 0\}$	276.76	0%
$\{1, e^{-1}, e^{-2}, e^{-3}, 0\}$	292.56	5%
$\{1, 0.8, 0.6, 0.4, 0.2, 0\}$	339.21	18%
$\{1, 0.5, 0.25, 0.125, 0.0625, 0\}$	337.63	18%

generated trees for 10,000 trials. Notice that although the expected costs are higher in the actual runs, the percentage savings obtained in the experiments are close to the savings predicted by the formulas.

TABLE 13. Experimental comparison of OCP vs. other policies at depth 10.

Cutoff Policy	*Cost*	Savings
$\{1, 0.91, 0.39, 0.04, 0\}$	321.47	0%
$\{1, e^{-1}, e^{-2}, e^{-3}, 0\}$	332.71	3%
$\{1, 0.8, 0.6, 0.4, 0.2, 0\}$	378.18	15%
$\{1, 0.5, 0.25, 0.125, 0\}$	387.80	17%

TABLE 14. Fit parameters for trees of depth 15 and branching factor 2.

$N_{\text{succ}}(c)$	$e^{-16.609c^3+28.891c^2-19.713c+10.12}$
$N_{\text{fail}}(c)$	$e^{-13.046c^3+23.647c^2-16.822c+8.9266}$
$P_{\text{fail}}(c)$	$e^{-0.026\ln^3(f(c))+0.37\ln^2(f(c))-1.799\ln(f(c))+2.755}$

Repeating the process for trees of depth 15 yields an optimal cutoff policy of $\{1, 0.91, 0.52, 0.12, 0.05, 0.007, 0\}$, with an expected cost to solution of 11572 nodes. The parameters of the fits can be found on Table 14. As before, for comparison purposes only, a few other policies were selected at random; their expected costs appear in Table 15. Table 16 shows that the estimates of the cost in this case are much larger than the expected costs obtained by running the algorithm with the policy on randomly generated trees. We do not yet understand why there is such a disparity between the theoretical and experimental costs at depth 15. The savings reported by the experimental costs, however, are similar to those predicted by the equations.

TABLE 15. Theoretical comparison of OCP vs. other policies at depth 15.

Cutoff Policy	⟨WDS⟩	Savings
$\{1, 0.91, 0.52, 0.12, 0.05, 0.007, 0\}$	11572.15	0%
$\{1, e^{-1}, e^{-2}, e^{-3}, e^{-4}, e^{-5}, 0\}$	12090.63	4%
$\{1, 0.8, 0.6, 0.4, 0.2, 0\}$	17191.64	33%
$\{1, 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625, 0.0078125, 0\}$	12480.66	7%

TABLE 16. Experimental comparison of OCP vs. other policies at depth 15.

Cutoff Policy	Cost	Savings
$\{1, 0.91, 0.52, 0.12, 0.05, 0.007, 0\}$	4231.92	0%
$\{1, e^{-1}, e^{-2}, e^{-3}, e^{-4}, e^{-5}, 0\}$	4353.59	3%
$\{1, 0.8, 0.6, 0.4, 0.2, 0\}$	9726.31	56%
$\{1, 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625, 0.0078125, 0\}$	5002.37	15%

5.5.1 Performance Profiles

Given WDS, LDS, and DFS there is a way to predict how these algorithms will perform with respect to one another by sampling for the number of nodes explored

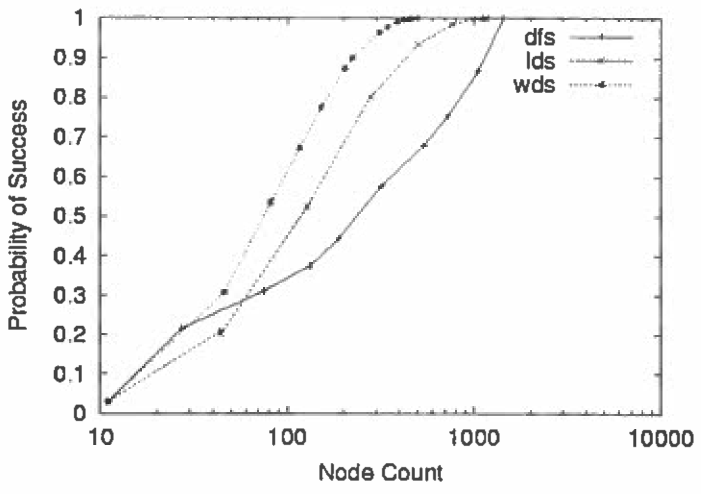


FIGURE 36. If mistakes tend to occur at the bottom of a tree of depth 10, DFS may outperform LDS, but WDS is likely to outperform both.

and the probability of success after exploring that many nodes. This information can then be graphed in a "performance profile". This performance profile shows the total number of nodes explored vs. the probability of success for each candidate cutoff value. In the case of DFS, there is a single cutoff value, so we plot instead how the probability of success increases as the number of nodes explored increases.

We obtained these performance profiles for LDS, DFS, and WDS with different weight distributions that represent different mistake probability distributions. Each data point in the profile shows the probability of success after exploring n nodes. The data points for LDS and WDS reflect how they would perform if the cutoff corresponding to n was the only one in the cutoff policy. That is, all three algorithms show how they would perform if they explored only n nodes.

Figure 36 shows the three curves when mistakes are more likely to occur at

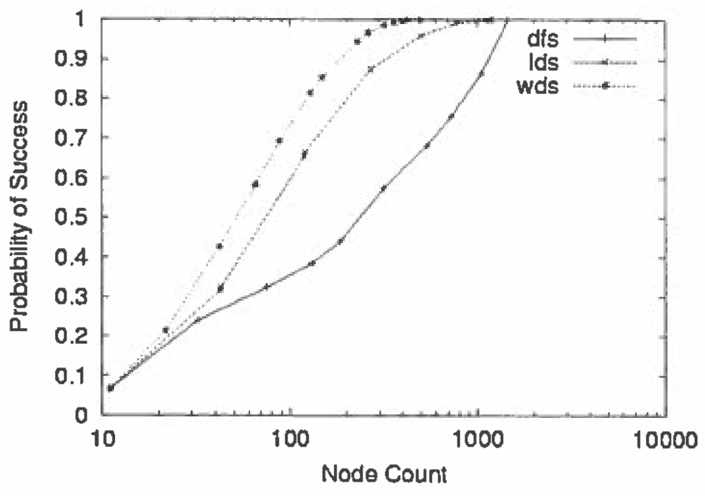


FIGURE 37. If mistakes tend to occur at the top of a tree of depth 10, LDS will outperform DFS, but WDS is again likely to outperform both.

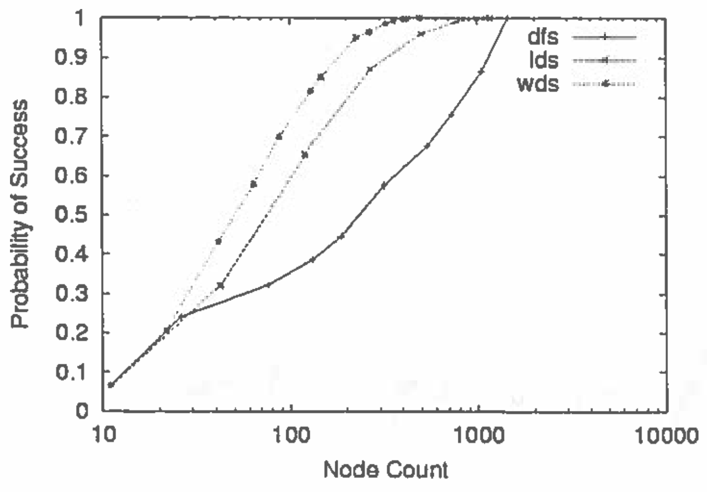


FIGURE 38. Profiles with uniform mistake probability for a tree of depth 10 are similar in shape to the profiles when mistakes tend to occur at the top.

the bottom of the search tree. Initially, DFS is marginally better than WDS and that LDS does worse than both WDS and DFS, as expected. As the number of nodes increases, however, LDS surpasses DFS. This may indicate that LDS may have some advantage by probing the space instead of backtracking, even when the mistakes are more likely to occur at the bottom.

Figure 37 shows the profiles when mistakes are more likely to occur at the top of the search tree. We see that this time LDS does better than DFS, as expected. WDS however, is still promising to outperform both LDS and DFS.

We also graphed the profiles for uniform mistake probability. Notice that the graph in Figure 38 looks extremely similar to the graph where the mistakes are made at the top. In fact, if it was not for the outstanding performance of DFS at the very initial portion of the search when the mistakes are made at the bottom, all three profiles would look roughly the same. The graphs confirm that WDS is the most promising algorithm regardless of where the mistakes are being made. The profile for WDS shows we should expect savings between 10% and 40% with respect to LDS, and between 75% and 85% with respect to DFS.

Table 17 shows values along the three different curves (top, bottom, and uniform mistake distribution) for 50% and 85% probability of success. Table 18 shows values along the three curves after exploring 50, 100, and 200 nodes. Observe that DFS seems to do pretty much the same regardless of where the mistakes are more likely to occur. One surprising result is that the performance of the algorithms does not vary greatly if mistakes are distributed towards the top or uniformly. We have chosen to assume a uniform mistake distribution for the remaining experiments.

Figure 39 shows that WDS seems to get better as depth increases. At depth

TABLE 17. Comparing number of nodes explored with different mistake probability distributions for a tree of depth 10 and branching factor 2.

mistakes	P_{succ}	DFS	LDS	WDS
top	0.5	225	175	52
	0.85	1000	250	155
bottom	0.5	225	130	75
	0.85	1000	215	190
uniform	0.5	225	75	50
	0.85	1000	250	160

TABLE 18. Comparing probability of success with different mistake probability distributions for a tree of depth 10 and branching factor 2.

mistakes	N_{succ}	DFS	LDS	WDS
top	50	0.27	0.37	0.49
	100	0.36	0.6	0.73
	200	0.47	0.8	0.92
bottom	50	0.28	0.24	0.33
	100	0.34	0.44	0.62
	200	0.46	0.68	0.87
uniform	50	0.28	0.37	0.5
	100	0.36	0.59	0.74
	200	0.47	0.79	92

15, the savings range between 35% and 40% with respect to LDS between the ranges of success probability of 0.3 and 0.95, and between 90% and 95% with respect to DFS in that same range. Notice also that the best savings we get by running LDS instead of running DFS is of 90% with probability of success of 0.8 at depth 15. The savings we could potentially obtain by running WDS are greater than any savings we can achieve running LDS.

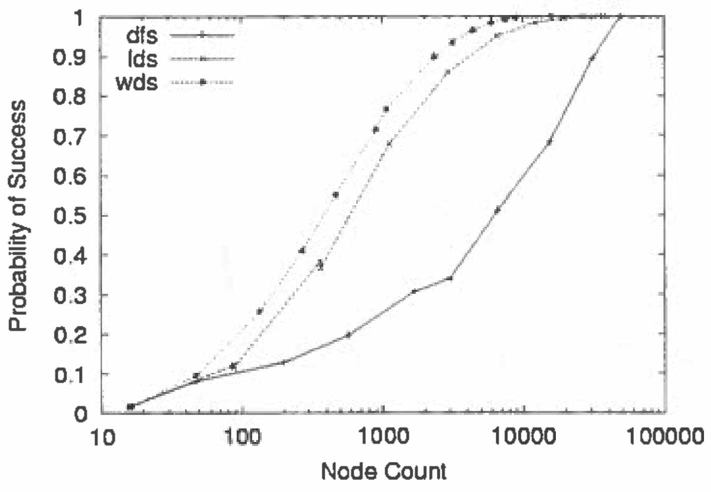


FIGURE 39. Profiles with uniform mistake probability for a tree of depth 15.

5.5.2 Experimental Results

5.5.2.1 Comparing WDS With LDS and DFS

The results in the previous section indicate that WDS should outperform LDS and DFS. In this section we show that WDS with the OCP confirms these results. We ran DFS, LDS, and WDS with the OCP on 10,000 randomly generated trees at

TABLE 19. Comparing probability of success with uniformly distributed mistake probability for a tree of depth 15 and branching factor 2.

N_{succ}	DFS	LDS	WDS
100	0.11	0.14	0.22
1000	0.25	0.65	0.75
1000	0.59	0.97	0.99

TABLE 20. Comparing number of nodes explored with uniformly distributed mistake probability for a tree of depth 15 and branching factor 2.

P_{succ}	DFS	LDS	WDS
0.5	6200	590	380
0.85	27000	2700	1700

TABLE 21. Comparing mean run time costs of WDS with LDS and DFS.

Depth	WDS	LDS	Savings	DFS	Savings
10	318.06	402.13	21%	501.79	37%
15	4296.14	5945.23	28%	15883.1	73%

depth 10 and depth 15. Table 21 shows the average cost results and savings of WDS using the OCP with respect to the other two algorithms. As depth increases, the savings with respect to DFS increase substantially, but the savings with respect to LDS increase only slightly.

Expected cost to solution is just one measure we may be interested in. Alternatively, we may be interested in the number of nodes required to solve certain percentage of the instances. The distribution of the cost in this case may be more important than the average cost to solution.

The research community has known for quite some time that the average cost of search algorithms is greatly affected by a few runs which take significantly longer to solve [32]. Moreover, recent studies on backtracking algorithms have shown that when applied to hard scheduling problems, they tend to exhibit "heavy tailed" distributions [20, 19]. Search algorithms that suffer from the phenomenon of heavy-tailed distributions exhibit large variability when run on different random instances. The large variability is an indicator that the distributions have no moments. If DFS, LDS, and WDS using the OCP suffer from heavy tailed distributions, we will be unable to provide meaningful statistics.

In order to find out whether DFS, LDS, and WDS using the OCP exhibit heavy-tailed distributions, we ran these algorithms on successively smaller sets of instances. If the mean cost grows with the size of the sets of instances, we may suspect the algorithms have infinite modes. We ran DFS on sets of 500, 1000, 5000, and 10,000 instances. The mean costs resulted in 506, 530, 501, and 501 respectively. The mean cost remained fairly stable across the different sets of instances. This suggests that DFS running on these randomly generated instances does not suffer from heavy-tailed distributions.

We ran LDS on other sets of 500, 1000, 5000, and 10,000 instances. The mean costs resulted in 426, 403, 396, and 387. The mean cost in LDS is not growing with the number of instances, on the contrary, it looks as though it is decreasing slowly. There seems to be no evidence of a heavy-tailed distribution. For WDS using the OCP we registered costs of 324, 291, 322, and 311, on other sets of 500, 1000, 5000, and 10,000 instances. The results for WDS also seem to indicate no presence of a heavy-tailed distribution.

We are now prepared to collect statistics on the run times of the three algorithms. Table 22 shows statistics for DFS, LDS, and WDS using the OCP run on 10,000 instances of depth 10. Table 23 shows the statistics collected in 10,000 instances of depth 15. WDS seems to be the all-around best-behaved algorithm, with lowest mean, median, and 90th percentile cost. As expected, LDS is next and DFS is last. Notice though, that LDS exhibits a larger variability than WDS or DFS with the hardest instance taking 2.5 times longer than the hardest instance for WDS and 4.1 times longer than the hardest instance for DFS. Surprisingly enough, DFS solves the hardest instances in less time than LDS and WDS. Because the median of both LDS and WDS is relatively low and the maximum 90th percentile are so high, it seems to indicate that the distributions for LDS and WDS are plagued with a lot of instances that have very low cost and a few instances have a very high cost. On the other hand, the run times of DFS seem to be more tightly distributed.

TABLE 22. Statistical measures on costs of WDS with the OCP, LDS, and DFS on trees of depth 10.

Algorithm	Mean	Std. Dev.	Max.	Min.	Median	90th Percentile
WDS	318.06	425.21	3250	11	143	779
LDS	402.13	645.97	8411	11	176	1024
DFS	501.79	501.39	2021	11	297	1239

TABLE 23. Statistical measures on costs of WDS with the OCP, LDS, and DFS on trees of depth 15.

Algorithm	Mean	Std. Dev.	Max.	Min.	Median	90th Percentile
WDS	4296.14	9047.56	104,991	16	1410	10,797
LDS	5945.23	14,473.5	172,101	16	1511	13,801
DFS	15,883.1	16,163.8	64,713	16	9237	10,801

5.5.2.2 Performance of WDS With the OCP vs. Other Policies

Unfortunately, with our approach, we are only able guarantee that the OCP is a local optimum. There are a couple of sources for "error". First, there is error in the sampling. Second, the calculus we have applied can only guarantee that the solution of the differential equation is an optimum, not necessarily a global one.

An interesting question, then, is how does the OCP we obtained analytically fare against the best policies found by Monte Carlo sampling the space of policies. We ran WDS on randomly chosen policies with length of at most d, for $d = 10$ and $d = 15$. We ran WDS first on a set of 10,000 policies for 20 runs on each policy and selected the best 20 policies and then ran WDS for 10,000 runs on each one. The results show that the OCP derived theoretically near-optimal with a cost of 7% above the best policy found by sampling the space of policies. The OCP has the seventh lowest cost in the table. The cost of the second best policy is 3% above the best, and the costs of the next three policies are around 5% above the best. Thus, a cost of 7% above the best found can be considered near-optimal.

Table 25 shows the best 20 policies and their associated costs for an average of 10,000 runs on trees of depth 15. This table also shows that the OCP ranks fifth in the table with cost of 13% above the best policy found.

5.6 Extensions

In the following two sections we present extensions to our approach. The first extension considers trees with larger branching factors. As long as the estimates of N_{succ}, N_{fail}, and P_{fail} account for the larger branching factor, the differential equation (5.5) can also be used in this case. The only change that needs to be made is in the random tree generator. The rest of the methodology remains intact.

In the second extension we show that out methodology to obtain optimal cutoff policies works when we replace simple node counts with arbitrary cost functions. We illustrate the method by replacing the node count functions with step functions where the cost is zero up to a bound B and one for each node explored

TABLE 24. Mean run time costs of WDS with best 20 policies at depth 10.

Cost	Policy
299.84	{1.000000, 0.101957, 0.026455, 0.000000}
307.45	{1.000000, 0.103030, 0.041567, 0.002773, 0.002469, 0.002067, 0.000758, 0.000576, 0.000000}
314.91	{1.000000, 0.117165, 0.000000}
315.88	{1.000000, 0.408098, 0.016906, 0.002747, 0.000607, 0.000000}
319.64	{1.000000, 0.648653, 0.147971, 0.011884, 0.004824, 0.002382, 0.001058, 0.000000}
320.63	{1.000000, 0.300238, 0.129889, 0.034504, 0.000000}
321.46	OCP = {1.0, 0.91, 0.39, 0.04, 0.0}
322.64	{1.000000, 0.921724, 0.720230, 0.159114, 0.025058, 0.001015, 0.000000}
324.73	{1.000000, 0.350930, 0.057579, 0.020750, 0.018496, 0.000000}
327.26	{1.000000, 0.204153, 0.062747, 0.040472, 0.000254, 0.000212, 0.000110, 0.000094, 0.000000}
329.23	{1.000000, 0.565363, 0.241233, 0.031652, 0.009894, 0.000000}
330.84	{1.000000, 0.351592, 0.108586, 0.048869, 0.011125, 0.004995, 0.001292, 0.000000}
334.77	{1.000000, 0.510999, 0.300885, 0.025008, 0.005139, 0.000000}
336.45	{1.000000, 0.698481, 0.145082, 0.066625, 0.000000}
336.60	{1.000000, 0.930000, 0.520000, 0.030000, 0.000000}
338.60	{1.000000, 0.295984, 0.125523, 0.061131, 0.010236, 0.000000}
340.34	{1.000000, 0.249269, 0.135495, 0.048001, 0.004293, 0.003081, 0.002313, 0.002206, 0.000000}
381.98	{1.000000, 0.360323, 0.054391, 0.053541, 0.053209, 0.014010, 0.009604, 0.000000}
394.05	{1.000000, 0.851095, 0.276955, 0.000000}
404.59	{1.000000, 0.506034, 0.371222, 0.274198, 0.101129, 0.035316, 0.000000}
410.27	{1.000000, 0.757354, 0.239927, 0.094898, 0.069917, 0.044889, 0.028685, 0.001943, 0.000000}

after the bound B. We show that the OCP with non-uniform cost (characterized by the step function) has a greater chance of success in B nodes than the OCP obtained with uniform cost, thus showing that our methodology can be used not only to minimize expected cost to solution but also to maximize probability of success.

TABLE 25. Mean run time costs of WDS with best 20 policies at depth 15.

Cost	Policy
3848.31	$\{1.000000, 0.618507, 0.116420, 0.019014, 0.002390, 0.000082, 0.000010, 0.000009, 0.000000\}$
4001.46	$\{1.000000, 0.271098, 0.042164, 0.011837, 0.001161, 0.000000\}$
4039.29	$\{1.000000, 0.215591, 0.020938, 0.003623, 0.000000\}$
4163.29	$\{1.000000, 0.743364, 0.270866, 0.045029, 0.014205, 0.002333, 0.001790, 0.000000\}$
4231.91	OCP $= \{1.0, 0.91, 0.52, 0.12, 0.05, 0.007, 0.0\}$
4252.77	$\{1.000000, 0.291582, 0.028233, 0.005603, 0.000000\}$
4287.10	$\{1.000000, 0.249269, 0.135495, 0.048001, 0.004293, 0.003081, 0.002313, 0.002206, 0.000000\}$
4525.17	$\{1.000000, 0.725772, 0.398219, 0.139808, 0.040861, 0.017700, 0.008955, 0.000000\}$
4636.77	$\{1.000000, 0.853571, 0.365738, 0.360295, 0.116228, 0.011448, 0.000000\}$
4644.19	$\{1.000000, 0.655995, 0.317268, 0.055951, 0.023631, 0.000000\}$
4761.19	$\{1.000000, 0.350930, 0.057579, 0.020750, 0.018496, 0.000000\}$
4850.93	$\{1.000000, 0.445856, 0.102995, 0.041117, 0.026055, 0.011394, 0.000000\}$
5108.13	$\{1.000000, 0.109136, 0.001144, 0.001013, 0.000396, 0.000213, 0.000000\}$
5314.70	$\{1.000000, 0.226362, 0.079137, 0.078306, 0.022843, 0.000000\}$
5404.99	$\{1.000000, 0.265749, 0.096618, 0.087898, 0.054165, 0.039858, 0.011807, 0.000000\}$
5651.00	$\{1.000000, 0.958648, 0.050802, 0.000000\}$
5823.18	$\{1.000000, 0.306294, 0.078853, 0.043805, 0.034386, 0.000000\}$
5892.25	$\{1.000000, 0.641244, 0.542524, 0.423270, 0.100461, 0.099017, 0.032859, 0.000000\}$
6061.96	$\{1.000000, 0.761503, 0.437299, 0.398145, 0.067022, 0.000000\}$
6084.58	$\{1.000000, 0.059104, 0.059004, 0.037542, 0.025726, 0.000000\}$
7974.58	$\{1.000000, 0.912097, 0.710259, 0.182504, 0.121012, 0.112378, 0.099879, 0.000000\}$

5.6.1 Results for Larger Branching Factors

The methodology we proposed works well for trees with a branching factor of two, but how does it scale to trees with larger branching factors? We present the results for trees with a branching factor of three, the extension to larger branching factors is straightforward.

Recall that the tree generator started out with signal strength, transformed it into a meaningful weight, and then used the weights to find out p and m. These two parameters determine the density and distribution of the goals. The weights are obtained as usual. We need to find out what changes in the relationship between the weights and the parameters of the search space as we increase the branching factor.

There are $2^3 = 8$ possible goal configurations since each child can be either good or bad. We are only interested in configurations where the parent is a good node. Table 26 shows the probabilities of occurrence of the 7 remaining configurations. Here, $p_1 = \frac{w_1}{w_1+w_2+w_3}$, $p_2 = \frac{w_2}{w_1+w_2+w_3}$, and $p_3 = \frac{w_3}{w_1+w_2+w_3}$.

TABLE 26. Subtree configurations at branching factor 3.

Good Nodes	Probability of Occurrence
left only	$p_1 3m^2$
left and middle	$p_3 3m$
left, middle, and right	$1 - 3m - 3m^2$
left and right	$p_2 3m$
middle	$p_2 3m^2$
middle and right	$p_1 3m$
right	$p_3 3m^2$

The probabilities are derived analogously to the case with branching factor 2 presented in Section 5.3.

TABLE 27. Fit parameters for trees of depth 10 and branching factor 3.

$\mathrm{N}_{\mathrm{succ}}(c)$	$e^{-147.698c^3+123.884c^2-36.456c+9.973}$
$\mathrm{N}_{\mathrm{fail}}(c)$	$e^{-154.819c^3+140.880c^2-43.353c+10.604}$
$\mathrm{P}_{\mathrm{fail}}(c)$	$e^{-0.122\ln^3(f(c))+2.876\ln^2(f(c))-22.678\ln(f(c))+59.845}$

As before, we want to produce performance profiles that will give some indication of what is the greatest savings we can expect from running WDS versus

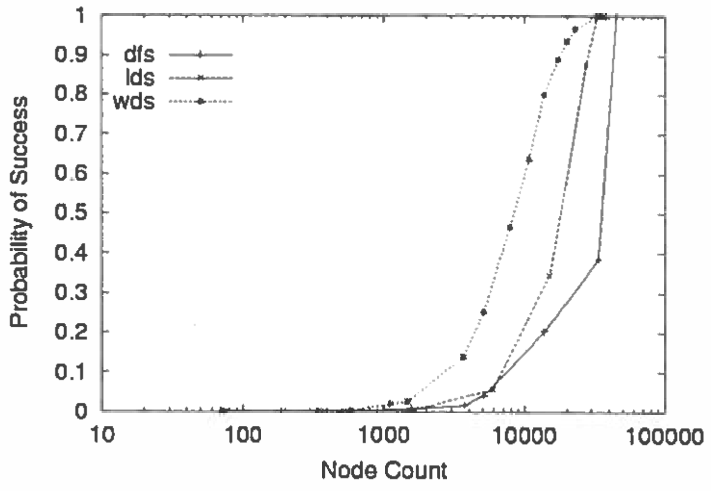


FIGURE 40. Profiles for a tree of depth 10 and branching factor 3 with uniform mistake probability distribution.

running LDS or DFS. We then will show the actual performance of WDS with the optimal cutoff policy and finally, we will show the distribution of the expected costs on 10,000 instances.

As in the previous section, we select cutoff values at random, run 10,000 trials and record the data for N_{succ}, N_{fail}, and P_{fail}. Next, we find best fits for these functions. We use the fits and their derivatives to compute the OCP via (5.5). The fit functions appear on Table 27. The resulting OCP is $\{1, 0.1, 0.01, 0.002, 0\}$.

WDS using the OCP beats both LDS and DFS. The average cost for WDS is 29,909 while the cost for LDS is 51,451 and for DFS is 36,727. WDS represents a savings of 42% with respect to LDS and 19% with respect to DFS. Notice that the cost of LDS is about one and a half times that of DFS. The larger branching factor is a great disadvantage for LDS since every time LDS explores a discrepancy it is forced to explore all discrepancies at the same depth regardless of the strength of their signal. In this case, where mistakes tend to occur uniformly with depth, LDS spends most of its time recovering from mistakes that occur near the top and neglects discrepancies that occur elsewhere. DFS recovers more gracefully.

The performance profiles for LDS, DFS, and WDS are shown in Figure 40. Table 28 shows the probability of success of each of the three algorithms after exploring 6000, 10000, and 16000 nodes. Notice that the probability of success of WDS is substantially higher than the probability of the other two algorithms. As expected, as branching factor increases, the effectiveness of LDS decreases. This is also confirmed by Table 29 which shows the number of nodes each algorithm has to explore to reach a 35%, 50%, and 85% rate of success.

TABLE 28. Comparing probability of success with uniformly distributed mistake probability for a tree of depth 10 and branching factor 3.

N_{succ}	DFS	LDS	WDS
6000	0.07	0.06	0.33
10000	0.16	0.22	0.6
16000	0.19	0.32	0.8

TABLE 29. Comparing number of nodes explored with uniformly distributed mistake probability for a tree of depth 10 and branching factor 3.

P_{succ}	DFS	LDS	WDS
0.35	28000	16000	6100
0.5	36000	18000	8300
0.85	42000	27000	17000

We also collected statistics on the distribution of costs of the three algorithms over 10,000 instances. Table 30 shows statistics for DFS, LDS, and WDS using the OCP on trees of depth 10 and branching factor 3. Again, we see that the cost distribution for DFS looks tighter in the sense that there is not a very large variability between the costs of exploring the different instances. WDS and LDS, on the other hand, seem to have cost distributions that are skewed towards a few hard instances. Still, the mean and median costs are much lower in WDS than in the other two algorithms.

TABLE 30. Statistical measures on costs of WDS with the OCP, LDS, and DFS.

Algorithm	Mean	Std. Dev.	Max.	Min.	Median	90th Percentile
WDS	29,908.8	21,586.6	135,929	132	16,529	59,152
LDS	51,451	36,673	206,271	208	43,755	134,260
DFS	36,727.9	14,568	83,657	176	42,875	44,656

Following the analysis for branching factor 2, we did a Monte Carlo sampling of the space of possible cutoff policies. The results of the best 20 policies and their associated costs is shown on Table 31. The OCP ranks seventh with a cost of 13% above the best policy found. The results we obtain for depth 10 and branching factor of 3 are similar in structure to the results we obtained with a branching factor of 2. We see that there are two policies that have the lowest cost within a 2% of each other. Then, one policy with cost around 5% above the lowest found and then there are a handful of policies with cost around 10% above the lowest found and then the OCP with cost about 13% above the lowest found.

TABLE 31. Mean run time costs of WDS with best 20 policies at depth 10 and branching factor 3.

Cost	Policy
21305.74	{1.000000, 0.005110, 0.000147, 0.000041, 0.000017, 0.000011, 0.000005, 0.000000}
21774.12	{1.000000, 0.334680, 0.002707, 0.000042, 0.000000}
22494.23	{1.000000, 0.007812, 0.000147, 0.000138, 0.000000}
23343.48	{1.000000, 0.613077, 0.007940, 0.001388, 0.000913, 0.000464, 0.000043, 0.000037, 0.000000}
23346.46	{1.000000, 0.881345, 0.639681, 0.435177, 0.015028, 0.003142, 0.000253, 0.000031, 0.000000}
23896.45	{1.000000, 0.408098, 0.016906, 0.002747, 0.000607, 0.000000}
24558.30	OCP = {1.0, 0.1, 0.01, 0.002, 0.0}
24630.92	{1.000000, 0.457697, 0.318313, 0.287678, 0.057839, 0.001208, 0.000104, 0.000040, 0.000000}
24685.77	{1.000000, 0.118350, 0.040347, 0.002820, 0.000077, 0.000002, 0.000001, 0.000001, 0.000000}
25070.94	{1.000000, 0.411166, 0.015887, 0.004218, 0.001777, 0.000000}
25464.27	{1.000000, 0.028633, 0.003895, 0.000000}
25862.07	{1.000000, 0.388510, 0.150736, 0.011502, 0.000050, 0.000000}
26006.70	{1.000000, 0.947783, 0.006669, 0.003588, 0.000000}
26923.44	{1.000000, 0.510999, 0.300885, 0.025008, 0.005139, 0.000000}
27010.78	{1.000000, 0.720496, 0.042478, 0.006511, 0.004400, 0.000312, 0.000000}
28585.16	{1.000000, 0.516739, 0.305036, 0.039879, 0.007685, 0.006585, 0.000699, 0.000125, 0.000000}
28725.52	{1.000000, 0.102880, 0.010876, 0.000000}
29013.16	{1.000000, 0.743364, 0.270866, 0.045029, 0.014205, 0.002333, 0.001790, 0.000000}
29636.64	{1.000000, 0.701797, 0.014743, 0.000000}
32568.68	{1.000000, 0.415026, 0.262861, 0.018776, 0.010662, 0.008724, 0.001759, 0.000743, 0.000000}
33283.64	{1.000000, 0.540681, 0.540658, 0.459826, 0.036205, 0.009636, 0.007797, 0.000000}

5.6.2 Non-Uniform Cost

In this section, we begin to answer the question of how to obtain the OCP when the cost of expanding nodes varies in time. The expected cost to solution is formulated as in the case where uniform cost is assumed, except that the node count is replaced for the corresponding cost function. This provides a simplified but general way to take into account the cost of reasoning. In cases where the cost function is a step function, it also provides a means to maximize the probability of success within each step.

Assume the cost is given as a function of node $\mathrm{C}(n)$. The cost of expanding $\mathrm{N}_{\mathrm{succ}}(c)$ nodes at cutoff c is:

$$\mathrm{K}_{\mathrm{succ}}(c) = \int_1^{\mathrm{N}_{\mathrm{succ}}(c)} \mathrm{C}(n)dn \tag{5.11}$$

We define the cost of expanding $\mathrm{N}_{\mathrm{fail}}(c)$ nodes in a similar way:

$$\mathrm{K}_{\mathrm{fail}}(c) = \int_1^{\mathrm{N}_{\mathrm{fail}}(c)} \mathrm{C}(n)dn \tag{5.12}$$

Recall that:

$$\langle \mathrm{WDS} \rangle = \int_1^{\infty} di\ (\mathrm{N}_{\mathrm{fail}}(\mathcal{C}_i) - \mathrm{N}_{\mathrm{succ}}(\mathcal{C}_i))\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_i) + \mathrm{N}_{\mathrm{succ}}(\mathcal{C}_i)\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_{i-1}) \tag{5.13}$$

is the general expression for the expected cost to solution with uniform cost of expanding a node (we assume the cost equals the number of nodes expanded, but it could easily be a constant times the number of nodes expanded). It is easy to see that we can replace $\mathrm{N}_{\mathrm{fail}}$ by $\mathrm{K}_{\mathrm{fail}}$ and $\mathrm{N}_{\mathrm{succ}}$ by $\mathrm{K}_{\mathrm{succ}}$ in (5.13) to obtain the new expression in terms of the cost functions:

$$\langle \mathrm{WDS} \rangle = \int_1^{\infty} di\ (\mathrm{K}_{\mathrm{fail}}(\mathcal{C}_i) - \mathrm{K}_{\mathrm{succ}}(\mathcal{C}_i))\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_i) + \mathrm{K}_{\mathrm{succ}}(\mathcal{C}_i)\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_{i-1}) \tag{5.14}$$

and derive cutoff policies as before but using this new cost function.

For instance, assume a fixed node bound B, is given, such that nodes explored before the node bound are essentially free and nodes outside the bound have unit cost. The cost function may look like:

$$\mathrm{C}(n) = \begin{cases} 0 & \text{for } n < B \\ 1 & \text{for } n \geq B \end{cases}$$

Replacing the cost in (5.11) and in (5.12) we get:

$$\mathrm{K}_{\mathrm{succ}}(c) = \begin{cases} 0 & \text{if } \mathrm{N}_{\mathrm{succ}}(c) < B \\ \int_B^{\mathrm{N}_{\mathrm{succ}}(c)} dn = \mathrm{N}_{\mathrm{succ}}(c) - B & \text{otherwise} \end{cases} \tag{5.15}$$

and

$$\mathrm{K}_{\mathrm{fail}}(c) = \begin{cases} 0 & \text{if } \mathrm{N}_{\mathrm{succ}}(c) < B \\ \int_B^{\mathrm{N}_{\mathrm{fail}}(c)} dn = \mathrm{N}_{\mathrm{fail}}(c) - B & \text{otherwise} \end{cases} \tag{5.16}$$

Notice that $\langle\textsc{wds}\rangle$ is zero if the cutoff instructs WDS to explore less than B nodes. Otherwise, replacing (5.15) and (5.16) in (5.14) we obtain:

$$\langle\textsc{wds}\rangle = \int_1^\infty di\ (\mathrm{N}_{\mathrm{fail}}(\mathcal{C}_i) - \mathrm{N}_{\mathrm{succ}}(\mathcal{C}_i))\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_i) + (\mathrm{N}_{\mathrm{succ}}(\mathcal{C}_i) - B)\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_{i-1}). \tag{5.17}$$

This equation is similar to (5.3) except for the fact that B appears in the last term. Following the steps in Section 4.3, we obtain:

$$\int_1^\infty di\ ((\mathrm{N}_{\mathrm{fail}}'(\mathcal{C}_i) - \mathrm{N}_{\mathrm{succ}}'(\mathcal{C}_i))\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_i) + (\mathrm{N}_{\mathrm{fail}}(\mathcal{C}_i) - \mathrm{N}_{\mathrm{succ}}(\mathcal{C}_i))\mathrm{P}_{\mathrm{fail}}'(\mathcal{C}_i) +$$
$$\mathrm{N}_{\mathrm{succ}}'(\mathcal{C}_i)\mathrm{P}_{\mathrm{fail}}(\mathcal{C}_{i-1}) + (\mathrm{N}_{\mathrm{succ}}(\mathcal{C}_{i+1}) - B)\mathrm{P}_{\mathrm{fail}}'(\mathcal{C}_i)) = 0.$$

As before, let

$$T(\mathrm{N}_{\mathrm{succ}}, \mathrm{N}_{\mathrm{succ}}', \mathrm{N}_{\mathrm{fail}}, \mathrm{N}_{\mathrm{fail}}', \mathrm{P}_{\mathrm{fail}}, \mathrm{P}_{\mathrm{fail}}', \mathcal{C}_i) =$$

$$(\mathrm{N_{fail}}'(\mathcal{C}_i) - \mathrm{N_{succ}}'(\mathcal{C}_i))\mathrm{P_{fail}}(\mathcal{C}_i) + (\mathrm{N_{fail}}(\mathcal{C}_i) - \mathrm{N_{succ}}(\mathcal{C}_i))\mathrm{P_{fail}}'(\mathcal{C}_i).$$

Substituting for T and solving for $\mathrm{N_{succ}}(\mathcal{C}_{i+1})$:

$$\mathrm{N_{succ}}(\mathcal{C}_{i+1}) = B - \frac{T(\mathrm{N_{succ}}, \mathrm{N_{succ}}', \mathrm{N_{fail}}, \mathrm{N_{fail}}', \mathrm{P_{fail}}, \mathrm{P_{fail}}', \mathcal{C}_i) - \mathrm{N_{succ}}'(\mathcal{C}_i)\mathrm{P_{fail}}(\mathcal{C}_{i-1})}{\mathrm{P_{fail}}'(\mathcal{C}_i)}. \tag{5.18}$$

Notice that this expression is the same as the expression for the cutoff policy with uniform cost, except for the extra factor of B nodes. Since the first cutoff is 1 according to our border conditions, and since $\mathrm{N_{fail}}(1) = \mathrm{N_{succ}}(1) = d+1$ the cost of expanding nodes in this first iteration is zero. The expression for the second cutoff reduces to:

$$\mathcal{C}_2 = \mathrm{N_{succ}}^{-1}(B).$$

The cutoff policy instructs WDS to explore the first B nodes and then fall back to the original equation for the cutoff policy.

Given a node bound B, then, the interesting test is to see if WDS with the non-uniform cost optimal cutoff policy has a higher chance of success than WDS with the uniform cost optimal cutoff policy after exploring exactly B nodes. Table 32 shows theoretical and experimental results for trees at depth 10 and branching factor 2, for different bounds. Although the predicted probability of success, shown under the column labeled "Theoretical $\mathrm{P_{succ}}$", is always substantially higher than the average probability of success after 10,000 runs on randomly generated trees, the data shows that the relative difference between the predicted probability of success of the uniform and the non-uniform OCP is roughly preserved in the experimental results. We do not presently understand the reason for the disparity. Notice also that the WDS with the non-uniform OCP has a much higher rate of success than with the uniform OCP up until 250 nodes are explored, and then both policies have roughly the same rate of success.

TABLE 32. Comparison of success probability of the uniform cost OCP vs. the non-uniform cost OCP given different node bounds at depth 10 and branching factor 2.

	Bound	Policy	Theoretical P_{succ}	Experimental P_{succ}
uniform	50	{1, 0.91, 0.39, 0.04, 0}	0.46	0.18
non-uniform	50	{1, 0.32, 0.002, 0}	0.54	0.25
uniform	80	{1, 0.91, 0.39, 0.04, 0}	0.61	0.3
non-uniform	80	{1, 0.22, 0.01, 0}	0.69	0.35
uniform	120	{1, 0.91, 0.39, 0.04, 0}	0.77	0.41
non-uniform	120	{1, 0.15, 0}	0.81	0.44
uniform	250	{1, 0.91, 0.39, 0.04, 0}	0.95	0.57
non-uniform	250	{1, 0.04, 0}	0.96	0.56
uniform	300	{1, 0.91, 0.39, 0.04, 0}	0.97	0.61
non-uniform	300	{1, 0.02, 0}	0.98	0.61

We also did experiments for trees with depth 10 and branching factor 3. Table 33 shows the experimental results averaged over 10,000 runs, for different node bounds. As before, the best chance of success is attained early in the search. After about 15,000 nodes, both policies perform roughly alike. This is no surprise since the cutoff values that correspond to 15,000 and 20,000 are close in value to the last cutoff value in the uniform cost OCP.

To summarize, given a node bound B, search first all nodes with weight greater than or equal to the weight that corresponds to searching B nodes. Then continue searching according to the non-uniform cost OCP.

The result may not be as clear cut in the case where the cost function has several steps, for instance, in the case where the cost varies between several bounds.

TABLE 33. Comparison of success probability of the uniform cost OCP vs. the non-uniform cost OCP given different node bounds at depth 10 and branching factor 3.

	Bound	Policy	Experimental P_{succ}
uniform	10,000	$\{1, 0.1, 0.01, 0.002, 0\}$	0.23
non-uniform	10,000	$\{1, 0.15, 0\}$	0.34
uniform	15,000	$\{1, 0.1, 0.01, 0.002, 0\}$	0.45
non-uniform	15,000	$\{1, 0.004, 0\}$	0.43
uniform	20,000	$\{1, 0.1, 0.01, 0.002, 0\}$	0.58
non-uniform	20,000	$\{1, 0.002, 0\}$	0.56

We speculate that the values considered by the OCP will depend on where the node bounds lie along the performance profile. For instance, if two bounds are such that the probability of success for both is essentially the same, the OCP will only consider one of the two values. Verifying our intuitions is a subject for future research.

CHAPTER VI

NUMBER PARTITIONING

Given a finite set of elements and a size for each of the elements in the set, the partitioning problem [14] is to find out whether there is a subset of the elements such that the sum of the sizes of the elements in the subset equals the sum of the remaining elements of the set. Unless the sizes add up to an even number, a perfect partition will not exist, the original definition is modified slightly to allow for a difference of 0 or 1 between the subset sums. In the number partitioning problem we assume the elements are numbers and their size is given by the quantities they represent.

The number partitioning problem is NP-complete, even if the numbers are ordered. Thus, we can assume that the numbers are sorted in descending order and that they form a sequence. Although number partitioning is NP-complete, it can be solved in pseudo-polynomial time. If A is a sequence of positive integers whose sum is S, then the optimal residue (difference between the the two partitions) can be determined in time polynomial in nS, where n is the length of A. The pseudo-polynomial algorithm is based on the observation that for any subset $A' \subseteq A$, the sum $\sum_{a \in A'} a$ must be an integer between 1 and S. Dynamic programming is used to ask whether the sum is equal to 1, or 2, up to $\lfloor S/2 \rfloor$. The partial sum closest to $S/2$ gives the minimal residue. This pseudo-polynomial algorithm, which requires time and space at least linear in S is of little practical value when S is large.

```
      8 7 6 5 4

(8       ) (       )
(8       ) (7      )
(8       ) (7 6    )
(8 5     ) (7 6    )
(8 5 4   ) (7 6    )
```

FIGURE 41. The greedy heuristic yields a residue of 4 on the sequence $(8, 7, 6, 5, 4)$.

6.1 Heuristics for Number Partitioning

6.1.1 The Greedy Heuristic

Jones and Beltramo [27] describe a greedy heuristic in which the two partitions are initially empty and the elements of the given instance are added, one at a time, to the partition with the smaller sum. If there is a tie, as in the beginning when both partitions are empty, a partition is chosen at random. This process continues until no elements are left.

Borrowing an example from Korf [35], suppose we are given the sequence $(8, 7, 6, 5, 4)$. Figure 41 shows how numbers would be partitioned according to the greedy heuristic. There is a perfect partition in this case, which consists of placing 8 and 7 in one partition and 6, 5, and 4 in the other partition. The greedy heuristic yields a partition with a residue of 4, far from optimal.

```
8 7 6 5 4    1 6 5 4
  6 5 4 1      1 4 1
    4 1 1        3 1
      3 1          2
```

FIGURE 42. The KK heuristic yields a residue of 2 on the sequence $(8, 7, 6, 5, 4)$.

6.1.2 The Karmakar-Karp Heuristic

A better heuristic emerges from the realization that it is not necessary to maintain both subset sums in order to minimize the residue. It suffices to keep track of the residue between the two partitions.

The set-differencing method of Karmakar and Karp [29] replaces the two largest numbers by their difference, effectively committing to placing these two numbers in different subsets, while deferring the decision about which subset each will be placed in. The difference is treated as any other number, inserting where it belongs in the sequence of remaining numbers. This operation, called differencing, is applied recursively to the sequence of remaining numbers. Figure 42 shows the KK heuristic working on the sequence $(8, 7, 6, 5, 4)$. Each application of the heuristic on the left sequence results on the sequence on the right. The left column shows the sequence of remaining numbers while the column on the right shows the result of the differencing operation. In this particular case, the KK heuristic does better than the greedy heuristic, but it still fails to find the optimal partition. Korf [35] shows that for problems with random integers distributed uniformly between 0 and 10 billion, the KK heuristic finds much better solutions, on average, than the greedy heuristic.

6.2 Complete Karmakar-Karp

Given that none of the heuristics lead to a perfect partition, if we want to find an optimal solution, the obvious algorithm is to search a binary tree, where the left branch assigns the next unassigned number to one set and the right branch assigns it to the other set. If we reach a terminal node whose subset difference is 0 or 1, representing the perfect partition, we terminate the search. Otherwise, we return the best difference found during the search.

Korf [35] observes that there are several optimizations that may reduce the search space. If we reach a node where the difference between the current subset sums is greater than or equal to the sum of all the remaining unassigned numbers, the best we can do is assign the remaining numbers to the smaller subset. The second optimization stems from the fact that the number partitioning problem is symmetric. Once we choose to assign the first number to one partition we need not consider the alternative of assigning that number to the other partition. The same consideration is valid when the sum of the subsets is the same, so we should always break ties by deciding to put the first number in the first subset, for instance. Other optimizations are possible but they may depend on the heuristic used.

The complete Karmakar-Karp algorithm CKK, searches a binary tree, where each node replaces the two largest numbers [35]. The left branch replaces them by their difference, while the right branch replaces them by their sum. The difference is inserted in order in the remaining sequence, while the sum is simply appended to the head of the sequence since it is always the largest element of the sequence. Search proceeds in depth-first order, so the first solution found by CKK is the KK solution. Figure 43 shows CKK running on the sequence $(8, 7, 6, 5, 4)$. Note that

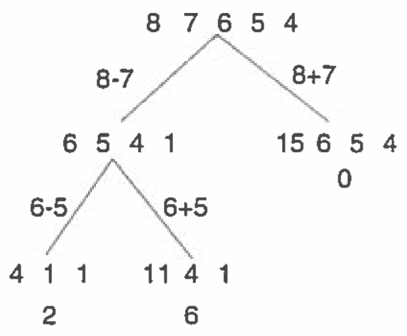


FIGURE 43. The CKK algorithm finds the optimal partition for the sequence $(8,7,6,5,4)$.

the subtrees underneath nodes where the largest number is greater than or equal to the sum of the remaining unassigned numbers are pruned. Korf shows that the CKK algorithm is more efficient than doing depth-first search with the greedy heuristic.

6.3 Improved Limited Discrepancy Search

Korf realized that by doing a small amount of extra accounting one can modify LDS so that, under certain conditions, it generates only paths with exactly k discrepancies, instead of generating paths with less than or equal to k discrepancies. This is done by keeping track of the remaining depth to be searched. If it is less than or equal to the number of discrepancies, then it suffices to explore only right branches beneath that node (since paths with left branches have been explored in previous iterations). Korf calls this modified algorithm improved limited discrepancy search ILDS [36]. He shows that in the best case, ILDS saves a factor of (d+2)/2 nodes.

Korf compared the performance of CKK, LDS and ILDS on number partition-

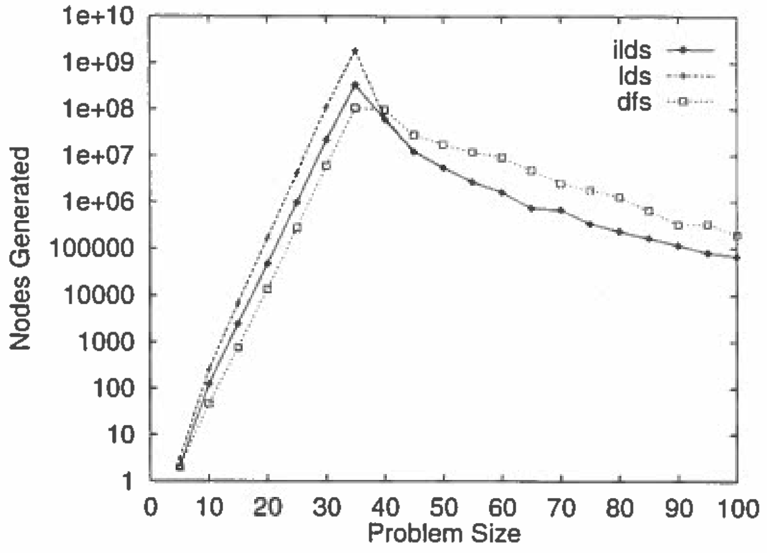


FIGURE 44. ILDS and LDS beat DFS on the easy region, but DFS beats both in the hard region.

ing. For his experiments, he chose random ten-digit integers uniformly distributed between 0 and 10 billion. He varied the number of integers partitioned from 5 to 100 in increments of 5, and for each data point, he averaged 100 trials. We reproduced his results which are shown in Figure 44.

Notice that the problem difficulty for all three algorithms, increases with increasing problem size. There is a peak in the average cost at around 35 and after that the cost decreases. This peak in the cost is related to the phase-transition phenomenon observed in problems as they transition from being unsatisfiable to being satisfiable [8, 43, 62, 31, 55]. Perfect partitions do not exist for small problems (before the peak) so the entire tree must be searched. The larger the problem, the larger the tree, so more nodes are generated.

It has been shown elsewhere [15] that the phase transition for number partitioning occurs when there are approximately as many numbers as the $\log_2(N)/0.96$

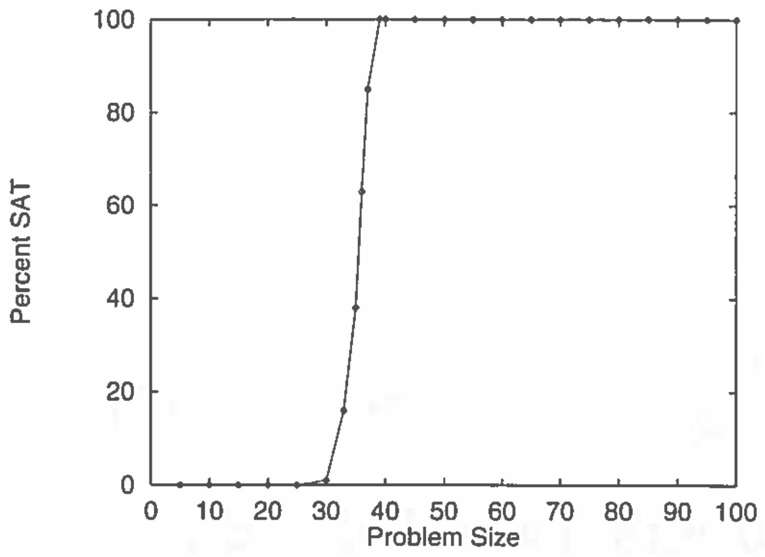


FIGURE 45. The phase transition with numbers drawn from a uniform random distribution between 0 and 10^{10}.

where N is the largest number. For numbers of size between 0 and 10 billion, the phase transition occurs at around 35 (34.6 to be exact), which is exactly where we see a peak in the average cost to solution in Figure 44. Figure 45 shows the percentage of satisfiable instances for problem sizes between 5 and 100. Note that the transition occurs quickly. If the problem size is smaller than 25, then the instance is almost certainly unsatisfiable, and if it is larger than 39, then the instance is almost certainly satisfiable. The data points in the figure that belong to the transition correspond to problem sizes of 30, 33, 35, 36, 37, and 39.

6.4 Robust Cutoff Policies vs. ILDS

We saw in Section 5.4, that the performance of LDS may be improved by skipping some iterations. We established that on randomly generated trees, the robust cutoff policy ROCP is near-optimal and easy to construct. The ROCP

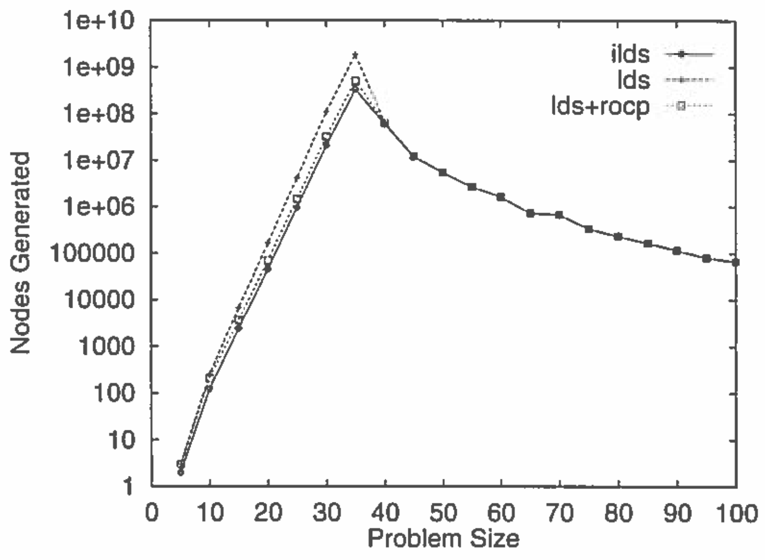


FIGURE 46. ROCP improves on LDS, but ILDS still outperforms both LDS and LDS using the ROCP.

includes all iterations up to $d/3$ then every other iteration up to $d/2$ and finally jumps to d. We were curious as to how the ROCP would perform with respect to ILDS on the number partitioning problems. Figure 46 shows that the ROCP improves on the performance of LDS but not enough to outperform ILDS. Korf's improvement amounts to a savings of 50%, while the ROCP amounts to a savings of 45%. Notice also that for problem sizes larger than 40, there are no savings. Both LDS and ILDS succeed in the early iterations.

Korf's improvement is not incompatible with the ROCP. However there is a subtlety in the implementation. Korf's improvement causes ILDS to explore only right branches when the remaining depth is equal to or less than the number of discrepancies. This works if we know that ILDS explored all paths at one lesser discrepancy in the previous iterations. Since the ROCP instructs ILDS to skip some iterations, the condition under which only the right branch is generated needs to

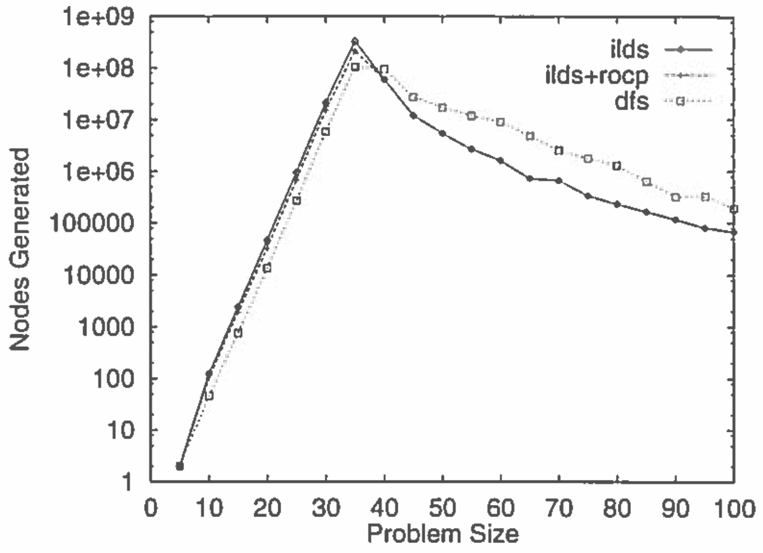


FIGURE 47. ROCP improves on ILDS in the hardest region, but DFS still outperforms both LDS and LDS using the ROCP.

be revised. At a discrepancy count of k, we do not want to generate paths with l or less discrepancies when l is the discrepancy count corresponding to the previous cutoff. Thus, we can run ILDS using the ROCP by changing the condition to check whether the remaining depth is less than or equal to the previous cutoff (in exploring the first cutoff, there is no redundancy, so the condition plays no role).

We ran ILDS using the ROCP, and as Figure 47 shows, ILDS using the ROCP beats ILDS with the standard policy. The savings are not enough to outperform DFS (CKK) in the hardest problems. ILDS with the ROCP outperforms ILDS by 30%. Again, the savings disappear as problems become easy.

Figure 48 shows a magnified view of the region around the phase transition. Notice that DFS outperforms the other algorithms when the problem size is 35, and most instances are unsatisfiable, but the performance DFS degrades at around 40, where all the instances are satisfiable.

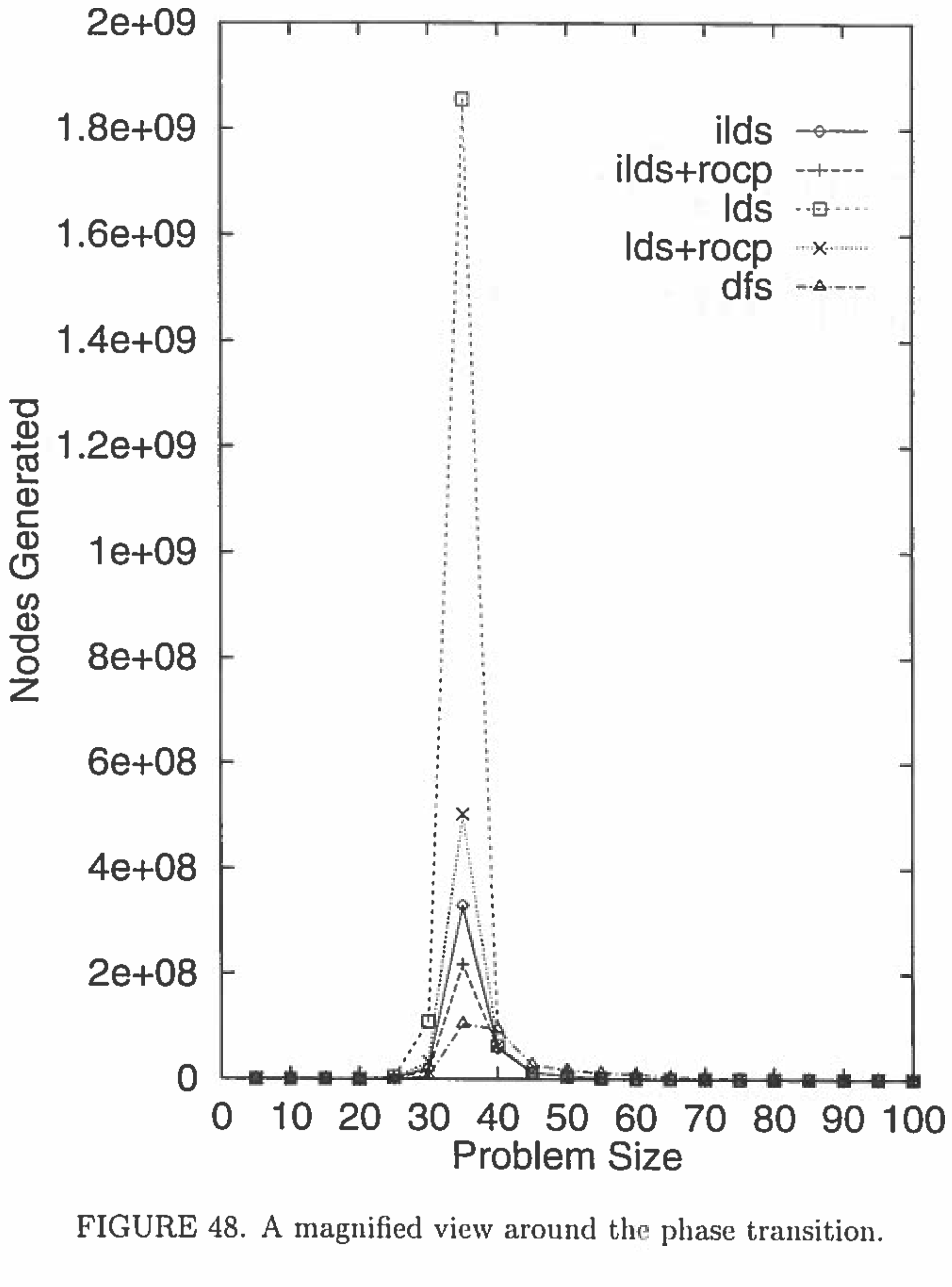


FIGURE 48. A magnified view around the phase transition.

6.5 Weighted Discrepancy Search

In order to run weighted discrepancy search on the number partitioning problems, we need to determine the weights. In LDS and ILDS the "weights" are constant, since all right branches are considered equal. We need to determine weights that will prefer right branches that are more likely to contain solutions.

Once suitable weights are found, we need to collect profiling information on the number of nodes explored and probability of success at candidate cutoff values. Then we will find best fits to these functions, calculate their first derivatives, and use the differential equation (5.5) in Section 5.2 to compute the optimal cutoff policy OCP. Finally, we will run WDS using the OCP and compare its performance to ILDS.

When problems are too large to profile in reasonable time, it may be possible to profile smaller problems, obtain cutoff policies for the smaller problems, and then extrapolate the policies to policies for the larger instances. We intend to obtain policies for problem instances with numbers between 0 and 10 billion by extrapolating the policies obtained for the smaller problems.

The problem instances depend on two parameters: the size of the largest element in the set and the number of elements in the set. We quantify the size of the largest element in terms of the number of digits D. Let N denote the number of elements to be partitioned. Since the hardness of the instances depends on N, we decided to establish a base point, $b = \lfloor \log_2(10^D) \rfloor$, and vary the N with respect to this base point. The *ratio* r between N and the base point b determines where the problem instance lies in the phase transition. For instance, if $D = 10$ and $N = 60$, then $r = 1.82$, and all instances of the problem are satisfiable. If $D = 10$

and $r = 1$ then we know that most of the instances are likely to be unsatisfiable.

6.5.1 Weights for Number Partitioning

The KK heuristic is, in some sense, trying to make numbers as small as possible as fast as possible, so that there is more choice on where to put numbers, and so increase the chance of finding a partition. In some sense, KK captures all we know about number partitioning and the CKK does not aim at comparing two different choices that are located in two different right branches at different depths.

The CKK weight is defined as $\frac{a_0 - a_1}{a_0 + a_1}$. The best weight we found after examining a few other weighting schemes is $\frac{a_2}{a_0 + a_1}$. This is some measure of how a_2 compares to $a_0 + a_1$. If $a_0 + a_1$ is close to a_2, then we have a greater chance of reducing or at least maintaining the residue. If the sum is much larger, then it is probably a bad choice.

Here is a list of the other weight functions we tried and a brief explanation of why we tried them.

1. $\frac{a_0 - a_1}{(a_0 + a_1)d}$, where d is the current depth. Since KK reduces the problem space, we assumed that dividing by the current depth d could produce some improvement.

2. $\frac{a_0 - a_1}{(a_0 + a_1)n}$, where n is the length of the sequence that remains to be partitioned. For the same reason we tried the previous scheme, we tried dividing by n but we did not see an improvement.

3. $\frac{a_2}{(a_0 + a_1)d}$, dividing the best scheme so far by d did not improve the weights.

4. $\frac{a_2}{(a_0 + a_1)n}$, dividing by n did not improve either.

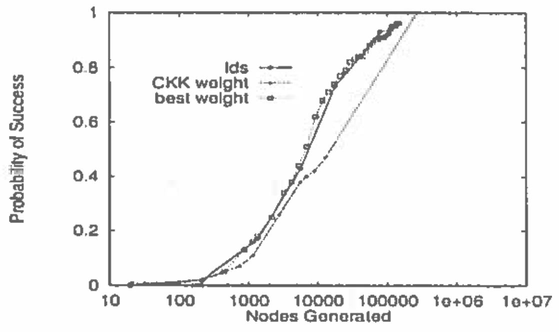
5. $(\frac{a_0-a_1}{a_0+a_1})^{\frac{n}{10}}$, varying the CKK weight nonlinearly with the length of the sequence did not help either.

6. $\frac{(a_2-a_3)}{(a_0+a_1-a_2)}$, here we tried to do some lookahead, by trying to compute the ratio between the current right branch and decision to keep going left one more level. This weighting scheme works worse than the others.

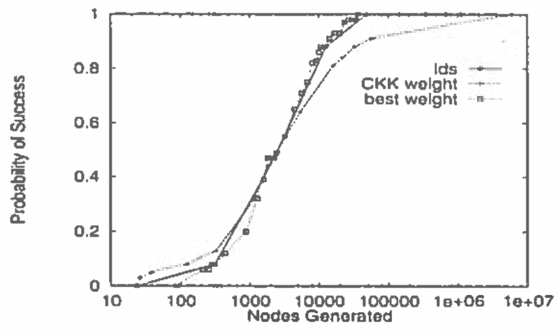
7. $\frac{(a_2+a_3)}{(a_0+a_1)}$, this looked as good as the one we chose for best weight, but we decided to go with the simpler calculation instead.

It was not necessary to obtain policies for each weight function we tried. Performance profiles offer a good way to quickly get an estimate of how well WDS performs with a particular weighting scheme. If the profile of WDS explores fewer nodes with higher probability of success than ILDS, then WDS should outperform ILDS. Figure 49 shows the behavior of WDS with the CKK weight and the best weight so far, with respect to changes in r. Observe that when $r = 1.2$ and when $r = 1.82$ the WDS with the CKK weight is a clear loser. When $r = 1.4$, however, it starts outperforming both ILDS and WDS with what we have called the "best weight" found so far. The potential savings, however, vanish as the number of nodes increases.

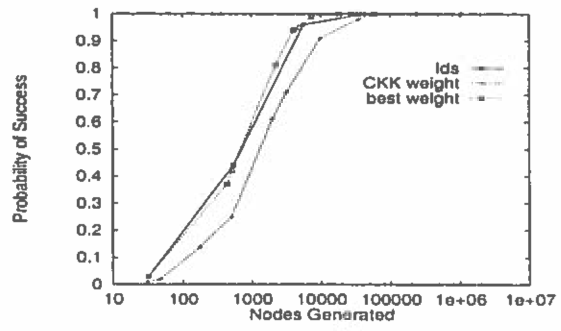
It is worth saying that the weight function in WDS may not have a direct dependence on the distance to the fringe, so it may not be possible to translate Korf's improvement of LDS in the new setting. Our implementation of WDS is equivalent to LDS rather than ILDS when we use constant weights for the right branch. The fact that WDS with the "best weight" seems to perform similarly to ILDS confirms that the weight function is having a positive impact on the search.



(a)



(b)



(c)

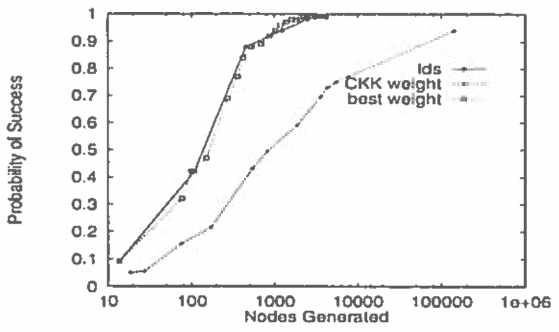
FIGURE 49. Profiles for ILDS, WDS with CKK weights, and WDS with best weights found so far, for $D = 6$ and $r = 1.2$ (a), $r = 1.4$ (b), and $r = 1.82$ (c).

Figure 50 shows the behavior of WDS with the different weights as we vary D. Notice how poorly WDS with the CKK weight does when $D = 4$. When $D = 5$, the performance of WDS with the CKK weight degrades as the number of nodes increases. The profile at $D = 6$ indicates that the CKK weight is getting better as D increases, however, the "best weight" behaves better with respect to changes in the parameters of the problem instances (D and r).

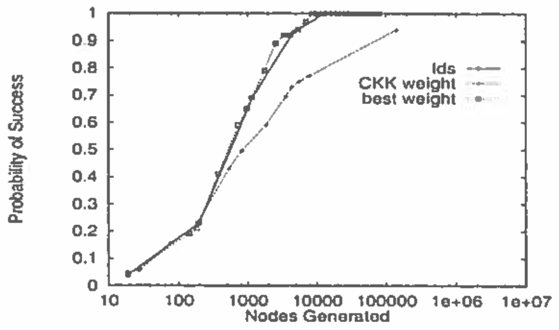
6.5.2 Obtaining Optimal Cutoff Policies

We profiled instances with random numbers drawn from a uniform distribution between zero and ten thousand ($D = 4$), one hundred thousand ($D = 5$), one million ($D = 6$), and ten million ($D = 7$). We chose the following values for r: 1.0, 1.2, 1.4, and 1.82. Figure 51 shows the logarithm of the number of nodes explored in case of failure vs. the logarithm of the cutoff for D between 4 and 7 and a ratio of 1.4.

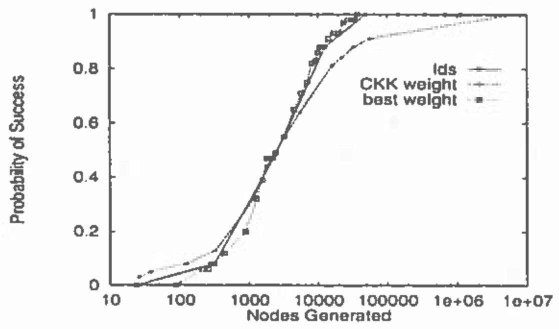
Notice that there is an apparent discontinuity in the log plot of the number of nodes explored at a cutoff value of 0.4. This means that the number of nodes grows very slowly during what look like "plateaus" and then grows fast where we see the "cliffs". The discontinuity of the plot is not really a discontinuity in the number of nodes explored vs. cutoff value, it just shows a discontinuity in the rate of growth. If there was a real discontinuity, our approach would fail, since we assume all functions are continuous and differentiable. We still try to find best fits to these functions and the fits probably ignore the discontinuities. We assume the fits will provide approximations to the optimal cutoff policy. The actual OCP should account for the cliffs and plateaus.



(a)



(b)



(c)

FIGURE 50. Profiles for ILDS, WDS with CKK weights, and WDS with best weights found so far, for $D = 4$ (a), $D = 5$ (b), and $D = 6$ (c), and $r = 1.4$.

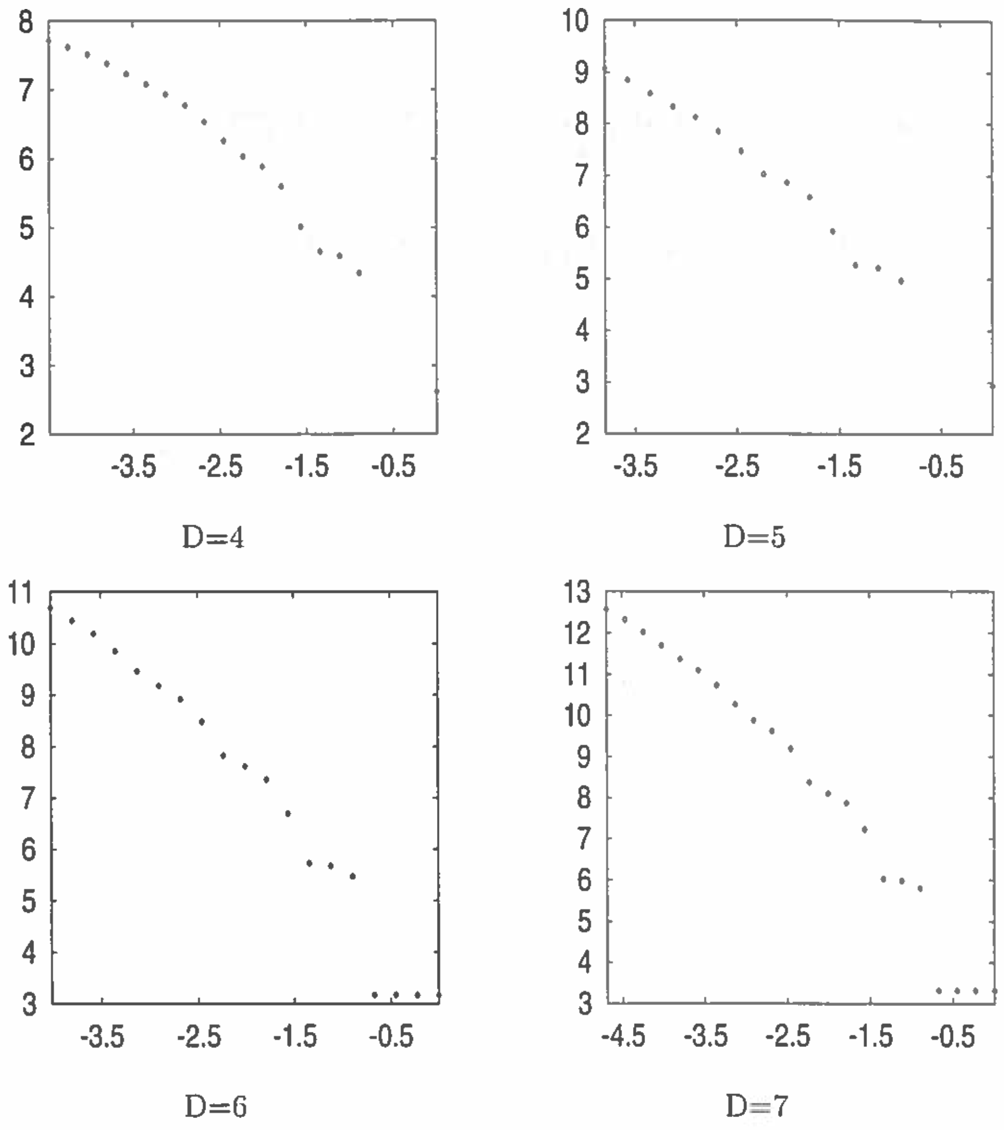


FIGURE 51. The y-axis shows the logarithm of the number of nodes explored in case of failure. The x-axis shows the logarithm of the cutoff value. $R = 1.4$.

One possible explanation for the apparent discontinuity is that the weight distribution is such that there are few nodes with weight higher than 0.4. Two more apparent discontinuities occur at about 0.1 and 0.05. After that, the number of nodes seems to increase exponentially with each successive increase in cutoff. Intuitively, what these profiles are telling us is that the optimal cutoff policy for these problems should not include more than one cutoff value in the plateaus, i.e., between 1 and 0.4, between 0.4 and 0.1, and between 0.1 and 0.05. The cliffs and plateaus also appear in the data as we vary r. Appendix C contains the figures that show the data for $r = 1.0$, $r = 1.2$, and $r = 1.82$.

TABLE 34. Fit functions for number partitioning.

$\mathrm{N}_{\mathrm{succ}}(c)$	$e^{a_s \log(c)^4 + b_s \log(c)^3 - c_s \log(c)^2 + d_s \log(c) + e_s}$
$\mathrm{N}_{\mathrm{fail}}(c)$	$e^{a_f \log(c)^3 + b_f \log(c)^2 - c_f \log(c) + d_f}$
$\mathrm{P}_{\mathrm{fail}}(c)$	$e^{a_p \log(f(c))^3 + b_p \log(f(c))^2 - c_p \log(f(c)) + d_p}$

We find the best fits to the points in Figure 51 and the number of nodes in case of success at candidate cutoffs and the probability of failure in terms of the number of nodes explored. The fit functions and parameters are given in Table 34. These functions are the best fits for all values of D and r. Figure 52 shows the fit for $\mathrm{N}_{\mathrm{fail}}(c)$ when $D = 6$ and $r = 1.4$. Notice that the fit smooths out the function.

We calculated the first derivatives for the functions $\mathrm{N}_{\mathrm{succ}}$, $\mathrm{N}_{\mathrm{fail}}$, and $\mathrm{P}_{\mathrm{fail}}$, and used them along with (5.5) to compute the respective OCPs. Table 35 through Table 38 show the OCPs we found by solving (5.5) using the fit functions we just described. Notice that there is no particular regularity in the way policies change as we vary the number of digits. This suggest that extrapolating the OCPs for

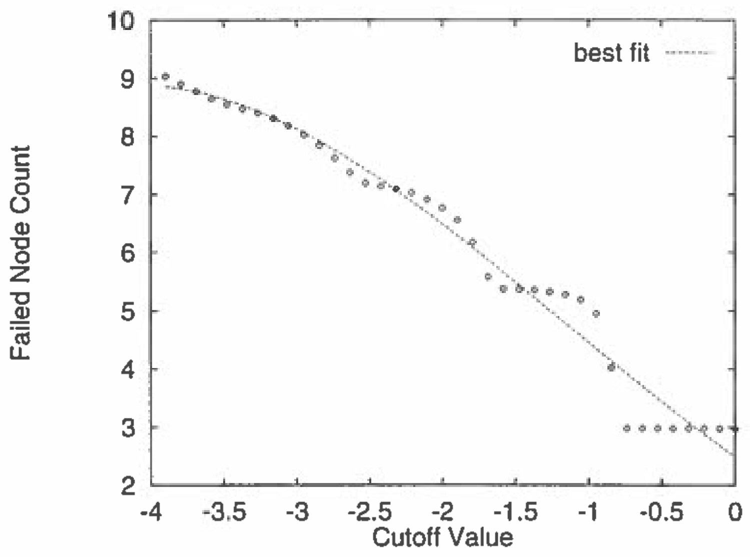


FIGURE 52. Example of a best fit to number partitioning data with $a_f = 0.077$, $b_f = 0.268$, $c_f = -1.769$, and $d_f = 2.497$.

smaller values of D may not be as straightforward as we initially thought. Table 39 shows the policies we obtained after selecting the cutoff values that appear more consistently in the OCPs as D increased.

Take $r = 1.0$ for example. The OCP for $D = 4$ is $\{1, 0.03, 0\}$, for $D = 5$ it is $\{1, 0.04, 0\}$, for $D = 6$ it is $\{1, 0.2, 0.001, 0\}$, and for $D = 7$ it is $\{1, 0.03, 0.003, 0\}$. In these policies we see that there is a cutoff at 0.4 or 0.2, then a cutoff at 0.03 or

TABLE 35. OCPs obtained using the differential equation for $r = 1.0$.

D	N	OCP
4	13	$\{1, 0.03, 0\}$
5	17	$\{1, 0.04, 0\}$
6	20	$\{1, 0.2, 0.001, 0\}$
7	23	$\{1, 0.03, 0.003, 0\}$

TABLE 36. OCPs obtained using the differential equation for $r = 1.2$.

D	N	OCP
4	15	$\{1, 0.1, 0.004, 0\}$
5	20	$\{1, 0.1, 0.02, 0\}$
6	24	$\{1, 0.2, 0.02, 0.009, 0.008, 0.009, 0.006, 0\}$
7	27	$\{1, 0.03, 0\}$

TABLE 37. OCPs obtained using the differential equation for $r = 1.4$.

D	N	OCP
4	18	$\{1, 0.4, 0.07, 0.005, 0.0006, 0.0005, 0.0004, 0\}$
5	23	$\{1, 0.4, 0.08, 0.02, 0.009, 0.006, 0\}$
6	28	$\{1, 0.2, 0.04, 0.01, 0\}$
7	32	$\{1, 0.2, 0\}$

0.04, one last cutoff at 0.003 or 0.001, before ending in 0. One possible "extrapolated" cutoff policy is $\{1, 0.4, 0.03, 0.001, 0\}$.

Because the profiles exhibit discontinuities, and the fits are unable to capture this effect, we decided to obtain cutoff policies that included only one cutoff from each plateau in the data, exemplified by Figure 51. These points represent

TABLE 38. OCPs obtained using the differential equation for $r = 1.82$.

D	N	OCP
4	23	$\{1, 0.07, 0.06, 0.05, 0\}$
5	30	$\{1, 0.8, 0.5, 0.1, 0\}$
6	36	$\{1, 0.9, 0.6, 0.3, 0.05, 0\}$
7	41	$\{1, 0.09, 0.05, 0.04, 0\}$

TABLE 39. CPs extrapolated using the differential equation for $D = 10$.

r	N	CP$_{\text{eqn}}$
1.0	33	$\{1, 0.4, 0.03, 0.001, 0\}$
1.2	39	$\{1, 0.2, 0.02, 0.009, 0.004, 0\}$
1.4	46	$\{1, 0.4, 0.07, 0.02, 0.009, 0.005, 0.0006, 0.0004, 0\}$
1.82	60	$\{1, 0.9, 0.5, 0.003, 0\}$

substantial increases in the number of nodes explored. We chose to include the cutoff values that mark the end of the plateaus, where the plateaus meet their corresponding cliffs.

Let us compare the policies obtained via the differential equation with the policies obtained by recording the discontinuities in the profiles. Table 40 through Table 43 show the cutoff policies CPs, obtained from the profile information.

TABLE 40. CPs obtained from profiles for $r = 1.0$.

D	N	CP
4	13	$\{1, 0.09, 0\}$
5	17	$\{1, 0.2, 0.01, 0\}$
6	20	$\{1, 0.05, 0.002, 0\}$
7	23	$\{1, 0.1, 0.01, 0.003, 0\}$

Observe that these cutoff policies exhibit a certain regularity as D increases. The policies tend to grow with D and the cutoff values at smaller values of D tend to repeat at larger values of D. The extrapolation in this case is much easier. We study how the policies change, and guess what the policy should be at $D = 10$. The "extrapolated" policies appear in Table 44.

Finally, we ran WDS with the "best weights" on 100 instances of random

TABLE 41. CPs obtained from profiles for $r = 1.2$.

D	N	CP
4	15	$\{1, 0.1, 0.03, 0\}$
5	20	$\{1, 0.3, 0.05, 0\}$
6	24	$\{1, 0.1, 0.02, 0\}$
7	27	$\{1, 0.2, 0.03, 0.004, 0\}$

TABLE 42. CPs obtained from profiles for $r = 1.4$.

D	N	CP
4	18	$\{1, 0.3, 0.05, 0\}$
5	23	$\{1, 0.4, 0.1, 0.05, 0\}$
6	28	$\{1, 0.4, 0.2, 0.05, 0\}$
7	32	$\{1, 0.4, 0.2, 0.06, 0.01, 0\}$

problems with $D = 10$ and $r = 1.0$, $r = 1.2$, $r = 1.4$, and $r = 1.82$, using the extrapolated policies we obtain by both methods. We compare the results in Table 45.

WDS using CP_{eqn} represents a savings of 28% at $r = 1.0$, 9% at $r = 1.2$, but ILDS does 0.05% better at $r = 1.4$ and 18% better at $r = 1.8$. WDS using

TABLE 43. CPs obtained from profiles for $r = 1.82$.

D	N	CP
4	23	$\{1, 0.4, 0\}$
5	30	$\{1, 0.4, 0.2, 0\}$
6	36	$\{1, 0.4, 0.2, 0.07, 0\}$
7	41	$\{1, 0.4, 0.2, 0.08, 0\}$

TABLE 44. CPs extrapolated from profiles for $D = 10$.

r	N	CP_{prof}
1.0	33	$\{1, 0.2, 0.03, 0.004, 0.0005, 0\}$
1.2	39	$\{1, 0.4, 0.2, 0.06, 0.01, 0.006, 0\}$
1.4	46	$\{1, 0.4, 0.2, 0.08, 0.02, 0\}$
1.82	60	$\{1, 0.1, 0.01, 0.003, 0.0005, 0\}$

TABLE 45. Performance of WDS using extrapolated CPs for $D = 10$ vs. ILDS and DFS.

r	N	CP_{eqn}	CP_{prof}	ILDS	DFS
1.0	33	84,888,636	37,996,272	118,720,047	33,656,653
1.2	39	91,453,286	79,977,987	100,433,955	108,351,708
1.4	46	11,392,182	11,916,800	10,838,692	24,260,347
1.82	60	1,984,352	1,848,350	1,622,172	9,092,317

CP_{prof} represents savings of 68% at $r = 1.0$, 20% at $r = 1.2$, but ILDS does 0.09% better at $r = 1.4$ and 18% better at $r = 1.8$. The savings obtained by WDS in this case more than offset the cost paid when it does worse than ILDS. Observe that DFS at $r = 1.0$ represents a savings of 72% over ILDS, but of just 11% over WDS using CP_{prof} and of 60% over WDS using CP_{eqn}. Overall, it looks like WDS using CP_{prof} performs well in the hardest and the easiest problems, whereas ILDS is more sensitive. In general, one may not know where the hardest problems lie, so WDS may be a better alternative than ILDS.

6.6 Conclusions

Applying our methodology to number partitioning was not as straightforward as we had expected. We had problems finding a weight function that had a chance of improving the constant weight function (weight function of LDS and ILDS), and more research is needed to determine whether there are weight functions that characterize where KK is making mistakes. The KK heuristic seems to capture most, if not all, the information we have on number partitioning. The signal of the right branches, then, is not particularly informative. We learned that in this case, ILDS may not be an unreasonable choice of algorithm. Other heuristics, such as those studied by Ruml et al. [49] and Gent and Walsh [15] may be more informative.

We also found that there were apparent discontinuities in the functions for the number of nodes explored in terms of the cutoff value. These jumps in value suggest that there are periods of stagnation where the cutoff decreases but where the number of nodes does not increase substantially. Our approach works under the assumptions the fits are continuous. The policies we obtain using continuous (smooth) fits are not as accurate as we believe they could be. Furthermore, there are other ways to form "extrapolated" cutoff policies from the given data. Another way to obtain cutoff policies for the larger problem is to extrapolate the parameters of the fit functions instead of the policies themselves. Since we did not consider that our fit functions were reliable, we did not even consider this possibility. The question of whether there are problems that exhibit enough "regularity" in the parameters of the fits to allow for meaningful extrapolations is still open. One could use standard numerical procedures for doing extrapolations [47]. Further research is needed to understand the reason behind the cliffs and plateaus and to devise better methods to extrapolate cutoff policies.

The policies we obtained by looking at the discontinuities indicate that better policies exist. WDS using these policies is less sensitive to transitions between hard problem instances and easy problem instances. As expected, WDS performs as well as ILDS in the satisfiable region and it performs as well as DFS in the unsatisfiable region.

One issue we have neglected to mention is the issue of branching factor. In the traditional number partitioning problem, the objective is to partitioning the set in two, but it is also possible to generalize this problem to partitioning the set in multiple mutually exclusive and collectively exhaustive subsets. Korf [35] briefly discusses how to generalize KK and CKK for multi-way partitioning. Since the branching factor increases in multi-way partitioning, we might expect that WDS to outperform ILDS, but this may only be possible if we are able to find informative weights.

CHAPTER VII

RELATED WORK

7.1 Discrepancy-Based Search Methods

Other extensions to LDS have been made in the literature. Depth-bounded discrepancy search DDS [61] uses a depth bound to restrict the search to discrepancies at depths smaller than the bound. For instance, at a depth bound of 2 and discrepancy count 2, only leaves on paths with at most 2 discrepancies that occur at depths 1 and 2 are explored. In DDS the bound and the discrepancy count are increased between iterations. DDS was developed to force LDS to explore paths with discrepancies at the top before paths with discrepancies at the bottom. WDS addresses this issue naturally by handling weights that depend on the signal strength instead of discrepancy counts. Since WDS adjusts automatically, we expect it to be a better alternative than DDS.

Interleaved depth-first search IDFS [41, 42] does a parallel search on a subset of subtrees called active subtrees, and when finished it does DFS on what is left. IDFS partitions the search space into parallel and sequential levels. Like LDS, IDFS treats all discrepancies alike. Meseguer provides no means to choose the active subtrees or to determine their order, other than hint that these decisions should be biased towards the heuristic and be such that the top of the tree is explored early. Good performance of IDFS depends on correctly selecting the depth of the parallel levels, their distribution (consecutive or not), and the number of levels. Meseguer provides no guidance on how to set any of these parameters.

The parallel search in IDFS aims at spreading the search effort throughout different portions of the search tree, while the sequential search focuses on a particular region. WDS distributes its search effort naturally depending on the cutoff policy and the weight distribution. If nodes with high weights are scattered, WDS will spread out its search effort throughout different portions of the tree. If nodes with high weights are clustered, WDS will focus on that cluster. As in IDFS, in WDS nodes are placed in separate classes. Each class in WDS is determined by a cutoff value. Nodes with weight greater than or equal to a particular cutoff value, belong to the class associated with that value. The number (length of the policy) and size of each class (cutoff values) is determined analytically in order to maximize utility. We believe WDS naturally achieves what Meseguer and Walsh [42] set out to do.

Improved limited discrepancy search [36] presents a small savings by using the depth to the fringe of the tree to determine that certain paths need not be regenerated. The weight function, in WDS, may not have a direct dependence on the distance to the fringe, so it may not be possible to integrate this modification in the new setting.

Other search methods that assign weights to nodes or branches are also related to WDS. Most successful algorithms that use weights to restrict the search are variations of best-first search, such as iterative deepening A* and recursive best-first search.

7.2 Best-First Search Methods

A* is a best-first search algorithm that explores nodes in strict order of decreasing cost [23]. A* is guaranteed to find the optimal cost node if the heuristic is admissible. An admissible heuristic never overestimates the true value of a choice. Since at any point in time A* contains all candidate paths in memory, it may potentially use an exponential amount of memory. A more space-efficient implementation is iterative deepening A* [33], IDA*. IDA* saves space by iterating with increasing cutoffs and exploring nodes depth-first within each iteration. The cutoff is set to the smallest cost that did not make the previous cutoff. IDA* is complete and optimal with the same caveats as A*.

WDS relates to IDA* in the sense that they are both particular cases of best-first search where nodes in each iteration are explored in depth-first order. In WDS a node with lower weight may be explored before a node with higher weight. If a cutoff policy for WDS jumps from 0.58 to 0.35, then it is possible that WDS will expand a node with weight 0.4 before it expands one with weight 0.5, since both weights are between 0.58 and 0.35. IDA* keeps track of the minimum weight that exceeded the current cutoff and increases the cutoff by this amount before the next iteration. However, if values are drawn from a continuous space, each iteration of IDA* would explore a single new leaf node. Sarkar et al. [53] were the first to propose a fix for this problem. They observed that in cases where IDA* worked well, it expanded exponentially many more nodes in each new iteration. They proposed a method to adjust successive cutoffs that groups the set of values which exceed the current cutoff into buckets. The buckets have counts associated with them and the next cutoff is selected so as to be as close as possible to a geometric progression. They showed IDA*_CR outperforms IDA* when real valued estimates are required. Their results confirm the intuition that a good candidate cutoff instructs WDS to search exponentially many more nodes than previous cutoffs. The difference between IDA*_CR and our approach is that we do not need to require that cutoffs follow a geometric progression. It is a natural consequence of the tradeoff between the number of nodes explored and the probability of success.

Recursive best-first search [34] is a modification of best-first search that requires space $O(bd)$. RBFS is a recursive algorithm that expands nodes with value smaller than the current threshold in depth-first order. A new threshold value is set every time the algorithm recurses. Korf [34] has shown that RBFS dominates IDA* in all cases, but sometimes the savings don't outweigh the additional complexity in the implementation.

RBFS can simulate LDS if the discrepancy count of a node is considered as the cost of the node. In RBFS, each iteration starts with the last path of the previous iteration, so it does not need to generate the first path of each iteration. The savings is only significant when a solution is found very early in the second iteration.

7.3 Systematic vs. Nonsystematic Search Methods

A systematic search method is complete and it is not redundant [45]. That is, it considers all alternative paths and it never considers any one of them more than once. A systematic method often works by constructing a new solution from the old one by keeping track of the alternatives it has explored. In contrast, nonsystematic search methods start out with a complete candidate solution and make iterative repairs hoping to improve the candidate solution. Nonsystematic methods have more freedom of movement since they are allowed to modify the candidate solution anywhere they want. Nonsystematic methods sacrifice the guarantees of completeness and non-redundancy in hopes of exploring better alternatives more quickly.

LDS and WDS are redundant because work in early iterations is repeated in later iterations, but they are also complete since they are guaranteed to search the whole space in the last iteration. They both work their way through the search space by constructing a solution one step at a time. One advantage of constructive methods is that they can easily be combined with propagation techniques [28]. These techniques work by extending the current assignment, instantiating unassigned variables with assignments that are logically entailed by the current assignment. A propagation procedure finds assignments that would also be found by search, but it does so without exploring the alternatives. These techniques identify regions of the search space that do not need to be searched. It is not clear how propagation techniques can be combined with nonsystematic search in order to rule out some set of repairs.

7.4 Doing The Right Thing

It is possible to cast our architecture in terms of agents interacting and reasoning in terms of their environment. Agents first sense their environment, interpreting and transforming their perceptions into information useful for reasoning. Agents reason to decide on a course of action and, finally, execute the action. Rational, intelligent agents decide on the action that maximizes their expected utility.

Sensing in our framework corresponds to the reception of the input parameters: depth and breadth of the tree, and heuristics. Interpreting and transforming the input parameters into useful information is achieved by the weighting scheme. The random instances are generated according to the input parameters. Reasoning involves sampling the space of candidate cutoff values, building the performance profiles, and recording the number of nodes explored and the probability of failure. It also involves using this information to obtain the optimal search protocol. Acting involves running the optimal search protocol on the problem or problems at hand. Notice that the performance profiles provide information on the expected quality of the performance of the agent. This allows the agent to make the best possible informed decision. During the reasoning process, the agent considers all possible lines of action, a consequence of using calculus of variations. The agent ends up choosing the line of action that maximizes its utility.

Our approach is not the first one to address the problem of "doing the right thing". Russell and Wefald [51] propose architectures for maximizing the utility of the search according to the value of expanding one node at a time. This approach spends too much computation time at a node level and it is prone to thrashing. Imagine two completely different competing lines of decisions with the root node as the only common ancestor. Russell and Wefald's algorithm will oscillate between both lines, redoing all the work each time the utility of the line it is on is worse than that of the competing line. If this happens at each new node, the algorithm will spend all of its allowed run time switching between the two lines of decisions. Fixes to this problem have been devised. One fix involves relaxing the problem by considering all nodes within some bound ϵ as equal, and searching the nodes within the bound at the same time in some convenient order (DFS for instance). Our approach goes much further, by recognizing that only big changes in probability of success matter, grouping nodes in larger classes.

Russell and Wefald's algorithms attempt to balance the tradeoff between deliberating and acting. In addition to an evaluation function, they assume a variance function, which gives an estimate of how much the true value of a given decision is likely to vary from its static value (value before deliberation). At each step, their algorithm compares the value and variance of the best choice computed so far and the second best choice. If the best choice is clearly better than the second best (taking variance into account), then there is no point in computing further and a commitment to the best choice should be made. Also, if the two top choices have similar values but both have very low variance, then computing will not help much and they suggest choosing one of the two at random.

If the tradeoff between deliberation and action can be quantified into a performance profile, and if it is possible to find a general expression for this tradeoff in terms of an arbitrary cutoff (or allocation) policy, it may be possible to find the optimal allocation policy for deliberation.

7.4.1 Anytime Algorithms

There are other approaches that use performance profiles in order to compute the best allocation of resources. Anytime algorithms [5] are algorithms whose quality of results improves gradually as computation time increases, offering a tradeoff between resource consumption and output quality. This tradeoff is characterized in a performance profile, which shows how the solution quality improves with time.

In anytime algorithms, an imprecise answer is generated quickly and refined through a number of iterations. Anytime algorithms can either statistically achieve a required quality by a given deadline or produce some answer at any time, which may or may not be useful. In either case, the quality of the answer improves with time.

Anytime algorithms can be turned into optimal deliberation algorithms by intersecting performance profiles and assigning time to the segment with the steepest slope (representing the most gain). The different performance profiles arise from competing choices. Each competing choice exhibits a different performance profile according to how the algorithm would perform if it worked on that condition.[1]

In this work, we do not have a small and finite set of conditions. Each possible cutoff value is a condition and we have infinitely many of them. The performance profiles we build in order to compute the optimal cutoff policy approximate the performance profile that shows the best WDS can do. These profiles are built from a sample of cutoff values. The performance profile that represents the most gain is determined analytically. The solution to the differential equation gives us the optimal cutoff (or allocation) policy.

Contract anytime algorithms are algorithms that return increased value as they are given more time. However, contract algorithms must be told in advance how much time they are going to get and may not return any answer if they are given less time than initially allotted. WDS provides these guarantees in the particular case where a cost function is associated with node expansions. In the case we studied in the previous chapter, where there is a single node bound B, the OCP guarantees that a goal will be found within the time allotted with probability

[1]This description is accurate for the static case, where no new observations are available after the search starts. Boddy and Dean [5] conclude that static case performs close enough to the global (dynamic) case that this analysis is good enough for either case in practice.

$P_{\text{succ}}(B)$ and with probability $1 - P_{\text{succ}}(B)$, the goal will be found later. Moreover, if given more time than what is required to explore B nodes, the probability of success will increase in time. These guarantees are analogous to the kinds of guarantees that contract algorithms provide.

7.4.2 Game-Tree Search

Interesting games, like chess, have huge search trees. Chess has a branching factor of 35 and with a total space of about 35^{100} nodes, considering 50 moves for each player. The game search community has spent considerable effort in finding techniques to help focus the search and reduce the "effective" branching factor. Two commonly used techniques are narrowness [17] and conspiracy numbers [40].

Narrowness first focuses the search where opponents have fewer responses. This concept is believed to be related to conspiracy numbers. Conspiracy numbers are computed by counting the number of nodes below a node whose evaluation function would have to change in order to change the node's value. Smaller conspiracy numbers are considered more likely to change the value of the best position, and thus are searched first.

Narrowness and conspiracy numbers are loosely related to WDS. In WDS search focuses where the signal is strongest. We believe a strong discrepancy is usually an indication that either the discrepancy is as good as the heuristic choice, or the heuristic is unable to make a clear choice and it is therefore likely to make a mistake. If the heuristic choice is a mistake, the choice transitions from potentially having a high "value" before the search to having a low "value" after the search. Thus, WDS, like algorithms that use narrowness or conspiracy numbers, focuses on portions of the tree which are more likely to change the value of the best choice.

7.4.3 Mathematical Methods

Other mathematical approaches to optimal search exist [54, 57]. Stone's approach [57] is similar to ours in the sense that he also groups nodes into regions he calls "cells". Stone also uses mathematics (more specifically, Lagrangian operators) to derive the optimal search protocol. His approach differs from ours in that he assumes that there is a non-zero probability that the goal may not be found given that it is in the cell that is being searched. This may happen, for example, if a nonsystematic method is used to search within cells.

Stone's approach only focuses on finding the best allocation of effort on the set of given cells to maximize the probability of success. In contrast, our approach focuses on determining the "cells" (with the condition that each cell has to be a superset of the previous one) and assumes that search within each cell is deterministic (DFS). Stone's approach is more flexible in that it allows for arbitrary jumping between regions of the search space, but our approach is guaranteed to find a solution if there is one. Also, Stone does not provide hints as to how cells should be chosen, where the probabilities come from, or how to obtain constraints on the effort of searching each cell. Our approach addresses all of these questions.

Simon and Kadane [54] attempt to quantify optimal search as that which maximizes the expected return. As in WDS, the expected return is calculated based on estimates of probability of success and number of nodes explored. Simon and Kadane use the ratio of probability of success versus nodes explored to determine whether search should change course. This calculation is done during the search.

They define "best sets" of nodes and prove that search should continue in the current best set as long as the ratio of expected return continues to increase and in fact until it becomes lower than the best ratio for some other branch.[2] They do not provide, however, methods to estimate the probability of success or the size of the best sets. Their algorithm does a best-first search on the possible best sets. Like best-first search, their approach requires exponential memory. Construction of the best sets is done top-to-bottom, unlike our approach where the "best set" is determined by heuristic strength and the optimal cutoff policy.

[2]It is unclear from their article how the "best sets" are determined, we believe the best sets are subtrees.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

The work in this dissertation began in an effort to quantitatively address the problem of how to achieve rational behavior while searching for a solution to a problem. We assumed the search space to be structured as a tree and that solutions can only be found at leaf nodes. Any solution node is as good as any other, so finding any solution is enough to succeed. We have also assumed that the trees have uniform depth and are complete. A good heuristic, which is positively correlated with the likelihood that a node leads to a goal, is given as input. Also given are the average depth and breadth of the tree, and a cost function associated with node expansions.

We developed weighted discrepancy search based on the observations that the optimal search protocol for this problem should (1) eliminate candidate leaf nodes quickly, (2) avoid thrashing while searching through best candidates first, and (3) account for heuristic mistakes. We also devised a principled methodology to obtain optimal cutoff policies that maximize expected utility.

Using a characterization of the search space in terms of the heuristic, average depth and breadth of the search tree, we showed how to obtain the optimal search protocol OCP. The methodology unfolds as follows:

1. formulate the expression for the expected cost to solution in terms of the number of nodes explored and the probability of success at arbitrary cutoff values;

2. find analytical (or numerical) approximations for the number of nodes explored and the probability of success at candidate cutoff values;

3. using the calculus of variations to minimize the expression for the expected cost to solution, find the differential equation that the optimal cutoff policy has to satisfy;

4. solve the differential equation using the estimates found in 2, and obtain the OCP;

5. run WDS, using the OCP, on the problem at hand.

Weights in WDS have the property that they can correlate with the actual value of the choice more strongly than discrepancies do. Because WDS may simulate other backtracking algorithms, WDS using the OCP performs at least as well as the best of these algorithms in any given situation. Furthermore, we showed in Section 5.5.2 that WDS using the OCP outperforms 1-SAMP, DFS, and LDS.

Other contributions have been made in this work. We have adopted and extended Harvey's characterization of the search space in terms of p and m. We have shown how to construct a random instance generator that simulates the behavior of the heuristic. We have also shown that it is possible to estimate the number of nodes explored and the probability of success by running Monte Carlo simulations using this random instance generator. Furthermore, we showed in Section 4.1 that the results obtained by doing the simulations are good approximations to those obtained analytically. We have discovered, in passing, that good policies are such that the probability of success increases substantially between cutoff values. This result allowed us to discover that the performance of limited discrepancy search can be improved by skipping a few iterations.

Although the bulk of this work is concerned with uniform cost functions, we showed in Section 5.6.2 that our methodology is also valid when a non-uniform cost function associated with node expansions is given as input. In fact, WDS using the optimal cutoff policy for a non-uniform cost function has a higher probability of success than WDS using the uniform cost optimal cutoff policy.

Further experimentation should confirm the utility of our approach. We believe it would be interesting to compare our results in the case of non-uniform cost functions with results obtained by heuristic-biased stochastic sampling [7]. Heuristic-biased stochastic sampling HBSS, is like ISAMP except that instead of randomly probing the search space, probes are biased towards the heuristic. Like ISAMP, HBSS is also not guaranteed to search the whole space and may also search the same node twice. Search in HBSS stops after B nodes have been explored, where B is given as input.

Good performance of HBSS depends on the bias function used. A fair comparison between HBSS and WDS using the optimal cutoff policy would involve running HBSS with a bound of B nodes and running WDS using the optimal policy for the corresponding non-uniform cost function, as described in Section 5.6.2. While WDS is designed to maximize the probability of success within the first B nodes explored, HBSS probes randomly according to the given bias function. Comparing the two approaches on different problems should illustrate the advantages of one over the other.

Another topic of research that has not been addressed but is very relevant to the work in this dissertation is the development of other cost functions that arise in practice. An example of a useful cost function is a function that accounts for the passage of time in reasoning. The only approach we are aware of that tackles this problem directly is called step logic [11]. Step logics account for the passage of time in reasoning by associating a unit cost with each reasoning step, so that the longer the reasoning, the greater the cost. But this characterization is not accurate if the set of conclusions obtained by the longer derivation is more useful than those that were obtained faster. The utility of the conclusions matters as much as the time it takes to derive them. Our approach has the advantage that it allows us to treat the "time value of inferences" in the same way we would treat any other cost associated with inferences.

Other questions regarding cost functions remain unanswered. For instance, what kinds of cost functions characterize the cost of reasoning involved in determining the value of a choice? What kinds of cost functions characterize the utility of reasoning given the passage of time? What is the cost associated with executing the course of action that has been chosen? Is it possible to quantify the trade-off between deliberation and action? These are only a few of the many possible directions for future research. Other possible extensions are outlined in the following sections.

8.1 Weight Functions

In this section we mention possible changes to the weight function that would have to be made if the heuristic encoded information on dependencies between siblings and on lookahead.

8.1.1 When Signal Strength of Siblings is not Independently Assigned

In order to calculate the weights, we assumed that the signal strength of siblings was independently assigned. This allowed us to conclude that the probability that both the left node and the right node are good is the product of the probabilities that each one individually is good. That is, we assumed that $p_{lr} = p_l p_r$ where $p_l = \frac{w_l}{w_l+w_r}$ and $p_r = \frac{w_r}{w_l+w_r}$. However, if success on the right node depends on success on the left node, the signal strength, which quantifies success of a node, is not independently assigned. In this case, the product must be replaced by a conditional probability analysis. For instance, if the right node's success depends on a function f of the probability of the left node's success, then $p_{lr} = \mathrm{pr}(p_r \mid f(p_l))$. Some studies on game trees, however, show that the dependence between siblings is too small to matter [13]. It would be interesting to do studies in other domains.

8.1.2 When Heuristics Use Lookahead

Heuristics that explore the subtree of height l and use these values to estimate the potential value of the choice instead of assigning the "face-value" of the choice, are said to "look ahead" l levels. If the heuristic uses lookahead in order to evaluate the best choice, then the value of the choice depends on the value of the choices in its subtree. In other words, the values of the subtree are backed up to determine the value of the choice. The current weighting scheme provides no mechanism to consider backed up values. A different way to calculate weights is needed in order to ensure that the information is used in some meaningful way.

Most of the research in weighting schemes with lookahead comes from game-playing [13, 17, 45]. Node evaluation functions without lookahead for two player games measure how favorable one position is to one player relative to the other. The position is evaluated with respect to both players and one measure is subtracted from the other to yield the value of the position. A large positive value corresponds to a strong position for one player (the maximizer), while a large negative value corresponds to a strong position for the other (the minimizer). Values for internal nodes are computed by backing up the values at the leaves, always taking the maximum for the maximizer and the minimum for the minimizer in order to determine the best move.

In games of imperfect information, where one is unsure about (say) the cards held by the opponents, the value of choices is usually determined by a probabilistic analysis [13, 45]. Alternatively, we may try to predict the outcome of the game by Monte Carlo sampling on the space of possible moves [17]. We have argued that weights are reasonable approximations to the real probability of success of a node and we have characterized the weight distribution function. However, we have done so without taking into consideration weights in subtrees underneath the nodes.

Baum and Smith [3] propose a way to combine probabilistic information about child nodes to derive consistent probabilistic information about the parent nodes. One way to do this is to collect information on the distribution of the value of the choices underneath a node given possible assignments. Baum and Smith assign the value of a particular move by summing over the assignments in which that move is the best one. It might be interesting to explore the possibility of designing weighting schemes for WDS that use lookahead information.

8.2 Learning Optimal Policies

Lookahead is one way of learning the "true values" of choices. However, if the subtrees of height l do not include leaf nodes, lookahead will only improve the estimate without revealing the true value. One way to learn the "true values" of choices is to record the values at the leaves during the search. Learning the true value of choices may help determine the optimal cutoff policy for the particular instance that is being searched.

If the "real value" and the heuristic value for the choices differ, it may be possible to adjust the cutoff policy accordingly. The new values could be used to generate new instances and find a new optimal cutoff policy. In practice, this may prove to be too expensive, especially if the differences between the true values and the heuristic values are small. Alternatively, the new values could be used to compare the "real" weight distribution with the heuristic weight distribution, and use the analytic results to obtain a new cutoff policy. Further research is needed to determine whether either of these are viable approaches and the conditions under which either of these would improve on the results obtained without learning.

An alternative approach to integrating learning in our framework is to find a new formulation of the expected cost to solution in terms of the utility of expanding one new node at a time, instead of a new iteration at a time. At some level, this is what "traditional" utility-based approaches do [51]. At this time, however, we do not see any way to obtain such a formulation without being exposed to the same problems of traditional approaches, we discussed in Section 7.4.

8.3 SAT and UNSAT Instances

Throughout this work we assumed the problem instances were satisfiable, that is, problems known to have at least one solution. However, it is not always known whether a particular set of instances is satisfiable (SAT), or not (UNSAT). Is it possible to determine the optimal cutoff policy for WDS when a mixed set of instances is given? In a mixed set of instances, some instances are SAT and others UNSAT, but at any given time, we do not know whether the particular instance we are searching is SAT or UNSAT. Furthermore, the cost of the UNSAT instances dominates the cost of the SAT instances, since in the UNSAT instances, WDS searches the whole search space before realizing that the instance in UNSAT. Thus, we would expect the optimal cutoff policy to instruct WDS to behave as DFS sooner if there are many UNSAT instances than if there are only a few.

Intuitively, if 90% of the instances are UNSAT, it should be the case that the OCP is biased towards the policy corresponding to DFS. Conversely, given 100 instances, 90 of them SAT, the "actual cost" of the 100 instances is the weighted average of the cost in the case where it is SAT and the cost in the case where it is UNSAT. That is, the actual cost is 90 times the expected cost in SAT instances plus 10 times the expected cost of failing to find a solution. The optimal cutoff policy for this case is found by minimizing the expression for this actual cost, much as we would in the case where all of the 100 instances are SAT.

8.4 Non-Uniform Depth Trees

In practice, trees are not necessarily complete or of uniform depth. Assume, for instance, that after making $k < d$ decisions, all constraints are violated. At this point, instead of valuing the remaining $d-k$ variables, WDS should backtrack. Effectively, a leaf would be reached at depth k. The equations estimating the probability of success and the number of node expansions at candidate cutoffs assume the tree has uniform depth and is complete. The simulations, however, can easily be extended to the case of non-uniform-depth trees.

It is possible to estimate the branching factor as a function of depth by Monte Carlo sampling. Thus, it is possible to estimate the mistake and heuristic probabilities from the weights at that depth. If $b = 1$ the child inherits the same weight of the parent. We showed how to determine p and m from the weights for $b = 2$ in Section 5.3, and for $b = 3$ in Section 5.6.1. The extension to larger values for b is straightforward. Once the mistake and heuristic probabilities have been determined, the tree can be generated as usual and the rest of the methodology can be applied to obtain optimal cutoff policies.

8.5 Large Search Spaces

There are several ways in which our framework can be extended to provide results for large search spaces. First, if the assumptions scale, the results could scale as well. In Section 3.3, we justified our assumptions in terms of job-shop scheduling problems. We found that assumptions that hold for small subsets of a problem instance hold for the large instance as well. In particular, we found that heuristic behavior, depth, and branching factors of the subset spaces were characteristic of the larger space. We have not verified, however, that an optimal cutoff policy obtained for a small subset can be extended to an optimal cutoff policy for the large problem instance.

Estimating the functions N_{succ}, N_{fail}, and P_{fail} is the most expensive step in our methodology. Numerical approximations obtained via simulations may be too expensive for large search spaces. Analytical approximations are cheaper, but they depend on knowing the weight distribution function. In Section 3.3, we showed that the weight distribution is lognormal and that the parameters of the distribution vary with depth. However, we have not used the weight distribution function to obtain analytic estimates for WDS. This is one direction that should be explored. In Section 5.1.1.1, we used the analytical approximations to derive results for LDS up to depth 100, so we believe that these approximations would provide a viable approach to obtain results for WDS, as well.

Another way to avoid the simulations is to study the best fit functions for N_{succ}, N_{fail}, and P_{fail}. Studying these functions may reveal whether the parameters of the fits scale predictably with depth and branching factor. We could then, predict the shape of the performance profile and obtain the OCP directly from that prediction. Still, we would have to verify that the predicted performance profile reflects the properties of the large search problem.

This leads to three questions of theoretical interest: Is is possible to characterize the class of performance profiles that yield optimal cutoff policies?[1] Is it possible to characterize the class of trees where WDS behaves according to these profiles? Are there any problems of interest with search trees in this class?

This dissertation lays the groundwork for tackling these and related questions. We hope that others will find the questions interesting, and the tools we have presented useful.

[1]Andrew Parkes, personal communication.

APPENDIX A

JOB-SHOP SCHEDULING PROBLEMS

The following job-shop problem instances can be obtained from the OR Library maintained by Beasley at http://mscmga.ms.ic.ac.uk/info.html.

The scheduling code we used to study the problems is the same code used by Crawford [10] to solve resource constrained project scheduling problems RCPS. The difference between job-shop JS, and RCPS is that in JS each task can only use one machine at a time, whereas if the resources in RCPS are machines, each task can use more than one resource. Also, in JS, precedence constraints exist only between tasks of the same job, but not between jobs. In RCPS, more complex precedence constraints are allowed.

The format in which the problems are specified is different in JS than it is in RCPS, because the ontology of the problems is different. In the job-shop format, each instance consists of a line of description, a line containing the number of jobs and the number of machines, and then one line for each job, listing the machine number and processing time for each step of the job. The machines are numbered starting with 0.

The following is a sample description for mt06 (also called ft06).

```
++++++++++++++++++++++++++++

instance ft06

++++++++++++++++++++++++++++
Fisher and Thompson 6x6 instance, alternate name (mt06)
6 6
2  1  0  3  1  6  3  7  5  3  4  6
1  8  2  5  4 10  5 10  0 10  3  4
2  5  3  4  5  8  0  9  1  1  4  7
1  5  0  5  2  5  3  3  4  8  5  9
2  9  1  3  4  5  5  4  0  3  3  1
1  3  3  3  5  9  0 10  4  4  2  1
++++++++++++++++++++++++++++
```

In contrast, in the format we used, tasks from different jobs are not distinguished other than by precedence constraints. The data contains several sections.

Section 1 has the following format:

```
Task ID        Proc. Time  0 1 2 3 4 5
```

The numbers 0 through 5 correspond to machine numbers. Each task ID begins with the string `asm_1.step_`.

Section 2 describes the precedence constraints. Section 3 is not needed for job-shop scheduling problems and Section 4 describes the resources and their capacities. In job-shop problems, the each resource (machine) has capacity 1.

The following is the description in the format we used for mt06.

```
section1
asm_1.step_001 00:01 0 0 1 0 0 0
asm_1.step_002 00:03 1 0 0 0 0 0
asm_1.step_003 00:06 0 1 0 0 0 0
asm_1.step_004 00:07 0 0 0 1 0 0
asm_1.step_005 00:03 0 0 0 0 0 1
asm_1.step_006 00:06 0 0 0 0 1 0
asm_1.step_007 00:08 0 1 0 0 0 0
asm_1.step_008 00:05 0 0 1 0 0 0
asm_1.step_009 00:10 0 0 0 0 1 0
asm_1.step_010 00:10 0 0 0 0 0 1
asm_1.step_011 00:10 1 0 0 0 0 0
asm_1.step_012 00:04 0 0 0 1 0 0
asm_1.step_013 00:05 0 0 1 0 0 0
asm_1.step_014 00:04 0 0 0 1 0 0
asm_1.step_015 00:08 0 0 0 0 0 1
asm_1.step_016 00:09 1 0 0 0 0 0
asm_1.step_017 00:01 0 1 0 0 0 0
asm_1.step_018 00:07 0 0 0 0 1 0
asm_1.step_019 00:05 0 1 0 0 0 0
asm_1.step_020 00:05 1 0 0 0 0 0
asm_1.step_021 00:05 0 0 1 0 0 0
asm_1.step_022 00:03 0 0 0 1 0 0
asm_1.step_023 00:08 0 0 0 0 1 0
asm_1.step_024 00:09 0 0 0 0 0 1
asm_1.step_025 00:09 0 0 1 0 0 0
asm_1.step_026 00:03 0 1 0 0 0 0
asm_1.step_027 00:05 0 0 0 0 1 0
asm_1.step_028 00:04 0 0 0 0 0 1
asm_1.step_029 00:03 1 0 0 0 0 0
asm_1.step_030 00:01 0 0 0 1 0 0
```

```
asm_1.step_031 00:03 0 1 0 0 0 0
asm_1.step_032 00:03 0 0 0 1 0 0
asm_1.step_033 00:09 0 0 0 0 0 1
asm_1.step_034 00:10 1 0 0 0 0 0
asm_1.step_035 00:04 0 0 0 0 1 0
asm_1.step_036 00:01 0 0 1 0 0 0

section2
asm_1.step_001 asm_1.step_002
asm_1.step_002 asm_1.step_003
asm_1.step_003 asm_1.step_004
asm_1.step_004 asm_1.step_005
asm_1.step_005 asm_1.step_006
asm_1.step_007 asm_1.step_008
asm_1.step_008 asm_1.step_009
asm_1.step_009 asm_1.step_010
asm_1.step_010 asm_1.step_011
asm_1.step_011 asm_1.step_012
asm_1.step_013 asm_1.step_014
asm_1.step_014 asm_1.step_015
asm_1.step_015 asm_1.step_016
asm_1.step_016 asm_1.step_017
asm_1.step_017 asm_1.step_018
asm_1.step_019 asm_1.step_020
asm_1.step_020 asm_1.step_021
asm_1.step_021 asm_1.step_022
asm_1.step_022 asm_1.step_023
asm_1.step_023 asm_1.step_024
asm_1.step_025 asm_1.step_026
asm_1.step_026 asm_1.step_027
asm_1.step_027 asm_1.step_028
asm_1.step_028 asm_1.step_029
asm_1.step_029 asm_1.step_030
asm_1.step_031 asm_1.step_032
asm_1.step_032 asm_1.step_033
asm_1.step_033 asm_1.step_034
asm_1.step_034 asm_1.step_035
asm_1.step_035 asm_1.step_036

section3
```

```
section4

Assembly Zone Maximum Occupancy

Zone.Z0 1
Zone.Z1 1
Zone.Z2 1
Zone.Z3 1
Zone.Z4 1
Zone.Z5 1
```

The subset problems are the following:

Problem 1: We took a subset of 4 jobs out of the 6 jobs in the original problem.

```
section1
asm_1.step_001 00:01 0 0 1 0 0 0
asm_1.step_002 00:03 1 0 0 0 0 0
asm_1.step_003 00:06 0 1 0 0 0 0
asm_1.step_004 00:07 0 0 0 1 0 0
asm_1.step_005 00:03 0 0 0 0 0 1
asm_1.step_006 00:06 0 0 0 0 1 0
asm_1.step_007 00:08 0 1 0 0 0 0
asm_1.step_008 00:05 0 0 1 0 0 0
asm_1.step_009 00:10 0 0 0 0 1 0
asm_1.step_010 00:10 0 0 0 0 0 1
asm_1.step_011 00:10 1 0 0 0 0 0
asm_1.step_012 00:04 0 0 0 1 0 0
asm_1.step_013 00:05 0 0 1 0 0 0
asm_1.step_014 00:04 0 0 0 1 0 0
asm_1.step_015 00:08 0 0 0 0 0 1
asm_1.step_016 00:09 1 0 0 0 0 0
asm_1.step_017 00:01 0 1 0 0 0 0
asm_1.step_018 00:07 0 0 0 0 1 0
asm_1.step_019 00:05 0 1 0 0 0 0
asm_1.step_020 00:05 1 0 0 0 0 0
asm_1.step_021 00:05 0 0 1 0 0 0
asm_1.step_022 00:03 0 0 0 1 0 0
asm_1.step_023 00:08 0 0 0 0 1 0
asm_1.step_024 00:09 0 0 0 0 0 1
```

```
section2
asm_1.step_001 asm_1.step_002
asm_1.step_002 asm_1.step_003
asm_1.step_003 asm_1.step_004
asm_1.step_004 asm_1.step_005
asm_1.step_005 asm_1.step_006
asm_1.step_007 asm_1.step_008
asm_1.step_008 asm_1.step_009
asm_1.step_009 asm_1.step_010
asm_1.step_010 asm_1.step_011
asm_1.step_011 asm_1.step_012
asm_1.step_013 asm_1.step_014
asm_1.step_014 asm_1.step_015
asm_1.step_015 asm_1.step_016
asm_1.step_016 asm_1.step_017
asm_1.step_017 asm_1.step_018
asm_1.step_019 asm_1.step_020
asm_1.step_020 asm_1.step_021
asm_1.step_021 asm_1.step_022
asm_1.step_022 asm_1.step_023
asm_1.step_023 asm_1.step_024

section3

section4

Assembly Zone Maximum Occupancy

Zone.Z0 1
Zone.Z1 1
Zone.Z2 1
Zone.Z3 1
Zone.Z4 1
Zone.Z5 1
```

Problem 2: We took a subset of 4 jobs out of the original problem.

```
section1
asm_1.step_001 01:23 0 0 0 1 0 0 0
asm_1.step_002 00:01 0 0 0 0 0 0 1
asm_1.step_003 01:36 0 0 1 0 0 0 0
asm_1.step_004 00:54 1 0 0 0 0 0 0
```

```
asm_1.step_005 00:30 0 0 0 0 1 0 0
asm_1.step_006 01:20 0 1 0 0 0 0 0
asm_1.step_007 01:21 0 0 0 0 0 1 0
asm_1.step_008 00:09 0 0 0 1 0 0 0
asm_1.step_009 00:49 0 0 0 0 1 0 0
asm_1.step_010 00:32 0 0 0 0 0 0 1
asm_1.step_011 00:19 1 0 0 0 0 0 0
asm_1.step_012 01:32 0 1 0 0 0 0 0
asm_1.step_013 01:05 0 0 0 0 0 1 0
asm_1.step_014 01:28 0 0 1 0 0 0 0
asm_1.step_015 01:04 0 0 0 0 1 0 0
asm_1.step_016 00:04 1 0 0 0 0 0 0
asm_1.step_017 01:08 0 0 0 0 0 1 0
asm_1.step_018 01:19 0 1 0 0 0 0 0
asm_1.step_019 00:21 0 0 0 0 0 0 1
asm_1.step_020 01:24 0 0 0 1 0 0 0
asm_1.step_021 01:32 0 0 0 0 1 0 0
asm_1.step_022 01:06 0 0 0 1 0 0 0
asm_1.step_023 00:51 0 0 0 0 1 0 0
asm_1.step_024 01:23 1 0 0 0 0 0 0
asm_1.step_025 01:36 0 0 1 0 0 0 0
asm_1.step_026 01:08 0 1 0 0 0 0 0
asm_1.step_027 00:38 0 0 0 0 0 1 0
asm_1.step_028 00:38 0 0 0 0 0 0 1

section2
asm_1.step_001 asm_1.step_002
asm_1.step_002 asm_1.step_003
asm_1.step_003 asm_1.step_004
asm_1.step_004 asm_1.step_005
asm_1.step_005 asm_1.step_006
asm_1.step_006 asm_1.step_007
asm_1.step_008 asm_1.step_009
asm_1.step_009 asm_1.step_010
asm_1.step_010 asm_1.step_011
asm_1.step_011 asm_1.step_012
asm_1.step_012 asm_1.step_013
asm_1.step_013 asm_1.step_014
asm_1.step_015 asm_1.step_016
asm_1.step_016 asm_1.step_017
asm_1.step_017 asm_1.step_018
```

```
asm_1.step_018 asm_1.step_019
asm_1.step_019 asm_1.step_020
asm_1.step_020 asm_1.step_021
asm_1.step_022 asm_1.step_023
asm_1.step_023 asm_1.step_024
asm_1.step_024 asm_1.step_025
asm_1.step_025 asm_1.step_026
asm_1.step_026 asm_1.step_027
asm_1.step_027 asm_1.step_028

section3

section4

Assembly Zone Maximum Occupancy

Zone.Z0 1
Zone.Z1 1
Zone.Z2 1
Zone.Z3 1
Zone.Z4 1
Zone.Z5 1
Zone.Z6 1
```

Problem 3: We took a subset of 4 jobs out of the original problem.

```
section1
asm_1.step_001 00:51 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
asm_1.step_002 00:43 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
asm_1.step_003 01:20 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
asm_1.step_004 00:18 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
asm_1.step_005 00:38 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
asm_1.step_006 00:24 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_007 01:07 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_008 00:15 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
asm_1.step_009 00:24 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
asm_1.step_010 01:12 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
asm_1.step_011 00:45 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
asm_1.step_012 01:20 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
asm_1.step_013 01:04 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
asm_1.step_014 00:44 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_015 01:28 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
asm_1.step_016 00:40 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
asm_1.step_017 01:28 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
asm_1.step_018 01:17 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
asm_1.step_019 00:59 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
asm_1.step_020 00:20 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
asm_1.step_021 00:52 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_022 01:10 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
asm_1.step_023 00:40 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_024 00:32 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
asm_1.step_025 01:16 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
asm_1.step_026 00:43 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
asm_1.step_027 00:31 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
asm_1.step_028 00:21 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_029 00:05 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
asm_1.step_030 00:47 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_031 00:32 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_032 00:49 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_033 00:05 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
asm_1.step_034 01:04 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
asm_1.step_035 00:58 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
asm_1.step_036 01:20 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
asm_1.step_037 01:34 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
asm_1.step_038 00:11 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
asm_1.step_039 00:26 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_040 00:26 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
asm_1.step_041 00:59 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
asm_1.step_042 01:25 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
asm_1.step_043 00:47 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
asm_1.step_044 01:36 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
asm_1.step_045 00:14 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_046 00:23 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
asm_1.step_047 00:09 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
asm_1.step_048 01:15 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_049 00:37 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
asm_1.step_050 00:43 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
asm_1.step_051 01:19 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_052 01:15 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
asm_1.step_053 00:34 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
asm_1.step_054 00:20 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
asm_1.step_055 00:10 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
asm_1.step_056 01:23 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

```
asm_1.step_057 01:08 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
asm_1.step_058 00:52 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
asm_1.step_059 01:06 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
asm_1.step_060 00:09 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

section2
asm_1.step_001 asm_1.step_002
asm_1.step_002 asm_1.step_003
asm_1.step_003 asm_1.step_004
asm_1.step_004 asm_1.step_005
asm_1.step_005 asm_1.step_006
asm_1.step_006 asm_1.step_007
asm_1.step_007 asm_1.step_008
asm_1.step_008 asm_1.step_009
asm_1.step_009 asm_1.step_010
asm_1.step_010 asm_1.step_011
asm_1.step_011 asm_1.step_012
asm_1.step_012 asm_1.step_013
asm_1.step_013 asm_1.step_014
asm_1.step_014 asm_1.step_015
asm_1.step_016 asm_1.step_017
asm_1.step_017 asm_1.step_018
asm_1.step_018 asm_1.step_019
asm_1.step_019 asm_1.step_020
asm_1.step_020 asm_1.step_021
asm_1.step_021 asm_1.step_022
asm_1.step_022 asm_1.step_023
asm_1.step_023 asm_1.step_024
asm_1.step_024 asm_1.step_025
asm_1.step_025 asm_1.step_026
asm_1.step_026 asm_1.step_027
asm_1.step_027 asm_1.step_028
asm_1.step_028 asm_1.step_029
asm_1.step_029 asm_1.step_030
asm_1.step_031 asm_1.step_032
asm_1.step_032 asm_1.step_033
asm_1.step_033 asm_1.step_034
asm_1.step_034 asm_1.step_035
asm_1.step_035 asm_1.step_036
asm_1.step_036 asm_1.step_037
asm_1.step_037 asm_1.step_038
```

```
asm_1.step_038 asm_1.step_039
asm_1.step_039 asm_1.step_040
asm_1.step_040 asm_1.step_041
asm_1.step_041 asm_1.step_042
asm_1.step_042 asm_1.step_043
asm_1.step_043 asm_1.step_044
asm_1.step_044 asm_1.step_045
asm_1.step_046 asm_1.step_047
asm_1.step_047 asm_1.step_048
asm_1.step_048 asm_1.step_049
asm_1.step_049 asm_1.step_050
asm_1.step_050 asm_1.step_051
asm_1.step_051 asm_1.step_052
asm_1.step_052 asm_1.step_053
asm_1.step_053 asm_1.step_054
asm_1.step_054 asm_1.step_055
asm_1.step_055 asm_1.step_056
asm_1.step_056 asm_1.step_057
asm_1.step_057 asm_1.step_058
asm_1.step_058 asm_1.step_059
asm_1.step_059 asm_1.step_060

section3

section4

Assembly Zone Maximum Occupancy

Zone.Z0 1
Zone.Z1 1
Zone.Z2 1
Zone.Z3 1
Zone.Z4 1
Zone.Z5 1
Zone.Z6 1
Zone.Z7 1
Zone.Z8 1
Zone.Z9 1
Zone.Z10 1
Zone.Z11 1
Zone.Z12 1
Zone.Z13 1
Zone.Z14 1
```

APPENDIX B

MAPLE CODE

This chapter contains code in the Maple language that we used in this dissertation to produce simplifications of equations and calculations of expected costs to solution. The code consists of a main file called `probpm-clean.mpl` which contains code to calculate the expected cost given a cutoff policy. It also contains code to hill climb in the space of cutoff policies. This file requires `lds-good.mpl` and `lds-formula-numeric.mpl`. The code in `lds-good.mpl` calculates the cost of the iteration where we assume success and the code in `lds-formula-numeric.mpl` numerically calculates the cost of the iteration where we assume success, using the simplifications explained in the following section.

B.1 Simplification of $N_{succ}(c)$

Using `lds-good.mpl`, we generate expressions for particular depth, discrepancy value, heuristic probability, p, and one last parameter, w. This last parameter is in place for the distribution of weights. In the case of LDS, this distribution is constant, but in the case of WDS it is not, and the formula is expressed so that it can be used for either LDS or WDS. If it is used to calculate the expected cost for WDS, w has to be replaced by a call to the weight distribution.

The general strategy here is to try to collect terms in order to find a pattern. The work is too detailed and extensive to present in its entirety so we will just present the case for a tree of depth 2 and cutoff of 1. The objective is to show that the recursive formula for exploring a good node (4.9) corresponds to (4.17). In order to do this, we have isolated the coefficients of terms of the form $w^k(1-p)^k$. These coefficients are the ones that appear as $C_k(i_1,\ldots,i_k,h)$ in (4.17).

Starting from (4.9) we will rearrange terms in order to establish a correspondence between these terms and the expressions in Table 46. First, we load the maple routines:

```
> read 'lds-good.mpl':
```

Then, we invoke the procedure at depth 2, with at most 2 discrepancies and heuristic probability p, and weight w.

```
> good(2,0,2,p,w);
```

$$1+p\,(1+p+(1-p)\,(1+w))+(1-p)\,(2+w+w\,(1+p+(1-p)\,(1+w)))$$

The weight w does not stand for the actual value of the weight, it stands for the probability that the weight is greater than or equal to the cutoff. This probability is one if the exponent of the weight is smaller than or equal to the cutoff. We will use this fact later when we determine the expression for one cutoff value.

The terms are roughly as follows: 1 is the cost of the root, then we either succeed on the left with probability p times the probability of the left subtree or we fail on the left and succeed on the right subtree (which is given by the third term of the sum).

Let us focus on the second term. This is the case where we succeed on the left. We do so if we count the root of that subtree, and with probability p we succeed on the left of that subtree and with probability $1-p$ we succeed on the right in which case, we explore the left node and the right (denoted by w). This w denotes a right turn at depth 2. Although this fact is not important now, it will be important when we collect the terms and show that Table 46 generates the correct expressions.

Now focus on the case where we failed on the left subtree. In this case, we either succeed on the left (and explore 2 nodes plus the right turn) or we fail on the left and succeed on the right, and so on.

First a trick. We want to preserve the term (1-p) so we will substitute this term for m.

```
> subs(1-p=m,%);
```

$$1+p\,(1+p+m\,(1+w))+m\,(2+w+w\,(1+p+m\,(1+w)))$$

In order to illustrate the rest of the process, we will settle on a single cutoff. Let us focus on the second iteration, that is when the cutoff is 1. We need to eliminate from this expression all terms with w^2, since those will be the ones representing the cost of exploring paths with two right branches.

```
> subs(w^2=0,simplify(%));
```

$$1+p+p^2+p\,m+2\,p\,m\,w+2\,m+2\,m\,w+m^2\,w$$

Next, we collect the terms of the expression in terms of m and the weight, w.

```
> collect(collect(%,m),w);
```

$$((2\,p+2)\,m+m^2)\,w+p+(p+2)\,m+1+p^2$$

Notice that the terms that do not multiply w can be simplified to equal $1+d$:

```
> simplify(subs(m=1-p,p+(p+2)*m+1+p^2));
```

$$3$$

Let us focus on the two terms that multiply w: $(2p+2)m$ and m^2. These are the terms that should correspond to terms in Table 46. We have not explained the indices i_k. There are as many indices as possible right turns. So in the case where the cutoff is 1, there is only one index, namely i_1. The indices in Table 46 have the following ranges: i_1 ranges from 0 to $d-1$, and for each successive subindex, k, i_k ranges from 0 to $i_{k-1}-1$. So index i_1 ranges from 0 to 1. This means that in order to get the factor of m^2w we must add the result of replacing i_1 for 0 with the result of replacing i_1 for 1 in the entry that lies in the intersection between $(1-p)^2$ and w in Table 46. The expression in Table 46 that intersects w and $(1-p)^2$ should be equal to 1. In the case where $i_1 = 0$, this term is equal to 1, and when $i_1 = 1$, it is equal to 0, so in this case, the expression in the table is correct.

Let us verify that the factor of m and w, $2p+2$, is also correct. When $i_1 = 0$, the expression in Table 46 that intersects w and $(1-p)$ is equal to $1+p$. When $i_1 = 1$, this expression equals $1+p$. The sum of both expressions is $2+2p$ as desired.

This is not proof that the expressions in the table are correct. It is just an example. The reader who wishes to verify the rest of the expressions may do so by using the code we provide below and the method we illustrated above.

Notice there are terms with $(1-p)^k$ and terms with w^k. Collecting the coefficients of these terms yields the expressions in Table 46.

The following code implements the recursive calculation of success.

```
#
#  lds-good.mpl : This is the precursor to lds-formula-numeric.mpl
#


#
# good: depth is the total depth of the tree,
#       k is the curent depth,
#       p and w are the parameters
#
# Invoke as good(depth,0,disc,p,w)
#

good := proc(depth,k,c,p,w)

  local kk,dk,nw,expr,x,y,z;

  if k = depth then
    expr := 1;
  else
    if c = 0 then
      expr := 1 + good(depth,k+1,c,p,w);
```

TABLE 46. Collection of coefficients of terms $(1-p)^h w^k$.

Weight	$(1-p)$
1	$\sum_{k=0}^{d-1}(d-k-0)p^k$
w	$(i_1+1)\sum_{k=0}^{d-i_1-2} p^k + \sum_{k=d-i_1-1}^{d-1} p^k$
w^2	$(i_2+1)\sum_{k=0}^{d-i_1-2} p^k$
w^3	$(i_3+1)\sum_{k=0}^{d-i_1-2} p^k$
w^4	$(i_4+1)\sum_{k=0}^{d-i_1-2} p^k$
Weight	$(1-p)^2$
1	
w	$\sum_{k=d-i_1-1}^{d-2}(d-k-1)p^k$
w^2	$(i_2+1)\sum_{k=d-i_1-1}^{d-i_2-3} p^k + \sum_{k=d-i_2-2}^{d-2} p^k$
w^3	$(i_3+1)\sum_{k=d-i_1-1}^{d-i_2-3} p^k$
w^4	$(i_4+1)\sum_{k=d-i_1-1}^{d-i_2-3} p^k$
Weight	$(1-p)^3$
1	
w	
w^2	$\sum_{d-i_2-2}^{d-3}(d-k-2)p^k$
w^3	$(i_3+1)\sum_{k=d-i_2-2}^{d-i_3-4} p^k + \sum_{k=d-i_3-3}^{d-3} p^k$
w^4	$(i_4+1)\sum_{k=d-i_2-2}^{d-i_3-4} p^k$
Weight	$(1-p)^4$
1	
w	
w^2	
w^3	$\sum_{k=d-i_3-3}^{d-4}(d-k-3)p^k$
w^4	$(i_4+1)\sum_{k=d-i_3-3}^{d-i_4-5} p^k + \sum_{k=d-i_4-4}^{d-4}(d-k-3)p^k$

```
    else
      kk := k+1;
      nw := w;
      x := good(depth,kk,c,p,w);
      y := bad(depth,kk,c,p,nw);
      z := good(depth,kk,c-1,p,nw);
      expr := 1+p*x+(1-p)*(y+nw*z);
    fi;
  fi;

end;

bad := proc(depth,k,c,p,w)

  local sum,expr,dk,dj,j,nw,x;

  if k = depth then
    expr := 1;
  else
    if c = 0 then
      expr := 1+depth-k;
    else
      dk := depth-k;
      sum := 0;
      for j from 0 to dk-1 do
        dj := depth-j;
        nw := w;
        x := bad(depth,dj,c-1,p,nw);
        sum := sum + nw * x;
      od;
      expr := 1+dk+sum;
    fi;
  fi;

end;
```

The following code calculates the expected number of nodes of the successful iteration according to the nested sums in (4.17).

```
#
```

```
#  lds-formula-numeric.mpl: computes the cost of the
#  iteration where we expect to succeed.
#


coef := proc()

  local j,k,c1,i,d,p;

  j := nargs-3;
  k := args[nargs-2];
  d := args[nargs-1];
  p := args[nargs];
  if k <= 0 then
     ERROR ('last argument should be greater than zero.');
  fi;
  if k = j then
    if k = 1 then
      c1 := ((args[j]+1)*sum(p^i,i=0..d-args[j]-(j+1)))
            +sum(p^i,i=(d-args[j]-j)..(d-j));
    else
      c1 := ((args[j]+1)
       *sum(p^i,i=((d-args[j-1]-(j-1))..(d-args[j]-(j+1)))))
       +sum(p^i,i=(d-args[j]-j)..(d-j));
    fi;
  elif k < j then
    if k = 1 then
      c1 := (args[j]+1)*sum(p^i,i=0..d-args[k]-(k+1));
    else
      c1 := (args[j]+1)
*sum(p^i,i=(d-args[k-1]-(k-1))..(d-args[k]-(k+1)));
    fi;
  elif j < k then
    c1 := sum((d-i-j)*p^i,i=(d-args[j]-j)..(d-(j+1)));
  fi;
  c1
end;

#  To calculate the weight, we record the depths at which
#  we make right turns.
# wcond := proc()
#   local l,k,i;
#   l := [d-args[1]];
#   for i from 2 to nargs do
#    k := op(l);
#    l := [d-args[i],k];
#   od;
#  l :=[seq(d-i, i=args[1]..args[nargs])];
#   l
# end;

# Here, the weight is constant.

wcond := proc(w)
  local l;
  l := w;
  l
end;

formula := proc(p,w,depth,cutoff)

  local sum1,sum2,j,k,total;
  global summand,C,W,a,b;

  for j from 1 to depth do
    summand[j] := 0;
  od;
  if (1 <= cutoff) then
    b[1] := 1;
    for a[1] from 0 to depth-1 do
      new_loop(depth,p,w,a[1],b[1],cutoff);
      if (2 <= cutoff) then
        a_loop(depth,p,w,2,cutoff);
      fi;
    od;
  fi;
  total := 1+depth+sum(summand[k],k=1..depth);
  sort(expand(total));

end;
```

```
a_loop := proc(depth,p,w,v,c)

  global a,b,ind;

  b[v] := b[v-1]+1;
  for a[v] from 0 to ind[b[v]-1]-1 do
    new_loop(depth,p,w,a[v],b[v],c);
    if (v < c) then
      a_loop(depth,p,w,v+1,c);
    fi;
  od;

end;

new_loop := proc(depth,p,w,value,index,cutoff)

  local h,l,s,arguments,wfactor;
  global ind,C,W,summand;

  ind[index] := value;
  arguments[index] := seq(ind[h], h=1..index);
  W[index] := wcond(w,arguments[index]);
  for l from 1 to index+1 do
    C[index][l] := coef(arguments[index],l,depth,p);
  od;
  wfactor := product(W[n],n=1..index);
  summand[index] := summand[index] +
                    wfactor*sum((1-p)^s*C[index][s],s=1..index+1);

end;
```

B.2 Expected Cost Calculation

The following code computes the expected cost to solution. In order to calculate the expected cost to solution given $p = 0.8$, $m = 0.2$, a policy c for a tree of depth 10 and policy length 5 invoke `ndep_cost2(0.8,0.2,c,10,5)`.

```
#
#  probpm-clean.mpl: computes the expected cost
#  to solution.
```

```
#

fprob2 := proc(depth,p,m)

  local i,k;
  global fp2;

  for i from 0 to depth do
    fp2[i][0] := 1-p^i;
    fp2[0][i] := 0;
    fp2[i][i] := 0;
    fp2[i][depth] := 0;
  od;

  for i from 2 to depth do
    for k from 1 to i-1 do
#      if not assigned(fp2[i][k]) then
        fp2[i][k] := (1-2*m)*fp2[i-1][k]*fp2[i-1][k-1]+
             (p+2*m-1)*fp2[i-1][k]+
                     (1-p)*fp2[i-1][k-1];
#      fi;
    od;
  od;
end;

count2 := proc(depth)

  local i,k;
  global nc;

  for i from 0 to depth do
    nc[i][0] := 1+i;
    nc[i][depth] := 0;
    nc[i][i] := 2^(i+1)-1;
    nc[0][i] := 1;
  od;

  for i from 2 to depth do
    for k from 1 to i-1 do
#      if not assigned(nc[i][k]) then
        nc[i][k] := 1+nc[i-1][k]+nc[i-1][k-1];
```

```
#      fi;
    od;
  od;

end;

Digits := 20;
with(plots);

read `lds-formula-numeric.mpl`:

read `lds-good.mpl`;

ndep_cost2 :=  proc(hp,hm,policy,depth,length)

  local i,j,k,d,lscount,prob,node_cnt,total;
  global fp2,nc,node_cnt_num;

  d := depth;

#  if not assigned(nc[d][d]) then
    count2(d);
#  fi;

#  if not assigned(fp2[d][d]) then
    fprob2(d,hp,hm);
#  fi;

  total := 0;

  lscount[0] := nc[d][policy[0]];
  total := (1-fp2[d][policy[0]])*lscount[0];
  for i from 1 to length do
    prob := fp2[d][policy[i-1]]-fp2[d][policy[i]];
    lscount[i] := nc[d][policy[i]];
    node_cnt := sum(nc[d][policy[j]],j=0..i-1)+lscount[i];
    total := total + prob * node_cnt;
  od;

  total;
end;
```

B.3 Hill Climbing on OCP Space

The following code does hill climbing on the space of policies. There is a noise parameter that controls when we make a sideways move instead of an uphill move.

In order to obtain all OCPs between the range of $p = 0.8$ and $p = 0.9$, with $m = 0.2$, for a tree of depth 10, invoke `get_ocp2(0.8,0.2,10)`. To calculate the OCP at a single point, invoke `get_ocp(0.8,0.2,10)`. To compute the costs of all neighbor policies, invoke `explore_neighborhood(0.8,0.2,ocp,10)` if the OCPis stored in variable `ocp`.

```
#
# The following code is also part of the file probpm-clean.mpl.
#

find_ocp := proc(p,m,w,depth,drnd,wrnd)

  local i,idx,val,wcp,cost,lrnd,rnd,li,ll,oc,cp,length,printing;
  global iterations,ocp,ocost,progress,maxiter;

  printing := 0;

  # when do we stop?
  length := nops(op(op(ocp)))-1;  # the last cutoff is always d
  if length = 0 or iterations > maxiter then
    RETURN(ocp,iterations);
  fi;

  ll := length;
  progress := progress+1;
  iterations := iterations+1;
  wcp := 'wcp';
  cp := copy(ocp);
  lrnd := rand(length);
#  w;
  if wrnd() < w then
    # we make a downhill move
#    li := lrnd(); # pick a random iteration and delete it
    # instead of picking randomly, pick the best one from
```

```
      # all except the last iteration which has to stay in
      # the policy for completeness.
      ll := length-1;
      for li from 0 to ll do
        for i from 0 to li-1 do
          wcp[i] := cp[i];
        od;
        for i from li to length-1 do
          wcp[i] := cp[i+1];
        od;
        cost := ndep_cost2(p,m,wcp,depth,ll);
        if cost < ocost then
          ocost := cost;
          ocp := copy(wcp);
          if printing=1 then
            print (ocp);
            printf("new cost = %f\n",ocost);
          fi;
          maxiter := max(maxiter,2*iterations);
          progress := 0;
        fi;
      od;
    else
      # we make a random move
      idx := lrnd();
      if (idx >= length) then
        ERROR(`RANDOM is destroying the cutoff policy`); fi;
      if idx = 0 then
        rnd := rand(cp[1]);
      else
#         if idx = length then  # this is a redundant check!
#           rnd := rand(cp[length]);
#           print(cp[length]);
#         else
           rnd := rand(cp[idx-1]..cp[idx+1]);
#           print(cp[idx-1],cp[idx+1]);
#         fi;
      fi;
      val := rnd();
      if val = depth then  # we have decided to do DFS
        for i from 0 to idx-1 do
        wcp[i] := cp[i];
      od;
      ll := idx;
    else
      wcp := copy(cp);
    fi;
    wcp[idx] := val;
    cost := ndep_cost2(p,m,wcp,depth,ll);
    if cost < ocost then
      ocost := cost;
      ocp := copy(wcp);
      if printing=1 then
        print (ocp);
        printf("new cost = %f\n",ocost);
      fi;
      maxiter := max(maxiter,2*iterations);
      progress := 0;
    fi;
  fi;

  find_ocp(p,m,w,depth,drnd,wrnd);

end;

get_ocp := proc(p,m,depth)

  local seed,length,cost,w,w_seed,drnd,wrnd;
  global iterations,ocp,ocost,progress,maxiter;

  w_seed := readlib(randomize)();
  drnd := rand(depth-1);
  wrnd := rand(2^32)/2.0^32;

  maxiter := 100;
  iterations := 0;
  progress := 0;
  seed := get_seed(depth);
  length := nops(op(op(seed)))-1;  # the last cutoff is always d
#  w := 0.15;
  w := 0.25;
#  w := 0.5;
```

```
  ocost := ndep_cost2(p,m,seed,depth,length);
  ocp := copy(seed);
  find_ocp(p,m,w,depth,drnd,wrnd);

  print(ocp,ocost,iterations,progress);

end;

get_ocp2 := proc(p,m,depth)

  local pi,i;
  global ocp2;

  for i from 0 to 10 do
   pi := p+i*.01;
   print (pi);
   ocp2[i] := get_ocp(pi,m,depth);
  od;

end;

get_seed := proc(depth)

  local i,cp;

  # seed with LDS

  for i from 0 to depth do
    cp[i] := i;
  od;

  # seed with all even iterations only

#  for i from 0 to floor(depth/2) do
#    cp[i] := 2*i;
#  od;
#  if floor(depth/2) <> ceil(depth/2) then cp[i] := depth; fi;

  cp;
 end;
```

```
explore_neighborhood := proc(p,m,policy,depth)

  local i,j,cp,co,length;

  length := nops(op(op(policy)))-1;

  print(length);
  for i from 0 to length-1 do
    cp := copy(policy);
    for j from i+1 to length do
      cp[j-1] := policy[j];
    od;
    co := ndep_cost2(p,m,cp,depth,length-1);
    print (cp);
    print (i,co);
  od;

end;
```

APPENDIX C

NUMBER PARTITIONING GRAPHS

The figures in this appendix complement the discussions in Section 6.5.2 on the behavior of the "best weights" in number partitioning problems parameterized by the maximum number of digits in the elements of the instance and the ratio which determines how many elements are in each instance. A ratio r, of 1.82 means there are $1.82 \log(N)$ elements in the instance. All figures show data for $N = 4$, $N = 5$, $N = 6$, $N = 7$, and $r = 1.82$, $r = 1.4$, $r = 1.2$, and $r = 1.0$.

Figure 53 through Figure 55 show the logarithm of the number of nodes explored in case of failure vs. the logarithm of the cutoff value. Figure 56 through Figure 59 show the logarithm of the number of nodes explored in case of success vs. the logarithm of the cutoff value. Figure 60 through Figure 63 show the logarithm of the probability of failure vs. the number of nodes in case of failure.

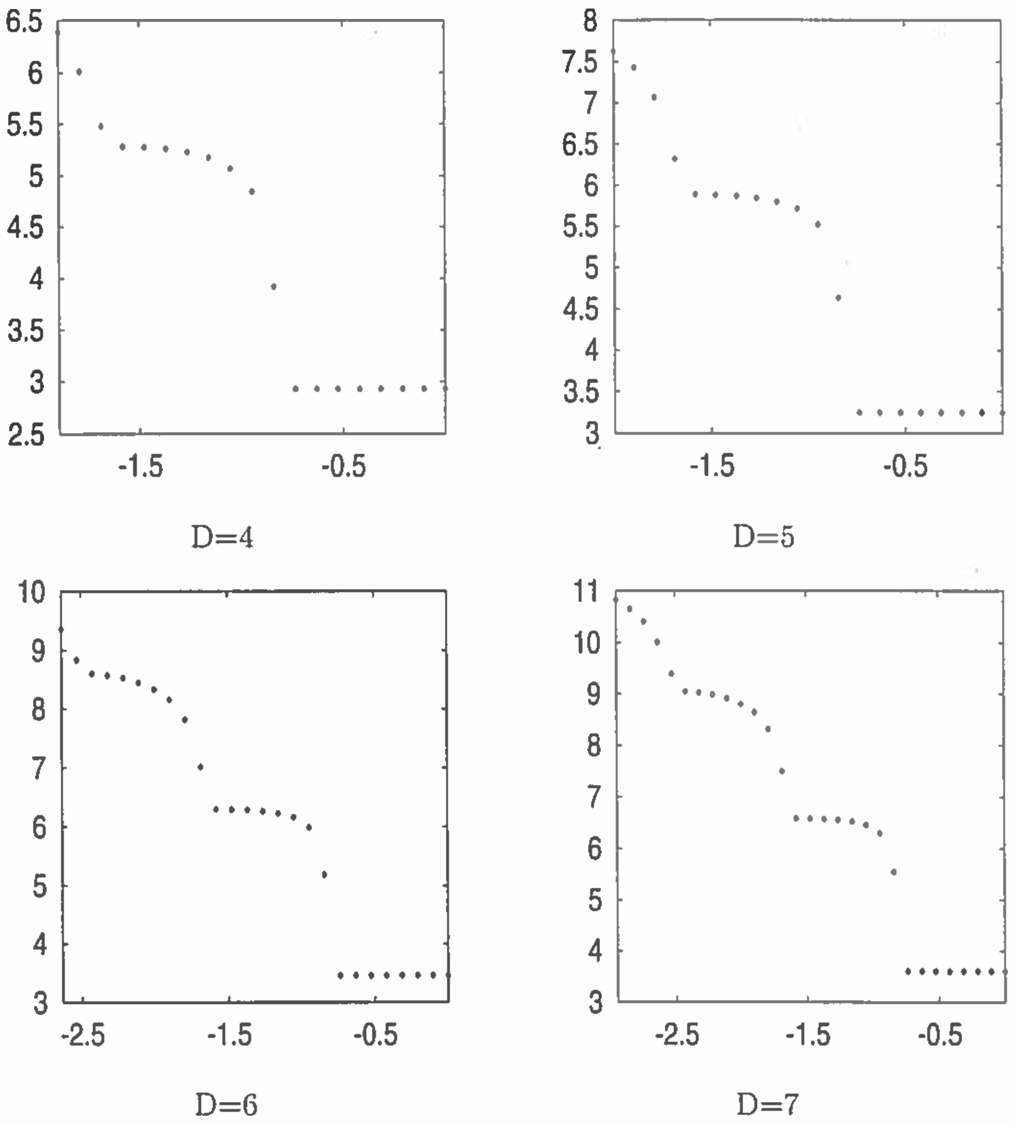


FIGURE 53. The y-axis shows the logarithm of the number of nodes explored in case of failure. The x-axis shows the logarithm of the cutoff value. $r = 1.82$.

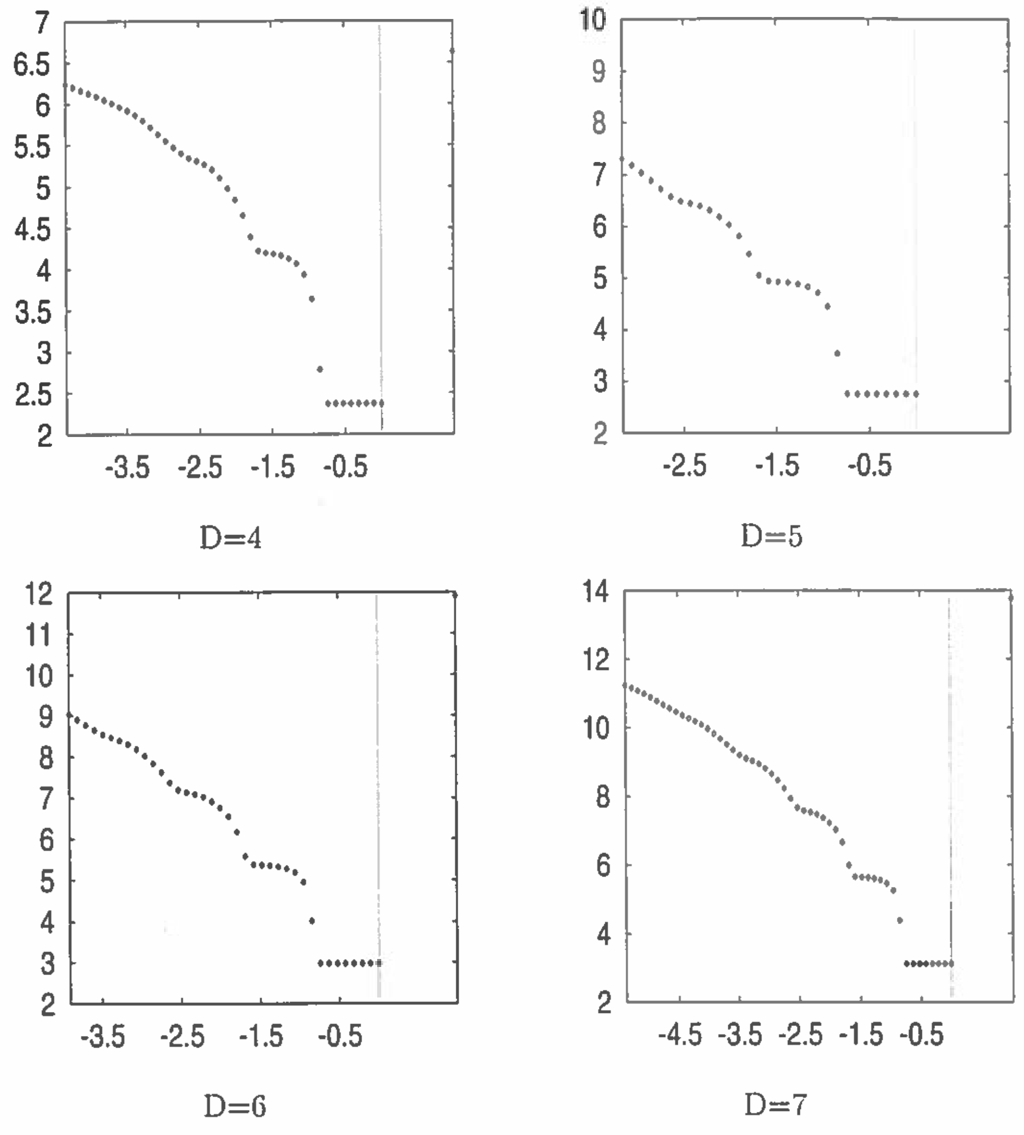


FIGURE 54. The y-axis shows the logarithm of the number of nodes explored in case of failure. The x-axis shows the logarithm of the cutoff value. $r = 1.2$.

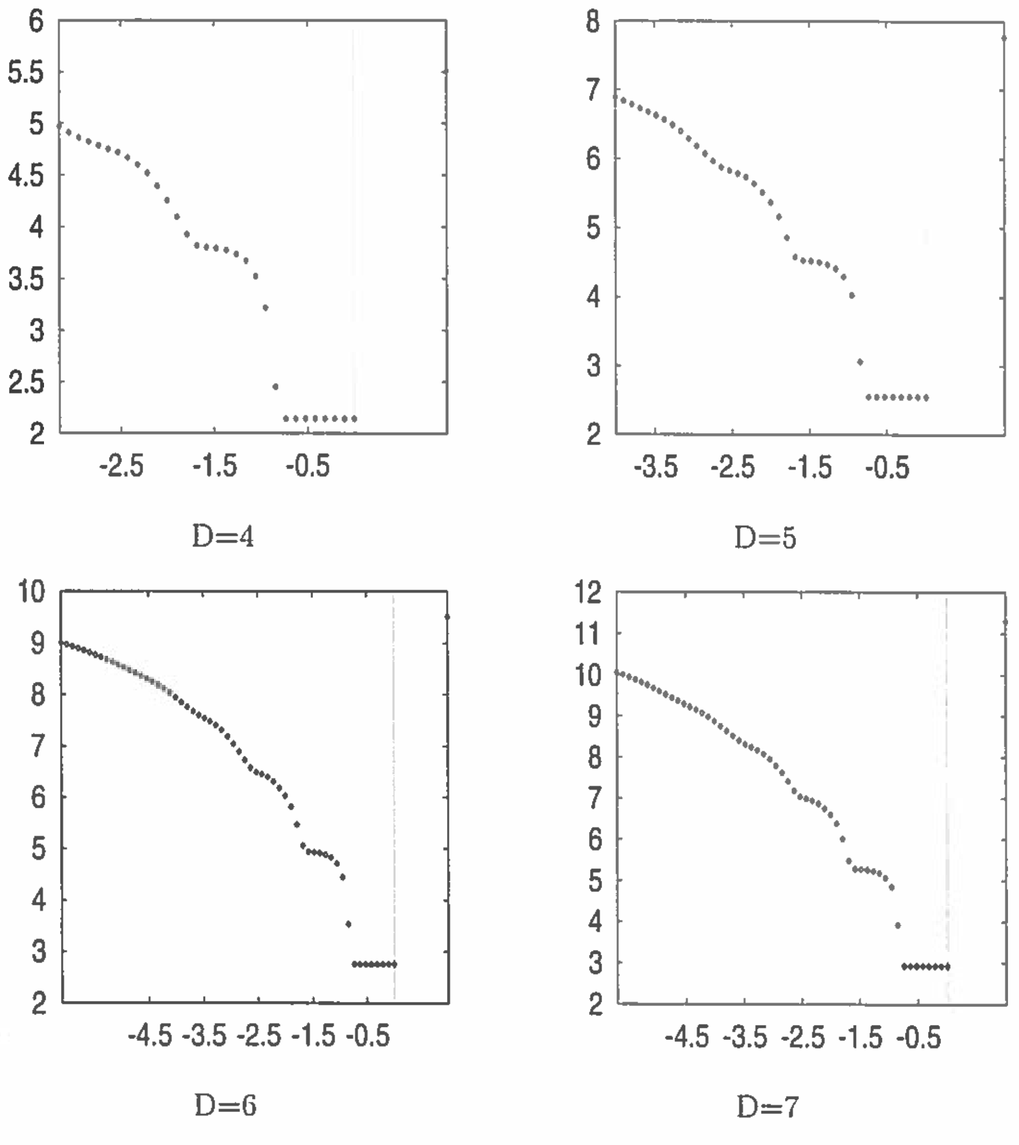


FIGURE 55. The y-axis shows the logarithm of the number of nodes explored in case of failure. The x-axis shows the logarithm of the cutoff value. $\tau = 1.0$.

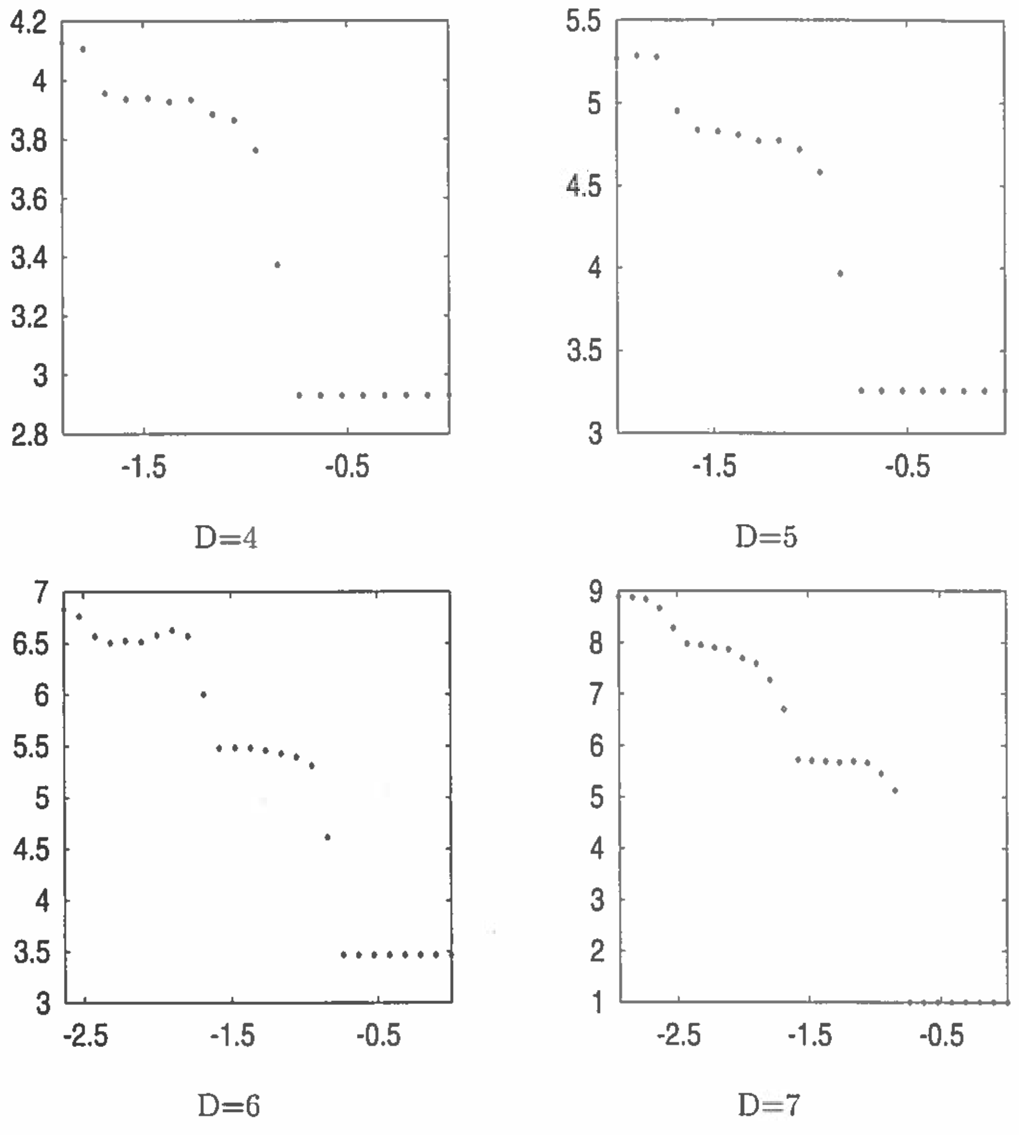


FIGURE 56. The y-axis shows the logarithm of the number of nodes explored in case of success. The x-axis shows the logarithm of the cutoff value. $r = 1.82$.

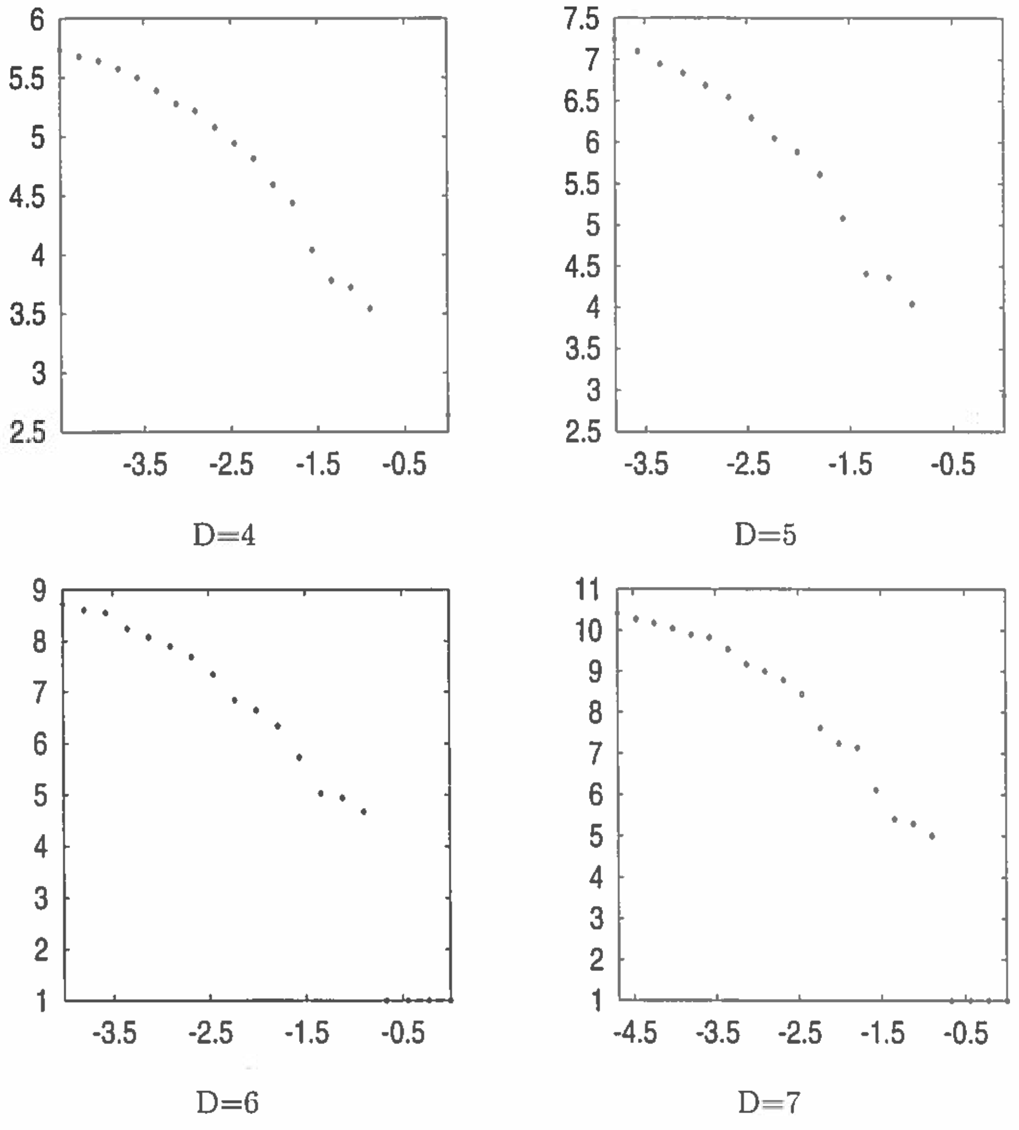


FIGURE 57. The y-axis shows the logarithm of the number of nodes explored in case of success. The x-axis shows the logarithm of the cutoff value. $r = 1.4$.

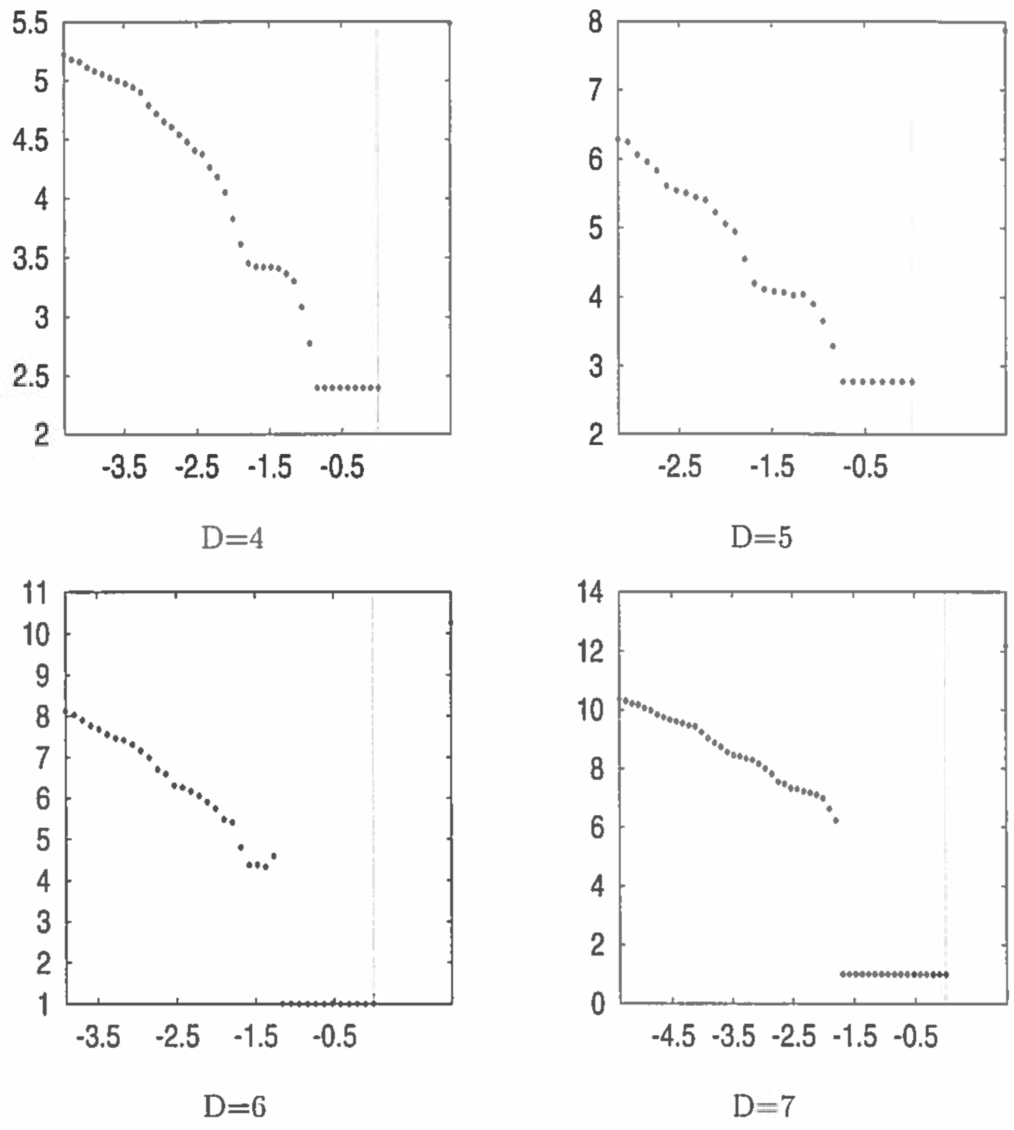


FIGURE 58. The y-axis shows the logarithm of the number of nodes explored in case of success. The x-axis shows the logarithm of the cutoff value. $r = 1.2$.

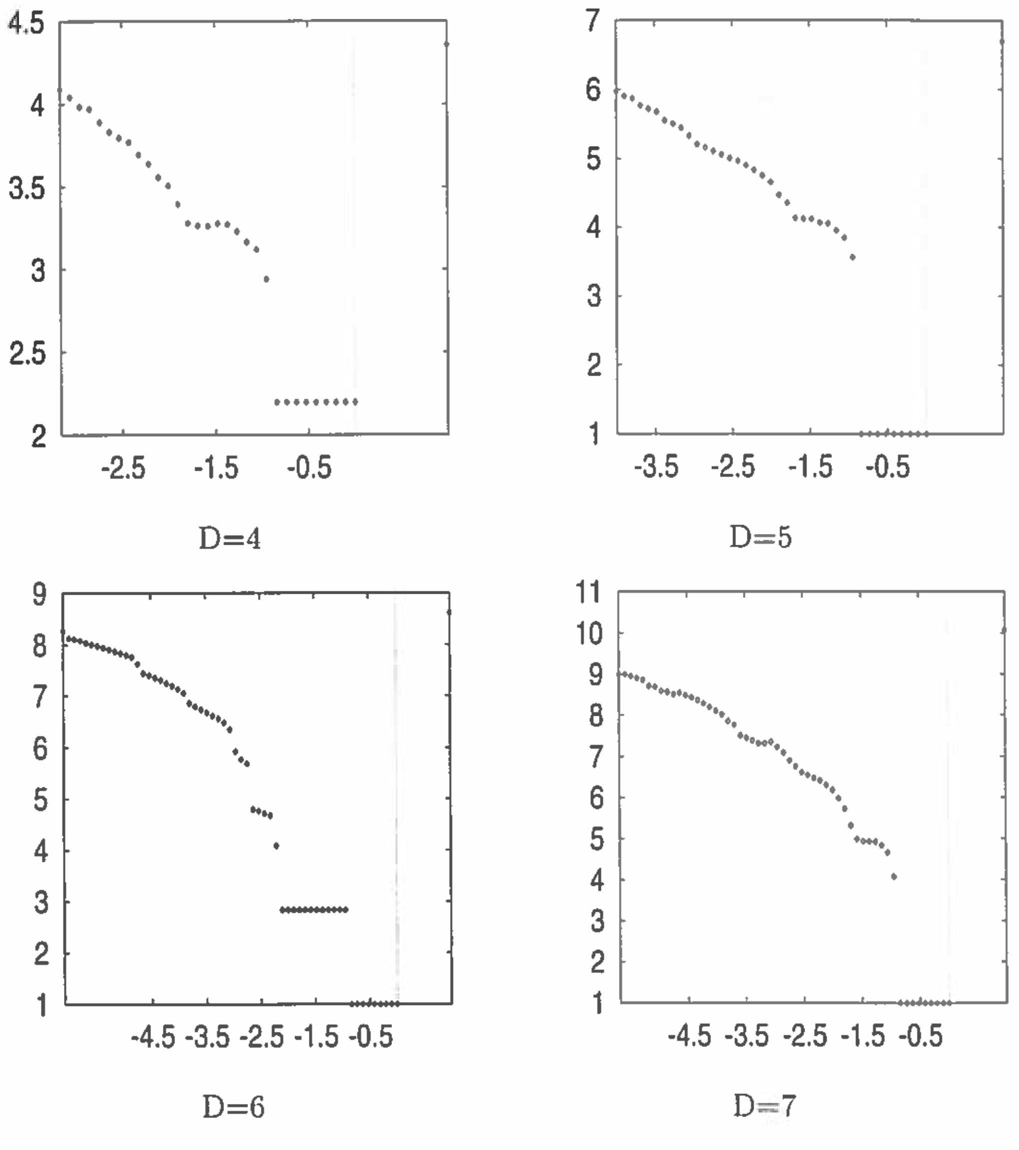


FIGURE 59. The y-axis shows the logarithm of the number of nodes explored in case of success. The x-axis shows the logarithm of the cutoff value. $r = 1.0$.

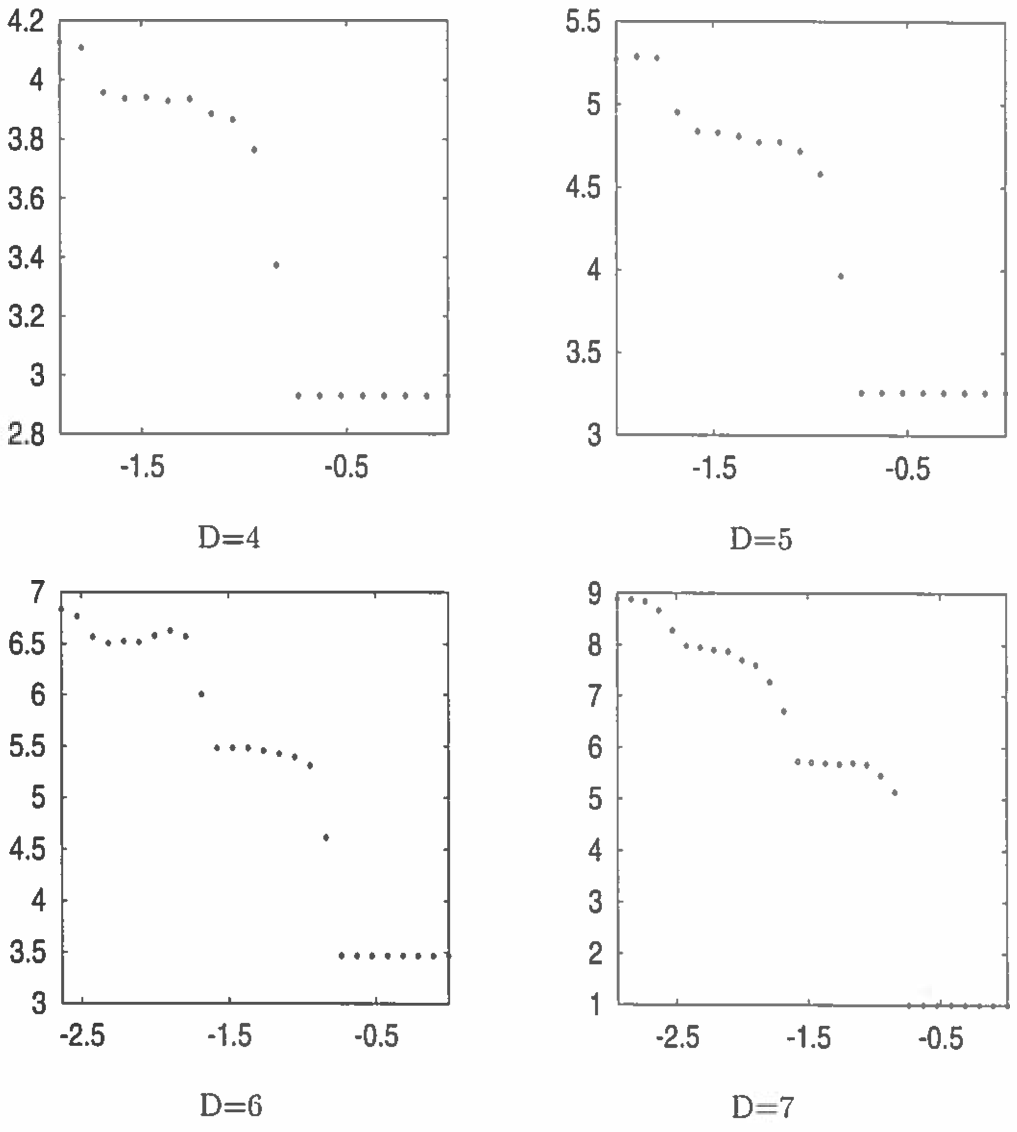


FIGURE 60. The y-axis shows the logarithm of the probability of failure. The x-axis shows the logarithm of the number of nodes explored. $r = 1.82$.

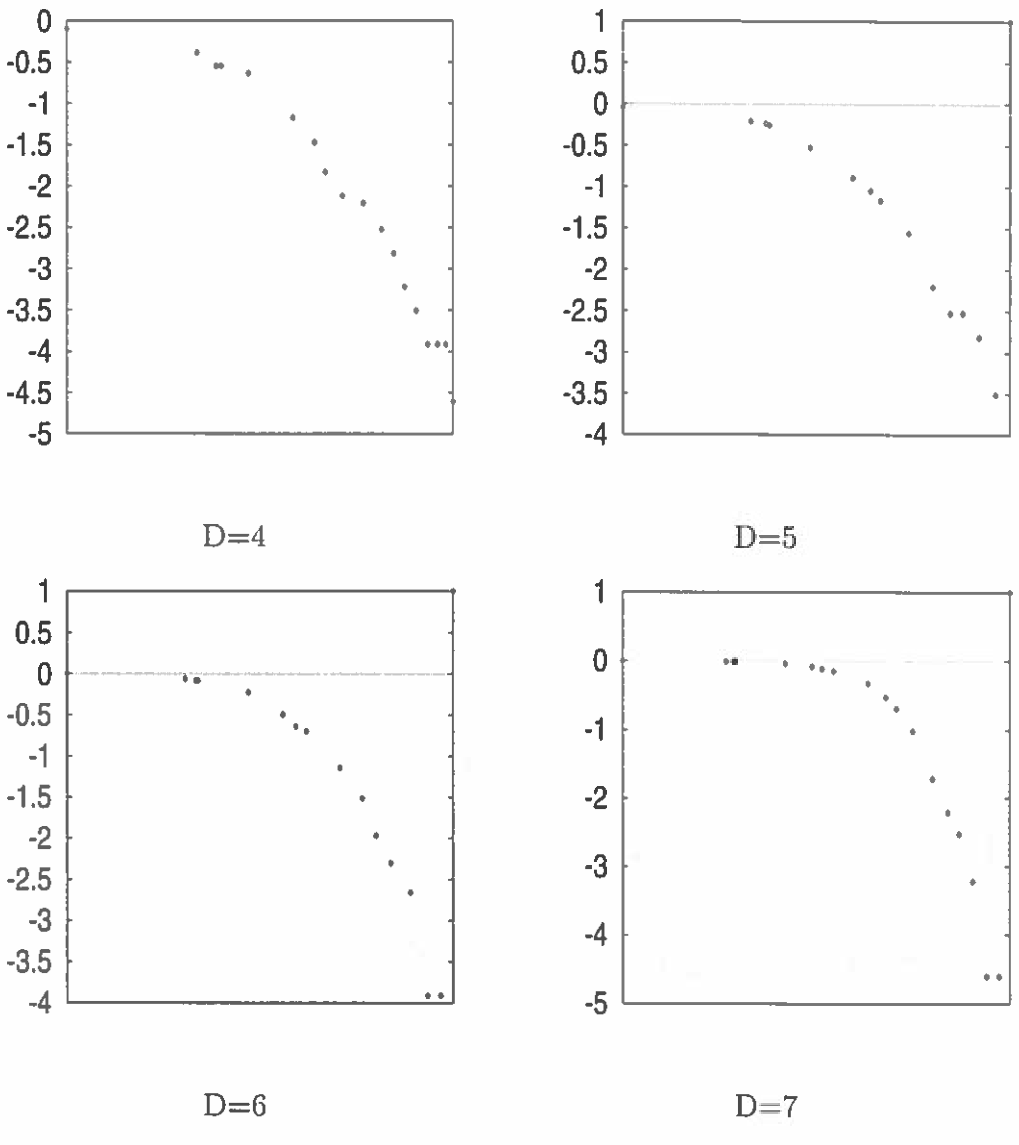


FIGURE 61. The y-axis shows the logarithm of the probability of failure. The x-axis shows the logarithm of the number of nodes explored. $r = 1.4$.

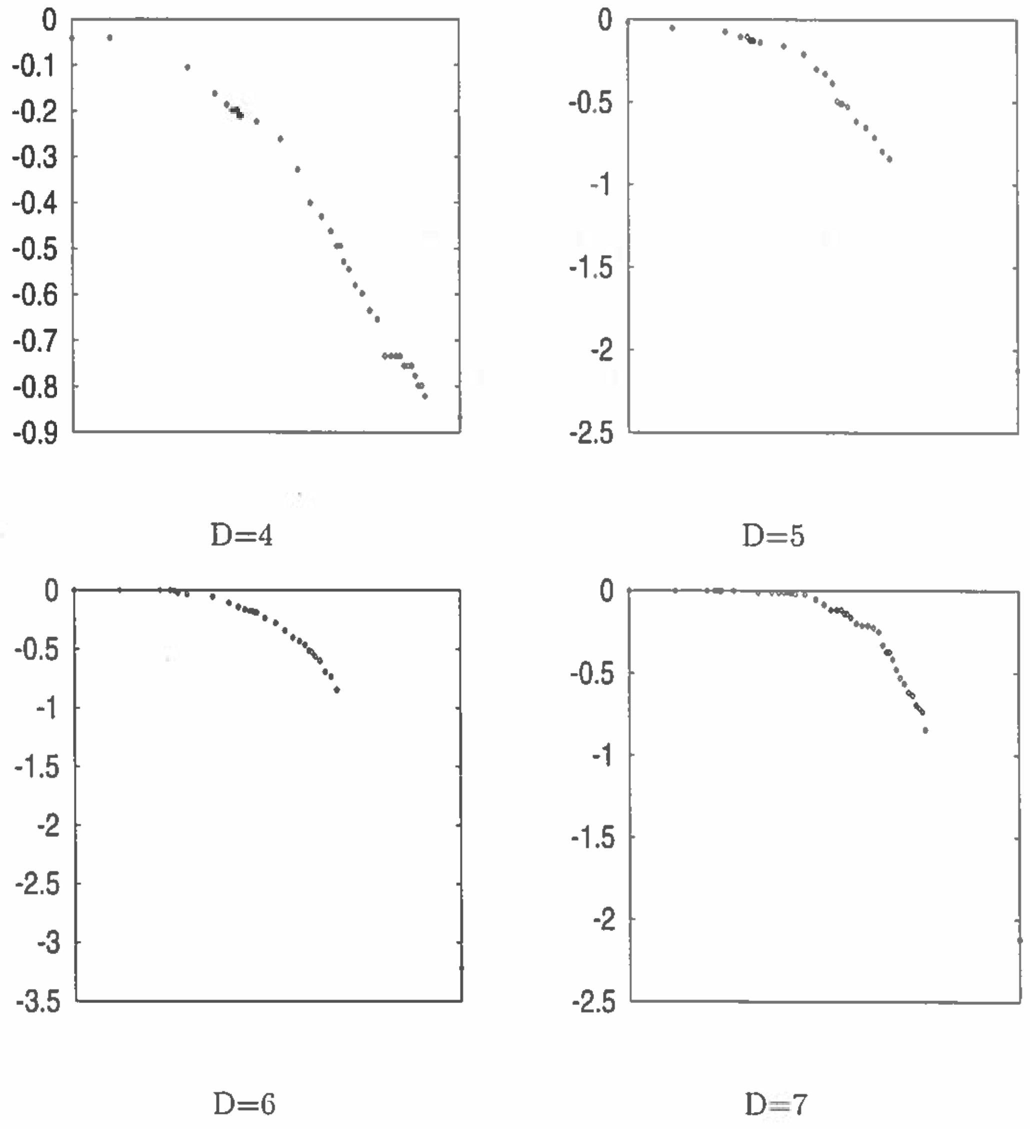


FIGURE 62. The y-axis shows the logarithm of the probability of failure. The x-axis shows the logarithm of the number of nodes explored. $r = 1.2$.

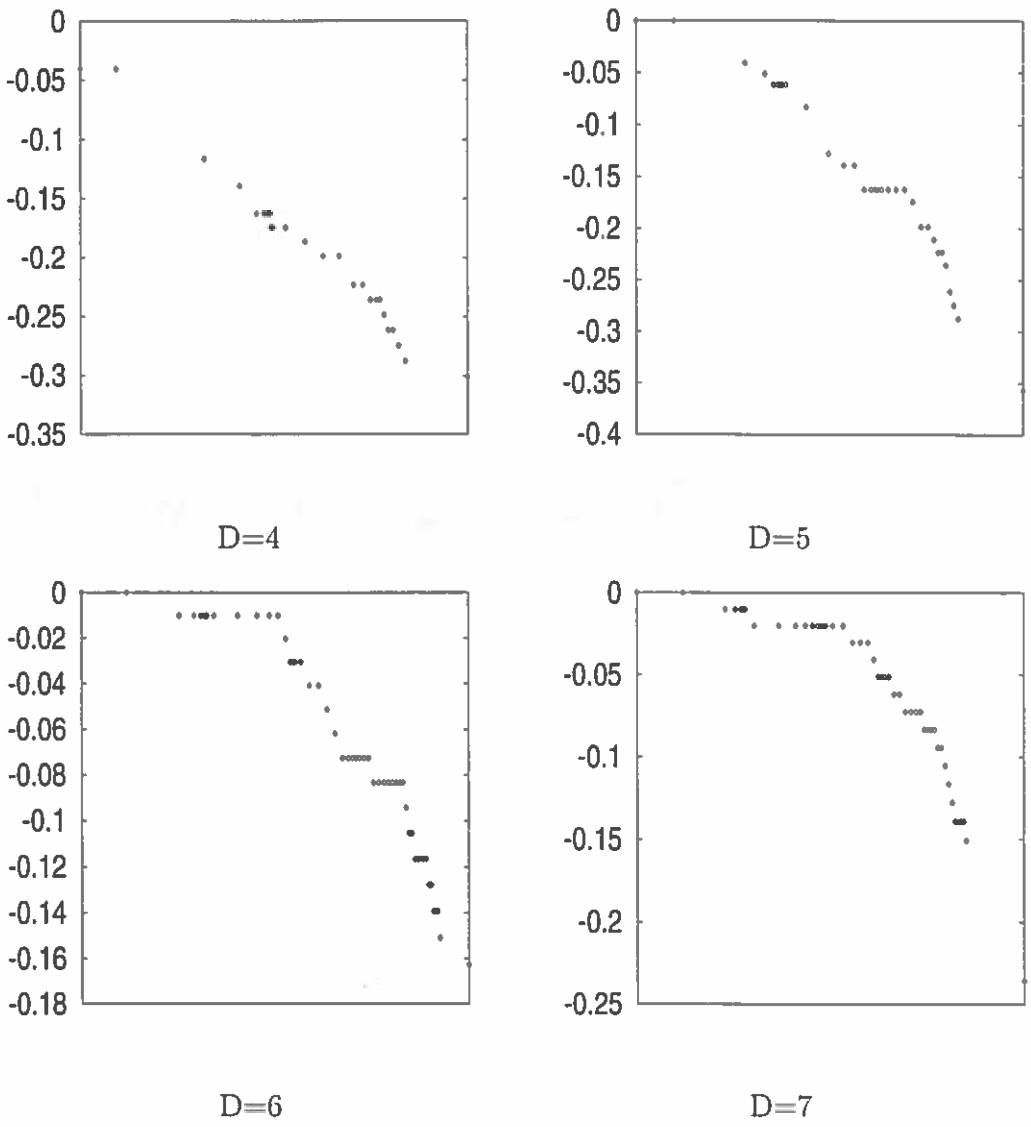


FIGURE 63. The y-axis shows the logarithm of the probability of failure. The x-axis shows the logarithm of the number of nodes explored. $r = 1.0$.

BIBLIOGRAPHY

[1] John Aitchison and James Alan Calvert Brown. *The Lognormal Distribution.* Cambridge University Press, Cambridge, MA, 1957.

[2] David Basin and Toby Walsh. Difference unification. In Ruzena Bajcsy, editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 116–122, August 1993.

[3] Eric Baum and Warren Smith. *Best Play for Imperfect Players and Game Tree Search.* NEC Research Institute, 4 Independence Way, Princeton, NJ 08540, March 1993. Draft Report.

[4] Hans Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12:23–40, 1979.

[5] Mark Boddy and Thomas Dean. Decision-theoretic deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–286, 1994.

[6] Kenneth Boese. *Models for Iterative Global Optimization.* PhD thesis, University of California, Los Angeles, Los Angeles, CA, 1996.

[7] John Bresina. Heuristic-biased stochastic sampling. In Kenneth Ford, editor, *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 271–278, August 1996.

[8] Peter Cheeseman, Bob Kanefsky, and William Taylor. Where the really hard problems are. In John Mylopoulos and Raymond Reiter, editors, *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 331–340, August 1991.

[9] Cheng-Chung Cheng and Stephen Smith. Generating feasible schedules under complex metric constraints. In Kenneth Ford, editor, *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1086–1091, August 1994.

[10] James Crawford. An approach to resource constrained project scheduling. In G.F. Luger, editor, *Proceedings of the 1996 Artificial Intelligence and Manufacturing Research Planning Workshop*, Albuquerque, NM, 1996.

[11] Jennifer Drapkin and Donald Perlis. Step-logics: An alternative approach to limited reasoning. In *Proceedings of the 7th European Conference on Artificial Intelligence*, pages 160–163. Pitman Publishing, July 1986.

[12] H. Fisher and G.L. Thompson. *Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules.* Prentice Hall, Englewood Cliffs, NJ, 1963.

[13] Michael Frank. Advances in decision-theoretic AI: Limited rationality and abstract search. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1994.

[14] Michael Garey and David Johnson. *Computers and Intractability: a guide to the theory of NP-Completeness.* W.H. Freeman and Co., San Francisco, CA, 1979.

[15] Ian Gent and Toby Walsh. Analysis of heuristic for number partitioning. *Computational Intelligence*, 14(3):430–451, 1998.

[16] Matthew Ginsberg. Do computers need common sense? In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 620–626. Morgan Kaufmann, November 1996.

[17] Matthew Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 584–589, August 1999.

[18] Matthew Ginsberg and Will Harvey. Iterative broadening. *Artificial Intelligence*, 55(2–3):363–383, 1992.

[19] Carla Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 431–437, July 1998.

[20] Carla Gomes, Bart Selman, Ken McAloon, and Carol Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In Reid Simmons, Manuela Veloso, and Stephen Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 208–213, June 1998.

[21] Irving Good. *Good Thinking: the foundations of probability and its applications.* University of Minnesota Press, Minneapolis, MN, 1983.

[22] Frederick Gravetter and Larry Wallnau. *Statistics for the Behavioral Sciences.* West Publishing Company, St. Paul, MN, 1985.

[23] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum path costs. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[24] Will Harvey. *Nonsystematic Backtracking Search.* PhD thesis, Stanford University, Stanford, CA, 1995.

[25] Will Harvey and Matthew Ginsberg. Limited discrepancy search. In Chris Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 607–613, August 1995.

[26] Bernardo Huberman and Tad Hogg. Phase transitions in artificial intelligence systems. *Artificial Intelligence*, 33:155–171, 1987.

[27] D. Jones and M. Beltramo. Solving partitioning problems with genetic algorithms. In R. Belew and L. Brooker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 442–449, 1991.

[28] Ari Jónsson. *Procedural Reasoning in Constraint Satisfaction.* PhD thesis, Stanford University, Stanford, CA, 1997.

[29] N. Karmakar and R.M. Karp. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, University of California, Berkeley, Berkeley, CA, 1982.

[30] S. Kirkpatrick, C.D. Gelatt, and M. P. Vecci. Optimization by simulated annealing. *Science*, 220:671–680, 1993.

[31] Scott Kirkpatrick and Bart Selman. Critical behavior in the satisfiability of random boolean expressions. *Science*, 264:1297–1301, 1994.

[32] Donald Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.

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

[34] Richard Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[35] Richard Korf. From approximate to optimal solutions: A case study of number partitioning. In Chris Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 266–272, August 1995.

[36] Richard Korf. Improved limited discrepancy search. In Kenneth Ford, editor, *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 286–291, August 1996.

[37] Richard Korf and David Chickering. Best-first minimax search. *Artificial Intelligence*, 84:299–337, 1996. Section 4.

[38] Pat Langley. Systematic and nonsystematic search strategies. In James Hendler, editor, *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 145–152. Morgan Kaufmann, June 1992.

[39] Stephen Lawrence. *Resource Constrained Project Scheduling: an Experimental Investigation of Heuristic Scheduling Techniques*. Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1984. (Supplement).

[40] David McAllester. Conspiracy numbers for minimax search. *Artificial Intelligence*, 35:287–310, 1988.

[41] Pedro Meseguer. Interleaved depth-first search. In Martha Pollack, editor, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1382–1387, August 1997.

[42] Pedro Meseguer and Toby Walsh. Interleaved and discrepancy based search. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 239–243. Pitman Publishing, August 1998.

[43] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In William Clancey, editor, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–362, July 1992.

[44] Andrew Parkes. Clustering at the phase transition. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 340–345. AAAI Press/The MIT Press, July 1997.

[45] Judea Pearl. *Heuristics*. Addison-Wesley Publishing Co., Reading, MA, 1984.

[46] Iu Petrov. *Variational Methods in Optimum Control Theory*. Academic Press, New York, NY, 1968.

[47] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes in C: the art of scientific computing*. Cambridge University Press, Cambridge, MA, second edition, 1992.

[48] Paul Purdom. Tree size by partial backtracking. *SIAM Journal on Computing*, 7(4):481–491, 1978.

[49] Wheeler Ruml, Thomas Ngo, Joe Marks, and Stuart Shieber. Easily searched encodings for number partitioning. *Journal of Optimization Theory and Applications*, 89(2):251–291, 1996.

[50] Stuart Russell and Peter Norvig. *Artificial Intelligence: a modern approach.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[51] Stuart Russell and Eric Wefald. *Do the Right Thing.* The MIT Press, Cambridge, MA, 1991.

[52] Norman Sadeh. Look-ahead techniques for micro-opportunistic job shop scheduling. Technical Report CMU-CS-91-102, Carnegie Mellon University, Pittsburgh, PA, 1992.

[53] U.K. Sarkar, P.P. Chakrabarti, S. Ghose, and S.C. De Sarkar. Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence*, 50(2):207–221, 1991.

[54] Herbert Simon and Joseph Kadane. Optimal problem-solving search: All-or-none solutions. *Artificial Intelligence*, 6:235–247, 1975.

[55] Barbara Smith. Phase transition and the mushy region in constraint satisfaction problems. In A.G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 100–104, 1994.

[56] Stephen Smith and Cheng-Chung Cheng. Slack-based heuristics for constraint satisfaction scheduling. In Kenneth Ford, editor, *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 139–144, August 1993.

[57] Lawrence Stone. *Theory of Optimal Search*, volume 118. Academic Press, New York, NY, 1975.

[58] Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278–285, 1993.

[59] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, UK, 1993.

[60] R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by local search. Technical Report COSOR 94–05, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994. Revised Version.

[61] Toby Walsh. Depth-bounded discrepancy search. In Martha Pollack, editor, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1388–1393, August 1997.

[62] Colin Williams and Tad Hogg. Extending deep structure. In Kenneth Ford, editor, *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 152–157, August 1993.

[63] Makoto Yokoo. Why adding more constraints makes a problem easier for hill-climbing algorithms: Analyzing landscapes of CSPs. In Gert Smolka, editor, *Third International Conference on Principles and Practice of Constraint Programming*, pages 356–370, October 1997.