NONSYSTEMATIC BACKTRACKING SEARCH

A DISSERTATION SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE AND THE COMMITTEE ON GRADUATE STUDIES OF STANFORD UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

By William D. Harvey March 1995

© Copyright 1995 by William D. Harvey All Rights Reserved I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Matthew L. Ginsberg (Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Michael R. Genesereth

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jean-Claude Latombe

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

Many practical problems in Artificial Intelligence have search trees that are too large to search exhaustively in the amount of time allowed. Systematic techniques such as chronological backtracking can be applied to these problems, but the order in which they examine nodes makes them unlikely to find a solution in the explored fraction of the space. Nonsystematic techniques have been proposed to alleviate the problem by searching nodes in a random order. A technique known as iterative sampling follows random paths from the root of the tree to the fringe, stopping if a path ends at a goal node. Although the nonsystematic techniques do not suffer from the problem of exploring nodes in a bad order, they do reconsider nodes they have already ruled out, a problem that is serious when the density of solutions in the tree is low. Unfortunately, for many practical problems the order of examing nodes matters and the density of solutions is low. Consequently, neither chronological backtracking nor iterative sampling has good average case performance.

We present a new search algorithm called bounded backtrack search that combines the merits of backtracking and iterative sampling. The algorithm backtracks chronologically until reaching a backtrack bound, whereupon it immediately backtracks to the root and restarts on a new random path. We show that the new algorithm does not suffer from the problems of the alternatives, and we derive theoretical conditions that guarantee better average case performance. We then present experimental results in job shop scheduling to show that the theoretical conditions and the expected performance hold for real problems.

Our analysis of search space properties of real problems shows that chronological backtracking does not use successor ordering heuristics effectively. The accuracy of the heuristics for early decisions is critical to the algorithm's efficiency, whereas the accuracy is relatively unimportant deep in the tree because the alternatives are considered quickly. Unfortunately, heuristics are typically least reliable for the early decisions—when they are most important.

We present a second new algorithm called *limited discrepancy search* that examines nodes in increasing order of "discrepancies," or points of disagreement with a problem's heuristics. We show that the algorithm has exceptional average case performance when the heuristics are accurate and reasonable performance when the heuristics are bad. We conjecture that the challenge of solving large problems will be met more quickly by advances in heuristics than by the evolution of brute force methods, and we argue that techniques like limited discrepancy search that rely on heuristics yet explore good alternatives when the heuristics fail will be the most practical in the future.

Acknowledgements

Thinking of the people who have helped with this work, either directly or indirectly, I am reminded that I have been very fortunate. My Mother, Dr. Diane Harvey, taught me that few challenges are impossible if you apply yourself with determination and confidence. It would be hard for anyone to grow up with these beliefs, though, without the love and support of his family. Thank you Mom, Dad, and Ben for giving me a chance.

I don't think anyone ever intends to be in graduate school a long time, but somehow it usually works out that way. Thank goodness I have friends who are going through the same ordeal. I think John Heymach, Frank Castro, and Marco Tortonese all know that we do not get to stop running Marathons together once we become doctors. I also need to thank my business partners Hugh Holbrook and Chris Hondl for shouldering our responsibilities when I was in Oregon.

Research is not an individual effort. Without the help of Andrew Baker, Ari Jonsson, Jimi Crawford, Don Geddis, Scott Roy, and Dave Smith I would be lost on some obscure problem that I could not solve by myself. I think all of us have enjoyed and learned from the productive discussions in the research groups at Stanford and CIRL.

When I entered graduate school, I was good at engineering solutions to practical problems but I was not good at distilling the scientific content of solutions that seemed to work. My advisor, Matt Ginsberg, had the patience both to convince me there was a difference and to show me how to be a scientist. I hope that the work in this thesis is evidence that he succeeded.

Contents

A	bstract			
A	ckno	wledge	ements	vi
1	Intr	oduct	ion	1
	1.1	Syster	maticity	2
	1.2	\mathbf{Backt}	racking	3
	1.3	Contr	ibutions	5
		1.3.1	Bounded Backtrack Search	5
		1.3.2	Limited Discrepancy Search	6
		1.3.3	Experimental Results	7
	1.4	Overv	riew	7
2	Bou	ınded	Backtrack Search	9
	2.1	Introd	luction	10
	2.2	Mista	kes	10
		2.2.1	When Mistakes Are Costly For Depth First Search	11
		2.2.2	When Mistakes Are Costly For Iterative Sampling	13
	2.3	The N	Mistake Probability	13
		2.3.1	If Goals Were Randomly Distributed	14
		2.3.2	As Goals are Distributed in Scheduling Problems	17
		2.3.3	The Effect of Heuristics	17
		2.3.4	Ignoring "Small" Mistakes	19
		2.3.5	Do Heuristics Affect Small Mistakes?	20

		2.3.6	What about Bigger Mistakes?	21	
	2.4	The E	Sounded Backtrack Search Algorithm	23	
		2.4.1	General Conditions	23	
		2.4.2	Algorithm	24	
		2.4.3	Example	26	
		2.4.4	Relationship to Depth First Search	27	
		2.4.5	Relationship to Iterative Sampling	27	
		2.4.6	Relationship to Depth First Search with Restarts	28	
	2.5	Avera	ge Case Cost Analysis	28	
		2.5.1	General Case	29	
	2.6	Analy	sis for Full Binary Trees	35	
		2.6.1	Full Binary Trees	35	
		2.6.2	Crossover Point	11	
		2.6.3	Choosing the Lookahead	15	
		2.6.4	If There Were Pruning	17	
	2.7	Concl	usion	17	
3	Lim	ited D	Discrepancy Search 4	8:	
	3.1	Introd	luction	19	
	3.2	3.2 Existing Strategies			
		3.2.1	Iterative Sampling	60	
		3.2.2	Backtracking	60	
		3.2.3	Other Nonsystematic Methods	5 1	
	3.3	Discre	$_{ m pancies}$	51	
	3.4	Upper	Bounds	53	
	3.5	The L	ikelihood of Finding a Solution	66	
		3.5.1	Wrong Turns	6	
		3.5.2	Estimating Heuristic Probability	8	
		3.5.3	Analysis of Early Iterations	8	
		3.5.4	Success Probability of Chronological Backtracking 6	35	
		3 5 5	Success Probability of Iterative Sampling 6	ւհ	

	3.6	Comp	arison with Existing Techniques	66		
	3.7	Variat	ions and Extensions	69		
		3.7.1	Variable Reordering	70		
		3.7.2	Using Different Heuristics	70		
		3.7.3	Combining LDS with Bounded Backtrack Search	70		
		3.7.4	Local Optimization Using LDS	72		
	3.8	Concl	usion	73		
4	Exp	erime	ntal Results	74		
	4.1	Introd	luction	75		
	4.2	Sched	uling as a CSP	76		
	4.3	Measu	ring Performance	77		
	4.4	Comp	arison with AI Techniques	78		
	4.5	Comp	arison with OR Techniques	80		
	4.6 Performance of the Five Strategies					
		4.6.1	Stochastic Iterative Broadening	82		
		4.6.2	Backtracking with Restarts	84		
		4.6.3	Iterative Sampling	85		
		4.6.4	Multiprobe: Bounded Backtrack Search	87		
		4.6.5	Limited Discrepancy Search	89		
	4.7	Sched	uling as SAT	89		
	4.8	Variat	ions	92		
		4.8.1	Weakening Value Order Heuristics	92		
		4.8.2	Weakening Variable Order Heuristics	95		
		4.8.3	Incremental Systematicity	95		
	4.9	Conclu	usion	98		
5	Con	clusio	$\mathbf{n}\mathbf{s}$	100		
\mathbf{A}	Rela	ated A	algorithms	104		
	A.1	Chron	ological Backtracking	105		
	1 2	Itarati	ive Sampling	106		

	A.3	Iterative Broadening	107		
В	A P	A Profile of Job Shop Scheduling			
	B.1	Introduction	108		
	B.2	The Search Space Without Heuristics	110		
	B.3	The Search Space With Heuristics	119		
	B. 4	Conclusions	127		
	B. 5	Earlier Graphs	127		
\mathbf{C}	A P	rofile of Scheduling Problems as SAT	128		
D	A P	rofile of Random 3SAT	131		
	D.1	Introduction	131		
	D.2	Characterizing the Search Tree	132		
	D.3	Getting to the Good Part	133		
	D.4	Grappling with the Difficult Part	134		
	D.5	Other Graphs, Other Problems	135		
	D.6	Conclusion	135		
\mathbf{E}	Col	ecting Profile Data	143		
	E.1	Node Classes	143		
	E.2	Traversing the Search Tree	144		
	E.3	Incorporating Heuristics	145		
F	Imp	lementation Considerations	147		
	F.1	Multiprobe	147		
	F.2	Weighted Random Selection of Values	149		
	F.3	Walk Start: Restarting with Different Variables	150		
	F.4	Application of CSP Heuristics to Scheduling	151		
Bi	bliog	raphy	155		

List of Figures

2.1	Nodes C and D are mistakes	10
2.2	Mistakes are costly when they have large subtrees below them	12
2.3	Small mistakes are costly for iterative sampling	14
2.4	What m_k would look like if goals were distributed randomly	16
2.5	In real problems, m_k is often fairly constant for all k	18
2.6	Heuristics have unpredictable effects on m_k	19
2.7	The effective mistake probability, m_k' , with one node lookahead	20
2.8	This is not a misprint: m'_k , with one node lookahead and heuristics	20
2.9	Ratio of \overline{m}' with lookahead to \overline{m} without	22
2.10	Depth first search algorithm	24
2.11	Bounded backtrack search algorithm	25
2.12	BBS-Probe fails to discover the goal node $G.$	26
2.13	Calculating average case cost (no lookahead)	29
2.14	Theoretical crossover point for full binary trees of height 10	42
2.15	Optimal lookahead for full binary trees of height 10	43
2.16	Theoretical crossover point for full binary trees of height 30	43
2.17	Optimal lookahead for full binary trees of height 30	44
2.18	Performance of BBS for various fixed lookahead amounts	46
3.1	Limited Discrepancy Search	53
3.2	Execution trace of LDS	54
3.3	The four possibilities for a node and its children	57
3.4	Probes 0 through 4 of iteration 1	59
3.5	For one discrepancy, $a_{3,3}$ is a component of $a_{4,0}$	64

3.6	A problem of height 30	6
3.7	A problem of height 100	69
3.8	Limited discrepancy search with bounded backtrack	71
4.1	Iterative sampling and chronological backtracking are not competitive.	79
4.2	Front runners	80
4.3	OR results from 1994 survey on the same benchmark	81
4.4	Iterative broadening with cutoff c	83
4.5	Backtracking with restarts, timeout T	8
4.6	Variations of iterative sampling	86
4.7	Adding bounded backtrack to isamp is a major improvement	8
4.8	Any small lookahead value works—some slightly better than others	88
4.9	Bounded backtrack improves LDS	90
4.10	Multiprobe and WSAT	91
4.11	Weighted random selection can be worse than the alternatives	93
4.12	Weighted random selection can be better than the alternatives	94
4.13	Randomize variable order—but not too much	96
4.14	Incremental systematicity improves BBS	97
4.15	Incremental systematicity with iterative sampling	98
A.1	Depth first search with a time limit	105
A.2	Iterative sampling.	106
A.3	Iterative broadening.	107
B.1	The numbering scheme for Sadeh's problems	109
B.2	Distribution of nodes over depth range for Sadeh (rand) no. 1	113
B.3	$m_k'(0)$ and $m_k'(1)$ for Sadeh (rand) no. 1	113
B.4	Ratio of \overline{m}' to \overline{m} for Sadeh (rand) no. 1	114
B. 5	h(l) for Sadeh (rand) no. 1, magnified on right	114
B.6	f(k,0) for Sadeh (rand) no. 1	115
B.7	Cost of bad subtrees by height l for Sadeh (rand) no. 1	115
B.8	Cost, height of bad subtrees at depth k for Sadeh (rand) no. 1	116

B.9 $m'_k(0)$ and $m'_k(1)$ for Sadeh (rand) no. 11
B.10 $m_k'(0)$ and $m_k'(1)$ for Sadeh (rand) no. 21
B.11 $m_k'(0)$ and $m_k'(1)$ for Sadeh (rand) no. 31
B.12 $m_k'(0)$ and $m_k'(1)$ for Sadeh (rand) no. 41
B.13 $m'_k(0)$ and $m'_k(1)$ for Sadeh (rand) no. 51
B.14 Distribution of nodes over depth range for Sadeh (heur.) no. 1 12
B.15 $m'_k(0)$ and $m'_k(1)$ for Sadeh (heur.) no. 1
B.16 Cost of bad subtrees by height l for Sadeh (heur.) no. 1
B.17 $h(l)$ for Sadeh (heur.) no. 1, magnified on right
B.18 Ratio of \overline{m}' to \overline{m} for Sadeh (heur.) no. 1
B.19 $f(k,0)$ for Sadeh (heur.) no. 1
B.20 $m_k'(0)$ and $m_k'(1)$ for Sadeh (heur.) no. 11
B.21 $m_k'(0)$ and $m_k'(1)$ for Sadeh (heur.) no. 21
B.22 $m'_k(0)$ and $m'_k(1)$ for Sadeh (heur.) no. 31
B.23 $m'_k(0)$ and $m'_k(1)$ for Sadeh (heur.) no. 41
B.24 $m_k'(0)$ and $m_k'(1)$ for Sadeh (heur.) no. 51
C.1 Cost of bad subtrees by height for SadehSat (w/heuristics) no. 10 12
C.2 Cost of bad subtrees by height for SadehSat (w/heuristics) no. 15 13
D.1 Distribution of nodes over depth range for Random 3SAT no. 6 13
D.2 $m'_k(0)$ and $m'_k(1)$ for Random 3SAT no. 6
D.3 $h(l)$ for Random 3SAT no. 6, magnified on right
D.4 Cost of bad subtrees by height l for Random 3SAT no. 6 13
D.5 Ratio of \overline{m}' to \overline{m} for Random 3SAT no. 6
D.6 $f(k,0)$ for Random 3SAT no. 6
D.7 Cost, height of bad subtrees at depth k for Random 3SAT no. 8 14
D.8 Distribution of nodes over depth range for Random 3SAT no. 6 14
D.9 $m'_k(0)$ and $m'_k(1)$ for Random 3SAT no. 8
D.10 Cost of bad subtrees by height l for Random 3SAT no. 8
D 11 Cost, height of had subtrees at depth k for Random 3SAT no. 8

E.1	Every interior node is in one of three classes	143
F.1	Multiprobe.	148
F.2	Function to "randomize" value order	149
F.3	Function for restarting on different variables	150
F.4	There are 34 ways for $a < b$, only 3 for $b < a$	152
F.5	The area of possible configurations	154

Chapter 1

Introduction

1.1 Systematicity

Backtracking [9, 28, 50] is a simple and intuitive algorithm for systematically exploring a set of alternatives in search of one or all solutions to a problem. The algorithm is said to be systematic because it considers all of the alternatives in the set and it never considers any one of them more than once [40]. Searching a finite set of alternatives systematically has the advantage that it is guaranteed to terminate with a solution or a proof that no solution exists. Unfortunately, though, for many problems of practical interest the set of alternatives is too large to search exhaustively in the amount of time the situation allows. Under these circumstances, the importance of systematicity comes into question.

If every alternative had an equal chance of being a solution to the problem, then the optimal search protocol for finding a solution in a bounded amount of time would be the one that considered or ruled out alternatives the fastest. However, for most real world problems the alternatives do not have an equal chance of being solutions, and their success or failure is anything but independent. It is usually possible for two similar alternatives to fail for the same reason, in which case the failure of one affects the conditional probability that the other succeeds. In light of this observation, the optimal search protocol is not necessarily the fastest to consider new alternatives. Some alternatives are better than others.

Nonsystematic backtracking search strategies are variations of the backtracking algorithm that sacrifice the guarantees of completeness and nonredundancy in order to explore better alternatives more quickly. We present two new algorithms, bounded backtrack search and limited discrepancy search. Both are based on a realistic model of search space properties from which the notion of a "better" alternative is derived. Bounded backtrack search examines alternatives that have little in common with other alternatives known to be failures. Limited discrepancy search examines alternatives that have the fewest "discrepancies," or points of disagreement with a problem's heuristics.

1.2 Backtracking

Constraint satisfaction problems [30, 37, 41] provide a useful framework for research on backtracking techniques. The problem is to find a complete assignment for a set of variables that satisfies a given set of constraints limiting specific combinations of variable assignments. Backtracking explores the set of alternatives, or complete variable assignments, by picking a variable and branching sequentially on its possible values. For each value, it explores all possible assignments to the remaining variables, extending the partial assignment at each level of recursion.

Much of the research on backtracking in constraint satisfaction aims to avoid unnecessary search by proving or remembering that certain partial assignments are inconsistent with the constraints of the problem. Once a partial assignment is known to be inconsistent, a backtracking algorithm can ignore all extensions of the assignment without danger of missing a solution. Research on this topic can be divided, with some overlap, into the categories of consistency preprocessing, constraint propagation, intelligent backtracking and dependency directed backtracking.

- By reasoning about a problem's constraints, consistency processing techniques [16, 19, 34, 51] eliminate values from the domains of variables without search. The effectiveness of the approach was first demonstrated in scene recognition problems [51] by dramatically reducing the number of possible interpretations for a line drawing of blocks with shadows. With a sufficient degree of consistency processing, a problem can be solved without search altogether [19]. Unfortunately, the consistency techniques can be as expensive as the search itself, depending on the domain and the structure of the constraints [30].
- Consistency reasoning can also be applied dynamically after each variable instantiation in a backtracking search algorithm. Known as constraint propagation [21, 30, 34, 37], the procedures eliminate values that are inconsistent with the constraints and the current partial assignment. The efficiency concerns of consistency processing are more critical when the procedures are applied dynamically because they are invoked many times. So called "hybrid systems" [25, 41]

combining a small degree of constraint checking with intelligent backtracking appear to the most effective.

- After detecting that a partial assignment is inconsistent with the constraints of the problem, chronological backtracking returns to the most recent variable assignment and tries an alternative value, even if the former value was in no way responsible for the failure. Intelligent backtracking seeks to avoid this "thrashing" behavior by identifying backtrack points that are responsible for the failure and returning directly to them [7, 21]. A simple intelligent backtracking scheme called backjumping [21, 22, 41] is widely used as an improvement over chronological backtracking.
- Even with perfectly chosen backtrack points, backtracking can still consider partial assignments containing specific combinations of variable assignments previously found to be inconsistent. This redundancy can be eliminated by remembering the causes of each failure when it is first detected. Subsequently, any partial assignment containing a known cause of a failure can be pruned. Dependency directed backtracking [46] and truth maintenance systems [14] attempt to store all causes as they are discovered. Storage and indexing are principal concerns with this approach. Most practical techniques restrict the set of stored causes at a cost of expanding some nodes that full dependency directed backtracking would not have to expand [15, 21, 22].

While considerable research has gone into keeping backtrack search from exploring provably bad alternatives, relatively little has gone into guiding it to explore plausibly good alternatives. Most related research focuses on choosing the order for variable instantiation in a manner that minimizes the size of the overall search tree. In its simplest form, "search rearrangement" always instantiates next the variable that has the fewest possible remaining values [6]. Even the basic principles of search rearrangement have been shown to dramatically increase the search efficiency [25, 53]. More complicated methods depend on models that are less likely to be apply to real world problems [42].

Search rearrangement methods still achieve their efficiency by reducing the size of the search tree rather than by exploring more promising areas of the tree early. When the objective is to find a solution in a bounded amount of time, the order of examining alternatives can be more important than the size of the search tree or the amount of redundancy. Our own earlier work on iterative broadening [24] introduced the notion of a "bad" node and developed a probabilistic model for the likelihood that a node considered by the search would succeed. Langley [32] adopted the model to compare the expected performance of chronological backtracking with a nonsystematic technique called iterative sampling. Under plausible conditions, iterative sampling was shown to have better average case performance than traditional backtracking. But would the conditions hold for real world problems? A series of scheduling experiments [11] appeared to demonstrate that they would. Nonsystematic techniques were indeed legitimate alternatives to backtracking for large search spaces.

The research in this thesis comes from the observation that backtracking techniques and nonsystematic techniques are compatible. By combining the merits of both, we have been able to develop new algorithms that have significantly better average case performance than both chronological backtracking and iterative sampling. Since chronological backtracking [50], iterative sampling [32], and iterative broadening [24] are all closely related to the new algorithms, we will refer them frequently. For reference, we have included these algorithms in Appendix A.

1.3 Contributions

1.3.1 Bounded Backtrack Search

Bounded backtrack search is a new algorithm that combines elements of backtracking and iterative sampling. In some ways, backtracking and iterative sampling are very much alike. Like backtracking search, iterative sampling extends a partial assignment until reaching a consistent total assignment or detecting an inconsistency with the constraints of the problem. Upon detecting an inconsistency, though, iterative sampling immediately backtracks all the way to the first assignment. In contrast,

chronological backtracking algorithms backtrack only to the most recent assignment. Bounded backtrack search is a nonsystematic derivative of chronological backtracking that incorporates a bound on the length of a sequence of backtracks. The algorithm backtracks chronologically until reaching the backtrack bound, at which point it is forced to backtrack to the first assignment and restart. We show theoretically that for a tree of height d the optimal backtrack bound lies properly between iterative sampling's bound of zero and chronological backtracking's bound of d. We present bounded backtrack search as a simple modification to the backtracking algorithm that dramatically decreases the expected time to find a solution.

1.3.2 Limited Discrepancy Search

For many large problems of practical interest, carefully tuned value ordering heuristics guide the search toward regions of the space that are likely to contain solutions. For some problems, the heuristics often lead directly to a solution—but not always. Limited discrepancy search addresses the problem of what to do when the heuristics fail. We show theoretically and experimentally that chronological backtracking is not an effective use of accurate value ordering heuristics. A good heuristic may make mistakes very rarely, but if it ever makes a mistake high in the search tree, traditional backtracking strategies will not recover to consider the alternative in the time allowed.

Our intuition is that when the heuristic fails, it probably would have succeeded if it had only not made a small number of "wrong turns" along the way. For a tree of height d, there are only d ways the heuristic could make a single wrong turn, and only $\binom{d}{2}$ ways it could make two. If the number of wrong turns is small, they can be discovered by systematically searching all paths that differ from the heuristic path in at most a small number of decision points, or "discrepancies." Limited discrepancy search is a backtracking algorithm that searches the nodes of the tree in increasing order of such discrepancies. We derive an upper bound for exhaustively searching the space with a limited number of discrepancies, and we show formally that limited discrepancy search has a much greater chance of finding a solution in a limited amount of time than the alternative approaches.

1.3.3 Experimental Results

Job shop scheduling is both a problem of significant practical importance and a representative of a large class of problems on an important frontier of tractability. Studied in both the Operations Research and Artificial Intelligence communities, job shop scheduling is an ideal domain for evaluating the performance of the algorithms we are introducing.

We present a comprehensive evaluation of the algorithms based on an established benchmark of large scheduling problems. We compare our results with AI search techniques and dedicated OR scheduling programs. The results confirm our theoretical expectations about the performance of traditional backtracking techniques, and show that while iterative sampling performs well on easy scheduling problems [11], its performance on larger problems is less than competitive. We test a variety of non-systematic backtracking techniques and find categorically that they all outperform chronological backtracking and iterative sampling by significant margins. Of the better algorithms, a combination of bounded backtrack with limited discrepancy is the algorithm of choice. We argue on theoretical grounds that the algorithm will remain superior as technology and science meet the challenge of scaling up from research problems to the problems of the real world.

1.4 Overview

The body of the thesis is contained in three chapters that we recommend reading in sequence. In Chapter 2 we introduce the search space properties that form the basis of our theoretical analysis. We go on to describe the shortcomings of various existing search techniques and show how bounded backtrack search avoids the problems. We conclude with a formal analysis of the expected time to find a solution with bounded backtrack search, comparing the theoretical performance with that of iterative sampling and chronological backtracking. In Chapter 3, we extend our analysis to tree search with heuristics. We present the limited discrepancy search algorithm and analyze its expected performance. We conclude with a discussion of variations and extensions that are likely to be useful in practice. Finally in Chapter 4 we present

our experimental results. We first compare the search techniques experimentally and then evaluate them individually, testing variations and adjustments to their parameters. A number of these individual experiments have unexpected results that have implications for many search algorithms. We explore these results in the later part of the chapter and suggest various topics for further investigation.

For reference, the first appendix contains psuedocode descriptions of the related search techniques. In Appendices B and C we attempt to profile the search spaces of job shop scheduling problems to show that the theoretical properties of the search spaces that we argued would exist in real world problems actually do. For comparison, we profile the search space of random 3SAT problems in Appendix D. In the following appendix we describe our method of profiling. In Appendix F we give the *Multiprobe* algorithm that implements a variety of nonsystematic backtracking strategies.

Chapter 2

Bounded Backtrack Search

2.1 Introduction

Many problems in AI are formulated as tree search problems. The search space consists of the nodes of tree, some subset of which are recognized as goal nodes representing solutions. While this formulation defines what nodes a search algorithm might have to examine, it often does not provide precise information about how many nodes a search algorithm would be expected to examine before finding a solution. One reason is, of course, that the performance of different search procedures depends on a wide variety of factors other than the size of the search space or even the density of solutions. When these factors are significant, the standard average case cost analyses of the search procedures do not apply.

2.2 Mistakes

One factor that is important to backtracking search algorithms is the notion of a mistake. We define a bad node as a node that does not have any goals in the subtree below it. A good node is any node that is not bad. A mistake is a bad node whose parent is a good node. In the example shown in Figure 2.1, good nodes are drawn with a solid dot, and the bad nodes are drawn with an "X". The only goal node is node E.

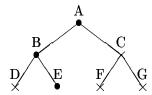


Figure 2.1: Nodes C and D are mistakes.

Good and bad nodes were introduced by Ginsberg and Harvey [24] in the context of iterative broadening. We argued that for a tree of constant branching factor,

it is reasonable to assume, in the absence of other information, that the number of good children of a good node remains constant for all depths. Under such conditions, iterative broadening was shown to outperform depth first search by significant margins.

Without committing to a constant number of good children per good node, we define the mistake probability m_k as the likelihood that a randomly selected child of a good node at depth k is a mistake node. If, for example, every good node in a tree of branching factor b had s good children, then the chance of selecting a mistake child at random from the children of a good node at any depth would be $\frac{b-s}{b}$. We will see that the mistake probability is a search space property that significantly affects the average case performance of backtracking search procedures.

In the remainder of this section, we will attempt to demonstrate how mistakes relate to backtracking. Then in Section 2.3, we will investigate the mistake probability as a property of the search space. In the following Section 2.4 we will present a bounded backtrack search algorithm that has the potential to exploit this property to achieve better performance than alternative backtracking strategies. In Section 2.5 we will begin a formal analysis of the search algorithm, which will carry through to the end of the chapter. The experimental results comparing bounded backtrack search to other procedures are given in Chapter 4.

2.2.1 When Mistakes Are Costly For Depth First Search

One reason for the effectiveness of iterative broadening is that it reduces the cost of making a mistake early in the search tree. A mistake that leads to a large subtree that has no solutions is a serious threat to depth first search. If depth first search has the misfortune of investigating the mistake node before its good sibling, the algorithm commits to exploring the entire subtree below the mistake before exploring any other portions of the search tree. For many practical problems of interest, the size of the subtree may be sufficiently large that the algorithm cannot recover in the amount of time it is allowed to run.

Consider the problem shown in Figure 2.2. This tree has a height of just three, but you could easily imagine a large tree with the same structure. If depth first search

explores the nodes left to right, the first commitment could be fatal mistake.

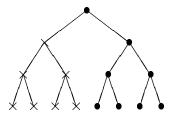


Figure 2.2: Mistakes are costly when they have large subtrees below them.

Two things are particularly alarming about this example. The first is that by many standards, the problem is extremely easy: Half the fringe nodes are goal nodes—There is only one possible mistake in the entire search space. The fact that such a simple problem could actually be unsolvable by depth first search within a reasonable amount of time is an indication that the algorithm may not be well suited to the task.

The second alarming fact is that problems of this structure are not obscure pathological cases; they are perfectly natural. For example, a few months ago I needed to travel to a wedding in Laredo, Texas on a busy holiday weekend. I first made the decision to fly into Laredo instead of driving. After many exasperating phone calls with travel agents, I gave up and decided to fly to San Antonio and drive to Laredo from there. Once I told the travel agent that I had this flexibility, booking a flight was easy.

It turned out that there was only one airline that flew into Laredo that weekend, and that airline was completely booked. All of my attempts to fly to Laredo on various connecting flights failed for the same reason: they had to connect with American Eagle, which was booked. The point is that what enabled me to solve this problem was that I got exasperated and gave up on flying to Laredo long before exhausting all the ways that it might have been possible. If I had committed to trying *all* possible connecting flights before considering driving, I would still be on the phone.

At the time, my commitment to flying into Laredo seemed like a natural first step in coming up with a plan. In many AI problems, the early commitments in the search are considerably more arbitrary, which makes the threat of early mistakes even more serious.

2.2.2 When Mistakes Are Costly For Iterative Sampling

One proposed approach for dealing with the problem of early mistakes is to abandon any vestige of systematicity. Iterative sampling [32] employs the strategy of boldly traversing a random path down from the root of the tree directly to a fringe node. If the fringe node is a goal, it is returned as a solution. Otherwise, the algorithm restarts on another random path from the root of the tree, repeating until it finds a solution.

This algorithm has the advantage that for a balanced search tree it doesn't matter how goal nodes are distributed. In the example of Figure 2.2 or the above Laredo predicament, iterative sampling would probably find a solution in about two tries.

Unfortunately, iterative sampling is not without its own weaknesses. Consider the tree shown in Figure 2.3. As with the last example, this tree is small but you could imagine a much larger tree with the same structure. If you search this tree with depth first search, you will find a solution in a short amount of time even if you make a mistake at every possible opportunity. The size of the tree is only 2d + 1 nodes, so any systematic procedure is guaranteed to find a solution in as many nodes. Iterative sampling, however, does not provide this guarantee. Consider how lucky iterative sampling has to be to find a solution on any given probe. It has to not make a mistake d times in a row. Thus the chance of finding a solution is $1/2^d$. Iterative sampling will search, on average, about 2^d nodes before finding a solution. Like the problems that were difficult for depth first search, problems that share this structure are easy to imagine and likely to occur frequently in practice.

2.3 The Mistake Probability

The examples of the previous section suggest that mistakes are an important factor in the performance of various search algorithms. We would like to establish that the

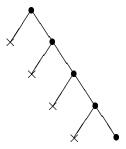


Figure 2.3: Small mistakes are costly for iterative sampling.

mistake probability is actually a meaningful property of the search space. For clarity, we will continue to use binary trees for examples, but the arguments apply to any branching factor.

The analysis of iterative broadening [24] was based on the assumption that the mistake probability is the same at all depths. Let us first consider an example where this assumption is not true. Then we will examine some representatives of "real world problems" to test whether the assumption is reasonable in practice.

2.3.1 If Goals Were Randomly Distributed

Say that someone gave you a problem represented as a full binary tree of height d. This person tells you that one fourth of the nodes on the fringe are goal nodes. We make the assumption that the search space has a mistake probability m_k that is approximately the same for all depths k. We will denote the constant mistake probability m.¹ We can calculate m from the knowledge that one quarter of the fringe nodes are goals, as follows:

Since there are 2^d nodes on the fringe, there must be $\frac{1}{4} \times 2^d$ goal nodes. The branching factor is constant, so the chance of selecting a bad node from the children of a randomly selected good parent at depth k is equal to the chance of selecting a bad node from the collection of all nodes with good parents at depth k. Letting b_{k+1}

¹Since the tree has only one root node and at least one of its two children must be good, m_0 must be either zero or one half. Thus m_k cannot be exactly m for all k in this tree.

and g_{k+1} be the number of bad and good nodes with good parents at k,

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

Each good parent has exactly two children, so,

$$b_{k+1} + g_{k+1} = 2g_k (2.2)$$

Combining equations 2.1 and 2.2 confirms that the number of good nodes grows at each depth k by $2(1-m_k)$. Assuming $m_k \approx m$,

$$2^d (1-m)^d \approx \frac{1}{4} 2^d$$
 $1-m \approx \left(\frac{1}{4}\right)^{1/d}$
 $m \approx 1 - \left(\frac{1}{4}\right)^{1/d}$

We can make a few observations about m before evaluating this expression. The mistake probability m_k depends only on good nodes at depth k. By definition, a good node must have at least one good successor. In a binary tree, at least one of the good node's two children must be good. Consequently, the value of m_k cannot ever be greater than 1/2 for a binary tree.² In fact, if m_k is 1/2 for all depths, then the tree must have a single goal node, because at depth zero it has a single good node, and at any depth k+1 it has exactly one half, times two, times the number of good nodes it has at depth k.

If the given tree had height twenty, we would calculate that m is about 0.07. Having calculated m, we could proceed to analyze the expected performance of various search algorithms for this space, as we will do at some length later on in this chapter. Unfortunately, this analysis is only as good as the original assumption that the mistake probability is the same for all depths. Let us say the person who generated the problem secretly decided which nodes on the fringe were goal nodes by flipping coins. The real mistake probability would then vary quite dramatically with depth.

²In general, it cannot be greater than $\frac{b-1}{b}$ for a tree of branching factor b.

The graph of the real m_k is shown in Figure 2.4. The calculated mistake probability of 0.07 is a bad representative of the actual mistake probability at just about every depth!³

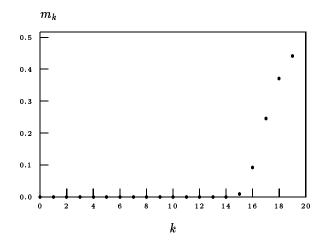


Figure 2.4: What m_k would look like if goals were distributed randomly.

When goal nodes are randomly distributed, the assumption that the mistake probability is the same at all depths is clearly wrong. For most practical search problems, however, there is no compelling reason to believe the goals are randomly distributed. Often we have no real idea how the goal nodes are distributed. In the absence of this information, we might assume that the goals are randomly distributed, or we might assume that the mistake probability is constant. The intuitive basis for the latter assumption is usually stronger, but it would be comforting to see some experimental

³The mistake probability is calculated as follows: a node at depth k can be classified by whether its children are bad or good. The node's left and right child could be good and bad (case 1), bad and good (case 2), good and good (case 3), or bad and bad (case 4). The height of a child's subtree is d-k-1. The number of fringe nodes in a child's subtree is 2^{d-k-1} . For a given probability p that a fringe node is a goal, the probability p that a child's subtree has no goals, implying the child is bad, is $(1-p)^{2^{d-k-1}}$; the probability that a child is good is 1-x. The mistake probability is the ratio of bad nodes with good parents at depth p to all nodes, good or bad, with good parents at p. There is one such bad node with a good parent at depth p for each case 1 or case 2 node at p. There are two nodes (bad or good) with good parents at depth p for each case 1, case 2, or case 3 node. There are zero bad or good nodes with good parents at depth p for each case 4 node. The ratio is thus one half times the probability of (case 1 or case 2) divided by the probability of (case 1 or case 2 or case 3), which is $\frac{p}{2}(1-p)$.

evidence in its support. In the next few sections, we will present some results drawn from practical scheduling problems.

2.3.2 As Goals are Distributed in Scheduling Problems

Job shop scheduling [26] is a good example of a practical problem for which the intuitions supporting a constant mistake probability are fairly strong. Job shop scheduling is the problem of finding a schedule, or ordering, for a set of operations on a number of machines that satisfies a given set of time and ordering constraints.⁴ The problem can be formulated as a search tree of binary decisions, where each decision is an ordering decision between two operations that contend for a single resource: to prevent a resource conflict, either operation A must complete before operation B begins, or vice versa.

We could expect that at any point in the search a "wrong" ordering decision could be made, committing to a partial schedule that had no feasible extensions. Such mistakes might go undetected until deeper in the search (yielding a tree like the one in Figure 2.2) or they might be discovered early by a pruning mechanism (yielding a tree as in Figure 2.3). When mistakes are detected is an important consideration that we will return to later. For the moment, though, we are only concerned with the plausibility of making a mistake at any depth in the search tree.

Figure 2.5 shows the mistake probability for a range of depths taken from the end of a prototypical job shop scheduling problem [43, 45]. Appendix E explains how this data was collected and gives a number of additional examples. What is interesting to note here is that although the mistake probability is not constant, it isn't following the curve of randomly distributed nodes either. Of the two assumptions, the constant mistake probability is a better explanation of the data.

2.3.3 The Effect of Heuristics

In search problems formulated as CSPs, heuristics are generally used to select which variable to instantiate next and which value to assign first [6]. When the CSPs are

⁴Job shop scheduling is discussed at length in Chapter 4.

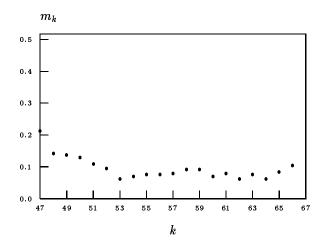


Figure 2.5: In real problems, m_k is often fairly constant for all k.

formulated as tree search problems, the value selection heuristics become successor orderings, and the variable selection heuristics determine the arrangement of decisions in the search tree.

Since successor ordering is not critical to understanding or analyzing the search space properties we hope to capitalize on in this chapter, we will discuss value selection heuristics only in passing. Variable selection heuristics, however, might affect our assumptions about the search space, since the order of decisions in the search tree is relevant to whether the mistake probability is the same at all depths.

It is not intuitively obvious how heuristics would affect the mistake probability in general, and it will almost certainly depend on the individual heuristics and problems. Job shop scheduling problems have very powerful variable and value selection heuristics that we thought might be illustrative. The same scheduling problem searched for Figure 2.5 with random variable selection was searched using heuristic variable selection for Figure 2.6. The effect would certainly have been difficult to predict.

The bizarre effect in Figure 2.6 appears to be idiosyncratic. We tested a variety of scheduling problems using heuristics (see Appendix B), and all of the curves were idiosyncratic. There appears to be no intuitive or experimental way to predict the effect of heuristics on the mistake probability. The assumption that the mistake probability is constant, while not true for any particular problem, remains a reasonable a

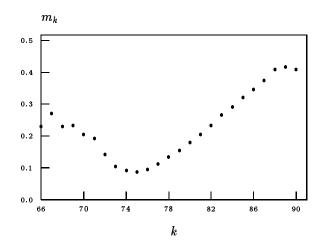


Figure 2.6: Heuristics have unpredictable effects on m_k .

priori assumption about the search space.

2.3.4 Ignoring "Small" Mistakes

Looking back to Figure 2.3, we might wonder whether the mistake probability would be affected significantly by "looking ahead" one node. We could define the effective mistake probability for lookahead l to be the probability of selecting a bad child among children of a good parent, considering only children whose subtrees have height at least l. For a lookahead of one, the effective mistake probability calculation simply ignores dead end nodes, whose subtrees have height zero. Thus by this definition, the effective mistake probability of the problem in Figure 2.3 is zero at all depths for any $l \geq 1$.

Using the term "lookahead" may be a little misleading here because we are still talking about properties of a search space rather than parameters of a search procedure. The definition of effective mistake probability depends strictly on the search space. We will get to search procedures in Section 2.4, once we have established the search space properties we are hoping to exploit.

With a one node lookahead, the effective mistake probability of Figure 2.5 drops to that shown in Figure 2.7. The drop is significant, confirming our intuition that in practical problems pruning mechanisms may detect many mistakes relatively early.

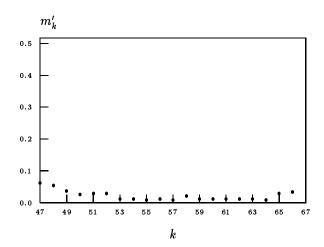


Figure 2.7: The effective mistake probability, m'_k , with one node lookahead.

2.3.5 Do Heuristics Affect Small Mistakes?

Figure 2.7 shows the effective mistake probability without heuristics. We saw in Section 2.3.3 that heuristics can have a large and unpredictable effect on the mistake probability at different depths. Do the effects become more understandable if we ignore small mistakes? Figure 2.8 shows the same problem as the previous figures, this time searched with heuristics and a lookahead of one.

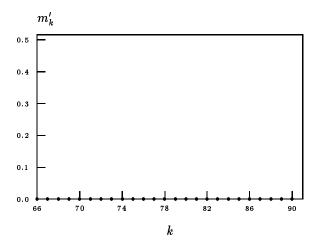


Figure 2.8: This is not a misprint: m'_k , with one node lookahead and heuristics.

The graph in Figure 2.8 is remarkable! The effective mistake probability is wiped out by the heuristics. Although this is just one problem, all of the other scheduling problems we tested had similar results (see Appendix B).

As we discuss in Appendix E, the data in these graphs only reflects the bottom depths of the search tree. The depths of the recorded range are shown marking the horizontal axis. Since these problems were not all trivial to solve with depth first search even with the heuristics, it must not be the case that the effective mistake probability is actually zero at all depths. Somewhere high in the search tree must be a possibility of making a serious mistake. This makes sense. Early in the search, the scheduling heuristics are less informed. Decisions are more arbitrary. We might even expect the mistake probability to begin high and gradually decrease deeper in the search tree. Our expectation would contrast sharply with the steep increase of the mistake probability for randomly distributed goals.

2.3.6 What about Bigger Mistakes?

The effect of a one node lookahead is so dramatic in the job shop scheduling problems that we would have been foolish not to consider trying deeper lookahead. In Figure 2.9, we graph the ratio of the effective mistake probability with lookahead to the mistake probability without lookahead, for a range of lookahead amounts on the scheduling problem shown in earlier figures.⁵ This problem was solved without heuristics. Since we have already seen that heuristics reduce the mistake probability to zero with a lookahead of just one (Figure 2.8), the additional lookahead is of no advantage with these heuristics.

In Figure 2.9, the value \overline{m}' is the average of m'_k for a given lookahead l over all depths in the examined portion of the search space (See Appendix E). Since the mistake probability m'_k remains roughly constant over depth, the average value is a reasonable representative.⁶ According to the graph, there is a dramatic reduction in the effective mistake probability between a lookahead of zero and one (the ratio steps

⁵The $\overline{m}'(l)$ graphs for most of Sadeh's problems look pretty much alike. See Appendix B for graphs of the other problems.

⁶The geometric mean is more appropriate for calculations involving the mistake probability, but here we just want to show how the mistake probability varies with lookahead.

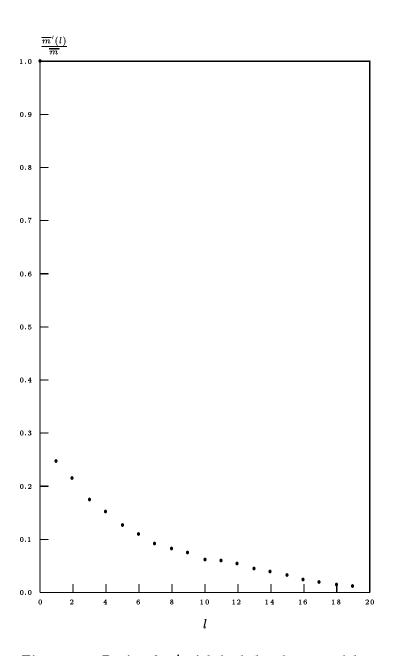


Figure 2.9: Ratio of \overline{m}' with lookahead to \overline{m} without.

down from 1.0 for a lookahead of zero to about $\frac{1}{4}$ for a lookahead of one).⁷ There is additional reduction as the lookahead increases. It is clear that deeper lookahead affects the mistake probability. To determine whether the effects are significant, we turn to a discussion of search procedures.

2.4 The Bounded Backtrack Search Algorithm

In the preceding sections, we have discussed lookahead strictly as a property of the search space. In reference to existing procedures, we may have suggested that there was some benefit and cost in a search procedure's performing lookahead, but we have tried to postpone the topic until this section.

We would now like to begin the discussion of how one can take advantage of search space properties to improve the performance of a backtracking search algorithm. The first step will be to define a general backtracking procedure that will serve as a framework for comparison. Depth first search and iterative sampling are interesting special cases of this procedure at two ends of a spectrum of possibilities. The central question we hope to answer in this chapter is whether backtracking procedures "in between" depth first search and iterative sampling could have better performance than either of them, and if so, under what conditions.

2.4.1 General Conditions

The following search algorithm is for binary search trees. It could easily be extended to arbitrary branch factors, but we are presenting the version for binary trees because the analysis is clearer and the lessons more direct. Every node has exactly zero or two successors, returned in an ordered list by the Successors function. A node that has zero successors is a *fringe* node. Some subset of the fringe nodes are identified as goals. The rest are dead ends. The task of the search algorithm is to search the tree for any goal node and to return the first goal node found. We are not requiring the

⁷When the lookahead reaches the height of the search tree, the ratio becomes zero. This data is taken from a subtree about twenty levels high (see Appendix E), so the fact that the curve approaches zero at twenty is expected.

search algorithm to find all goal nodes or to return with failure if there are no goal nodes in the search tree. We are also not stipulating that the search algorithm cannot revisit nodes already examined. In short, we are not requiring the search algorithm to be systematic.

We are interested in the average cost of the search algorithm to find a goal node given certain properties of the search space concerning the likelihood and cost of mistakes. The average cost of the search algorithm will be measured as the number of nodes examined.

2.4.2 Algorithm

The bounded backtrack search algorithm is a nonsystematic derivative of the simple depth first search procedure. To make it clear what the differences are, we will give a pseudocode description of depth first search (DFS), followed by bounded backtrack search (BBS).

```
DFS(node)
 1
      if timeout
 2
          return timeout-result
 3
      if GOAL-P(node)
 4
          return node
      for child in Successors (node)
 5
 6
          result \leftarrow DFS(child)
 7
          if result \neq NIL
 8
               return result
 9
      return NIL
```

Figure 2.10: Depth first search algorithm.

Both the algorithms do the same thing when they discover a goal node: they return it back up the tree, unwrapping the recursive calls. The only difference comes from when they run into a dead end node. BBS-PROBE keeps track, as it backtracks, of how far it has backtracked, i.e., the height of the subtree rooted at the node from

```
BBS-PROBE(node, lookahead)
  1
      if GOAL-P(node)
            return \langle node, 0 \rangle
  2
  3
       maxheight \leftarrow 0
  4
      for child in RANDOM-PERMUTATION(SUCCESSORS(node))
  5
            \langle result, height \rangle \leftarrow BBS-Probe(child, lookahead)
  6
            maxheight \leftarrow Max(maxheight, 1 + height)
  7
            if result \neq NIL
  8
                 return \langle result, 0 \rangle
  9
            if height \geq lookahead
                 return \langle \text{NIL}, maxheight \rangle
  10
  11 return \langle NIL, maxheight \rangle
BBS(node, lookahead)
  1
       loop until timeout
  2
            \langle result, height \rangle \leftarrow BBS-PROBE(node, lookahead)
            if result \neq NIL
  3
  4
                 return result
  5
      return timeout-result
```

Figure 2.11: Bounded backtrack search algorithm.

which it is backtracking. Whenever BBS-PROBE has backtracked from a subtree of height lookahead, it continues backtracking all the way to the root of the tree, ignoring unexplored siblings as it goes. Since BBS-PROBE leaves nodes unexplored, it may miss a goal node that exists in the original search tree. However, when BBS-PROBE fails, BBS tries again. With different random permutations of the lists of successors (line 4), the new probe has another chance at finding a solution.

2.4.3 Example

Suppose we applied the BBS algorithm with a lookahead of one to the search tree shown in Figure 2.12. Assume that the RANDOM-PERMUTATION function returned the successors in left to right order for a particular iteration of BBS-PROBE. Then on that probe, BBS-PROBE would not expand node G on account of the fact that it has backtracked to C from a subtree of height one, rooted at D. Note, however, that when BBS-PROBE backtracks to D from E, it continues to explore the siblings of E since the backtrack was from a subtree of height less than the lookahead of one.

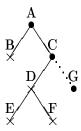


Figure 2.12: BBS-PROBE fails to discover the goal node G.

2.4.4 Relationship to Depth First Search

If the lookahead amount is greater than or equal to the height of the entire search tree, then a single iteration of BBS-PROBE is equivalent to DFS, with the exception that the successors are explored in random order. The exception can be significant when the Successors function takes into account heuristic information to order the nodes in a manner which expedites finding a solution. In Section 4.8.1 we discuss ways of incorporating heuristics with the Random-Permutation function by stochastically perturbing the original order, but it remains a consideration that the need of BBS for randomness competes with the efficacy of successor ordering heuristics. For the analysis in this chapter, we will assume there is no sacrifice in randomizing the successor order.

2.4.5 Relationship to Iterative Sampling

If, at the other extreme, the lookahead amount is zero, then the BBS algorithm is equivalent to Langley's iterative sampling algorithm [32]. However, there is an implementation factor that counting nodes does not consider. Since iterative sampling never backtracks to anything but the root node, it does not have to save state as it traverses a path. In practice, implementations of the Successors function typically copy and modify the previous state to generate the successors, or record changes in a manner that can be unwound. In both cases, the overhead required to support backtracking may be a significant expense that iterative sampling does not have to pay. Since we are analyzing search algorithms based on the number of nodes expanded, this expense will not be a factor in our analysis. In Chapter 4, we report experimental results showing that algorithmic improvements generally outweigh the constant factors for nontrivial problems.

⁸Note, however, the observation in Section 2.6.1.

⁹Bounded backtrack search only needs to keep the last l states for a lookahead of l since if it ever backtracks beyond l it backtracks all the way to the beginning. For especially large but easy problems, even this linear factor in storage may be significant. An implementation of the SAT solver Tableau [10] is bound by storage limitations for some problems that would be alleviated if it only recorded l levels of state.

2.4.6 Relationship to Depth First Search with Restarts

Depth first search with restarts, or RDFS [26], is another modification of depth first search that overcomes the problem of early mistakes by sacrificing systematicity. RDFS iteratively retries depth first search using randomized value selection and a timeout of some number of nodes. RDFS is similar to bounded backtrack search, and in some circumstances the algorithms are identical for suitably chosen timeout values.

Consider a probe of bounded backtrack search that is lucky enough (or unlucky enough) to make it all the way to a dead end node at the maximum depth of the tree without backtracking. Failing at depth d, this probe will backtrack to depth d-1 and then try the alternative at depth d. If that node is also not a solution the probe will backtrack to depth d-2, continuing in this manner to explore the entire subtree of height l below the $d-l^{th}$ node on its original path. When it backtracks up to the parent at d-l-1, the backtrack bound forces it to return all the way to the beginning. If the subtree of height l was full and RDFS had the timeout value of $d-l+2^{l+1}-1$, an RDFS probe would explore the identical nodes, given the same successor orderings.

The significant difference between the search algorithms comes when a probe makes "recoverable" mistakes of height less than l along the way. The nodes explored in the subtrees of the recoverable mistakes count against RDFS's eventual timeout, but do not affect the bounded backtrack probe.

RDFS could be given a timeout value of $(2^{l+1}-1)+(d-l)2^l$ nodes to accommodate recoverable mistakes of height l along the way, but this would have the side effect that if there were not many small excursions, "extra" time could be spent exploring subtrees of height exceeding l.

2.5 Average Case Cost Analysis

The average case cost of bounded backtrack search is the expected number of nodes searched to find a solution. Since successive probes (i.e., iterations of BBS-PROBE) are selected at random with replacement, this cost is simply the expected cost of a probe divided by the likelihood that a probe succeeds (Figure 2.13). The expected

cost of a probe is the average of all possible iterations of BBS-PROBE weighted by the frequency in which they occur. The likelihood could be computed in a variety of ways depending on the chosen assumptions about the search space. Our analysis will be based on assumptions about the likelihood of making a mistake at any level in the search tree, and about the height of the subtree below such a mistake.

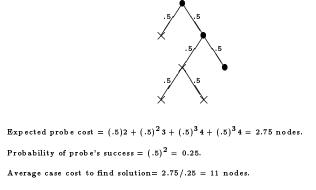


Figure 2.13: Calculating average case cost (no lookahead).

2.5.1 General Case

In Section 2.3 we introduced m_k , the probability of making a mistake at depth k in the search tree by expanding a bad child at depth k+1 before a good sibling. For certain special cases, m_k and the height of the search tree are all that is needed to predict the performance of the algorithm (see Section 2.6). In general, though, the performance of bounded backtrack search will depend on the height and size of the subtrees below mistakes. In this section, we will derive formulas for bounded backtrack's expected time to find a solution in terms of functions f(k,l) and c(k,l) characterizing the height and size of subtrees below mistake nodes.

Let f(k,l) be the probability that the subtree below a bad node at depth k in the search tree will be of height l. We assume this probability is independent of the particular path from the root of the tree to the bad node. We know a few things

about the function f(k,l) a priori. If there is a bad node at the deepest level in the tree, the subtree below that bad node must have height zero, for otherwise the bad node could not have been at the deepest level. By the same reasoning, we know that for any particular depth k, the value of f(k,l) is zero outside the range $0 \le l < d-k$.

The data of Section 2.3.4 show that for scheduling problems, most of the subtrees below mistake nodes have height zero. Thus we might expect that in general, for any depth k the function f(k,l) begins on a spike at l=0 and rapidly falls off to zero at l=d-k+1. How quickly the function falls off will be an important factor in calculating the cost of the search.

Probability of a Probe's Success

Based on m_k and f(k,l) we can calculate the probability of success for a probe of bounded backtrack search. We will assume solutions all lie at the deepest level of the tree, but the tree is not necessarily full, i.e., bad branches may be detected early and pruned. This assumption holds for the search trees of a wide variety of problem formulations from constraint satisfaction problems to searches using iterative deepening [29].

In order to find a solution, the probe needs to avoid making any "fatal" mistakes d times in a row. A fatal mistake is a mistake that it cannot recover from by backtracking because the height of the subtree is as much as the allowed lookahead. Since the tree is binary, the probability of making a fatal mistake at any depth k is one minus m_k times the probability that the height of a subtree of a bad node at depth k+1 is at least as much as the lookahead.

Letting the function $F(k,l) = \sum_{i < l} f(k,i)$, the probability of success for a probe of bounded backtrack search with a lookahead l is,

$$\prod_{0 \le k < d} 1 - m_k (1 - F(k+1, l)) \tag{2.3}$$

Thus the expected number of probes is,

$$\prod_{0 \le k \le d} \frac{1}{1 - m_k (1 - F(k+1, l))} \tag{2.4}$$

The Cost of Searching a Bad Node

Computing the average cost of a probe requires knowing the size of subtrees below bad nodes in addition to their heights (the subtrees are not necessarily full). We will assume the size of the subtree depends only on its depth in the original tree and its height. We denote the cost of a subtree of height l at depth k as c(k, l).

Let us first consider the cost of making a fatal mistake at depth k.¹⁰ Examining the bad node itself counts as one node. There is a chance of f(k,0) that the bad node has no successors, in which case the total cost of the mistake is one. However, with a chance of (1-f(k,0)), the search continues with the two successors of the bad node. There are two possibilities: (1) If the subtree of the first child has height less than the lookahead amount, then the search will fully expand the subtree and go on to explore the second child when it returns; (2) If the subtree of the first child has height greater than or equal to the lookahead amount, then the search will explore a restricted part of the subtree, but ignore the sibling when it returns. Letting A_1 and A_2 be the costs of the respective two possibilities, we have the cost of searching the subtree of a bad node at depth k (less than d) with bounded backtrack search using a lookahead of l:

$$Bad(k,l) = 1 + (1 - f(k,0))[F(k+1,l)A_1 + (1 - F(k+1,l))A_2]$$
 (2.5)

The cost of case (1) is the sum of the cost of searching the two successors. Since the cost of searching a bad node is independent of the path leading to it, we know nothing about the two successors other than the fact that the subtree of the first has height less than the lookahead. Thus the cost of searching the second is just Bad(k+1,l). The subtree of the first child will be fully explored, but its height must be less than the lookahead amount. Given f(k,0) and c(k,l), the cost of exploring this subtree is the weighted average of the costs of subtrees of the allowable heights. Thus, for $F(k+1,l) \neq 0$, 11

¹⁰That is, expanding a bad node at depth k whose parent at depth k-1 was not bad.

¹¹When F(k+1,l)=0, the coefficient of A_1 cancels with the zero denominator.

$$A_1 = Bad(k+1,l) + \frac{1}{F(k+1,l)} \sum_{j < l} f(k+1,j)c(k+1,j)$$
 (2.6)

In case (2), the second child is ignored. The first child is a bad node whose subtree has height at least l. Letting BadX(k,l,x) be the cost of exploring a bad node whose subtree has height at least x,

$$A_2 = BadX(k+1,l,l) \tag{2.7}$$

The function BadX(k,l,x) is similar to Bad(k,l) except that it takes into account the fact that the subtree is known to be of height at least x. When x is zero, the two functions are the same,

$$BadX(k,l,0) = Bad(k,l) \tag{2.8}$$

When x > 0, then BadX is defined as follows: As before, examining the bad node itself counts as one node. However, since x > 0 there is no chance that the bad node has no successors. There are three possibilities, depending on the height of the subtree of the first child: (1) If the subtree of the first child has height less than x - 1, then the search will fully expand the subtree and go on to explore the sibling, which is known to have a subtree of height at least x - 1 since the parent's subtree has height at least x; (2) If the subtree has height at least x - 1 but less than l, then the search will also fully expand the subtree and explore the sibling, but nothing is known about the height of the sibling's subtree; (3) If the subtree of the first child has height greater than or equal to the lookahead amount l, then the search will explore a restricted part of its subtree, but ignore the sibling when it returns. Letting B_1 , B_2 and B_3 be the costs of the respective possibilities, we have,

$$BadX(k,l,x) = 1 + F(k+1,x-1)B_1 + \left(\sum_{x-1 \le j < l} f(k+1,j)\right) B_2 + (1 - F(k+1,l)) B_3$$
(2.9)

The cost of the first case is the cost of exhaustively searching the first child, whose height is less than x-1, plus the cost of searching the sibling, whose height is at least x-1. For $F(k+1,x-1) \neq 0$, 12

$$B_1 = BadX(k+1,l,x-1) + rac{1}{F(k+1,x-1)} \sum_{j < x-1} f(k+1,j) c(k+1,j) \hspace{0.5cm} (2.10)$$

Since nothing is known about the height of the sibling in the second case, the cost of searching it is Bad instead of BadX:

$$B_2 = Bad(k+1,l) + rac{1}{\sum_{x-1 \leq j < l} f(k+1,j)} \sum_{x-1 < j < l} f(k+1,j) c(k+1,j) \qquad (2.11)$$

The cost of case (3) is the same as case (2) for Bad,

$$B_3 = A_2 = BadX(k+1, l, l) (2.12)$$

The cost of searching a bad node at the deepest level of the tree is one, since it has no nodes beneath it.

$$BadX(d,l,x) = Bad(d,l) = 1 (2.13)$$

Combining equations 2.5 through 2.13, the recurrences give the cost of searching a bad node at depth k with lookahead l in terms of f(k,l), c(k,l), and m_k .

The Cost of Searching a Good Node

Let us now consider the cost of searching a good node at depth k. To begin with, the good node itself counts as one. Then there are two possibilities depending on whether its first child is a mistake or a good node. With probability $(1 - m_k)$ the first child is a good node; and with probability m_k it is a mistake. Letting C_1 and C_2 be the respective costs, for k < d,

When F(k+1,x-1)=0, the coefficient of B_1 cancels with the zero denominator. Similar conditions hold for B_2 and C_2 , below.

$$Good(k,l) = 1 + (1 - m_k)C_1 + m_kC_2$$
(2.14)

It is easy to see that if the first child is a good node, the second child will never be explored. Thus the cost of searching a good node is,

$$C_1 = Good(k+1,l) (2.15)$$

The cost of searching a bad node first depends on whether the mistake is a "fatal" mistake. The mistake will be fatal with a probability of (1 - F(k+1,l)), and recoverable with a probability of F(k+1,l). The cost of a fatal mistake is the cost of searching a bad node whose subtree has height at least l, namely BadX(k+1,l,l). The cost of a recoverable mistake is the cost D_1 of exploring the bad child with a short subtree, which is $\frac{1}{F(k+1,l)} \sum_{j < l} f(k+1,j)c(k+1,j)$, plus the cost D_2 of searching the good sibling, which is Good(k+1,l).

$$C_2 = (1 - F(k+1,l))BadX(k+1,l,l) + F(k+1,l)(D_1 + D_2)$$
(2.16)

Where,

$$D_1 = \frac{1}{F(k+1,l)} \sum_{i \in I} f(k+1,j) c(k+1,j)$$
 (2.17)

$$D_2 = Good(k+1,l) \tag{2.18}$$

The cost of searching a good node at the deepest level is obviously one, so Equations 2.14, 2.15, and 2.16 define the cost of searching a good node at any depth in the tree. Taking k = 0, the recurrences yield the expected cost of one probe, or iteration of BBS-PROBE. In combination with the results of Section 2.5.1, this yields the average case cost of bounded backtrack search for any given lookahead in terms of three properties of the search space, f(k, l), c(k, l), and m_k .

2.6 Analysis for Full Binary Trees

In this section, we will apply the general analysis to full binary trees, for which f(k, l) and c(k, l) are known.

2.6.1 Full Binary Trees

In a full binary tree, the height of a subtree below any node extends to the maximum depth of the tree. Thus the function f(k,l), the probability that a subtree below a bad node at depth k will be of height l, is simply a delta function at k = d - l. For any k, the cost of any subtree of height l is just $2^{l+1} - 1$.¹³ In keeping with Section 2.3.2, we will assume for the sake of analysis that the probability m_k of making a mistake at depth k is constant for all depths. In practice, we could not expect m_k to be exactly constant, but the deviations are unlikely to affect the predicted performance significantly. We will denote this probability m. Thus we characterize the search space of a full binary tree as follows:

$$f(k,l) = [l = d - k] (2.19)$$

$$c(k,l) = 2^{l+1} - 1 (2.20)$$

$$m_k = m (2.21)$$

In the remainder of this section, we will apply the recurrences of the general analysis to a search space of these characteristics, working toward closed form solutions for the expected number of nodes examined by bounded backtrack search, iterative sampling, and depth first search.¹⁴

¹³For any k there are no subtrees of any height other than d-k, so the value of the cost function c(k, l) for $l \neq d-k$ is irrelevant.

¹⁴The derivations in this section have been validated by evaluating the general recurrence (Equation 2.14) with the above functions (Equations 2.19 to 2.21) on a set of example trees and comparing the results with the results of the expressions at each step in the derivation.

Bounded Backtrack Search

Let us begin by working on the Bad and BadX recurrences. Substituting the functions f and c above into Equations 2.5 through 2.13 yields the following:

$$Bad(k,l) = 1 + [k < d]([d-k-1 < l]A_1 + [d-k-1 \ge l]A_2)$$
 (2.22)

$$A_1 = Bad(k+1,l) + (2^{d-k}-1)$$
 (2.23)

$$A_2 = BadX(k+1,l,l) (2.24)$$

$$BadX(k,l,0) = Bad(k,l) (2.25)$$

$$BadX(k, l, x) = 1 + [d - k - 1 < x - 1]B_1$$

 $+[x - 1 \le d - k - 1 < l]B_2$
 $+[l \le d - k - 1]B_3$ (2.26)

$$B_1 = BadX(k+1,l,x-1) + [d-k-1 < x-1](2^{d-k}-1)$$
 (2.27)

$$B_2 = Bad(k+1,l) + [x-1 \le d-k-1 < l](2^{d-k}-1)$$
 (2.28)

$$B_3 = BadX(k+1,l,l)$$
 (2.29)

$$BadX(d,l,x) = Bad(d,l) = 1$$
(2.30)

We can solve the recurrence in pieces. In Equation 2.27, when d-k-1 < x-1,

$$B_1 = BadX(k+1, l, x-1) + (2^{d-k} - 1)$$
(2.31)

Since k is increased on the recursive call, the condition d - k - 1 < x - 1 still holds for the new value of k (even though x is decreased). Thus the coefficients of B_2 and B_3 in Equation 2.26 remain zero. Combining Equations 2.26 and 2.31,

$$BadX(k,l,x) = BadX(k+1,l,x-1) + 2^{d-k}$$
(2.32)

This recurrence holds until k reaches d, when BadX(k,l,x) = 1 by Equation 2.30.¹⁵ The solution to the recurrence is just the sum of d - k + 1 increasing powers of two,

¹⁵ If x reaches zero before k reaches d, then $BadX(k, l, 0) = Bad(k, l) = Bad(k + 1, l) + 2^{d-k}$ by Equations 2.25 and 2.22. This recurrence is identical to the BadX recurrence (Equation 2.32), also stopping at k = d.

or $2^{d-k+1}-1$. Thus by Equation 2.31, $B_1=2^{d-k+1}-2$. A similar argument extends to B_2 , so,

$$B_1 = B_2 = 2^{d-k+1} - 2 (2.33)$$

In Equation 2.26, when $l \leq d-k-1$ the coefficients of B_1 and B_2 are zero. The recurrence has the form BadX(k,l,x)=1+BadX(k+1,l,l) until l=d-k, at which point $BadX(k,l,x)=2^{d-k+1}-1$ from Equation 2.33. The solution to this recurrence is just d-k-l plus the value of the stopping case. The coefficient B_3 is one less than BadX(k,l,x):

$$B_3 = d - k - l - 2 + 2^{l+1} (2.34)$$

Combining Equations 2.26 through 2.34,

$$BadX(k,l,x) = [d-k-1 < l](2^{d-k+1}-1) + [d-k-1 \ge l](d-k-l+2^{l+1}-1)$$
(2.35)

We now turn to the cost of searching a good node in a full binary tree. Substituting f and c and Equation 2.35 into Equations 2.16 and 2.17,

$$C_2 = [d-k-1 \ge l](d-k-l+2^{l+1}-2) + [d-k-1 < l](D_1 + D_2)$$
 (2.36)

$$D_1 = 2^{d-k} - 1 \qquad \text{for } d-k-1 < l \qquad (2.37)$$

 D_1 is the expected cost of subtrees at depth k+1 that are shorter than l. In a full binary tree, all subtrees at depth k+1 have height exactly d-k-1, so Equation 2.37 is defined if (and only if) d-k-1 < l, the range for which its coefficient is non-zero.

We can now solve the recurrence of Equation 2.14 in pieces. When d - k - 1 < l,

$$Good(k,l) = 1 + (1-m)Good(k+1,l) + m(2^{d-k} - 1 + Good(k+1,l))$$

$$= 1 + Good(k+1,l) + m(2^{d-k} - 1)$$

$$= Good(k+1,l) + (1-m) + m2^{d-k}$$
(2.38)

For k = d, Good(k, l) = 1. For k < d, the recurrence adds d - k times the quantities (1-m) and m times a decreasing power of two. After some simplification,

$$Good(k,l) = (d-k+1)(1-m) + m \sum_{k \le j \le d} 2^{d-j}$$
$$= m(2^{d-k+1} - 2 - d + k) + d - k + 1$$
(2.39)

When $d-k-1 \geq l$, that is, when d-k > l,

$$Good(k,l) = 1 + (1-m)Good(k+1,l) + m(d-k-l+2^{l+1}-2)$$
(2.40)

The value of the recurrence when k finally reaches d-l comes from above (Equation 2.39), substituting d-l for k. Let $E_2=1+l+m(2^{l+1}-2-l)$, the stopping value of the recurrence at k=d-l. To simplify the form of the recurrence, let $E_1=m(d-l+2^{l+1}-2)+1$. Solving Equation 2.40,

$$Good(k,l) = 1 + (1-m)Good(k+1,l) + m(d-k-l+2^{l+1}-2)$$

$$= (1-m)Good(k+1,l) + E_1 - mk$$
(2.41)

Solving this recurrence is more tedious than technically difficult. Since the recurrence stops at k = d - l, E_2 (the value of Good(k, l) at k = d - l) is multiplied by (1 - m) for d - k - l iterations. The E_1 and mk terms are summed and multiplied by (1 - m) on each successive iteration yielding,

$$Good(k,l) = E_{2}(1-m)^{d-l-k}$$

$$+ \sum_{0 \le j < d-l-k} (E_{1}(1-m)^{j} - (1-m)^{j}m(k+j))$$

$$= E_{2}(1-m)^{d-l-k}$$

$$-(\frac{E_{1}}{m} - k)((1-m)^{d-l-k} - 1)$$

$$-m \sum_{0 \le j < d-l-k} j(1-m)^{j}$$

$$(2.42)$$

The closed form solution of the sum $\sum_{0 \le j < y} j a^j$ is $\frac{(y-1)a^{y+1} - ya^y + a}{(a-1)^2}$. Thus,

$$Good(k,l) = E_{2}(1-m)^{d-l-k} - (\frac{E_{1}}{m}-k)((1-m)^{d-l-k}-1) - m\frac{(d-l-k-1)(1-m)^{d-l-k+1} - (d-l-k)(1-m)^{d-l-k} + (1-m)}{m^{2}}$$

$$= E_{2}(1-m)^{d-l-k} - (\frac{E_{1}}{m}-k)((1-m)^{d-l-k}-1) + (d-l-k+\frac{1}{m}-1)(1-m)^{d-l-k} - \frac{1-m}{m}$$

$$= E_{2}(1-m)^{d-l-k} - (d-l+2^{l+1}-2+\frac{1}{m}-k)((1-m)^{d-l-k}-1) + (d-l-k+\frac{1}{m}-1)(1-m)^{d-l-k} - \frac{1-m}{m}$$

$$= E_{2}(1-m)^{d-l-k} + (1-(1-m)^{d-l-k})(2^{l+1}-1) + (d-l-k+1-k)(2^{l+1}-1)$$

The components of this last equation (Equation 2.43) are simple enough to understand in terms of the search. Consider a good node at depth k somewhere above d-l. Any probe from the good node will follow a straight path without backtracking down to depth d-l. To that point, there is a $(1-m)^{d-l-k}$ chance of having not made a mistake. If the probe gets to that depth without making a mistake, then it will explore the full subtree below until finding a solution. The expected number of nodes to search this subtree is E_2 (Equation 2.39). This is the first component of Equation 2.43. The chance of having made a mistake somewhere along the path to

depth d-l is $1-(1-m)^{d-l-k}$. If a mistake has been made, then the search will explore the full subtree below, which is known to have no solutions (since it is below a bad node). The cost of exploring the full subtree is $2^{l+1}-1$. This is the second component of Equation 2.43. The third component, d-l-k, is just the number of nodes expanded along the path to depth d-l.

Taking k=0, Equation 2.43 gives the final closed form expression for the expected number of nodes expanded by one probe of bounded backtrack search for lookahead $l \leq d$:

$$ProbeCost_{BBS}(l) = (1 + l + m(2^{l+1} - 2 - l))(1 - m)^{d-l} + (1 - (1 - m)^{d-l})(2^{l+1} - 1) + d - l$$

$$(2.44)$$

The likelihood that any individual probe succeeds is $(1-m)^{d-l}$, for any $l \leq d$. Thus the expected number of probes is,

$$NumProbes_{BBS}(l) = \frac{1}{(1-m)^{d-l}}$$
 (2.45)

The product of ProbeCost and NumProbes (2.44 and 2.45) is the expected number of nodes that bounded backtrack search will expand before finding a solution, searching a full binary tree that has mistake probability m.

Depth First Search

The analysis of depth first search follows directly from the above analysis, observing that depth first search is an extreme case of bounded backtrack search taking the height of the tree as the lookahead amount. Substituting l = d into Equation 2.44,

$$ProbeCost_{DFS} = 1 + d + m(2^{d+1} - 2 - d)$$
 (2.46)

Since a single "probe" of depth first search examines the whole tree until finding a solution, one probe suffices. This equation shows that the average case performance of depth first search is proportional to the mistake probability. A practical consequence

of this fact is that successor ordering heuristics that decrease the chance of making a mistake will generally have only a marginal effect on the average case performance. When successor ordering heuristics are used to battle the exponentiality of a search space, depth first search is unlikely to be the search method of choice. We will return to these implications in Section 2.6.2.

Iterative Sampling

Iterative sampling is another extreme case of bounded backtrack search. Substituting l = 0 into Equation 2.44, or realizing it's obvious,

$$ProbeCost_{ISAMP} = 1 + d (2.47)$$

$$NumProbes_{ISAMP} = \frac{1}{(1-m)^d}$$
 (2.48)

In contrast to depth first search, the performance of iterative sampling varies with the d^{th} power of m, suggesting that successor ordering heuristics would be much more effective for this search procedure. Unfortunately, iterative sampling depends on at least some randomness in the value selection, which competes with the successor ordering heuristic if one is used (see Section 4.8.1).

2.6.2 Crossover Point

It is clear from the above analysis that for a particular depth, iterative sampling will be superior to depth first search for full binary trees with small mistake probabilities. However, when there is only a unique solution, depth first search will be superior because it avoids searching nodes redundantly.

When there is single solution, the probability of making a mistake at any good node is 1/2, since only one of its two successors could have the goal in the subtree below. When m = 1/2, the average cost of depth first and iterative sampling searches are,

$$Unique Cost_{DFS} = 2^d + \frac{d}{2} \tag{2.49}$$

$$Unique Cost_{ISAMP} = 2^{d}(1+d) \tag{2.50}$$

Thus in the large depth limit, depth first search will explore only 1/d times as many nodes as iterative sampling in order to find a unique solution.

Somewhere in between a small mistake probability and a mistake probability of 1/2, depth first search and iterative sampling will have equivalent performance. The following graphs show this crossover point for varying m at heights ten and thirty. The range for m goes from 0.01 for a search space having many solutions (depending on the depth) to 0.5 for a search space with a unique solution. As expected, depth first search is affected only linearly by m, making the crossover point fairly dramatic.

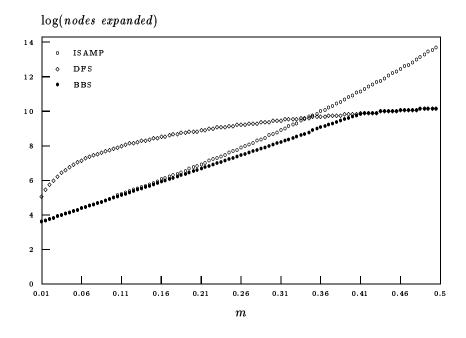


Figure 2.14: Theoretical crossover point for full binary trees of height 10.

In Figures 2.14 and 2.16, the solid line is the performance of bounded backtrack search, with the optimal lookahead chosen for each value of m. Since both l=0 and l=d are legitimate lookahead values, this line is guaranteed to go above neither of the other lines. What is interesting is whether it ever goes below both lines. In such a circumstance, bounded backtrack search is superior to both iterative sampling and

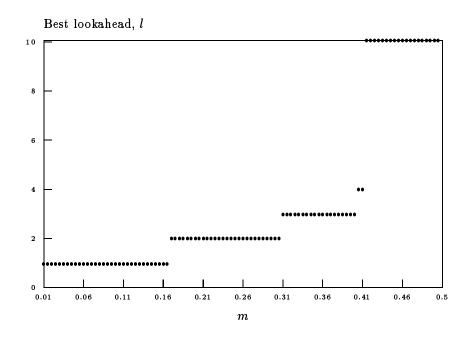


Figure 2.15: Optimal lookahead for full binary trees of height 10.

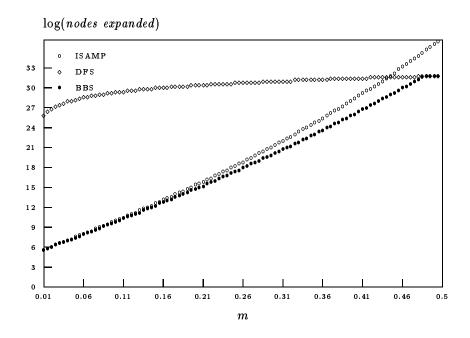


Figure 2.16: Theoretical crossover point for full binary trees of height 30.

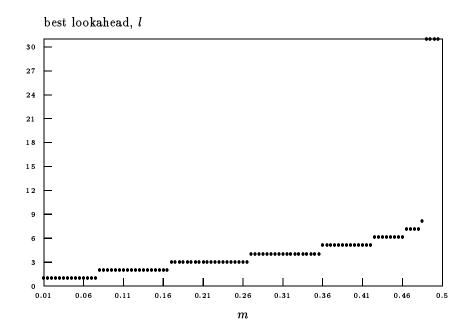


Figure 2.17: Optimal lookahead for full binary trees of height 30.

depth first search. The figures show that it indeed does. Furthermore, as the depth of the search increases (comparing Figure 2.14 with depth ten to Figure 2.16 with depth thirty), the range of superiority covers most problems with multiple solutions.

Analytically, the crossover point between iterative sampling and depth first search is at particular values of m and d such that,

$$\frac{1+d}{(1-m)^d} = 1 + d + m(2^{d+1} - 2 - d) \tag{2.51}$$

Observing that m is always in the range $0 \le m \le .5$ and letting p = 1 - m, we can see that as $d \to \infty$,

$$egin{array}{lcl} rac{1+d}{(1-m)^d} &=& 1+d+m(2^{d+1}-2-d) \ &d&pprox&(1-m)^d(d+m2^d) \ &pprox&p^d(d+2^d-p^{d+1}) \ &pprox&(2p)^d \end{array}$$

$$\log d \approx d \log 2 + d \log p$$
 $\log p \approx \frac{\log d}{d} - \log 2$
 $p \approx e^{-\log 2}$
 $m \approx \frac{1}{2}$ (2.52)

Thus in the large depth limit, iterative sampling is superior to depth first search for any full binary tree with m < 1/2.¹⁶

2.6.3 Choosing the Lookahead

Figures 2.15 and 2.17 show the optimal lookahead values used by bounded backtrack search in the corresponding graphs 2.14 and 2.16. Oddly, the optimal lookahead values are fairly small for most of the m range and then jump sharply to d at some point near m=.5. In experiments with scheduling problems,¹⁷ we have found the performance of bounded backtrack search not to be very sensitive to the particular lookahead value chosen. This is a helpful property for applying the algorithm to real world problems where an analytically optimal lookahead cannot be determined. In practice, you can just try a few lookahead values on a training set of problems and pick the best value.¹⁸

Figure 2.18 shows the theoretical performance of bounded backtrack search for a variety of lookahead amounts ranging from l=0, which is equivalent to iterative sampling, to l=d, which is equivalent to depth first search. We find that for full binary trees, the choice of l makes more difference than it did in our scheduling experiments. It remains the case, however, that a reasonably optimal lookahead can be found by trying a few alternatives on a training set of problems.

¹⁶The condition that m < 1/2 is stronger than the condition that there be multiple solutions, as it implies the number of solutions grows exponentially with increasing depth.

¹⁷These experiments were on non-full binary trees.

¹⁸Try increasing powers of two. We have found a lookahead value of four or eight to work well in general.

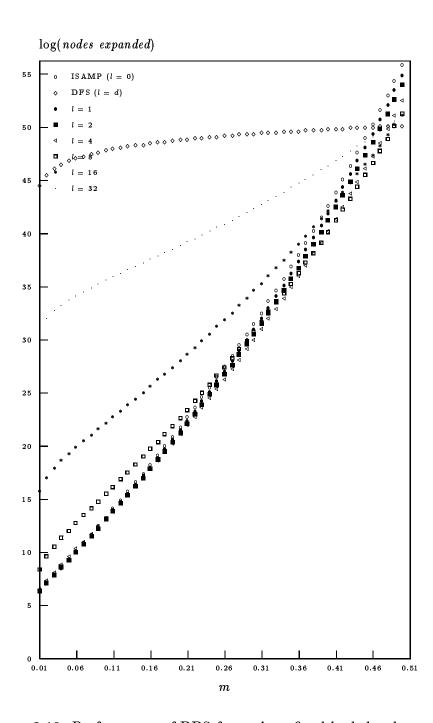


Figure 2.18: Performance of BBS for various fixed lookahead amounts.

2.6.4 If There Were Pruning

Figures 2.14 and 2.16 show that for full binary trees, bounded backtrack search will have better performance than iterative sampling or depth first search in the triangular region between the three curves. We have seen, though, that the search trees of scheduling problems are anything but full. In fact, most subtrees below mistake nodes are pruned right at the mistake nodes themselves. To better predict the performance of bounded backtrack search on problems with these characteristics, we profiled scheduling search spaces to find the distribution of the mistake subtree heights. We expected the f(k,l) curve to drop quickly from its peak at l=0, trailing off to zero at l = d - k + 1. We found, to our surprise, that after the initial peak the subtrees were fairly evenly distributed among all the possible heights.¹⁹ Yet despite the early pruning, the number of nodes in a mistake subtree of height l still grew exponentially with l. This meant that while the vast majority of mistakes had small subtrees, some had very large subtrees. Ironically, these are the two pathological cases for iterative sampling and depth first search that we discussed earlier in Section 2.2.1 and Section 2.2.2. We would thus expect that for problems with these characteristics, the triangular regions in graphs like Figures 2.14 and 2.16 would be much larger.

2.7 Conclusion

Our investigation of job shop scheduling has shown that real world problems can have characteristics that seriously hurt the average case performance of iterative sampling and depth first search. If iterative sampling is too eager to backtrack, depth first search is too reluctant. In a sense, bounded backtrack search is a compromise between the two extremes. We have shown theoretically that bounded backtrack can be a significant improvement over these existing techniques. Our profiles of scheduling search spaces indicate that the improvements will be even greater on real problems. The experimental results that test this optimistic outlook are given in Chapter 4.

¹⁹Details of the experiment are given in Appendix B.

Chapter 3

Limited Discrepancy Search

3.1 Introduction

In practice, many search problems have spaces that are too large to search exhaustively by any method. One can often find solutions in a small fraction of the search space by relying on carefully tuned heuristics to guide the search toward regions of the space that are likely to contain solutions. For many problems, heuristics can lead directly to a solution—most of the time. In this chapter, we consider what to do when the heuristics fail.

We will focus our attention on procedures for solving tree search problems. This will allow us to state our objective more succinctly: For tree search problems with heuristically ordered successors, is there a search procedure that is more likely to find a solution in a given time limit than chronological backtracking and iterative sampling?

We will address this question as follows: In the next section, we comment on existing algorithms. We then introduce limited discrepancy search and derive upper bounds for the algorithm. In Section 3.5, we endeavor to quantify the properties of the successor ordering heuristic, which leads to a detailed analysis of the algorithm. In the following section we compare the performance of limited discrepancy search with other techniques based on theoretical properties of the search space. Finally, in Section 3.7 we discuss variations of LDS that we believe will be useful for solving real world problems. Experimental results are given in Chapter 4.

3.2 Existing Strategies

Consider a tree search problem for which the successor ordering heuristic is so good that it almost always leads directly to a solution. Such problems are common both in practice and in areas of AI research such as planning and scheduling [45, 52]. If the heuristic is good enough, one might be satisfied with an algorithm that follows the heuristic and just gives up if the heuristic fails to lead to a solution, an algorithm we will call "onesamp" [26, 45]. However, if that performance is not satisfactory, one is confronted with the question of what search algorithm to use when onesamp leads

to a dead end. Iterative sampling and backtracking are two candidates.

3.2.1 Iterative Sampling

Iterative sampling [32], or isamp, is the simple idea of following random paths, or probes, from the root until eventually discovering a path that leads to a solution. At each node on a path, one of the successors is selected at random and expanded. Then one of its successors is selected at random, and so on until reaching a goal node or dead end. If the path ends at a dead end, isamp starts a new probe, beginning again at the root.

Since the algorithm samples with replacement, the chance that any one probe ends at a goal node is not affected by the failure of earlier probes. Thus, assuming a fixed number of solutions, one can reduce the chance of not finding a solution to an arbitrarily small positive amount by taking enough samples.

Iterative sampling has been shown to be effective on problems where the solution density is high [11], but its performance as a fallback procedure for onesamp is more questionable because it ignores the successor ordering heuristic. If the heuristic were the key to solving the problem despite a low solution density, one would not expect iterative sampling to be effective. We have experimented with biasing the random selection of successors according to the heuristic, but the results have been discouraging (see Section 4.8.1).

3.2.2 Backtracking

An alternative fallback procedure is simply to backtrack chronologically when one-samp fails. Our experiments in Section 4.4 with scheduling show that falling off the heuristic with chronological backtracking does not provide much improvement over onesamp itself, and the analysis of mistakes in Chapter 2 provides an explanation. There is a reasonable chance that somewhere early in onesamp's path, it made a mistake by selecting a successor that had no goal nodes in the entire subtree below it. Once it made this early mistake and committed to that successor's subtree, none of the subsequent decisions made any difference.

If the subtree below a mistake is large, chronological backtracking will spend all of the allowed run time exploring the empty subtree without ever making it back up to the last decision that actually mattered. If one is counting on the heuristics to find a goal node in a small fraction of the search space, then chronological backtracking puts a tremendous burden on the heuristics early in the search and a relatively light burden on the heuristics deep in the search. Unfortunately, for many problems the heuristics are *least* reliable early in the search, before making decisions that reduce the problem to a size for which the heuristics become reliable. Because of the uneven reliance on the heuristics, it is unlikely that chronological backtracking is making the best use of the heuristic information.

3.2.3 Other Nonsystematic Methods

Other nonsystematic backtracking strategies address the problem of early mistakes and at the same time make use of heuristic information. We present experimental results for depth first search with restarts (RDFS), variations of iterative broadening (SIB), and multiprobe (BBS) in Chapter 4. All of the methods balance their dependence on heuristics with the need to search other alternatives by making random moves some of the time instead of following the heuristic recommendation. From the standpoint of relying on the heuristics, these methods are all fundamentally the same as isamp, so we will not go into any further detail describing the algorithms here.

3.3 Discrepancies

Let us return to the tree search problems for which the successor ordering heuristic almost always leads onesamp directly to a solution. The concern, though, will be what to do in those cases for which the heuristic does not lead to a solution. Our intuition is that when onesamp fails, the heuristic probably would have lead to a solution if it had only not made one or two "wrong turns" that got it off track. It ought to be possible to systematically try following the heuristic at all but one decision point. If that fails, we can try following the heuristic at all but two decision points. If the

number of wrong turns is small, we will find a solution in a few restricted searches of the space.

We call the decision points at which we do not follow the heuristic "discrepancies." Limited discrepancy search embodies the idea of iteratively searching the space with a limit on the number of discrepancies allowed on any path. The first iteration, with a limit of zero discrepancies, is just like onesamp. The next iteration searches all possibilities with at most one discrepancy, and so on.

The algorithm is shown in Figure 3.1. We will assume the search tree is binary. Successors is a function that returns a list of the either zero or two successors, with the heuristic preference first. The tree is not required to be a full binary tree, although our examples will use full trees.

In Figure 3.1, x is the discrepancy limit. We iteratively call LDS-PROBE, increasing x each time. LDS-PROBE does a depth first search traversal of the tree, limiting the number of discrepancies to x. When eventually x reaches d, the maximum depth of the tree, LDS-PROBE searches the entire tree exhaustively. Thus the search is guaranteed to find a goal node if one exists and is guaranteed to terminate if there are no goal nodes.

Since each iteration of LDS-PROBE limits the number of discrepancies to x instead of restricting the search to those nodes with exactly x discrepancies, iteration n reexamines all of the nodes that iterations 0..n-1 already examined (see Figure 3.2). We will see in Section 3.4 that the redundancy is not a significant factor in the complexity of the search.

Figure 3.2 shows a trace of LDS exhaustively searching a full binary tree of height three. The heuristic orders nodes left to right. The twenty pictures show all the paths to depth three, in order. The dotted lines and open circles represent nodes that were not backtracked over since the previous picture, so you can follow the trace by looking at the pictures in sequence. Counting all the black circles gives the total number of nodes expanded in the search, forty.

Forty nodes seems like a lot more than fifteen, the number of nodes that depth first search would require to search the tree exhaustively. Let us take a closer look at the amount of redundancy.

```
LDS-PROBE (node, k)
  1
      if GOAL-P(node)
  2
          return node
  3
      s \leftarrow \text{SUCCESSORS}(node)
  4
      if NULL-P(s)
  5
          return NIL
  6
      if k = 0
  7
          return LDS-PROBE(FIRST(s), 0)
  8
      else
  9
           result \leftarrow LDS-PROBE(SECOND(s), k-1)
  10
          if result \neq NIL
  11
               return result
  12
          return LDS-PROBE(FIRST(s), k)
LDS(node)
      for x \leftarrow 0 to maximum depth
 2
          result \leftarrow \text{LDS-PROBE}(node, x)
  3
          if result \neq NIL
  4
               return result
  5
      return NIL
```

Figure 3.1: Limited Discrepancy Search.

3.4 Upper Bounds

When we introduced limited discrepancy search, we argued that when one samp failed, the heuristic probably only made a few wrong turns. Consider a full binary search tree of height 1000. There is only one possibility at depth 1000 with zero discrepancies. There are 1000 possibilities with one discrepancy. An iteration of LDS-PROBE with a discrepancy limit of one expands about 500 nodes, on average, per fringe node. Thus even searching the space restricted to one discrepancy requires examining about 500,000 nodes. The iteration with a zero discrepancy limit examines only 1000 nodes. Even regarding this iteration as redundant, the 1000 nodes are only a small fraction of the 500,000 examined on the next iteration.

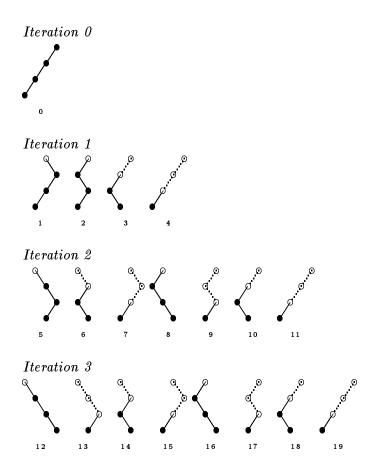


Figure 3.2: Execution trace of LDS.

It is easy to see that the number of possibilities with x discrepancies is $\binom{d}{x}$ where d is the depth of the tree. Ignoring the internal nodes, the cost of each iteration grows by a factor of about d for $x \ll d$. This suggests the following upper bound for the total number of possibilities examined by x + 1 iterations of the algorithm (exhaustively searching the space restricted to x or fewer discrepancies):

$$\sum_{k=0}^{x} \begin{pmatrix} d \\ k \end{pmatrix} (x+1-k) \tag{3.1}$$

The (x+1-k) factor comes from the fact that each iteration explores all the possibilities with at most k discrepancies, instead of exactly k discrepancies. The first iteration examines $\binom{d}{0}$ possibilities. The next iteration examines $\binom{d}{0} + \binom{d}{1}$. The next iteration examines $\binom{d}{0} + \binom{d}{1} + \binom{d}{2}$, and so on.

We can rewrite Equation 3.1 as,

$$(x+1)\sum_{k=0}^{x} {d \choose k} - \sum_{k=0}^{x} k {d \choose k}$$

$$(3.2)$$

Dropping the second sum and replacing $\binom{d}{k}$ with $\binom{d}{x}$ in the first sum, we get an upper bound,

$$(x+1)^2 \left(\begin{array}{c} d \\ x \end{array}\right) \tag{3.3}$$

For $x \ll d$, bound on the total cost of x iterations grows by about d as x increases, suggesting that the cost of earlier iterations is insignificant compared to the last. For small x, a tighter upper bound N can be derived by approximating $\binom{d}{k}$ by d^k :

$$N < \sum_{k=0}^{x} (x+1-k)d^k$$
 (3.4)

$$= (x+1)\sum_{k=0}^{x} d^{k} - \sum_{k=0}^{x} k d^{k}$$
 (3.5)

$$= (x+1)\left(\frac{d^{x+1}-1}{d-1}\right) - \frac{d-(x+1)d^{x+1}+xd^{x+2}}{(1-d)^2}$$
(3.6)

The second sum in Equation 3.5 has the form $\sum_{0 \le j < y} j a^j$, whose closed form solution is $\frac{(y-1)a^{y+1}-ya^y+a}{(a-1)^2}$.

$$= \frac{(x+1)(d^{x+1}-1)(d-1)-d+(x+1)d^{x+1}-xd^{x+2}}{(1-d)^2}$$
 (3.7)

$$= \frac{d^{x+2} - (x+2)d + x + 1}{(1-d)^2} \tag{3.8}$$

$$< d^x \left(\frac{d}{d-1}\right)^2 \tag{3.9}$$

For large d, $\left(\frac{d}{d-1}\right)^2$ is approximately one, so the upper bound for the number of possibilities searched by LDS with a discrepancy limit of x is approximately d^x . For large search spaces, it is clear that only small values of x are of practical interest. In our experiments with job shop scheduling, "small" meant one or two.

3.5 The Likelihood of Finding a Solution

Recognizing that for many search problems we will not have time to search the space exhaustively, we would like to know the likelihood of finding a solution, using the various methods, in the amount of time we actually are willing to wait. To be precise, we will have to formalize what we mean for the heuristic to make a "wrong turn."

3.5.1 Wrong Turns

For simplicity, we will consider only the case of a full binary tree. The two children of each choice point are assumed to be in the order of heuristic preference. We will further assume that if a choice point has a goal node in the subtree below it, then with probability p (the heuristic probability) its first child has a goal node in its subtree. In the 1-p chance that the first child does not have a goal, the other child must have a goal since the choice point has only two children. In this case the heuristic has made a wrong turn by putting the children in the "wrong" order.

The notion of a wrong turn is closely related to the mistake probability. In Chapter 2, we defined a bad node to be a node that does not have any goal nodes in its subtree. We defined a mistake to be a bad node whose parent is not bad. The mistake probability, m, is the probability that a randomly selected child of a good node is bad. If the heuristic orders successors randomly, the heuristic probability is the dual of the

mistake probability, p = 1 - m. If the heuristic does better than random selection, p > 1 - m. Conceivably, the heuristic could do worse than random selection, but in that case it would be better just not to use it. The worst possible heuristic (the "anti-heuristic") has p = 1 - 2m, making a wrong turn at every possible opportunity.²

Figure 3.3 shows the four possibilities for a node and its children. The "x" indicates a bad node, the solid dot a good node. In the figure, p is the probability that a node is in class X or Y (the two classes with good left children) given that it is not in class W (the only class where the parent is bad). The mistake probability m is one half the probability that a node is Y or Z (the classes with one mistake child) given that it is not W. The conditional probabilities of other combinations X, Y, and Z follow, as shown in the figure.

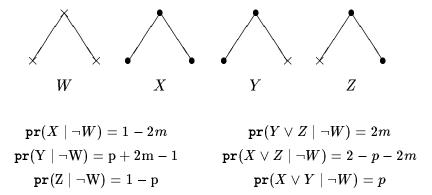


Figure 3.3: The four possibilities for a node and its children.

In Chapter 2, we argued that the mistake probability m would be constant over all depths in the tree. We found experimentally that this assumption was reasonable, which lead to analytical estimates for the expected run time of various algorithms. In particular, the chance of finding a solution on a random path to depth d (i.e., using isamp) is simply $(1-m)^d$. Using heuristics, onesamp has probability p^d of finding a solution on its one and only path.

²See Figure 3.3. Since the four cases are exhaustive, $pr(X \vee Y | \neg W) + pr(Y \vee Z | \neg W) \ge 1$. Thus $p + 2m \ge 1$, and $p \ge 1 - 2m$.

3.5.2 Estimating Heuristic Probability

We can derive an estimate of p by running onesamp on a large training set of problems from the domain of interest. Let s be the success rate of onesamp on the training set. Since the probability of onesamp is p^d , we have $p = s^{1/d}$. If s is small, the training set may have to be impractically large to get a reliable estimate. For some problems, though, s is not small. Heuristics developed for job shop scheduling have been shown to yield a probability s that is nearly one for small research problems [45]. We have found in earlier experimental work on the same problems [26] that even standard CSP heuristics can yield a success rate of about seventy-five percent. On larger scheduling problems [49] the success rate of onesamp is less, but more sophisticated heuristics from Operations Research keep onesamp competitive with other search techniques [8]. It appears that for many large tree search problems, the heuristics are the key to finding a solution in a small fraction of the search space. Under these conditions, LDS can offer a substantial improvement over onesamp.

3.5.3 Analysis of Early Iterations

Preliminaries

The early iterations of limited discrepancy search are the most important, practically, because for large problems the cost of successive iterations grows at a rate that quickly exceeds any reasonable amount of time one would be willing to wait. We will focus our attention on the second iteration, with d+1 probes. To simplify the analysis, we will assume that we start with this iteration, skipping the onesamp probe at the beginning of the normal LDS run. Since searching with a discrepancy limit of one includes the path with zero discrepancies, we will get the onesamp probe on this iteration also. The heuristic path is in fact the last probe on the iteration, as shown in Figure 3.4.

The figure shows the five probes of the one discrepancy iteration on a search tree of height four. We will assume the problem has a solution, so the root node is a good node, drawn as a solid dot. We do not know a priori the goodness of the other internal nodes, so they are drawn as open circles. For the sake of analysis, we will

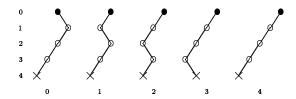


Figure 3.4: Probes 0 through 4 of iteration 1.

assume when analyzing a particular probe that the earlier probes failed, so it is helpful to draw the terminal nodes as bad nodes. The nodes are numbered for each probe starting with the root node, from zero to four.

It will be useful to define a few propositions for a search tree of height d:

- $good_{i,j} \equiv \text{Node } j$ of probe i is a good node.
- $succeed_i \equiv good_{i,d}$
- $redundant_i \equiv \exists_{l < i} succeed_l$

We can then define the probabilities:

- $ullet \ g_{i,j} \equiv \mathtt{pr}(good_{i,j+1} \mid good_{i,j} \land \neg redundant_i)$
- $s_i \equiv pr(succeed_i \mid \neg redundant_i)$
- $a_{i,d} \equiv pr(redundant_i)$

Conditional Probability that a Probe Succeeds

Our objective is to calculate the likelihood of finding a solution in i probes, in terms of m and p. For a tree of height d, we will denote this probability $a_{i,d}$. The probability of finding a solution on the first probe, for example, is just $a_{1,d}$. The probability of finding a solution in two probes $(a_{2,d})$ is one minus the probability of failing to find a solution on both the first and second probes. These failures are not independent

events since the first and second probes begin from the same node (see Figure 3.4).³ The first decision of the first probe (going right) is a possible explanation for the probe's failure. If this decision were the mistake, then the first decision of the second probe (going left) would have to be good. Thus the failure of the first probe increases the likelihood that the second probe will succeed.

The failure of the second probe increases the likelihood that the third will succeed since their second decisions (i.e., third nodes) are alternatives. However, the failure of the second probe also decreases the likelihood that the third will succeed since their first decisions (second nodes) are the same. Let s_i be the conditional probability that probe i succeeds given that all earlier probes have failed. For probe i to succeed, none of the decisions in the path can be mistakes. If $g_{i,j}$ is the probability that node j+1 on the path succeeds given that its parent is good, then s_i is just the product of the $g_{i,j}$ for $0 \le j < d$. Thus if we knew $g_{i,j}$ for all i and j, we could calculate $a_{i,d}$, our original objective.

Conditional Probability that a Node is Good

The probability that a node is good is unaffected by the failure of earlier probes that do not contain the node's parent. We can derive expressions for $g_{i,j}$ for three cases, j > i, j = i, and j < i. The first case is the easiest, because for j > i the failure of the earlier probes is obviously unrelated. Node j + 1 is the left child of node j. Thus,

For
$$j > i$$
, $g_{i,j} = pr(good_{i,j+1} \mid good_{i,j}) = p$ (3.10)

In Figure 3.4, for example, the conditional probability that node 4 of probe 2 is good is not affected by $\neg redundant_2$, i.e., the fact that probes 0 and 1 failed. The case that j=i is also not affected by the failure of the earlier probes. Given that node 2 of probe 2 is good, the probability that node 3 is good is just the probability that node 2 is X or Z, given that it is not W. From Figure 3.3,

For
$$j = i$$
, $g_{i,j} = pr(X \vee Z \mid \neg W) = 2 - p - 2m$ (3.11)

³We assume that whether or not a path beginning from a good node ends in a goal is independent of the success or failure of probes that have no nodes in common with it.

The case that j < i is affected by the $\neg redundant_i$ condition. To ground the reasoning, we will begin with an example for a specific i and j. We then lift the example to the case j = i - 1, and finally generalize the result to j < i.

Case
$$j = 2, i = 3$$

Consider the likelihood that node 3 of probe 3 is good, given that its parent is good. The parent, node 2, would have to be either X or Y, given that it could be X or Y or Z. The failure of probe 2 increases the likelihood of Y, because the decision to go right at node 2 in probe 2 could have been the reason for the probe's failure (but only if node 2 were Y). The probability that node 3 of probe 3 is good given that its parent is good and probe 2 failed is $g_{3,2}$. Treating the four node classes of Figure 3.3 as predicates (subscripting them with the coordinates of the parent node), we can see from Figure 3.4 that,

$$g_{3,2} = \operatorname{pr}(good_{3,3} \mid good_{3,2} \land \neg redundant_3) \tag{3.12}$$

$$= \operatorname{pr}(X_{2,2} \vee Y_{2,2} \mid X_{2,2}Z_{2,3} \vee Y_{2,2}W_{2,3} \vee Z_{2,2}Z_{2,3}) \tag{3.13}$$

Applying Bayes' Rule,

$$g_{3,2} = \frac{\operatorname{pr}(X_{2,2}Z_{2,3} \vee Y_{2,2}W_{2,3})}{\operatorname{pr}(X_{2,2}Z_{2,3} \vee Y_{2,2}W_{2,3} \vee Z_{2,2}Z_{2,3})}$$
(3.14)

Since $\neg W_{2,2}$ is entailed by all of the events, the ratio of the probabilities of the events conditioned on $\neg W_{2,2}$ is the same as the ratio of the unconditional probabilities.⁴

$$g_{3,2} = \frac{\operatorname{pr}(X_{2,2}Z_{2,3} \mid \neg W_{2,2}) + \operatorname{pr}(Y_{2,2}W_{2,3} \mid \neg W_{2,2})}{\operatorname{pr}(X_{2,2}Z_{2,3} \mid \neg W_{2,2}) + \operatorname{pr}(Y_{2,2}W_{2,3} \mid \neg W_{2,2}) + \operatorname{pr}(Z_{2,2}Z_{2,3} \mid \neg W_{2,2})}$$
(3.15)

Figure 3.4 shows that $pr(X_{2,2}Z_{2,3} \mid \neg W_{2,2}) = pr(Z_{2,3} \mid \neg W_{2,3})pr(X_{2,2} \mid \neg W_{2,2})$. Thus,

 $^{^4}$ When $A \to B$, $\operatorname{pr}(A \mid B) = \operatorname{pr}(A)/\operatorname{pr}(B)$. Thus, when $A_1 \to B$ and $A_2 \to B$, $\operatorname{pr}(A_1)/\operatorname{pr}(A_2) = \operatorname{pr}(A_1 \mid B)/\operatorname{pr}(A_2 \mid B)$.

$$g_{3,2} = \frac{\operatorname{pr}(Z_{2,3} \mid \neg W_{2,3}) \operatorname{pr}(X_{2,2} \mid \neg W_{2,2}) + \operatorname{pr}(Y_{2,2} \mid \neg W_{2,2})}{\operatorname{pr}(Z_{2,3} \mid \neg W_{2,3}) \operatorname{pr}(X_{2,2} \mid \neg W_{2,2}) + \operatorname{pr}(Y_{2,2} \mid \neg W_{2,2}) + \operatorname{pr}(Z_{2,3} \mid \neg W_{2,3}) \operatorname{pr}(Z_{2,2} \mid \neg W_{2,2})} \tag{3.16}$$

Finally, substituting the values from Figure 3.3,

$$g_{3,2} = \frac{(1-p)(1-2m) + (p+2m-1)}{(1-p)(1-2m) + (p+2m-1) + (1-p)^2}$$
$$= \frac{2mp}{2mp + (1-p)^2}$$
(3.17)

Case j = i - 1

The general form of Equation 3.12 for j < i is,

$$g_{i,j} = \operatorname{pr}(X_{i,j} \vee Y_{i,j} \mid (X_{i,j} \vee Y_{i,j} \vee Z_{i,j}) \wedge \neg redundant_i)$$
(3.18)

The condition $\neg redundant_i$ means all the earlier probes failed. We assumed that the failures of probes not containing node j would not affect $g_{i,j}$, so we can replace $\neg redundant_i$ with a symbol F_i whose meaning is "all earlier probes containing node j failed." Dropping the subscripts since they are always i and j,

$$g = \operatorname{pr}(X \vee Y \mid (X \vee Y \vee Z) \wedge F) \tag{3.19}$$

$$= \operatorname{pr}(X \vee Y \mid XF \vee YF \vee ZF) \tag{3.20}$$

Applying Bayes' Rule, as in Equation 3.14,

$$g = \frac{\operatorname{pr}(XF \vee YF)}{\operatorname{pr}(XF \vee YF \vee ZF)}$$
(3.21)

As in Equation 3.15, we can add the condition $\neg W$ since it is entailed by all the events.

$$g = \frac{\operatorname{pr}(XF \vee YF \mid \neg W)}{\operatorname{pr}(XF \vee YF \vee ZF \mid \neg W)}$$
(3.22)

$$= \frac{\operatorname{pr}(XF|\neg W) + \operatorname{pr}(YF|\neg W)}{\operatorname{pr}(XF|\neg W) + \operatorname{pr}(YF|\neg W) + \operatorname{pr}(ZF|\neg W)}$$
(3.23)

$$= \frac{\operatorname{pr}(F|X)\operatorname{pr}(X|\neg W) + \operatorname{pr}(F|Y)\operatorname{pr}(Y|\neg W)}{\operatorname{pr}(F|X)\operatorname{pr}(X|\neg W) + \operatorname{pr}(F|Y)\operatorname{pr}(Y|\neg W) + \operatorname{pr}(F|Z)\operatorname{pr}(Z|\neg W)} (3.24)$$

Letting a, b, and c stand for $pr(F \mid X)pr(X \mid \neg W), pr(F \mid Y)pr(Y \mid \neg W)$ and $pr(F \mid Z)pr(Z \mid \neg W),$

$$g = \frac{a+b}{a+b+c} \tag{3.25}$$

When j=i-1 there is only one earlier probe (probe i-1) that contains node j. The chance that it fails, given that node j is X or Z, is $(1-(\operatorname{pr}(X\vee Y\mid \neg W))^{d-j-1})$. If node j is Y, the probe definitely fails. Thus for j=i-1,

$$a = \operatorname{pr}(X \mid \neg W)(1 - (\operatorname{pr}(X \vee Y \mid \neg W))^{d-j-1})$$
 (3.26)

$$b = \operatorname{pr}(Y \mid \neg W) \tag{3.27}$$

$$c = \operatorname{pr}(Z \mid \neg W)(1 - (\operatorname{pr}(X \vee Y \mid \neg W))^{d-j-1})$$
 (3.28)

Substituting the values from Figure 3.3,

For
$$j = i - 1$$
, $g_{i,j} = \frac{(1 - 2m)(1 - p^{d-j-1}) + p + 2m - 1}{(1 - 2m)(1 - p^{d-j-1}) + p + 2m - 1 + (1 - p)(1 - p^{d-j-1})}$

$$(3.29)$$

Case j < i

The general form of $g_{i,j}$ for j < i is the same as Equation 3.25 with different values for a and b. When j = i, the failure of earlier probes is irrelevant to $g_{i,i}$ because none of the earlier probes contains node i, i. When j = i - 1, probe i - 1 is relevant because it contains node i, i - 1. When j = i - 2, both probes i - 1 and i - 2 contain node i, i - 2. In general, for j = i - n the previous n probes are relevant. The earliest of the

n probes is relevant when node i, i-n is X or Z, in which case one of the following d-j-1 decisions in the earliest probe must have been a mistake. The probability of that is $(1-(\operatorname{pr}(X\vee Y\mid \neg W))^{d-j-1})$, the scaling term of c in Equation 3.28. The remaining n-1 previous probes are relevant when node i, i-n is X or Y. We can regard the previous n-1 probes starting from depth j+1 as an independent smaller problem of height d-j-1, setting up a recurrence (see Figure 3.5).

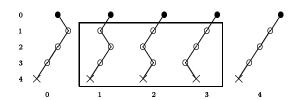


Figure 3.5: For one discrepancy, $a_{3,3}$ is a component of $g_{4,0}$.

For example, the failure of the boxed nodes in Figure 3.5 has probability $1 - a_{3,3}$, a component of the expression for $g_{4,0}$. The probability that all n-1 previous probes fail, given that node i, j is X or Y, is $1 - a_{i-j-1,d-j-1}$. We can thus compute $g_{i,i}$ in the form of Equation 3.25, as follows:

$$\text{For } j < i, \quad g_{i,j} = \frac{a+b}{a+b+c} \tag{3.30}$$

where,

$$a = \operatorname{pr}(X \mid \neg W)(1 - a_{i-j-1,d-j-1})(1 - (\operatorname{pr}(X \vee Y \mid \neg W))^{d-j-1})$$

$$= (1 - 2m)(1 - a_{i-j-1,d-j-1})(1 - p^{d-j-1})$$

$$b = \operatorname{pr}(Y \mid \neg W)(1 - a_{i-j-1,d-j-1})$$
(3.31)

$$b = pr(1 + \neg w)(1 - a_{i-j-1,d-j-1})$$

$$= (p + 2m - 1)(1 - a_{i-j-1,d-j-1})$$
(3.32)

$$c = \operatorname{pr}(Z \mid \neg W)(1 - (\operatorname{pr}(X \vee Y \mid \neg W))^{d-j-1})$$

= $(1-p)(1-p^{d-j-1})$ (3.33)

 $[\]overline{{}^5 ext{If node}}\ i, i-n ext{ is } W ext{ or } Z, ext{ then the } n-1 ext{ probes } can't ext{ succeed}; ext{ they are all bad by depth } 1-n+1.$

Probability of Success in i Probes

We define $a_{0,d} = 0$, observing that the probability of finding a solution in zero probes is zero. The above equations, in combination with Equations 3.10 and 3.11, define $g_{i,j}$ in terms of $a_{i,d}$, p, and m. From here, the definitions of s_i and $a_{i,d}$ follow directly. The probability that a non-redundant probe succeeds, s_i , is the product of the probabilities of each node in the path,

$$s_i = \prod_{j=0}^{d-1} g_{i,j} \tag{3.34}$$

The chance of not finding a solution in i probes or fewer is the chance of not finding a solution in i-1 probes or fewer, $(1-a_{i-1,d})$, times the chance of not finding a solution on next probe given that the earlier probes failed, $(1-s_{i-1})$. Thus the probability of finding a solution in i probes or fewer is,

For
$$d, i > 0$$
, $a_{i,d} = 1 - (1 - a_{i-1,d})(1 - s_{i-1})$ (3.35)

Graphs for $a_{i,d}$ are shown in Section 3.6. For comparison, the curves are drawn beside analogous curves for chronological backtracking and iterative sampling.

3.5.4 Success Probability of Chronological Backtracking

We remarked in Section 2.6.1 that successor ordering heuristics will only have a marginal effect on the overall performance of chronological backtracking. The expected cost for the search is,

$$ProbeCost_{DFS} = 1 + d + m(2^{d+1} - 2 - d)$$
 (3.36)

The mistake probability, m, is only a constant factor in the equation. Replacing m with (1-p) in the equation only reduces the cost to (1-p)/m times the original exponential number of nodes, and the heuristic successor ordering has even less of an effect than such a replacement.

We would like to compare the probability of success over time against that of limited discrepancy search. This can be calculated for backtracking with the following recurrence:

$$s_{t,d} = ps_{t-1,d-1} + (1-p)s_{t-2^d,d-1}$$
(3.37)

The probability of finding a solution in a tree of height d in no more than t nodes is p times the probability of finding a solution in the left subtree with one fewer nodes plus (1-p) times the probability for the right subtree with t minus the number of nodes in the left subtree and the parent.⁶ We take $s_{t,d} = 0$ for $t \leq 0$, and $s_{t,0} = 1$ for t > 0.

This calculation is measured in number of nodes examined, whereas the LDS calculations are measured in number of probes, each probe potentially d + 1 nodes counting the root.⁷ To compensate for the difference, we give backtracking d nodes per probe plus one.

3.5.5 Success Probability of Iterative Sampling

Since iterative sampling samples with replacement, the probability that all i probes fail is the product of the probabilities that the individual probes fail. A probe fails with the likelihood of $1-(1-m)^d$. Thus the success probability for iterative sampling with i samples is,

$$s_{\text{isamp}} = 1 - (1 - (1 - m)^d)^i$$
 (3.38)

3.6 Comparison with Existing Techniques

Figures 3.6 and 3.7 show the theoretical success probability curves for iterative sampling (isamp), chronological backtracking (DFS), and limited discrepancy search (LDS) for various heuristic probabilities, p. The graphs show the probability of

⁶With probability (1-p) the left child is a mistake, in which case the search wastes $2^d - 1$ nodes examining the left subtree first (plus 1 for the parent).

⁷The number is somewhat smaller on average because LDS does not restart from the root node on each probe. The relative advantage that chronological backtracking draws from the few extra nodes is negligible, a fact that is obvious from the graph.

finding a solution in some number of probes, i. The DFS curve is given d nodes per probe plus one. It is also given the highest of the heuristic probabilities shown in each figure.

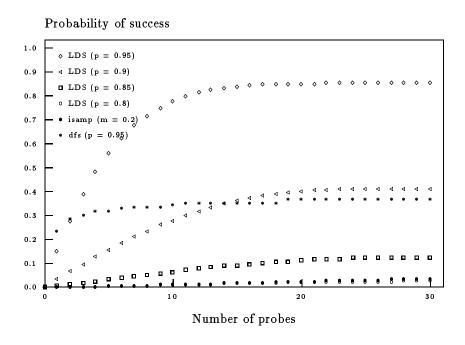


Figure 3.6: A problem of height 30.

Figure 3.6 is a problem of height 30, with a mistake probability m of 0.2. The problem has about a billion fringe nodes of which a few more than a million are goals.⁸ With a solution density of 1/1000, we would expect iterative sampling to sample about 500 fringe nodes before finding a solution (807, to be exact).⁹ By many accounts, a problem with a solution density of 0.001 is fairly easy problem. It takes only 807(30) = 24,210 nodes, on average, to find a solution using iterative sampling. The expected number of probes is even slightly more than the number of probes required to find a solution with a 50 percent probability, 560(30) = 16,800.¹⁰

⁸The number of goals is $(2-2m)^d$.

⁹The expected number of probes is $1/(1-m)^d$.

¹⁰For example, the expected number of coin flips before getting heads is two, whereas it takes only a single coin flip to have a 50 percent probability of getting heads. The number of probes required to achieve a given success probability s is, $\frac{\log(1-s)}{(1-(1-m)^d)\log(1-(1-m)^d)}$.

In practice we may be interested in the number of nodes required to find a solution with a higher probability of success. The number of nodes required by iterative sampling for an 80 percent success probability on this problem is 1300(30) = 39,000. Compare this to the performance of limited discrepancy search. For p = 0.95, LDS has an 80 percent probability with just eleven probes, or 990 nodes. The savings, nearly a factor of forty, depends on the heuristic to order successors correctly seven out of eight times that one of the successors is a mistake.

For p=0.8=1-m, the heuristic orders the successors correctly half the time, no better than random selection. The p=0.8 curve in the figure shows that the performance of LDS is slightly worse than iterative sampling under these conditions. For p=0.85,0.9, and 0.95 the heuristic orders nodes correctly five, six, and seven out of eight times. The curves show that the expected performance of LDS increases dramatically with the better p.

The DFS curve for p = 0.95 rises only marginally above the probability that its first fringe node is a goal, 0.21.¹¹ The futility of DFS is even more clear in the deeper search shown in Figure 3.7.

The problem of Figure 3.7 has height 100, and quite a few nodes. The density of solutions for m=0.1 is about 2.6×10^{-5} . Iterative sampling needs 26,096 probes, or 2.6 million nodes to have a 50 percent chance of success. If, as in the earlier problem, the heuristic orders nodes correctly seven out of eight times (p=0.975 for m=0.1), LDS has a 50 percent chance with just twenty probes (2,000 nodes), a three order of magnitude savings over iterative sampling. The savings is also about a factor of 1000 for a success probability of 80 percent.

Beyond 80 percent, the three orders of magnitude savings is more doubtful, though perhaps not as doubtful as the graph seems to suggest. The one discrepancy iteration ends after 101 probes. The later probes of the one discrepancy iteration have much of their path in common (see Figure 3.4), so the likelihood that one of these later probes succeeds given that the others failed is small. After 101 probes, though, the two discrepancy iteration begins to explore "fresh" paths again. Consequently, we would expect the LDS curve to rise more steadily again where the graph leaves off.

¹¹The probability that the first fringe node examined by DFS is a goal is simply p^d .

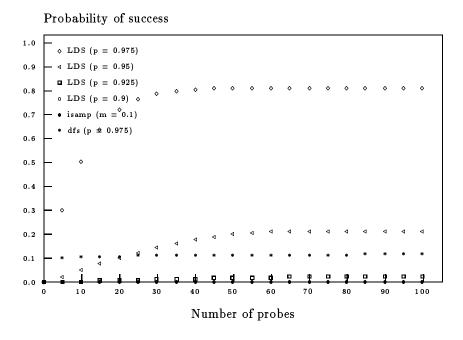


Figure 3.7: A problem of height 100.

3.7 Variations and Extensions

The reason we have focused on analyzing the early iterations of limited discrepancy search is that we believe in practice they are the only iterations that matter. Earlier, we argued on mathematical grounds that they would be more important than the later iterations. We will now take the position that in practice the later iterations don't matter at all. The reason is that if your objective is to maximize the probability of finding a solution in a given number of nodes, there are always more promising things to do than use those nodes on later iterations of limited discrepancy search.

This section discusses a few of the more promising things to do. Since some involve combinations with other techniques and others depend on search space properties that are difficult to quantify, this discussion will be less formal than the earlier sections. From here, we will view limited discrepancy search as a tool that can be used in combination with other techniques to craft an effective search procedure for a given real world problem.

3.7.1 Variable Reordering

Constraint satisfaction problems and SAT problems are formulated as tree search by fixing an order for the variables to be instantiated or determining the order dynamically as the search progresses. In either case, a node in the search tree is a choice point for the possible instantiations of a particular variable. If an effective heuristic does not solve the problem with a limit of one discrepancy for some chosen variable order, it may still solve the problem with one discrepancy given a different variable order. The "wrong turn" instantiations that the heuristic makes on the first variable order may even follow from unit propagation on the second. This suggests the simple technique of repeating the one discrepancy iteration of LDS with different variable orders. When variable order is determined dynamically, it may suffice just to begin the search with a different variable on each iteration. We present related experimental results in Section 4.8.2.

3.7.2 Using Different Heuristics

If you have alternative heuristics, you can try repeating the one or two discrepancy limit iterations of LDS with the different heuristics. If one heuristic is unlucky and makes more than two wrong turns on a given problem, some other heuristic may be luckier. In general, what is hard for one heuristic may not be hard for another. LDS is an effective way to give one heuristic a reasonable chance before switching to a secondary heuristic.

3.7.3 Combining LDS with Bounded Backtrack Search

LDS can easily be combined with bounded backtrack search to produce an algorithm that does not count discrepancies that fail quickly toward the discrepancy limit. The combined algorithm, an alternative to LDS of Figure 3.1, is given in Figure 3.8.

We have found experimentally that the LDS-BBS algorithm outperforms both LDS and BBS on job shop scheduling problems (see Section 4.4). Indeed, for scheduling LDS-BBS appears to be the algorithm of choice among all nonsystematic backtracking strategies. There is a compelling theoretical argument for why it performs

```
LDS-BBS-PROBE (node, k, lookahead)
       if GOAL-P(node)
  1
  2
             return \langle node, 0 \rangle
  3
       s \leftarrow \text{SUCCESSORS}(node)
  4
      if k > 0
  5
             s \leftarrow \text{REVERSE}(s)
  6
      i \leftarrow 0
  7
       branch count \leftarrow 0
       maxheight \leftarrow 0
  8
       for child in s
  9
  10
            if k = 0 and branchcount \geq 1
  11
                  break
            if k \geq 0 and i = 0
  12
                  k' \leftarrow k-1
  13
  14
             else k' \leftarrow k
  15
             \langle result, height \rangle \leftarrow LDS-BBS-PROBE(child, k', lookahead)
  16
             maxheight \leftarrow Max(maxheight, 1 + height)
  17
            if result \neq NIL
  18
                  return \langle result, 0 \rangle
  19
            i \leftarrow i + 1
  20
            if height \geq lookahead
  21
                  branchcount \leftarrow branchcount + 1
  22 return \langle NIL, maxheight \rangle
LDS-BBS(node, lookahead)
       for x \leftarrow 0 to maximum depth
  1
  2
             \langle result, height \rangle \leftarrow \text{LDS-BBS-PROBE}(node, x, lookahead)
  3
            if result \neq NIL
  4
                  return result
  5
       return NIL
```

Figure 3.8: Limited discrepancy search with bounded backtrack.

better than LDS alone. In Chapter 2, we found that many mistakes resulted in quick, if not immediate, failures. If a heuristic makes few wrong turns to begin with, it makes even fewer wrong turns that exceed the backtrack bound. Adding a bounded backtrack enables limited discrepancy search to discover solutions with a discrepancy limit of no more than the number of wrong turns that exceed the backtrack bound, potentially reducing the number of required iterations. Since the cost of each LDS iteration grows by a factor of d, the savings can be substantial. The added cost of the backtrack bound is relatively insignificant. Adding a backtrack bound of one node can cost at most a factor of 2. A backtrack bound of l costs at most a factor 2^{l} . The upper bound is very conservative since the heuristic, by assumption, makes few mistakes.

3.7.4 Local Optimization Using LDS

For problems like scheduling, LDS can be used to search the neighborhood of an existing solution. The one discrepancy iteration of LDS is modified to begin following the path of the previous best solution instead of following the heuristic. At the depth of the discrepancy, the algorithm diverges from the previous solution and follows the heuristic for the remaining decisions. If the path ends in a solution that is better than the previous best, it can be adopted immediately or stored as a contender for the basis of the next iteration.

This variation of LDS requires some measure of the "goodness" of a solution. For scheduling problems, the schedule length is often the appropriate measure. Searching for a schedule that takes less than time L, if successful, produces a schedule that takes time L'. A set of standard LDS iterations can be repeated with the lower time bound L', or the optimization variant of LDS can be applied to consider variations of the previous schedule that differ by at least one discrepancy.¹³

¹²This cost factor is per node. Paths can also be longer.

¹³Alternatively, the time bound can be adjusted by binary division. A single iteration of LDS, though, is not a decision procedure, so failure to find a schedule for a given time bound is no proof that no such schedule exists.

3.8 Conclusion

Limited discrepancy search is an effective way to make use of heuristic information in tree search problems. Our own experiments (see Section 4.4) and the experiments of other researchers [8] have shown that chronological backtracking is not. In our analysis, we have attempted to offer an explanation why.

Chapter 4 presents experimental findings that seem to corroborate our theoretical expectations. The experimentation was based on general CSP heuristics applied to job shop scheduling problems. Even using these weak heuristics, variations of limited discrepancy search were among the best of all algorithms tested. Since LDS is more sensitive than other techniques to the accuracy of the heuristics, we expect that improving the heuristics would have a greater effect on LDS than it would on the alternatives. Consequently, we are very optimistic about the applicability of LDS to solving real world problems for which heuristics can be tuned to a high degree of accuracy.

Chapter 4

Experimental Results

4.1 Introduction

The body of our experimental work is based on job shop scheduling problems. We chose this domain for a number of reasons. Scheduling is a problem of obvious practical significance, but it is also a problem that has been studied in the research communities of both Artificial Intelligence and Operations Research. Known to be NP-Complete, job shop scheduling is a representative of a large class of problems on an important frontier of tractability.

We hope that the algorithms presented in this thesis apply to many useful domains. Certainly, the intuitions behind our theoretical arguments seem to be broadly applicable. We have analyzed the search space properties that make the algorithms efficient, and we have given formulas for computing the expected performance of the algorithms based on these properties.

In this chapter, we put the theoretical arguments to test. Prior to conducting these experiments, we profiled the search spaces of small job shop scheduling problems to ascertain that the theoretical properties actually do exist. The results of these profiles for job shop scheduling, and for comparison Random 3SAT, are given in Appendices B, C and D. Indeed, the scheduling domain does appear to have the necessary properties. To evaluate the performance of the algorithms, we chose an established benchmark of job shop scheduling problems. We ran many variations of each algorithm in hopes of both measuring the algorithms' best efforts and determining how sensitive they were to particular choices for their parameters.

We report our findings beginning with an evaluation of the performance of the nonsystematic backtracking strategies. We compare our results with existing AI search techniques and OR scheduling programs. Then in Section 4.6 we investigate the algorithms individually in more detail. Finally, we report on topics relating to heuristics that affect all of the algorithms.

4.2 Scheduling as a CSP

Job shop scheduling is the well known NP-Complete problem of finding a schedule for a set of nm operations of various durations on m machines subject to a set of sequence constraints and a bound on the overall length of the schedule [20]. The operations are organized as n sequences, or jobs, of m operations each. Each of the m operations of a job is assigned to a different one of the m machines. Thus each of the m machines has an unordered set of n operations to be performed on it, one operation from each job. The desired schedule is a set of start times for the operations satisfying the conditions that (1) no two operations on the same machine overlap, (2) the operations of a job are performed in the given sequence without overlap, and (3) all operations complete in the time allowed by the bound on the schedule length.

The problem can be formulated as a constraint satisfaction problem of finding a total order for the n operations on each machine while still satisfying the sequence constraints of the jobs. A total order of n operations can be derived from n(n-1)/2 binary decisions among pairs of operations. For each pair of distinct operations $\langle a_i, a_j \rangle$ on the same machine, either a_i must precede a_j or a_j must precede a_i . Since there are m machines, there are mn(n-1)/2 such decisions. In the CSP formulation, these ordering decisions are variables. Their domain is $\{<,>\}$.

A constraint satisfaction problem can be solved as tree search by branching on the possible values of the variables in either a predetermined or dynamic order. Most CSP implementations of job shop scheduling do some amount of constraint reasoning to propagate the effects of ordering decisions. Typically, many of the ordering decisions are forced by the effects of others, resulting in far fewer than mn(n-1)/2 real choice points. In our tree search formulation, variable instantiations that follow from constraint reasoning do not constitute nodes. Thus a path from the root of the tree to a terminal node consists of a sequence of no more than mn(n-1)/2 binary decisions ending in a node that has no children. Since paths can consist of different numbers

¹The problem statement for the NP-completeness proof in Garey and Johnson [20] allows jobs to have fewer than m operations and only stipulates that no two adjacent operations of a job use the same machine. For ease of representation, though, many job shop benchmarks have m operations per job, one per machine [33, 47].

of choice points, the tree is not necessarily of uniform depth.

The order in which variables are instantiated and values assigned can have a dramatic effect on the size of the search trees and efficacy of the search algorithms. For our experiments, we have applied standard CSP heuristics to the domain [26]. These heuristics are weak by scheduling standards, but they provide a clear benchmark for measuring the relative performance of the search algorithms.

4.3 Measuring Performance

There are a number of ways to evaluate the performance of search algorithms on scheduling problems. We have described job shop scheduling as a satisficing problem: find a schedule that meets a certain set of criteria. One evaluation method for satisficing problems is to compare how long it takes an algorithm to find solutions to all of the problems in a given benchmark. But what if an algorithm cannot solve one of the problems in the amount of time you are willing to wait? Does that algorithm lose? If losing is too strong a penalty, how much do you penalize the algorithm relative to others that did solve all the problems?

Another evaluation method is to compare the number of problems solved in a given time limit per problem. Scheduling problems are typically fairly easy for a search algorithm to solve, or extremely hard [26]. Consequently, the relative performance of search algorithms is not particularly sensitive to the time limit chosen; for a given problem, a reasonable time limit is usually easily enough or not nearly enough. For a benchmark consisting of a large set of problems, the "scores" of the search algorithms are good indications of their performance.²

The scheduling domain has a third evaluation method that is arguably more representative of an algorithm's utility for solving real world problems. In the real world, scheduling is often an optimization problem: find a schedule of minimum length. The performance of search algorithms on scheduling problems can be measured by the lengths of the schedules they are able to find in a given amount of time. Following

²We reported early results [26] by graphing the time required to solve x percent of the problems. This curve, the transpose of the cumulative distribution function [27], demonstrated that the "scores" method was quite reasonable for scheduling problems.

Vaessens, Aarts and Lenstra [49], we compute a schedule's percent above optimal length, e^3 . An optimizing search algorithm produces shorter schedules given more run time. The function e_t for a search algorithm is taken to be the percent above optimal length of the best schedule it can find in fewer than t nodes. For a benchmark that has more than one problem, \bar{e}_t is the average of e_t for the problems in the benchmark. In keeping with Vaessens et al. we will call \bar{e} the mean relative error, or MRE. Among the evaluation criteria we considered, the MRE curves appear to be the clearest measurement of performance.

Formulated as a CSP, though, scheduling is not an optimization problem. To transform the satisficing problem into an optimization problem, we run the search algorithms iteratively, decreasing the bound on the schedule length each time to slightly less than the length of the last schedule found. At any point in time, the algorithm's best schedule is the basis for computing e_t .

The job shop scheduling benchmark that we used consisted of thirteen problems taken from the Operations Research literature [33, 47, 49].⁴ The problems varied in size from 10×10 to 15×15 or 20×10 jobs and machines. Although the depths of the search trees varied, most were in the range of 100 to 800 choice points.

4.4 Comparison with AI Techniques

Figure 4.1 shows the performance of the basic nonsystematic backtracking strategies that we have discussed in this thesis: bounded backtrack (BBS), limited discrepancy (LDS), depth first search with restarts (RDFS), iterative broadening (SIB), and the combination of LDS with BBS (LDS-BBS). We compare their performance with the performance of iterative sampling and chronological backtracking (DFS). The figure clearly demonstrates the effectiveness of the nonsystematic strategies. In 500,000 nodes, iterative sampling achieves an MRE of 27.9 percent, chronological backtracking 15.5 percent. All of the nonsystematic backtracking strategies are in the 3 to 5 percent range. With the exception of iterative broadening, the nonsystematic algorithms

³The optimal length is taken to be the length of the best reported schedule for the problem as of November 1994.

⁴All of the problems are available electronically by sending a message to o.rlibrary@ic.ac.uk.

appear to be fairly predictable.

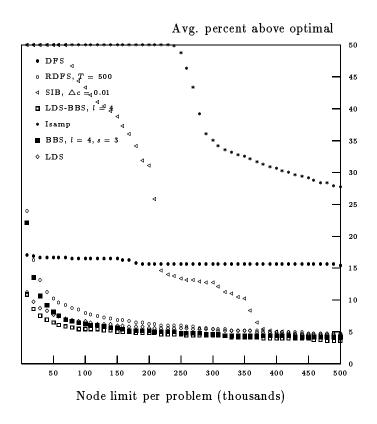


Figure 4.1: Iterative sampling and chronological backtracking are not competitive.

It is interesting to note that chronological backtracking makes almost no progress after a short amount of time. This corroborates our theoretical predictions about the performance of chronological backtracking. Other researchers have observed the same phenomenon and responded by writing backtrack free tree search algorithms, "onesamp" in our terminology. What we have shown here is that while chronological backtracking may not be an effective use of run time, other nonsystematic backtracking strategies are.

Figure 4.2 compares the performance of the various nonsystematic backtracking strategies. While the difference between 3 and 5 percent may seem small in Figure 4.1, it is quite large in the context of just the better algorithms.

We discuss the five basic nonsystematic strategies individually in the later sections of this chapter. Each of the strategies has a number of variations and parameters

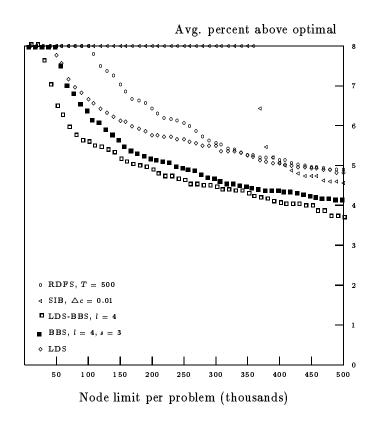


Figure 4.2: Front runners.

that affect the search. In this graph, we have chosen good representatives for all of the strategies. We should point out, though, that for all five basic algorithms, the performance is not dramatically affected by different choices for the parameters, so long as they are reasonable. One could imagine that an algorithm might perform well on a set of problems only when its parameters are tuned to "magic" values that are dramatically better than other values. This appears not to be the case for any of these algorithms.

4.5 Comparison with OR Techniques

The thirteen problems of our benchmark were not picked at random. A recent survey of the Operations Research results in job shop scheduling uses this exact set of thirteen

problems [49]. By taking the same benchmark, we can compare our results against the results of a large community of researchers. The authors, programs, and MRE ratings from this survey are reproduced in Figure 4.3.

Author	Program	MRE
NowickiS[38]	TSAB	0.45
Dell'AT [17]	TS3	0.92
$\operatorname{BarnesC}[5]$	TS2	1.35
AppleCook[3]	Shuffle2	2.05
VLaarAL[31]	SA	2.17
MatsuoSS[35]	CSSA	2.31
NuijtAVV[39]	RCS	2.31
${ m AppleCook}[3]$	Shuffle1	3.15
${ m AppleCook}[3]$	Bottle-6	3.22
DornPesch[18]	SB-GA(60)	3.38
${ m ArtsVLU}[1]$	SA1	3.5
${ m ArtsVLU}[1]$	SA2	3.54
DornPesch[18]	SB-GA(40)	3.65
${ m AdamsBZ}[2]$	SBII	3.75
${f BalasLV[4]}$	SB4	3.98
${ m AppleCook}[3]$	Bottle-5	4.14
${ m ArtsVLU}[1]$	GLS2	4.59
AppleCook[3]	Bottle-4	4.88
${ m BalasLV}[4]$	SB3	5.01
${ m ArtsVLU}[1]$	GLS1	5.05
$\mathrm{DellaTV}[12]$	GA2	5.15
${ m ArtsVLU}[1]$	TA1	7.83
DornPesch[18]	P-GA	8.13
${ m AdamsBZ}[2]$	SBI	8.31

Figure 4.3: OR results from 1994 survey on the same benchmark.

Since this is not a scheduling thesis, our objective has not been to write a scheduler to compete with the programs reported in this survey. Still, it is important that the problems we use to test our search algorithms are representative of problems people care about. Our goal has been to achieve respectable performance on an established benchmark solely through advances in search technology.

Our best results are in the 3 to 5 percent MRE range. Although this performance

is not as good as the best OR scheduling results, it is respectable even by the standards of the dedicated scheduling programs.

The scheduling programs of the OR survey fit into a few broad categories: local search techniques, genetic algorithms, constraint satisfaction, and neural networks. The best reported results come from a recent local search technique known as "taboo search," [48] but as the authors of the survey point out, "these benefits come at the expense of a non-trivial amount of testing and tuning." The best representative in the survey of a constraint satisfaction program is the RCS program ("randomized constraint satisfaction"). Incorporating bottleneck reasoning derived from Sadeh's work [43] with better heuristics and restarts, RCS has extremely strong performance using a backtrack free tree search algorithm. RCS is the type of program that potentially could benefit from using the search results presented in this thesis.

4.6 Performance of the Five Strategies

4.6.1 Stochastic Iterative Broadening

Throughout this thesis, we have referred to our earlier work [24] that introduced the iterative broadening algorithm. Based on the search space property that it is possible to make mistakes that lead to subtrees with no goal nodes (see Chapter 2), our analysis of iterative broadening showed that iteratively searching a tree of uniform branch factor b with an artificial (smaller) branch factor cutoff c that increased by one each iteration could produce orders of magnitude savings in the expected time to find a solution. Since c would eventually reach b on a later iteration, the algorithm was guaranteed to search the entire space and eventually return a solution if one existed.

We adapted the iterative broadening algorithm to binary search trees by using a fractional branch factor. The number of children expanded from a particular node was determined probabilistically. If the branch factor cutoff c were 1.1, then the second child would be expanded with probability 0.1. Likewise, if c were 1.9 the second child would be expanded with probability 0.9. Using this technique, the branch factor cutoff could be increased at any rate on successive iterations. We call this technique

stochastic iterative broadening, or SIB.

We experimented with various rates of increase for the branch factor cutoff, and with various non-increasing fixed values. Our intuition was that there may be some fixed value of c that was optimal for a particular domain. The increasing c value of iterative broadening might quickly pass up the optimal value, causing most of the allowed run time to be spent on later iterations with too large a branch factor. The results are shown in Figure 4.4.

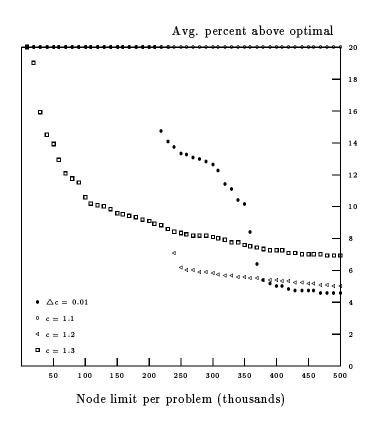


Figure 4.4: Iterative broadening with cutoff c.

We were surprised to learn that increasing c on successive iterations in the standard iterative broadening manner actually outperformed all fixed values of c we considered. Furthermore, SIB was competitive with some of the other more sophisticated backtracking strategies.

It appears from our experiments that the most serious drawback to iterative broadening is that the performance over time is unpredictable. If, for example, our experiment had run only to 250,000 nodes, we might have drawn vastly different conclusions about the optimal branch factor cutoff. The performance of the nonsystematic methods will vary in different domains, and iterative broadening remains a viable alternative. For job shop scheduling, though, the bounded backtrack and limited discrepancy techniques appear to be better suited to the domain.

4.6.2 Backtracking with Restarts

In Chapter 2 we introduced RDFS, a simple modification to chronological backtracking that periodically restarts from the root. The algorithm orders the children of a node randomly so that it does not explore the same portion of the search tree every time it restarts. The performance of RDFS is shown in Figure 4.5.

For a simple algorithm, its performance is extraordinarily good. A well chosen timeout parameter T yields an MRE only a few points over the MREs of much more sophisticated algorithms. The choice for the timeout parameter appears to be important, though. Taking T=500, RDFS achieves approximately the same MRE in 250,000 nodes as taking T=1000 does in 500,000 nodes. A likely explanation is that the extra 500 nodes of an iteration with a 1000 node timeout do not significantly improve the likelihood of the probe's finding a solution.

The performance of RDFS is also fairly sensitive to variable and value ordering variations. See Section 4.8.1 and Section 4.8.2 for a complete discussion of the variations.

We commented in Chapter 2 that RDFS is closely related to bounded backtrack search. We expected its performance to be similar to BBS, and the experiments seem to corroborate this expectation. As we discuss below (see Section 4.6.4), BBS is likely to be more practical for most situations than RDFS. Nevertheless, the strong performance of RDFS emphasizes how easily the early mistakes problem of chronological backtracking can be overcome.

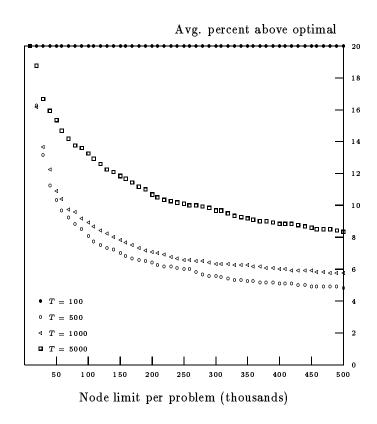


Figure 4.5: Backtracking with restarts, timeout T.

4.6.3 Iterative Sampling

For constraint satisfaction problems, the term "iterative sampling" is loosely applied to a few algorithms that differ in important ways. The choices are whether or not to randomize variable order and whether or not to randomize value order. There are four possible combinations, whose performance is graphed in Figure 4.6. The tree search definition, iterative sampling [32], randomizes values but not variables. The CSP definition, isamp [11], randomizes both values and variables. The backtrack free heuristic algorithm we have been calling onesamp randomizes neither values nor variables. And the last combination, previously unnamed (call it rsamp), randomizes variables but not values.

The most striking result of this experiment is that rsamp—the algorithm that didn't even have a name—has the best performance of the four variations, and by

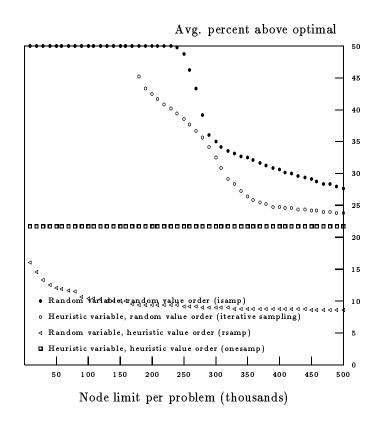


Figure 4.6: Variations of iterative sampling.

a significant margin. Rsamp has probably escaped prior attention as a legitimate algorithm because its properties are difficult to characterize without understanding, in a formal sense, what the variable and value ordering heuristics actually do. It is quite possible, for example, that no amount of run time would reduce the probability of not finding a solution on a solvable problem to some positive ϵ . Nevertheless, the dominant performance of rsamp deserves some notice, if even just an anecdote about the importance of variable ordering (see Section 4.8.2 for related experiments).

It is interesting that heuristic variable order appears to be better than random when values are chosen randomly, although the margin is somewhat less convincing. We can see also that even in 500,000 nodes, neither of the two common variations of the algorithm catches up to the technique of following the single heuristic path (onesamp). We are using fairly weak heuristics for these experiments, so this effect is likely to be amplified in practice by more carefully tuned heuristics.

4.6.4 Multiprobe: Bounded Backtrack Search

In Chapter 2, we gave the bounded backtrack algorithm for binary search trees. There is a straight forward generalization of BBS for trees of arbitrary and not necessarily uniform branch factor. The generalized BBS algorithm, called *Multiprobe*, is reprinted in Figure F.1 for reference. In reference to this implementation, we will use the names BBS and *Multiprobe* interchangeably.

Bounded backtrack can be used in basically two capacities: as a simple way to improve existing algorithms such as isamp, or as a stand alone search algorithm that is competitive on its own merits. Figure 4.7 shows the results of using bounded backtrack in the first capacity.

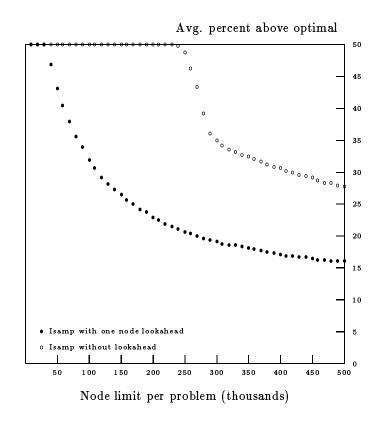


Figure 4.7: Adding bounded backtrack to isamp is a major improvement.

The result of adding just one node of bounded backtrack to isamp is fairly dramatic, reducing the MRE from 27.9 to 16.1. Neither value is competitive with the

other nonsystematic methods or the OR algorithms, but this is hardly surprising. Isamp is a simple algorithm and it uses no heuristics. If it were competitive, one would have to question the progress made by research in the field. The point of the graph is just to show that bounded backtrack is an effective speed-up technique.

In the second capacity, *Multiprobe* can be tuned by making decisions about how much to randomize variable and value order, and what lookahead value l to use as the backtrack bound. Randomizing variable and value order is a decision that affects all of the nonsystematic algorithms, so we consider these issues separately in Section 4.8.1 and Section 4.8.2. Figure 4.8 compares the performance of *Multiprobe* with different lookahead values.

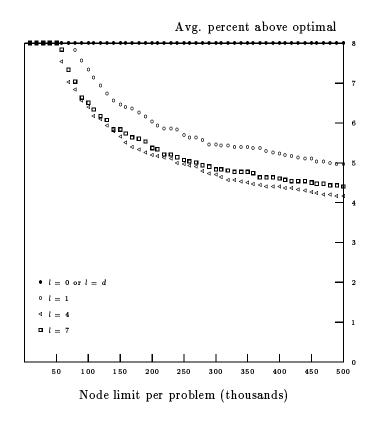


Figure 4.8: Any small lookahead value works—some slightly better than others.

We find that these experiments confirm our theoretical projections that the particular choice of the lookahead parameter is not tremendously significant, within the range of reasonable values.⁵ The practical impact of this is that *Multiprobe* is easy to tune. We have found that trying a few increasing powers of two (1, 2, 4, 8, and 16, for example), is generally sufficient experimentation to get nearly optimal performance. For search trees of nonuniform depth, it is much easier to pick an efficient and reliable backtrack bound for *Multiprobe* than it is to pick a restart timeout for RDFS since the backtrack bound is independent of the length of the solution paths, whereas the restart timeout is not.

4.6.5 Limited Discrepancy Search

Like isamp, limited discrepancy search can be improved by adding a bounded backtrack. As shown in Figure 4.9, even without a bounded backtrack its performance is extremely good. With a lookahead of four nodes, it has the best performance of any of the algorithms tested.

There are strong reasons to believe that variations of LDS also have the most promise in application to real world problems. We commented earlier that the value ordering heuristics for many real world problems are likely to be tuned to a higher degree of accuracy than ours. LDS has the most to gain by such improvement.

The remarkable success of rsamp shows the importance of variable ordering, and in particular the effectiveness of trying different variable orders. We would thus expect running the one-discrepancy iteration of LDS iteratively with different variable orders (but without increasing the discrepancy limit) to have tremendous potential. Limited discrepancy search can be viewed as a way to get a lot of mileage from an accurate value ordering heuristic. The basic idea could be customized or incorporated into many types of search strategies that are appropriate for any given domain.

4.7 Scheduling as SAT

The CSP formulation of the scheduling problem can be further translated into a satisfiability problem by taking the ordering decisions to be propositions in a theory

⁵The extremes are chronological backtracking and iterative sampling, whose performance we are claiming is significantly different.

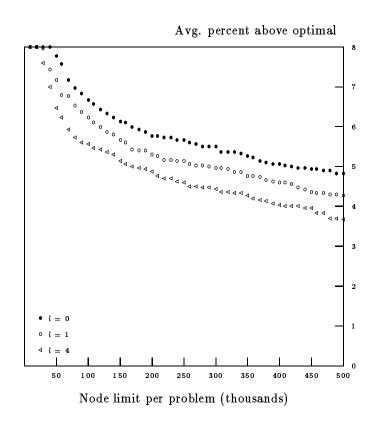


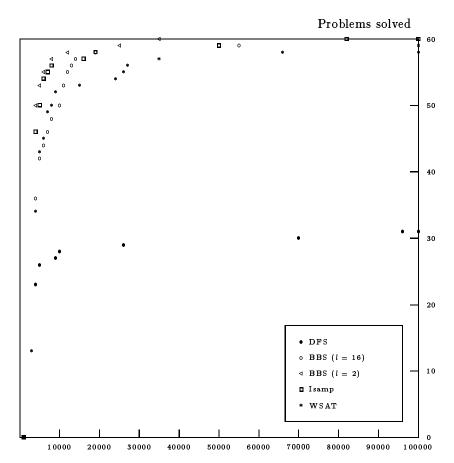
Figure 4.9: Bounded backtrack improves LDS.

that enforces their temporal coherence and the other constraints of the scheduling problem. The details of the translation can be found in Crawford and Baker's report of their experimental results on the applicability of satisfiability algorithms to scheduling [11].

To compare the nonsystematic backtracking strategies with WSAT [11], we wrote a Davis-Putnam [13] style satisfiability engine modeled after Crawford's *Tableau* program [10]. Using *Multiprobe* as the search algorithm of the SAT engine, we were able to compare bounded backtrack search with WSAT on a common problem representation. The results are shown in Figure 4.10.

The translation to SAT yields a much less efficient problem representation,⁶ severely limiting the size of problems we were able to test. We used Sadeh's benchmark of sixty

⁶The expansion factor was about 100 for the problems we tested.



Timeout per problem, in units of nodes for tree search and 100 flips for WSAT

Figure 4.10: Multiprobe and WSAT.

job shop scheduling problems [43]. These problems, each over a megabyte in SAT form, were near the limit of our workstations' capacity after they were converted to the internal representations of the search programs. We evaluated the performance of the programs by graphing the number of problems solved with a given time

⁷Crawford and Baker comment that the size of the SAT problem representation of scheduling problems is of order nd + c, where n is the number of operations, d the number of time points, and c the number of constraints [11]. Sadeh's problems had approximately 100 time points and 50 operations. The problems of our main benchmark had as many as 2000 time points and 200 operations. Assuming the nd term dominates in the calculation, the SAT form of the main benchmark problems would exceed 50 megabytes per problem. The internal representation is an additional super-linear expansion factor.

limit per problem. Figure 4.10 shows the performance of chronological backtracking, isamp, and several bounded backtrack variations in comparison to WSAT. Expanding a node in tree search required approximately the same amount of CPU time as WSAT required for 100 flips, so the curves are scaled appropriately.

Figure 4.10 shows that isamp performs better than WSAT even without the aid of bounded backtrack. The isamp curve corroborates the results of Crawford and Baker [11]. The WSAT curve is from Crawford and Baker. Adding a small bounded backtrack to isamp appears to be a small gain, although a large bounded backtrack appears to be a net loss. The two node lookahead version of BBS solves all the problems in about half the time as isamp. Chronological backtracking, representing the limiting case for a large backtrack bound, yields the disastrous results that echo of earlier experiments and theoretical predictions.

To help understand why adding a bounded backtrack to isamp did not increase the level of performance more significantly, we profiled the search space of the SAT scheduling problems in the same manner that we did originally to understand the search space properties that bounded backtrack exploits. Our results, reported in Appendix C, are not as complete as our other profiles (Appendix B and D) due to the size of the search spaces. It appears, though, that there are few mistake nodes with small subtrees below them. Since these are the only mistakes that bounded backtrack can help isamp to avoid, the potential benefit of a bounded backtrack appears to be limited with this problem representation.

4.8 Variations

4.8.1 Weakening Value Order Heuristics

To avoid following the same search path on every probe, many of the nonsystematic backtracking strategies select the child to expand at random. Unfortunately, selection on a purely random basis nullifies the effect of heuristically sorting the children. We

⁸These data represent the performance of WSAT after unit propagating to quiescence on the original SAT theory and performing several other preprocessing steps. The performance of WSAT on the original theories after direct translation from the CSP form was significantly worse [11].

have seen that sorting the children can be very important for solving hard problems. It seemed likely that there would be a middle ground, some weighted random selection that satisfied the demand for non-redundancy in the search without completely disregarding the advice of the heuristic.

Our scheduling problems were all binary search trees, so the issue involves only deciding which of the two children to expand first. If we always followed the heuristic, we would expand the left child first 100 percent of the time. If we selected randomly, we would expand the left child first about 50 percent of the time. Somewhere in between, we opined, would be a probability that yields better performance than either of the extremes.

Figure 4.11 compares the performance of iterative sampling selecting the heuristic value 100 percent, 87.5 percent, and 50 percent of the time.

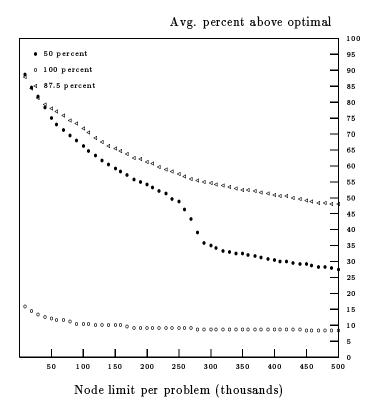


Figure 4.11: Weighted random selection can be worse than the alternatives.

The result of the experiment is very surprising. The curve for weighted random

selection is dramatically worse than both of the two extremes. We repeated the experiment with a Multiprobe implementation that used a four node bounded backtrack and heuristic variable order (iterative sampling uses random variable order). The curves for selecting the heuristic value 100 percent, 93.75 percent, 87.5 percent, and 50 percent of the time are shown in Figure 4.12.

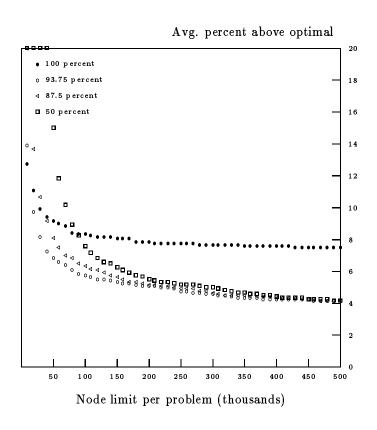


Figure 4.12: Weighted random selection can be better than the alternatives.

At about 4 percent MRE, both intermediate weights perform better than the extremes with the *Multiprobe* implementation. Weighting appears not to be the predictable tuning issue we expected. With respect to the efficacy of intermediate weighting values, Figure 4.11 and Figure 4.12 are opposites. To explain the difference between them, we would need a theoretical model that encompassed the effects of variable ordering on real problems. We suggest this as a topic for further investigation.

⁹See Section F.2 for a discussion of our implementation of weighted random value selection with non-binary branching factors.

4.8.2 Weakening Variable Order Heuristics

We have observed in earlier experiments [26] that good variable ordering heuristics for scheduling can occasionally produce bad variable orders. Restarting with random variable orders eliminates the problem, but at the same time does not exploit the heuristic. The nonsystematic techniques ought to be able to take partial advantage of the heuristic without committing to any particular order. The same reasoning applied to value ordering was refuted by the experiments of the last section, though, so we conducted a similar experiment for variable ordering.

In many domains, the entire path of variable instantiations can be largely determined by the selection of the first variable. Starting the probes of the nonsystematic algorithms on random first variables avoids the problem of pathological variable orders, but sacrificing the heuristic even this much could be a net loss. Figure 4.13 shows the result of "walk start" variable selection, which we describe in detail in Section F.3. The search algorithm is iterative sampling with random value order.

The figure shows that the performance of walk start variable selection is about the same as heuristic variable order for this search algorithm, and random order is worse than either alternative. The efficacy is likely to depend on the formulation of the problem, but this experiment is an indication that this type of strategy will work. Unlike the value ordering experiments, these experiments had the predicted results.

4.8.3 Incremental Systematicity

In the spirit of iterative broadening, we considered an algorithm that increased the backtrack bound by one level each probe. In d probes, the bound would reach the full height of the tree and the tree would be searched exhaustively. Thus, the modified algorithm could report that there were no solutions if it terminated without finding one.

This iterative algorithm was seriously flawed, though, since successive probes took exponentially longer to complete. Instead of combining the best elements of systematic and nonsystematic techniques, it seemed to combine the worst: it ignored the value ordering heuristic and it exhausted the run time exploring paths that failed for

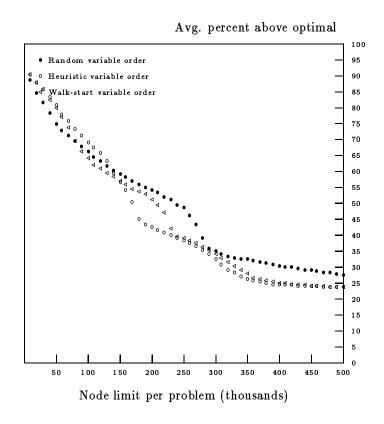


Figure 4.13: Randomize variable order—but not too much.

the same reason.

A much better modification was suggested by Jonsson.¹⁰ The early decisions could be spread out systematically by starting successive probes from the "fringe" nodes of a depth limited search of the tree to depth k. Once we have tried one probe from each node at depth k in the tree, we increase k and start the next probe from the first node in the tree at depth k+1.

For example, beginning with k = 0, the first probe starts at the root of the tree. Then we increase k to one. In a full binary tree there are two nodes at depth one, so the second and third probes start from them. We then increase k to two and start the next four probes from the nodes at depth two. When k eventually reaches the height of the tree, the whole tree is searched systematically.

¹⁰Personal communication with Ari Jonsson, November 1994.

This "incremental systematicity" modification to bounded backtrack search gradually enforces that every probe be different. Unlike the first approach, it does not waste time exploring large subtrees, and it has the additional appeal that it enforces systematicity from the top down, ignoring the heuristic first where it is least likely to be accurate. The details of the algorithm are given in Section F.

Figure 4.14 shows the performance cost of the incremental systematicity modification.

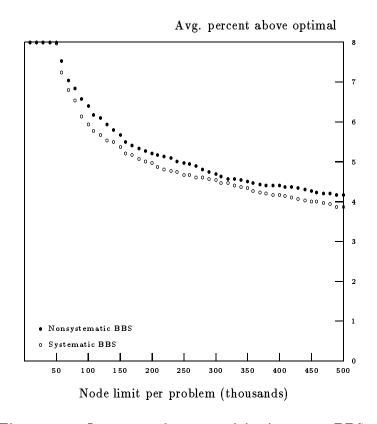


Figure 4.14: Incremental systematicity improves BBS.

We expected that the modification would guarantee completeness at a small cost. We were surprised to find that it did so at no cost whatsoever. In fact, the modification actually improved the average case performance slightly. We repeated the experiment for iterative sampling. The results are shown in Figure 4.15.

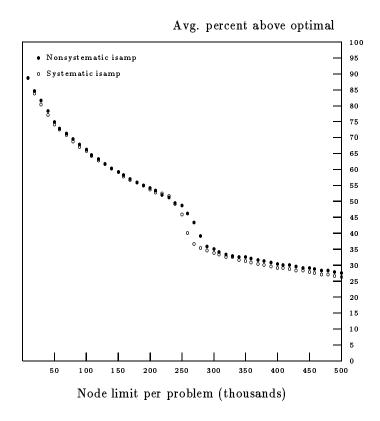


Figure 4.15: Incremental systematicity with iterative sampling.

4.9 Conclusion

The evaluation of the algorithms shows clearly that chronological backtracking is not an effective use of computational resources for problems like job shop scheduling for which tree search has the possibility of making early mistakes. The more the success of the search is dependent upon heuristics, though, the less attractive iterative sampling is as an alternative. As discouraging as the progress of depth first search looks, its overall performance is still better than iterative sampling on the problems we tested.

We find that the problem of early mistakes is surprisingly easy to overcome with nonsystematic methods that retain some of the benefits of the heuristics. The most striking example, restarting depth first search periodically with random value orders, yields a dramatic improvement in rate of progress and overall performance with only a small change to the original algorithm.

The best levels of performance, though, appear to require more sophisticated algorithms. Variations of bounded backtrack and limited discrepancy search scored between 3.5 and 4.5 percent above optimal in our benchmark, a step above the other nonsystematic methods and a very large step above iterative sampling and chronological backtracking. The combination of limited discrepancy with bounded backtrack has the best overall performance of any of the algorithms tested.

The problems of the benchmark, while large by contemporary research standards, are not considered large relative to the types and sizes of scheduling problems it would be useful to solve in the real world. Because of the complexity of scheduling, it is likely that the challenge of scaling up from research problems to real world problems will be met more quickly by advances in heuristics than by the evolution of brute force methods. We expect that in the fullness of time, techniques that depend on heuristics yet recover gracefully with search when the heuristics fail will win out over techniques whose ability to search depends on ignoring heuristic advice.

Chapter 5

Conclusions

From the beginning, my work has been motivated by the challenges of solving real problems. Dating back to 1989, I developed iterative broadening to improve the performance of search algorithms for crossword puzzle generation. Ironically, it was again with crossword puzzle generation that I discovered bounded backtrack search five years later. Iterative sampling had shaken my belief in the heuristic importance of systematicity, freeing me to investigate nonsystematic backtracking strategies for the first time.

By early 1994, though, I was still investigating systematic algorithms. I had made minor algorithmic improvements to backjumping and was attempting to gather experimental data to show the significance. One after another, my experiments were disappointments. On large search spaces, it appeared to be hard to improve search efficiency by ruling out provably bad parts of the space. The results were not enough to draw scientific conclusions, but they were enough to shape the course of my future research.

As soon as I began exploring nonsystematic algorithms, I started to make measurable progress on real problems again. I noticed that modifying iterative sampling to spend more time on the fringe of the tree improved its efficiency. In related experiments, I found that periodically restarting depth first search with different variable orders virtually eliminated its problem with "early mistakes." Incrementally, the experiments led to the discovery of bounded backtrack search. The performance results were astounding. With a small dictionary, my Lisp implementation of bounded backtrack search could generate 6×6 blank crosswords. With a large dictionary (containing more than a few dubious words), I generated a 7×7 , the largest computer generated blank crossword ever reported.¹

After the success of the crossword experiments, I turned my attention to understanding why the algorithm performed well. Building on the iterative broadening research, I developed a model that had a strong intuitive basis. Although I could now explain bounded backtrack's performance, I had a lingering suspicion that the assumptions were in some way linked to crossword puzzle search. To validate the

 $^{^1}$ Once I could generate New York Times 15×15 puzzles quite easily, I turned to blank crosswords for more challenging benchmarks.

results experimentally, I needed a benchmark from another domain.

I chose to use an established benchmark of scheduling problems from Operations Research. This enabled me to compare my results not only against Artificial Intelligence search techniques but also against dedicated scheduling programs. The scheduling experiments did show conclusively that bounded backtrack search was an improvement over alternatives. However, they also revealed a weakness in the technique. Scheduling is a domain that has very strong value ordering heuristics. Bounded backtrack search depends on ignoring the heuristic to search new parts of the space. In these experiments, the merits of the algorithm more than compensated for the loss of the heuristic, but in larger problems the heuristics were likely to become more important. Fundamentally, I thought, there ought to a way to make better use of value ordering heuristics.

These thoughts motivated the development of limited discrepancy search. I found that even with relatively weak heuristics (by scheduling standards), limited discrepancy search was competitive with the best search algorithms tested. Since the technique makes good use of value ordering heuristics, I expected its performance to improve relative to other techniques on larger problems. I also experimented with combining limited discrepancy and bounded backtrack search. The combination resulted in the best of all algorithms tested, leading me to explore other combinations and variations of the algorithms. Some of the experiments had unexpected results.

In the course of this work, I discovered a clever way to make bounded backtrack search guarantee completeness at a small cost of obedience to the heuristic at early stages in the search. With this modification, the algorithm could terminate, proving that a problem had no solutions. I was surprised to discover that in addition to guaranteeing completeness, the modification actually improved the algorithm's average case performance. Other experiments had more unexpected positive results. An unlikely variation of iterative sampling that used heuristic value ordering, for example, turned out to have remarkably good performance.

Productive lines of research tend not to start and stop as much as they gradually lead into other lines of research. These experiments have been no exception. I have shown that nonsystematic backtracking strategies are eminently practical, but the

experiments have also revealed facets of searching large problems that my theoretical model and tools of analysis are too blunt to explain. Search spaces of real problems are not random. If we are to solve large problems, we need to acknowledge that they are not random, and we need to seek to understand their characteristics.

Appendix A

Related Algorithms

Chronological backtracking [50], iterative sampling [32], and iterative broadening [24] are all closely related bounded backtrack search and limited discrepancy search. We give a pseudocode description of the algorithms here for reference.

In our notation, a node is a branch point for the possible values of an unassigned variable in a CSP. The possible values are those values from the domain of the variable that are not known to be inconsistent with the constraints of the problem.

A.1 Chronological Backtracking

Chronological backtracking, or depth first search (DFS), systematically searches all nodes in the tree if run to completion. However, if the algorithm is given a limited amount of time to run, it may return with a timeout signal in lieu of a solution (Figure A.1).

```
DFS(node)
      if timeout
 1
 2
          return timeout-result
 3
     if GOAL-P(node)
 4
          return node
 5
     for child in Successors (node)
 6
          result \leftarrow DFS(child)
 7
          if result \neq NIL
 8
              return result
 9
      return NIL
```

Figure A.1: Depth first search with a time limit.

In line 5, the **for in** loop sequentially assigns *child* the elements of the successors list, Successors (node). A node representing a consistent total assignment is recognized by the function Goal-P(node).

A.2 Iterative Sampling

Iterative sampling follows a random path from the root of the tree to a node on the fringe. The algorithm terminates when it finds a solution or reaches the time limit. For comparison to the other algorithms, we will give a recursive implementation of the algorithm (Figure A.2).

```
IS-PROBE (node)
     if timeout
 1
 2
         return timeout-result
 3
     if GOAL-P(node)
 4
         return node
     if Null-P(Successors(node))
 5
 6
         return NIL
     else return IS-PROBE(SELECT-RANDOMLY(SUCCESSORS(node)))
ITERATIVE-SAMPLING (node)
 1
     loop
 2
          result \leftarrow \text{IS-Probe}(node)
         if result \neq NIL
 3
 4
              return result
```

Figure A.2: Iterative sampling.

In IS-PROBE, line 7 selects an element at random from the successors list. If the successors list is empty, line 6 returns NIL. ITERATIVE-SAMPLING always returns either a solution or a timeout signal.¹

¹In Section 4.8.3 we presented a modification to the algorithm that guarantees an exhaustive search of the tree if run to completion.

A.3 Iterative Broadening

Iterative broadening repeats a restricted depth first search that considers at most c successors of any node. The branch factor cutoff c begins at one and increases by one on successive iterations. For a tree of constant branching factor b, the c^{th} iteration searches the tree exhaustively (Figure A.3).

```
IB-PROBE(node, c)
      if timeout
  1
  2
           return timeout-result
  3
      if GOAL-P(node)
  4
           return node
  5
      j \leftarrow c
  6
      for child in Successors (node)
  7
           result \leftarrow \text{IB-Probe}(child, c)
  8
           if result \neq NIL
  9
                return result
           j \leftarrow j-1
  10
           if i = 0
  11
  12
                return NIL
  13 return NIL
ITERATIVE-BROADENING (node)
  1
      for i \leftarrow 1 to b
  2
           result \leftarrow \text{IB-PROBE}(node, i)
           if result \neq NIL
  3
  4
                return result
  5
      return NIL
```

Figure A.3: Iterative broadening.

Like DFS, ITERATIVE-BROADENING returns a solution, a certificate that there are no solutions, or a timeout signal. Iterative broadening is well suited for tree search problems with large branching factors.²

²In Section 4.6.1 we also adapted the algorithm to binary search trees.

Appendix B

A Profile of Job Shop Scheduling

B.1 Introduction

A number of AI studies on scheduling [11, 43, 45] have used as a benchmark a set of sixty job shop scheduling problems developed by Sadeh for his doctoral thesis [43]. These problems are both small and easy by contemporary standards, yet they have been embraced by the community at large as representative of real world scheduling problems—all reasons which make them ideal candidates to profile in an attempt to illuminate various important characteristics of scheduling search spaces.

The sixty problems consist of six sets of ten samples each, constructed by randomly generating ready times and deadlines according to certain distribution parameters for the sets. The details of how the problem sets differ are not particularly important here since the search space characteristics that we are studying were shared by all of the samples. For reference, though, the names of the sixty problems are given in Figure B.1.

All of the problems consisted of five machines and ten jobs, each job a sequence of five operations to be performed on the five machines in a non-overlapping schedule with the operations of the other jobs. In the CSP formulation of the problem (see Section 4.2), the search tree consists of a series of ordering decisions between jobs on the same machine. For Sadeh's problems, this implies a maximum search depth of 225 decisions. Since many orderings follow from constraint propagation, the effective

no.	filename	no.	filename
0	e0ddr1-10-by-5-1	30	enddr2-10-by-5-1
1	e0ddr1-10-by-5-10	31	enddr2-10-by-5-10
2	e0ddr1-10-by-5-2	32	enddr2-10-by-5-2
3	e0ddr1-10-by-5-3	33	enddr2-10-by-5-3
4	e0ddr1-10-by-5-4	34	enddr2-10-by-5-4
5	e0ddr1-10-by-5-5	35	enddr2-10-by-5-5
6	e0ddr1-10-by-5-6	36	enddr2-10-by-5-6
7	e0ddr1-10-by-5-7	37	enddr2-10-by-5-7
8	e0ddr1-10-by-5-8	38	enddr2-10-by-5-8
9	e0ddr1-10-by-5-9	39	enddr2-10-by-5-9
10	e0ddr2-10-by-5-1	40	ewddr1-10-by-5-1
11	e0ddr2-10-by-5-10	41	ewddr1-10-by-5-10
12	e0ddr2-10-by-5-2	42	ewddr1-10-by-5-2
13	e0ddr2-10-by-5-3	43	ewddr1-10-by-5-3
14	e0 ddr 2- 10 - by - 5 - 4	44	ewddr1-10-by-5-4
15	e0ddr2-10-by-5-5	45	ewddr1-10-by-5-5
16	e0ddr2-10-by-5-6	46	ewddr1-10-by-5-6
17	e0ddr2-10-by-5-7	47	ewddr1-10-by-5-7
18	e0ddr2-10-by-5-8	48	ewddr1-10-by-5-8
19	e0ddr2-10-by-5-9	49	ewddr1-10-by-5-9
20	enddr1-10-by-5-1	50	ewddr2-10-by-5-1
21	enddr1-10-by-5-10	51	ewddr2-10-by-5-10
22	enddr1-10-by-5-2	52	ewddr2-10-by-5-2
23	enddr1-10-by-5-3	53	ewddr2-10-by-5-3
24	enddr1-10-by-5-4	54	ewddr2-10-by-5-4
25	enddr1-10-by-5-5	55	ewddr2-10-by-5-5
26	enddr1-10-by-5-6	56	ewddr2-10-by-5-6
27	enddr1-10-by-5-7	57	ewddr2-10-by-5-7
28	enddr1-10-by-5-8	58	ewddr2-10-by-5-8
29	enddr1-10-by-5-9	59	ewddr2-10-by-5-9

Figure B.1: The numbering scheme for Sadeh's problems.

height of the search tree, or maximum path length of legitimate decisions, is generally much less than 225. The height varies substantially depending on the variable order heuristics that determine which ordering decisions to consider first (see Section F).

We profiled all sixty problems. Not all of the profiles were the same, as they generally were for the random 3SAT experiments discussed in Section D. Nevertheless, taking one sample from each set gives a fair representation of the set of problems at large. To further conserve on space, we will give the full profile of only one of the problems (the first) and then selectively cull out the mistake probability graphs from the profiles of the other five problems to point out relevant differences or similarities.

In Chapter 2, we discussed some of the differences between search spaces of scheduling problems searched with variable order heuristics and the search spaces of the same problems searched without heuristics. The difference is so great that we will present two profiles—one of the search with heuristics and one of the search without. Our discussions and presentation of the data follow in the next two sections.

B.2 The Search Space Without Heuristics

Figure B.2 shows the distribution of nodes, good nodes, and goals in the profiled portion of the search space of the first problem. Each profile search had a timeout of 10,000,000 nodes, which in general was not large enough to search the entire space exhaustively. The graphs thus reflect data collected from subtrees of the search spaces that we assume are representative of the full search spaces, at least with respect to the relevant statistics (see Section E).

Figure B.2 shows that in contrast to the random 3SAT problems, the proportion of good nodes to nodes appears to be at least fifty percent throughout the difficult region of the space. More important than the actual proportion is the mistake probability shown in Figure B.3. The mistake probability is low and reasonably constant throughout the recorded range. The effective mistake probability with a lookahead of one is extremely small throughout the range, suggesting that a lookahead of one in a bounded backtrack algorithm may offer a significant advantage over iterative sampling.

For an individual problem, the number of possible values for the mistake probability at each depth k is limited by the number of nodes at k. For example, at depth zero, there is only one node. For the mistake probability to be defined, at least one of its two children must be good. Thus the only variable is whether or not the other child is also good; the mistake probability can be 0.5 or 0—two possible values. Since the profile search did not collect many meaningful samples outside of the depth range 140 to 180, the graph of the mistake probability outside this range is not statistically reliable. The figures of Chapter 2 restricted the range to depths at which there were at least 100 samples, cutting out the noise at the ends. In this appendix, however, we are showing all of the data and leaving the editing to the reader.

The dramatic difference between the mistake probability with a lookahead of zero and one brings to question the utility of larger lookahead amounts for bounded backtrack algorithms. Figure B.4 shows that after the dramatic step from zero to one, the effective mistake probability appears to decrease linearly with lookahead amount. The effective mistake probability also decreased linearly for the random 3SAT problems, but from the value at l=0 instead of a substantially lower value at l=1. Figure B.5 gives more information about the heights of trees below mistake nodes. h(l) is the probability that the subtree below a mistake node has height l, given that the mistake node was not deep enough to make l impossible. This graph shows that the vast majority of subtrees have height zero, but beyond that the subtree heights appear to be fairly evenly distributed.

We investigate the zero height subtrees further in Figure B.6. This graph shows the probability for each depth k that a subtree below a mistake node at k has height zero. Amazingly, it is nearly one throughout the entire space. This strongly suggests that a bounded backtrack algorithm with a lookahead of one would offer significant improvement over iterative sampling. Contrast Figure B.6 with the graph of f(k,0)

¹Since there are relatively few nodes high enough in the tree to have tall subtrees, the straight forward probability distribution of mistake subtree heights is heavily biased toward nodes deep in the tree. h(l) is defined in a way that gives all depths equal representation: h(l) is the probability that the subtree below a mistake node has height l, selecting the mistake node at random from the set of nodes at a particular depth k, selected at random from the set of depths not exceeding d-l.

for random 3SAT problems (Figure D.6)—for random 3SAT, f(k,0) is zero throughout most of the space.

Small mistake subtrees, while relevant to the performance of iterative sampling, do not dramatically affect the performance of the systematic methods since exhaustively searching a subtree only takes as much time as there are nodes to examine. Large mistake subtrees, on the other hand, are a threat for systematic methods for the same reason. The fact that most of the mistake subtrees are small is encouraging for depth first search, but it only takes one large mistake subtree to destroy depth first search's average performance. Do such large subtrees exist? Figure B.5 shows there are at least some subtrees with height exceeding forty. Figure B.7 shows the that the size of these subtrees can be as much as 40,000 nodes. In general, it appears that the size of the subtrees is growing according to their height at about $2^{l/4}$. It is likely that the there are many larger mistake subtrees too high in the tree for the profile search to discover. Figure B.8 confirms that although the average cost of a mistake subtree is generally quite small, the maximum cost can be prohibitively large.

The remaining figures in this section show the mistake probabilities for the five other problems. The mistake probability appears to be roughly constant for most of the depth range. Near the edges, the number of samples was too few to get a reliable estimate, which explains the noisiness of the data. It does appear that the mistake probability is higher on the left side of the graphs, suggesting that it may be higher earlier in the search tree. Although this affect would be consistent with our intuitions about the scheduling problems, we have been unable to confirm it experimentally.

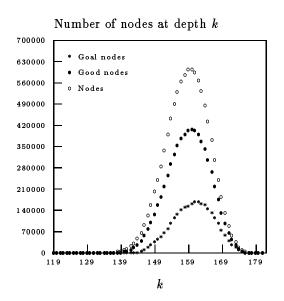


Figure B.2: Distribution of nodes over depth range for Sadeh (rand) no. 1.

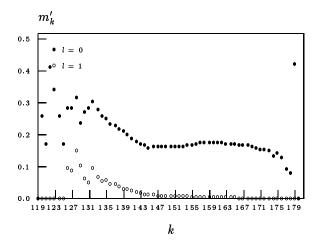


Figure B.3: $m_k'(0)$ and $m_k'(1)$ for Sadeh (rand) no. 1.

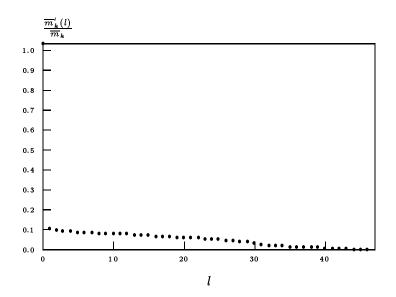


Figure B.4: Ratio of \overline{m}' to \overline{m} for Sadeh (rand) no. 1.

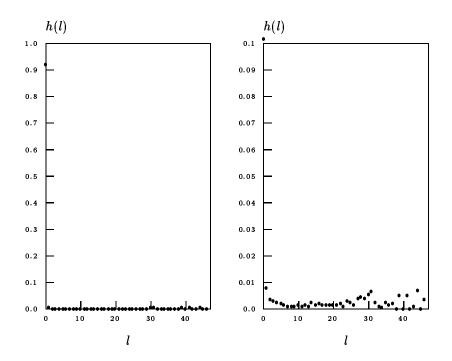


Figure B.5: h(l) for Sadeh (rand) no. 1, magnified on right.

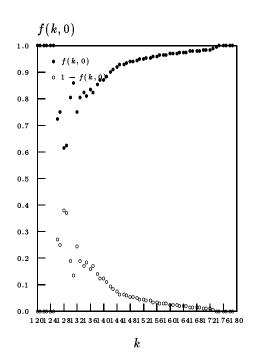


Figure B.6: f(k,0) for Sadeh (rand) no. 1.

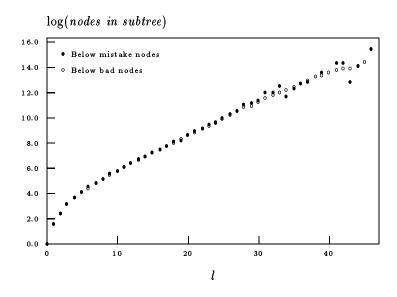


Figure B.7: Cost of bad subtrees by height l for Sadeh (rand) no. 1.

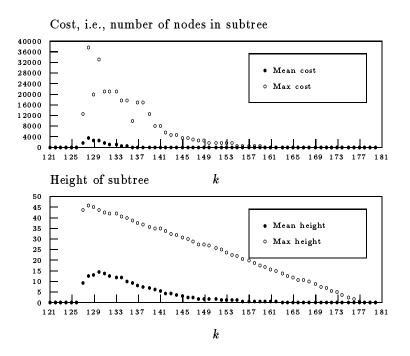


Figure B.8: Cost, height of bad subtrees at depth k for Sadeh (rand) no. 1.

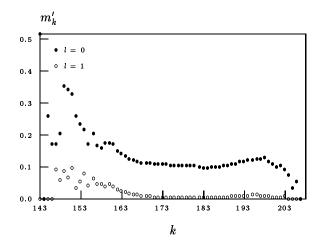


Figure B.9: $m_k'(0)$ and $m_k'(1)$ for Sadeh (rand) no. 11.

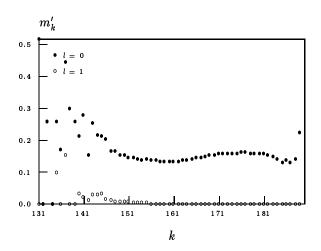


Figure B.10: $m'_k(0)$ and $m'_k(1)$ for Sadeh (rand) no. 21.

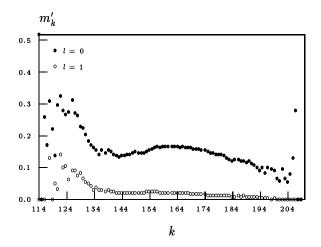


Figure B.11: $m_k'(0)$ and $m_k'(1)$ for Sadeh (rand) no. 31.

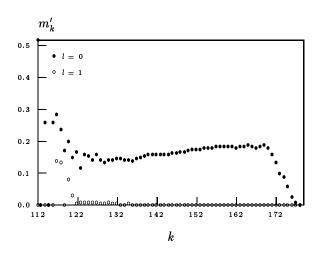


Figure B.12: $m'_k(0)$ and $m'_k(1)$ for Sadeh (rand) no. 41.

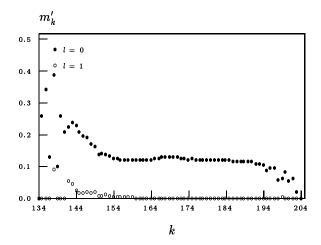


Figure B.13: $m_k'(0)$ and $m_k'(1)$ for Sadeh (rand) no. 51.

B.3 The Search Space With Heuristics

The variable order heuristics for the CSP formulation of job shop scheduling attempt to put critically constrained decisions first (see Section F). An effect of this preference is to reduce the height of the search space. With the heuristics, most of the solutions for the first problem are found between depths 72 and 92 (Figure B.14), about half as deep as in the search space of the same problem without heuristics (Figure B.2). The proportion of goal nodes to nodes, although still surprisingly high, is *lower* with the heuristics than without. Why the would the heuristics help? The mistake probability in Figure B.15 helps to explain.

While graph of the mistake probability is anything but constant, the graph of the effective mistake probability with a lookahead of one is zero for the entire profiled range. All of the mistakes in the profiled range had subtrees of height zero! The graphs for the other five problems in Figure B.20 through B.24 show similar effects. Since not all of the problems were solved by depth first search without backtracking from a subtree of height greater than one, there must be some more serious mistake nodes higher up in the tree, before the recorded range. Unfortunately, tracking these mistakes high in the tree would exceed our current computational resources.

Figure B.16 shows the cost of bad subtrees appears to be growing at about $2^{l/2}$. If we were to take that as an estimate of the rate of growth of the full search space, we would expect that the search space would have about 2^{45} nodes, out of range for our current resources and profiling procedure. It is interesting to note that while the search space without variable ordering heuristics was twice as high, the cost of bad subtrees was growing with half the exponent $(2^{l/4})$. This leads us to speculate that the efficacy of the heuristics may not derive from a reduction in the size of the search space, which is the conventional wisdom. The magic of the scheduling heuristics may be more subtle than we currently understand.

The remaining figures in this section (B.17, B.18, and B.19) don't show anything that we do not already know from the other graphs. We include these figures for comparison with the search space of scheduling without heuristics and the search spaces of the random 3SAT problems. It is easy to miss the interesting regions in

these graphs. Figure B.17, showing the distribution of subtree heights, has only one non-zero point. Figure B.18 shows that the ratio of the effective mistake probability with lookahead l to the mistake probability without lookahead is zero for any l > 0. The effect is further confirmed by Figure B.19. The one missing point at depth 61 in this graph reveals that there were no mistake nodes at that depth.

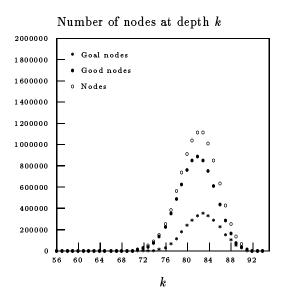


Figure B.14: Distribution of nodes over depth range for Sadeh (heur.) no. 1.

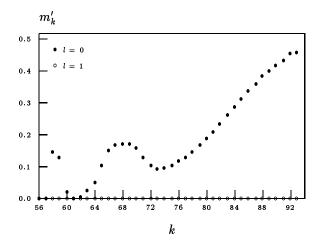


Figure B.15: $m_k'(0)$ and $m_k'(1)$ for Sadeh (heur.) no. 1.

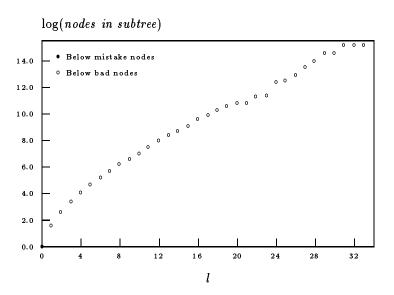


Figure B.16: Cost of bad subtrees by height l for Sadeh (heur.) no. 1.

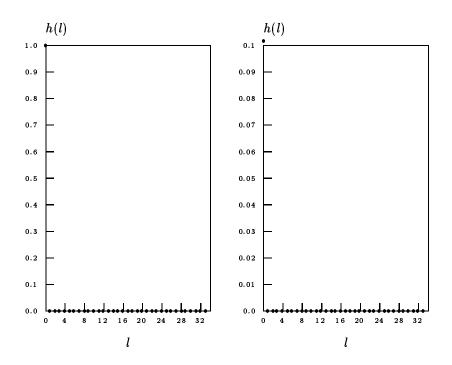


Figure B.17: h(l) for Sadeh (heur.) no. 1, magnified on right.

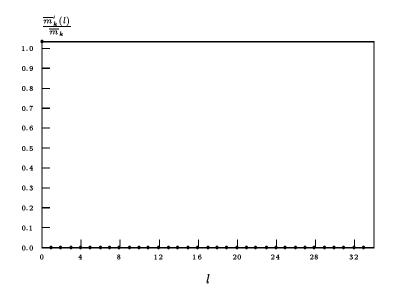


Figure B.18: Ratio of \overline{m}' to \overline{m} for Sadeh (heur.) no. 1.

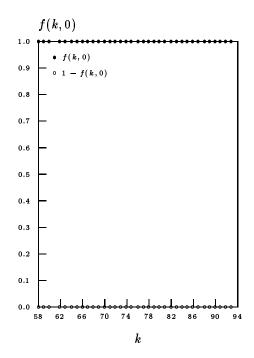


Figure B.19: f(k,0) for Sadeh (heur.) no. 1.

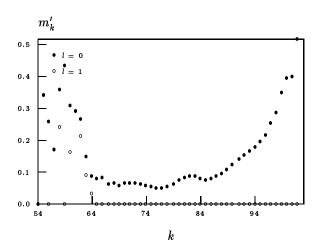


Figure B.20: $m_k'(0)$ and $m_k'(1)$ for Sadeh (heur.) no. 11.

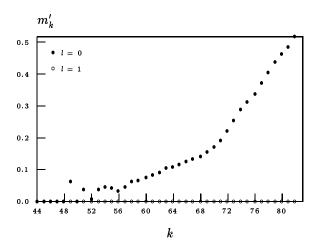


Figure B.21: $m_k'(0)$ and $m_k'(1)$ for Sadeh (heur.) no. 21.

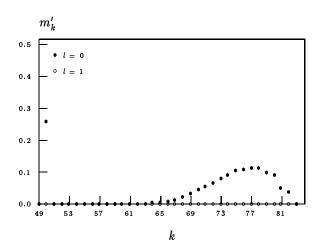


Figure B.22: $m_k'(0)$ and $m_k'(1)$ for Sadeh (heur.) no. 31.

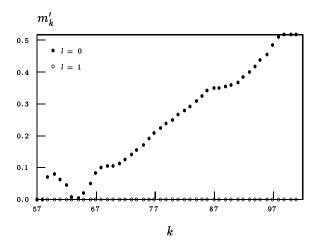


Figure B.23: $m_k'(0)$ and $m_k'(1)$ for Sadeh (heur.) no. 41.

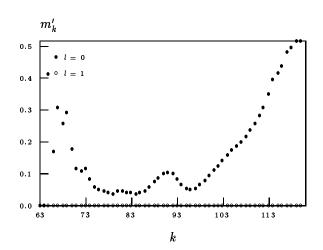


Figure B.24: $m_k'(0)$ and $m_k'(1)$ for Sadeh (heur.) no. 51.

B.4 Conclusions

The search spaces of the job shop scheduling problems appear to have low mistake probabilities at all depths, with or without variable order heuristics. The vast majority of mistake subtrees have a height of zero. Nevertheless, it also appears to be possible to make a mistake high in the tree that leads to a large subtree with no solutions. This combination of search space properties suggests that a bounded backtrack search algorithm would be more effective than iterative sampling or depth first search.

The heuristics have some peculiar effects on the mistake probability. Without lookahead, the mistake probability varies dramatically over the depth range. With lookahead, the effective mistake probability is zero over the entire depth range. While these effects argue strongly for a bounded backtrack algorithm, they do lead us to question our understanding of the scheduling heuristics. We speculate that the efficacy of the heuristics may come from more subtle factors than just the reduction in the size of the search space.

B.5 Earlier Graphs

Figures showing job shop scheduling problems throughout the thesis identify the problems by problem number for brevity. The mapping of problem number to filename for the problems from Sadeh's thesis [43] is given in Figure B.1.²

There are two differences between the graphs in Chapter 2 and the graphs in this section. The first difference is that the graphs in Chapter 2 without heuristics used random value selection, whereas the graphs in this section used heuristic value selection (see Section E for an explanation). The difference affects the depth range but not the properties that the graphs illustrate. The second difference is that the earlier graphs used a minimum samples cutoff of one hundred samples per data point. In this section, our cutoff was one sample; we graphed all data points.

²Parsed versions of the files in Lisp readable format are available over the Web at URL: http://cirl.uoregon.edu/harvey.

Appendix C

A Profile of Scheduling Problems as SAT

Our SAT encoding of Sadeh's job shop scheduling problems yields relatively large representations of the problems. Whereas the CSP representation of a typical problem is only a few thousand bytes, the SAT representation is typically larger than a megabyte. The SAT formulations of Sadeh's problems uses about 20,000 propositions and 100,000 clauses. Many of the variables get assigned by unit propagation, so the actual number of choice points, or decisions along the deepest path in the tree is far fewer than 20,000. We profiled six of Sadeh's problems represented in the SAT encoding. With a timeout of 100,000 nodes, two of the problems were solved. The shallowest solutions of the solved problems were found at a depth of approximately 2,000; about 18,000 variable assignments followed from unit propagation.

A striking difference between these profiles and the profiles of the same problems in the CSP formulation is the number of mistake nodes in the search space deep in the tree. The search space of the CSP formulation is fraught with mistake nodes at all levels. The search space of the SAT formulation apparently has few or no mistakes deep in the search tree. Deep in the tree of the SAT formulation, it appears that if a node is bad, so is its sibling; and if a node is good, so is its sibling. Clearly, any tree that has at least one goal node and at least one terminal node that is not a goal must have a mistake node. In the SAT formulations, however, none of these mistakes

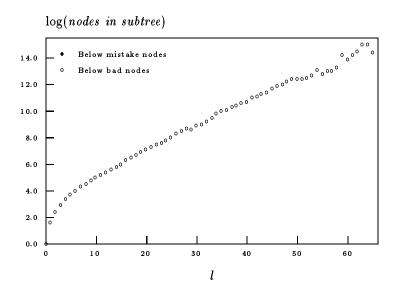


Figure C.1: Cost of bad subtrees by height for SadehSat (w/heuristics) no. 10.

were among the first 100,000 nodes examined.

Our conclusion from the profiled data is that the SAT formulation of the scheduling problems has a mistake probability of approximately zero deep in the tree. There is no evidence that any small lookahead would reduce the effective mistake probability. The one mistake node that must exist has a large, tall subtree below it. There may be other mistake nodes that have smaller subtrees below them, but these mistake nodes, if they exist, are in a part of the search tree that our profiling methods were not able to illuminate. Given the size of the tree, it is unlikely that we could record the required data even with a computer many orders of magnitude faster than our current resources. Due to these limitations, the graphs of the mistake probability are omitted in this section.

Our recorded data is sufficient to show the cost of bad subtrees as a function of their height, which appears to grow at a rate of approximately $2^{l/6}$ to $2^{l/4}$. In contrast, the growth rate of the bad subtree cost in the CSP formulations is about $2^{l/2}$ with heuristics and $2^{l/4}$ without. The results are shown in the graphs in this section.

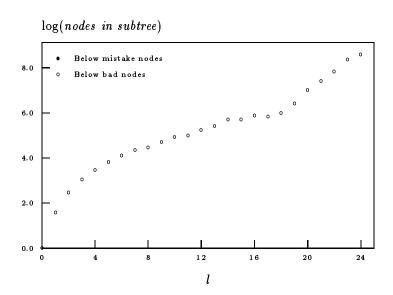


Figure C.2: Cost of bad subtrees by height for SadehSat (w/heuristics) no. 15.

Appendix D

A Profile of Random 3SAT

D.1 Introduction

Randomly generated 3SAT problems are propositional theories consisting of n propositions, or variables, and c clauses. Each clause is a disjunction of three literals, each literal a variable or its negation. Empirically, researchers have found that difficult random 3SAT problems can be generated by fixing the proportion of clauses to variables $(\frac{c}{n})$ at about 4.26 [10, 11]. This ratio is believed to be the *crossover point*, the point at which half of the random 3SAT theories so generated are satisfiable, and half not [10, 36, 44].

We generated a set of 100 random 3SAT problems, each with 250 variables and 1,065 clauses (i.e., near the crossover point). Of the 100 problems generated, 47 were satisfiable. We profiled the 47 satisfiable problems using a Davis-Putnam style satisfiability engine [36] modeled after Crawford's *Tableau* program [10]. With a time-out of 1,000,000 nodes, we searched the space of eight of the problems exhaustively and the rest partially, collecting statistics along the way as discussed in Section E. The statistics reveal a number of peculiar characteristics of the random 3SAT search spaces that are strikingly different from the search spaces of scheduling problems.

The profiles of the 47 problems fell about evenly into two categories, for which Problem 6 and Problem 8 are good representatives. All of the problems in the first category had similar profiles to Problem 6, and all of the problems in the second

category had similar profiles to Problem 8 (we discuss the differences in Section D.5). The complete profiles for these two problems are given in figures at the end of this section.

D.2 Characterizing the Search Tree

Each decision point in the search contributes two nodes to the search space: one for assigning the variable true, and the other for assigning it false. Variable assignments that follow from unit propagation do not count as additional nodes. The search continues until all of the variables are assigned, but since many variables are assigned by unit propagation, the actual height of the tree is far less than 250, the height it would have to be if variable assignments from unit propagation counted as nodes.

The solutions for most of the problems were clustered at about depth 50 in the tree, which is shown in Figure D.1 and Figure D.8. Problem 6 also had a number of solutions at depth 40, but this appears to have been idiosyncratic. Naively, we might measure the difficulty of the problem by the height of the tree, or even the average depth of the fringe nodes. Such a characterization of the search space would be gravely misleading. Notice that in Figure D.1, all of the nodes at depth 36 or greater are good nodes. Thus every path below depth 32 ends at a goal node! The real difficulty of the problem is getting to one of the good nodes at depth 32, of which there are very few.

The graph in Figure D.2 of the mistake probability shows that every decision to depth 23 except one was a choice between a good and bad variable assignment. Thus the number of good nodes at depth 23 is exactly two. Since there are about 32,000 nodes at that depth alone, the chance of arriving at one of the good nodes by a random path is relatively small.¹

The precipitous drop of the mistake probability is consistent with the observation that no nodes deeper than depth 36 are bad. The reason none of these deep nodes fails appears to be that by about depth 30, either all of the clauses in the theory

¹Experimentally, we have observed very bad performance of iterative sampling on these problems, a result that appears to be consistent with our profile of the search space.

are satisfied by the partial variable assignment, or the partial variable assignment is inconsistent with the theory. When all of the clauses are satisfied by a partial assignment, every possible assignment to the remaining variables must be consistent with the theory. If there are n remaining variables, there will be a full binary tree of height n below the path representing the partial assignment.

The difficult part of the search space is the upper tree of height 30. Hanging from the few good paths through the difficult part are the full binary trees of about height 25 consisting entirely of good nodes.² Before running the experiments, we expected that there would be a few trees of height 25 consisting entirely of bad nodes hanging from the difficult part. These large bad subtrees would be bear traps for depth first search. On account of the traps, we would have expected alternative search techniques like bounded backtrack search or branch factor cutoff search to perform better, but to our surprise there were no such traps. Figure D.7 shows that by depth 33 in the search, the tallest bad subtree is only eight high, with a negligible number of nodes. The upper tree of height 30 is indeed the only difficult part of the problem.

D.3 Getting to the Good Part

Searching the space as a tree, the only way to get to the large "good" trees hanging from the difficult part is through the one or two paths in the difficult part representing partial assignments that are consistent with the theory. Thus for tree search, the problem is at least as hard as solving the difficult part, notwithstanding the good trees. For total assignment techniques [44], however, the presence of the good trees may not be irrelevant since the good nodes can be reached from other places in the search space without necessarily having to follow one of a few good paths. Of course, the search space of arbitrary total assignments is much larger than the size of the tree searched by Davis-Putnam style algorithms with unit propagation, so the tradeoff is

²The trees of good nodes are so large that in most cases the number of goal nodes represents about a quarter of the total number of nodes in the entire search space, an ironic statistic for problems that have proven extremely difficult to solve.

not obviously a good one.³ Trying to achieve the pruning benefits of unit propagation within the total assignment paradigm is a topic of current research. The prospect of efficiently searching the fringe of the tree remains an open question.

D.4 Grappling with the Difficult Part

Tree search techniques all require finding one of a few successful paths through the difficult part of the search space, the upper tree of about height 30. By studying the profile of this tree, we may be able to find characteristics of the search space that could be exploited by a search algorithm to improve its average case performance. The mistake probability shown in Figure D.2 is not promising. It shows that there are only a few good paths. Every decision matters.

We know from earlier theoretical results (see Section 2.6.2) that if the tree were full, depth first search would outperform iterative sampling by a factor of d, the height of the tree. However, this tree is not full. Figure D.4 shows that the number of nodes in a bad subtree of height 36 is about 2^{17} . The size of bad subtrees appears to be growing at $2^{l/2}$, significantly less than the 2^{l} growth rate of full binary trees.⁴ Could we hope to reduce the effective mistake probability with lookahead? Figure D.3 shows that the heights of mistake subtrees are fairly evenly distributed, not at all like distribution for the scheduling problems. We can see exactly what the effect of lookahead would be for the effective mistake probability in Figure D.5. The data are again not promising. The ratio goes down, as it has to,⁵ but it does not drop precipitously to suggest a particular lookahead would be efficacious.

In the scheduling problems, the probability that the height of a mistake subtree is zero was extremely high—nearly one—for most of the search. In the random 3SAT

³Experiments have shown empirically, however, that it is a very good tradeoff for random 3SAT [44]. GSAT and its variants outperform tree search algorithms by many orders of magnitude on these problems.

⁴If this growth rate holds true for random 3SAT problems at the crossover point independent of the size of the problem, the $2^{l/2}$ growth rate could be useful for estimating the search space size based on an estimate for l.

⁵When the lookahead is equal to the depth of the tree, it is impossible to make a mistake. Thus the effective mistake probability for l = d has to be zero.

problems, the probability is zero for most of the search (see Figure D.6). Whereas a lookahead of one would dramatically affect the mistake probability for the scheduling problems, it would have no effect whatsoever in most of the search tree for random 3SAT.

D.5 Other Graphs, Other Problems

The profile of Problem 8 is shown in Figure D.8 through Figure D.11. The difference between the mistake probability in Figure D.2 and Figure D.9 is an artifact of the data collection rather than a reflection of a difference in the search space. With a timeout of 1,000,000 nodes, we were able to search Problem 6 exhaustively, but not Problem 8. While profiling Problem 8, the search timed out before backtracking past depth 31. Thus Figure D.9 shows only the second half of the search space, missing the difficult part. The part that it does show is consistent with the graph in Figure D.2, so there is no reason to believe there is any fundamental difference between the search spaces of the two problems.

The Tableau style search engine uses variable ordering heuristics to reduce the size of the search space [10]. We attempted to profile the space of Problems 6 and 8 using random variable ordering but even with a timeout increased ten fold to 10,000,000 nodes, no solutions were found. Our profile data from searches that do not reach any goal nodes is not aluminating for properties relating to mistakes for obvious reasons (they don't record any mistakes).

D.6 Conclusion

The search spaces of the scheduling problems had a number of properties that our non-systematic algorithms were able to exploit to improve performance. In Appendix B we attempted to show by profiling the search spaces of the scheduling problems that these properties really do exist. Our method of profiling a search space too large to search exhaustively relies on a few assumptions that we discuss in Appendix E. Although the assumptions are reasonable and there is no reason to doubt that they

hold, we wanted to be absolutely sure that the evidence we collected was not an artifact of the collection technique or a bug in the program. We needed some control experiment with which to compare the results. Our profile of random 3SAT problems has served as this control.

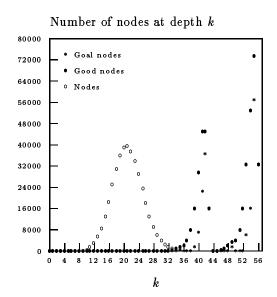


Figure D.1: Distribution of nodes over depth range for Random 3SAT no. 6.

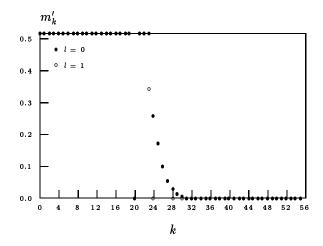


Figure D.2: $m_k'(0)$ and $m_k'(1)$ for Random 3SAT no. 6.

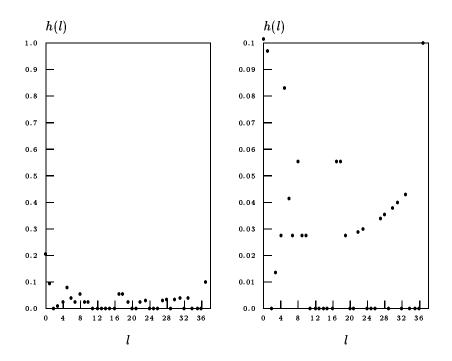


Figure D.3: h(l) for Random 3SAT no. 6, magnified on right.

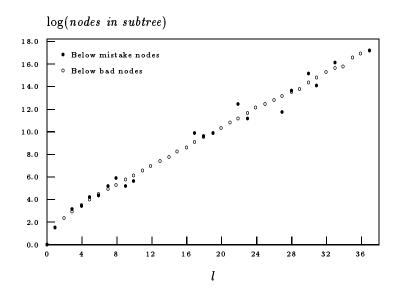


Figure D.4: Cost of bad subtrees by height l for Random 3SAT no. 6.

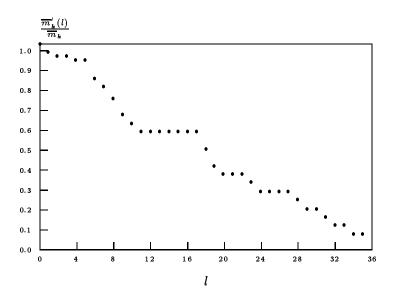


Figure D.5: Ratio of \overline{m}' to \overline{m} for Random 3SAT no. 6.

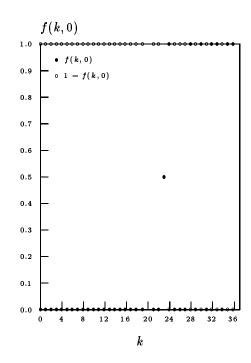


Figure D.6: f(k,0) for Random 3SAT no. 6.

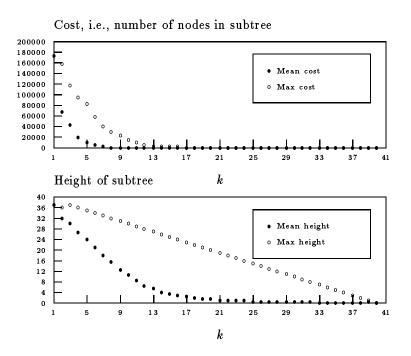


Figure D.7: Cost, height of bad subtrees at depth k for Random 3SAT no. 8.

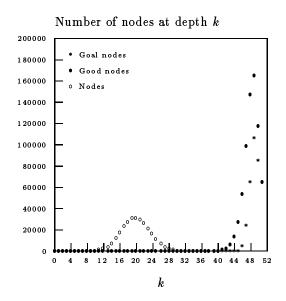


Figure D.8: Distribution of nodes over depth range for Random 3SAT no. 6.

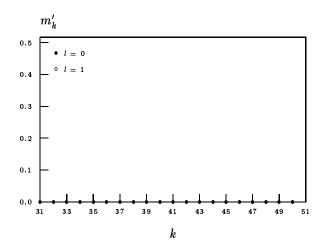


Figure D.9: $m_k'(0)$ and $m_k'(1)$ for Random 3SAT no. 8.

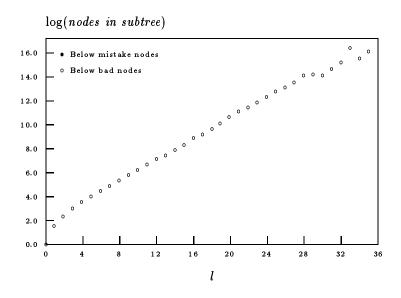


Figure D.10: Cost of bad subtrees by height l for Random 3SAT no. 8.

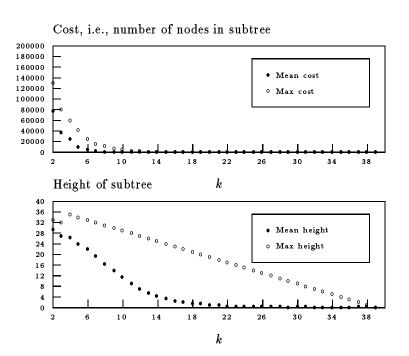


Figure D.11: Cost, height of bad subtrees at depth k for Random 3SAT no. 8.

Appendix E

Collecting Profile Data

Throughout this thesis there are a number of graphs showing properties of various search spaces. These graphs were generated by exhaustively searching some subtree of each search tree, collecting statistical data along the way. This section presents the details of how the data were collected and how the graphs were drawn.

E.1 Node Classes

The search trees that we profiled are all binary trees. Every node has zero or two successors. Ignoring the order of successors, every node that is not a dead end belongs to one of three classes, as shown in Figure E.1.

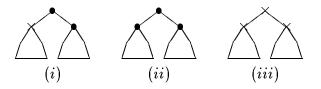


Figure E.1: Every interior node is in one of three classes.

A good node must have at least one good successor. A bad node has no good

successors. The only nodes that affect the mistake probability are those nodes in class (i) and (ii). Thus even an enormous subtree that consists only of class (iii) nodes, which might result from an "early mistake" in the search tree, has no effect on the mistake probability in the tree. When we profile a search tree, we record for every depth the number of nodes in each class and the heights of their subtrees.

E.2 Traversing the Search Tree

If we could exhaustively search the entire search tree of a problem, we could determine exactly the mistake probability m_k of the search tree by recording which nodes are good and bad at all levels of the tree. Unfortunately there are two problems with this approach. The first is that the search trees of the interesting problems grow exponentially, even with elaborate pruning mechanisms, so exhaustively searching the trees exceeds our computational resources. The second problem is that even if we could search the trees exhaustively, the exact mistake probability m_k of any individual problem is not really the property of interest. What we would really like is the mistake probability of a set of possible realistic problems for which the profiled problem is a good representative. We would like to know, for example, how likely it is to make a mistake early in the search tree. For any individual binary tree, m_0 is either one half or zero since the root node has only two children. For a set of possible problems, however, m_0 is likely to be some intermediate value.

In light of these difficulties, our approach to profiling a problem is to follow a path to some node at depth k and to exhaustively search the subtree below that node, choosing k such that the computational resources are not exceeded. We assume that the subtree below the chosen node, if it contains any goals, is representative of the entire search space. If the subtree does not contain any goal nodes, then it cannot contain any mistake nodes and therefore is insufficient to determine m_k ; in this uncommon circumstance, we report that the properties of the search tree are unknown.

When we graph m_k and various other data, we begin the graph at a depth at which there enough samples to be statistically significant. For example, even if we searched

a full binary tree exhaustively, we would have only one sample for m_0 . Requiring one hundred samples, we would begin graphing m_k at k=8.

E.3 Incorporating Heuristics

The scheduling heuristics of our experiments are both variable and value ordering heuristics for the CSP formulation of job shop scheduling [26, 45]. In the tree search formulation, CSP value ordering heuristics translate to successor ordering heuristics. In the exhaustively searched portion of the search tree, successor ordering heuristics do not affect the recorded statistics, since both successors of a node are searched regardless of their order.

CSP variable ordering heuristics affect the tree search formulation fundamentally, since instantiating variables in different orders yields different search trees. Our experiments show that search trees generated from scheduling CSPs with random variable ordering have different properties from search trees generated with heuristic variable ordering.¹

The statistical data are recorded from a subtree that we assume is representative of the entire search space. The path to the root of the subtree at depth k is a sequence of k decisions among successors. These decisions could be made randomly or they could be made according to a successor ordering heuristic. Using a successor ordering heuristic risks selecting a subtree that is not representative of the space at large. Not using a successor ordering heuristic risks making a really bad choice in the first k decisions, which almost always leads to a subtree with no goal nodes. Subtrees with no solutions do not affect m_k one way or the other since they contain no good nodes. We have found experimentally that any subtree with goals is typically a good representative of the space at large, regardless of the path to the subtree. To make the best use of our computational resources, we have chosen to use the successor ordering heuristics to determine the path to the subtree at depth k.

¹It could be argued that some of these differences help to explain why the variable ordering heuristics are effective in reducing the cost of solving the CSP. An analysis of variable ordering heuristics using the mistake probability properties is beyond the scope of our work but would be an interesting topic for future research.

For efficiency reasons, our translation of the scheduling problems from constraint satisfaction to tree search does not treat forced variable assignments (i.e., variable assignments following from unit propagation) as separate nodes. Thus the height of the search tree is only the length of the longest sequence of real choice points, which may be less than the number of variables or more than the depth of most solutions. Solution depth varies dramatically for the scheduling problems, which makes it difficult to predict the probability of a probe's success in these spaces based on the m_k graphs alone.

Appendix F

Implementation Considerations

F.1 Multiprobe

Multiprobe is the name of our implementation of bounded backtrack search. A sketch of the algorithm is given in Figure F.1. This implementation incorporates the completeness modification discussed in Section 4.8.3. As the value sysdepth increases to the maximum depth of the tree, the top part of the tree from the root to sysdepth is searched systematically. When sysdepth reaches the maximum depth, the entire tree is searched systematically. Multiprobe is thus a decision procedure, producing a solution or a proof that the search space has no solutions.

The functions Weighted-Random(n, weight) and Select-Value(list, index, promotedindex) are given in Figure F.2. Together they "randomize" the value order in a manner that works well for both large and small branch factors. The branchlimit parameter in line 10 is always one for standard bounded backtrack search. If the branch limit is increased on each iteration of Multiprobe, the result is iterative broadening with a bounded backtrack. The branch limit can also be used to simulate a fractional branch factor cutoff c by replacing the constant in line 5 with a function that returns |c| or [c] probabilistically (see Section 4.6.1).

Multiprobe can also be combined with limited discrepancy search. A simplified version of our implementation is given in Figure 3.8.

```
MP(node, depth, lookahead, randomness, sysdepth)
  1
      if GOAL-P(node)
  2
           return \langle node, 0 \rangle
  3
      s \leftarrow \text{SUCCESSORS}(node)
      branch count \leftarrow 0
  5
      branchlimit \leftarrow 1
  6
      maxheight \leftarrow 0
      promotedindex \leftarrow Weighted-Random(Length(s), randomness)
  8
      for i \leftarrow 0 below LENGTH(s)
  9
           child = Select-Value(s, i, promoted index)
           if sysdepth \leq depth and branchcount = branchlimit
  10
  11
                break
  12
           \langle result, height \rangle \leftarrow MP(child, depth + 1, lookahead, randomness, sysdepth)
  13
           maxheight \leftarrow Max(maxheight, 1 + height)
  14
           if result \neq NIL
                return \langle result, 0 \rangle
  15
  16
           if height > lookahead
  17
                branchcount \leftarrow branchcount + 1
  18 return \langle NIL, maxheight \rangle
MULTIPROBE (node, lookahead, randomness)
  1
      for sysdepth \leftarrow 0 to maximum depth
 2
           \langle result, height \rangle \leftarrow MP(node, 0, lookahead, randomness, sysdepth)
  3
           if result \neq NIL
                return result
  4
  5
      return NIL
```

Figure F.1: Multiprobe.

F.2 Weighted Random Selection of Values

Our implementation of Multiprobe allows arbitrary and non-uniform branching factors. While experimenting with crossword puzzle generation [23], we found a simple implementation for weighted random selection of values that worked well for both small and large branching factors. We first generate a random number i from a skewed probability distribution by taking the minimum of k random numbers from zero to the number of successors (i.e., values). Assuming the successors list is sorted in order of heuristic preference, we promote the i^{th} successor to the front of the list, leaving the order of the other successors unchanged. We found that promoting a bad value to the front was not particularly harmful because bad values tend to fail quickly and get caught by the bounded backtrack. Thus for large branching factors, this simple promotion scheme appeared to satisfy the need for random selection without significantly compromising the heuristic. The two functions in Figure F.2 are used in the Multiprobe algorithm shown in Figure F.1.

```
Weighted-Random (n, weight)
      w \leftarrow \text{RANDOM}(n)
 2
      for i \leftarrow 1 to weight - 1
           w \leftarrow \text{Min}(w, \text{Random}(n))
  3
      return w
Select-Value(list, index, promoted index)
  1
      if index = 0
 2
           return NTH(promoted index, list)
 3
      else if index \leq promoted index
  4
           return NTH(index - 1, list)
 5
      else return NTH(index, list)
```

Figure F.2: Function to "randomize" value order.

In the weighted probability distribution, the first value, or successor, has a probability $1-(\frac{n-1}{n})^k$ of selection. The last value has a probability $(\frac{1}{n})^k$. For the scheduling

problems, n was two. k is the randomness parameter to Multiprobe (See Figure F.1). We found a randomness value of three to work well in the scheduling domain, and five to work well for crosswords.

F.3 Walk Start: Restarting with Different Variables

The nonsystematic methods that explore the space with few probes relative to iterative sampling may benefit from restarting with a different first variable on each probe. Iterative broadening, depth first search with restarts, and bounded backtrack with a large lookahead are examples of such methods. In domains for which the variable ordering heuristic selects variables that were affected by recent instantiations to minimize thrashing in backtrack search, the entire path of variable instantiations can be largely determined by the selection of the first variable. It is possible for a variable ordering heuristic that is generally very good to occasionally commit to a pathological sequence of instantiations, one from which the value ordering heuristic has little chance of guiding the search to a solution.

The function in Figure F.3 can be used to select the first variable on successive probes from a heuristically ordered list. The function should be called with the number of variables n and the probe number i. The result is an index into the ordered list of variables for the variable on which probe i should start.

```
egin{array}{lll} 	ext{Weighted-CYCLE}(i,\ n) & 1 & i \leftarrow 	ext{Mod}(i,n*(n+1)/2) \ 2 & 	ext{for } x \leftarrow 1 	ext{ to infinity} \ 3 & 	ext{if } x > i \ 4 & 	ext{return } i \ 5 & 	ext{else } i \leftarrow i - x \ \end{array}
```

Figure F.3: Function for restarting on different variables.

The function cycles through the variables in a "weighted" cycle that selects the early variables more often than the later variables, in deference to the variable ordering heuristic. The pattern is,

0, 0,1, 0,1,2, 0,1,2,3, 0,1,2,3,4,

F.4 Application of CSP Heuristics to Scheduling

Some CSP search procedures instantiate variables in a fixed order. Others dynamically evaluate which variable to instantiate next at each decision point. A well known heuristic is to choose the most constrained variable to evaluate next [6]. Which variable is the most constrained can be estimated by counting the number of values from the domain of each variable that are consistent with the constraints and previous assignments. The most constrained variable is taken to be the one with the fewest remaining values.

The variables of the scheduling CSP all have two values representing the possible orders for two operations on the same machine.¹ Thus counting values doesn't provide an estimate for which variable is the most constrained. Nevertheless, we can estimate how constrained a variable is by considering the number of configurations each value represents.

We can think of a total order of operations on a machine as representing all possible sets of start times for those operations such that the operations don't overlap and the start times are consistent with the total order (i.e., solutions). Likewise, we can think of a value in the scheduling CSP (i.e., an order for two operations) as representing all

¹We are assuming zero and one valued variables are picked up by constraint propagation.

possible sets of start times (configurations) for the two operations that are consistent with the chosen order and the other constraints and assignments. Of course, without solving the problem it is impossible to know which configurations are consistent with all the constraints, but as a crude estimate we might consider counting the number of configurations that are consistent with the chosen order and the established earliest and latest possible start and finish times. This number of configurations, or the "space" as we will call it, can be measured for both continuous and discrete time formulations.

Consider the example of two operations a and b shown in figure F.4. We can measure the space for the a < b order as follows:

- 1. For a's starting positions from t = 0 to t = 1, b can start anywhere in its range from t = 3 to t = 9.
- 2. For a's starting positions from t = 1 to t = 6, b can start any time after a, up to t = 9. If a starts at t = 1, b's start time is unrestricted. If a starts at t = 6, b's start time is restricted by five hours. If a starts between t = 1 and t = 6, b's start time is restricted by how long after t = 1 a started.

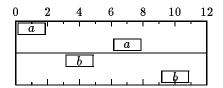


Figure F.4: There are 34 ways for a < b, only 3 for b < a.

For continuous time, the number of configurations is represented by the area of the graph shown in figure F.5. The discrete time calculation is slightly more complicated because of the endpoints. The space calculations for continuous time S_c and discrete time S_d are as follows:

$$p_1 = \min(\operatorname{lft}_a, \operatorname{est}_b)$$
 $x_1 = \max(0, p_1 - \operatorname{eft}_a)$
 $x_2 = \operatorname{lst}_b - \operatorname{est}_b$
 $x_{2p} = 1 + x_2$
 $p_2 = \max(\operatorname{eft}_a, \operatorname{est}_b)$
 $p_3 = \min(\operatorname{lft}_a, \operatorname{lst}_b)$
 $x_3 = \max(0, p_3 - p_2)$
 $x_4 = \operatorname{lst}_b - p_2$
 $x_{4p} = 1 + x_4$
 $S_c = x_1 x_2 + x_3 x_4 - x_3^2 / 2$
 $S_d = x_1 x_{2p} + x_3 x_{4p} - x_3 (x_3 + 1) / 2 + [p_1 \ge \operatorname{eft}_a] x_{2p} + [p_1 < \operatorname{eft}_a] [p_3 \ge p_2] x_{4p}$
(F.2)

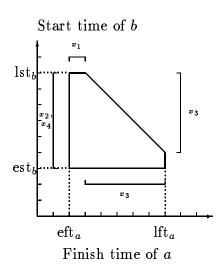


Figure F.5: The area of possible configurations.

These calculations estimate the number of configurations S that are consistent with the order a < b and the other constraints of the problem. The sum of S for a < b and S for b < a is thus an estimate of the number of remaining possible configurations for the operations a and b. Borrowing the intuition from CSP variable ordering, we may take the most constrained ordering decision to be the one with the fewest remaining possible configurations [26, 45].

Bibliography

- [1] E. Aarts, P. van Laarhoven, J. Lenstra, and N. Ulder. A computational study of local search algorithms for job shop scheduling. *ORSA J. Comput.*, to appear, 1994.
- [2] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Sci.*, 34, 1988.
- [3] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. ORSA J. Comput., 3, 1991.
- [4] E. Balas. Machine sequencing via disjunctive graphs: an implicit enumeration algorithm. *Oper. Res.*, 17, 1969.
- [5] J. Barnes and J. Chambers. Solving the job shop scheduling problem using tabu search. *IEEE Trans.*, to appear., 1994.
- [6] J. Bitner and E. M. Reingold. Backtrack programming techniques. Communications of the ACM, 18:651-655, 1975.
- [7] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1), 1981.
- [8] C. Cheng and S. Smith. Generating feasible schedules under complex metric constraints. In *Proc. of the Twelvth National Conference on Artificial Intelligence*, 1994.
- [9] J. Cohen. Non-deterministic algorithms. Computing Surveys, 2(2), 1979.

[10] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. In Proc. 11th AAAI, 1993.

- [11] J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In Proc. 12th AAAI, 1994.
- [12] F. D. Croce, R. Tadei, and G. Volta. A genetic algorithm for the job shop problem. Comput. Oper. Res., to appear, 1994.
- [13] M. Davis and H. Putnam. A computing procedure for quantification theory. Journal of the ACM, 7, 1960.
- [14] J. de Kleer. An assumption-based truth maintenance system. Artificial Intelligence, 28, 1986.
- [15] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. Artificial Intelligence, 41, 1990.
- [16] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 1989.
- [17] M. Dell'Amico and M. Trubian. Applying tabu search to the job-shop scheduling problem. Ann. Oper. Res., 41, 1993.
- [18] U. Dorndorf and E. Pesch. Evolution based learning in a job shop scheduling environment. Comput. Oper. Res., to appear, 1994.
- [19] E. Freuder. A sufficient condition for backtrack-free search. J. ACM, 29(1), 1982.
- [20] M. Garey and D. Johnson. Computers and Intractability. W. H. Freeman and Company, 1979.
- [21] J. Gaschnig. Performance measurement and analysis of certain search algorithms. PhD thesis, Carnegie-Mellon University, 1979.

[22] M. Ginsberg. Dynamic backtracking. Journal of Artificial Intelligence Research, 1, 1993.

- [23] M. Ginsberg, M. Frank, M. Halpin, and M. Torrance. Search lessons learned from crossword puzzles. In Proc. 8th AAAI, 1990.
- [24] M. Ginsberg and W. Harvey. Iterative broadening. Artificial Intelligence, 1992.
- [25] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, 14, 1980.
- [26] W. Harvey. Search and job shop scheduling. Technical Report CIRL TR 94-1, CIRL, University of Oregon, 1994.
- [27] J. Kalbfleisch. Probability and Statistical Inference. Springer-Verlag, 1985.
- [28] D. Knuth. Estimating the efficiency of backtrack programs. *Math Comp.*, 29, 1975.
- [29] R. Korf. Depth-first iterative deepening: An optimal admissible tree search. Artificial Intelligence, 27(1), 1985.
- [30] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. AI Magazine, 13(1), 1992.
- [31] P. V. Laarhoven, E. Aarts, and J. Lenstra. Job shop scheduling by simulated annealing. Oper. Res., 40, 1992.
- [32] P. Langley. Systematic and nonsystematic search strategies. In Artificial Intelligence Planning Systems: Proceedings of the First International Conference, 1992.
- [33] S. Lawrence. Resource constrained project scheduling: an experimental investigation of heuristic scheduling. Technical Report Working paper GSIA, Carnegie-Mellon University, 1984.

[34] A. Mackworth and E. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Artificial Intelligence, 25, 1985.

- [35] H. Matsuo, C. Suh, and R. Sullivan. A controlled search simulated annealing method for the general jobshop scheduling problem. Technical Report Working paper 03-04-88, Graduate School of Business, University of Texas, Austin, 1988.
- [36] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In Proc. 10th AAAI, 1992.
- [37] B. Nadel. Constraint satisfaction algorithms. Computational Intelligence, 5, 1989.
- [38] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. Technical report, Institute of Engineering Cybernetics, Technical University of Wroclaw, 1993.
- [39] W. Nuijten, E. Aarts, D. van Erp Taalman Kip, and K. van Hee. Job shop scheduling by constraint satisfaction. Technical Report Computer Science Note 93/39, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, 1993.
- [40] J. Pearl. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, 1984.
- [41] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. Computational Intelligence, 9(3), 1993.
- [42] P. Purdom. Search rearrangement backtracking and polynomial average time. Artificial Intelligence, 21, 1983.
- [43] N. Sadeh. Look-ahead techniques for micro-opportunistic job shop scheduling. Technical Report CMU-CS-91-102, School of Computer Science, Carnegie Mellon Univ., 1992.

[44] B. Selman and H. Kautz. Local search strategies for satisfiability testing. In Proc. of DIMACS Workshop on Maximum Clique, Graph Coloring, and Satisfiability, 1993.

- [45] S. Smith and C. Cheng. Slack-based heuristics for constraint satisfaction scheduling. In Proc. of the Eleventh National Conference on Artificial Intelligence, 1993.
- [46] R. Stallman and G. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer aided circuit analysis. *Artificial Intelligence*, 9(2), 1977.
- [47] E. Taillard. Benchmarks for basic scheduling problems. *Journal of Operational Research*, 64(2), 1993.
- [48] E. Taillard. Parallel taboo search technique for the jobshop scheduling problem. ORSA J. Comput., to appear., 1994.
- [49] R. Vaessens, E. Aarts, and J. Lenstra. Job shop scheduling by local search. Technical Report COSOR 94-05, Eindhoven University of Technology, 1994.
- [50] R. Walker. An enumerative technique for a class of combinatorial problems. In Combinatorial analysis; Proceedings of Symposium in Applied Mathematics, Vol. X, 1960.
- [51] D. Waltz. Understanding line drawings of scenes with shadows. In P. Winston, editor, *The psychology of computer vision*. McGraw-Hill, New York, 1975.
- [52] D. Wilkins. Practical Planning: Extending the Classical AI Planning Paradigm. Morgan Kaufman, San Mateo, California, 1988.
- [53] R. Zabih and D. McAllester. A rearrangement search strategy for determining propositional satisfiability. In Proc. 7th AAAI, 1988.