UNIVERSITY OF CALIFORNIA
Irvine

Algorithms and Heuristics for Constraint Satisfaction Problems

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Information and Computer Science

by

Daniel Hunter Frost

Committee in charge:
Professor Rina Dechter, Chair
Professor Dennis Kibler
Professor Richard H. Lathrop

1997

The dissertation of Daniel Hunter Frost is approved,
and is acceptable in quality and form for
publication on microfilm:

Committee Chair

University of California, Irvine

1997

To Kathy

Contents

List of Figures

List of Tables

Acknowledgements

Six years of graduate school would never have started, continued happily, or ended successfully without the help of many people. The love, encouragement, and support from my parents, Hunter and Carolyn Frost, and my grandmother, Rhoda Truax Silberman, has made a world of difference. My wonderful wife and friend Kathy has helped me every step of the way. Our daughters Sarah and Betsy made the last five years much more interesting and enjoyable.

I am indebted to Professor Rina Dechter, my advisor, who introduced me to the topic of constraint satisfaction, provided the first challenge – to solve problems with more than one hundred variables – that got me started on the course that led to this dissertation, and provided just the right amounts of prodding and leeway throughout my research. I would like to thank my committee, Professor Dennis Kibler and Professor Rick Lathrop, for their support and reading of my work. The National Science Foundation and the Electric Power Research Institute provided financial support for my Research Assistantship.

I've enjoyed many pleasant hours with my friends and crewmates Jeui Chang, Karl Kilborn, Chris Merz, Scott Miller, Harry Yessayan, and Hadar Ziv. To my colleagues in the same "boat" as me, Kalev Kask, Irina Rish, and Eddie Schwalb, I say thank you and clear sailing to port. Thanks also to Heidi Skolnik and Lluís Vila for your friendship.

Curriculum Vitae

1978	A.B. in Folklore and Mythology, Harvard University.
1985	M.S. in Computer Science, Metropolitan College, Boston University.
1993	M.S. in Information and Computer Science, University of California, Irvine.
1997	Ph.D. in Information and Computer Science, University of California, Irvine.
	Dissertation: *Algorithms and Heuristics for Constraint Satisfaction Problems*

Abstract of the Dissertation

Algorithms and Heuristics
for Constraint Satisfaction Problems

by

Daniel Hunter Frost

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1997

Professor Rina Dechter, Chair

This dissertation presents several new algorithms and heuristics for constraint satisfaction problems, as well as an extensive and systematic empirical evaluation of these new techniques. The goal of the research is to develop algorithms which are effective on large and hard constraint satisfaction problems.

The dissertation presents several new combination algorithms. The BJ+DVO algorithm combines backjumping with a dynamic variable ordering heuristic that utilizes a forward checking style look-ahead. A new heuristic for selecting a value called Look-ahead Value Ordering (LVO) can be combined with BJ+DVO to yield BJ+DVO+LVO. A new learning, or constraint recording, technique called jump-back learning is described. Jump-back learning is particularly effective because it takes advantage of effort that has already been expended by BJ+DVO. This type of learning can be combined with either BJ+DVO or BJ+DVO+LVO. Learning is shown to be helpful for solving optimization problems that are cast as a series

of constraint problems with successively tighter cost-bound constraints. The constraints recorded by learning are used in subsequent attempts to find a solution with a lower cost-bound.

The algorithms are evaluated in the dissertation by their performance on three types of problems. Extensive use is made of random binary constraint satisfaction problems, which are generated according to certain parameters. By varying the parameters across a range of values it is possible to assess how the relative performance of algorithms is affected by characteristics of the problems. A second random problem generator creates instances modeled on scheduling problems from the electric power industry. Third, algorithms are compared on a set of DIMACS Challenge problems drawn from circuit analysis.

The dissertation presents the first systematic study of the empirical distribution of the computational effort required to solve randomly generated constraint satisfaction problems. If solvable and unsolvable problems are considered separately, the distribution of work on each type of problem can be approximated by two parametric families of continuous probability distributions. Unsolvable problems are well fit by the lognormal distribution function, while the distribution of work on solvable problems can be roughly modelled by the Weibull distribution. Both of these distributions can be highly skewed and have a long, heavy right tail.

Chapter 1
Introduction

1.1 Introduction

It would be nice to say to a computer, "Here is a problem I have to solve. Please give me a solution, or if what I'm asking is impossible, tell me so." Solving problems on a computer without programming: such is the stuff that dreams of Artificial Intelligence are made on. Such is the subject of this dissertation.

For many purposes, writing a computer program is an effective way to give instructions to a computer. But when it is easy to state the desired result, and difficult to specify the process for achieving it, another approach may be preferred. Constraint satisfaction is a framework for addressing such situations, as it permits complex problems to be stated in a purely declarative manner. Real-world problems that arise in computer vision, planning, scheduling, configuration, and diagnosis [63] can be viewed as constraint satisfaction problems (CSPs).

Many algorithms for finding solutions to constraint satisfaction problems have been developed since the 1970s. These techniques can be broadly divided into two categories, those based on backtracking search and those based on constraint propagation. Although the two approaches are often studied separately, several algorithms which combine them have been devised. In this dissertation we present several new algorithms and heuristics, most of which have both search and constraint propagation components. The basic algorithm in most of the research described here is called BJ+DVO. This new algorithm combines two well-proven techniques, backjumping and a dynamic variable ordering heuristic. We also develop several effective extensions to BJ+DVO: BJ+DVO+LVO, which uses a new value ordering heuristic; and BJ+DVO+Learning, which integrates a new variety of constraint recording learning. Additionally, we show the effectiveness of extensive constraint propagation when integrated with bactracking in the BT+IAC algorithm.

The study of algorithms for constraint satisfaction problems has often relied upon experimentation to compare the relative merits of different algorithms or heuristics. Experiments for the most part have been based on simple benchmark problems, such as the 8-Queens puzzle, and on randomly generated problem instances. In the 1990s, the experimental side of the field has blossomed, due to several developments, including the increasing power of inexpensive computers and the identification of the "cross-over" phenomenon, which has enabled hard random problems to be generated easily. A major contribution of this thesis is the report of systematic and extensive experiments. Most of our experiments were conducted with parameterized random binary problems. By varying the parameters of the random problem generator, we can observe how the relative strength of different algorithms is affected by the type of problems they are applied to. We also define a class of random problems that model scheduling problems in the electric power industry, and report the performance of several algorithms on those constraint satisfaction problems. To complement these random problems, we report on experiments with benchmark problems drawn from the study of circuits, and which have been used by other researchers. These experiments show that the new algorithms we present can improve the performance of previous techniques by an order of magnitude on many instances.

Conducting and reporting experiments with large numbers of random problem instances raises several issues, including how to summarize accurately the results and how to determine an adequate number of instances to use. These issues are challenging in the field of constraint satisfaction problems because in large sets of random problems a few instances are always much harder to solve than the others. We investigated the the distribution of our algorithms' computational effort on random problems, and found that it can be summarized by two standard probability distributions, the Weibull distribution for solvable problems, and the lognormal distribution for unsolvable CSPs. These distributions can be used to improve the reporting of experiments, to aid in the interpretation of experiments, and possibly to improve the design of experiments.

In the remainder of this Introduction we describe more fully the constraint satisfaction problem framework, and then provide an overview of the thesis and a summary of our results.

1.2 Background

1.2.1 Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) consists of a set of n variables, $X_1, \ldots, X_n$, and a set of constraints. For each variable X_i a *domain* D_i with d elements $\{x_{i1}, x_{i2}, \ldots, x_{id}\}$ is specified; a variable can only be assigned a value from its domain. A *constraint* specifies a subset of the variables and which combinations of value assignments are allowed for that subset. A constraint is a subset of the Cartesian product $D_{i_1} \times \ldots \times D_{i_j}$, consisting of all tuples of values for a subset $(X_{i_1}, \ldots, X_{i_j})$ of the variables which are compatible with each other. A constraint can also be represented in other ways which may be more convenient. For instance, if X_1, X_2, and X_3 each have a domain consisting of the integers between 1 and 10, a constraint between them might be the algebraic relationship $X_1 + X_2 + X_3 > 15$.

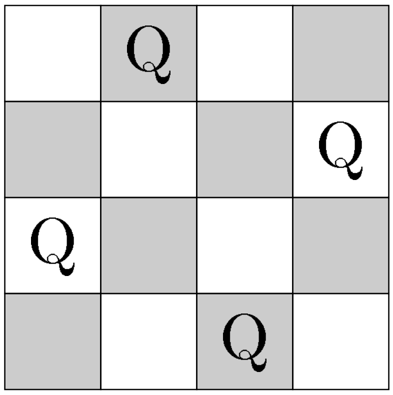


Figure 1.1: A solution to the 4-Queens problem. Each "Q" represents a queen. No two queens share the same row, column, or diagonal.

A *solution* to a CSP is an assignment of values to all the variables such that no constraint is violated. A problem that has a solution is termed *satisfiable* or *consistent*; otherwise it is *unsatisfiable* or *inconsistent*. Sometimes it is desired to find all solutions; in this thesis, however, we focus on the task of finding one solution, or proving that no solution exists. A *unary* constraint specifies one variable. A *binary* constraint pertains to two variables. A binary CSP is one in which each constraint is unary or binary. A constraint satisfaction problem can be represented by a *constraint graph* that has a node for each variable and an arc connecting each pair of variables that are contained in a constraint. In general, constraint satisfaction tasks are computationally intractable (NP-hard).

As a concrete example of a CSP, consider the N-Queens puzzle. An illustration of the 4-Queens puzzle is shown in Fig. 1.1. The desired result is easy to state: place N chess queens on an N by N chess board such that no two queens are in the same row, column, or diagonal. In comparison to this short statement of the goal, a specification of a computer program that solves the N-Queens puzzle would be quite lengthy, and would deal with data structures, looping, and possibly function calls and recursion. The usual encoding of the N-Queens problem as a CSP is based on the observation that any solution will have exactly one queen per row. Each row is represented by a variable, and the value assigned to each variable, ranging from 1 to N, indicates the square in the row that has a queen. A constraint exists between each pair of variables. Fig. 1.2 shows a constraint satisfaction representation of the 4-Queens problem, using this scheme. The four rows are represented by variables R1, R2, R3, R4. The four squares in each row, on one of which a queen must be placed, are called c1, c2, c3 and c4. The constraints are expressed as relations, that is, tables in which each row is an allowable combination of values. The task of a CSP algorithm is to assign a value from {c1, c2, c3, c4} to each variable R1, R2, R3, R4, such that for each pair of variables the respective pair of values can be found in the corresponding relation. The constraint graph of the N-Queens puzzle is fully connected, for any value of N, because the position of a Queen on one row affects the permitted positions of Queens on all other rows.

Another example of a constraint satisfaction problem is Boolean satisfiability (SAT). In SAT the goal is to determine whether a Boolean formula is satisfiable. A Boolean formula is composed of Boolean variables that can take on the values *true* and *false*, joined by operators such as $\vee$ (and), $\wedge$ (or), $\neg$ (negation), and "()" (parentheses). For example, the formula

$$(P \vee Q) \wedge (\neg P \vee \neg S)$$

is satisfiable, because the assignment (or "interpretation") (P=*true*, Q=*true*, S=*false*) makes the formula true.

1.2.2 Methods for Solving CSPs

Two general approaches to solving CSPs are search and deduction. Each is based on the idea of solving a hard problem by transforming it into an easier one. Search works in general by guessing an operation to perform, possibly with the aid of a heuristic, A good guess results in a new state that is nearer to a goal. For CSPs, search is exemplified by backtracking, and the operation performed is to extend a partial solution by assigning a value to one more variable. When a variable is encountered such that none of its values are consistent with the partial

Variables: R1, R2, R3, R4. (rows)					
Domain of each variable: {c1, c2, c3, c4} (columns)					
Constraint relations (allowed combinations):					
R1 R2	R1 R3	R1 R4	R2 R3	R2 R4	R3 R4
c1 c3	c1 c2	c1 c2	c1 c3	c1 c2	c1 c3
c1 c4	c1 c4	c1 c3	c1 c4	c1 c4	c1 c4
c2 c4	c2 c1	c2 c1	c2 c4	c2 c1	c2 c4
c3 c1	c2 c3	c2 c3	c3 c1	c2 c3	c3 c1
c4 c1	c3 c2	c2 c4	c4 c1	c3 c2	c4 c1
c4 c2	c3 c4	c3 c1	c4 c2	c3 c4	c4 c2
	c4 c1	c3 c2		c4 c1	
	c4 c3	c3 c4		c4 c3	
		c4 c2			
		c4 c3			

Figure 1.2: The 4-Queens puzzle, cast as a CSP.

solution (a situation referred to as a *dead-end*), backtracking takes place. The algorithm is time exponential, but requires only linear space.

Improvements of backtracking algorithm have focused on the two phases of the algorithm: moving forward (look-ahead schemes) and backtracking (look-back schemes) [15]. When moving forward, to extend a partial solution, some computation is carried out to decide which variable and value to choose next. For variable ordering, a variable that maximally constrains the rest of the search space is preferred. For value selection, however, the least constraining value is preferred, in order to maximize future options for instantiation [40, 18, 71].

Look-back schemes are invoked when the algorithm encounters a dead-end. They perform two functions. First, they decide how far to backtrack, by analyzing the reasons for the dead-end, a process often referred to as *backjumping* [31].

Second, they can record the reasons for the dead-end in the form of new constraints, so that the same conflicts will not arise again. This procedure is known as *constraint learning* and *no-good recording* [84, 15, 5].

Deduction in the CSP framework is known as constraint propagation or consistency enforcing. The most basic consistency enforcing algorithm enforces *arc-consistency*, also known as *2-consistency*. A constraint satisfaction problem is arc-consistent if every value in the domain of every variable is consistent with at least one value in the domain of any other selected variable [60, 50, 25]. In general, i-consistency algorithms ensure that any consistent instantiation of $i-1$ variables can be extended to a consistent value of any ith variable. A problem that is i-consistent for all i is called *globally* consistent. Because consistency enforced during search is applied to "future" variables, which are currently unassigned, it is used as a "look-ahead" mechanism.

In addition to backtracking search and constraint propagation, two other approaches are stochastic local search and structure-driven algorithms. Stochastic methods move in a hill-climbing manner in the space of complete instantiations [56]. In the CSP community the most prominent stochastic method is GSAT [58]. This algorithm improves its current instantiation by "flipping" a value of a variable that will maximize the number of constraints satisfied. Stochastic search algorithms are incomplete and cannot prove inconsistency. Nevertheless, they are often extremely successful in solving large and hard satisfiable CSPs [78].

Structure-driven algorithms cut across both search and consistency-enforcing algorithms. These techniques emerged from an attempt to characterize the topology of constraint problems that are tractable. *Tractable classes* were generally recognized by realizing that enforcing low-level consistency (in polynomial time) guarantees global consistency for some problems. The basic graph structure that supports tractability is a tree [51]. In particular, enforcing arc-consistency on a tree-structured network ensures global consistency along some ordering. Most graph-based techniques can be viewed as transforming a given network into an equivalent tree. These techniques include *adaptive-consistency, tree-clustering*, and *constraint learning*, all of which are exponentially bounded by the *tree-width* of the constraint graph [25, 18, 19]; the *cycle-cutset* scheme, which separates a graph into tree and non-tree components and is exponentially bounded by the constraint graph's *cycle-cutset* [15]; the *bi-connected component method*, which is bounded by the size of the constraint graph's largest component [25]; and backjumping, which is exponentially bounded by the depth of the graph's depth-first-search tree [16]. See [16] for details and definitions. The focus of this thesis is on complete search algorithms such as backtracking.

1.3 Methodology

1.3.1 Random problem generators

Our primary technique in this dissertation for evaluating or comparing algorithms is to apply the algorithms to parameterized, randomly generated, binary CSP instances. A CSP generator is a computer program that uses pseudo-random numbers to create a practically inexhaustible supply of problem instances with defined characteristics such as number of variables and number of constraints. Our generator takes four parameters:

- N, the number of variables;
- D, the size of each variable's domain;
- C, an indicator of the number of constraints; and
- T, an indicator of the tightness of each constraint.

We often write the parameters as $\langle N, D, C, T\rangle$, e.g. $\langle 20, 6, .9789, .167\rangle$. The random problems all have N variables. Each variable has a domain with D elements. Each problem has $C \times N \times (N-1)/2$ binary constraints. In other words, C is the proportion of possible constraints which exist in the problem. We use C' to refer to the actual number of constraints. Each binary constraint permits $(1 - T) \times D^2$ value pairs. The constraints and the value pairs are selected randomly from a uniform distribution. This generator is the CSP analogue of Random K-SAT [58] for satisfiability problems.

Significant limitations to the use of a random problem generator should be noted. The most important is that the random problems may not correspond to the type of problems which a practitioner actually encounters, risking that our results are of little or no relevance. We believe, however, that experiments with random problems do reveal interesting characteristics about the algorithms we study. Our emphasis is primarily on how algorithm performance changes in response to changing characteristics of the generated problems, and on how different algorithms compare on different classes of problems. Another hazard with computer generated problems is that subtle biases, if not outright bugs, in the implementation may skew the results. The best safeguard against such bias is the repetition of our experiments, or similar ones, by others; to facilitate such repetition we have made our instance generating program available by FTP, and it has been evaluated and adopted by several other researchers.

1.3.2 Performance measures

Three statistics are commonly used to measure the performance of an algorithm on a single CSP instance: CPU time, consistency checks, and search space size (nodes). CPU time (in this work always reported in seconds on a SparcStation 4 with a 110 MHz processor) is the most fundamental statistic, since the goal of most research into CSP algorithms is to reduce the time required to solve CSPs. Every aspect of the computer program that implements an algorithm influences the resulting CPU time, which is both the strength and the limitation of this measure. Reported CPU times for two algorithms implemented by different programmers and run on different machines are generally incomparable. We have endeavored to make CPU time as unbiased and useful a statistic as possible. All CPU times reported in this thesis are based on the same computer program. Different algorithms are implemented with different blocks of code, but the same underlying data structures are used throughout, and as much code as possible is shared. There is still some risk that one algorithm may benefit from a more clever or efficient implementation than another, but the risk has been minimized.

A second measure of algorithm performance is the number of consistency checks made while solving the problem. A consistency check is a test of whether a constraint is violated by the values currently assigned to variables. Since the consistency check subroutine is performed frequently in any CSP algorithm, counting the number of times it is invoked is a good measure of the overall work of the algorithm.

A third measure is the size of the search space explored by the algorithm. Each assignment of a value to a variable counts as one "node" in the search tree. Knowing the size of the search space gives a sense of how many assignments the algorithm made that did not lead to a solution. Comparing the ratio of consistency checks to nodes for different algorithms is a good way to see the relative amount of work per assignment that each algorithm does.

In general we are concerned with measures of computer time, but not of space (memory). Backtracking search generally requires space that is linear in the size of the problem. Our implementation uses tables that have size of approximately n^2d^2, where n is the number of variables and d is the number of values per variable, but the program runs easily in main memory for the size of problems with which we have experimented. In Chapter 7 we discuss a learning algorithms that potentially require exponential space.

1.3.3 The CSP cross-over point

In 1991, Cheeseman *et al.* [10] observed a sharp peak in the average problem difficulty for several classes of NP-hard problems, random instance generators, and particular values of the parameters to the generator. Mitchell *et al.* [58] extended this observation to Boolean satisfiability. Specifically, they observe experimentally that for 3-SAT problems (each clause has 3 variables), the average problem hardness peaks when the ratio of clauses to variables is about 4.3. Moreover, this ratio corresponds to a combination of parameters which yields an equal number of satisfiable and unsatisfiable instances. With fewer than $4.3N$ clauses, almost all problems have solutions and these solutions are easy to find. With more clauses, almost no problems have solutions, and it is easy to prove unsatisfiability. A set of parameters which yields an equal number of satisfiable and unsatisfiable problems, and which corresponds to a peak in average problem hardness, is often called a "cross-over point", and the phenomenon in general is sometimes referred to as a "phase transition"[1].

The existence of a cross-over point for binary CSPs and the random problem generator described above was shown empirically by Frost and Dechter [27]. Similar observations with a different generator are reported in [69]. An illustration of the cross-over point for binary CSPs appears in Fig. 1.3. For fixed values of N, D, and T, 10,000 problems were generated at varying values of C. The cross-over point is between C=.0505, where 55% of the problems are solvable and the average number of consistency checks is 3,303, and C=.0525, with 46% solvable and 3,398 average consistency checks. The figure illustrates that, for these values of N, D, and T, if C is chosen to be less than .04 or greater than .08, the problems will tend to be quite easy. Increasing N seems to make the peak higher and narrower

[1]The term phase transition is by analogy with physics, e.g. ice turns to water at a certain critical temperature. The evidence for a similar abrupt change in the characteristics of the random problems, and not just in their average hardness, is lacking at this point, I believe. Therefore my use of the term phase transition does not imply any model of an underlying explanation.

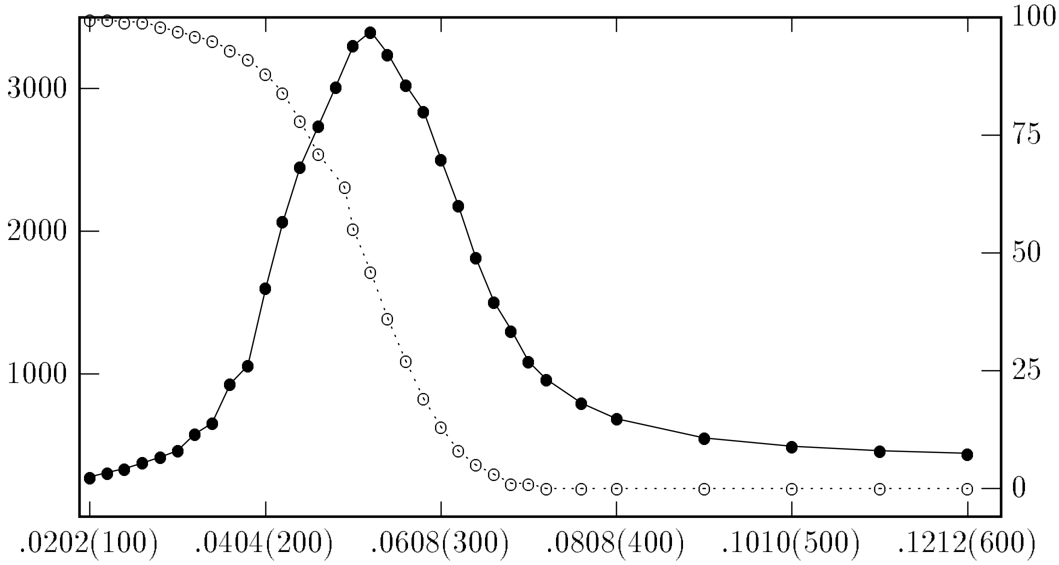


Figure 1.3: The cross-over point. Results from a set of experiments using algorithm BJ+DVO and parameters $N = 100$, $D = 3$, $T = .222$, and varying values of C. Bullets ($\bullet$) indicate average consistency checks over 10,000 instances (left hand scale). Circles ($\circ$) indicate percentage solvable (right hand scale). x-axis is parameter C (with actual number of constraints $CN(N-1)/2$ in parentheses).

[79], and only a small range of C (or whichever parameter is being varied) leads to problems which aren't almost trivially easy. Before the discovery of the phase transition phenomenon and its relation to the 50% solvable point, it was therefore to difficult to locate and experiment with hard random problems.

Because we generate CSPs based on four parameters, the situation is somewhat more complex than for 3-SAT: if any three parameters are fixed and the fourth is varied a phase transition can be observed. It might be more accurate to speak of a cross-over "ridge" in a five-dimensional space where the "height" dimension is the average difficulty, and the CSP four parameters make the other four dimensions. In Fig. 1.3 the five dimensions are reduced to two by holding N, D, and T constant.

1.4 Related Work

From the 1970's through the early 1990's, several empirical studies of constraint satisfaction algorithms based on random problems were conducted, notably by Gaschnig [31], Haralick and Elliott [40], Nudel [65], Dechter [15], Dechter and Meiri [17], and Bessiére [6]. Stone and Stone [85] and Nadel [62] conducted experiments based on the N-Queens problem.

The idea of combining two or more CSP algorithms to create a hybrid algorithm has received increasing attention in recent years. Nadel [62] describes a systematic approach to combining backtracking with varying degrees of partial arc-consistency. The approach was continued by Prosser [68], who considers several backtracking-based algorithms. Ginsberg's Dynamic Backtracking algorithm [36, 4] combines several techniques into a tightly integrated algorithm. Sabin and Freuder [74] show that arc-consistency can be combined effectively with search.

1.5 Overview of the Dissertation

The dissertation has ten chapters. Chapter 2 is a review of standard algorithms from the literature. The chapter is both a literature review, although no attempt has been made at completeness, and an introduction to the algorithms and heuristics on which the research in this thesis is based.

In Chapter 3 we address an issue that has been unresolved in the CSP research community for many years: what is the best way to summarize and present the results from experiment on many random CSP instances? A rightward skew in the empirical distribution of search space or any other measure makes standard statistics, such as the mean, median, or standard deviation, difficult to interpret. We show empirically that the distribution of effort required to solve CSPs can be approximated by two standard families of continuous probability distribution functions. Solvable problems can be modelled by the Weibull distribution, and unsolvable problems by the lognormal distribution. These distributions fit equally well over a variety of backtracking based algorithms. By reporting the parameters of the Weibull and lognormal distributions that best fit the empirical distribution, it is possible to accurately and succinctly convey the experimental results. We also show that the mathematical derivation of the lognormal and Weibull distribution functions parallels several aspects of CSP search.

Chapters 4 through 8 present several new algorithms and heuristics, together with extensive empirical evaluation of their performance. In Chapter 4 we describe an algorithm, dubbed BJ+DVO, which combines three different techniques for solving constraint satisfaction problems: backjumping, forward checking, and dynamic variable ordering. We show empirical results indicating that the combination algorithm is significantly superior to its constituents. BJ+DVO forms the platform for two additional contributions, described in Chapters 6 and 7, which are shown able to improve its performance.

Chapter 5 is a comparative study of several algorithms that enforce different amounts of consistency during search. We compare four algorithms, forward checking, partial looking ahead, full looking ahead, and arc-consistency, and show empirically that the relative performance of these algorithms is strongly influenced by the tightness of the problem's constraints. In particular, we show that on problems with a large number of loose constraints, it was best to do the least amount of consistency enforcing. When there were relatively few constraints and they were tight, more intensive consistency enforcing paid off. We also propose and evaluate three new heuristics which can usefully control how much time the search algorithm should spend looking ahead. We conclude that none of these heuristic dominates the algorithms without heuristics. Finally, the chapter describes a technique called AC-DC, for arc-consistency domain checking, which improved the integration of an arc-consistency algorithm with backtracking search.

In Chapter 6 we describe a new value ordering heuristic called look-ahead value ordering (LVO). Ideally, if the right value for each variable is known, the solution to a CSP can be found with no backtracking. In practice, even a value ordering heuristic that offers a slight improvement over random guessing can be quite helpful in reducing average run time. LVO uses the information gleaned from forward checking style look-ahead to guide the value ordering. We show that LVO improves the performance of BJ+DVO on hard problems, and that, surprisingly, this heuristic is helpful even on instances that do not have solutions, due to its interaction with backjumping.

Chapter 7 presents jump-back learning, a new variant of CSP learning [15]. When a dead-end is encountered, the search algorithm learns by recording a new constraint that is revealed by the dead-end. Backjumping also maintains information that allows it, on reaching a dead-end, to jump back over several variables. Recognizing that backjumping and learning can make use of the same information inspired the development of jump-back learning. We show that when combined with BJ+DVO it is superior both to other learning schemes available in the literature and to BJ+DVO without learning on many problems.

Chapter 8 synthesizes the results of Chapters 4 through 7. A new algorithm, which combines look-ahead value ordering and jump-back learning, is described. This combination algorithm and five of the best algorithms from earlier chapters are compared on sets of random problems with large values of N, and on a suite of six DIMACS Challenge benchmark problems. We show that results on these non-random benchmarks largely confirm the observations made in earlier chapters based on random problems. We also show that, measured by CPU time, our algorithms' performance is on par with that of other systems being used for experimental research.

In Chapter 9 we show how scheduling problems of interest to the electric power industry can be formalized as constraint satisfaction problems. Producing an optimal schedule for preventative maintenance of generating units, while ensuring a sufficient supply of power to meet estimated demand, is a well-studied problem of substantial economic importance to every large electric power plant. We describe a random problem generator that creates maintenance scheduling CSPs, and report the performance of six algorithms two sets of these random problems.

We also describe in Chapter 9 a new use of jump-back learning that aids in the solution of optimization problems in the CSP framework. Constraint satisfaction problems are decision problems. Optimization problems, such as finding the best schedule, have an objective function which should be minimized. One way to find an optimal solution with CSP techniques is to solve a single problem multiple times, each time with a new constraint that enforces a slightly lower bound on the maximum acceptable value of the objective function. With learning, constraints learned during one "pass" of the problem can be applied again later. Our empirical results show this technique is effective on maintenance scheduling problems.

In Chapter 10 we conclude the dissertation by summarizing the contributions made, and suggest some promising directions for further research. We recapitulate that the goal of the thesis is to advance the study of algorithms and heuristics for constraint satisfaction problems by introducing several new approaches and carefully evaluating them on a variety of challenging problems.

Chapter 2

Algorithms from the Literature

2.1 Overview of the Chapter

In this chapter we review several standard algorithms and heuristics for solving constraint satisfaction problems. "Standard" is meant to convey that the algorithms are well-known and have formed the basis for the development of other algorithms. Thus the chapter is both a literature review, although no attempt has been made at completeness, and an introduction to several algorithms for CSPs. The emphasis is on algorithms and heuristics which we draw upon in later chapters. The organization of this chapter is motivated by the structure of the algorithms and heuristics, and not by the historical order in which they were developed. Of course, to a large extent the history of CSP algorithms has seen an increase in complexity and sophistication.

The search algorithms in this chapter are all based on backtracking, a form of depth-first search which abandons a branch when it determines that no solutions lie further down the branch. It makes this determination by testing the values chosen for variables against a set of constraints. A variation of backtracking called backmarking explores the same search tree as backtracking, but maintains two tables which summarize the results of earlier constraint tests, thus reducing the total number that need to be made. Another modification to backtracking is called backjumping. Three versions of backjumping are presented, each of which offers a successively greater ability to bypass sections of the search space which cannot lead to solutions. Some backtracking-based algorithms interleave a certain amount of consistency propagation. In this chapter we review three in this category, forward checking, Waltz's algorithm, and Gaschnig's DEEB. The last section of the chapter focusses on heuristics for variable ordering.

A uniform style for presenting each algorithm is adopted, in order to highlight both the similarities and the differences between methods. We do not describe many of the mechanics of dealing with the necessary data structures; although these mechanics are of importance and some interest they are incidental to the structure of the underlying algorithms. It is also worth noting that most of the algorithms described in this chapter were originally described recursively. Since processing a CSP with n variables can be approached as processing one variable and then proceeding to a sub-CSP with $n-1$ variables, a recursive formulation for many algorithms is natural. Nevertheless, we do not use recursion in this chapter, or elsewhere in the thesis. Partially this choice reflects the current style — recursion seems to have diminished popularity in the 1990's. It also enables a more explicit statement of the control structure. (See [68] for similar arguments against using recursion in pseudo-code.)

2.2 Definitions

As in Chapter 1, a *constraint satisfaction problem* (CSP) consists of a set of n variables, $X_1, \ldots, X_n$, and a set of constraints. For each variable X_i a *domain* $D_i = \{x_{i1}, x_{i2}, \ldots, x_{id}\}$ with d elements is specified; a variable can only be assigned a value from its domain. A *constraint* specifies a subset of the variables and which combinations of value assignments are allowed for that subset. A constraint is a subset of the Cartesian product $D_{i_1} \times \ldots \times D_{i_j}$, consisting of all tuples of values for a subset $(X_{i_1}, \ldots, X_{i_j})$ of the variables which are compatible with each other. A constraint can also be represented in other ways which may be more convenient. For instance, if X_1, X_2, and X_3 each have a domain consisting of the integers between 1 and 10, a constraint between them might be the algebraic relationship $X_1 + X_2 + X_3 > 15$.

A *solution* to a CSP is an assignment of values to all the variables such that no constraint is violated. A problem that has a solution is termed *satisfiable* or *consistent*; otherwise it is *unsatisfiable* or *inconsistent*. Sometimes it is desired to find all solutions; in this thesis, however, we focus on the task of finding one solution, or proving that no solution exists. A *binary* CSP is one in which each constraint involves at most two variables. A constraint satisfaction problem can be represented by a *constraint graph* that has a node for each variable and an arc connecting each pair of variables that are contained in a constraint.

A variable is called *instantiated* when it is assigned a value from its domain. A variable is called *uninstantiated* when no value is currently assigned to it. Reflecting the backtracking control strategy of assigning values to variables one at a time, we sometimes refer to instantiated variables as *past* variables and uninstantiated variables as *future* variables. We use "$X_i{=}x_j$" to denote that the variable X_i is instantiated with the value x_j, and "$X_i \leftarrow x_j$" to indicate the act of instantiation.

The variables in a CSP are often given an order. We denote by $\vec{x}_i$ the instantiated variables up to and including X_i in the ordering. If the variables were instantiated in order $(X_1, X_2, \ldots, X_n)$, then $\vec{x}_i$ is shorthand for the notation $(X_1{=}x_1, X_2{=}x_2, \ldots, X_i{=}x_i)$.

A set of instantiated variables $\vec{x}_i$ is *consistent* or *compatible* if no constraint is violated, given the values assigned to the variables. Only constraints which refer exclusively to instantiated variables X_1 through X_i are considered; if one or more variables in a constraint have not been assigned values then the status of the constraint is indeterminate. A value x for a single variable X_{i+1} is consistent or compatible relative to $\vec{x}_i$ if assigning $X_{i+1} = x$ renders $\vec{x}_{i+1}$ consistent.

A variable X_i is a *dead-end* when no value in its domain is consistent with $\vec{x}_{i-1}$. We distinguish two types of dead-ends. X_i is a *leaf* dead-end if there are constraints prohibiting each value in D_i, given $\vec{x}_{i-1}$. X_i is found to be an *interior* dead-end when some values in D_i are compatible with $\vec{x}_{i-1}$, but the subtree rooted at X_i does not contain a solution. Different algorithms may define or test for consistency in different ways. The term dead-end comes from analogy with searching through a maze. At a dead-end in a maze, one cannot go left, right, or forward, and must retrace one's steps.

The most basic consistency enforcing algorithm enforces *arc-consistency*. A constraint satisfaction problem is arc-consistent, or *2-consistent*, if every value in the domain of every variable is consistent with at least one value in the domain of any other selected variable [60, 50, 25]. In general, i-consistency algorithms guarantee that any consistent instantiation of $i-1$ variables can be extended to a consistent value of any ith variable.

An individual constraint among variables $(X_{i_1}, \ldots, X_{i_j})$ is called *tight* if it permits a small number of the tuples in the Cartesian product $D_{i_1} \times \ldots \times D_{i_j}$, and *loose* if it permits a large number of tuples. For example, assume variables X_1 and X_2 have the same domain, with at least three elements in it. The constraint $X_1{=}X_2$ is a tight constraint. Once one variable is assigned a value, only one choice exists for the other variable. On the other hand, the constraint $X_1 \neq X_2$ is a loose constraint, as instantiating one variable prohibits only one possible value for the other.

2.3 Backtracking

A simple algorithm for solving a CSP is *backtracking* [89, 37, 8]. Backtracking works with an initially empty set of consistent instantiated variables and tries to extend the set to a new variable and a value for that variable. If successful, the process is repeated until all variables are included. If unsuccessful, another value for the most recently added variable is considered. Returning to an earlier variable

Backtracking

1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable in the ordering. Set $D'_{cur} \leftarrow D_{cur}$.
2. (Choose a value.) Select a value $x \in D'_{cur}$ that is consistent with all previous variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$ (X_{cur} is a dead-end), go to 3.
   (b) Pop x from D'_{cur} (that is, select an arbitrary value and remove it from D'_{cur}).
   (c) For every constraint defined on X_1 through X_{cur}, test whether it is violated by $\vec{x}_{cur-1}$ and $X_{cur}{=}x$. If so, go to (a).
   (d) Instantiate $X_{cur} \leftarrow x$, and go to 1.
3. (Backtrack.) If X_{cur} is the first variable, exit with "inconsistent." Otherwise, set cur equal to the index of the previous variable. Go to 2.

Figure 2.1: The Backtracking algorithm.

in this way is called a backtrack. If that variable doesn't have any further values, then the variable is removed from the set, and the algorithm backtracks again. The simplest backtracking algorithm is called chronological backtracking because at a dead-end the algorithm returns to the immediately earlier variable in the ordering.

As presented in Fig. 2.1, the backtracking algorithm has three sections. The first is "step forward," in which a new variable is selected to be the current variable, denoted X_{cur}. If all variables have been assigned values, then the search process is complete and the algorithm returns with a solution. In the second section of the algorithm an attempt is made to assign a value to the current variable. The value chosen must not cause any constraints to be violated. If a compatible value is found, then control returns again to the first step, and another variable is chosen. If no compatible value could be found for the current variable, then the algorithm goes to the "backtrack" step, where it returns to the immediately previous variable. If the current variable is the first variable it is not possible to backtrack, and the algorithm returns with an indicator that it failed to find a consistent solution.

For simplicity in the pseudo-code, we consider each variable domain, D_i, to be a set. We can test whether the set is empty, that is, is $D_i = \emptyset$? We can remove one element of the set with a *pop* function. (Unless specified, the element is chosen arbitrarily.) In addition to the fixed value domains D_i, the algorithm employs mutable value domains D'_i, such that $D'_i \subseteq D_i$. D'_i holds the possibly proper subset of D_i which has not yet been examined under the current instantiation of variables X_1 through X_{i-1}. In other words, a value in the set $D_i - D'_i$ is either the current value assigned to X_i, is inconsistent with $\vec{x}_{i-1}$, or is consistent with $\vec{x}_{i-1}$ but does not lead to a solution. If the values of each D_i are ordered (e.g. they are the integers from 1 to d) and they are considered in this order, then it is not necessary to maintain the D'_i sets. Knowing the current value of a variable, we know that all previous values have been tried. (It may be convenient to use a value that is not in the domain to indicate that a variable is currently unassigned.) We describe backtracking with the D' sets because of the greater generality they afford, and because we will use the D' sets extensively in describing later algorithms, particularly those such as forward checking that "filter" the domain of uninstantiated variables. For the sake of uniform treatment, we describe backtracking with the D' sets.

Step 2 (c) in the backtracking algorithm is implemented by performing *consistency checks*, that is, tests of whether the variables in a constraint, as instantiated, are consistent with the constraint. Consistency checking is performed frequently and constitutes a major part of the work performed by any CSP algorithm. Hence a count of the number of consistency checks is a common measure of the overall work of the algorithm. The cost (in CPU time) of a consistency check depends on how the constraints are represented internally in the computer. If a constraint is stored as a list of compatible tuples, then the program will have to search through this list; sorting or indexes can be used to reduce the average time required. When the constraints are loose, it may be more efficient to store only the incompatible tuples. A technique that allows consistency checking in a fixed amount of time is to represent constraints as a table of boolean values, with as many dimensions as there are variables in the constraint. A fourth possibility is to represent a constraint with a procedure. When the constraint is an easily tested quality such as equality, this method is the most efficient. If the CSP is a binary CSP, in which each constraint pertains to at most two variables, then step 2 (c) can be stated as

> 2. (Choose a value.)
>
>   (c) For all $X_i, 1 \leq i < cur$, test if X_i as instantiated is consistent with $X_{cur} = x$. If not, go to (a).

Stating the test in this way takes advantage of the fact that there can be at most one binary constraint between two variables. The test of consistency can be more efficient with binary CSPs, since the number of consistency checks is bounded by the number of variables. The more general version of 2 (c) of Fig. 2.1 requires one test for each constraint. Frequently a CSP has more constraints than variables.

The actions of a search algorithm can be described by a search tree. We illustrate this with a toy example shown in Fig. 2.2. The problem in this figure is a small coloring problem, in which the goal is to assign a color to each variable such that connected variables do not share the same color. Fig. 2.3 shows part of the search tree expanded when backtracking processes the CSP described in Fig. 2.2, using the ordering $(X1, X2, X3, X4, X5, X6, X7)$. Note that the problem has no solution.

The rest of this chapter describes several algorithms and heuristics which augment basic backtracking. We can categorize the algorithms by which of backtracking's three sections they concentrate on. Backmarking reduces the number of consistency checks that are performed in step 2 (c). The three versions of backjumping we describe are all designed to improve the choice of backtrack variable in step 3. Forward checking changes step 2 (c) to test the adequacy of a value selection by making sure the value is compatible with at least one value in the domain of every future variable. Static and dynamic variable order heuristics attempt to improve the choice of a variable in step 1. Another important category of

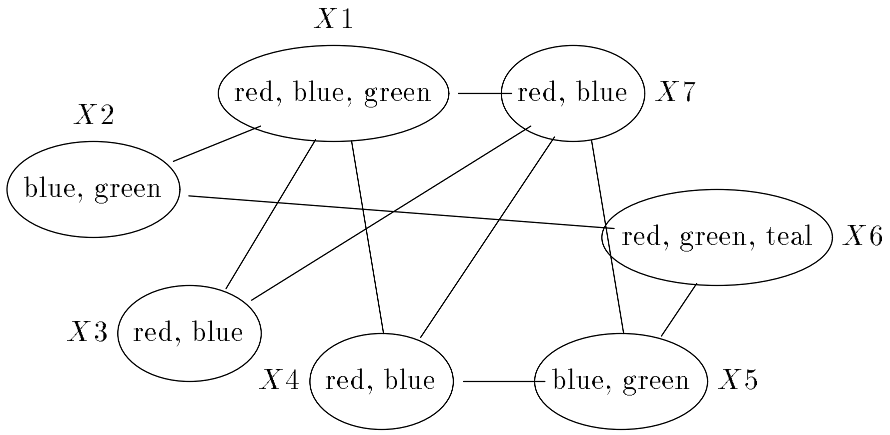


Figure 2.2: An example CSP; a modified coloring problem. The domain of each node is written inside the ellipse; note that not all nodes have the same domain. Arcs join nodes that must be assigned different colors.

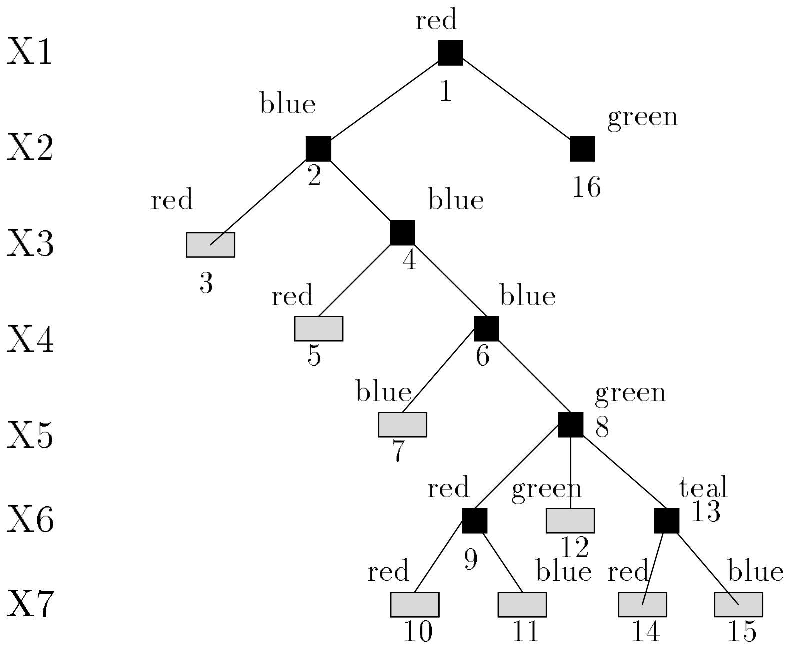


Figure 2.3: Part of the search tree explored by backtracking, on the example CSP in Fig. 2.2. Only the search tree below X1=red and X2=blue is drawn. A black square denotes an instantiation from which further search can continue. A gray rectangle denotes a value that is incompatible with some previous value. The nodes are numbered in the order in which they are popped in step 2 (b).

Backmarking (binary CSPs and static variable ordering only)

0. (Initialize tables.) Set all $M_{i,v} \leftarrow 0$; set all $L_i \leftarrow 0$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable in the ordering. Set $D'_{cur} \leftarrow D_{cur}$.
2. Select a value $x \in D'_{cur}$ that is consistent with all previous variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x_v from D'_{cur}. (v is the index of the domain value popped.)
   (c) If $M_{cur,v} < L_{cur}$, then go to (a).
   (d) Examine, in order, the past variables $X_i, L_{cur} \leq i < cur$; if X_i as instantiated conflicts with $X_{cur} = x_v$ then set $M_{cur,v} \leftarrow i$ and go to (a).
   (e) Instantiate $X_{cur} \leftarrow x_v$, set $M_{cur,v} \leftarrow cur$, and go to 1.
3. (Backtrack step.) If X_{cur} is the first variable, exit with "inconsistent." Otherwise, for all X_i after X_{cur+1}, if $cur < L_i$ then set $L_i \leftarrow cur$. Set $L_{cur} \leftarrow cur - 1$. Set cur equal to the index of the previous variable. Go to 2.

Figure 2.4: The Backmarking algorithm.

backtracking-based CSP algorithm consists of those that learn, or record additional constraints during search. These algorithms augment step 3.

2.4 Backmarking

A method to reduce consistency checking while backtracking is Gaschnig's *backmarking* [30, 40]. Backmarking requires that consistency checks be performed in the same order as variable instantiation. By keeping track of where consistency checks have succeeded or failed in the past, backmarking can eliminate the need to repeat unnecessarily checks which have been performed before and will again succeed or fail in the same way. Backmarking is restricted to binary CSPs and a static variable ordering. However, Bacchus and van Run [3] give a variation of backmarking that works with a dynamic variable ordering.

Backmarking requires two additional tables (see Fig. 2.4). The first table, with elements $M_{i,v}$, records the first variable that failed a consistency check with $X_i = x_v$. If $X_i = x_v$ is consistent with all earlier variables, then $M_{i,v} = i$. For instance, $M_{10,2} = 4$ means that X_4 as instantiated was found inconsistent with $X_{10} = x_2$, and that X_1, X_2 and X_3 did not conflict with $X_{10} = x_2$. The second table, with elements L_i, indicates the earliest variable which has changed its value since $M_{i,v}$ was set for X_i and any domain element v. If $M_{i,v} < L_i$, then the variable pointed to by $M_{i,v}$ has not changed, and $X_i = x_v$ will still fail when checked with $X_{M_{i,v}}$. Thus, there is no need to do any consistency checking and x_v can be rejected immediately. If $M_{i,v} \geq L_i$, then $X_i = x_v$ is consistent with all variables before X_{L_i}, and those checks can be skipped.

The structure of the Backmarking algorithm is almost identical to that of Backtracking. A step 0 has been added in which the new tables are initialized. Step 2 changes to reflect how the two tables are maintained and used. In step 2 (c), the backmarking tables are consulted and if an earlier variable is still instantiated with a value that conflicted with value x_v at an earlier point in the search then we can immediately reject x_v. Otherwise, the algorithm proceeds to 2 (d), checking compatibility only with variables which may have changed assignment since the last time X_{cur} was instantiated. If a conflict is found, this is recorded in table M. If no consistent for X_{cur} is found, the algorithm goes to step 3, where it updates the L table and then backtracks.

2.5 Backjumping

Backtracking (as well as Backmarking) can suffer from *thrashing*; the same dead-end can be encountered many times. If X_i is a dead-end, the algorithm will backtrack to X_{i-1}. Suppose a new value for X_{i-1} exists, but that there is no constraint between X_i and X_{i-1}. The same dead-end will be reached at X_i again and again until all values of X_{i-1} have been exhausted. For instance, the problem in Fig. 2.2 has a dead-end at $X7$ after the assignment $(X1 = red, X2 = blue, X3 = blue, X4 = blue, X5 = green, X6 = red)$. Backtracking returns to $X6$ and reinstantiates it as $X6 = teal$. But the same dead-end at $X7$ is re-encountered.

To reduce the amount of thrashing, an enhancement to backtracking called *backjumping* was proposed by Gaschnig in [31] (see Fig. 2.5). This algorithm is able to "jump" from the dead-end variable back to an earlier variable which, as instantiated, is a direct cause for the dead-end. When is a variable a direct cause of a dead-end? When the variable, plus zero or more other variables which precede it in the ordering, are instantiated in such a way that a constraint disallows some value (or values) of the dead-end variable. For example, imagine variables X_{10} and X_{20}, each with the domain $\{0, 1, 2\}$, and a constraint which permits any assignment to X_{10} and X_{20} except $(X_{10}{=}1, X_{20}{=}1)$. X_{10} and X_{20} also participate in other constraints. If X_{10} is instantiated to 1 and later X_{20} is a dead-end, then X_{10} is a cause of the dead-end because its assignment prohibits one value from X_{20}'s domain. In contrast, if X_{10} has the value 2 and X_{20} is a dead-end anyway, X_{10} is not a cause of the dead-end.

To locate a variable which is a cause of the dead-end, backjumping maintains an array $J_i, 1 \leq i \leq n$. J_i remembers the latest variable in the ordering that was tested for consistency with some value of X_i. If X_i is not a dead-end, then $J_i = i-1$. If X_i is a dead-end, then each value in D_i was tested for consistency with the earlier variables until some check failed, and J_i holds the index if the latest variable which is inconsistent with some value in D_i. It is critical that the order of consistency checking on instantiated variable be the same as the order of instantiation. This rule is easy to implement if all constraints are binary; with higher order constraints and a fixed variable ordering it is efficient to store or index the constraints in order of their second-to-last variable. For instance, suppose there are constraints

Gaschnig's Backjumping

1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable in the ordering. Set $D'_{cur} \leftarrow D_{cur}$. Set $J_{cur} \leftarrow 0$.
2. Select a value $x \in D'_{cur}$ that is consistent with all previous variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur}.
   (c) For $1 \leq i < cur$ (in ascending order): if $i > J_{cur}$ then set $J_{cur} \leftarrow i$; if $\vec{x}_i$ and $X_{cur}{=}x$ are inconsistent then go to (a).
   (d) Instantiate $X_{cur} \leftarrow x$ and go to 1.
3. (Backjump step.) If $J_{cur} = 0$ (there is no previous variable which shares a constraint with X_{cur}), exit with "inconsistent." Otherwise, select variable $X_{J_{cur}}$; call it X_{cur}. Go to 2.

Figure 2.5: Gaschnig's backjumping algorithm.

prohibiting $(X_2{=}a, X_6{=}b, X_{10}{=}c)$ and $(X_4{=}d, X_8{=}e, X_{10}{=}c)$. Assuming $X_2{=}a$, $X_4{=}d$, $X_6{=}b$, and $X_8{=}e$, then both constraints prohibit $X_{10}{=}c$. It is important that backjumping records X_6, not X_8, as preventing $X_{10}{=}c$, because changing the value assigned to X_8 does not make $X_{10}{=}c$ consistent.

If X_i is a dead-end, then we can be assured that all backtracking on variables between X_{J_i+1} and X_{i-1} will be fruitless because the cause of the dead-end at X_i is not addressed. The partial instantiation $\vec{x}_{J_i}$ causes every value for X_i to be inconsistent with some constraint, so changing variables after X_{J_i} will not eliminate the dead-end.

The first step of the Gaschnig's backjumping algorithm sets J_{cur} to 0, indicating that at this point no conflicts with X_{cur} have been found. In step 2 (c) J_{cur} is updated if a conflict is found at a later variable than any previous conflict. If a value is consistency with all previous instantiated variables, then J_{cur} will have the value $cur - 1$. Step 3 is now a backjump (instead of backtrack) step. After a

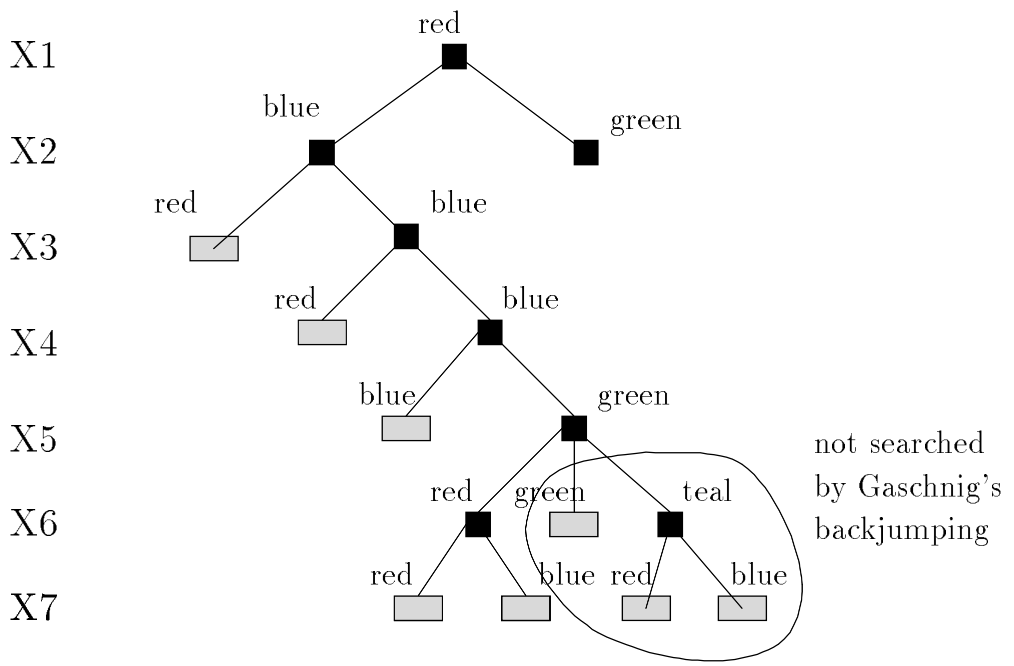


Figure 2.6: The search space explored by Gaschnig's backjumping, on the example CSP in Fig. 2.2. The nodes surrounded by the circle are explored by backtracking but not by Gaschnig's backjumping.

dead-end, the algorithm returns to $X_{J_{cur}}$, which is a variable that is a direct cause of the dead-end. If J_{cur} is 0, then either X_{cur} is the first variable, or there are no constraints between X_{cur} and the earlier variables. In either case, the problem has no solution. Recognizing the second case is particularly important in CSPs which consist of disjoint subproblems.

Referring again to the problem in Fig. 2.2, at the dead-end for $X7$, J_7 will be 3, because value *red* for $X7$ was ruled out by $X1$ and value *blue* was ruled out by $X3$, and no later variable had to be examined. The other values for $X6$ therefore do not need to be explored (see Fig. 2.6). On returning to $X3$, there are no further values to try ($D'_3 = \emptyset$). Since $J_3 = 2$, the next variable examined will be $X2$.

Graph-based Backjumping

0. (Initialize parent sets.) Compute P_i for each variable. Set $I_i \leftarrow P_i$ for all i.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable in the ordering. Set $D'_{cur} \leftarrow D_{cur}$.
2. Select a value $x \in D'_{cur}$ that is consistent with all previous variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur}.
   (c) For every constraint involving X_{cur} and no uninstantiated variables, test whether it is violated by $X_{cur} = x$. If a constraint is violated, go to (a).
   (d) Instantiate $X_i \leftarrow x$, and go to 1.
3. (Backjump.) If $I_{cur} = \emptyset$ (there is no previous variable in the induced parent set), exit with "inconsistent." Otherwise, set $I_{temp} \leftarrow I_{cur}$; set cur equal to the index of the last variable in I_{cur}. Set $I_{cur} \leftarrow I_{temp} \cup I_{cur} - \{X_{cur}\}$. Go to 2.

Figure 2.7: The Graph-based backjumping algorithm.

2.6 Graph-based backjumping

In Gaschnig's backjumping a jump occurs only after a dead-end which is a leaf in the search tree. A dead-end is also possible on an interior node. If all of the children of an interior node in the search tree lead to dead-ends (as happens with $X3$ in the example), then that node is called an interior dead-end. When an interior dead-end occurs, at least one value of the variable is compatible with the previous variables, but these compatible values did not lead to solutions.

Dechter's *graph-based backjumping* [15] is another variation on backtracking that can jump over variables in response to a dead-end. Unlike Gaschnig's backjumping, graph-based backjumping can jump back in response to an interior dead-end. To do so, it consults the *parent set* P_i of the dead-end variable X_i, where a parent of X_i is a variable that is connected to X_i in the constraint graph and precedes X_i in the variable ordering. After a dead-end at variable X_i, the algorithm jumps back based on information in the parent set. If X_i is a leaf dead-end, then the jump back is to the latest variable in the parent set. If X_i is an interior dead-end, then a new set is constructed by forming the union of X_i's parent set and those of all dead-end variables which have been found in the search tree below X_i. The algorithm jumps back to the latest variable in this induced parent set, which is called I_i in Fig. 2.7. To see why consulting I_i is necessary, consider a leaf dead-end at X_{15}, followed by a backjump to X_{10}, which is itself an interior dead-end. Suppose the parent set of X_{15} is $\{X_8, X_{10}\}$ and the parent set of X_{10} is $\{X_3, X_5\}$. If no dead-ends other than X_{15} occurred in the subtree below X_{10}, then $I_{10} = \{X_3, X_5, X_8\}$. It is necessary to jump back to X_8, because changing the value of that variable may permit X_{15} to be successfully instantiated and a solution to be found. Another way to look at is that some value of X_{10}, say x', was compatible with X_1 through X_9, but led to a dead-end at X_{15}. The dead-end at X_{15} requires the parents of X_{10} to be augmented with the parents of X_{15}, since those variables are necessary to explain why $X_{10}=x'$ did not lead to a solution.

In comparison with Gaschnig's backjumping, graph-based backjumping has the advantage of not having to update J_i after each unsuccessful consistency check. An offsetting overhead is that the parent sets have to be computed. In the pseudocode of Fig. 2.7, the parent sets are precomputed once and stored in a table denoted P_i; this step requires $O(n^2)$ space and $O(ce)$ time, where c is the number of constraints and e is the maximum number of variables in any constraint. Another disadvantage of using parent sets based on the constraint graph instead of actual conflicts discovered in consistency checking is that less refined information about the potential cause of the dead-end is utilized. A variable may be connected to the dead-end variable, but not, as currently instantiated, be a direct cause of the dead-end. In such a circumstance graph-based backjumping will not jump back as far as possible, and some avoidable thrashing will result.

Conflict-directed Backjumping

1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable in the ordering. Set $D'_{cur} \leftarrow D_{cur}$. Set $P_{cur} \leftarrow \emptyset$.
2. Select a value $x \in D'_{cur}$ that is consistent with all previous variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur}.
   (c) For $1 \leq i < cur$ (in ascending order): if $\vec{x}_i$ and $X_{cur}{=}x$ are inconsistent then add X_i to P_{cur} and go to (a).
   (d) Instantiate $X_{cur} \leftarrow x$ and go to 1.
3. (Backjump.) If $P_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent." Otherwise, set $P \leftarrow P_{cur}$; set cur equal to the index of the last variable in P. Set $P_{cur} \leftarrow P_{cur} \cup P - \{X_{cur}\}$. Go to 2.

Figure 2.8: The conflict-directed backjumping algorithm.

In the problem in Fig. 2.2, $P_7 = \{X1, X3, X4, X5\}$, so after a dead-end at $X7$ the next variable to be processed is $X5$, the last variable in P_7. $X5$ has no remaining values, so the algorithm will jump back to the last variable in $P_7 \cup P_5 - \{X5\}$, namely $X4$. Note that graph-based backjumping will not go back to $X2$, since it is not a parent of $X7$ nor of any of $X7$'s parents.

2.7 Conflict-directed backjumping

Prosser's conflict-directed backjumping algorithm [68] integrates the two ideas of jumping back to a variable that as instantiated conflicts with the dead-end variables, and of jumping back at interior dead-ends. When a subset of the variables are instantiated, they and any constraints which are now irrelevant can be temporarily removed from the constraint graph, creating a new *conditional* graph. A constraint involving X_i and a later variable X_j becomes irrelevant in two situations: the instantiated value of X_i is compatible with all values of X_j, or any value of X_j that conflicts with X_i as instantiated is also in conflict with a variable that is earlier than X_i. In the second case the constraint is considered irrelevant because even after changing the value of X_i it remains necessary to return to the earlier variable. In *conditional graph-based backjumping*, the parent set P_i^c is now recomputed each time X_i is instantiated, and only consists of previous variables which as instantiated conflict with some value of X_i. Conditional graph-based backjumping can substantially reduce the search space compared to regular graph-based backjumping, since P^c is often significantly smaller than P.

Conditional graph-based backjumping is essentially identical to Prosser's *conflict-directed backjumping* [68], described in Fig. 2.8. Because this is the standard version of backjumping now in use, we refer to it, here and later in the thesis, simply as backjumping. Steps 1 and 2 of this algorithm follow Gaschnig's backjumping closely, the most important change being in step 2 (c). In Gaschnig's backjumping it is sufficient to update a pointer to the deepest variable in conflict with some value of X_{cur}. Backjumping, like graph-based backjumping, remembers the set of variables which caused the dead-end. If a dead-end at X_i causes a jump back to X_h, and X_h turns out to be an interior dead-end, then the parents of X_i become the parents of X_h.

Backjumping will examine the fewest nodes of any algorithm presented so far, when presented with the problem of Fig. 2.2. After the dead-end at $X7$, it jumps to $X3$, since $(X1 = red, X3 = blue)$ conflicts with all the values of $X7$. $X3$ is a pseudo dead-end, so the algorithm will then jump again, back to $X1$.

A variation of backtracking called dependency-directed backtracking was proposed by Stallman and Sussman in the context of circuit analysis [84]. The basic idea is to record, as additional entries in the database of constraints, the cause of a dead-end, thus permitting the algorithm to backtrack directly to a variable which part of the cause. The concept is identical to that of backjumping. The term intelligent backtracking is also used for similar concepts in logic programming and truth maintenance systems.

2.8 Forward checking

Haralick and Elliott's *forward checking* algorithm [40] is a widely used variation of backtracking. In contrast to the backtracking and backjumping algorithms presented above, which remove values from the domain of the current variable by checking them against previously instantiated variables, forward checking instantiates a variable, and then removes conflicting values from the domains of all future (uninstantiated) variables. Forward checking rejects any value which would remove the last remaining value from the domain of a future variable. Of course, the values are not removed permanently. As with backtracking and backjumping, D' sets are employed to contain reduced domains. Temporary removal of values from D' sets is known as filtering. Algorithms which filter the domains of uninstantiated variables are often called look-ahead algorithms.

The forward checking algorithm, as described in Fig. 2.9, differs from backtracking in several ways. Step 0 sets up the D' sets to hold the complete domain for each variable. In step 1, D'_{cur} is not modified, since as the algorithm proceeds to a new variable it needs to retain the knowledge acquired in previous steps of which values in the domain of the new variable are inconsistent with the current partial instantiation. In step 2, note that forward checking is looking for a value in D'_{cur} that is compatible with at least one value for each *future* variable. There is no need to compare the elements of D'_{cur} with the previous variables, since only those value in D_{cur} that are compatible with $\vec{x}_{cur-1}$ remain in D'_{cur}.

Step 3 of forward checking specifies that after backtracking to an earlier variable, the D' sets should be restored to the values they had before the new X_{cur} was assigned its current value. To illustrate, suppose X_9 is assigned the value x_1

Forward Checking

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable in the ordering.
2. Select a value $x \in D'_{cur}$ that is consistent with at least one remaining value of each future variable. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur}.
   (c) Examine the future variables $X_i, cur < i \leq n$. Remove all v in D'_i that conflict with $\vec{x}_{cur-1}$ and $X_{cur} = x$. If doing so results in $D'_i = \emptyset$ for some i, then restore the D'_i's to their values before this step (c) was begun and go to (a).
   (d) Instantiate $X_{cur} \leftarrow x$ and go to 1.
3. (Backtrack step.) If there is no previous variable, exit with "inconsistent." Otherwise, set cur equal to the index of the previous variable. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 2.9: The forward checking algorithm.

and the sets D'_{10} through D'_n are filtered appropriately. If the algorithm backtracks to X_9, it will be assigned a new value, say x_2, and the filtering due to x_1 is no longer relevant and should be undone.

The authors of the forward checking algorithm described the motivation behind its development as "Lookahead and anticipate the future in order to succeed in the present" [40]. Forward Checking tends to make dead-ends occur much earlier in the search, as may be seen by comparing the behavior of this algorithm and the "look back" algorithms such as backjumping presented earlier. In the problem described in Fig. 2.2, instantiating $X1 = red$ reduces the domains of $X3, X4$ and $X7$. Instantiating $X2 = blue$ does not affect any future domain. The variable $X3$ only has $blue$ in its domain, and selecting that value causes the domain of $X7$ to be empty, so $X3 = blue$ is rejected and $X3$ is a dead-end. See Fig. 2.10.

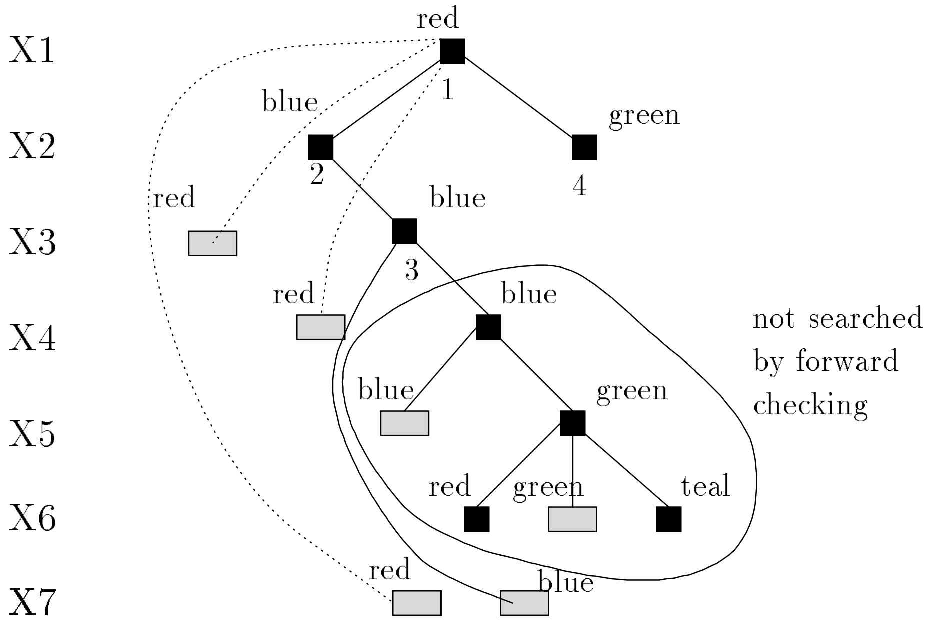


Figure 2.10: Part of the search space explored by forward checking, on the example CSP in Fig. 2.2. Only the search space below X1=red and X2=blue is drawn. Dotted lines connect values with future values that they filter out in step 2 (c).

Génisson and Jégou [33] have shown that the well-known Davis-Putnam (DP) procedure for propositional satisfiability [14, 13], in particular the version from 1962, interleaves search and domain filtering in a manner that is strictly equivalent to forward checking. Both the "Rule for Elimination of One-Literal Clauses" and the "Affirmative-Negative Rule" from the DP procedure can eliminate or simplify clauses, reducing redundancy. In CSP terms, these rules correspond to removing elements from the D' sets, as in step 2 (c) of forward checking, and replacing longer constraints with shorter one.

2.9 Arc-consistency

An important class of algorithms for processing CSPs are those that enforce arc-consistency. Arc-consistency algorithms, as well as procedures for any level of k-consistency, can be executed before or during search.

An essential component of any arc-consistency procedure is the REVISE subroutine [50] (see Fig. 2.11). This subroutine examines two variables, X_i and X_j, and determines whether each value y in D'_i has at least one compatible value in the domain of X_j. If no such compatible value exists, then $X_i{=}y$ cannot be part of a solution, so y is removed from D'_i. Our version of REVISE differs from the standard presentation in that it includes the current partial instantiation $\vec{x}_{cur}$ in the test for consistency on line 2. This is important if the CSP includes non-binary constraints.

```
REVISE(i, j)
1  for each value y ∈ D'_i
2      if there is no value z ∈ D'_j such that (x⃗_cur, X_i=y, X_j=z) is consistent
3      then remove y from D'_i
```

Figure 2.11: The Revise procedure.

There is a long history of arc-consistency algorithms. The simplest is due to Mackworth [50] and is called AC-1 (see Fig. 2.12). To make AC-1 more appropriate for interleaving with backtracking, we have added a tree-depth parameter d; therefore only future variables are checked for arc-consistency. Another change in our version is the that procedure returns immediately if an empty future domain is detected. AC-1 calls REVISE on each pair of future variables. If REVISE removes a value, then there may be repercussions on the domains of other variables, so again REVISE is called on each pair of future variables. The process continues until no values are removed – the CSP is arc-consistent.

AC-1 suffers from the defect that if a single value is removed from a variable's domain by REVISE, then an entire repetition of the loop in lines 2–6 of Fig. 2.12 must be made, and much needless consistency checking will be done. AC-1 requires $O(d^3ne)$ time in the worst case, where d is the size of the largest domain, n is the number of variables, and e is the number of constraints, and $O(e + nd)$ space.

Improved arc-consistency algorithms have been given the monikers AC-2 [50] (superseded by AC-3) and AC-3 [50], which has $O(d^3e)$ time complexity and $O(e+$

```
AC-1(d)
repeat
    for i ← d + 1 to n
        for j ← d + 1 to n
            REVISE(i, j)
            if D'_i = ∅
            then return
until no call to REVISE removed a value
```

Figure 2.12: The arc-consistency algorithm AC-1.

nd) space complexity. AC-4 [59] has an optimal time complexity of $O(ed^2)$, but suffers from several drawbacks. Its average time complexity is very near the worst-case, and the algorithm has a relatively high start-up cost in initializing several data structures. Moreover, the space complexity of AC-4 is $O(ed^2)$. AC-5 [88] is a general framework for arc-consistency algorithms, but is not itself a specific algorithm. AC-6 [6] keeps the optimal $O(ed^2)$ worst-case time complexity of AC-4, but has a superior space complexity of $O(ed)$. The improvements of the more advanced arc-consistency algorithms come from maintaining data structures which record which variables are possibly impacted by the removal of a value from the domain of another variable. How best to cache this information and whether doing so pays off in reduced processing time depends largely on the domain size and number of constraints in the constraint problem (see [6] for experimental comparisons of AC-3, AC-4 and AC-6).

As with AC-1, the heart of AC-3 is the REVISE procedure, which enforces arc-consistency on a single directed constraint arc (see Fig. 2.13). AC-3 views each (binary) constraint as two directed arcs, and maintains a set, Q, of every directed arc which needs to be tested for arc-consistency. If the arc $X_i \rightarrow X_j$ is in Q, it means the AC-3 must use REVISE to verify that every value in the domain of X_i is compatible with at least one value in the domain of X_j. If REVISE removes a value from the domain of future variable X_i, every future variable which shares a constraint with X_i is identified, and the arcs from those variables to X_i are added to Q. This is necessary because the value just removed from X_i may be the only one compatible with, say, $X_k = x_k$. The AC-3 algorithm continues until all arcs have been removed from Q without any new ones being added.

AC-3(d)
$Q \leftarrow \{\text{arc}(i,j) | i > d, j > d\}$
repeat
    select and delete any arc(p, q) in Q
    REVISE(p, q)
    if $D'_p = \emptyset$
    then return
    if REVISE removed a value from D'_p
    then $Q \leftarrow Q \cup \{\text{arc}(i,p) | i > d\}$
until $Q = \emptyset$

Figure 2.13: Algorithm AC-3.

2.10 Combining Search and Arc-consistency

Early and influential algorithms that combine search and consistency enforcing by performing an arc-consistency procedure after each variable instantiation are due to Waltz [90] and Gaschnig [31]. More recent algorithms of this type were developed by Nadel [61] and Sabin and Freuder [74].

Waltz's system takes as input as set of line segments, and constructs a description of a scene of three-dimensional objects which plausibly could give rise to a line drawing with the given line segments. Line segments which meet define junctions, which may be corners, edges, or the result of shadows. Waltz defines 11 different junction types. Each line segment or edge in a junction can be labeled as a shadow edge, a concave edge, a convex edge, an obscuring edge, or a crack edge. When considering a junction in isolation there are usually many possible labelings for the junction's line segments, but a line segment must be labeled the same way in each junction it is part of. Waltz found that by comparing adjacent

Waltz's algorithm (close to original)

1. (Before search.)
   (a) Use domain-specific knowledge to reduce the number of possible edge-labelings at each junction.
   (b) (Enforce arc-consistency.) Repeat until no edge labels are removed: Eliminate a possible label for an edge when the edge cannot be assigned the same label at another junction.
2. Choose an unlabeled edge J to label.
3. Eliminate from J's set of possible labels all those which don't have matches in J's neighbors (those junctions which share a line segment with J).
4. Choose a label for J.
5. Check each of J's neighbors, eliminating from their sets of possible labels those that don't match how J's line segments have been labeled. If a neighbor has a label removed from its set of possible labels, repeat the arc-consistency step 1 (b).
6. (This step is unstated but implied.) Backtrack if a junction's set of possible labels becomes empty.

Figure 2.14: Waltz's algorithm.

junctions and labeling the line segments so that no local contradictions were made, his system usually found only a few lines which were not uniquely specified, and thus very little search was required.

Although Waltz does not explicitly state the algorithm used, we have reconstructed it in Fig. 2.14 In Fig. 2.15, we cast Waltz's algorithm in the common framework used for other algorithms in the chapter.

The primary differences between Waltz's algorithm and Haralick and Elliott's forward checking is that Waltz's algorithm performs a step before search which enforces arc-consistency, and then enforces arc-consistency after each instantiation (labeling). This additional work before and during search was effective in the application that Waltz studied, because the domains are relatively large (typically there are five to ten ways to label a junction after the domain specific selection

Waltz's algorithm (reconstruction)

0. (Initialize.) For all variables X_i, set $D'_i \leftarrow D_i$. Enforce arc-consistency, which may remove some values from some D' sets.
1. If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set *cur* equal to the index of the next variable in the ordering.
2. Select a value $x \in D'_{cur}$, as follows:
   - (a) If $D'_{cur} = \emptyset$, go to 3.
   - (b) Pop x from D'_{cur}.
   - (c) Examine the future variables $X_i, cur < i \leq n$. Remove all v in D'_i which conflict with $\vec{x}_{cur-1}$ and $X_{cur} = x$. If doing so results in $D'_i = \emptyset$ for some i, then restore the D'_i's to their values before this step (c) was begun and go to (a).
   - (d) Instantiate $X_{cur} \leftarrow x$.
   - (e) Enforce arc-consistency. If doing so leaves a D' set without any elements, go to (a). Otherwise, go to 1.
3. (Backtrack step.) If there is no previous variable, exit with "inconsistent." Otherwise, set *cur* equal to the index of the previous variable. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 2.15: A reconstructed version of Waltz's algorithm.

rules are applied) and the constraints are equality constraints, which are quite tight. In contrast, Haralick and Elliot performed most of their experimental work on problems with much looser constraints.

Nadel [62] proposes a series of partial arc-consistency algorithms, called $\mathrm{AC}^1/_5$, $\mathrm{AC}^1/_4$, $\mathrm{AC}^1/_3$ and $\mathrm{AC}^1/_2$ that are designed to be interleaved with a backtracking based tree-search algorithm. The fractional suffixes indicate approximately how much work the algorithm does in proportion to full arc-consistency. Forward checking, which corresponds to backtracking plus $\mathrm{AC}^1/_4$, is judged the best in the experiments Nadel reports.

2.11 Full and Partial Looking Ahead

In addition to forward checking, Haralick and Elliott [40] defined two algorithms called partial looking ahead and full looking ahead which do more consistency enforcing than forward checking and less than arc-consistency.

The additional processing done by full looking ahead is a limited form of arc-consistency, in effect performing a single iteration of the AC-1's outer loop (see Fig. 2.16). The full looking ahead subroutine can be called in step 2 (c) of forward checking (Fig. 2.9), by modifying that step in the following manner:

2. (Choose a value.)

   (c) (Forward checking style look-ahead.) Examine the future variables $X_i, cur < i \leq n$, For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i; if D'_i is now empty, go to (d) (without examining other X_i's).
   (Additional looking ahead.)

      i. If *Algorithm* = FLA, perform FULL-LOOKING-AHEAD(i);
      ii. Else if *Algorithm* = PLA, perform PARTIAL-LOOKING-AHEAD(i).

To illustrate how full looking ahead works, suppose there are three future variables, X_{25} with current domain $\{a, b\}$, X_{26} with current domain $\{a, b\}$ and X_{27} with current domain $\{b\}$. Also suppose there is an inequality constraint (as in graph coloring) between X_{25} and X_{26} and between X_{26} and X_{27}. Full looking ahead will process X_{25} and reject neither of its values, since they both have a compatible value in the domain of X_{26}. When full looking ahead processes X_{26}, it removes the value b because there is no allowable match for b in the domain of X_{27}. Arc-consistency would later go back and remove a from X_{25}'s domain, because it no longer has a consistent match in X_{26}'s domain, but full looking ahead does not do this.

Partial looking ahead [40] seems to have been proposed with the same general motivation that inspired the new heuristics presented later in this chapter: a notion

```
FULL-LOOKING-AHEAD(d)
for i ← d + 1 to n
    for j ← d + 1 to n
        REVISE(i, j)
        if D'_i = ∅
        then return
```

Figure 2.16: The full looking ahead subroutine.

that it might be possible to do a bit less work than that done by full looking ahead, and yet to achieve most of the benefits. This scheme employed by partial looking ahead is to check each future variable only with those other future variables later than it (see Fig. 2.17). Dechter and Pearl [18] note that partial looking ahead performs directional arc-consistency at each step, and this observation may explain the motivation for the original development of the algorithm. However, when using dynamic variable ordering the order of uninstantiated variables is not known, and so the set of variables which are "after" a future variable is essentially a random one. Of course it is possible to define an ordering for the uninstantiated variables, but this ordering is unlikely to be the order in which variables are actually instantiated.

2.12 Variable Ordering Heuristics

In describing the algorithms in the previous sections, we have assumed that the order of the variables is static, that is, unchanging as the algorithm proceeds. In

```
PARTIAL-LOOKING-AHEAD(d)
for i ← d + 1 to n
    for j ← i + 1 to n
        REVISE(i, j)
        if D'_i = ∅
        then return
```

Figure 2.17: The partial looking ahead subroutine.

practice this is not necessarily the case, which requires modifying some algorithms. In this section we first consider several heuristics for static variable ordering, and then the dynamic variable ordering scheme most frequently used.

Waltz [90] proposed a variable ordering heuristic, motivated by the desire to "eliminate as many possibilities as early as possible." First, label junctions on the scene/background border, since those junctions tend to have few possible labels. Second, label junctions which share an edge with those junctions in the first step. In general constraint satisfaction terms, we can interpret these guidelines as "select a variable which has few values" (e.g. $|D'|$ is small). Intuitively, if we want the size of the search tree to be as small as possible, it is probably better to put nodes with small branching factor first.

In some application areas and in many instances of artificial data, all variables have the same size domain. In such cases, static variable ordering schemes have to rely on the constraint graph. We now present two which do so.

The *minimum width* (MW) heuristic [25] orders the variables from last to first by selecting, at each stage, a variable in the constraint graph that connects to the minimal number of variables that have not been selected yet. For instance, in the CSP from Fig. 2.2, we could choose $X2$, $X3$ or $X6$ to be the last variable, since each is connected to two other variables. If we arbitrarily select $X2$ to be last, then $X6$ will be chosen to be second-to-last, since it now participates in one constraint ($X2$ and constraints involving it having been eliminated), the minimum. Now both $X3$ and $X5$ connect to two other nodes; we can choose $X5$ to be third-to-last. Continuing in this manner, the final ordering might be $X7, X3, X1, X4, X5, X6, X2$. There is usually more than one min width ordering of a CSP.

The *maximum cardinality* (MC) variable ordering heuristic [17] selects the first variable arbitrarily, and then selects each subsequent variable by choosing the one connected with the largest number of previously chosen variables. A variation of MC is to choose as the first variable the one that participates in the most constraints. Using this variation, a maximum cardinality ordering of the variables from Fig. 2.2 is $X1, X2, X3, X7, X4, X5, X6$.

The use of a variable ordering heuristic does not change the worst-case complexity of any backtracking algorithm. The most extensive set of experiments comparing the average case performance of MW and MC is reported in [17]. No clear superiority between the two was discernable (several other ordering schemes were studied as well), although MW was slightly superior.

Under a dynamic variable ordering scheme, the order of the variables is determined as search progresses, and may be differ from one branch of the search tree to another. Most dynamic variable ordering heuristics are based on an idea proposed by Haralick and Elliot [40] under the rubric "fail first." The main idea of the heuristic is to select the future variable with the smallest remaining domain, or D' set. Haralick and Elliot show via a probabilistic analysis that choosing the variable with the smallest number of remaining values minimizes the probability that the variable can be consistently instantiated, and thus "minimizes the expected length or depth of any branch" (p. 308). The fail first heuristic relies on a look-ahead technique such as forward checking to filter the domains of the future variables.

Several other dynamic variable ordering heuristics have been proposed. Those of Purdom [71] and Minton et al. [56] do not rely on a forward checking style look ahead, but most others do and can be considered variants of the Haralick and Elliott's fail first principle: Nudel [65], Gent et al. [34].

Chapter 3

The Probability Distribution of CSP Computational Effort

3.1 Overview of the Chapter

In this chapter we present empirical evidence that the distribution of effort required to solve CSPs can be approximated by two standard families of continuous probability distribution functions[1]. Solvable problems can be modelled by the Weibull distribution, and unsolvable problems by the lognormal distribution. These distributions fit equally well over a variety of backtracking based algorithms. We show that the mathematical derivation of the lognormal and Weibull distribution functions parallels several aspects of CSP search.

3.2 Introduction

This chapter reports our efforts to uncover regularities that exist in the distribution of effort required to solve CSPs, independent of the algorithm used (within the framework of backtracking based algorithms) or the specific parameters to the problem generating model. In all cases we consider satisfiable and unsatisfiable

[1]The general results and conclusions of this chapter were first reported in Frost *et al.* [29] and Rish and Frost [73]

problems separately, since we have found them to have completely different distributions. For satisfiable problems, we limit our attention to the effort required to find the first solution.

The problem we address has long been an open one. Many researchers have observed that the work required to solve constraint satisfaction problems exhibits a large variance. Knuth noted in 1975 [46] that "great discrepancies in execution time are characteristic of backtrack programs." Mitchell [57] writes "we can't even estimate how large a sample size would be needed to obtain valid confidence intervals for the sample mean." Kwan [49] shows that the distribution of effort on CSPs is not normal. Researchers have addressed the problem of unknown distribution by reporting a variety of statistics, including the mean, the median, the standard deviation, the minimum, the maximum, the 99th percentile, and the 99.9th percentile.

To illustrate the issues that may arise in reporting experimental results, consider the data in Table 3.2, which come from an experiment using the BJ+DVO algorithm (described in Chapter 4). The algorithm was applied to 10,000 random instances created by a Model A generator and parameters $\langle 100, 8, .0566, 0.50\rangle$. Each line of the table reports various statistics – average, median, standard deviation, maximum – after a certain number of instances had been processed. The table is organized in this manner to show how the average hardness of a sample of problems, measured by CPU time, can change as the size of the sample grows, mostly due to the effect of rare very hard problems. Mitchell [57] reports a similar table, based on an experiment with the Davis-Putnam procedure and random 3-CNF formulas, in which at sample size 100 the mean is 98, and after 5,000 instances the sample mean has become 12,000.

This pattern or distribution of problem hardness presents two sources of difficulty for someone designing and reporting an experiment with a CSP algorithm and randomly generated problems. The design question is, how many instances are enough? The goal is to have the sample be representative of the entire set of

sample size	average	median	hardest	st. deviation
50	12.08	3.18	60.60	32.44
100	13.25	3.30	235.00	33.53
200	17.33	3.62	290.83	45.37
300	19.60	3.58	1,019.22	72.82
400	19.28	3.42	1,019.22	69.16
500	18.65	3.37	1,019.22	65.36
800	20.31	3.42	1,176.63	71.85
1,000	18.96	3.32	1,176.63	67.59
1,500	18.00	3.16	1,176.63	69.65
2,000	20.20	3.08	3,083.43	101.49
5,000	19.26	3.13	3,083.43	88.93
10,000	19.05	3.15	3,083.43	76.44

Table 3.1: Cumulative statistics at varying points in a 10,000 sample experiment. Units are CPU seconds, algorithm is BJ+DVO, generator parameters are $\langle 100, 8, .0566, 0.50 \rangle$.

possible random problems with a certain set of parameters, requiring hundreds if not thousands of instances. The example of Table 3.2 shows that with a sample of a few hundred instances, the empirical mean can be about 10% too large or too smal, if we make the assumption, perhaps unwarranted, that the sample mean after 10,000 instances is close to the true population mean.

The second, related, question is which statistics to report. With more than a few dozen instances it is not feasible to report the results of each instance. The mean and the median are the most popular statistics to report, but these do not convey the long "tail" of difficult problems that often occurs. In order to convey more information, some authors have reported percentile points such as the hardness of the instance at the 99th and 99.9th percentiles, minimum and maximum values, and the standard deviation. In experiments involving large sets of randomly generated instances, the ideal would be to report the entire distribution of cost to solve. It might be imagined that doing so would be unwieldy. In this chapter we present evidence that such an approach is quite feasible, because the experimentally derived distributions are quite closely approximated by standard distribution functions from the field of statistics.

Our main findings are that the hardness distributions of solvable and unsolvable problems are distinctly different, with unsolvable problems being close to the lognormal distribution, and solvable problems being reasonably well approximated by the Weibull distribution. The lognormal and Weibull distributions are each parameterized by shape and scale parameters. By noting that the results of an experiment can be fit by a certain distribution with parameters x and y, it is possible to convey a complete understanding of the experimental results: the mean, median, mode, variance, and shape of the tail.

We use the number of nodes in the search space to measure the computational effort required to solve a problem instance. In [29] we measured consistency checks, with almost identical results. A limitation of counting the number of nodes in the search space as a proxy for overall effort of the algorithm is that this measure does not take into account the amount of work, and ultimately CPU time, expended by the algorithm at each node. This quantity can vary greatly; a theme of research in CSP algorithms that we will explore in later chapters is how to best trade off extra work at some nodes for a sufficiently reduced size in the total search space. Thus counting only the number of nodes is often not a fair comparison between algorithms; this should be borne in mind when reading the tables which contain results for multiple algorithms.

3.3 Random Problem Generators

3.3.1 Model A and Model B

In this chapter we use two random problem generators; following Smith and Dyer [82] we call them Model A and Model B. In other chapters we use the Model B generator only. In this chapter we focus on Model A because it has some properties of independence between constraints which simplifies the analysis at the end of the chapter.

Both generators use parameters N, D, T, and C. N and D have the same meaning in both models. As defined in Chapter 1, in Model B parameters C and T define the exactly number of constraints and prohibited value pairs per constraint. In Model A, C is the probability of a constraint existing between any pair of variables, and each constraint is statistically independent of the others. Parameter T in Model A defines the probability that a value in the domain of one variable in a constraint will be incompatible with a value in the domain of the other variable in the constraint. Two different problem instances under the Model A distribution can have different numbers of constraints, and two constraints in the model A distribution can prohibit a different number of value pairs.

3.3.2 Parameters at the Cross-over Point

The phase transition phenomenon, described in Chapter 1, is an important aspect of empirical CSP research. In this and succeeding chapters, many experiments are conducted with parameters at the cross-over point. Therefore, we briefly discuss the manner in which we determined combinations of parameters values at the cross-over point.

Two interesting aspects of the phase transition phenomenon are that the peak in average difficulty occurs at the 50% satisfiable point, and that the relationship between the number of constraints and the number of variables at the cross-over point is linear. Neither relationship has been proven analytically, but both have been well established for K-SAT and for binary CSPs [12, 27, 35]. An accurate formula for deriving sets of parameters at the cross-over point has not been found, but the following equation, due to Smith [81] and Williams and Hogg [91], is an approximation:

$$D^N(1-T)^{C'} = 1, \tag{3.1}$$

D^N is the total number of possible assignments of values to variables, and the probability of any given random assignment satisfying all C' constraints is $(1 -$

D	T	formula for C'
3	.111	$7.300N + 16.72$
3	.222	$3.041N + 13.72$
3	.333	$1.516N + 15.56$
3	.444	$0.725N + 16.78$
6	.111	$13.939N + 12.82$
6	.222	$6.361N + 6.56$
6	.333	$3.761N + 5.49$
6	.444	$2.408N + 5.41$
6	.556	$1.510N + 7.00$
9	.222	$8.284N + 3.86$
9	.333	$4.949N + 4.87$
9	.444	$3.280N + 4.32$

Table 3.2: Experimentally derived formulas for C' at the cross-over point, under Model B. $C' = CN(N-1)/2$ is the number of constraints in a CSP with parameters C and N.

$T)^{C'}$. The product is the expected number of solutions. When the expected number of solutions is about 1, then the parameters are at the cross-over point. The equation is not accurate because it makes an unwarranted assumption of independence between the constraints, and because it ignores the distribution of solutions. Solvable problems often have many solutions, so looking for the expected number of solutions to be 1 is an approximation. Nevertheless, the formula does describe the general relationship between the parameters at the cross-over point.

Equation (3.1) can be rewritten as

$$C' = N(-\log_{1-T} D), \tag{3.2}$$

which shows that for fixed settings of T and D the relationship between C' and N is linear. Similarly, (3.1) shows that for fixed N and D, increasing T means decreasing C' to stay at the cross-over point; or for fixed N and T, D must increase as C' decreases.

Because equation (3.1) is not accurate, the exact values of parameters at the cross-over point have to be determined experimentally. Empirically derived relationships between C' (number of constraints) and N (number of variables) are given in Fig. 3.2, for selected values of D and T. These formulas were determined through an iterative process. For a given set of parameters N, D, and T, 100 random problems were generated at each of several values of C. We used values of C corresponding to an integer number of constraints. A standard CSP algorithm was applied to each problem, and we noted the percentage that had solutions. From that information we could estimate values of C reasonably near the cross-over point. The process was repeated until the number of constraints at the cross-over value was ascertained within one or two. We then ran experiments with 2,000 instances at each of two or three values of C. The final result was usually one value of C that produced slightly more than 50% solvable, and one value of C that produced slightly fewer. We determined the cross-over value of C by linear interpolation. This is illustrated by the following example, for parameters N=25, D=3, T=.111:

C'	C	solvable (out of 2,000)
198	0.6600	1058
199	0.6633	988

Interpolating linearly, 1,000 is .83 of the way from 1,058 to 988, so we estimate the cross-over value of C' to be 198.83 in this example. When the cross-over values of C' were determined for a fixed D and T and five to ten values of N, we derived the parameters of the linear regression line using the Unix "pair" program [35].

3.4 Statistical Background

In this section we review some basic notions of statistics upon which the later sections are based. We focus on three probability distributions, the normal, the lognormal, and the Weibull.

3.4.1 Distribution functions

Given a random variable X, the probability distribution of X defines the probability that X will take on any particular value in its domain. If X is a continuous variable, then the probability distribution is a continuous probability distribution. Although our study is of discrete constraint satisfaction problems, we emphasize the continuous probability distributions to which they converge in the limit of increased sample size.

The probability distribution of a continuous random variable X can be defined in two ways. The cumulative distribution function (cdf) $F(x)$ specifies the probability that the value of X is less than or equal to x:

$$F(x) = P(X \leq x) \tag{3.3}$$

The probability density function (pdf) $f(x)$ is the derivative of the cumulative probability function. Thus, for a continuous variable,

$$F(x) = \int_{-\infty}^{x} f(t)dt \tag{3.4}$$

and in particular

$$F(b) - F(a) = \int_{a}^{b} f(t)\,dt = P(a < X \leq b). \tag{3.5}$$

For a continuous random variable X, the expected value $E(X)$, also called the mean, μ, is given by

$$E(X) = \int_{-\infty}^{\infty} xf(x)\,dx, \tag{3.6}$$

assuming the integral is absolutely convergent ($|x|f(x)dx < \infty$). The deviation from the mean of a particular value of X is $(X - \mu)$. The expected value of the square of the deviation is called the variance of X, and is written $\mathrm{Var}(X)$ or σ^2. It is computed as

$$\mathrm{Var}(X) = E[(X - \mu)^2] = E(X^2) - \mu^2. \tag{3.7}$$

The positive square root of the variance, σ, is called the standard deviation.

3.4.2 Sample Statistics and Empirical Distributions

The arithmetic sample mean $\bar{x}$ of n observations $(x_1, x_2, \ldots, x_n)$ is the sum of all observations divided by the number of observations, or

$$\bar{x} = \frac{1}{n} \sum_i x_i. \tag{3.8}$$

The sample variance s^2 is computed as

$$s^2 = \frac{1}{n-1} \sum_i (x_i - \bar{x})^2 \tag{3.9}$$

and the sample standard deviation s is the positive square root of s^2.

The empirical distribution function is important for goodness-of-fit tests we discuss in a later section. Let $x_1, x_2, \ldots, x_n$ denote an ordered random sample of size n. The empirical distribution function, or sample CDF, is calculated as

$$F_{emp}(x_i) = i/n. \tag{3.10}$$

Thus F_{emp} ranges from $1/n$ to 1.

3.4.3 The normal distribution

Perhaps the most widely used distribution in statistics is the normal distribution. The probability density function of the normal distribution is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right). \tag{3.11}$$

The pdf $f(x)$ is parameterized by μ, the mean of the distribution which has the range $-\infty < \mu < \infty$, and σ, the standard deviation of the distribution, which ranges $0 < \sigma < \infty$. The normal distribution with parameters μ and σ is denoted $N(\mu, \sigma^2)$. The standardized normal distribution, $N(0,1)$, has a cumulative distribution function $\Phi(x)$ defined by

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{1}{2}t^2}\, dt \tag{3.12}$$

In general, if $z = (x-\mu)/\sigma$ then $F(x) = \Phi(z)$.

Much of the usefulness of the normal distribution stems from the fact that, in the limit, the sum of independent random variables tends to be normally distributed. This is known as the central limit theorem.

Theorem 1 (Central Limit Theorem) *Let $(X_1, X_2, \ldots)$ be a sequence of independent random variables with cdf's $(F_1(X), F_2(X), \ldots)$, means $(\mu_1, \mu_2, \ldots)$, and variances $(\sigma_1^2, \sigma_2^2, \ldots)$. Let $a_n = X_1 + \cdots + X_n$, $\zeta_n = \mu_1 + \cdots + \mu_n$, and $\tau_n^2 = \sigma_1^2 + \cdots + \sigma_n^2$. Then*

$$\lim_{n\to\infty} P\left(\frac{a_n - \zeta_n}{\tau_n} < y\right) = \Phi(y), \tag{3.13}$$

or in other words

$$\lim_{n\to\infty} d_n \textit{ is distributed as } N(\zeta_n, \tau_n^2) \tag{3.14}$$

subject to certain conditions on the variances σ_i^2.

3.4.4 The lognormal distribution

A positive random variable X is lognormally distributed with parameters μ and σ if $Y = \ln X$ is normally distributed with mean μ and variance σ^2. We say X is distributed as $\Lambda(\mu, \sigma^2)$. The probability density function of the lognormal is

$$f(x) = \begin{cases} \frac{1}{\sqrt{2\pi}\sigma x} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right), & x > 0 \\ 0, & x \leq 0. \end{cases} \tag{3.15}$$

The parameters μ and σ are not the mean and standard deviation of the lognormal distribution. These statistics, and the variance, median and mode, are given by

$$\begin{aligned} E(X) &= \exp(\mu + \sigma^2/2) & (3.16) \\ \mathrm{StdDev}(X) &= \exp(\mu)(\exp(2\sigma^2) - \exp(\sigma^2))^{1/2} & (3.17) \\ \mathrm{Var}(X) &= \exp(2\mu + \sigma^2)(\exp(\sigma^2) - 1) & (3.18) \\ \mathrm{Median}(X) &= \exp(\mu) & (3.19) \\ \mathrm{Mode}(X) &= \exp(\mu - \sigma^2) & (3.20) \end{aligned}$$

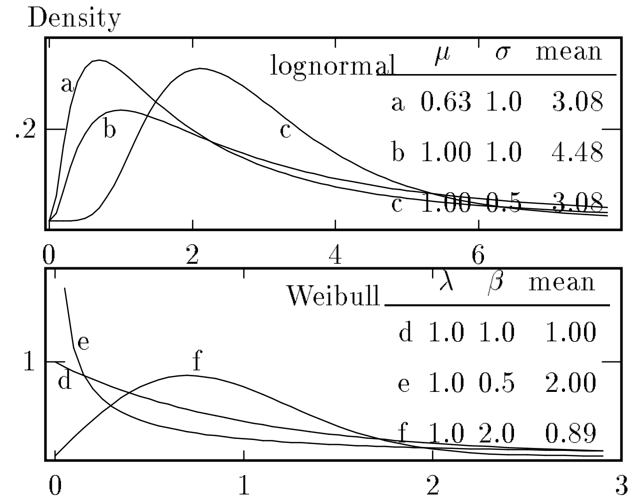


Figure 3.1: Graphs of the lognormal and Weibull density functions for selected parameter values.

The cumulative distribution function of the lognormal is

$$F(x) = \Phi\left(\frac{\ln x - \mu}{\sigma}\right). \tag{3.21}$$

The first graph in Fig. 3.1 shows the shape of the lognormal pdf for several values of μ and σ. When σ is small the pdf is relatively symmetric and the mean and median are close together. As σ increases, the lognormal distribution becomes more skewed, and the probability of a random instance being above the mean decreases.

3.4.5 The Weibull distribution

The Weibull distribution has wide applicability in reliability and lifetime studies. Its probability density function is

$$f(x) = \begin{cases} \lambda^\beta \beta x^{\beta-1} e^{-(\lambda x)^\beta}, & x > 0 \\ 0, & x \le 0 \end{cases} \tag{3.22}$$

and the cdf is

$$F(x) = \begin{cases} 1 - e^{-(\lambda x)^\beta}, & x > 0 \\ 0, & x \le 0. \end{cases} \tag{3.23}$$

The parameter λ is interpreted as scale and β as shape. The mean, E, of a Weibull distribution is given by $E = \lambda^{-1}\Gamma(1 + \beta^{-1})$ where $\Gamma(\cdot)$ is the Gamma function. There is also a three parameter version of the Weibull distribution, in which x is replaced by $x-\alpha$ in the above equations; α is called the *origin* of the distribution. When $\beta = 1$, the Weibull distribution is identical to the exponential distribution.

The second graph in Fig. 3.1 shows the shape of the Weibull pdf for several values of λ and β. When $\beta \leq 1$, the pdf has no mode (maximum value) and is monotonically decreasing. When β is greater than 1, the pdf has a maximum which is greater than 0; the greater is β, the further to the right on the graph the maximum appears.

3.4.6 Statistical Significance and Tail Ratio Test

It is important to be able to measure and report the goodness of fit between a hypothesized distribution function and a sample created experimentally. We measure the discrepancy in two ways, with a standard statistical technique called the Kolmogorov-Smirnov statistic, and with a measure we developed call the Tail Ratio.

The Kolmorgorov-Smirnov (KS) test statistic is based on the maximum difference between the cdf of the hypothesized distribution and the empirical cdf of the sample. Let $F(x)$ be the cdf of the hypothesized distribution, and define

$$\begin{aligned} D^{+} &= \max_i(i/n - F(x_i)) = \max_i(F_{emp}(x_i) - F(x_i)) \\ D^{-} &= \max_i(F(x_i) - (i-1)/n) = \max_i(F(x_i) - F_{emp}(x_{i-1})) \\ D_{max} &= \max(D^{+}, D^{-}). \end{aligned}$$

D^{+} is the maximum distance the empirical cdf goes above the hypothesized cdf, and D^{-} is the maximum distance below. Fig. 3.2 shows the cdf for the lognormal with $\mu = 12.00$ and $\sigma = 0.44$. Fig. 3.3 shows a small section of an empirical distribution based on a sample and which demonstrates D_{max} visually. D_{max} is the KS statistic;

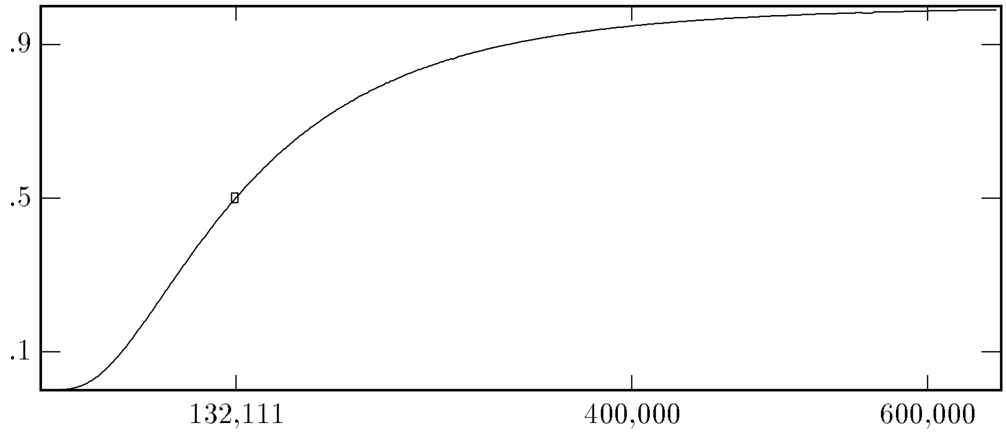


Figure 3.2: The cumulative distribution function for $\Lambda(12.00, 0.44)$ (see the fifth line in Fig. 3.6). The curve is truncated at $(F(646650) = 0.99$. The small square indicates the portion magnified in the next Figure. 132,111 is the median of the distribution.

we report it in tables under the heading "KS". This value can range from 0 to 1, with a smaller value indicating a better fit. In general terms, when the product of the KS statistic and the square root of the number of samples is less than 1, a close match between the distribution and the sample is indicated. To interpret the statistical significance of the KS statistic, it is necessary to know critical values that correspond to particular level of significance. If specific parameters of the distribution are part of the hypothesis that is being tested, then these critical values are readily available. In our case, however, the distribution parameters are estimated from the sample data, and there is no simple way to derive critical values. We report the KS statistic solely as an indicator of the relative goodness-of-fit of different samples to the corresponding lognormal or Weibull functions.

The KS statistic is not particularly sensitive at the tails of the distribution, since at these points both the hypothesized distribution and the sample's empirical distribution tend to come together at 0 or 1. Indeed, we have found that D_{max} is usually found near the median of the sample. Sometimes experimenters are particularly interested in the behavior of the rare hard problems in the right tail. Therefore, we have devised a measure called the Tail Ratio (TR) which focusses

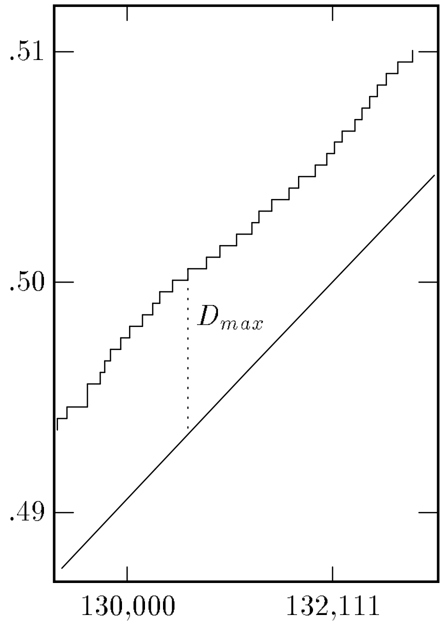


Figure 3.3: Computing the KS statistic D_{max}, when comparing the cdf for $\Lambda(12.00, 0.44)$ (smooth line) and the empirical cdf for the experiment described in the fifth row of Fig. 3.6 (stepwise line). Note that only a small part of the curve near the maximum difference is shown.

on the goodness-of-fit in the right tail of the distribution. In general, the tail ratio of two distributions with cdfs F_1 and F_2 is parameterized by a value α, and is the ratio of probabilities that a random variable will be greater than α in each distribution:

$$\mathrm{TR}_\alpha = (1 - F_1(\alpha))/(1 - F_2(\alpha)). \tag{3.24}$$

In practice, we always set α equal to the number of nodes explored for the sample instance at the 99th percentile. For example, out of 5,000 instances the 4,950th hardest one might require 2,000,000 nodes, so $\alpha = 2{,}000{,}000$. F_1 is the cdf of the lognormal or Weibull function, and F_2 is the empirical distribution function for the sample. Because of the way α is selected, $F_2(\alpha) = .99$. Thus,

$$\mathrm{TR} = \frac{1 - F(\text{99th percentile in sample})}{1 - .99} \tag{3.25}$$

where F is the appropriate cdf. If the TR is 1.0, the match is perfect and the distribution accurately models the 99th percentile of the right tail. A number less than 1 indicates that the distribution does not predict as many hard instances as were actually encountered; when greater than 1 the TR indicates the distribution predicts too many hard instances.

3.4.7 Estimating Distribution Parameters

The field of statistics has developed methods for estimating the parameters of a distribution from sample values. It is almost always assumed that the distribution function is known. The "universal method for optimal estimation of unknown parameters" [75] is the maximum likelihood method, which chooses parameter values which maximize the probability that the observed sample would occur.

For the lognormal distribution we use the maximum likelihood estimator (MLE) [11]. Let $\{x_1, x_2, \ldots, x_n\}$ be the n samples. The estimate of μ, called $\hat{\mu}$, and of σ^2, called $\hat{\sigma}^2$, are computed as

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} \ln x_i \tag{3.26}$$

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n} (\ln x_i - \hat{\mu})^2. \tag{3.27}$$

The MLE for the Weibull distribution is not recommended when $\beta < 2$ [11], which is usually the case for the data we encounter

We therefore use a modified moment estimator (MME) [11]. Again, let the n samples be $\{x_1, x_2, \ldots, x_n\}$, with x_1 the smallest in the set. Let $\bar{x}$ be the sample mean, s^2 be the sample variance, and $\hat{\lambda}$, $\hat{\beta}$ and $\hat{\alpha}$ be the estimates for the Weibull parameters. Where $\Gamma(\cdot)$ is the Gamma function, let

$$\hat{\Gamma}_k = \Gamma\left(1 + \frac{k}{\hat{\beta}}\right). \tag{3.28}$$

The first step of the MME is to iterate over possible values of $\hat{\beta}$, finding a value which satisfies the equality

$$\frac{s^2}{(\bar{x} - x_1)^2} = \frac{\hat{\Gamma}_2 - \hat{\Gamma}_1^2}{\left((1 - n^{-1/\hat{\beta}})\hat{\Gamma}_1\right)^2}. \tag{3.29}$$

We varied $\hat{\beta}$ from 0.010 to 2.000 in steps of .001, selecting the value that minimized the difference between the two sides in (3.29). With $\hat{\beta}$ derived, the MME computes the other parameters as

$$\hat{\lambda} = \frac{s}{\sqrt{\hat{\Gamma}_2 - \hat{\Gamma}_1^2}} \tag{3.30}$$

$$\hat{\alpha} = \bar{x} - \hat{\lambda}\hat{\Gamma}_1. \tag{3.31}$$

3.5 Experiments

The primary goal of the experiments is to show the goodness-of-fit between the empirical distributions of the samples and the lognormal (for unsolvable problems) and Weibull (for solvable problems) distributions. We use both Model A and Model B random problem generators. We view the random CSP generator as defining a distribution over the set of all CSPs. This distribution is a distribution over CSP instances, not over the work to process these instances. For a given set of parameters $\langle N, D, C, T \rangle$, CSPs which cannot be output by the generator have a probability of 0 in the distribution, while all CSPs which can be generated have an equal probability under the distribution.

Model B has been more widely used in recent empirical studies of CSP algorithms than Model A. The advantage of Model A, for the purposes of this chapter, is that the independence of constraints and prohibited value pairs simplifies the analysis of why the lognormal distribution appears in unsatisfiable problems. We discuss this topic in section 3.6.

Because there are many experiments to report, we start with an outline of the section. In subsection 3.5.1 we show that the distributions are observed under a variety of algorithms; the experiments in this section use Model A with combinations of parameters that are at or near the 50% solvable cross-over point. Section 3.5.2 show how increasing the sample size, from 100 up to 1,000,000 instances, reduces

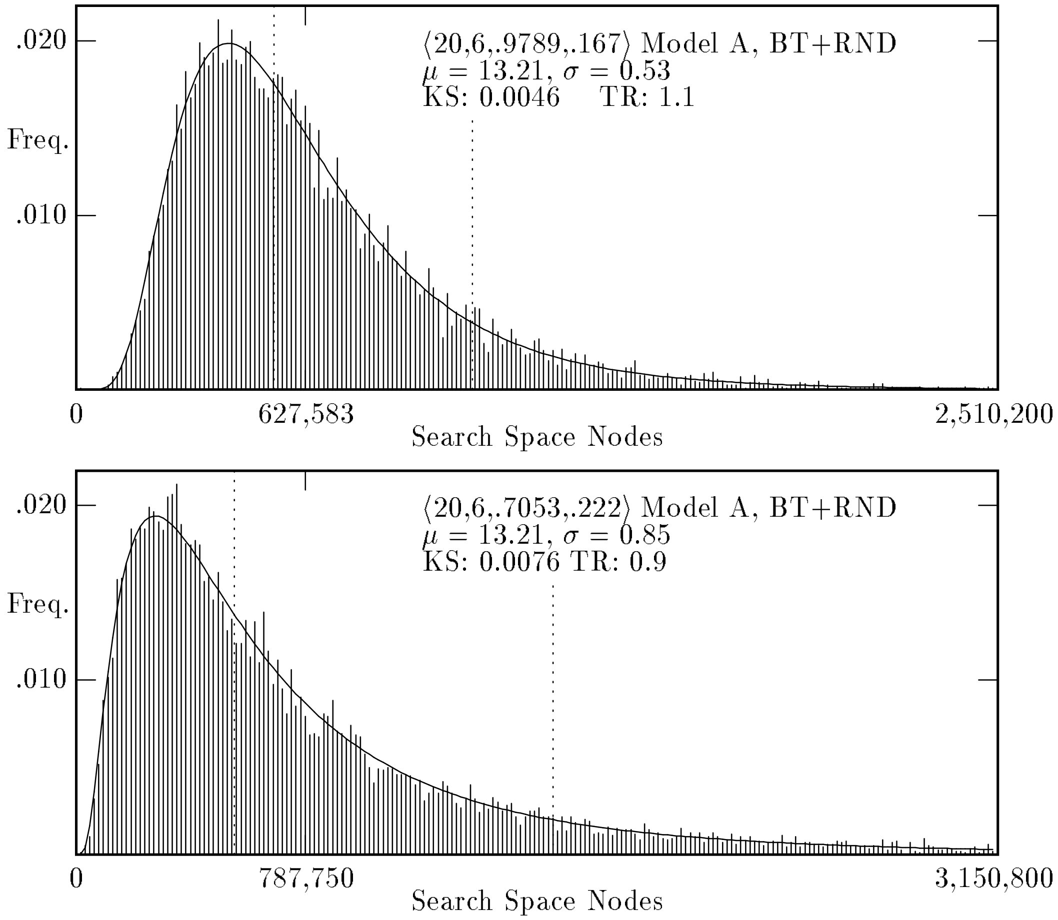


Figure 3.4: Graphs of sample data (vertical bars) and the lognormal distribution (curved line). Based on the unsolvable portion of 20,000 instances for each graph. Algorithm is simple backtracking with random variable ordering (BT+RND).

the seemingly random fluctuations in the histogram bars in the graphs. In subsection 3.5.3, we show that empirical distributions can also be approximated by the lognormal and Weibull functions outside the 50% satisfiable region as well. Subsection 3.5.4 shows the impact of using Model B on the goodness-of-fit.

Throughout our experiments we employ the same procedures and report the data in a consistent format. Our experimental procedure consisted of selecting various sets of parameters for the random CSP generator, then generating 20,000 or more instances for each set, and applying one or more search algorithms to each instance. For each instance and algorithm we recorded whether a solution was found

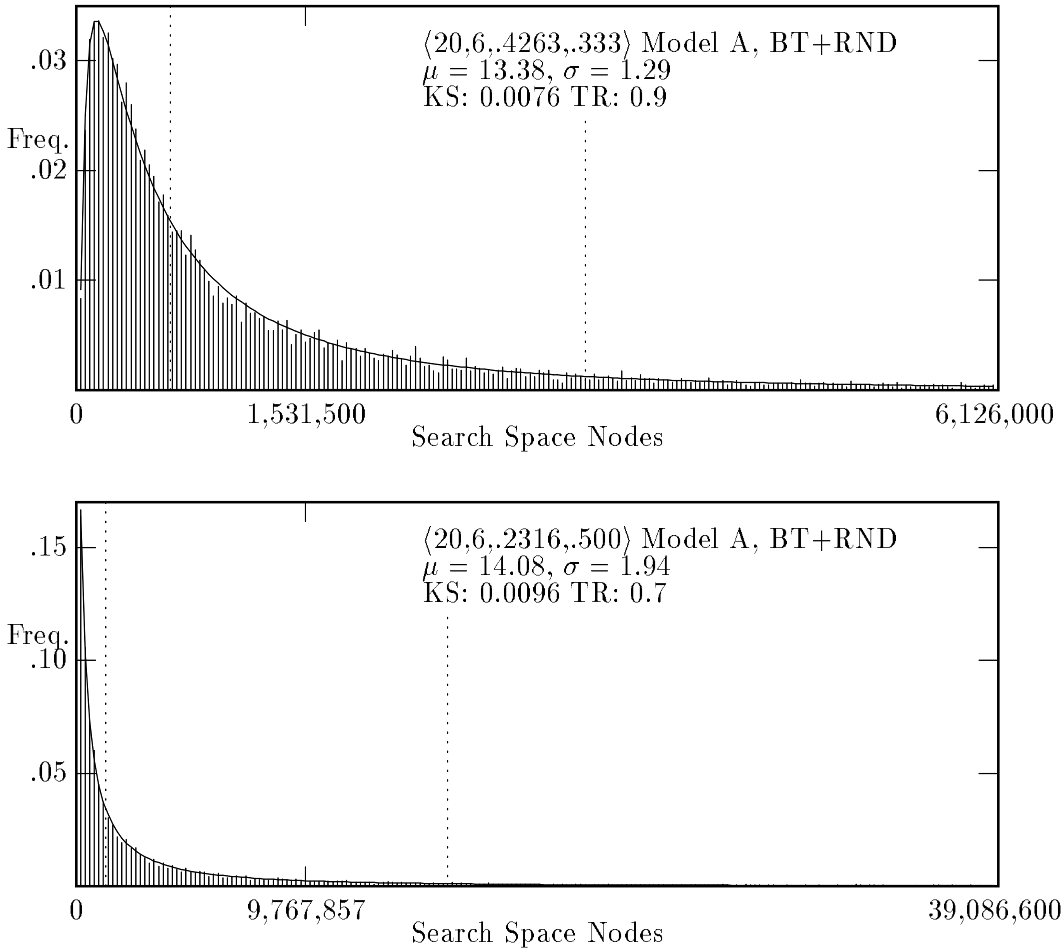


Figure 3.5: Continuation of Fig. 3.4, with different problem generator parameters.

and the number of nodes in the search space. We derived parameters for the distributions, and measured the closeness of the fit using the KS (Kolmogorov-Smirnov) and TR (Tail Ratio) statistics. Each line in Fig. 3.3 and Fig. 3.4 represents one experiment with one algorithm on the unsolvable (Fig. 3.3) or solvable (Fig. 3.4) instances generated from one set of parameters. The column labeled "Mean" in these tables shows the mean number of nodes for an experiment. The "μ", "σ", "λ", "β", and "α" columns show the estimated value for these parameters. Since values for λ are typically very small, we show them multiplied by 10^6. In the "solv" column is the percentage of solvable problems that were generated with the given parameters. The "KS" column holds the Kolmogorov-Smirnov statistic, and the tail ratio measure is reported in the "TR" column.

We also show some experimental results and distributions in a pictorial fashion, as in Fig. 3.4, Fig. 3.5, Fig. 3.6, and Fig. 3.7. These graphs can be difficult to interpret, because each one has its own scale for the x and y axes. The sample data is represented by a histogram of vertical bars. The samples in each case have been grouped into 200 bins; a problem that required between $(i-1) \times w + 1$ and $i \times w$ nodes is reported in the ith bin. The "width" of the bin, w, is set to be equal to one-fiftieth of the mean, unless noted otherwise. When this is so the sample mean, which is noted on the x-axis, is one fourth of the way across the graph, from left to right. Instances greater than four times the mean are not pictured, but are considered in estimating the parameters. The height of each vertical line is in proportion to the fraction of the sample that fell within the corresponding range of nodes, as indicated on the y-axis. The continuous line on the charts indicates the distribution function; the height of the line at i, $0 < i \leq 200$, is proportional to $F(i \times w) - F((i-1) \times w)$, where F is the cdf and w is the bin width. The vertical dotted lines indicate the median and 90th percentile of the data.

For example, in the top graph of Fig. 3.4, the mean of the 12,125 unsolvable instances was 627,583. Dividing the mean by 50 and ignoring the remainder yields a bin width, w, of 12,551. The 100th bin, to focus on one in particular, contains 34 samples which required between 1,242,550 and 1,255,100 nodes in the search. The height of the histogram line representing bin 100 is therefore $34/12125 = 0.002804$. Using the lognormal cdf F with parameters $\mu = 13.21$ and $\sigma = 0.53$, the height of the curved line at $x = 100$ is $F(1225100) - F(1242550) = 0.941210 - 0.938968 = 0.002242$. Since $0.002242 < 0.002804$, at this point along the x-axis the vertical line extends above the curved line. Samples above $200 \times 12551 = 2510200$ are not shown on the graph. There were 17 such instances in this experiment, ranging from 2,583,312 to 5,024,460. Truncating the extreme right tails of the graphs is

$Model\langle$ N, D, C, T $\rangle$	Unsolvable / Lognormal					
Parameters	Mean	μ	σ	solv	KS	TR
Algorithm: BT+RND						
A $\langle$20, 6, .9789, .167$\rangle$	627,583	13.21	0.53	39%	0.0046	1.1
A $\langle$20, 6, .7053, .222$\rangle$	787,750	13.21	0.85	40%	0.0076	0.9
A $\langle$20, 6, .4263, .333$\rangle$	1,531,500	13.38	1.29	42%	0.0111	0.9
A $\langle$20, 6, .2316, .500$\rangle$	9,601,654	14.08	1.94	43%	0.0096	0.7
Algorithm: BJ+RND						
A $\langle$20, 6, .9789, .167$\rangle$	180,251	12.00	0.44	39%	0.0077	1.0
A $\langle$20, 6, .7053, .222$\rangle$	167,360	11.79	0.68	40%	0.0078	0.9
A $\langle$20, 6, .4263, .333$\rangle$	149,273	11.41	0.99	42%	0.0096	0.8
A $\langle$20, 6, .2316, .500$\rangle$	117,744	10.68	1.36	43%	0.0122	1.4
Algorithm: FC+RND						
A $\langle$20, 6, .9789, .167$\rangle$	9,421	9.05	0.45	39%	0.0036	1.0
A $\langle$20, 6, .7053, .222$\rangle$	8,477	8.81	0.68	40%	0.0071	0.9
A $\langle$20, 6, .4263, .333$\rangle$	8,659	8.49	1.05	42%	0.0124	0.7
A $\langle$20, 6, .2316, .500$\rangle$	20,296	8.34	1.66	43%	0.0189	0.5
Algorithm: BT+MW						
A $\langle$20, 6, .9789, .167$\rangle$	509,301	13.01	0.51	39%	0.0058	1.0
A $\langle$20, 6, .7053, .222$\rangle$	85,687	11.19	0.57	40%	0.0096	0.8
A $\langle$20, 6, .4263, .333$\rangle$	17,855	9.50	0.75	42%	0.0129	0.8
A $\langle$20, 6, .2316, .500$\rangle$	4,381	7.90	0.93	43%	0.0218	0.9
Algorithm: BJ+DVO						
A $\langle$20, 6, .9789, .167$\rangle$	639	6.42	0.29	39%	0.0073	1.1
A $\langle$20, 6, .7053, .222$\rangle$	321	5.71	0.35	40%	0.0068	1.0
A $\langle$20, 6, .4263, .333$\rangle$	136	4.81	0.46	42%	0.0128	0.8
A $\langle$20, 6, .2316, .500$\rangle$	53	3.78	0.58	43%	0.0357	0.4

Table 3.3: Goodness-of-fit between unsolvable CSP instances and the lognormal distribution. This table compares a variety of algorithms.

unfortunate but essential for maintaining a scale appropriate to the rest of the graph.

3.5.1 First Set of Experiments: Variety of Algorithms

The first set of experiments are designed to explore how well the lognormal and Weibull distributions fit the data over a range of settings of C and T, and for

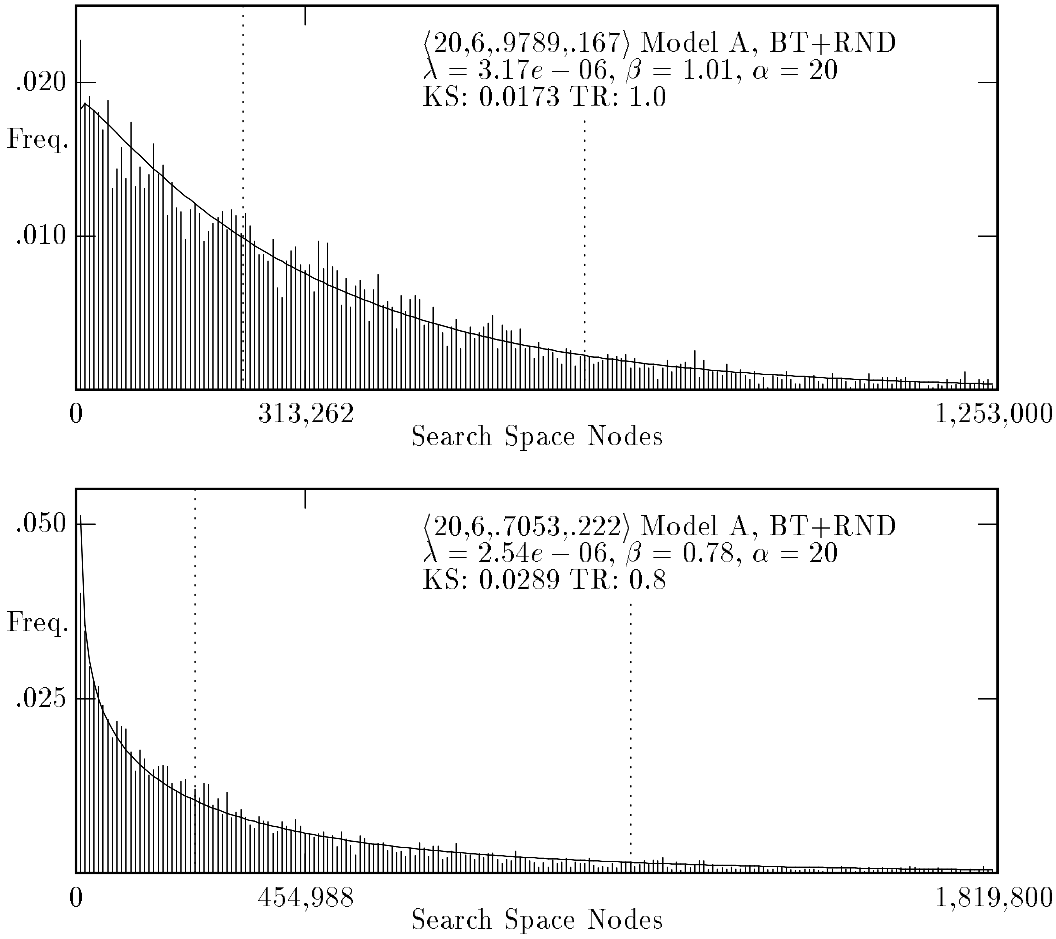


Figure 3.6: Graphs of sample data (vertical bars) and the Weibull distribution (curved line). Based on the solvable portion of 20,000 instances for each graph. Algorithm is simple backtracking with random variable ordering (BT+RND).

a variety of algorithms. We set N=20 and D=6, and chose four combinations of T and C which result in a roughly equal number of solvable and unsolvable instances. The algorithms used are all based on backtracking search and are described in detail in Chapter 2. The basic algorithm, backtracking, can be run with a fixed random variable ordering (BT+RND) or using the min-width variable ordering heuristic (BT+MW). We also experiment with backjumping and a random variable ordering (BJ+RND) and forward checking with random variable ordering (FC+RND). We selected a variety of relatively simple algorithms in order to demonstrate that the correspondence with continuous distributions is not the consequence of any specific

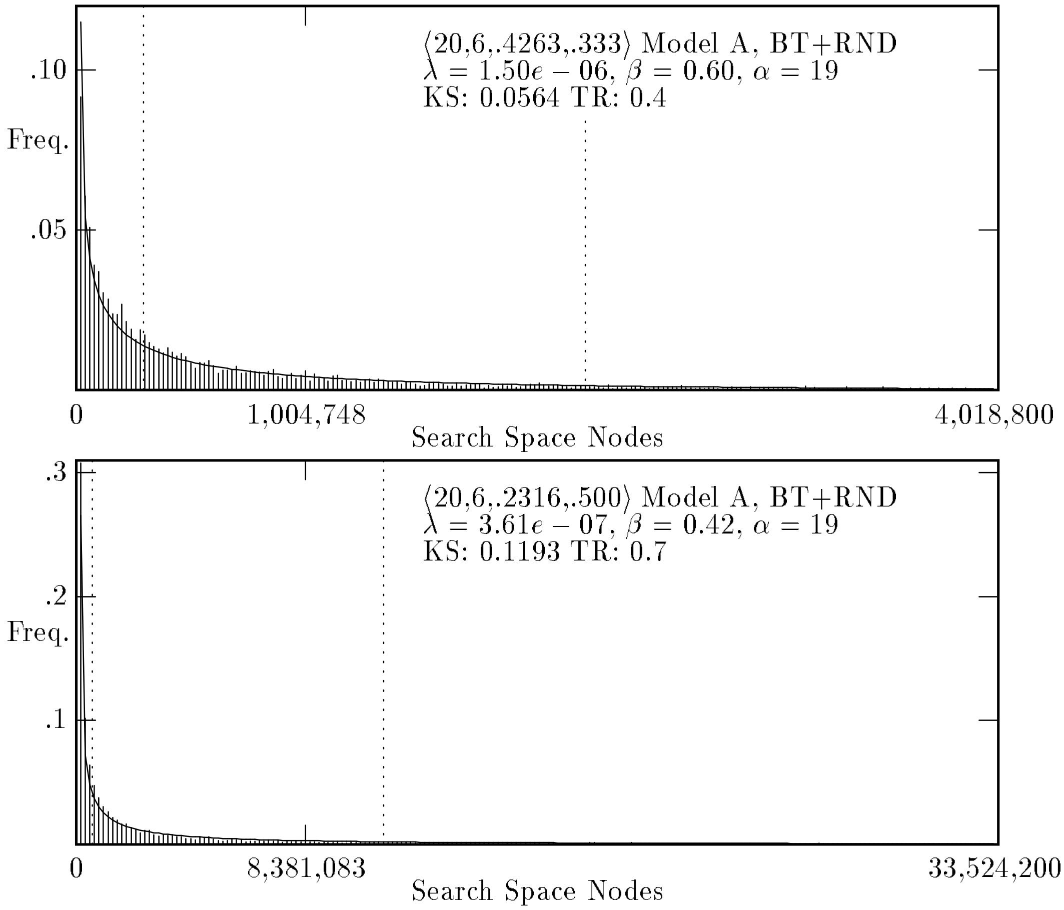


Figure 3.7: Continuation of Fig. 3.6.

heuristic, but holds for many varieties of backtracking search. As an example of a sophisticated variation of backtracking search that combines backjumping, forward checking style look-ahead, and a dynamic variable ordering heuristic, we use BJ+DVO, described in chapter 4.

The results of the experiments are presented in Table 3.3 (unsolvable problems) and Table 3.4 (solvable problems). The results for the experiments with BT+RND are shown graphically in Fig. 3.4, Fig. 3.5, Fig. 3.6, and Fig. 3.7.

The fit between unsolvable problems and the lognormal distribution is quite good. It is least good for the problems with parameters $\langle 20, 6, .2316, .500\rangle$, which have the sparsest graphs and the tightest constraints. For the other sets of parameters, the TR statistic ranges from 0.7 to 1.1, and KS is not greater than 0.0129. Fig. 3.4 and Fig. 3.5 show visually that the lognormal distribution accurately reflects the shape of the data.

The fit between the Weibull distribution and the distribution of solvable problems is much less good, but not entirely unsatisfactory. The visual evidence in Fig. 3.6 and Fig. 3.7 is that the fit is good enough to use the Weibull distribution for capturing the overall shape of the empirical distribution. The fit is least good to the "left" of the median line, where the Weibull distribution is often substantially larger or smaller than the observed value. It is possible to choose parameters for the Weibull distribution which create a closer fit on the easiest problems (the left part of the graph), but these parameters always cause an extreme mismatch between the Weibull and empirical distributions elsewhere on the graph.

We observe a pattern that holds for both solvable and unsolvable problems: the sparser the constraint graph, the greater the variance of the distribution, indicated by larger σ and smaller λ. The effect is visible in Table 3.3 and Table 3.4 when comparing rows with the same N, D, and algorithm. Parameters T and C are inversely related at the 50% satisfiable point, so the effect may be due to increasing T as well. The pattern holds even with BT. BJ, FC, and BT+MW can exploit tight constraints and a sparse graph to make such problems much easier. BT does not, but we still find greater variance with lower C.

Model⟨ N, D, C, T ⟩	Solvable / Weibull						
Parameters	Mean	$\lambda \times 10^6$	β	α	solv	KS	TR
Algorithm: BT+RND							
A ⟨20, 6, .9789, .167⟩	313,262	3.17	1.01	19	39%	0.0173	1.0
A ⟨20, 6, .7053, .222⟩	454,988	2.54	0.78	19	40%	0.0289	0.6
A ⟨20, 6, .4263, .333⟩	1,004,748	1.50	0.60	19	42%	0.0564	0.4
A ⟨20, 6, .2316, .500⟩	8,248,756	0.36	0.42	19	43%	0.1193	0.7
Algorithm: BJ+RND							
A ⟨20, 6, .9789, .167⟩	85,121	11.39	1.08	19	39%	0.0202	1.4
A ⟨20, 6, .7053, .222⟩	82,993	12.94	0.87	19	40%	0.0142	0.7
A ⟨20, 6, .4263, .333⟩	72,204	17.78	0.69	19	42%	0.0307	0.6
A ⟨20, 6, .2316, .500⟩	45,942	37.92	0.54	19	43%	0.0696	0.5
Algorithm: FC+RND							
A ⟨20, 6, .9789, .167⟩	4,794	200.34	1.12	19	39%	0.0158	1.3
A ⟨20, 6, .7053, .222⟩	4,920	214.65	0.89	19	40%	0.0155	0.7
A ⟨20, 6, .4263, .333⟩	5,810	219.69	0.69	19	42%	0.0388	0.5
A ⟨20, 6, .2316, .500⟩	18,129	132.14	0.46	19	43%	0.1014	0.7
Algorithm: BT+MW							
A ⟨20, 6, .9789, .167⟩	246,677	3.99	1.04	19	39%	0.0144	1.0
A ⟨20, 6, .7053, .222⟩	46,478	21.99	0.95	19	40%	0.0120	0.8
A ⟨20, 6, .4263, .333⟩	11,798	94.48	0.82	19	42%	0.0207	0.6
A ⟨20, 6, .2316, .500⟩	9,899	163.89	0.57	19	43%	0.1833	4.0
Algorithm: BJ+DVO							
A ⟨20, 6, .9789, .167⟩	324	2,817	1.41	0	39%	0.0276	1.4
A ⟨20, 6, .7053, .222⟩	182	5,008	1.38	0	40%	0.0322	1.1
A ⟨20, 6, .4263, .333⟩	91	10,035	1.39	0	42%	0.0741	0.5
A ⟨20, 6, .2316, .500⟩	47	19,478	1.46	0	43%	0.1631	0.1

Table 3.4: Goodness-of-fit between solvable CSP instances and the Weibull distribution. This chart compares a variety of algorithms.

3.5.2 1,000,000 instances

In the this section we report on an experiment in which almost two million CSP instances – approximately 1,000,000 unsolvable instances – were created and processed with the BJ+DVO algorithm. The goal was to explore visually the effect of varying the number of samples in a chart. One difficulty in interpreting charts such as those in Figs. 3.4 and 3.5 is the fluctuations in the heights of the histogram bars. Apparently more samples are necessary to smooth the histogram.

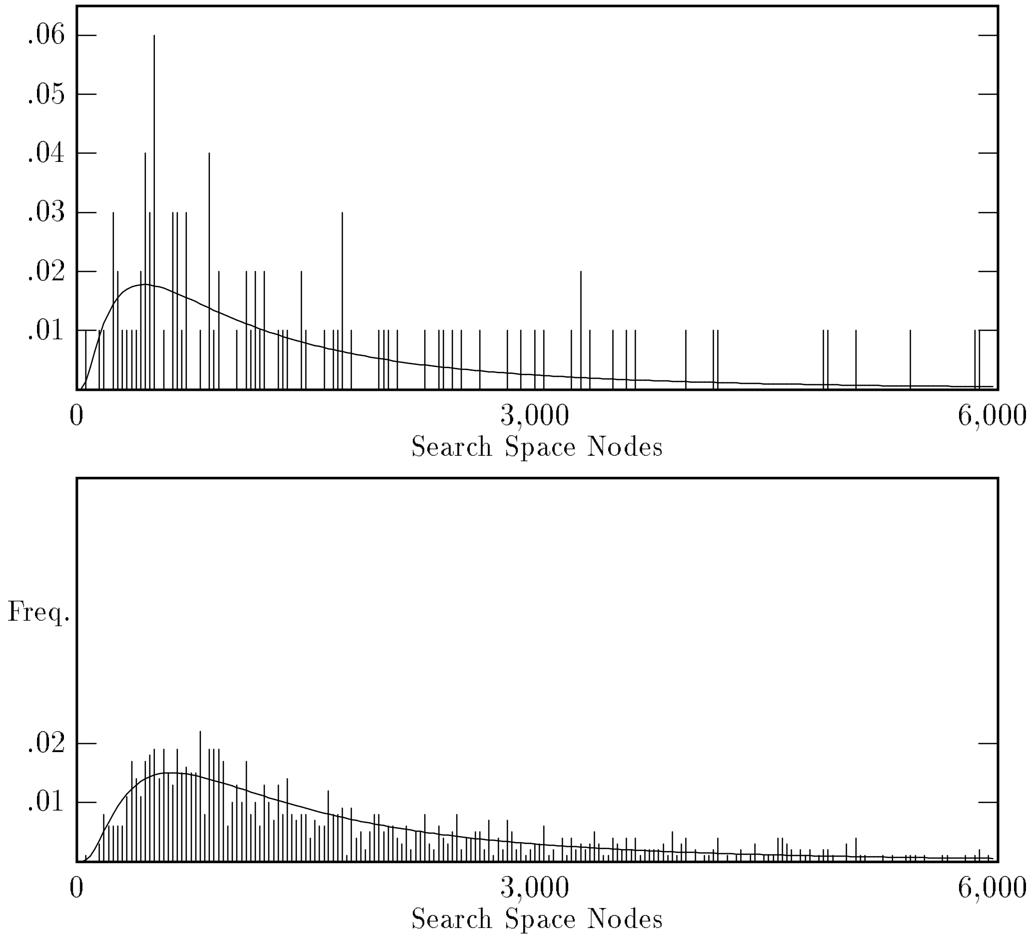


Figure 3.8: Unsolvable problems from parameters $\langle 50, 6, .1576, .333\rangle$, using algorithm BJ+DVO. The top chart is based on the first 100 unsolvable instances, the bottom chart on the first 1,000 unsolvable instances. Each histogram bar represents the number of sample instances in a range of 30 nodes.

The parameters used in this experiment were $\langle 50, 6, .1576, .333\rangle$, and only unsolvable instances are reported. The relevant charts are in Fig. 3.8 through Fig. 3.12. The samples displayed are cumulative; e.g. the 100 instances in the top chart of Fig. 3.8 are also included in the bottom chart, along with the next 900 in the experiment.

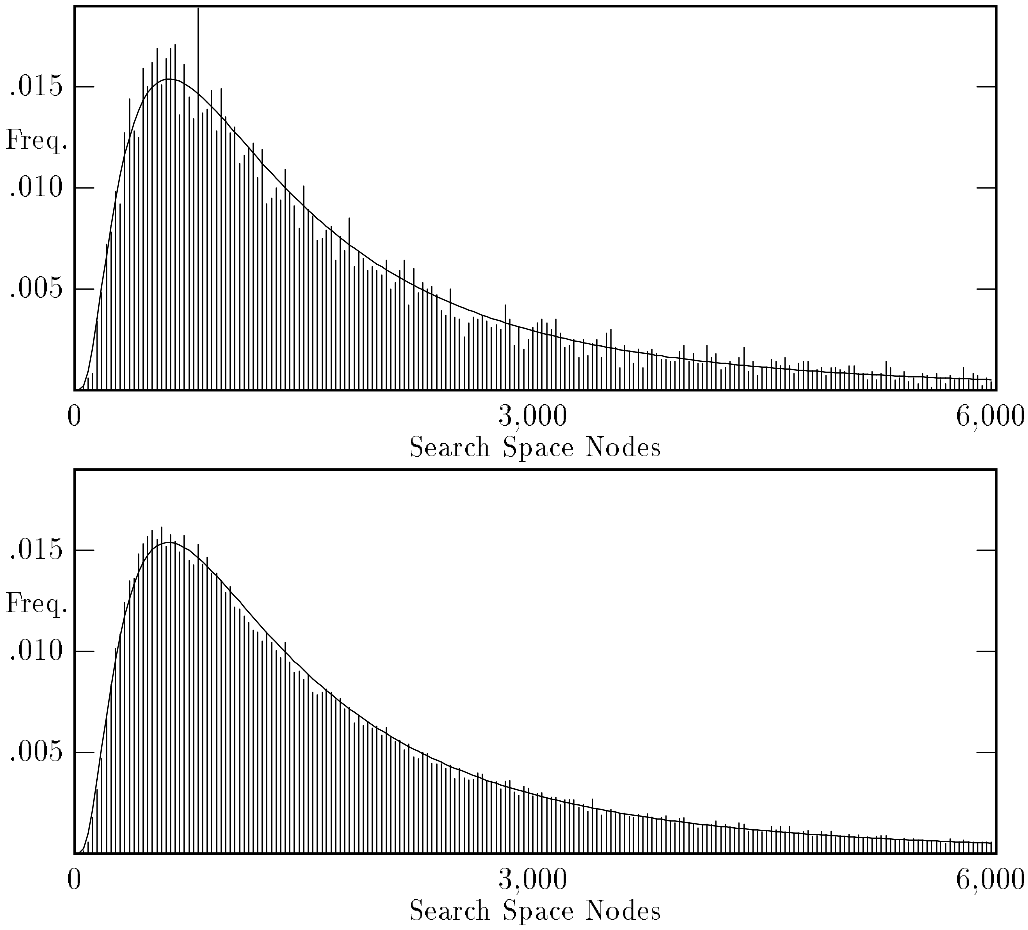


Figure 3.9: Continuation of Fig. 3.8. The top chart is based on the first 10,000 unsolvable instances, the bottom chart on the first 100,000 unsolvable instances.

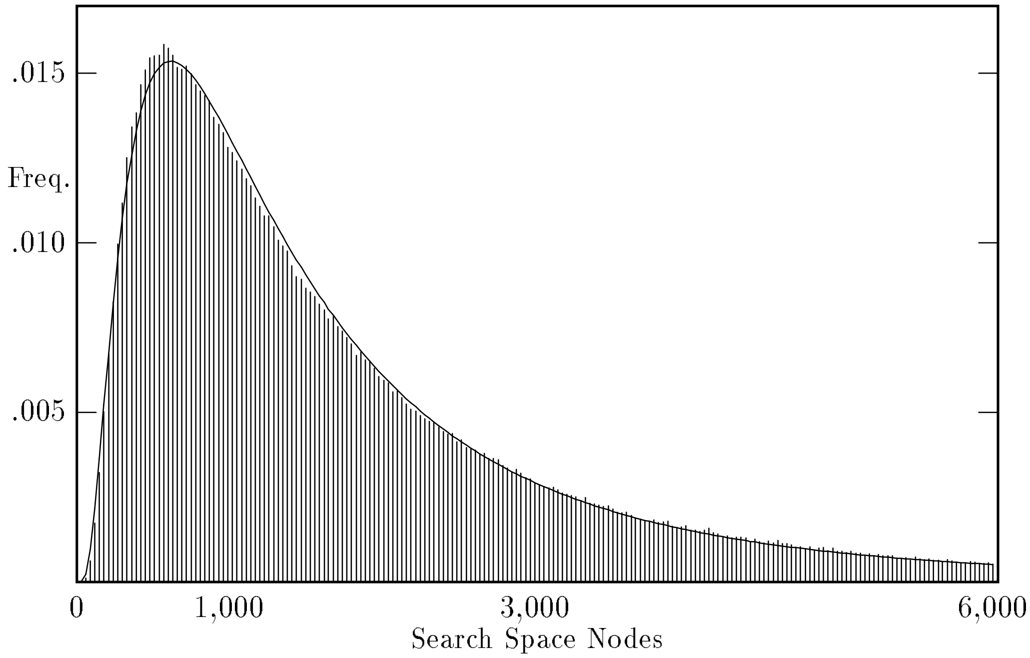


Figure 3.10: Continuation of Fig. 3.8 and Fig. 3.9. This chart is based on 1,000,000 unsolvable instances.

Figs. 3.8, 3.9, and 3.10 show histograms and lognormal distributions based on samples of sizes 100, 1,000, 10,000, 100,000, and 1,000,000. In each chart the same x-axis scale is used – one histogram bar represents a range of 30 nodes. The distributions are truncated above 6,000 nodes, so there are 200 bars in each chart. The y-axis scales are the same within each figure, but change from figure to figure. In each chart, the lognormal distribution pictured is based on parameters estimated from that chart's sample.

The top chart in Fig. 3.8 is based on 100 instances, and so displays many gaps in the histogram. Most of the histogram bars have a height of 0.00 (that is, no bar) or 0.01, because exactly zero or one instance (out of one hundred) fell into the corresponding range. With 100 instances the chart looks awful, and it is nearly impossible to tell by eye the goodness-of-fit between the sample data and the lognormal distribution display. If this were the only set of data to display, clearly a better scale could be chosen.

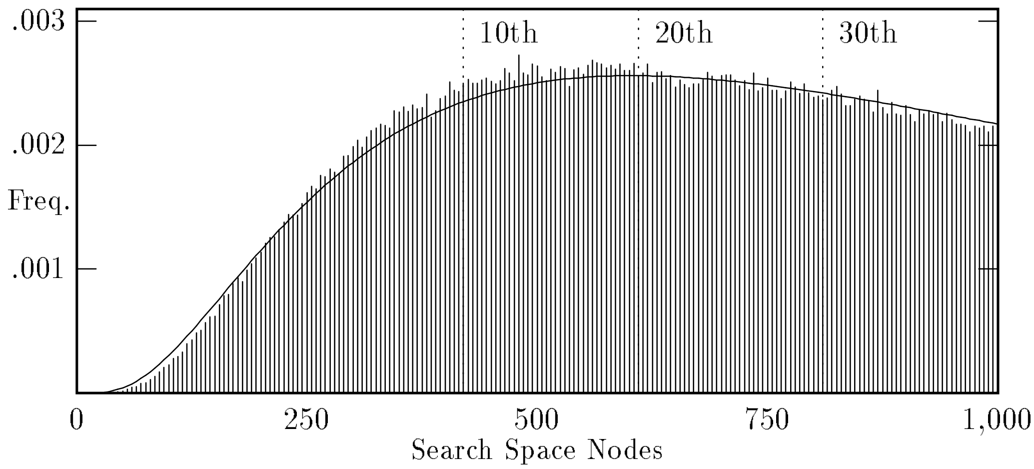


Figure 3.11: A "close-up" view of the data in Fig. 3.10. Each histogram bar represents a range of 5 nodes (in comparison to 30 in Fig. 3.10; only instances with search space less than 1,000 nodes. The dotted lines indicate the 10th, 20th, and 30th percentile lines in the sample.

In the subsequent charts in Figs. 3.8, 3.9, and 3.10 the histograms look smoother and smoother. By 1,000,000 samples (Fig. 3.10) almost all fluctuation from bar to bar has been dampened. It is also possible to see that there is a systematic mismatch between empirical distribution and the lognormal distribution, particularly at the mode, where the peak of the empirical distribution seems to be a bit higher and to the left of the lognormal. It is possible, of course, that this mismatch is due to sampling error and would disappear if sufficient further instances were added to the sample. It is more likely that the true underlying distribution of problem difficulty diverges slightly from the lognormal distribution. Because solving CSPs requires finite units of work and the lognormal and Weibull distributions are continuous, we can never expect perfect agreement.

To explore the mismatch around the mode more closely, Fig. 3.11 shows a subset of the data from Fig. 3.10, grouped in histogram bars of width 5, and truncated above 1,000 nodes. In this "close-up" view, each bar represents about one sixth as much data as in Fig. 3.10. The lognormal distribution (with parameters estimated by the maximum likelihood estimator) predicts slightly too many instances around 100 nodes, too few instances in the range of 250 to 600 nodes,

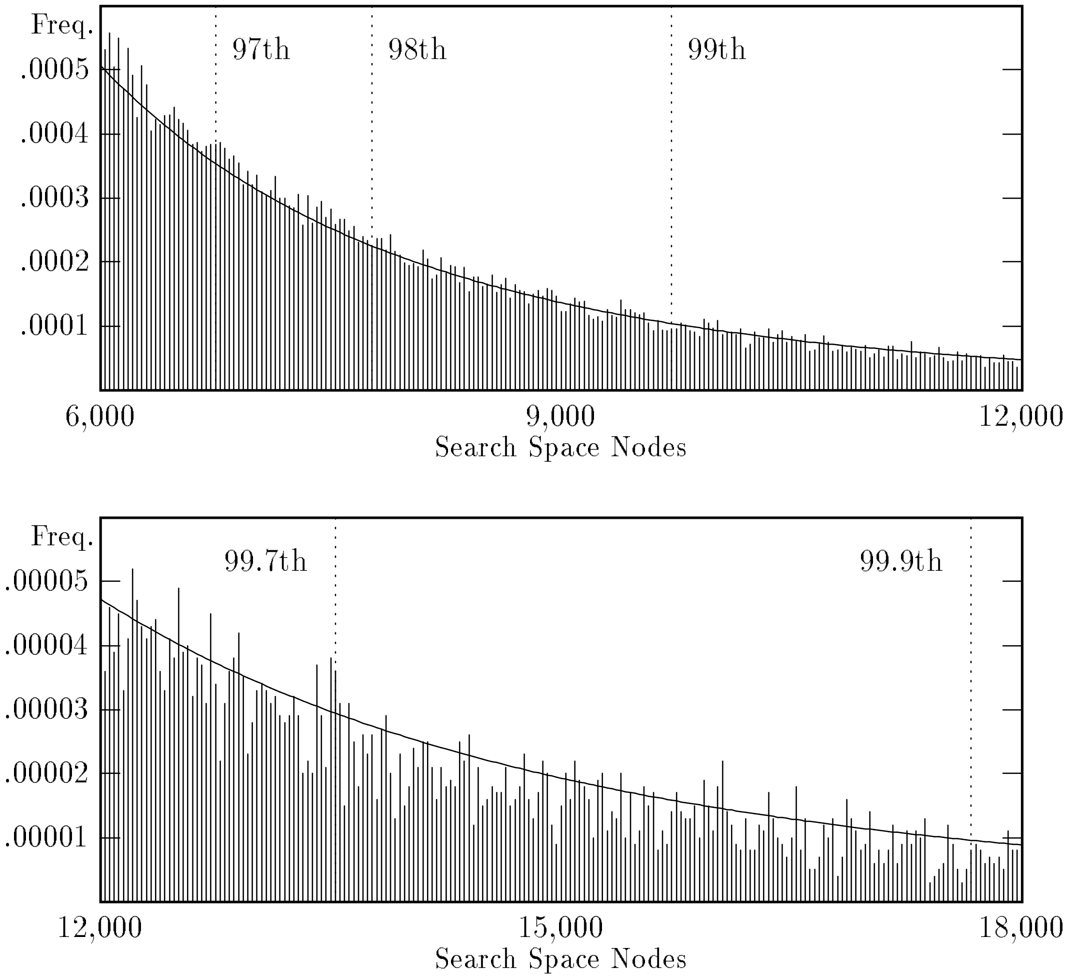


Figure 3.12: The two graphs are continuations of Fig. 3.10, showing data in the tail of the distributions. The top chart extends Fig. 3.10 from 6,000 to 12,000 nodes, and the bottom chart continues from 12,000 to 18,000 nodes. The x-axis scale is the same in Fig. 3.10 and these charts – 30 nodes, but the vertical scale varies. The dotted lines show the indicated percentiles in the sample.

sample size	sample mean	μ	σ
50	1,703	6.969	1.046
100	1,723	7.012	0.981
1,000	1,971	7.192	0.885
10,000	1,913	7.173	0.876
100,000	1,912	7.172	0.879
1,000,000	1,914	7.173	0.880

Table 3.5: Mean number of nodes and estimated values of μ and σ for the lognormal distribution, using the maximum likelihood estimator, based on varying sample sizes.

and just about the right number from 600 to 1,000 nodes. The visual evidence, from this one experiment, is that the sample is not drawn from an exactly lognormal distribution, but that the approximation is sufficiently close for almost all purposes.

In Fig. 3.12 we show some of the tail of the distribution with 1,000,000 instances. This figure should be viewed as a continuation of Fig. 3.10, extending the chart to those instances with a search space of 6,000 to 12,000 nodes (top chart) and 12,000 to 18,000 nodes (bottom chart). The x-axis scale of the three charts is identical (30 nodes), but the y-axis scales are modified as appropriate. Again the goodness-of-fit of the lognormal seems quite good to the eye, although above the 99.7th percentile it seems to be overestimating slightly the actual number of instances.

It is interesting to observe how the maximum likelihood estimates of the lognormal parameters change as the sample size increases. This data is tabulated in Table 3.5. The estimates of μ and σ are reasonably accurate after even 50 samples, and by 1,000 samples they are almost identical to their values at 1,000,000 instances.

3.5.3 Parameters not at the cross-over point

$Model\langle$ N, D, C, T $\rangle$	Unsolvable / Lognormal					
Parameters	Mean	μ	σ	solv	KS	TR
Algorithm: BJ+DVO						
A $\langle$20, 6, .5263, .222$\rangle$	27,906	9.93	0.79	99%	0.1147	1.0
A $\langle$20, 6, .5526, .222$\rangle$	21,390	9.70	0.72	99%	0.0623	0.1
A $\langle$20, 6, .5789, .222$\rangle$	18,138	9.52	0.75	98%	0.0327	0.7
A $\langle$20, 6, .6053, .222$\rangle$	16,595	9.44	0.73	94%	0.0176	0.6
A $\langle$20, 6, .6316, .222$\rangle$	14,231	9.29	0.73	86%	0.0168	0.7
A $\langle$20, 6, .6579, .222$\rangle$	11,947	9.13	0.71	73%	0.0078	0.8
A $\langle$20, 6, .6842, .222$\rangle$	10,014	8.96	0.69	55%	0.0114	0.9
A $\langle$20, 6, .7105, .222$\rangle$	8,279	8.78	0.69	37%	0.0094	0.9
A $\langle$20, 6, .7368, .222$\rangle$	6,544	8.55	0.67	22%	0.0109	0.8
A $\langle$20, 6, .7632, .222$\rangle$	5,175	8.33	0.65	11%	0.0145	1.0
A $\langle$20, 6, .7895, .222$\rangle$	4,043	8.11	0.62	5%	0.0141	0.7
A $\langle$20, 6, .8158, .222$\rangle$	3,122	7.87	0.58	2%	0.0136	0.6
A $\langle$20, 6, .8421, .222$\rangle$	2,458	7.65	0.55	1%	0.0147	0.6
A $\langle$20, 6, .8684, .222$\rangle$	1,946	7.44	0.51	0%	0.0148	0.7
A $\langle$20, 6, .8947, .222$\rangle$	1,574	7.25	0.47	0%	0.0102	0.8
A $\langle$20, 6, .9211, .222$\rangle$	1,295	7.07	0.44	0%	0.0126	0.8
A $\langle$20, 6, .9474, .222$\rangle$	1,078	6.90	0.41	0%	0.0082	0.7
A $\langle$20, 6, .9737, .222$\rangle$	908	6.74	0.38	0%	0.0065	0.8
A $\langle$20, 6, 1.000, .222$\rangle$	773	6.59	0.34	0%	0.0029	0.9

Table 3.6: Goodness-of-fit for a set of experiment with fixed values for N, D and T and varying values of C. Unsolvable problems; algorithm is FC+RND.

Model⟨ N, D, C, T ⟩	Solvable / Weibull					
Parameters	Mean	λ	β	solv	KS	TR
Algorithm: BJ+DVO						
A ⟨20, 6, .5000, .222⟩	1,700	1,176.75	0.50	100%	0.0537	0.8
A ⟨20, 6, .5263, .222⟩	2,065	861.27	0.54	99%	0.0342	0.6
A ⟨20, 6, .5526, .222⟩	2,587	614.72	0.58	99%	0.0352	0.7
A ⟨20, 6, .5789, .222⟩	3,221	444.70	0.62	98%	0.0257	0.7
A ⟨20, 6, .6053, .222⟩	3,827	339.84	0.68	94%	0.0187	0.8
A ⟨20, 6, .6316, .222⟩	4,597	263.44	0.74	86%	0.0154	0.7
A ⟨20, 6, .6579, .222⟩	4,849	215.65	0.87	73%	0.0154	0.7
A ⟨20, 6, .6842, .222⟩	4,981	218.92	0.94	55%	0.0174	0.7
A ⟨20, 6, .7105, .222⟩	4,696	217.47	0.95	37%	0.0168	0.7
A ⟨20, 6, .7368, .222⟩	4,412	231.24	0.96	22%	0.0186	0.6
A ⟨20, 6, .7632, .222⟩	3,876	256.53	1.01	11%	0.0314	0.5
A ⟨20, 6, .7895, .222⟩	3,162	311.33	1.04	5%	0.0393	0.7
A ⟨20, 6, .8158, .222⟩	2,748	351.12	1.10	2%	0.0384	1.1
A ⟨20, 6, .8421, .222⟩	2,221	441.03	1.05	1%	0.0538	0.5

Table 3.7: Goodness-of-fit for a set of experiment with fixed values for N, D and T and varying values of C. Solvable problems; algorithm is FC+RND.

In the previous experiments the parameters were selected such that approximately equal numbers of solvable and unsolvable instances were generated. Parameters at the cross-over point are used by many experimenters when evaluating algorithms, and so are of particular interest. We now report on a set of experiments which indicate that the lognormal and Weibull distributions remain good approximations of the empirical distributions for problems generated by parameters not at the cross-over point. Our technique is to fix N=20, D=6, and T=.222 and then to vary parameter C from .500, where all problems have solutions, up to 1.00, where no problems have solutions. C is incremented in steps of .0263, which corresponds to 5 constraints. The results are reported in Table 3.6 (problems without solutions) and Table 3.7 (problems with solutions).

In the unsolvable problems, there is a clear trend of the Kolmogorov-Smirnov statistic improving (decreasing) with increasing C and decreasing percent solvable. The Tail Ratio does not exhibit any pattern. The cause of the improving K-S figure is difficult to pinpoint. It may be related to a similar pattern seen in Fig. 3.3, where the experiments with higher values of C tend to have lower values of σ and better goodness-of-fit, as measured by K-S.

In contrast to the data from unsolvable instances, the solvable problems show a clear deterioration of goodness-of-fit between the Weibull distribution and the experimental samples when the problem parameters are not near the cross-over point. As with the unsolvable problems, there may be a confounding factor, in this case a correlation between the number of nodes and the K-S statistics. For relatively low and high values of C (e.g below .6 or above .8) the problems are easier and less like the Weibull distribution.

Even when focusing exclusively on parameter combinations at the cross-over point, there are still so many combinations that extrapolation from empirical data must be done with caution. If we look at the complete range of possible parameters, it becomes much more difficult to know whether results from one cross-section of the four-dimensional parameter space have any predictive value for other sections of the space. We therefore limit our conclusions in this section to the statement that we have seen no evidence, either in Tables 3.6 and 3.7 or in other experiments we have conducted, to indicate a severe fall off in goodness-of-fit when problem parameters are not at the 50% solvable point.

3.5.4 Model A and Model B

The experiments reported above were all conducted with Model A, in which the parameters C and T specify the probability of a constraint and of a prohibited value pair, respectively. As discussed earlier, Model B has become more widely used in recent years, and therefore we conducted a set of experiments with Model B. Results in [29] were based on experiments with Model B. Our primary goal was to determine whether the change in generating model, and thus distribution of CSP instances, makes an appreciable difference in the applicability of the lognormal and Weibull distributions to summarizing empirical distributions. The results we report in this chapter will have limited interest if they are contingent on a particular family of problem distributions. Although Model A and Model B are similar distributions of CSPs, we view this comparison as a first step towards judging the wider applicability of the continuous distribution functions.

The results are presented in Fig. 3.13 (for unsolvable problems) and Fig. 3.14 (for solvable problems). We have sorted the experiments in descending order of parameter C, the number of constraints (third number in the angle brackets), because this highlights an interesting pattern. On problems with relatively dense graphs, approximately $C > .1500$, the lognormal distribution fits Model A problems better than it fits Model B problems. When C is less than .1500, the fit is better with Model B. The cause of this pattern is unknown. Our conclusion from the data presented is that the choice of Model A or Model B does not substantially affect the goodness-of-fit of the lognormal and Weibull distributions.

3.6 Distribution Derivations

Selecting the best distribution, or model, to describe a set of data is as much an art as a science. It is often the case that no simple textbook distribution provides a completely satisfactory fit for all samples. Moreover, more than one distribution may match the data equally well, or different distributions may be superior for different samples from the same family of underlying distributions. Rish and Frost [73] report that unsatisfiable CSPs can be modelled reasonably well by several other distributions besides the lognormal. In particular, the gamma distribution can in some cases provide a better fit to a sample dataset than the lognormal distribution. Gomes *et al.* [38] propose using the Pareto distribution to model a set of satisfiable instances.

Model⟨ N, D, C, T ⟩	Unsolvable / Lognormal					
Parameters	Mean	μ	σ	solv	KS	TR
Algorithm: BJ+DVO						
A ⟨50, 6, .3722, .167⟩	19,807	9.72	0.59	32%	0.0093	1.3
A ⟨50, 6, .2653, .222⟩	8,220	8.78	0.70	35%	0.0114	1.4
A ⟨60, 6, .2260, .222⟩	19,343	9.55	0.80	17%	0.0093	1.2
A ⟨50, 6, .1576, .333⟩	1,911	7.18	0.87	38%	0.0095	1.2
A ⟨60, 6, .1356, .333⟩	3,404	7.65	0.98	18%	0.0191	1.0
A ⟨75, 6, .0577, .500⟩	1,022	5.86	1.46	13%	0.0391	0.6
Algorithm: BJ+DVO						
B ⟨50, 6, .3722, .167⟩	40,488	10.55	0.36	43%	0.0109	1.1
B ⟨50, 6, .2653, .222⟩	17,048	9.63	0.48	48%	0.0148	1.3
B ⟨60, 6, .2260, .222⟩	38,465	10.42	0.52	14%	0.0108	1.2
B ⟨50, 6, .1576, .333⟩	3,826	8.02	0.68	53%	0.0097	1.4
B ⟨60, 6, .1356, .333⟩	6,674	8.52	0.75	15%	0.0177	1.4
B ⟨75, 6, .0577, .500⟩	2,290	6.82	1.31	13%	0.0157	0.7

Figure 3.13: This table compares goodness-of-fit between Model A and Model B, using unsolvable problems and the lognormal distribution. The rows are arranged in decreasing order of parameter C, fraction of constraints. Parameters are in bold face when the lognormal fit Model A better than Model B, based on the KS statistic.

Model⟨ N, D, C, T ⟩	Solvable / Weibull					
Parameters	Mean	λ	β	solv	KS	TR
Algorithm: BJ+DVO						
A ⟨50, 6, .3722, .167⟩	12,748	78.82	0.99	32%	0.0152	1.2
A ⟨50, 6, .2653, .222⟩	5,667	183.41	0.92	35%	0.0135	1.0
A ⟨50, 6, .1576, .333⟩	1,531	711.05	0.85	38%	0.0463	0.6
A ⟨50, 6, .0832, .500⟩	386	2,958.40	0.79	36%	0.1820	0.2
A ⟨60, 6, .2260, .222⟩	17,760	59.06	0.91	17%	0.0287	1.7
A ⟨60, 6, .1356, .333⟩	4,329	256.19	0.83	18%	0.0377	0.6
Algorithm: BJ+DVO						
B ⟨50, 6, .3722, .167⟩	19,563	47.99	1.21	43%	0.0258	1.5
B ⟨50, 6, .2653, .222⟩	8,637	110.77	1.13	48%	0.0140	1.1
B ⟨50, 6, .1576, .333⟩	2,212	450.19	1.01	53%	0.0207	0.6
B ⟨50, 6, .0832, .500⟩	505	2,125.03	0.87	52%	0.1313	0.2
B ⟨60, 6, .2260, .222⟩	24,430	38.92	1.15	14%	0.0222	1.1
B ⟨60, 6, .1356, .333⟩	5,090	193.10	1.04	15%	0.0391	0.3

Figure 3.14: Comparing goodness of fit when generating problems with Model A and Model B – solvable problems.

For these reasons, it is appealing to select a model not only on the basis of goodness-of-fit to several sets of samples of data, but also because the theory behind the distribution corresponds with our understanding of the process being modelled, in this case backtracking search. In this section we briefly outline such a correspondence between the lognormal distribution and unsatisfiable problems, and between the Weibull distribution and satisfiable problems. The goal is a better appreciation of the degree to which the statistical model can capture the distribution of backtracking search on CSPs, and perhaps a heightened understanding of the backtracking search process itself.

3.6.1 Deriving the Lognormal

Several models have been proposed to derive the lognormal distribution [1]. One approach is to start directly with the definition of the lognormal distribution. Recall that a positive random variable X is lognormally distributed with parameters μ and σ^2 if $Y = \ln X$ is normally distributed with mean μ and variance σ^2. Equivalently, $X = e^Y$, for a normally distributed variable Y. Finding such a Y seems plausible in the context of search trees. Suppose that each search tree can be associated with a depth d, such that D^d = the number of nodes in the search tree, where D is the number of values per variables and thus the maximum branching factor of the search tree. For instance, d might be the average depth of a leaf dead-end. If d is a normally distributed random variable, then the lognormal distribution of nodes could be understood in terms of this d. For this line of reasoning to be interesting, d should be some well-defined property of the search tree. We have not found a d that meets the requirements, and so turn our attention to another source of the lognormal distribution.

The lognormal distribution can be derived from the "law of proportionate effect" [1]. This law says that if the growth rate of a variable at each step in a process is in random proportion to its size at that step, then the size of the variable at time n will be approximately lognormally distributed. In other words, if the value of a random variable at time i is Z_i, and if the relationship

$$Z_i = Z_{i-1} \times X_i \tag{3.32}$$

holds, where $(X_1, X_2, \ldots, X_n)$ are independent random variables, then the distribution of Z_i for large enough i is lognormally distributed. We formalize this notion as a corollary to the central limit theorem:

Corollary 1 (Law of Proportionate Effect) *Let $(X_1, X_2, \ldots)$ be a sequence of independent positive random variables as in Theorem 1 (restricted to $X_i > 0$). Let $(\hat{X}_1, \hat{X}_2, \ldots)$ be the natural logarithms of $(X_1, X_2, \ldots)$, with means $(\hat{\mu}_1, \hat{\mu}_2, \ldots)$, and variances $(\hat{\sigma}_1^2, \hat{\sigma}_2^2, \ldots)$. Let $a_n = X_1 \times \cdots \times X_n$, $\hat{\zeta}_n = \hat{\mu}_1 + \cdots + \hat{\mu}_n$, and $\hat{\tau}_n^2 = \hat{\sigma}_1^2 + \cdots + \hat{\sigma}_n^2$. Then*

$$\lim_{n\to\infty} P(a_n < y) = \Phi\left(\frac{\ln y - \hat{\zeta}_n}{\hat{\tau}_n}\right) \tag{3.33}$$

or in other words

$$\lim_{n\to\infty} a_n \textit{ is distributed as } \Lambda(\hat{\zeta}_n, \hat{\tau}_n^2). \tag{3.34}$$

Proof. We defined

$$a_n = X_1 \times \cdots \times X_n,$$

and taking the logarithm of both sides yields

$$\ln a_n = \ln X_1 + \cdots + \ln X_n.$$

Since the $\ln X_i$'s are independent random variables, by the central limit theorem

$$\lim_{n\to\infty} \ln a_n \text{ is distributed as } N(\hat{\zeta}_n, \hat{\tau}_n^2) \tag{3.35}$$

and by the definition of the lognormal distribution,

$$\lim_{n\to\infty} a_n \text{ is distributed as } \Lambda(\hat{\zeta}_n, \hat{\tau}_n^2). \tag{3.36}$$

Q.E.D.

We can summarize by stating that the law of proportionate effect is a multiplicative corollary of the central limit theorem. In the limit, the product of a series of independent and arbitrarily distributed random numbers is lognormally distributed. The only restriction is that the individual random numbers must be positive, since the lognormal probability density function is only defined for $X > 0$.

In the context of constraint satisfaction problems, we will now show that the number of nodes on each level of the search tree explored by backtracking is distributed approximately lognormally. We restrict our attention to the backtracking algorithm with a fixed random variable ordering (BT+RND), and to problems with no solution (so that the entire search tree is explored). We also assume that the problems are generated according to Model A. Thus the the probability that there is a constraint between any particular pair of variables is completely determined by C, and is independent of any constraints between other pairs of variables.

Our approach is to look at each level of the search tree explored by backtracking. Since the variable ordering is fixed, a level in the tree corresponds to a particular variable. The number of nodes on one level is the number of times the algorithm tried to instantiate that variable with one of its values. We show that this number of nodes per level is approximately lognormally distributed.

Let N_i be the number of nodes on level $i, 1 \leq i \leq n$, of the search tree, for a CSP instance with no solution. We define the branching factor b_i to be the ratio N_i/N_{i-1} for $2 \leq i \leq n$ and $b_1 = D$ (the size of the domain of the first variable). Note that b_i can be greater than or less than 1. The value of b_i depends on the constraints between X_{i-1} and the earlier variables. We can then write

$$N_k = b_1 \times b_2 \times \ldots \times b_k. \tag{3.37}$$

Each b_i, say, b_{10}, is a random variable which takes on one value per CSP instance. The distributions of the b_i's are related, since they all depend on the parameters $\langle N, D, C, T \rangle$ and the backtracking search algorithm. But each value of b_i for a particular search tree is independent of the others, because it depends on the number of constraints between X_{i-1} and the variables prior to X_{i-1} in the ordering, and in Model A this number is independent of the existence of constraints between other variables. Because the b_i's are independent random variables, an important condition of the law of proportionate effect is met. However, our analysis deviates from the law because it is possible for b_i to have the value of 0. In fact, because these CSPs have no solutions, it is inevitable that for each problem (search tree), some $b_j = 0$, and for all $k, k > j, b_k$ is undefined. Therefore we only see an approximate correspondence between backtracking search and the derivation of the lognormal distribution via the law of proportionate effect.

3.6.2 Deriving the Weibull

Two derivations of the Weibull distribution are common, first from the notion of an increasing or decreasing hazard rate, and second from the distribution of the smallest order statistic [53]. Both approaches have intuitive correspondences to backtracking search.

The notion of hazard rate has wide application in reliability studies. In actuarial statistics the same concept in known as the "force of mortality." In extreme value theory it is called the "intensity function." In CSP solving, we might call this rate the completion rate.

The hazard rate formalizes the notion that the probability of an event or failure may be conditioned on lifetime or waiting time. The hazard rate, $h(x)$, is defined as

$$h(x) = \frac{f(x)}{1 - F(x)}, \tag{3.38}$$

where $F(x)$ is a cumulative distribution function and $f(x)$ the associated probability density function. If a problem is not solved at time x, $h(x) \cdot \Delta x$ is the probability of completing the search in $(x, x + \Delta x)$. The Weibull distribution can be derived when the hazard rate is some power function of x [53]. The completion rate of the Weibull distribution is $h(x) = \lambda^{\beta} \beta x^{\beta - 1}$, which increases with x if $\beta > 1$

and decreases with x for $\beta < 1$. For the exponential distribution, a special case of the Weibull distribution with $\beta = 1$, $h(x) = \lambda$ is constant. Thus when $\beta < 1$, each node in the search tree has a smaller probability of being the last one than the one before it. In backtracking search on solvable problems, the effort required to find a solution is strongly influenced by the number of solutions and their distribution in the search space. As search continues without completion, the probability that there are many solutions decreases. This in turn increases the probability that the search will take a relatively large amount of effort. The decreasing completion rate of the Weibull distribution reflects the observation that easy solvable problems often have many solutions, and not finding a solution early in the search increases the estimate of how long the search will take.

The Weibull distribution can also be derived from the study of the smallest extreme. Let $(X_1, X_2, \ldots, X_n)$ be a random sample of n observations from distribution F. It is possible to compute the distribution of the minimum value in $(X_1, X_2, \ldots, X_n)$, called the first order statistic, as a function of F and n. For several types of function F, and as n becomes large, the distribution of the first order statistic is a Weibull distribution. In particular, the distribution of the first order statistic from a Weibull distribution is also a Weibull distribution.

Consider a backtracking search strategy in which all the subtrees rooted at level r in the search tree are searched in parallel. For instance, if $r = 4$ and D (the number of values per variable) $= 3$, then there are 3^4 such subtrees, although some subtrees may not exist because of dead-ends before level 4. Let S_i be the amount of work (e.g. search space or consistency checks) expended on the i'th subtree. If the problem has a solution, let S' be the amount of effort devoted to search in the first subtree which produces a solution. Then the total effort required (over all subtrees and over all processors) will be approximately $D^r \times S'$, less some time not spent on subtrees which finish without a solution after less than S' work. If we can approximate the distribution of finding a solution in a subtree with a Weibull distribution, then the distribution over the entire tree may also have a Weibull distribution.

3.7 Related Work

The primary factor which distinguishes our work from similar studies is that we focus not only on the expected effort required to solve CSPs, but also on the variance and distribution.

Haralick and Elliott [40] show how to compute the expected number of nodes and consistency checks for backtracking and forward checking, based on a model of random CSP similar to Model A.

Nudel [65] works with a model of CSP distribution in which instances are randomly chosen from the set of all problems that have a specified number of compatible value pairs, I_{ij}, for every pair of variables X_i and X_j.

Mitchell ([57]) shows results from a set of experiments in which the run time mean, standard deviation, and maximum value all increase as more and more samples are recorded. The finding is similar to that in Fig. 3.2, and is entirely consistent with the Weibull and lognormal distributions, as both tend to have long tails and high variance. Hogg and Williams ([41]) provide an analytical analysis of the exponentially long tail of CSP hardness distributions. Their work suggests that the distributions at the 50% satisfiable point are quite different than the distributions elsewhere in the parameter space. Selman and Kirkpatrick ([77]) have noted and analyzed the differing distributions of satisfiable and unsatisfiable instances. Kwan ([49]) has recently shown empirical evidence that the hardness of randomly generated CSPs and 3-coloring problems is not distributed normally.

3.8 Concluding remarks

More accurate summarization of experimental results was our initial motivation for investigating the distribution of CSP hardness. It remains primary, but several other benefits of this study are also worth highlighting. Well-developed statistical techniques, based on the assumption of a known underlying distribution, are available for estimating parameters based on data that have been "censored" above a certain point [64]. This may aid the interpretation of an experiment in which runs are terminated after a certain time point. In certain cases, it may be advantageous to design an experiment with a time-bound, knowing that the loss of accuracy in estimating the parameters of the distribution due to not having any data from the right tail is more than compensated for by the increased number of instances that can be run to completion.

Knowing the distribution will also enable a more precise comparison of competing algorithms. For instance, it is easier to determine whether the difference in the means of two experiments is statistically significant if the population distributions are known. Knowing that the distribution is *not* normal will prevent the researcher from relying on a statistical test that makes an assumption of normality.

Knowledge of the distribution function can be used in resource-limited situations to suggest an optimum time-bound for an algorithm to process a single instance. Examples would be running multiple algorithms on a single instance in a time-sliced manner, as proposed in [42], and environments where the goal is to complete as many problems as possible in a fixed time period.

The most important direction in which this line of research can be pursued is to determine whether the distribution of work required to solve real-world CSPs can also be usefully approximated by simple continuous distributions, perhaps the lognormal and the Weibull, or perhaps other distribution functions. In applications such as scheduling there are usually a large number of CSPs to solve which can be thought of as coming from an (unknown) distribution. The distribution of solving-time for these problems may also display useful regularities.

Chapter 4

Backjumping and Dynamic Variable Ordering

4.1 Overview of Chapter

We propose an algorithm, dubbed BJ+DVO, which combines conflict-based backjumping and a dynamic variable ordering heuristic which incorporates forward checking style filtering of future domains[1]. Experimental evaluation shows that BJ+DVO is effective on random problem instances created with a wide range of parameters.

4.2 Introduction

In Chapter 2 we discussed two standard constraint satisfaction algorithms, conflict-based backjumping (BJ) and forward checking (FC). Although the two algorithms are both based on backtracking, they take quite different approaches to identifying and rejecting instantiations which conflict with a constraint. In this chapter we develop an algorithm that combines backjumping with a dynamic variable ordering heuristic which is based on information acquired by a forward checking style processing of future variables.

[1] BJ+DVO was first described and evaluated in Frost and Dechter [27].

Backjumping can be termed a "look-back" algorithm. Like backtracking, backjumping rejects a potential value for a variable if it is incompatible with the current set of instantiated variables. Backjumping goes beyond backtracking in sophistication because it is able to jump back over variables that are not responsible for a dead-end.

Forward checking uses a "look-ahead" approach. After a value v is assigned to a variable X, all values in the domains of future variables that are incompatible with $X = v$ are removed while $X = v$. Forward checking does not need to test whether a value in the domain of the current variable is compatible with the previous partial instantiation, since incompatible values were temporarily filtered out when the earlier variables were assigned values.

Because backjumping and forward checking use two orthogonal methods for improving CSP search, it is natural to wonder whether it makes sense to combine the two approaches. Worst-case analyses such as that of Kondrak and van Beek [47] offer no answer, since we do not expect a combination algorithm to have better worst-case performance.

Prosser [68] performed a small experimental study, limited to random permutations of a single problem, with several algorithms including "BJ+FC", which combines backjumping and forward checking but uses a fixed variable ordering. The new element in algorithm BJ+DVO is that it combines, in effect, BJ+FC with dynamic variable ordering. In addition, we present an extensive and systematic evaluation on large instances, which was not done before. Our empirical evidence demonstrates that the combination of backjumping, forward checking, and dynamic variable ordering is indeed a strong performer on a wide variety of CSPs.

Backjumping with DVO

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$. Set $P_i \leftarrow \emptyset$ for $1 \leq i \leq n$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable, selected according to a VARIABLE-ORDERING-HEURISTIC (see Fig. 4.2). Set $P_{cur} \leftarrow \emptyset$.
2. Select a value $x \in D'_{cur}$. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.
   (c) Examine the future variables $X_i, cur < i \leq n$. For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i and add X_{cur} to P_i; if D'_i becomes empty, go to (d) (without examining other X_i's).
   (d) Go to 1.
3. (Backjump.) If $P_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent." Otherwise, set $P \leftarrow P_{cur}$; set cur equal to the index of the last variable in P. Set $P_{cur} \leftarrow P_{cur} \cup P - \{X_{cur}\}$. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 4.1: The BJ+DVO algorithm.

4.3 The BJ+DVO Algorithm

This chapter reports an algorithm which is designed to combine the desirable features of backjumping, forward checking, and dynamic variable ordering. We call this combination BJ+DVO, and it is described in Fig. 4.1.

Step 1 of BJ+DVO utilizes a variable ordering heuristic, which selects an uninstantiated variable to be the next in the ordering. The variable ordering heuristic we use is described in Fig. 4.2 and is discussed below.

Step 2 of BJ+DVO is patterned after the forward checking algorithm in Section 2.8, with two changes. The first change is that conflicts are recorded using parent sets P_i. The parent sets have the same function as in backjumping: P_i is a set of variables X_k, each of which is the last in a partial instantiation

$\vec{x}_k$ which conflicts with some value of X_i. The presence of a forward checking style look-ahead mechanism affects how the P's are updated. In backjumping, earlier variables are added to P_{cur}. In BJ+DVO, X_{cur} is added to the parent set of a future variable. This reflects the fact that pure look-back algorithms (such as backjumping) compare the current variable with earlier variables, while an algorithm having a look-ahead component, such as BJ+DVO, removes values from the domains of future variables.

The other change to step 2 marks a departure from all the earlier algorithms in Chapter 2. In the earlier algorithms, if a value x from D'_{cur} was found incompatible, either by a look back or a look forward check with other variables, there was a "go to (a)" step which continued the search with the next value of the current variable. In contrast, BJ+DVO's step 2 always proceeds to step 1 after assigning a value to X_{cur}. If that value causes the domain of a future variable X_e to become empty, then the variable ordering heuristic will select X_e to be the next variable, and after step 2 (a) (for X_e with an empty domain) is executed, control will go the backjump step, step 3. The backjump from X_e will of course be a step back to the immediately preceeding variable, since it was the instantiation of that variable which caused X_e to have an empty domain. Thus in BJ+DVO, a backjump from a leaf dead-end is always to the immediately preceeding variable. The fact that selecting to be next a variable with an empty domain makes Gaschnig's backjumping – that is, backjumping from leaves in the search tree – redundant was noted in [3] and [47].

In step 2 (b), the values of each variable are considered in an arbitrary but fixed order.

Step 3 of BJ+DVO is identical to the step 3 of the backjumping algorithm from chapter 2, section 6. The most recent variable in the P_{cur} set is identified, and that variable becomes current, with its parent set being merged with the parent set of the dead-end variable.

VARIABLE-ORDING-HEURISTIC

1. If no variable have yet been selected, select the variable that participates in the most constraints. In case of a tie, select one variable arbitrarily.
2. Let m be the size of the smallest D' set of a future variable.
   (a) If there is one future variable with D' size $= m$, then select it.
   (b) If there is more than one, select the one that participates in the most constraints (in the original problem), breaking any remaining ties arbitrarily.

Figure 4.2: The variable ordering heuristic used by BJ+DVO.

The main idea of the variable ordering heuristic, described in Fig. 4.2, is to select the future variable with the smallest remaining domain. As noted in Chapter 2, this idea was proposed by Haralick and Elliot [40] under the rubric "fail first." We have found that augmenting the fail-first strategy with the tie-breaking rules described in step 1 and step 2 (b) of Fig. 4.2 produces a 10% to 30% improvement in performance, when compared to BJ+DVO without the tie-breakers (data not shown). The guiding intuition behind the tie-breakers is to select the variable that is the most constraining, and thus most likely to reduce the size of the D' sets of those variables selected after it.

4.4 Experimental Evaluation

The goal of the experiments was to determine whether combining backjumping, forward-checking style look-ahead, and dynamic variable ordering into a single procedure would be an effective combination. Six algorithms or combination algorithms were employed: BT+MW (simple backtracking with the min-width variable ordering heuristic), BJ+MW (conflict-directed backjumping with min-width), FC+MW (forward checking with min-width), BJ+FC+MW (backjumping with forward checking and min-width), BT+DVO (backtracking with dynamic variable ordering), and BJ+DVO (backjumping with dynamic variable ordering). We selected a variety of parameters for the random problem generator, and for each set of parameters generated 2,000 problem instances. The experiments show the relative performance of the six algorithms over a range of values for D, T, and C, and for increasing N. For selected value of D and T, we generated instances with several values of N and C. Because the parameters are all near the cross-over point, there are about 1,000 solvable and unsolvable instances for each set. Tables 4.1, 4.2, 4.3, and 4.4 report mean CPU seconds (on a SparcStation 4, 110 MHz processor) for each experiment, with unsolvable and solvable problems reported separately. CPU seconds are rounded to two decimal points; reported values of "0.00" indicate that the average was less than 0.005. A position in the tables is in parentheses when the experiment was stopped part way through; "n.r." indicates that the experiment was not run.

For convenience, experiments with $D = 3$ are reported in Table 4.1 and 4.2, while experiments with $D = 6$ are summarized in Table 4.3 and 4.4. Within each table, sets of experiments with different values of parameter T are divided by horizontal lines.

The first four columns of Tables 4.1 through 4.4 show the results of using algorithms which have a fixed variable ordering. In all cases the combination BT+FC+MW performed best, and simple backtracking, BT+MW, least well. The relative performance of BJ+MW and FC+MW depended on the domain size parameter D. On problems with D=3, BJ+MW solved problems using less CPU time and FC+MW, while FC+MW was better with D=6.

The experiments in general support the conclusion that the BJ+DVO combination is an extremely effective one. Its only serious rival, among the algorithms we tested, is BT+DVO when parameter T is relatively small, that is, when the constraints are loose. With D=3 and T=.111, BJ+DVO is marginally worse than BT+DVO on unsolvable problems, and marginally better on solvable problems

D=3	Mean CPU seconds (1,000 unsolvable instances)					
Parameters	+MW				+DVO	
	BT	BJ	FC	BJ+FC	BT	BJ
$\langle 25, 3, .6633, .111\rangle$	0.11	0.02	0.03	0.02	0.00	0.01
$\langle 50, 3, .3118, .111\rangle$	(17.72)	0.68	2.20	0.53	0.03	0.04
$\langle 75, 3, .2032, .111\rangle$	n.r.	16.47	(157.33)	9.78	0.17	0.19
$\langle 100, 3, .1509, .111\rangle$	n.r.	n.r.	n.r.	n.r.	0.73	0.77
$\langle 125, 3, .1199, .111\rangle$	n.r.	n.r.	n.r.	n.r.	2.69	2.82
$\langle 150, 3, .0995, .111\rangle$	n.r.	n.r.	n.r.	n.r.	9.40	9.74
$\langle 175, 3, .0850, .111\rangle$	n.r.	n.r.	n.r.	n.r.	33.81	34.37
$\langle 200, 3, .0742, .111\rangle$	n.r.	n.r.	n.r.	n.r.	104.72	106.15
$\langle 25, 3, .2967, .222\rangle$	0.02	0.01	0.01	0.01	0.00	0.00
$\langle 50, 3, .1355, .222\rangle$	8.13	0.09	0.47	0.09	0.01	0.01
$\langle 75, 3, .0872, .222\rangle$	n.r.	0.75	(40.03)	0.61	0.05	0.04
$\langle 100, 3, .0790, .222\rangle$	n.r.	7.11	n.r.	4.10	0.24	0.12
$\langle 150, 3, .0421, .222\rangle$	n.r.	(93.80)	n.r.	n.r.	(49.32)	0.86
$\langle 175, 3, .0358, .222\rangle$	n.r.	n.r.	n.r.	n.r.	n.r.	2.10
$\langle 200, 3, .0313, .222\rangle$	n.r.	n.r.	n.r.	n.r.	n.r.	4.35
$\langle 250, 3, .0249, .222\rangle$	n.r.	n.r.	n.r.	n.r.	n.r.	44.72
$\langle 50, 3, .0751, .333\rangle$	(22.06)	0.02	0.44	0.02	0.01	0.00
$\langle 75, 3, .0476, .333\rangle$	n.r.	0.09	(49.35)	0.09	0.04	0.02
$\langle 100, 3, .0343, .333\rangle$	n.r.	0.38	n.r.	0.29	18.84	0.05
$\langle 125, 3, .0267, .333\rangle$	n.r.	1.05	n.r.	0.79	(60.50)	0.12
$\langle 150, 3, .0218, .333\rangle$	n.r.	3.15	n.r.	2.21	n.r.	0.35
$\langle 175, 3, .0185, .333\rangle$	n.r.	7.65	n.r.	4.69	n.r.	0.43
$\langle 200, 3, .0160, .333\rangle$	n.r.	n.r.	n.r.	(21.83)	n.r.	0.86
$\langle 300, 3, .0105, .333\rangle$	n.r.	n.r.	n.r.	n.r.	n.r.	18.81

Table 4.1: Comparison of six algorithms on unsolvable instances generated with parameter D=3. Parentheses indication partial completion; n.r. means not run.

D=3	Mean CPU seconds (1,000 solvable instances)					
Parameters	+MW				+DVO	
	BT	BJ	FC	BJ+FC	BT	BJ
$\langle 25, 3, .6633, .111 \rangle$	0.05	0.01	0.02	0.01	0.00	0.00
$\langle 50, 3, .3118, .111 \rangle$	(9.08)	0.33	1.43	0.26	0.02	0.02
$\langle 75, 3, .2032, .111 \rangle$	n.r.	8.20	(129.87)	5.40	0.09	0.09
$\langle 100, 3, .1509, .111 \rangle$	n.r.	n.r.	n.r.	n.r.	0.33	0.35
$\langle 125, 3, .1199, .111 \rangle$	n.r.	n.r.	n.r.	n.r.	1.22	1.26
$\langle 150, 3, .0995, .111 \rangle$	n.r.	n.r.	n.r.	n.r.	4.34	4.42
$\langle 175, 3, .0850, .111 \rangle$	n.r.	n.r.	n.r.	n.r.	15.99	15.91
$\langle 200, 3, .0742, .111 \rangle$	n.r.	n.r.	n.r.	n.r.	46.29	45.32
$\langle 25, 3, .2967, .222 \rangle$	0.01	0.00	0.00	0.01	0.00	0.00
$\langle 50, 3, .1355, .222 \rangle$	6.26	0.05	0.39	0.05	0.01	0.01
$\langle 75, 3, .0872, .222 \rangle$	n.r.	0.41	31.76	0.34	0.03	0.03
$\langle 100, 3, .0790, .222 \rangle$	n.r.	3.83	(482.79)	n.r.	0.32	0.08
$\langle 150, 3, .0421, .222 \rangle$	n.r.	(58.75)	n.r.	n.r.	13.28	0.68
$\langle 175, 3, .0358, .222 \rangle$	n.r.	n.r.	n.r.	n.r.	25.67	1.37
$\langle 200, 3, .0313, .222 \rangle$	n.r.	n.r.	n.r.	n.r.	(38.14)	3.44
$\langle 250, 3, .0249, .222 \rangle$	n.r.	n.r.	n.r.	n.r.	n.r.	32.01
$\langle 50, 3, .0751, .333 \rangle$	(9.46)	0.01	0.29	0.02	0.01	0.01
$\langle 75, 3, .0476, .333 \rangle$	n.r.	0.06	(32.07)	0.07	0.03	0.01
$\langle 100, 3, .0343, .333 \rangle$	n.r.	0.17	n.r.	0.18	0.26	0.03
$\langle 125, 3, .0267, .333 \rangle$	n.r.	0.54	n.r.	0.50	(16.91)	0.05
$\langle 150, 3, .0218, .333 \rangle$	n.r.	1.21	n.r.	1.07	n.r.	0.11
$\langle 175, 3, .0185, .333 \rangle$	n.r.	4.17	n.r.	2.29	n.r.	0.17
$\langle 200, 3, .0160, .333 \rangle$	n.r.	n.r.	n.r.	(6.38)	n.r.	0.28
$\langle 300, 3, .0105, .333 \rangle$	n.r.	n.r.	n.r.	n.r.	n.r.	7.14

Table 4.2: Comparison of six algorithms on solvable instances generated with parameter D=3. Parentheses indication partial completion; n.r. means not run.

D=6	Mean CPU seconds (1,000 unsolvable instances)					
Parameters	+MW				+DVO	
	BT	BJ	FC	BJ+FC	BT	BJ
$\langle 35, 6, .8420, .111\rangle$	(1,189)	(167.00)	(52.18)	42.12	1.34	1.46
$\langle 40, 6, .7308, .111\rangle$	n.r.	n.r.	n.r.	n.r.	3.17	3.45
$\langle 50, 6, .5796, .111\rangle$	n.r.	n.r.	n.r.	n.r.	16.00	17.56
$\langle 60, 6, .4797, .111\rangle$	n.r.	n.r.	n.r.	n.r.	81.07	88.22
$\langle 25, 6, .5533, .222\rangle$	1.44	0.44	0.31	0.29	0.05	0.06
$\langle 40, 6, .3346, .222\rangle$	(61.78)	13.87	8.78	6.38	0.51	0.56
$\langle 50, 6, .2653, .222\rangle$	n.r.	n.r.	75.05	45.00	2.00	2.20
$\langle 60, 6, .2192, .222\rangle$	n.r.	n.r.	n.r.	n.r.	7.53	8.17
$\langle 75, 6, .1744, .222\rangle$	n.r.	n.r.	n.r.	n.r.	54.71	58.92
$\langle 25, 6, .3333, .333\rangle$	0.30	0.10	0.08	0.08	0.02	0.02
$\langle 40, 6, .2000, .333\rangle$	(103.66)	1.75	1.28	0.91	0.14	0.15
$\langle 50, 6, .1576, .333\rangle$	n.r.	n.r.	7.11	4.10	0.44	0.46
$\langle 75, 6, .1038, .333\rangle$	n.r.	n.r.	n.r.	(32.38)	6.77	6.85
$\langle 100, 6, .0772, .333\rangle$	n.r.	n.r.	n.r.	n.r.	(91.47)	85.96
$\langle 25, 6, .2200, .444\rangle$	0.09	0.03	0.03	0.03	0.01	0.01
$\langle 50, 6, .1029, .444\rangle$	(10.70)	1.44	1.10	0.57	0.15	0.13
$\langle 60, 6, .0847, .444\rangle$	n.r.	5.35	4.55	1.81	0.41	0.33
$\langle 75, 6, .0670, .444\rangle$	n.r.	n.r.	n.r.	10.48	1.67	1.17
$\langle 100, 6, .0497, .444\rangle$	n.r.	n.r.	n.r.	n.r.	21.19	10.18
$\langle 50, 6, .0678, .555\rangle$	(9.36)	0.42	0.19	0.18	0.14	0.04
$\langle 75, 6, .0432, .555\rangle$	n.r.	3.41	2.54	2.01	49.06	0.32
$\langle 100, 6, .0319, .555\rangle$	n.r.	(46.21)	32.27	17.83	n.r.	5.59

Table 4.3: Comparison of six algorithms on unsolvable instances generated with parameter D=6. Parentheses indication partial completion; n.r. means not run.

D=6	Mean CPU seconds (1,000 solvable instances)					
Parameters	+MW				+DVO	
	BT	BJ	FC	BJ+FC	BT	BJ
$\langle 35, 6, .8420, .111\rangle$	(498.72)	(68.07)	(23.17)	17.23	0.58	0.63
$\langle 40, 6, .7308, .111\rangle$	n.r.	n.r.	n.r.	n.r.	1.25	1.36
$\langle 50, 6, .5796, .111\rangle$	n.r.	n.r.	n.r.	n.r.	6.67	7.31
$\langle 60, 6, .4797, .111\rangle$	n.r.	n.r.	n.r.	n.r.	33.92	36.88
$\langle 25, 6, .5533, .222\rangle$	0.69	0.20	0.15	0.14	0.03	0.03
$\langle 40, 6, .3346, .222\rangle$	(19.40)	7.40	4.85	3.36	0.25	0.27
$\langle 50, 6, .2653, .222\rangle$	n.r.	n.r.	40.55	23.10	0.96	1.05
$\langle 60, 6, .2192, .222\rangle$	n.r.	n.r.	n.r.	n.r.	3.75	4.05
$\langle 75, 6, .1744, .222\rangle$	n.r.	n.r.	n.r.	n.r.	27.81	29.87
$\langle 25, 6, .3333, .333\rangle$	0.17	0.05	0.05	0.04	0.01	0.01
$\langle 40, 6, .2000, .333\rangle$	(41.48)	1.06	0.87	0.58	0.08	0.09
$\langle 50, 6, .1576, .333\rangle$	n.r.	n.r.	5.11	2.62	0.25	0.27
$\langle 75, 6, .1038, .333\rangle$	n.r.	n.r.	n.r.	(12.56)	4.19	4.18
$\langle 100, 6, .0772, .333\rangle$	n.r.	n.r.	n.r.	n.r.	(61.99)	57.51
$\langle 25, 6, .2200, .444\rangle$	0.06	0.02	0.02	0.02	0.01	0.01
$\langle 50, 6, .1029, .444\rangle$	n.r.	1.20	1.25	0.49	0.11	0.09
$\langle 60, 6, .0847, .444\rangle$	n.r.	4.56	8.96	1.78	0.33	0.25
$\langle 75, 6, .0670, .444\rangle$	n.r.	n.r.	n.r.	7.99	1.99	1.15
$\langle 100, 6, .0497, .444\rangle$	n.r.	n.r.	n.r.	n.r.	23.36	7.91
$\langle 50, 6, .0678, .555\rangle$	(14.30)	0.31	0.18	0.16	0.12	0.04
$\langle 75, 6, .0432, .555\rangle$	n.r.	1.75	1.86	1.79	165.19	0.28
$\langle 100, 6, .0319, .555\rangle$	n.r.	(31.08)	39.47	13.29	n.r.	2.08

Table 4.4: Comparison of six algorithms on solvable instances generated with parameter D=6. Parentheses indication partial completion; n.r. means not run.

(see Table 4.1 and Table 4.2). When T is greater than .111, BJ+DVO substantially outperforms BT+DVO. With D=6, the dividing line is near T = .333. At this and smaller values of T, BJ+DVO underperforms BT+DVO, although by no more, on average, than 10%. When D=6 and T is greater than .333, BJ+DVO is substantially better than BT+DVO.

CPU seconds is the basis for comparing algorithms in Tables 4.1–4.4. Tables 4.5 and 4.6 are based on the same experiments as the earlier tables, but show mean consistency checks and mean nodes in search space for selected sets of parameters and for the algorithms BT+DVO and BJ+DVO only. In several cases where BJ+DVO requires more CPU time, on average, than BT+DVO, it makes fewer consistency checks and expands a slightly smaller search space. For instance, in the experiment with parameters $\langle 150, 3, .0995, .111\rangle$ and unsolvable problems (see the first line in Table 4.5), BT+DVO makes 7% more consistency checks than BJ+DVO (1,048,156 compared to 981,043) and searches 7% more nodes (37,171 compared to 34,674), yet on average requires only 97% as much CPU time. This pattern holds not only for the average, but also on an instance by instance basis. Over three fourths of the unsolvable problems with these parameters made between 1% and 15% more consistency checks with BT+DVO than with BJ+DVO, but finished in 85% to 99% as much CPU time.

On large problems (N=175 and N=200) generated with D=3 and T=.111, BJ+DVO is slightly worse than BT+DVO on unsolvable problems and slightly better than BT+DVO on solvable problems (when measuring CPU time). Unsolvable branches of solvable problems tend to be somewhat deeper than the average branch of an unsolvable problem with the same parameters. Thus solvable problems frequently have deeper search trees that unsolvable problems, and the more time the search spends deep in the search tree, the more opportunity exists for backjumping to be useful.

It can be interesting to examine not only the mean performance of algorithms, but the entire distribution of computational effort over a set of problem instances.

Parameters	Mean values (1,000 unsolvable instances)					
	BT+DVO			BJ+DVO		
	CC	Nodes	CPU	CC	Nodes	CPU
$\langle 150, 3, .0995, .111\rangle$	1,058,173	37,484	9.40	991,163	35,003	9.74
$\langle 175, 3, .0850, .111\rangle$	3,095,396	109,124	33.81	2,873,370	100,981	34.37
$\langle 200, 3, .0742, .111\rangle$	8,735,964	306,628	104.72	8,043,054	281,559	106.15
$\langle 100, 3, .0790, .222\rangle$	11,007	1,002	0.24	8,775	679	0.12
$\langle 150, 3, .0421, .222\rangle$	570,635	189,997	49.32	38,856	3,099	0.86
$\langle 175, 3, .0358, .222\rangle$				81,473	6,697	2.10
$\langle 200, 3, .0313, .222\rangle$				150,959	12,346	4.53
$\langle 100, 3, .0343, .333\rangle$	542,233	129,338	18.84	1,943	257	0.05
$\langle 125, 3, .0267, .333\rangle$	955,740	294,718	60.50	3,800	524	0.12
$\langle 150, 3, .0218, .333\rangle$				8,602	1,293	0.35
$\langle 200, 3, .0160, .333\rangle$				32,763	7,353	2.00
$\langle 50, 6, .5796, .111\rangle$	7,677,173	130,284	16.00	7,615,125	129,061	17.56
$\langle 60, 6, .4797, .111\rangle$	33,521,936	562,191	81.07	33,154,152	555,195	88.22

Table 4.5: Comparison of BT+DVO and BJ+DVO on unsolvable problems, measuring consistency checks, nodes in the search space, and CPU seconds.

Parameters	Mean values (1,000 solvable instances)					
	BT+DVO			BJ+DVO		
	CC	Nodes	CPU	CC	Nodes	CPU
$\langle 150, 3, .0995, .111\rangle$	477,156	17,808	4.34	441,225	16,387	4.42
$\langle 175, 3, .0850, .111\rangle$	1,425,292	52,737	15.99	1,300,459	47,932	15.91
$\langle 200, 3, .0742, .111\rangle$	3,786,347	138,251	46.29	3,378,094	122,824	45.32
$\langle 100, 3, .0790, .222\rangle$	10,253	1,744	0.32	5,267	488	0.08
$\langle 150, 3, .0421, .222\rangle$	214,932	48,817	13.28	28,526	2,740	0.68
$\langle 175, 3, .0358, .222\rangle$	346,607	79,455	25.67	48,208	4,368	1.37
$\langle 200, 3, .0313, .222\rangle$				112,886	10,309	3.63
$\langle 100, 3, .0343, .333\rangle$	6,882	1,745	0.26	1,609	276	0.05
$\langle 125, 3, .0267, .333\rangle$	279,194	79,311	16.91	1,609	276	0.05
$\langle 150, 3, .0218, .333\rangle$				2,727	449	0.11
$\langle 200, 3, .0160, .333\rangle$				40,426	7,148	2.29
$\langle 50, 6, .5796, .111\rangle$	3,171,321	54,924	6.67	3,143,167	54,353	7.31
$\langle 60, 6, .4797, .111\rangle$	13,908,893	237,554	33.92	13,745,036	234,363	36.88

Table 4.6: Comparison of BT+DVO and BJ+DVO on solvable problems, measuring consistency checks, nodes in the search space, and CPU seconds.

Taking advantage of the results from Chapter 3, we estimate the parameters of the lognormal distributions that most closely approximate the empirical distributions on unsolvable instances generated with $\langle 100, 3, 0.0343, 0.333\rangle$ and searched by algorithms BJ+MW, BJ+FC+MW, BT+DVO, and BJ+DVO. In Fig. 4.3 we graph the lognormal distributions, measuring both the number of consistency checks made (top chart) and the number of CPU seconds required (bottom chart). The estimated μ and σ parameters are specified inside the chart. Comparing the resulting curves, we see that BT+DVO and BJ+DVO solve the most problems with a small amount of effort – the distributions for these two algorithms have high modes far to the left. However, BT+DVO required a very large amount of effort on a few instances. The mean CPU time and consistency checks for BT+DVO is therefore higher than for the other algorithms, and its σ parameter is the highest, indicating a heavy right tail and a relatively large number of hard instances.

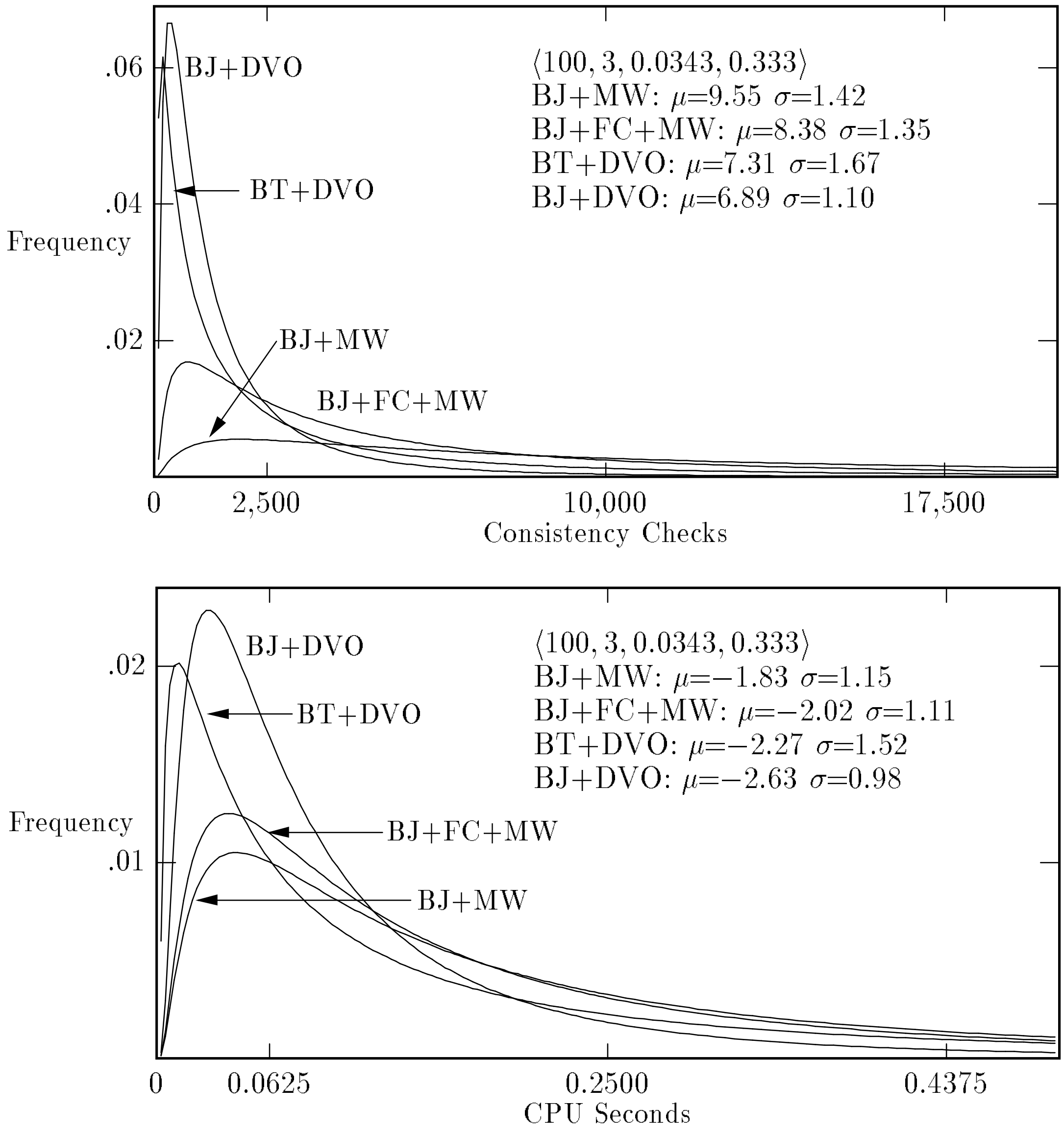


Figure 4.3: Lognormal curves based on unsolvable problems generated from parameters $\langle 100, 3, 0.0343, 0.333\rangle$. The top chart is based on consistency checks, the bottom chart on CPU seconds. μ and σ parameters were estimated using the Maximum Likelihood Estimator (see Chapter 3).

Data from some experiments reported in Table 4.1 is plotted in Fig. 4.4. The plots show that for this class of problems the performance of BJ+DVO scales up better than that of BT+DVO. In this figure the experimental results are summarized by the μ and σ parameters of the lognormal distribution. The maximum likelihood estimator described in Chapter 3 was used to determine μ and σ. In addition to plotting the data points for various parameter combinations, we also show the least square regression line. For both algorithms BT+DVO and BJ+DVO, a clear linear relationship between N and each parameter of the lognormal distribution is evident. The correlation coefficient is 0.989 or above in all four cases. We have observed a similar linear relationship with many other algorithms and sets of parameters for the problem generator, and also for solvable problems. A similar observation has been made by Crawford and Auton [12] concerning the growth of 3-SAT problems at the cross-over point. They found the logarithm of the average search space size linearly related to the number of variables with a factor of approximately 0.04.

In Chapter 3 we noted that in a lognormal distribution, the mean is $\exp(\mu + \sigma^2/2)$, the median is $\exp(\mu)$, and the variance is $\exp(2\mu + \sigma^2)(\exp(\sigma^2) - 1)$. If we extrapolate that an approximately linear relationship between μ and N and between σ and N holds at larger values of N, it is clear that the performance gap between BJ+DVO and BT+DVO will grow more pronounced on larger and larger problems. Linear growth in μ and σ corresponds to exponential growth in the mean and median problem difficulty, for the distribution of CSPs defined by the Model B generator. Not only are the mean and median smaller for BJ+DVO than for BT+DVO, but so is the variance, which is particularly sensitive to σ. The hardest problems for BT+DVO tend to be impacted the most by adding backjumping to the algorithm, since they have the largest search spaces and thus offer the most opportunities for large jumpbacks.

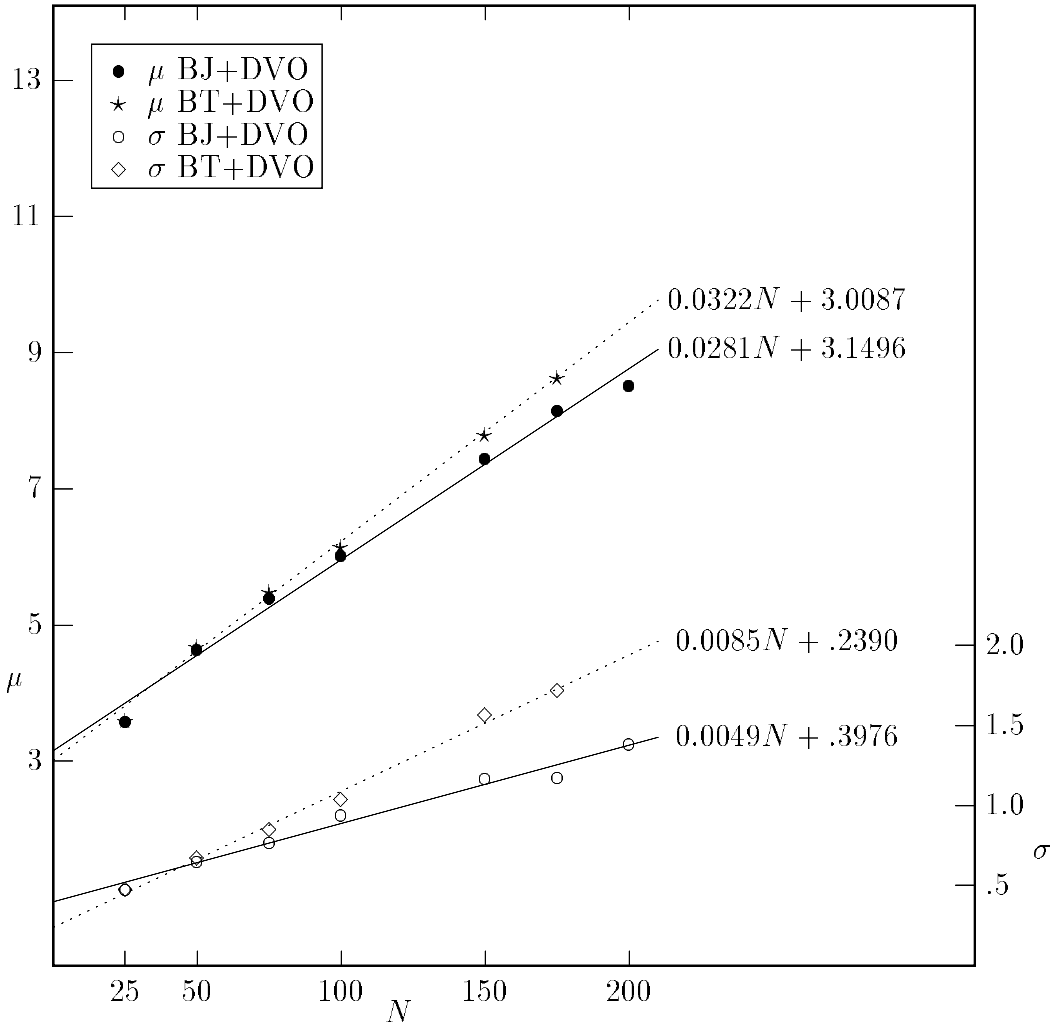


Figure 4.4: Data on search space size of unsolvable problems in experiments with parameters D=3, T=.222, and varying values of N and C (as in Table 4.1), using algorithms BT+DVO and BJ+DVO. Points represent estimated μ (left hand scale) and σ (right hand scale) for each algorithm, assuming a lognormal distribution. Lines (solid for BJ+DVO, dotted for BT+DVO) show best linear fit. The formula to the right of each line shows the slope and the y-axis intercept.

Parameters	Mean CPU seconds BT+DVO	BJ+DVO	Mean EJ ratio	Mean per 10,000 EJ ≥ 5	EJ ≥ 10
$\langle 175, 3, .0850, .111 \rangle$	33.81	34.37	0.61	225	54
$\langle 175, 3, .0358, .222 \rangle$	325.77	2.30	0.80	391	132
$\langle 175, 3, .0185, .333 \rangle$		0.43	1.16	651	271
$\langle 50, 6, .5796, .111 \rangle$	15.74	17.30	0.15	6	0
$\langle 60, 6, .2192, .222 \rangle$	7.60	12.41	0.19	13	0
$\langle 75, 6, .1038, .333 \rangle$	5.96	6.05	0.25	31	2
$\langle 100, 6, .0497, .444 \rangle$	21.19	10.18	0.39	94	15
$\langle 100, 6, .0319, .555 \rangle$		5.59	0.56	215	70

Table 4.7: Extract of data from Table 4.1 and Table 4.3, plus record of extra jumps made by backjumping. Unsolvable instances only.

4.5 Discussion

Why does the relative performance of BT+DVO and BJ+DVO seem to depend on the tightness of the constraints? The answer lies in the costs and benefits of backjumping. The overhead of backjumping, primarily maintaining and checking the P_i sets, does not seem to pay off, on average, on problems with relatively loose constraints. With tighter constraints, the average size of the parent set tends to shrink, because it is more likely that a single variable in the parent set is in conflict with multiple values of the dead-end variable. A smaller parent set increases the likelihood that backjumping will skip over a large number of variables in its jump. To assess this explanation quantitatively, we can examine more closely the behavior of backjumping.

During the execution of the BJ+DVO algorithm measurements were made pertaining to the effectiveness of backjumping; they are reported in Table 4.7. The measurements were made during a second, instrumented, run of the algorithm, so that the CPU times cited in the tables would not be affected by the overhead of recording additional information. Our goal was to discover how much jumping BJ+DVO was doing. We recorded, for each problem instance, the total number of "extra jumps" made, defining an extra jump as returning to a variable other than the immediately preceeding one. For instance, if after a dead-end at X_{20} the highest variable in P_{20} (the parent set of X_{20}) is X_{17}, this counts as 2 extra jumps. Dividing the total number of extra jumps by the number of interior dead-ends yields the "extra jump ratio" for an instance; the average for each set of parameters is reported in the "EJ ratio" column of Table 4.7. Leaf dead-ends were excluded from this calculation because, as discussed in section 4.3, the backjump from a leaf dead-end is always a single step to the immediately preceeding variable (no extra jumps) when DVO is in effect. The last two columns of Table 4.7 show the average number of large backjumps, size five or greater and size ten or greater, per 10,000 interior dead-ends.

It is not surprising to observe in Table 4.7 that as T increases and BJ+DVO becomes more effective than BT+DVO, the quantity of extra jumps increases. It is perhaps surprising to see how large an impact a seemingly small number of extra jumps can make. For instance, in the set of problems with parameters $\langle 175, 3, .0358, .222\rangle$ BJ+DVO requires half a percent as much CPU time as does BT+DVO, the result of jumping back on average slightly less than one extra variable per interior dead-end. In this particular set of unsolvable problems, one instance required 23.6 CPU hours with BT+DVO and only 2 CPU seconds with BJ+DVO (excluding this problem instance, the means would be 26.17 CPU seconds under BT+DVO and an unchanged 2.10 CPU seconds under BJ+DVO). BJ+DVO made 3,076 extra jumps on this instance and had 2,523 interior dead-ends, for an extra jump ratio of 1.22, somewhat higher than average. On this instance BJ+DVO once jumped over 88 variables, and had 59 jumps of 10 or more variables.

The combination of DVO and backjumping is particularly felicitous because of the complementary strengths of the backjumping and dynamic variable ordering components. Backjumping is more effective on sparser constraint graphs (low value of parameter C), since the average size of each "jump" tends to increase with increasing sparseness. The dynamic variable ordering heuristic, in contrast, tends

Parameters	single value domain
$\langle 75, 3, .2032, .111\rangle$	92.8%
$\langle 75, 3, .0872, .222\rangle$	78.9%
$\langle 75, 3, .0476, .333\rangle$	70.9%
$\langle 40, 6, .3346, .222\rangle$	80.2%
$\langle 40, 6, .2000, .333\rangle$	77.1%
$\langle 40, 6, .1308, .444\rangle$	72.4%

Table 4.8: Data on unsolvable problems with N=75, D=3 (first three lines), drawn from the same experiments as in Table 4.1, and data with N=40, D=6 (second three lines), drawn from the same experiments as in Table 4.3. The algorithm is BT+DVO, and the "single value domain" column reports the frequency in non-dead-end situations that there was a future variable with exactly one value in its domain.

to function better when there are many constraints (high value of parameter C), since each constraint provides information it can utilize in deciding on the next variable. We assessed this observation quantitatively by recording the performance of BT+DVO over a variety of values of C, while holding the number of variables N and the domain size D constant. Specifically, we measured the frequency with which BT+DVO found the size of the smallest future domain to be one. This is the situation where DVO can most effectively prune the search space. See Table 4.8, where the column labelled "single value domain" shows how often DVO found a variable with one remaining consistent value, in those cases where there wasn't an empty future domain. The decreasing frequency of single-valued variables as the constraint graph becomes sparse indicates that on those problems DVO has to make a less-informed choice about the variable to choose next.

4.6 Conclusions

We have introduced a new algorithm for solving CSPs called BJ+DVO, which combines three different techniques: backjumping, forward checking, and dynamic variable ordering. Our experimental evaluation shows that each of BJ+DVO's three constituent parts plays an important role in the overall performance of the algorithm, and that BJ+DVO is substantially better than any algorithm that uses just one or two of its constituents. BJ+DVO is a clear winner over the next best algorithm when the constraints are relatively tight, and is only slightly worse than BT+DVO on problems with loose constraints, such as T=.111. On these problems, BJ+DVO makes fewer consistency checks and explores a smaller search space than does BT+DVO, but still uses a small amount of additional CPU time, due to the cost of maintaining the tables for jumpback.

Chapter 5

Interleaving Arc-consistency

5.1 Overview of Chapter

Many techniques have been proposed for interleaving search and constraint propagation. In this chapter we compare four different schemes that do some amount of constraint propagation after each instantiation. Our experiments show that forward checking, which does the least amount constraint propagation at each step, is best on CSPs with many relatively loose constraints, while using the strongest propagation, arc-consistency, is beneficial on problems with few relatively tight constraints.

In the second part of the chapter we propose several new techniques for interleaving backtracking search with arc-consistency enforcement, with the ultimate goal of devising an approach that is highly effective on all varieties of CSPs. One of these techniques, called arc-consistency domain checking, improves the performance of arc-consistency. Three new heuristics, which control the amount of arc-consistency enforced at each step in the search, result in run times between those of forward checking and arc-consistency.

5.2 Introduction

In 1980, Haralick and Elliott ([40]) introduced the algorithms forward checking, partial looking ahead, and full looking ahead. Recall that forward checking ensures that each future variable has in its current domain, the D' set, at least one value that is compatible with the current instantiation. Partial and full looking ahead each perform additional processing to ensure compatibility among the future, uninstantiated variables. These three algorithms enforce a limited degree of arc-consistency. Over the last 17 years, forward checking has become one of the primary algorithms in the CSP-solver's arsenal, while partial and full looking ahead have received little attention. This neglect is due, no doubt, in large part to the negative conclusions about full looking ahead reached in [40]: "The checks of future with future units do not discover inconsistencies often enough to justify the large number of tests required."

One can envision a spectrum of backtracking-based algorithms, ordered from low to high based on how much consistency enforcing they do after each instantiation: backtracking (which does none), forward checking, partial looking ahead, full looking ahead, arc-consistency, and even higher levels such as path-consistency. Several algorithms which enforce arc-consistency during search have been proposed [90, 31, 61, 74].

In this chapter we have two goals. First, we want to determine empirically the relative merits of the algorithms along this spectrum, paying particular attention to the impact of constraint tightness (parameter T) and constraint density (parameter C) on the relative rankings. Because we concentrate on CSPs at the cross-over point, C and T are not two independent parameters, but there is an inverse relationship between them. Our results show that when C is high and T is low, it is best to do the least amount of consistency enforcing, and thus forward checking style look-ahead is best. When C is low and T is high, more intensive consistency enforcing pays off, and techniques that interleave full arc-consistency with search are beneficial. Why do our conclusions differ from those of Haralick and Elliott? There are two reasons. First, their experiments used CSPs with at most 10 variables, while we look at problems with well over 100 variables. The benefits of stronger consistency propagation may not outweigh the costs on smaller problems. Second, the problems in their experiments had complete (or almost complete) constraint graphs. Our results show that with many constraints forward checking is superior, but that on sparser problems it often is not.

In the second part of the chapter we show that the cost of running the arc-consistency subroutine can be substantially reduced by a minor technical modification to the way in which search and arc-consistency are integrated. We also present several new heuristics that enable the amount of arc-consistency performed to vary from problem to problem. Because different amounts of constraint propagation are superior on different types of problems, it would be useful to automatically recognize the optimum for each instance. Our experiments indicate that the heuristics are partially successful in achieving this goal.

Similar experimental studies on the effectiveness of enforcing arc-consistency during search have been performed by Bessière and Régin [7] and by El Sakkout *et al.* [23]. Systematic studies regarding the merits of different arc-consistency algorithms in the context of search have not been reported.

5.3 Look-ahead Algorithms

Arc-consistency and full and partial looking ahead, which all compare future variables with other future variables, can be integrated with a backjumping based algorithm such as BJ+DVO [70]. However, doing so makes backjumping's parent sets more difficult to maintain. Because our interest in this chapter lies primarily in comparing algorithms that enforce different amounts of consistency, we compare them in the context of backtracking only. Ultimately, the results learned with backtracking should be carried over to backjumping as well.

The BT+DVO algorithm, described in Fig. 5.1, is augmented to do additional consistency enforcing after each instantiation. The argument *Algorithm* controls whether one of the propagation subroutines PARTIAL-LOOKING-AHEAD, FULL-LOOKING-AHEAD, or AC-3 is to be invoked. If *Algorithm* is null (or any unrecognized value), then simple forward checking style processing is performed. BT+DVO forms the basis for all the algorithms compared in this chapter. When we refer to, for example, BT+DVO+PLA, or just PLA if the meaning is clear by context, we mean the BT+DVO algorithm with *Algorithm* = "PLA," which causes the PARTIAL-LOOKING-AHEAD subroutine to be invoked. Interleaving arc-consistency (IAC) is our term for enforcing arc-consistency after each variable instantiation.

When step 2 (d) does not call additional subroutines, BT+DVO does the same amount of look-ahead as forward checking. The difference between the two is that forward checking rejects a value that it detects will cause the domain of some future variable to be empty, while BT+DVO assigns the value (step 2 (b)) and then relies on the VARIABLE-ORDERING-HEURISTIC to make the empty-domain variable the next in the ordering. When that variable is selected, it is a dead-end. There is no difference between the two approaches in terms of consistency checks or search space.

The subroutines called by BT+DVO were described in Chapter 2, and are reprinted here for convenience: AC-3 (Fig. 5.2); REVISE, which is called by AC-3 (Fig. 5.3); FULL-LOOKING-AHEAD (Fig. 5.4); and PARTIAL-LOOKING-AHEAD (Fig. 5.5). We use AC-3 for the integrated arc-consistency algorithm because it is the most widely used arc-consistency algorithm. Other algorithms, such as AC-4, have better worst-case complexity, but in practice often do not perform as well.

Backtracking with DVO and varying amount of look-ahead
Input: *Algorithm* (one of {FLA, PLA, IAC})

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set *cur* equal to the index of the next variable, selected according to a VARIABLE-ORDERING-HEURISTIC (see Fig. 4.2).
2. Select a value $x \in D'_{cur}$. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.
   (c) (Forward checking style look-ahead) Examine the future variables $X_i, cur < i \leq n$, For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i; if D'_i is now empty, go to (e) (without examining other X_i's).
   (d) (Additional looking ahead.)
      i. If $Algorithm$ = FLA, perform FULL-LOOKING-AHEAD(i);
      ii. Else if $Algorithm$ = PLA, perform PARTIAL-LOOKING-AHEAD(i);
      iii. Else if $Algorithm$ = IAC, perform AC-3(i).
   (e) Go to 1.
3. (Backtrack.) If there is no previous variable, exit with "inconsistent." Otherwise, set *cur* equal to the index of the previous variable. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 5.1: The backtracking with dynamic variable ordering algorithm (BT+DVO) from Chapter 4, augmented to enforce varying degrees of arc-consistency after each instantiation.

5.4 First Set of Experiments

Our first experiment was designed to explore how the differing degrees of look-ahead embodied in BT+DVO, PLA, FLA, and IAC affect the performance of these algorithms, particularly in response to variation of constraint density (parameter C) and tightness (parameter T). We selected a diverse set of parameters for our random problem generator, all at the 50% solvable cross-over point,

AC-3(d)
$Q \leftarrow \{\text{arc}(i,j) | i > d, j > d\}$
repeat
    select and delete any arc(p,q) in Q
    REVISE(p,q)
    if $D'_p = \emptyset$
    then return
    if REVISE removed a value from D'_p
    then $Q \leftarrow Q \cup \{\text{arc}(i,p) | i > d\}$
until $Q = \emptyset$

Figure 5.2: Algorithm AC-3.

and with each generated 500 problems. These problems were then solved with each of the four algorithms, in each case coupled with dynamic variable ordering. The results of this experiment are reported in Table 5.1 and Table 5.2. A clear trend can be observed: less looking ahead is better with many constraints which are loose, while more looking ahead is superior with fewer, tighter constraints. We also note that the winner in each experiment was either forward-checking style look-ahead or interleaved arc-consistency – the least consistency processing or the most. FLA and PLA do an intermediate amount of constraint propagation, and for the parameters we selected it was too little or too much. Examining the data on an instance by instance basis confirmed the conclusions indicated by the means. For example, among the 232 unsolvable problems with parameters $\langle 75, 6, 0.1038, 0.333 \rangle$, BT+DVO was the best on 63 instances, BT+DVO+PLA on 0 instances, BT+DVO+FLA on 28 instances, and BT+DVO+IAC (the leader in mean CPU seconds) on 141 instances. However, we do not wish to over-generalize our results, as there are very likely classes of problems for which PLA or FLA exhibit the best balance between overhead and pruning, and would show the best CPU time.

It is not surprising that, for any of the experiments we ran, when the four algorithms are ranked in order of average number of search space nodes the result is BT+DVO > PLA > FLA > IAC. This ordering corresponds, in reverse order, to

Parameters	Algorithm	CC	Nodes	CPU
$\langle 125, 3, 0.1199, 0.111\rangle$	**BT+DVO**	382,724	13,603	2.64
	BT+PLA+DVO	11,317,017	7,797	22.49
	BT+FLA+DVO	12,666,962	4,091	23.43
	BT+IAC+DVO	10,458,562	2,923	12.98
$\langle 175, 3, 0.0358, 0.222\rangle$	BT+DVO	550,019	120,912	34.74
	BT+PLA+DVO	2,029,996	16,076	15.70
	BT+FLA+DVO	437,443	180	1.92
	BT+IAC+DVO	332,289	118	0.40
$\langle 150, 3, 0.0218, 0.333\rangle$	BT+DVO	977,861	236,475	66.49
	BT+PLA+DVO	414,706	3,898	4.17
	BT+FLA+DVO	9,644	9	0.06
	BT+IAC+DVO	6,978	6	0.01
$\langle 40, 6, 0.7308, 0.111\rangle$	**BT+DVO**	1,720,649	29,624	2.65
	BT+PLA+DVO	13,101,786	14,161	10.63
	BT+FLA+DVO	15,725,111	7,752	11.95
	BT+IAC+DVO	15,832,306	5,981	18.13
$\langle 60, 6, 0.2192, 0.222\rangle$	**BT+DVO**	1,811,001	58,239	6.89
	BT+PLA+DVO	14,356,286	17,794	17.88
	BT+FLA+DVO	13,208,022	7,134	15.02
	BT+IAC+DVO	13,355,728	5,025	13.37
$\langle 75, 6, 0.1038, 0.333\rangle$	BT+DVO	797,636	37,078	5.30
	BT+PLA+DVO	4,632,079	5,895	8.21
	BT+FLA+DVO	3,076,063	1,687	4.95
	BT+IAC+DVO	3,061,901	1,127	2.75
$\langle 100, 6, 0.0497, 0.444\rangle$	BT+DVO	1,434,782	107,211	19.29
	BT+PLA+DVO	2,629,167	3,096	6.82
	BT+FLA+DVO	887,442	441	2.16
	BT+IAC+DVO	788,671	266	0.66
$\langle 100, 6, 0.0391, 0.556\rangle$	BT+DVO	13,229,892	1,849,905	306.51
	BT+PLA+DVO	4,923,844	64,683	24.79
	BT+FLA+DVO	55,003	37	0.17
	BT+IAC+DVO	48,987	20	0.04

Table 5.1: Comparison of forward checking style look-ahead (BT+DVO), partial looking ahead (PLA), full looking ahead (FLA), and interleaving arc-consistency (IAC). Each number is the mean of about 250 unsolvable instances. The algorithm with the lowest mean CPU seconds in each group is in boldface.

Parameters	Algorithm	CC	Nodes	CPU
$\langle 125, 3, 0.1199, 0.111\rangle$	**BT+DVO**	149,024	5,650	1.06
	BT+PLA+DVO	4,417,305	3,341	8.87
	BT+FLA+DVO	4,976,763	1,782	9.29
	BT+IAC+DVO	4,134,198	1,313	5.24
$\langle 175, 3, 0.0358, 0.222\rangle$	BT+DVO	296,538	62,927	18.14
	BT+PLA+DVO	1,216,937	6,529	8.32
	BT+FLA+DVO	377,109	276	1.77
	BT+IAC+DVO	311,291	236	0.42
$\langle 150, 3, 0.0218, 0.333\rangle$	BT+DVO	290,827	54,584	14.66
	BT+PLA+DVO	96,959	466	0.76
	BT+FLA+DVO	65,027	151	0.47
	BT+IAC+DVO	64,690	150	0.10
$\langle 40, 6, 0.7308, 0.111\rangle$	**BT+DVO**	632,245	11,093	0.98
	BT+PLA+DVO	4,826,973	5,357	3.94
	BT+FLA+DVO	5,767,442	2,919	4.40
	BT+IAC+DVO	5,768,248	2,253	6.65
$\langle 60, 6, 0.2192, 0.222\rangle$	**BT+DVO**	895,240	29,792	3.47
	BT+PLA+DVO	7,129,740	9,237	8.94
	BT+FLA+DVO	6,572,723	3,721	7.52
	BT+IAC+DVO	6,564,987	2,621	6.65
$\langle 75, 6, 0.1038, 0.333\rangle$	BT+DVO	557,265	28,116	3.89
	BT+PLA+DVO	3,089,804	4,331	5.53
	BT+FLA+DVO	2,007,928	1,228	3.27
	BT+IAC+DVO	1,931,008	833	1.79
$\langle 100, 6, 0.0497, 0.444\rangle$	BT+DVO	1,267,104	128,593	21.59
	BT+PLA+DVO	1,578,365	2,341	4.14
	BT+FLA+DVO	448,510	304	1.14
	BT+IAC+DVO	400,663	224	0.36
$\langle 100, 6, 0.0391, 0.556\rangle$	BT+DVO	4,831,635	567,950	98.88
	BT+PLA+DVO	409,625	946	1.40
	BT+FLA+DVO	75,745	118	0.29
	BT+IAC+DVO	74,474	110	0.08

Table 5.2: Comparison of forward checking style look-ahead (BT+DVO), partial looking ahead (PLA), full looking ahead (FLA), and interleaving arc-consistency (IAC). Each number is the mean of about 250 solvable instances. The algorithm with the lowest mean CPU seconds in each group is in boldface.

REVISE(i, j)
1 **for** each value $y \in D'_i$
2 **if** there is no value $z \in D'_j$ such that $(\vec{x}_{cur}, X_i{=}y, X_j{=}z)$ is consistent
3 **then** remove y from D'_i

Figure 5.3: The Revise procedure.

the amount of consistency enforcing each algorithm does after instantiating a variable. It is well known that when subproblems have higher levels of local consistency the search space is smaller. Another factor is that additional processing after each instantiation eliminates values from the domains of future variables, and thus provides more information to guide the dynamic variable ordering heuristic. Consider the experiments with parameters $\langle 150, 3, 0.0218, 0.333\rangle$. For unsolvable problems, the average number of nodes in the search space was 9 for BT+FLA+DVO and 6 for BT+IAC+DVO, much less than the 236,475 nodes searched by BT+DVO. FLA and IAC rarely had to instantiate more than three variables before proving that the instance had no solution. For solvable problems with the same parameters, the average search space size is 151 for BT+FLA+DVO and 150 for BT+IAC+DVO, again much less than 54,584 required for BT+DVO. Recalling that these problems have 150 variables, we see that performing a large amount of look-ahead results in nearly backtrack-free search conditions. The overhead cost is that FLA and IAC perform between 500 and 2,000 consistency checks per node, while FC performs between 4 and six. For this set of parameters the intensive processing at each node is clearly a good investment.

Another view of some of the data presented in Table 5.1 and Table 5.2 is given in Figs. 5.6–5.11. The charts in these figures show the distribution of consistency checks (top charts) and CPU seconds (bottom charts) for selected sets of parameters. The distributions are represented by lognormal and Weibull curves with parameters estimated from the data.

Viewing the entire distribution conveys much more information than seeing only the average performance. In general, the distributions with BT+DVO has a

```
FULL-LOOKING-AHEAD(d)
for i ← d + 1 to n
    for j ← d + 1 to n
        REVISE(i, j)
        if D'_i = ∅
        then return
```

Figure 5.4: The full looking ahead algorithm.

higher σ for unsolvable problems and a lower β for solvable problems than do the distributions with greater consistency enforcing. High values of σ and low values of β reflect greater skewness in the data. We see in these experiments that the impact of additional look-ahead is most pronounced on the hardest problems in the distribution's tail.

In Fig. 5.7, the y-axis indicates the proportional frequency of problems making the number of consistency check or requiring the CPU time indicated on the x-axis. The curve labeled "BT+DVO" on the top chart of this Figure shows that with this algorithm a large proportion of the problems used less than 62,500 consistency checks. Of the four algorithms, the curves show that more problems required a small number of consistency checks with BT+DVO than with the other algorithms. Nevertheless, the mean number of consistency checks, for these problems, was higher with BT+DVO than with FLA or IAC (see Table 5.1). The reason is that the BT+DVO lognormal curve has a "heavier" tail than that of the other two algorithms. In other words, the hardest problems for BT+DVO are harder

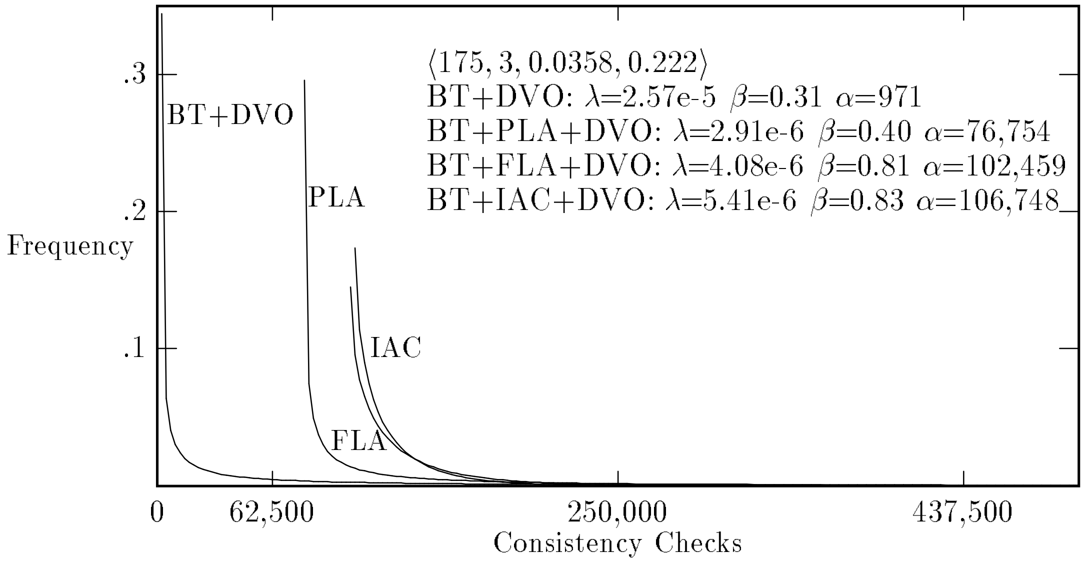
```
PARTIAL-LOOKING-AHEAD(d)
for i ← d + 1 to n
    for j ← i + 1 to n
        REVISE(i, j)
        if D'_i = ∅
        then return
```

Figure 5.5: The partial looking ahead algorithm.

than the hardest problem for FLA or IAC (measuring consistency checks), and this brings up the average for BT+DVO.

On problems with high C and low T, the extra work of arc-consistency can be detrimental, while the benefits are great when C is low and T is high [86]. Table 5.3 shows some relevant statistics gleaned from instrumenting the BT+DVO+IAC procedure. The first statistic, "Ratio CC / VR," measures the average number of consistency checks performed by AC-3 for each value removed from the domain of a future variable. When this measure is high, arc-consistency is relatively inefficient. The ratio tends to be lower with higher values of T.

The primary reason for performing interleaved arc-consistency, or indeed any amount of look-ahead, is to recognize future dead-ends. The second statistic, "Probability Empty Domain," in Table 5.3 shows how frequently the arc-consistency procedure uncovers a future dead-end that was not discovered first by forward checking style look-ahead. The table shows that this probability increases with smaller C and higher T. We also note that it tends to be lower for D=6 than for D=3, which is not surprising, since variables with larger domains are less likely to have all values removed, other factors being equal.



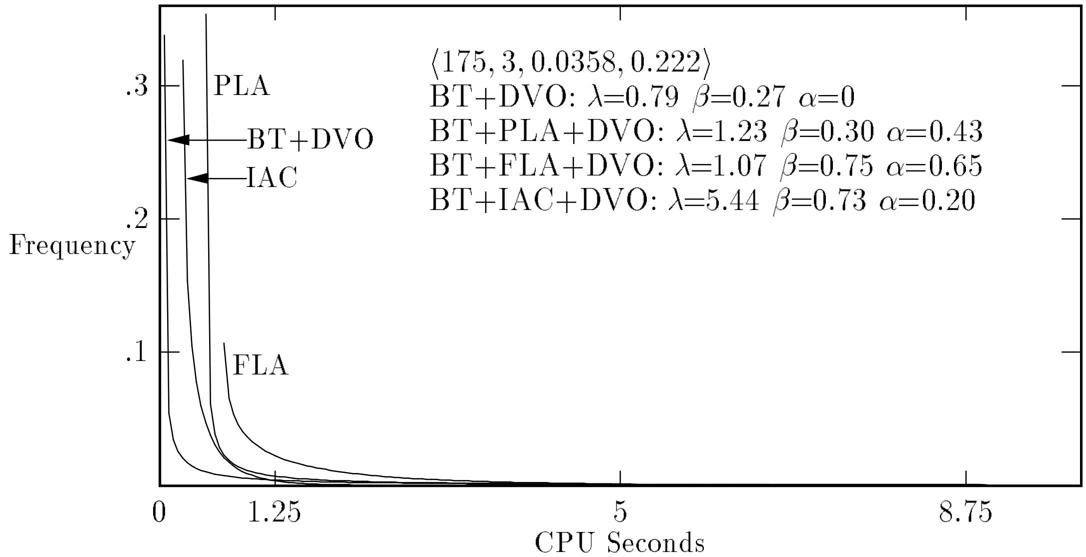
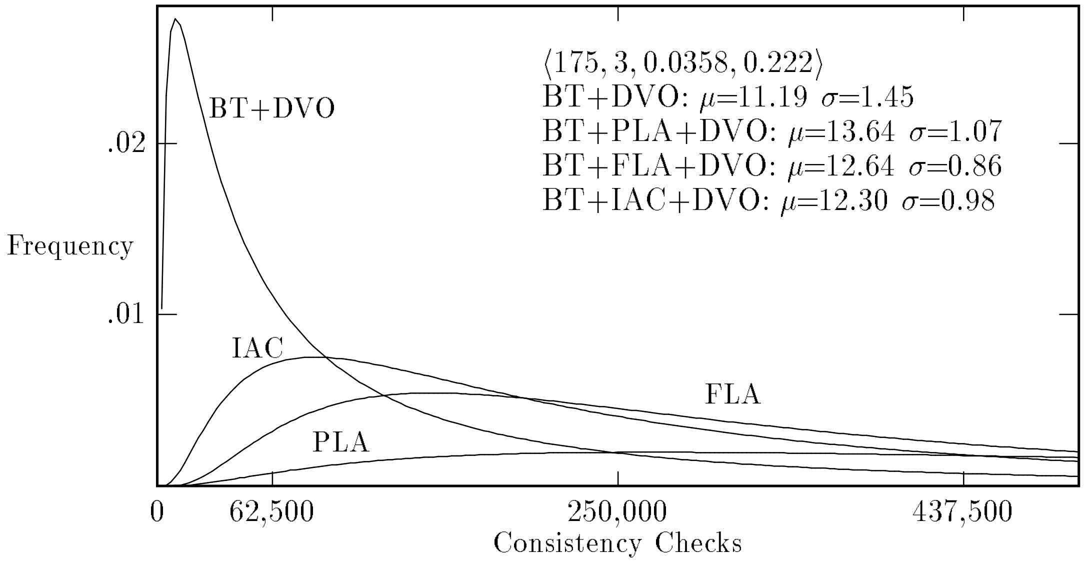


Figure 5.6: Weibull curves based on solvable problems generated from parameters $\langle 175, 3, 0.0358, 0.222\rangle$. The top chart is based on consistency checks, the bottom chart on CPU seconds. λ, β, and α parameters were estimated using the Modified Moment Estimator (see Chapter 3).



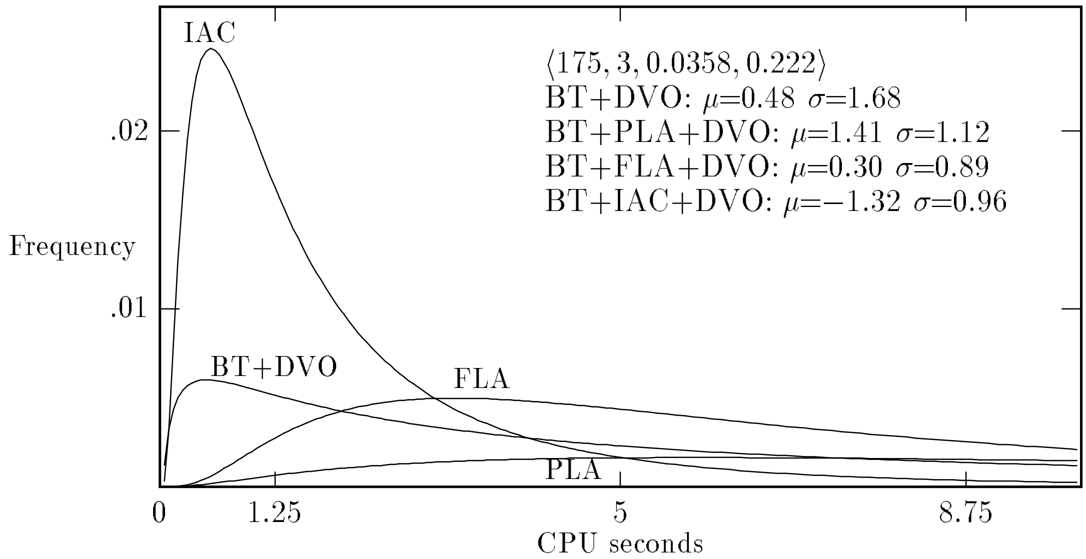


Figure 5.7: Lognormal curves based on unsolvable problems generated from parameters $\langle 175, 3, 0.0358, 0.222\rangle$. The top chart is based on consistency checks, the bottom chart on CPU seconds. μ and σ parameters were estimated using the Maximum Likelihood Estimator (see Chapter 3).

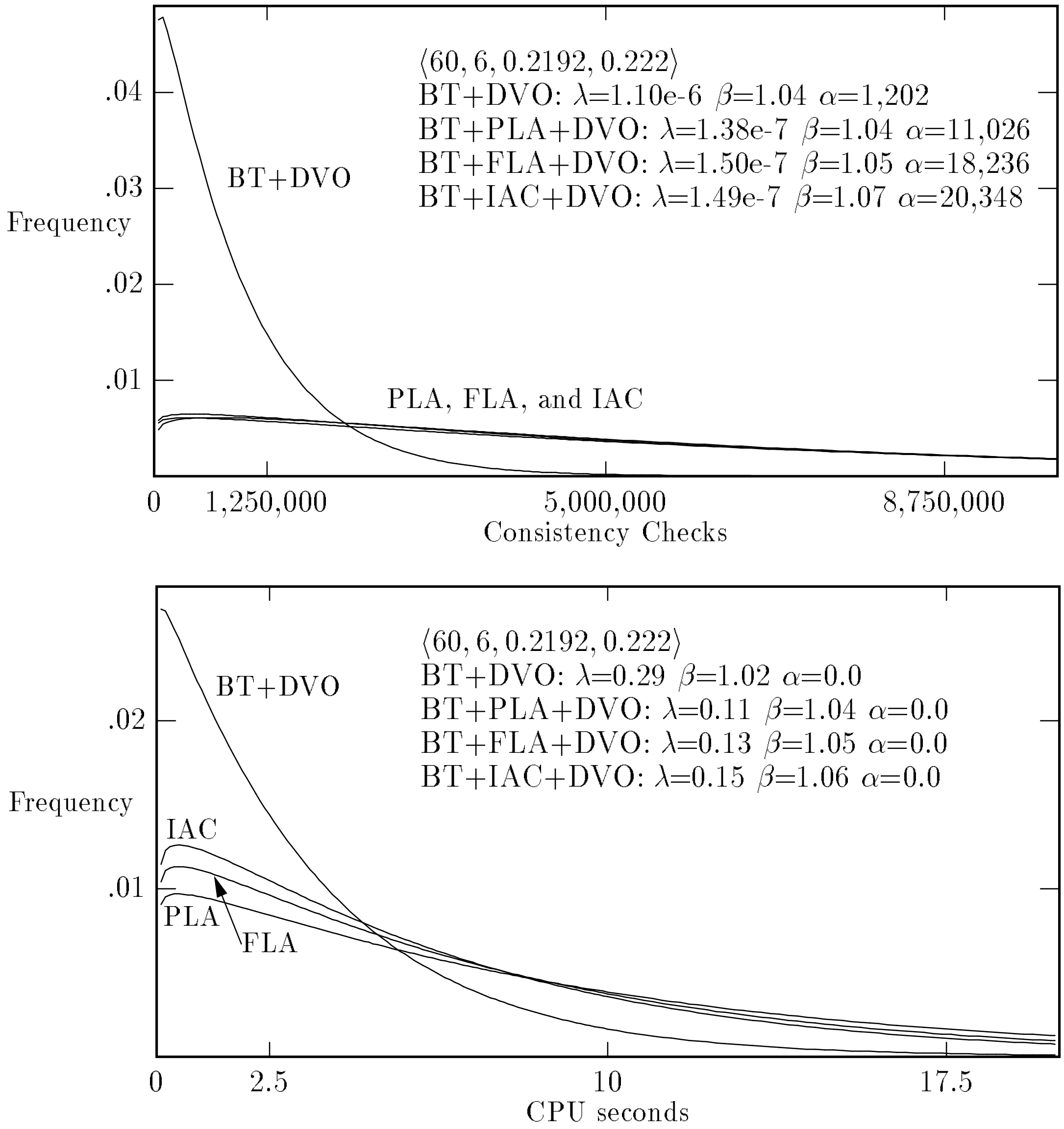


Figure 5.8: Weibull curves based on solvable problems generated from parameters $\langle 60, 6, 0.2192, 0.222\rangle$. The top chart is based on consistency checks, the bottom chart on CPU seconds. λ, β, and α parameters were estimated using the Modified Moment Estimator (see Chapter 3). The PLA, FLA, and IAC curves on the top chart are almost indistinguishable.

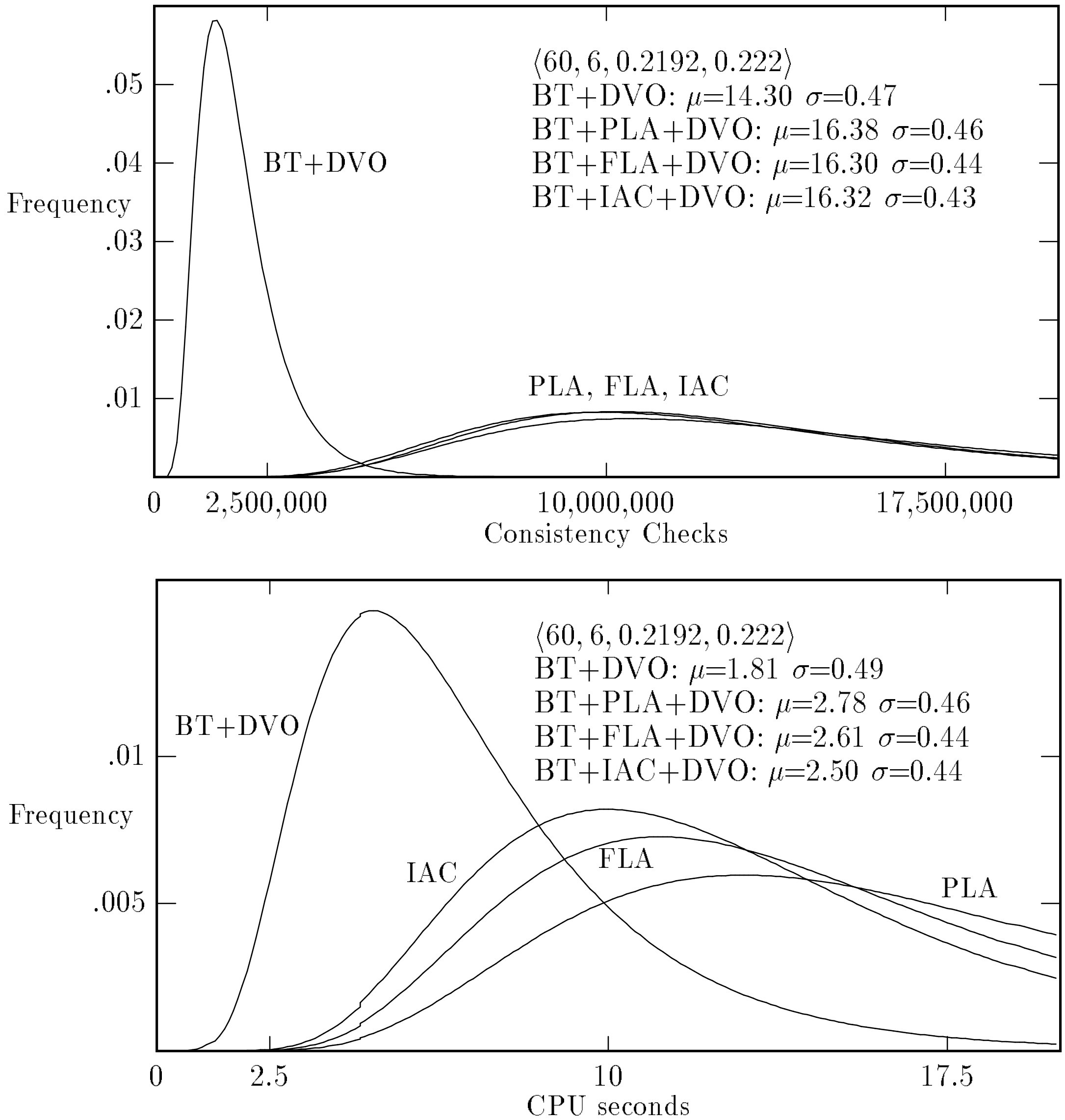
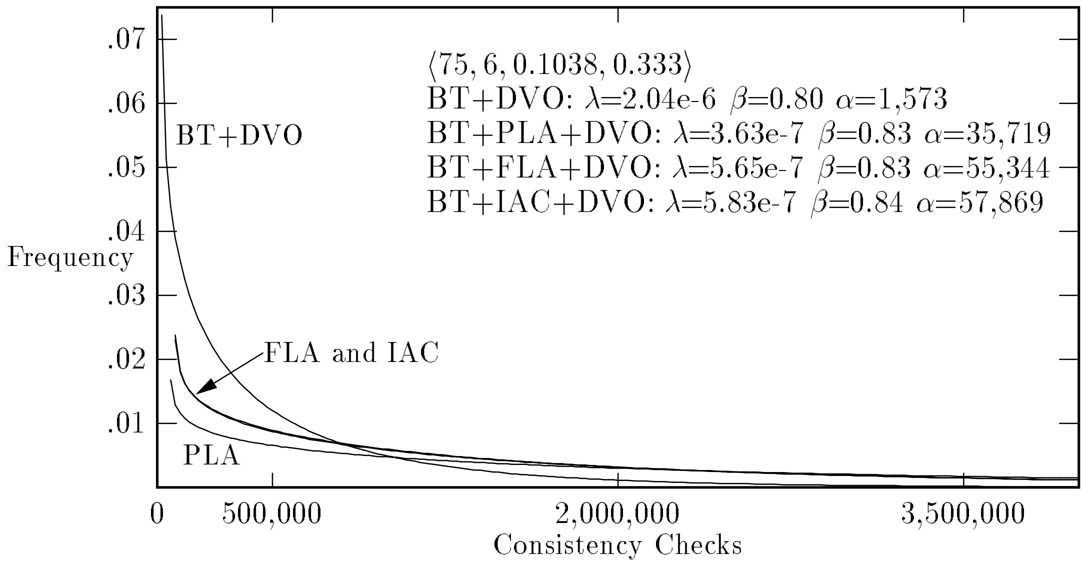


Figure 5.9: Lognormal curves based on unsolvable problems generated from parameters $\langle 60, 6, 0.2192, 0.222 \rangle$. The top chart is based on consistency checks, the bottom chart on CPU seconds. μ and σ parameters were estimated using the Maximum Likelihood Estimator (see Chapter 3). The PLA, FLA, and IAC curves on the top chart are almost indistinguishable.



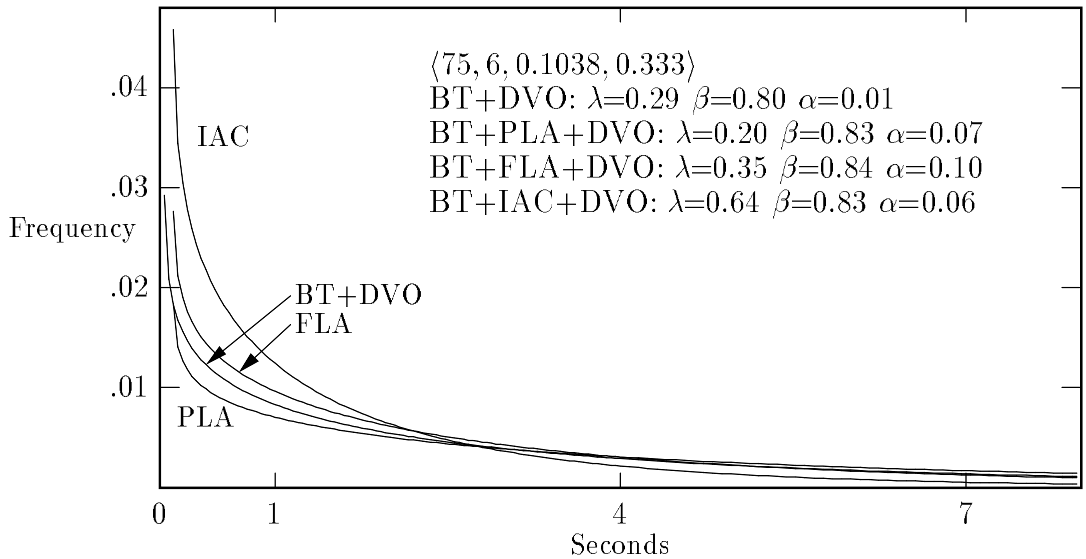


Figure 5.10: Weibull curves based on solvable problems generated from parameters $\langle 75, 6, 0.1038, 0.333 \rangle$. The top chart is based on consistency checks, the bottom chart on CPU seconds. λ, β, and α parameters were estimated using the Modified Moment Estimator (see Chapter 3). The FLA and IAC curves on the top chart are almost indistinguishable.

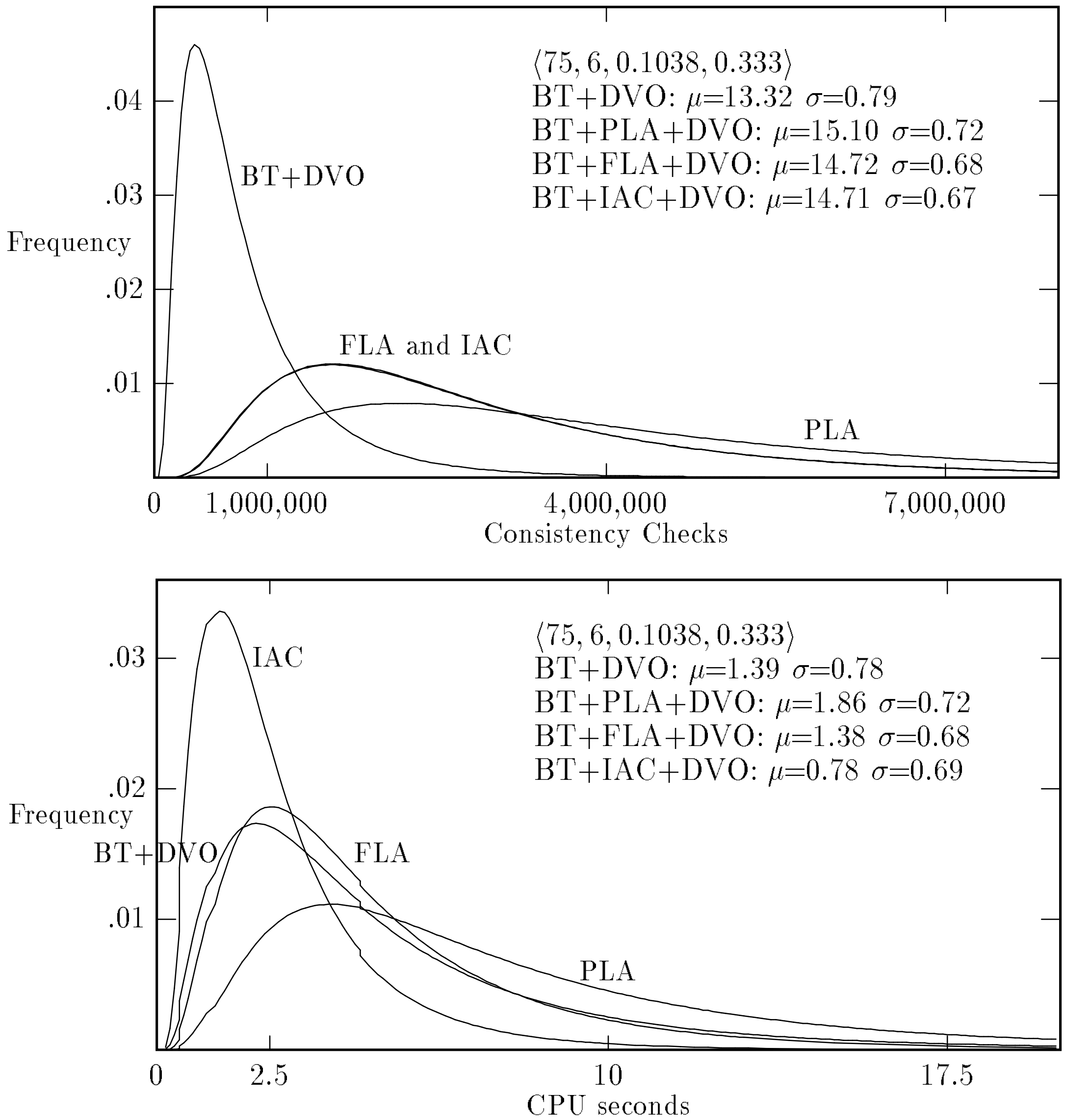


Figure 5.11: Lognormal curves based on unsolvable problems generated from parameters $\langle 75, 6, 0.1038, 0.333\rangle$. The top chart is based on consistency checks, the bottom chart on CPU seconds. μ and σ parameters were estimated using the Maximum Likelihood Estimator (see Chapter 3). The FLA and IAC curves on the top chart are almost indistinguishable.

The statistics in Table 5.3 show that when C is high and T is low, interleaving arc-consistency does many extra consistency checks for very little pay-off, as measured by dead-ends discovered. With low C and high T the AC-3 procedure is more efficient and produces a greater benefit.

Parameters	Ratio CC / VR	Probability Empty Domain
$\langle 125, 3, 0.1199, 0.111\rangle$	81.2	0.0000
$\langle 175, 3, 0.0358, 0.222\rangle$	33.4	0.0715
$\langle 150, 3, 0.0218, 0.333\rangle$	19.1	0.1844
$\langle 40, 6, 0.7308, 0.111\rangle$	191.8	0.0000
$\langle 60, 6, 0.2192, 0.222\rangle$	100.4	0.0012
$\langle 75, 6, 0.1038, 0.333\rangle$	59.7	0.0420
$\langle 100, 6, 0.0497, 0.444\rangle$	40.0	0.0956
$\langle 100, 6, 0.0391, 0.556\rangle$	31.8	0.1771

Table 5.3: Statistics from experiments in Figs. 5.1 and 5.2; just algorithm BT+DVO+IAC; solvable and unsolvable instances combined. The "Ratio CC / VR" column shows the ratio of consistency checks (CC) during the arc-consistency procedure to domain values removed (VR) during arc-consistency. "Probability Empty Domain" reports the observed probability that performing arc-consistency created an empty domain in a future variable.

Table 5.1 and Table 5.2 show only one large value of N for each combination of D and T. To show that these numbers are indicative of the trend with increasing N. Fig. 5.12 presents the results of experiments with BT+DVO, BT+DVO+FLA, and BT+DVO+IAC over a variety of values of N. The figure shows consistency checks and nodes expanded in the search tree, as well as CPU time. An approximately linear relationship between N and the logarithm of each measure is apparent, with the exception of BT+DVO on problems with T=.333. In this case the slope of the trend line increases around N=100, although more data points are required to determine whether or not the trend continues.

5.5 Variants of Interleaved Arc-consistency

The experiments presented so far in the chapter have demonstrated that the relative effectiveness of different consistency enforcing techniques is sensitive to the number of constraints in a CSP and the tightness of those constraints. In this section we define and evaluate several heuristics designed to adapt the amount of look-ahead in a dynamic fashion to the characteristics of the current problem. Our approach is to modify BT+IAC+DVO so that the amount of constraint propagation can vary between forward checking and full arc-consistency. Before turning to these heuristics, we first introduce a modification to the Interleaved Arc-consistency algorithm that improves its performance.

5.5.1 Domain checking

Our first modification to BT+IAC+DVO is quite simple, and is based on the observation that the AC-3 algorithm is most efficient when the queue of unexamined variable, called Q, is as small as feasible. As presented in Fig. 5.2, every future variable is entered in Q after a new variable is instantiated. A better idea is to put into Q, after each instantiation, only those variables which had a domain value removed during step 2 (c) of BT+DVO. This change is reflected in Fig. 5.13 and Fig. 5.14, which show a modified version of step 2 (c) and of the AC-3 procedure. We call the new arc-consistency procedure AC-DC, for domain checking.

Because the variables omitted from the queue Q by AC-DC are those which did not have any values removed by the forward checking style look-ahead, and which thus have no impact on the arc-consistency processing, BT+IAC+DVO with AC-DC explores exactly the same search space as BT+IAC+DVO with AC-3. The difference is in the number of consistency checks which need to be made, and in the CPU time required to solve a CSP instance. In general, using AC-DC requires about half as much CPU time as using AC-3. We present in Table 5.4, the results

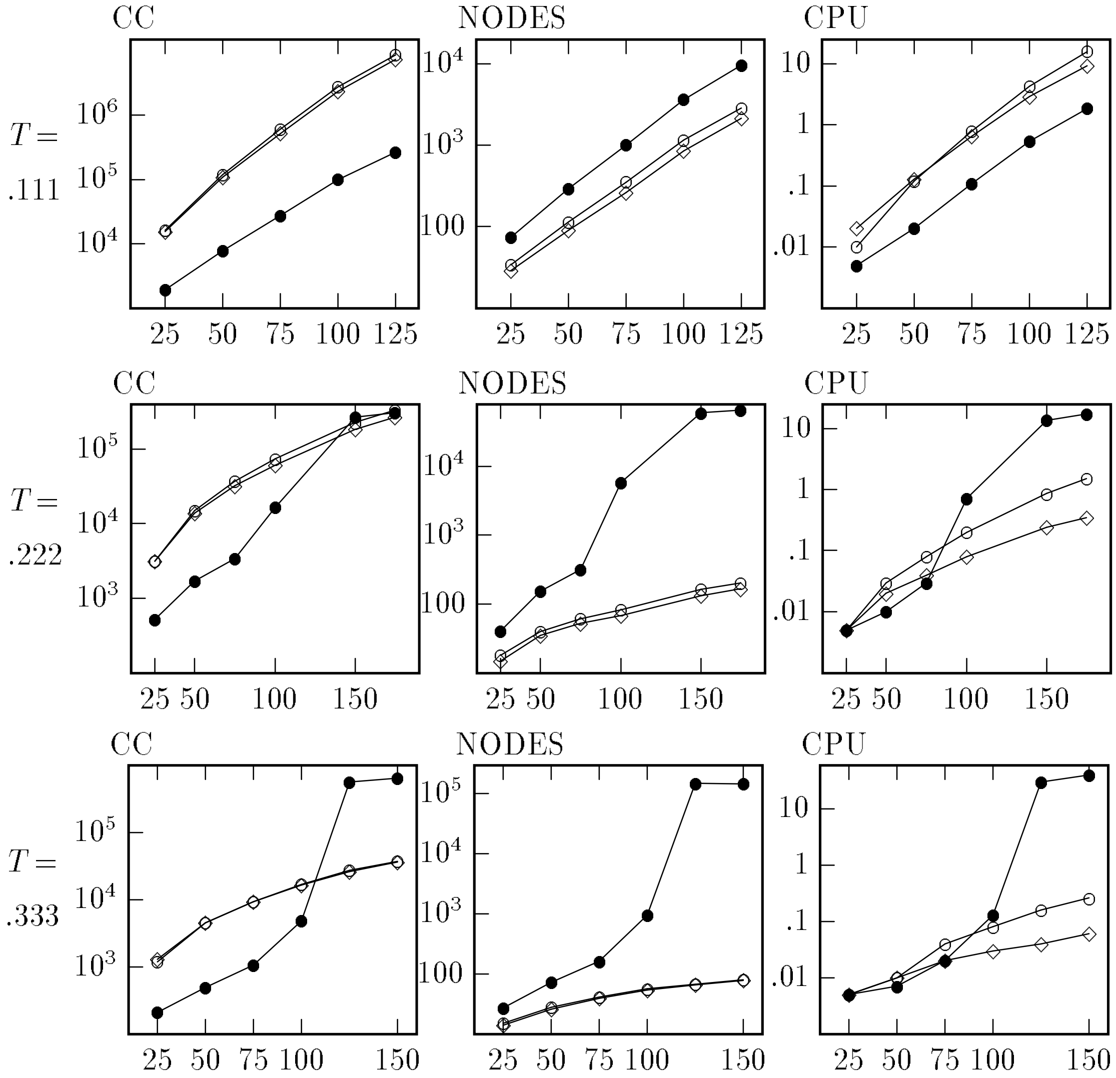


Figure 5.12: Comparison of BT+DVO (•), BT+DVO+FLA (○), and BT+DVO+IAC (◇). Each data point is the mean of 500 solvable and unsolvable instances generated with D=3, T=.111 (first row), T=.222 (second row) or T=.333 (third row), and N as indicated along the x-axis. Parameter C was set to the cross-over point. The left boxes on each row show consistency checks, the middle boxes show nodes expanded in the search tree, and the right boxes show CPU time in seconds. Note that the y-axes are logarithmic.

Backtracking with DVO and varying amount of arc-consistency

2. Select a value ...

   (c) (Forward checking style look-ahead) Examine the future variables $X_i, cur < i \leq n$, For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i and add X_i to Q; if D'_i is now empty, go to (e) (without examining other X_i's).

   (d) (Additional looking ahead.)

      iv. Else if *Algorithm* = IAC-DC perform AC-DC(i);

      v. Else if *Algorithm* = IAC-UNIT perform AC-UNIT(i);

      vi. Else if *Algorithm* = IAC-TRUNCATE perform AC-TRUNCATE(i);

      vii. Else if *Algorithm* = IAC-FLA perform AC-FLA(i).

Figure 5.13: Extended version of Fig. 5.1.

of an empirical comparison of the two variations of interleaving arc-consistency, as well as plain BT+DVO and several other variations described below. Because we did not find any significant difference in the performance of these algorithms on solvable and unsolvable problems, we combine both types in Table 5.4. Note that the generator parameters used in the Table 5.4 experiments are similar to those in Figs. 5.1 and 5.2, but have larger values of N.

AC-DC(d)
repeat
    select and delete any arc(p, q) in Q
    REVISE(p, q)
    if $D'_p = \emptyset$
    then return
    if REVISE removed a value from D'_p
    then $Q \leftarrow Q \cup \{\mathrm{arc}(i, p) | i > d\}$
until $Q = \emptyset$

Figure 5.14: Algorithm AC-DC, a modification of AC-3.

5.5.2 The unit variable heuristic

The intuition behind the unit variable heuristic (IAC-U) is closely coupled to the dynamic variable ordering scheme. One advantage of performing more look-ahead is that more values in the domains of future variables are removed. Consequently, the dynamic variable ordering heuristic is more likely to be effective. In the absence of a dead-end, we hope to find a future variable with a domain size of 1, as instantiating this single value represents a forced choice that will have to be made eventually. The unit variable heuristic terminates its consistency enforcing when the current domain of some future variable becomes 0 or 1. A future variable with only one value is called a "unit" variable. If a dead-end or unit variable is found, that variable will become the next in the ordering. The goal is to do enough looking ahead to guide effectively the variable ordering heuristic.

AC-UNIT(d)
0.1 **if** any future variable has domain size 1
0.2 **then return**
1 **repeat**
2   select and delete any arc(p, q) in Q
3   REVISE(p, q)
4   **if** $D'_p = \emptyset$ or $|D'_p| = 1$
5   **then return**
6   **if** REVISE removed a value from D'_p
7   **then** $Q \leftarrow Q \cup \{\text{arc}(i, p) | i > d\}$
8 **until** $Q = \emptyset$

Figure 5.15: Algorithm AC-DC with the unit variable heuristic.

Only two changes are required to the AC-DC algorithm in Fig. 5.14 to implement the unit variable heuristic (see Fig. 5.15). Lines 0.1 and 0.2 check for any future variable with only one value it its domain, and if one is found the algorithm returns. Line 4 is modified to test if the variable on which arc-consistency has just been enforced has just one value left; again the procedure returns immediately if so.

Empirical results of using the unit variable heuristic are reported in Table 5.4. CPU time for IAC with the heuristic was between BT+DVO and BT+IAC+DVO in seven out of eight parameter settings. Similar results hold on an instance by instance basis. Over all eight sets of parameters, including $\langle 75, 6, .1744, .222\rangle$, where using the unit variable heuristic produced average CPU time worse than both BT+DVO and BT+IAC+DVO, 87% of individual instance CPU times with the heuristic were between the CPU times of BT+DVO and BT+IAC+DVO.

Unfortunately, using the unit variable heuristic severely degraded the performance of BT+IAC+DVO on problems with relatively tight constraints. Stopping the arc-consistency procedure after a unit variable is produced is often harmful, largely because further processing often uncovers a variable with an empty domain, and the resulting dead-end cuts off an unfruitful branch of the search tree.

5.5.3 The full looking ahead method

Another method for guiding IAC to a reduced amount of arc-consistency can be developed by combining full looking ahead with AC-3's Q data structure. The result, called IAC-FLA, is a variant of IAC which performs a single pass over the Q set (see Fig. 5.16). Just as full looking ahead does not consider the impact of future variables on other future variables, IAC-FLA does not add variables to Q when they are modified by REVISE.

```
AC-FLA(d)
repeat
    select and delete any arc(p, q) in Q
    REVISE(p, q)
    if D'_p = ∅
    then return
until Q = ∅
```

Figure 5.16: Algorithm AC-DC, with the full looking ahead method.

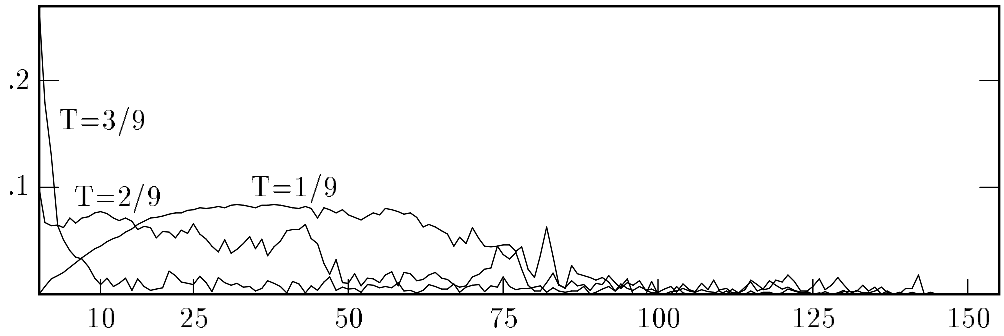


Figure 5.17: The fraction of future variable domain values removed by IAC (along the y-axis), as a function of depth in the search tree (along the x-axis). Each line represents the mean over 500 instances, for a single set of parameters with T=3, D=1/9, 2/9 or 3/9, and N and C as in Table 5.1.

The goal of achieving performance between BT+DVO and BT+IAC+DVO was largely achieved with IAC-FLA. Nevertheless, the results of IAC-FLA are unsatisfactory – slightly worse than IAC-U in all but one case in Table 5.4. As with IAC-U, it seems that this heuristic is often causing useful constraint propagation to be bypassed.

5.5.4 The truncation heuristic

To help us understand why IAC is better for certain classes of problem than for others, we modified our program to report what fraction of the values in the domains of future variables are removed during arc-consistency process. This fraction is a measure of the effectiveness of arc-consistency. In terms of the REVISE procedure in Fig. 5.3, we measured the ratio of times a value y is removed in line 3 to the number of values y considered in line 1. We tabulated this ratio relative to the value of d, the depth in the search tree which is a parameter to AC-DC. The results are shown in Fig. 5.17.

Studying Fig. 5.17 prompted the observation that when arc-consistency is most successful, that is, for T=3/9, most of its power seems to come at shallow levels in the search tree, where d is small. When the constraints are loose, e.g. T=1/9, arc-consistency is most likely to remove values somewhat deeper in the tree, but in such cases the cost to do so does not seem to be justified. Thus it seems that the most useful time to perform arc-consistency is after instantiating variables relatively high in the search tree. Interestingly, this conclusion seems to be counter to the intuition of Gaschnig, who proposed doing backtracking on the top of the tree and adding arc-consistency lower in the tree [31].

```
AC-TRUNCATE(d)
1    if d ≤ 10
2    then return
3    perform AC-DC
```

Figure 5.18: Algorithm AC-DC with the truncation heuristic.

We therefore developed a heuristic for IAC called truncation. The modification is simple: arc-consistency is performed only when the newly instantiated variable is at depth 10 or less in the search tree (see Fig. 5.18). At depth greater than 10, BT+IAC+DVO with truncation is identical to BT+DVO.

The truncation heuristic did not turn out to be a good performer in our experiments (see Table 5.4). We also tried truncation levels other than 10, but with no improvement. We were unable to complete the experiments with the truncation heuristic for some sets of parameters, because dozens of instances were taking several hours each.

5.5.5 Experimental comparison

We compared the algorithms and heuristics described above using a variety of parameters for the random problem generator. The average results are reported in Table 5.4. This table does not separate results for satisfiable and unsatisfiable instances, since we observed the same overall behavior on both types of problems.

Parameters	BT+DVO	IAC	IAC-DC	IAC-U	IAC-F	IAC-T
$\langle 150, 3, .0995, .111\rangle$	**3.02**	19.55	8.99	7.07	10.50	7.38
$\langle 200, 3, .0313, .222\rangle$		3.77	**1.92**	4.05	8.60	12.78
$\langle 200, 3, .0160, .333\rangle$		3.53	**1.01**	12.29	19.83	
$\langle 50, 6, .5794, .111\rangle$	**3.89**	20.69	12.74	4.93	7.08	8.19
$\langle 75, 6, .1744, .222\rangle$	17.50	41.28	**15.92**	23.44	29.00	25.01
$\langle 90, 6, .0861, .333\rangle$	51.66	8.83	**3.99**	7.25	8.99	43.17
$\langle 125, 6, .0395, .444\rangle$		4.04	**2.51**	8.87	10.55	
$\langle 150, 6, .0209, .556\rangle$		6.38	**4.34**	29.45	26.15	

Table 5.4: Comparison, by mean CPU seconds, of six algorithms based on BT+DVO: regular BT+DVO, BT+DVO with interleaved arc-consistency using AC-3 (IAC), BT+DVO with interleaved arc-consistency using domain checking (IAC-DC), BT+IAC+DVO-DC with the unit variable heuristic (IAC-U), BT+IAC+DVO-DC with the full looking ahead heuristic (IAC-F), and BT+IAC+DVO-DC with truncation at level 10 (IAC-T). Each number is the mean of 500 satisfiable and unsatisfiable instances. The best time in each row is in boldface. The BT+DVO and IAC-T positions are blank when we were unable to run all instances because too much CPU time was required.

On the problems in these experiments, the best combination was BT+DVO+IAC-DC, which had the lowest mean CPU time in six out of eight sets of problems.

5.6 Conclusions

In this chapter we studied one of the most intriguing questions in solving CSPs: what is the right balance between search and consistency enforcing? How can we make consistency enforcing cost-effective most of the time? We cannot provide definitive answers to these questions, but our experiments provide some insight into how they will be resolved for any particular problem or class of problems. We examined several known consistency methods interleaved with BT+DVO. We showed that when the constraints were tight and relatively few, interleaving an arc-consistency procedure after each instantiation was very effective on the problems we tested. In particular, a new variation of AC-3 called AC-DC reduced the number of consistency checks and was especially effective. When the constraints were loose and there were many constraints, arc-consistency was detrimental, and the best choice was to use lower levels of consistency, such as forward checking in the BT+DVO algorithm. Because the experiments were all conducted at the 50% satisfiable cross-over point, they cannot be considered a reliable guide to other regions of the parameter space.

We also studied several consistency enforcing approaches that lie between forward checking and interleaving arc-consistency in the amount of work performed after each instantiation. Ideally an intermediate approach would always perform close to the better of BT+DVO and BT+IAC+DVO. Two such algorithms were proposed by Haralick and Elliott: full looking ahead and partial looking ahead. We proposed three new heuristics: the unit variable heuristic, the full looking ahead heuristic, and the truncation heuristic. On the problems we used none of these techniques were successful enough to dislodge BT+DVO and BT+IAC+DVO as the algorithms of choice when looking ahead.

An important question is whether the addition of backjumping would improve the performance of an interleaved arc-consistency algorithm. On the problems we have experimented with the answer is probably no, because IAC reduces the search space so much that few opportunities arise for large jumps; however, a BJ+DVO+IAC algorithm might be effective on much larger problems.

Chapter 6

Look-ahead Value Ordering

6.1 Overview of the Chapter

Algorithms such as forward checking and integrated arc-consistency speed up backtracking by causing dead-ends to occur earlier in the search, and by providing information that is useful for dynamic variable ordering. In this chapter, we show that another use of looking ahead is a domain value ordering heuristic, which we call look-ahead value ordering or LVO[1]. LVO ranks the values of the current variable, based on the number of conflicts each value has with values in the domains of future variables. Our experiments show that look-ahead value ordering can be of substantial benefit, especially on hard constraint satisfaction problems.

6.2 Introduction

In this chapter we present a new heuristic for prioritizing the selection of values when searching for the solution of a constraint satisfaction problem. If a constraint satisfaction problem has a solution, knowing the right value for each variable would enable a solution to be found in a backtrack-free manner. When a CSP has only a small number of solutions, much time is often spent searching branches of the search space which do not lead to a solution. To minimize

[1]This work was first reported in Frost and Dechter [28].

backtracking, we should first try the values which are more likely to lead to a consistent solution. Our new algorithm, look-ahead value ordering (LVO), implements a heuristic that ranks the values of the current variable based on information gathered during a forward checking style look-ahead, determining the compatibility of each value with the values of all future variables. Although the heuristic does not always correctly predict which values will lead to solutions, it is frequently more accurate than an uninformed ordering of values. Our experiments show that while the overhead of LVO usually outweighs its benefits on easy problems, the improvement it provides on very large problems can be substantial. Interestingly, LVO often improves the performance of backjumping on problems without solutions.

Look-ahead value ordering does the same type of look-ahead as does the forward checking algorithm [40]. Because forward checking rejects values that it determines will not lead to a solution, it can be viewed as doing a simple form of value ordering. In this regard LVO is more refined, because it also orders the values that may be part of a solution.

6.3 Look-ahead Value Ordering

Look-ahead value ordering ranks the values of the current variable, based on the impact each value would have on the domains of the future variables. Combining BJ+DVO with LVO results in BJ+DVO+LVO; a description of this algorithm appears in Fig. 6.1. The algorithm is essentially the same as BJ+DVO from Chapter 4; the differences are in steps 1A, 2 (b), and 2 (c).

Step 1A of BJ+DVO+LVO is where the algorithm's look-ahead phase takes place. The current variable is tentatively instantiated with each value x in its domain D'_{cur}. BJ+DVO+LVO looks ahead, in a forward checking style manner, to determine the impact each x will have on the D' domains of uninstantiated

Backjumping with DVO and LVO

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur to be the index of the next variable, according to a VARIABLE-ORDERING-HEURISTIC. Set $P_{cur} \leftarrow \emptyset$.

1A. (Look-ahead value ordering.) Rank the values in D'_{cur} as follows: For each value x in D'_{cur}, and for each value v of a future variables $X_i, cur < i \leq n$, determine the consistency of $(\vec{x}_{cur-1}, X_{cur}{=}x, X_i{=}v)$. Using a heuristic function, compute the rank of x based on the number and distribution of conflicts with future values v.

2. Select a value $x \in D'_{cur}$. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop the highest ranked value x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.
   (c) (This step can be avoided by caching the results from step 1A.) Examine the future variables $X_i, cur < i \leq n$. For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i and add X_{cur} to P_i; if D'_i becomes empty, go to (d) without examining other X_i's.
   (d) Go to 1.
3. (Backjump.) If $P_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent." Otherwise, set $P \leftarrow P_{cur}$; set cur equal to the index of the last variable in P. Set $P_{cur} \leftarrow P_{cur} \cup P - \{X_{cur}\}$. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 6.1: Backjumping with DVO and look-ahead value ordering (LVO).

variables. In section 6.4, we discuss three heuristic functions that can be called in this step to rank the values.

In step 2 (b), the current variable is instantiated with the highest ranking value. If the algorithm returns to a variable because of a backjump, the highest ranked remaining value in its domain is selected. If the variable is re-instantiated after earlier variables have changed, then the ranking of the values has to be repeated in step 1A.

Step 2 (c) essentially disappears in BJ+DVO+LVO; once a value is actually selected, it would not make sense to repeat the look-ahead that has already been done. To avoid repeating consistency checks, our implementation saves in tables the results of step 1A. After a value is chosen in 2 (b), the appropriate D's and Ps of future variables are copied from these tables instead of being recomputed in step 2(c). The space required for these tables is not prohibitive. BJ+DVO uses $O(n^2d)$ space for the D' sets, where d is the size of the largest domain: n levels in the search tree (D' is saved at each level so that it does not have to be recomputed after backjumping) $\times$ n future variables $\times$ d values for each future variable. Our implementation of BJ+DVO+LVO uses $O(n^2d^2)$ space. There is an additional factor of d because at each level in the search tree up to d values are explored by look-ahead value ordering. Similarly, the space complexity for the P sets increases from $O(n^2)$ in BJ+DVO to $O(n^2d)$ for BJ+DVO+LVO. To solve a typical problem instance described in the next section, BJ+DVO required 1,800 kilobytes of random access memory, and BJ+DVO+LVO required 2,600 kilobytes. On our computer the additional space requirements of LVO had no discernable impact.

6.4 LVO Heuristics

We experimented with several LVO heuristics that rank values based on their conflicts with values of future variables.

The first heuristic, called min-conflicts (MC), considers each value in D'_{cur} and associates with it the number of values in the D' domains of future variables with which it is not compatible. The current variable's values are then selected in increasing order of this count. In other words, this heuristic chooses the value which removes the smallest number of values from the domains of future variables.

The second heuristic is inspired by the intuition that a subproblem is more likely to have a solution if it doesn't have variables with only one value. Each of the search trees rooted at some instantiation of X_{cur} is a different subproblem. A variable with only one remaining value makes a subproblem "fragile," in the sense that if that value is removed the subproblem has no solution. The max-domain-size (MD) heuristic therefore prefers the value in D'_{cur} that creates the largest minimum domain size in the future variables. For example, if after instantiating X_{cur} with value x_1 the $\min_{i \in \{cur+1,\ldots,n\}} \mid D'_i \mid$ is 2, and with $X_{cur}{=}x_2$ the min is 1, then x_1 will be preferred.

Our third heuristic function for selecting a value is based on the Advised Backtracking (ABT) algorithm of Dechter and Pearl [18]. ABT uses an estimate of the number of solutions in each subproblem to choose which value to instantiate next. The ES (Estimate Solutions) heuristic for LVO computes an upper bound on the number of solutions by multiplying together the domain size of each future variable, after values incompatible with a value in D'_{cur} have been removed. The upper bound is higher than the actual number of solutions if there are constraints between the future variables, as is usually the case. The value that leads to the highest upper bound on the number of solutions is chosen first.

In addition to the three LVO heuristics, we also experimented with a static value ordering heuristic we call static least-conflicts (S-LC). We call this heuristic static because it runs once, before search, and the order it assigns to the values does not change. Values are ranked in ascending order by the number of values of other variables with which they conflict. For instance, consider variable $X1$ in the coloring problem of Chapter 2 (Fig. 2.2). Its value *red* conflicts with three values in other variables, *blue* conflicts with four values, and *green* conflicts with one value. The S-LC heuristic will rank the values in order *(green, red, blue)*, and the values will always be considered in this order when $X1$ has to be instantiated. We developed the S-LC heuristic as a "straw-man" so that we could compare LVO heuristics against not just plain BJ+DVO, but also against BJ+DVO augmented with a value ordering that was very cheap to compute. As described in the following section, the S-LC heuristic performed quite well.

It is worth mentioning that we experimented with several other heuristics which were not as good as those reported in this chapter. In general, we found value ordering heuristics that seemed more "sophisticated" to be slightly poorer in practice. For example, we tried modifying the MD heuristic with several tie-breaking functions, because often more than one value results in the same minimum future domain size. Each tie breaker degraded the average performance of the algorithm. We have no explanation for this puzzling result.

6.5 Experimental Results

6.5.1 Comparing LVO heuristics

We conducted an experiment to determine the relative performance of the LVO heuristics. Eight sets of parameters near the cross-over point were selected, and with each set we generated 500 random CSPs. Five algorithms were applied to each instance: BJ+DVO (without a value ordering heuristic), BJ+DVO with S-LC, and BJ+DVO+LVO with the three heuristics described in the previous section. The results are reported in Tables 6.1 and 6.2. The tables show mean CPU time; mean consistency checks and number of nodes correlated closely with CPU time.

Tables 6.1 and 6.2 together contain 16 rows comparing the five algorithms. BJ+DVO+LVO with the MC heuristic had the lowest CPU time in 9 out of 16 cases. Second best was BJ+DVO with the S-LC heuristic, which was best in 5 out of 16. The MD and ES heuristics were each fastest in one case. The differences in average CPU time were mostly relatively small, but MC was clearly the best

	Mean CPU seconds (250 unsolvable instances)				
Parameters	No VO	S-LC	LVO-MC	LVO-MD	LVO-ES
$\langle 150, 3, .0995, .111\rangle$	11.90	11.40	12.50	12.28	13.39
$\langle 200, 3, .0313, .222\rangle$	5.15	4.79	4.81	5.04	4.97
$\langle 250, 3, .0125, .333\rangle$	14.88	8.52	8.03	12.00	8.44
$\langle 50, 6, .5796, .111\rangle$	21.72	21.24	21.58	21.00	21.61
$\langle 60, 6, .2192, .222\rangle$	10.53	10.48	10.37	10.38	10.62
$\langle 75, 6, .1038, .333\rangle$	7.55	7.49	7.36	7.46	7.56
$\langle 100, 6, .0497, .444\rangle$	13.48	12.07	11.49	12.70	11.91
$\langle 100, 6, .0319, .556\rangle$	7.94	5.03	5.37	6.98	5.68

Table 6.1: Comparison of BJ+DVO without LVO ("No VO"), with a static least-conflicts value ordering ("S-LC"), and with LVO using three heuristics described in the text.

heuristic for BJ+DVO+LVO. In the rest of the chapter, reference to LVO implies the MC heuristic.

6.5.2 Further experiments with LVO

We experimented further with value ordering by selecting several additional sets of parameters and with each set generating 500 instances that were solved with BJ+DVO, BJ+DVO+SLC, and BJ+DVO+LVO. Based on the experiments in Tables 6.1 and 6.2, as well as other tests, we concluded that LVO is most effective on problems with large numbers of variables, large domains, and near the cross-over point. Problems with these characteristics tend to take a large amount of computer time to solve. We adopted two strategies that allowed us to demonstrate the value of LVO on problems that were not too computationally expensive. The first strategy was to use problems with tight constraints and sparse constraint graphs. For instance, problems at N=100 and D=12 at the cross-over point might be extremely time consuming to solve, except that we used a small number (120, or C=.0242) of extremely tight constraints (T=.764 or 110/144). Another method for generating easier problems with large N and D is to select parameters that are

	Mean CPU seconds (250 solvable instances)				
Parameters	No VO	S-LC	LVO-MC	LVO-MD	LVO-ES
$\langle 150, 3, .0995, .111\rangle$	5.46	4.09	4.53	5.58	4.76
$\langle 200, 3, .0313, .222\rangle$	4.27	3.96	3.62	3.89	3.64
$\langle 250, 3, .0125, .333\rangle$	1.85	1.00	0.86	1.40	0.79
$\langle 50, 6, .5796, .111\rangle$	8.93	6.57	6.71	7.75	6.66
$\langle 60, 6, .2192, .222\rangle$	5.28	4.01	3.83	4.94	4.04
$\langle 75, 6, .1038, .333\rangle$	5.51	4.65	4.36	5.02	4.38
$\langle 100, 6, .0497, .444\rangle$	9.81	8.60	8.22	10.09	8.01
$\langle 100, 6, .0319, .556\rangle$	3.36	2.32	1.90	3.40	2.38

Table 6.2: Comparison of BJ+DVO without LVO ("No VO"), with a static least-conflicts value ordering ("S-LC"), and with LVO using three heuristics described in the text.

not exactly at the cross-over point. We used this approach for the experiment with $\langle 100, 9, .0606, .444\rangle$, where C is 95% of the estimated cross-over value of .0672.

The results of these experiments are summarized in Table 6.3 (unsolvable instances) and Table 6.4 (solvable instances). Because we used some parameter combinations that were not at the cross-over point, two experiments with few or no unsolvable problems do not appear in Table 6.3. Tables 6.3 and 6.4 are designed to present a multi-faceted view of our experimental results, and therefore several different statistics are reported. The first column shows the four parameters to the random problem generator in the format $\langle N, D, C, T\rangle$, and the percentage of instances that had solutions. The columns titled "Mean CC" and "Mean CPU" show the average values for the count of consistency checks made and the number of CPU seconds used (on a SparcStation 4, 110 MHz processor). Two columns show the estimated values of the μ and σ parameters of the lognormal distribution, in Table 6.3), and in Table 6.4 the estimated values of the λ and β parameters of the Weibull distribution (the λ value has been multiplied by 1,000,000 to make the units more convenient). These parameters are based on the distribution of the number of consistency checks. As discussed in Chapter 3, in the lognormal distribution the mean and the variance both increase with larger μ and σ. The variance is particularly sensitive to increases in σ. With the Weibull distribution,

Parameters % Solvable	Algorithm	Mean CC	Mean CPU	Parameters μ	Parameters σ	% Best
$\langle 150, 6, 0.0209, 0.556\rangle$	No VO	2,476,071	76.13	12.27	2.55	34.3
49%	S-LC	1,718,777	52.87	11.81	2.47	24.4
	LVO	1,358,316	44.48	11.66	2.38	41.3
$\langle 250, 3, 0.0236, 0.222\rangle$	No VO	2,024,230	104.27	13.07	1.82	10.4
87%	S-LC	1,630,480	83.12	12.88	1.78	85.1
	LVO	1,484,038	92.47	12.79	1.80	4.5
$\langle 100, 12, 0.0242, 0.764\rangle$	No VO	592,041	12.42	10.80	2.10	41.4
64%	S-LC	532,075	10.63	10.59	2.04	15.5
	LVO	434,904	9.88	10.56	1.96	43.1

Table 6.3: Results of several experiments on CSPs with various parameters; this table show results for unsolvable problems.

a larger value of λ or β results in smaller mean and variance. The final column in the tables indicates the relative frequency with which each algorithm had the best CPU time.

The experiments indicate that using either value order technique substantially improves the performance of BJ+DVO on these relatively large problems, both on problems with solutions and on those which do not have solutions. In all but one case, unsolvable problems with parameters $\langle 250, 3, .0236, .222\rangle$, the ranking by either consistency checks or CPU time is BJ+DVO+LVO first, BJ+DVO+S-LC second, and plain BJ+DVO last. The data in the "% Best" column indicate that even when the presence of LVO substantially improves the mean performance of BJ+DVO, on an instance by instance basis LVO does not always help; in fact, in only one case, $\langle 100, 9, .0638, .444\rangle$, does LVO beat the other two algorithms in more than half the instances. Moreover, according to the estimated σ and β parameters, adding LVO (or S-LC) to BJ+DVO does not result in a substantial impact on the overall shape of the distribution of computational effort.

To understand how LVO affects the entire set of random CSP instances and produces substantially lower means in consistency checks and CPU time, it is important to take into account the extremely skewed nature of the distributions.

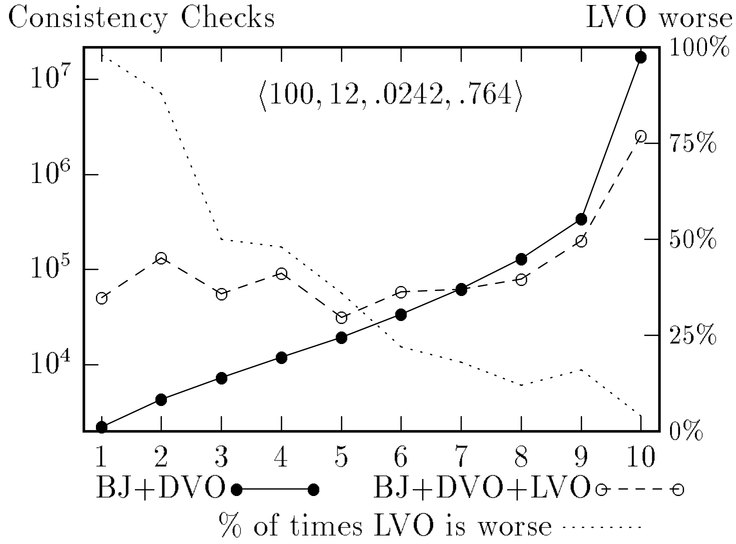


Figure 6.2: The 500 instances in one experiment were divided into 10 groups, based on the number of consistency checks made by BJ+DVO. Each point is the mean of 50 instances in one group; its position is based on the left-hand scale, which is logarithmic. The dotted line, showing the percentage of times BJ+DVO was better than BJ+DVO+LVO when measuring consistency checks, is related to the right-hand scale.

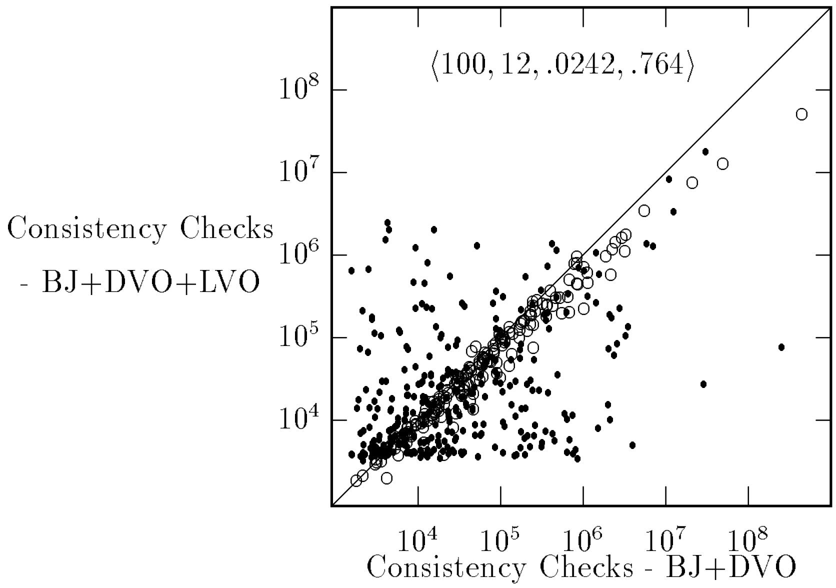


Figure 6.3: Each point (• = has solution, ○ = no solution) represents one instance out of 500. Note that both axes use logarithmic scales. Points below the diagonal line required fewer consistency checks with LVO.

Parameters % Solvable	Algorithm	Mean CC	 CPU	Parameters λ	 β	% Best
$\langle 150, 6, 0.0209, 0.556\rangle$	No VO	1,654,753	52.57	1.84	0.41	36.2
49%	S-LC	1,007,961	33.07	3.06	0.41	24.0
	LVO	713,982	25.75	3.86	0.43	39.8
$\langle 250, 3, 0.0236, 0.222\rangle$	No VO	273,155	14.36	6.27	0.55	37.0
87%	S-LC	259,352	14.11	8.63	0.47	25.9
	LVO	175,334	11.10	10.43	0.53	37.2
$\langle 100, 9, 0.0638, 0.444\rangle$	No VO	26,079,853	518.72	0.06	0.57	38.8
98%	S-LC	25,801,270	506.25	0.07	0.55	7.1
	LVO	17,746,277	386.91	0.12	0.49	54.1
$\langle 100, 9, 0.0606, 0.444\rangle$	No VO	1,930,435	32.78	0.86	0.56	25.6
100%	S-LC	1,349,394	23.41	1.58	0.48	29.0
	LVO	910,869	17.40	2.12	0.51	45.4
$\langle 100, 12, 0.0242, 0.764\rangle$	No VO	991,401	30.38	3.45	0.40	48.6
64%	S-LC	1,098,223	23.43	3.65	0.38	9.1
	LVO	434,904	9.88	8.57	0.46	42.3

Table 6.4: Results of several experiments on CSPs with various parameters; this table show results for solvable problems. The λ value has been multiplied by 10^6.

Lognormal distributions with $\sigma \geq 1.80$ and Weibull distributions with $\beta < .60$, which correspond to the empirical distributions of the data in the experiments, have long, heavy tails. For example, in our experiments with $\langle 100, 12, 0.0242, 0.764\rangle$, about half the CPU time was spent solving the hardest 25 of the 500 problems. It is therefore important to observe how the impact of LVO varies according to the hardness of the instances. Fig. 6.2 illustrates the skew in the distribution, and how LVO affects problems of different difficulties. For this figure, the 500 instances in one experiment were divided into ten groups of 50, based on the number of consistency checks required by BJ+DVO. The easiest 50 were put in the first group, the next easiest 50 in the second group, and so on. LVO is harmful for the first several groups, and then produces increasingly larger benefits as the problems become more difficult.

The scatter chart in Fig. 6.3 also indicates the distribution of the data. In this chart each bullet or circle represents one instance, its position on the chart indicating the number of consistency checks used by BJ+DVO (x-axis position)

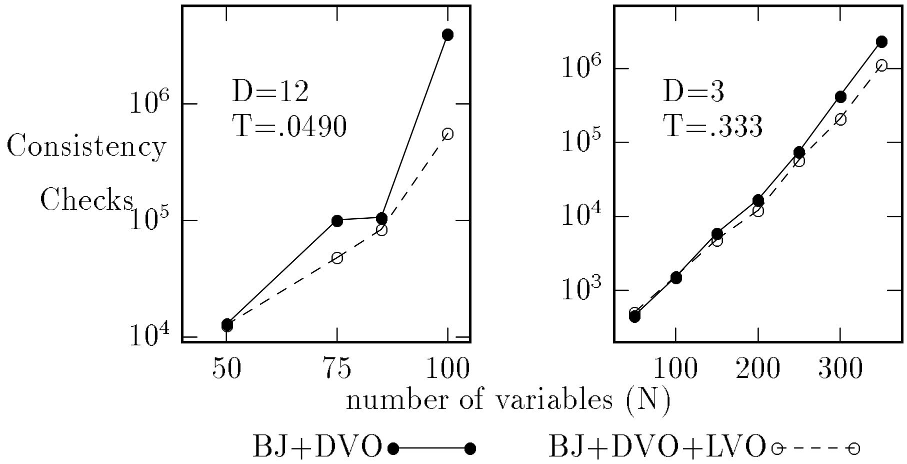


Figure 6.4: The benefits of LVO increases on problems with more variables. Each point is the mean of 500 instances. C is always set to the cross-over point. y-axis is logarithmic.

and by BJ+DVO+LVO (y-axis position). Many instances lie along or near the diagonal, demonstrating an approximately equal performance by both algorithms. Several dozen instances are very easy for one algorithm, requiring around 100 to 200 consistency checks, and somewhat harder for the other, requiring up to 100,000 consistency checks. The overall means are strongly affected by a few instances, both solvable and unsolvable, which require 10 or more times as much work with BJ+DVO than with BJ+DVO+LVO.

Figs. 6.2 and 6.3 show data drawn from just one set of parameters. The pattern of data from experiments with other parameters is quite similar. The overall conclusion we draw from our experiments with BJ+DVO and BJ+DVO+LVO is that on sufficiently difficult problems LVO almost always produces substantial improvement; on medium problems LVO usually helps but frequently hurts; and on easy problems the overhead of LVO is almost always worse than the benefit. Very roughly, "sufficiently difficult" is over 1,000,000 consistency checks and "easy" is under 10,000 consistency checks.

In general the statistics for CPU time are slightly less favorable for LVO than are the statistics for Consistency Checks, reflecting the fact that, in our

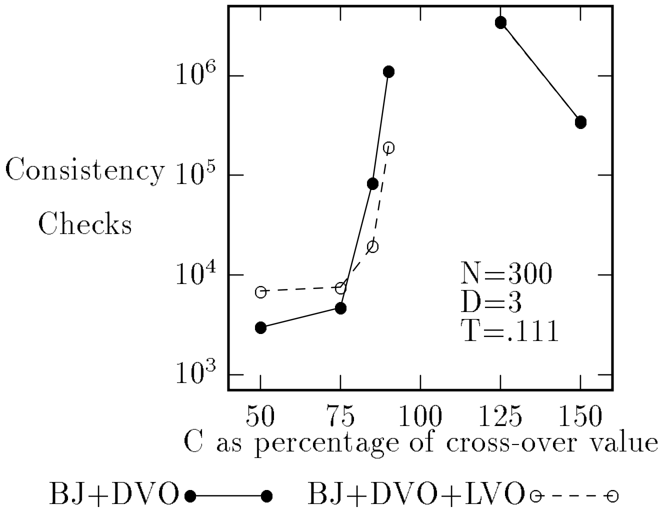


Figure 6.5: The varying effectiveness of LVO on problems not at the cross-over point. Each point on the chart represents the mean number of consistency checks from solving 500 CSP instances, using BJ+DVO and BJ+DVO+LVO. On over-constrained problems, the means of BJ+DVO and BJ+DVO+LVO are almost identical.

implementation, there is approximately a 5%–10% performance penalty in CPU time for LVO. This is caused by the need to store and copy the large tables that hold the results of looking ahead on different values of a variable (Step 2(c) of Fig. 6.1). One way to measure the overhead in the program which shows up in the CPU time but not in the count of consistency checks is to compute the ratio of consistency checks to CPU seconds. In the solvable problems with parameters $\langle 250, 3, 0.0236, 0.222\rangle$, the number of consistency checks per CPU second is 19,022 for BJ+DVO, 18,381 for BJ+DVO+S-LC, and 15,796 for BJ+DVO+LVO.

The graphs in Fig. 6.4 show that the impact of LVO increases as the number of variables increase. This is not surprising, as we have seen that within one set of parameters LVO is more effective on harder problems. Moreover, when variables have small domain sizes, a larger number of variables is required for LVO to have a beneficial impact. For instance, at N=75 and D=12, LVO improves BJ+DVO substantially (see Fig. 6.4), while with the small domain size D=3, the impact of LVO does not appear until N is larger than 200.

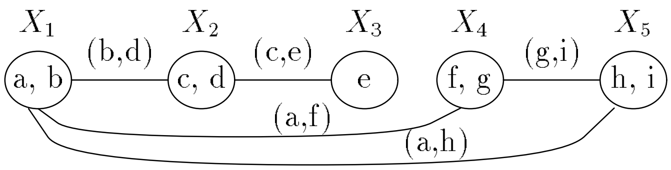


Figure 6.6: The constraint graph representing a CSP with 5 variables. The domain of each variable is listed inside its oval, and the constraints are indicated by arcs, with the *dis*allowed value pairs noted.

The efficacy of LVO also depends on how near the parameters are to the 50% solvable crossover point. As the data in Fig. 6.5 indicate, LVO is detrimental on very easy underconstrained problems, when C is less than 80% of the cross-over point value. These problems have many solutions, and the extra work LVO does exploring all values of a variable is almost always unnecessary. When problems are sufficiently overconstrained, C greater than 125% of cross-over value, LVO has very little effect on the number of consistency checks, and the points for BJ+DVO and BJ+DVO+LVO on Fig. 6.5 are indistinguishable.

6.6 LVO and Backjumping

With backtracking the order in which values are chosen does not affect the size of the search space on problems which have no solution, or when searching for all solutions. Therefore it may be surprising that a value ordering scheme can help BJ+DVO on instances that are unsatisfiable, as the data in Table 6.3 and Fig. 6.3 indicate. Nonetheless, our experiments show that adding LVO to BJ+DVO almost always changes the number of consistency checks used on unsolvable problems, often reducing them. One unsolvable instance required 442 million consistency checks without LVO and 52 million with LVO. As the following observation states, the reason is the interaction between backjumping and look-ahead value ordering.

Observation 1 *When searching for all solutions, or on problems which have no solution,*

1. *the order in which values are chosen does not affect the search space which backtracking explores;*
2. *the order in which values are chosen can affect the search space which backjumping explores.*

Proof. Part 1: Consider a search tree rooted at variable X. The n children of X are $X_1, X_2, \ldots, X_n$. The size of the search space $\mathrm{SS}(X)$ of this tree is $1 + \sum_{i=1}^{n} \mathrm{SS}(X_i)$. Since addition is commutative and the search spaces of the children do not interact, the order in which the search spaces rooted at the children of X are expanded will not affect $\mathrm{SS}(X)$.

Part 2: We will use a simple example; consider the problem depicted in Fig. 6.6, and assume $X_1 = a$ is assigned first. There are two value orders for X_2. If $X_2 = c$ is considered first, then X_3 will be a dead-end. $X_2 = d$ will be instantiated next, and an eventual dead-end at X_5 will lead to a jump back to X_4 and then to X_1. X_2 and X_3 will be jumped over because they are not connected to X_4 or X_5. On the other hand, if $X_2 = d$ is considered first, a different search space is explored because $X_2 = c$ is never encountered. Instead, X_2 and X_3 are jumped over after the dead-ends at X_4 and X_5.

Note that the observation holds whether a look-ahead or look-back method is used, and whether the variable ordering is static or dynamic. LVO can help on unsatisfiable problems, and on unsatisfiable branches of problems with solutions, by more quickly finding a consistent instantiation of a small set of variables which are later jumped over by backjumping.

6.7 Related Work

In general, domain-independent value ordering schemes have not been considered effective on CSPs, and relatively little work has been done on them. The success of our static least-conflicts heuristic was therefore unexpected, and is probably due to its being evaluated on larger and harder problems than previously used. This heuristic is similar to the Min-Conflicts heuristic developed by Minton *et al.* [56], although Minton *et al.*'s version is not static, and is used primarily for variable selection.

Geelen [32] describes an approach to value selection similar to ours. It is based on a forward checking style look-ahead, but does not employ backjumping. Empirical evaluation in [32] is based on the N-Queens problem.

Pearl [66] discusses similar value ordering heuristics in the context of the 8-Queens problem. His "highest number of unattacked cells" is the same as our max-conflicts heuristic, and his "row with the least number of unattacked cells" heuristic is the same as max-domain-size.

Dechter and Pearl [18] developed an Advised Backtrack algorithm which estimates the number of solutions in the subproblem created by instantiating each value. The estimate is based on a tree-like relaxation of the remainder of the problem. For each value, the number of solutions is counted, and the count is used to rank the values. Advised Backtrack was the first implementation of the general idea that heuristics can be generated from a relaxed version of the problem instance.

Sadeh and Fox [76] also use a tree-like relaxation of the remaining problem, in the context of job-shop scheduling problems. Their value ordering heuristic considers as well the impact of capacity constraints and demand on scarce resources.

6.8 Conclusions and Future Work

We have introduced look-ahead value ordering, an algorithm for ordering the values in a constraint satisfaction problem. Our experiments showed that for large and hard problems, LVO could improve the already very good BJ+DVO algorithm by over a factor of five.

We also evaluated a simple static value ordering heuristic called static least-conflicts. Although it is not able to react to changing conditions during search, this heuristic often was an improvement over plain BJ+DVO.

One drawback of LVO is that it is somewhat complex to implement, as it uses a set of tables to cache the results of values that have been examined during the ranking process but not yet instantiated. Manipulating these tables incurs a small CPU overhead. Another disadvantage of LVO is that on easy solvable problems, where there are many solutions and hence many acceptable value choices, it is usually detrimental. LVO needlessly examines every value of each variable along the almost backtrack-free search for a solution.

LVO is almost always beneficial on difficult instances that require over one million consistency checks. Unexpectedly, it even helps on problems without solutions when used in conjunction with backjumping.

We tested LVO using a forward checking level of look-ahead. It would also be interesting to explore the possibility that a more computationally expensive scheme, such as interleaved arc-consistency (see Chapter 5) or directional arc-consistency [18], will pay off in the increased accuracy of the value ordering. Another research direction is to reduce the overhead of LVO on easy problems. This might be achieved by only employing value ordering in the earlier levels of the search, or by having value ordering automatically "turn off" when it notices that current values are in conflict with relatively few future values, indicating an underconstrained problem. A simple way to eliminate the overhead of LVO on very easy problems would be to always run a non-LVO algorithm first; if that algorithm has not completed by, say, 100,000 consistency checks, it is cancelled and problem solving is restarted with an LVO version. At the price of 100,000 extra consistency checks on some difficult problems, the costs of LVO on the easy majority of problems would be avoided.

Chapter 7

Dead-end Driven Learning

7.1 Overview of the chapter

This chapter evaluates the effectiveness of learning for speeding up the solution of constraint satisfaction problems[1]. It extends previous work [15] by introducing a new and powerful variant of learning and by presenting an extensive empirical study on much larger and more difficult problem instances. Our results show that the addition of learning can speed up backjumping with dynamic variable ordering.

7.2 Introduction

Our goal in this chapter is to study the effect of *learning* in speeding up the solution of constraint problems. Learning has been studied in many branches of Artificial Intelligence. Shavlik and Dietterich [80] distinguish two fundamental ways in which a computer system can learn: it can "acquire new knowledge from external sources," which is usually called *empirical learning* or *inductive learning*, or it can "modify itself to exploit its current knowledge more effectively." This latter type of learning is often called *speedup learning*, and it is into this category that CSP learning falls.

[1]This research was first reported in Frost and Dechter [26].

The function of learning in search-based problem solving is to record in a useful way some information which is explicated during the search, so that it can be reused either later on the same problem instance, or on similar instances which arise subsequently. One application of this notion involves the creation of macro-operators from sequences and subsequences of atomic operators that have proven useful in solutions to earlier problem instances of the domain. This idea was exploited in STRIPS with MACROPS [24, 48]. A more recent approach is to learn heuristic control rules using explanation-based learning [54, 55].

The approach we take involves a during-search transformation of the problem representation into one that may be searched more effectively. This is done by enriching the problem description by new constraints, also called *nogoods*, which do not change the set of solutions, but make certain information explicit. The new constraints are essentially uncovered by resolution during the search process [52]. The idea is to learn from dead-ends; whenever a dead-end is reached we record a constraint explicated by the dead-end. Learning during search has the potential for reducing the size of the remaining search space, since additional constraints may cause unfruitful branches of the search to be cut off at an earlier point. The cost of learning is that the computational effort spent recording and then consulting the additional constraints may overwhelm the savings. Minton [55] refers to this trade-off as the *utility* problem: "the cumulative benefits of applying the knowledge [additional constraints] must outweigh the cumulative costs of testing whether the knowledge is applicable." Another potential drawback of the type of learning we propose with CSPs is that the space complexity can be exponential.

This type of learning has been presented in dependency-directed backtracking strategies by researchers interested in truth maintenance systems [84], and within intelligent backtracking for Prolog [9]. It was treated more systematically by Dechter [15] within the constraint network framework. In [15], different variants of learning were examined, taking into account the trade-off between the overhead

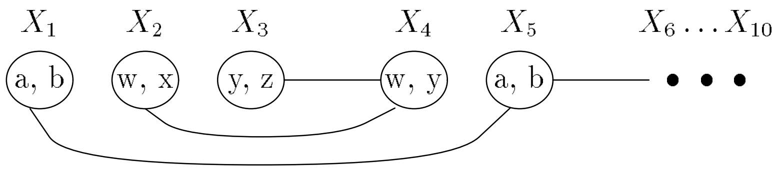


Figure 7.1: The constraint graph representing a CSP with ten variables. The domains of variables X_1 through X_5 are listed inside the ovals and constraints are indicated by arcs. Each constraint specifies that the two variables cannot be assigned the same value.

of learning and performance improvement. The results, although preliminary, indicated that learning during CSP search can be cost-effective. The empirical results in [15] indicate that on randomly generated instances only very restricted forms of learning paid off. When experimenting with the Zebra puzzle deeper forms of learning were quite effective; however, instances were solved with random variable orderings, and it remained unclear whether learning would still be effective under well-chosen orderings. Overall, the empirical evidence to-date is based on a small number of small-sized instances, under learning that restricts the size of constraints recorded.

The present study extends [15] in several ways. First, a new variant of learning, called jump-back learning, is introduced and is shown empirically to be superior to other types of learning. Secondly, we experiment with and without restrictions on the size of the constraints learned. Thirdly, we use the highly effective BJ+DVO algorithm from Chapter 4 as a comparison reference. Finally, our experiments use larger and harder problem instances than previously studied.

Because learning in CSPs operates by recording additional constraints, it is a variation of constraint propagation. Recall that a CSP is called k-consistent if every consistent instantiation of $k-1$ variables can be extended consistently to any kth variable. 2-consistency is often called arc-consistency, and 3-consistency is known as path-consistency. Arc-consistency is enforced by removing values from the domains of variables; higher levels of consistency are enforced by adding constraints or tightening existing constraints, so that $k-1$ size instantiations which cannot be consistently extended are themselves prohibited.

A desired level of consistency can be enforced on a CSP before search, or instead of search. Consistency enforcing can also be interleaved with search (see Chapter 5). In the latter case, the consistency enforcing algorithm is applied to a subproblem which results from instantiating a subset of the variables. If one of these variables is assigned a new value after backtracking, then the results of the earlier consistency enforcing are out-of-date and must be recalculated. As an example, consider the small CSP shown in Fig. 7.1, and assume that the problem has no solution due to the constraints among variables X_6 through X_{10}. Enforcing arc-consistency before search will not change the problem, since in all constrained pairs of variables each value is compatible with at least one value in the domain of the other variable. If path-consistency is enforced, a constraint will be added between X_2 and X_3 which prohibits the assignment $(X_2{=}w, X_3{=}y)$, since with these values there is no compatible extension to X_4. Now let us look at the behavior of various search algorithms, assuming that the variables are considered in order of increasing subscript, and the values are selected in alphabetical order. Forward checking will instantiate $(X_1{=}a, X_2{=}w)$ and upon considering $X_3{=}y$ will discover an empty domain in X_4, therefore rejecting y for X_3. Since the variables from X_5 on lead to a dead-end, at some point forward checking will return to X_1. It will assign $X_1{=}b$, then $X_2{=}w$, and then again discover that $X_3{=}y$ is not acceptable. The discovery has to be made twice since it was not remembered when the algorithm backtracked to X_1.

A similar discovery, "forgetting," and re-discovery will happen on this problem if interleaved arc-consistency (IAC from Chapter 5) is used. Look-ahead algorithms forget when they backtrack, so that what they learn – that particular values cannot be part of a solution – can be stored with low-order polynomial space requirements.In Fig. 7.1's example CSP, it might make sense to record that the cause of the dead-end in variable X_4 is the combination $(X_2{=}w, X_3{=}y)$. Adding a constraint prohibiting this combination to the problem's database of constraints means the dead-end at X_4 will not have to be discovered again.

7.3 Backjumping

It will be useful to briefly review the conflict-directed backjumping algorithm, since the new learning technique we propose is based in part on the mechanics of backjumping.

When a variable is encountered such that none of its possible values is consistent with previous assignments, a dead-end occurs and a backjump takes place. The idea is to jump back over several irrelevant variables to a variable which is more directly responsible for the current conflict. The backjumping algorithm identifies a parent set, that is, a subset of the variables preceding the dead-end variable which are inconsistent with all its values, and continues search from the last variable in this set. If that variable has no untried values left, then a interior dead-end arises and further backjumping occurs.

Consider, for instance, the CSP represented by the graph in Fig. 7.3. Each node represents a variable that can take on a value from within the oval, and the binary constraint between connected variables is specified along the arcs by the disallowed value pairs. If the variables are ordered $(X_1, X_5, X_2, X_3, X_4)$ and a dead-end is reached at X_4, the backjumping algorithm will jump back to X_5, since X_4 is not connected to X_3 or X_2.

7.4 Learning Algorithms

Upon a dead-end at X_i, when the current instantiation $S{=}(X_1{=}x_1, \ldots, X_{i-1}{=}x_{i-1})$ cannot be extended by any value of X_i, we say that S is a *conflict set*. Note that when using a look-ahead approach, as BJ+DVO does, X_i may be any future, uninstantiated, variable, and a dead-end at X_i means that the domain D'_i has become empty. An opportunity to learn new constraints is presented whenever backjumping encounters a dead-end, since had the problem included an explicit constraint prohibiting the dead-end's conflict-set, the dead-end would have been avoided. To learn at a dead-end, we record one or more new constraints which make explicit an incompatibility among variable assignments that already existed, implicitly.

There is no point in recording S as a constraint at this stage, because under the backtracking control strategy this state will not recur. However, if S contains one or more subsets that are also in conflict with X_i, then recording these smaller conflict sets as constraints may prove useful in the continued exploration of the search space.

Varieties of learning differ in the way they identify smaller conflict sets. In [15] learning is characterized as being either *deep* or *shallow*. Deep learning only records *minimal* conflict sets, that is, those that do not have subsets which are also conflict sets. Shallow learning allows non-minimal conflict sets to be recorded as well. Non-minimal conflict sets are easier to discover, but may be more expensive to store and applicable less frequently in the remaining search.

Learning can also be characterized by *order*, the maximum constraint size that is recorded. In [15] experiments were limited to recording unary and binary constraints, or first and second order learning. A loose upper-bound on the space complexity of i-th order learning is $(nd)^i$, where n is the number of variables and d is the largest domain size.

Learning methods can also be distinguished by the type of dead-end at which learning takes place. Leaf dead-ends occur when the search algorithm moves to a new variable deeper in the search tree and cannot assign it a compatible value. In terms of BJ+DVO (see Fig. 7.2), the sequence of steps 1-2(a)-3 is a leaf dead-end. When the algorithm backtracks or backjumps to a variable and that variable has no remaining compatible values, an interior dead-end takes place. The corresponding steps of BJ+DVO are 3-2a-3. Learning can take place at either type of dead-end, but the conflict sets at interior dead-ends tend to be larger. The conflict set of a leaf dead-end must account for the incompatibility of each value in the domain of the dead-end variable. The conflict set of an interior dead-end must account for the incompatibility of all values on the leaves of the sub-tree rooted at the dead-end variable.

Backjumping with DVO

Input: *type, order, leaf-only*

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur to be the index of the next variable, according to a VARIABLE-ORDERING-HEURISTIC. Set $P_{cur} \leftarrow \emptyset$.
2. Select a value $x \in D'_{cur}$. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.
   (c) Examine the future variables $X_i, cur < i \leq n$. For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i and add X_{cur} to P_i; if D'_i becomes empty, go to (d) (without examining other X_i's).
   (d) Go to 1.
3. Learn, then backjump.
   (a) If $P_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent."
   (b) If *leaf-only* = TRUE and X_{cur} was reached by a backjump, go to (g).
   (c) If *type* = VALUE, perform VALUE-BASED-LEARNING(*order*);
   (d) else if *type* = GRAPH, perform GRAPH-BASED-LEARNING(*order*);
   (e) else if *type* = DEEP, perform DEEP-LEARNING(*order*);
   (f) else if *type* = JUMP, perform JUMP-BACK-LEARNING(*order*).
   (g) Set $P \leftarrow P_{cur}$; set cur equal to the index of the last variable in P. Set $P_{cur} \leftarrow P_{cur} \cup P - \{X_{cur}\}$. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 7.2: The BJ+DVO algorithm, augmented to call a learning procedure.

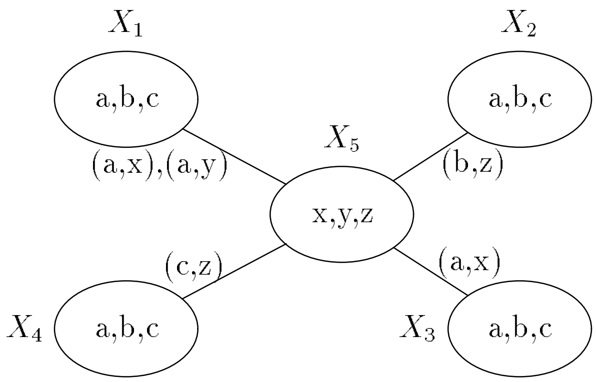


Figure 7.3: A small CSP. Note that the *dis*allowed value pairs are shown on each arc.

We experimented with four types of learning. Three were proposed by Dechter [15]: *graph-based shallow* learning, *value-based shallow* learning (called *full shallow* learning in [15]), and *deep* learning. We introduce a new variety of learning, called *jump-back* learning because of its reliance on backjumping's parent or "jump-back" set of variables. As described below, each learning algorithm relies on a subprocedure RECORD to add a new constraint or nogood to the problem representation. (We do not define RECORD, because it relies on the specific data structures used in the computer program.) In general, graph-based learning records the largest new constraints, and deep learning the smallest. Value-based learning and jump-back learning are intermediate forms that take different approaches to the tradeoff between minimizing the cost at each dead-end and learning useful constraints.

7.4.1 Value-based learning

In value-based learning, which is described in Fig. 7.4, all irrelevant variable-value pairs are removed from the initial conflict set S. If a variable-value pair $X_j = x_j$ does not conflict with any value of the dead-end variable then it is redundant and can be eliminated. For instance, if we try to solve the problem in Fig. 7.3 with the ordering $(X_1, X_2, X_3, X_4, X_5)$, after instantiating $X_1 = a, X_2 = b, X_3 = b, X_4 = c$, the dead-end at X_5 will cause value-based learning to record $(X_1 = a, X_2 = b, X_4 = c)$,

```
VALUE-BASED-LEARNING(order)
CS ← ∅          ; initialize conflict-set
for each instantiated variable X_i=x_i, 1 ≤ i ≤ cur,
    if X_i=x_i conflicts with some value of X_cur
    then add X_i=x_i to CS
if size of CS ≤ order
then RECORD(CS)
```

Figure 7.4: The value-based learning procedure.

since the pair $X_3 = b$ is compatible with all values of X_5. If all constraints in a CSP are binary, a three-dimensional table *CONF* of boolean values can be pre-computed before search. $CONF_{i,j,k}$ is TRUE if $X_i{=}x_j$ is in conflict with any value of X_k, and FALSE otherwise. Creating this table requires $O(n^2d)$ space and time complexity. Since there are at most $n-1$ variable preceeding the dead-end variable, by consulting *CONF* the time complexity of value-based learning at each dead-end is $O(n)$.

If the CSP has non-binary constraints, the same look-up table can be employed, but $CONF_{i,j,k}$ = TRUE will mean that $X_i{=}x_j$ in combination with some other variables is in conflict with at least one value of X_k. This approach maintains the efficient preprocessing requirement and the $O(n)$ time complexity per dead-end, but is undesirable because some variables will be retained in the conflict set unnecessarily. The alternatives are to construct a larger table in advance, or to do more extensive analysis of the conflict set at each dead-end, which will take more time.

7.4.2 Graph-based learning

Graph-based shallow learning is a relaxed version of value-based learning, where information on conflicts is derived from the constraint graph alone, without consulting the values currently assigned to variables (see Fig. 7.5). This approach may be particularly useful on sparse graphs. For instance, when applied to the CSP

GRAPH-BASED-LEARNING($order$)
$CS \leftarrow \emptyset$
for each instantiated variable $X_i, 1 \leq i \leq cur$,
  if there exists a constraint between X_i and X_{cur}
  then add $X_i{=}x_i$ to CS
if size of $CS \leq order$
then RECORD(CS)

Figure 7.5: The graph-based learning procedure.

in Fig. 7.3, graph-based shallow learning might record $(X_1{=}a, X_2{=}b, X_3{=}b, X_4{=}c)$ as a conflict set relative to a dead-end at X_5, since these variables are connected to X_5. The complexity of learning at each dead-end here is $O(n)$, since each variable is connected to at most $n-1$ other variables.

7.4.3 Jump-back learning

Jump-back learning uses as its conflict-set the parent set P that is explicated by the backjumping algorithm itself. Recall that BJ+DVO examines each future variable X_i and includes X_{cur} in the parent set P_i if X_{cur}, as instantiated, conflicts with a value of P_i that previously did not conflict with any variable. For instance in Fig. 7.3, when using the same ordering and reaching the dead-end at X_5, jump-back learning will record $(X_1{=}a, X_2{=}b)$ as a new constraint. These two variables are selected because the algorithm first looks at $X_1{=}a$ and, noting that it conflicts with $X_5{=}x$ and $X_5{=}y$, adds X_1 to P_5. Proceeding to $X_2{=}b$, the conflict with $X_5{=}z$ is noted and X_2 is added to P_5. At this point all values of X_5 have been ruled out, and the conflict set is complete. Since the conflict set needed for learning is

JUMP-BACK-LEARNING($order$)
$CS \leftarrow P_{cur}$
if size of $CS \leq order$
then RECORD(CS)

Figure 7.6: The jump-back learning procedure.

```
DEEP-LEARNING(order)
for each subset S of the instantiated variables x⃗_{cur−1},
            in order from smallest to largest,
    if every value of X_cur is in conflict with S and
        S is not a superset of any existing nogood,
    then if size of CS ≤ order
        then RECORD(CS)
```

Figure 7.7: The deep learning procedure.

already assembled by the underlying backjumping algorithm, the added complexity of computing the conflict set is constant. To achieve constant time complexity at each dead-end the parent set must be modified to include not only the parent variables but also their current values.

The conflict set identified by backjumping is not necessarily minimal. Refering to the problem in Fig. 7.3, if the variable ordering starts with (X_1, X_2), then the dead-end at X_5 will result in the minimal conflict set $(X_1{=}a, X_2{=}b)$, as discussed in the previous paragraph. But if the variable ordering is (X_3, X_1, X_2), then the parent set for X_5 will be $(X_3{=}a, X_1{=}a, X_2{=}b)$, which is not a minimal conflict set.

7.4.4 Deep learning

In deep learning all and only minimal conflict sets are recorded. With the CSP in Fig. 7.3, a dead-end at X_5 will cause deep learning will record two minimal conflict sets, $(X_1{=}a, X_2{=}b)$ and $(X_1{=}a, X_4{=}c)$. The deep learning algorithm can start with the value-based conflict set, generate its subsets and test whether they are conflict sets. Although this form of learning is the most accurate, its cost is prohibitive and in the worst-case is exponential in the size of the initial conflict set. As noted in [15], if r is the cardinality of the starting conflict set, we can envision a worst case where all the subsets of this set having $r/2$ elements are in

Parameters	Algorithm	CC	Nodes	CPU	Size
$\langle 125, 3, 0.1199, 0.111\rangle$	None	242,293	8,760	2.02	
	Graph	1,253,404	8,692	15.73	3.6
	Value	619,059	8,000	5.19	3.3
	Jump	518,454	7,828	3.14	3.3
	Deep	2,067,166	7,183	10.17	3.2
$\langle 175, 3, 0.0358, 0.222\rangle$	None	72,289	6,221	2.01	
	Graph	785,564	6,201	8.32	3.5
	Value	398,451	5,163	3.86	3.0
	Jump	26,637	2,008	0.56	2.5
	Deep	823,306	1,299	4.17	2.1
$\langle 150, 3, 0.0218, 0.333\rangle$	None	5,120	749	0.22	
	Graph	112,674	579	3.01	2.3
	Value	84,711	571	0.91	1.9
	Jump	1,332	188	0.07	1.8
	Deep	57,881	165	0.54	1.5

Table 7.1: Comparison of BJ+DVO, without learning ("None") and with four varieties of learning: Graph-based learning ("Graph"), Value-based learning ("Value"), Jump-back learning ("Jump"), and Deep Learning ("Deep"). All learning was restricted to 4th-order. Each number is the mean of 500 solvable and unsolvable instances. The "Size" column displays the average number of variables in the learned constraints.

conflict with X_{cur}. The number of minimal conflict sets will then be:

$$\#\text{min-conflict-sets} = \begin{pmatrix} r \\ r/2 \end{pmatrix} \cong 2^r,$$

which amounts to exponential time and space complexity at each dead-end.

7.5 Experimental Results

We evaluated the learning algorithms by combining them with BJ+DVO and solving several sets of random CSPs. The results are presented in the following subsections.

7.5.1 Comparing learning algorithms

The first experiment was designed to compare the effectiveness of the four learning schemes. Fig. 7.1 presents a summary of experiments with problems generated from three sets of parameters. 500 problems in each sets were generated and solved by five algorithms: BJ+DVO without learning, and then BJ+DVO with each of the four types of learning. In all cases a bound of four was placed on the size of the constraints recorded, and learning was limited to leaf dead-ends.

Looking at the size of the search space, the results reported in Fig. 7.1 indicate that the algorithms can be ranked as follows: no learning > Graph-based > Value-based > Jump-back > Deep. This result is expected: learning adds additional constraints which should reduce the size of the search space. Within the learning schemes, the size of the search space is inversely related to the average size of the learned constraints, since larger constraints tend to be less effective in the remainder of the search.

Because learning adds new constraints, it can increase the number of consistency checks performed later in the search. Moreover, learning applied to binary CSPs can add constraints with more than two variables, and these constraints can be more expensive to query[2]. Deep learning makes a particularly large number of consistency checks because at each dead-end it has to verify whether each subset of the value-based conflict set is a minimal conflict set.

This experiment demonstrates that only the new jump-back type of learning is effective on these reasonably large size problems. In the following discussion and figures, all references to learning should be taken to mean jump-back learning.

[2]In our implementation, binary constraints are represented by a four-dimensional array with one dimension for each variable and value, and a binary consistency check is therefore a constant time operation. Higher order constraints are stored in ordered lists, and checking consistency with one of these constraints is proportional to $r \times \log s$, where r is the size of the constraint, and s is the number of high-order constraints.

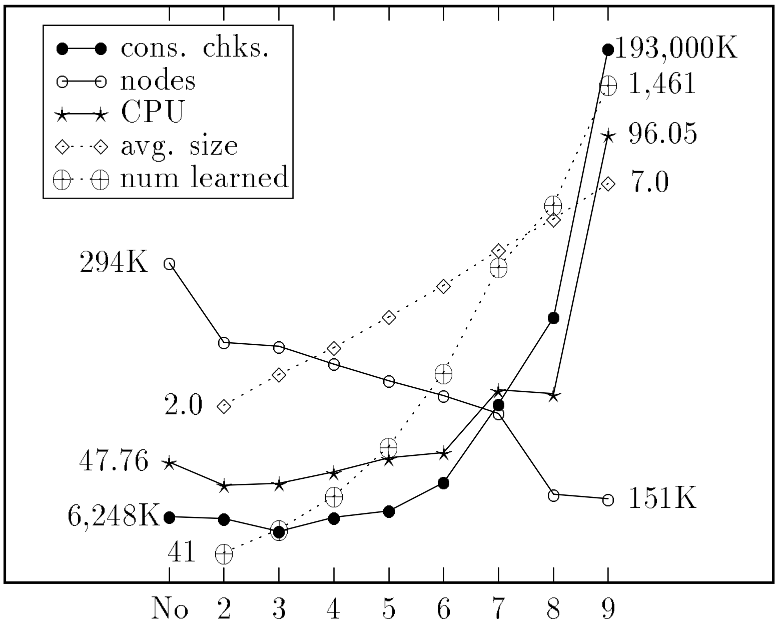


Figure 7.8: Results from experiments with parameters $\langle 100, 6, .0772, .333\rangle$ and varying orders of jump-back learning, as indicated on the x-axis ("No" signifies BJ+DVO without learning). Note that each statistic is on a different scale.

7.5.2 Testing i-th order learning

We now describe a set of experiments designed to investigate the impact on learning of restricting the size of the learned constraints. Recall that in ith-order learning, new constraints are recorded only if they include i or fewer variables. The experiments used random problems generated with parameters $\langle 100, 6, .0772, .333\rangle$ (reported in Fig. 7.8) and $\langle 125, 6, .0395, .444\rangle$ (reported in Fig. 7.9). For both experiments we generated 500 instances and processed them using BJ+DVO without learning, and using BJ+DVO with i-th order learning, where i ranged from 2 to 9. The graphs in the figures show the averages for number of consistency checks, the size of the search space, the CPU time in seconds, the number of constraints learned, and the size of the learned constraints. Each set of figures is plotted using a different y-axis scale, with the beginning and ending values indicated on the charts. In general, a higher order of learning results in a smaller search space, a larger number of learned constraints, and a larger average size of the learned

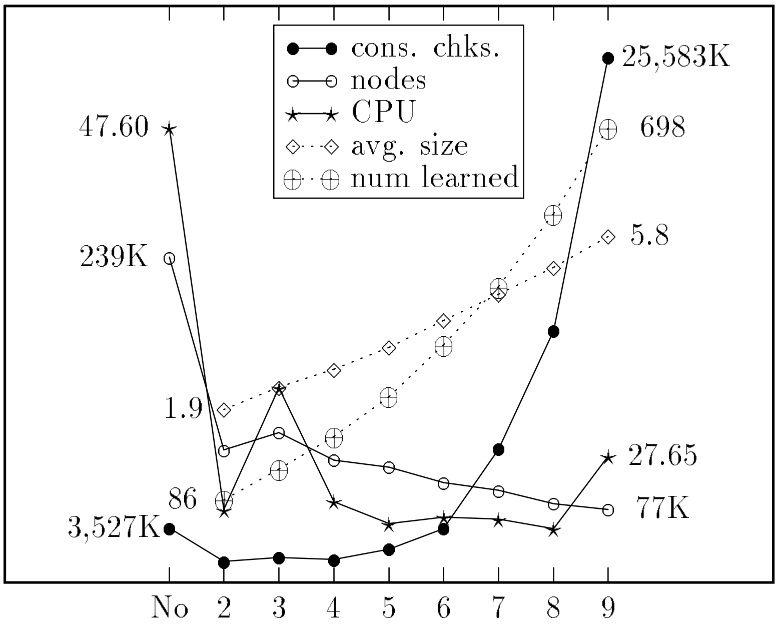


Figure 7.9: Results from experiments with parameters $\langle 125, 6, .0395, .444\rangle$ and varying orders of jump-back learning, as indicated on the x-axis. "No" signifies BJ+DVO without learning. Note that each statistic is on a different scale.

constraints. Although this relationship usually holds true on average, it is not necessarily the case on an instance by instance basis, since new constraints affect the dynamic variable ordering and occasionally produces a worse ordering. Charting the CPU time and number of consistency checks produces a U-shaped curve in respect to the learning order: a low order (2 – 4 for the problems with T=.333 and 2 – 8 for the problems with T=.444) reduces the average CPU time required, but if the order of learning is too high learning impairs the performance of BJ+DVO. This pattern holds true on many other sets of data we have examined: learning is best when limited. When the constraints are tighter in the original CSP, the optimum order of learning is higher. The problems with T=.444 in Fig. 7.9 are solved with in 23.23 CPU seconds, on average, with 8th-order learning, although this is perhaps negligibly lower than 24.40 CPU seconds with 2nd-order learning.

When the constraints are quite loose, the order of learning can have little effect. For example, on problems with D=3 and T=.111=1/9 almost all learned constraints have exactly 3 variables, since each of the three values of the dead-end variable is incompatible with a different previous variable. After non-binary constraints are added to the problem, it is possible for a dead-end to have a conflict set of more or less than 3 variables. Thus 2nd-order learning on such problems results in no learning, while learning without an order restriction is almost identical to 3rd-order learning.

7.5.3 Increasing the number of variables

Learning tends to have a greater impact on harder problems; an easy problem with relatively few dead-ends presents few opportunities for learning to come into play. To verify that learning is more beneficial on larger problems, we ran an experiment on six sets of random problems, where the number of variables varied from 50 in the first set to 300 in the sixth. The results are reported in Fig. 7.10. Only data from unsolvable problems is included so that estimates of the lognormal distribution's μ and σ parameters can be shown (see Chapter 3), but the growth in mean consistency checks for solvable problems is similar.

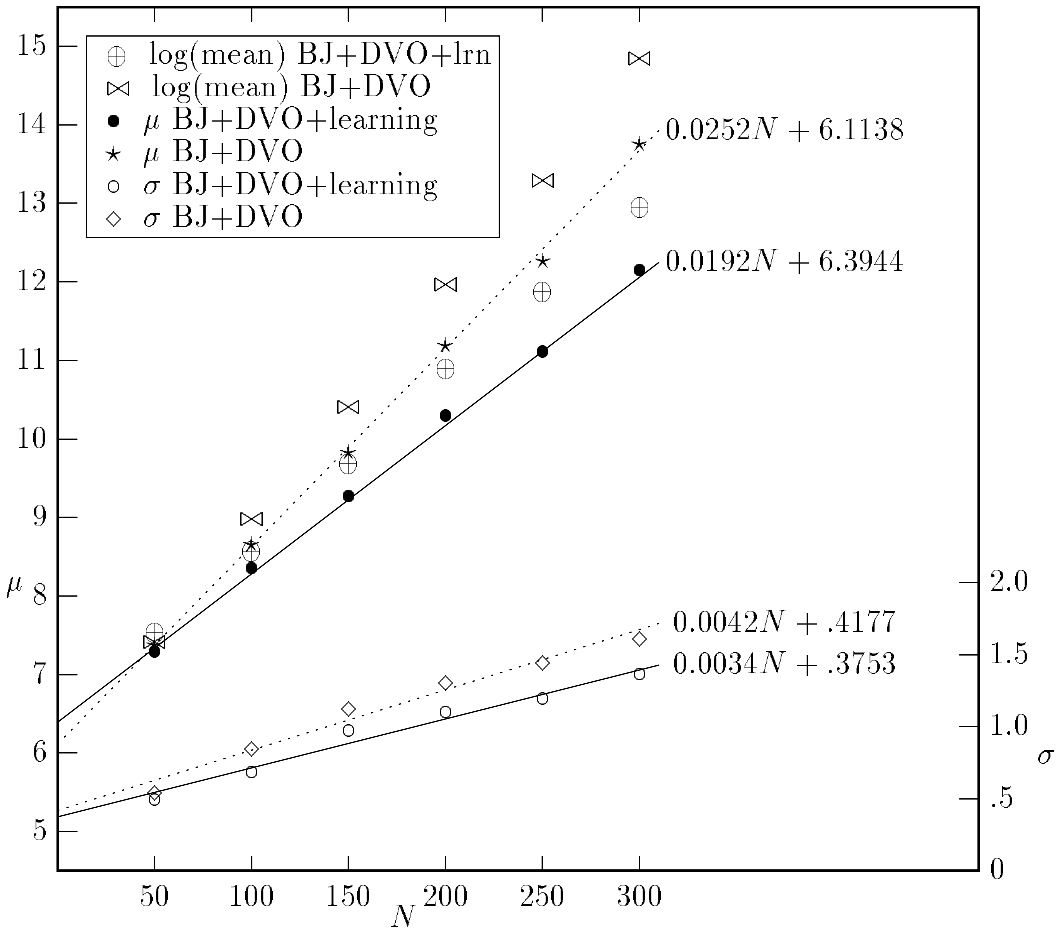


Figure 7.10: Data on number of consistency checks required on unsolvable problems in experiments with parameters D=3, T=.222, N varying from 50 to 300, and C set to the cross-over point, using BJ+DVO with and without 4th-order learning. Points represent estimated μ (left hand scale) and σ (right hand scale) for each algorithm, assuming a lognormal distribution. Lines (solid for BJ+DVO with learning, dotted for BJ+DVO without) show best linear fit. The formula to the right of each line shows the slope and the y-axis intercept. The graph also shows the natural logarithm of the mean number of consistency checks (left hand scale).

	Mean CPU seconds		
Decile	No learning	Learning	Ratio
1	951.09	67.46	14.10
2	333.70	52.22	6.39
3	159.94	27.72	5.77
4	100.43	19.70	5.10
5	65.92	7.76	8.49
6	43.35	5.85	7.41
7	28.63	5.17	5.54
8	12.55	2.38	5.27
9	6.30	1.35	4.66
10	1.57	0.56	2.79

Figure 7.11: Results from experiments comparing BJ+DVO with and without learning on random instances with parameters $\langle 300, 2, .0089, .333\rangle$. Mean CPU time for each decile of the data is reported.

The results in Fig. 7.10 are similar to those in Fig. 4.12 from Chapter 4. The presence of learning slows the growth in both μ and σ, so that the distribution of BJ+DVO is less skewed with learning than without. Fig. 7.11 shows the CPU times from the largest set of problems in Fig. 7.10, those with $N = 300$. The 500 instances, both solvable and unsolvable, have been divided into 10 groups of 50, based on the CPU time required by BJ+DVO without learning to solve each instance. The hardest 50 were put in the first group, the next hardest in the second group, and so on. On the hardest instances, the improvement due to learning was a factor of 14; for the easiest 50 problems learning still helps, but only by a factor of about three. As expected, learning is more effective on problem instances that have more dead-ends and larger search spaces, where there are more opportunities for each learned constraint to be useful.

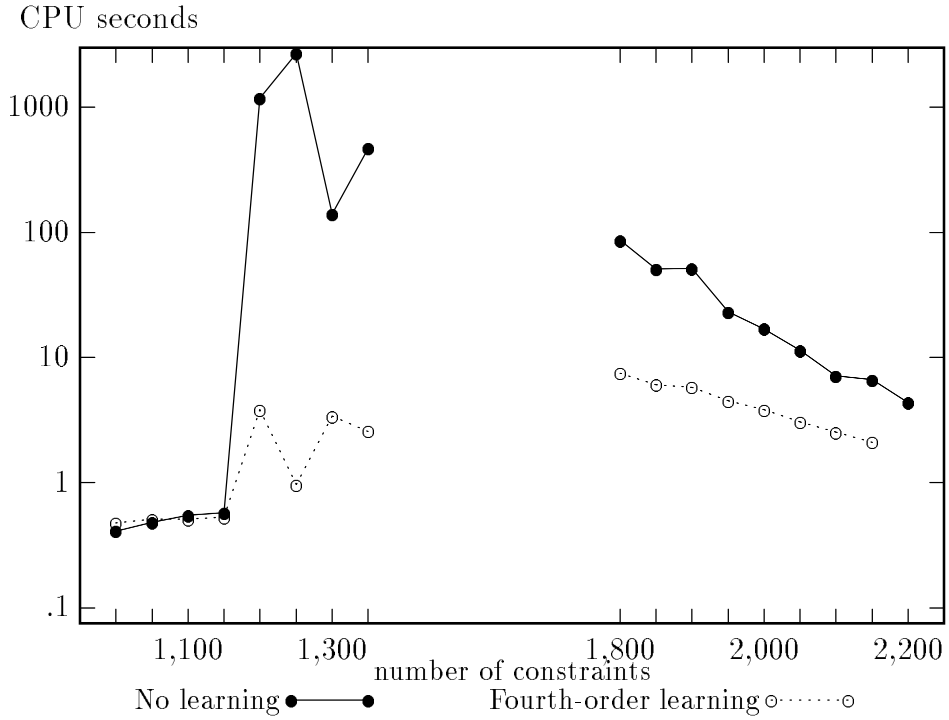


Figure 7.12: BJ+DVO without learning and with third-order learning, for N=500, D=3, T=.222, and non-crossover point values of C. All problems with fewer than 1,350 constraints were solvable; all problems with more than 1,800 had no solution.

7.5.4 Problems not at the cross-over point

Fig. 7.12 shows that with large enough N, problems do not have to be drawn from the 50% satisfiable area in order to be hard enough for learning to help. We experimented with problems not at the cross-over point by setting N=500, D=3, T=.222, and selecting values of C that are larger or smaller than the estimated cross-over point of .0123 (1,534 constraints). 250 instances were generated with each set of parameters. Learning was especially valuable on extremely hard solvable problems generated by slightly underconstrained values for C. For instance, at $\langle 500, 3, .0100, .222\rangle$, the hardest problem took 47 CPU hours without learning, and under one CPU minute with learning. The next four hardest problems took 4% as much CPU time with learning as without.

7.6 Average-case Space Requirements

It is worth noting that we did not find the space requirements of learning to be overwhelming, as has been reported by some researchers. For instance, even with a learning order of 9, only 698 constraints were learned, on average, for reasonably hard problems with parameters $\langle 125, 6, .0395, .444 \rangle$ (see Fig. 7.9). On the hardest problem in the set, which took 86 CPU minutes and made over two and a half billion consistency checks, 10,524 new constraints were added, with an average size of 7.37. Allowing 500 bytes to store such a large constraint and index it by each included variable-value pair, the run-time addition to computer memory is about five megabytes. Such an amount is not trivial, but does not pose a problem for modern computer systems. Furthermore, the order of learning can be limited to control the space used, if necessary. We have found that computer memory is not the limiting factor; time is.

7.7 Conclusions

We have introduced a new variant of learning, called jump-back learning, which is more powerful than previous versions because it takes advantage of processing already performed by the conflict-directed backjumping algorithm. Our experiments show that it is very effective when augmented on top of the strong BJ+DVO version of backjumping, resulting in at least an order of magnitude reduction in CPU time for some problems.

Learning seems to be particularly effective when applied to instances that are large or hard, since it requires many dead-ends to be able to augment the initial problem in a significant way. However, on easy problems with few dead-ends, learning will add little if any cost, thus perhaps making it particularly suitable for situations in which there is a wide variation in the hardness of individual problems. In this way learning is superior to other CSP techniques which modify the initial problem, such as by enforcing a certain order of consistency, since the cost will not be incurred on very easy problems.

An important parameter when applying learning is the order, or maximum size of the constraints learned. With no restriction on the order, it is possible to learn very large constraints that will be unlikely to prune the remaining search space.

Chapter 8

Comparison and Synthesis

8.1 Overview of Chapter

The previous chapters have described and evaluated a number of CSP algorithms. Each was shown to have interesting properties and to be useful on certain problems. This chapter reports several experiments designed to draw together the earlier results into a single coherent picture. BT+DVO, BJ+DVO, BT+DVO+IAC, BJ+DVO+LVO, and BJ+DVO+Learning are compared on both random problems and on a suite of problems from the Second DIMACS Implementation Challenge. We also introduce a new algorithms, BJ+DVO+LRN+LVO, which adds both jump-back learning and look-ahead value ordering to BJ+DVO.

8.2 Combining Learning and LVO

The look-ahead value ordering heuristic from Chapter 6 tentatively instantiates each value of the current variable and uses forward checking style look-ahead to gauge the value's impact on the remaining search space. The jump-back learning algorithm described in Chapter 7 records as constraints the conflict sets built by conflict-directed backjumping and used when a dead-end is encountered. These two orthogonal improvements to the BJ+DVO algorithm can be combined into a single algorithm, which we call BJ+DVO+LRN+LVO. The algorithm is described in Fig. 8.1.

BJ+DVO+LRN+LVO is modeled on BJ+DVO+LVO. The primary difference lies in step 1A (b). In learning algorithms without LVO, an opportunity to learn occurs at each dead-end. In the presence of LVO, the algorithm can learn a new nogood for each value of the current variable which will lead to a dead-end. For instance, suppose the current variable X_{50} has three values in its domain, $\{a, b, c\}$. The LVO mechanism in step 1A records the impact that $X_{50}{=}a$, $X_{50}{=}b$, and $X_{50}{=}c$, each have on future variable domains. If it happens that each value causes a future domain to be empty, then three nogoods will be recorded, differing only in their value for X_{50}.

8.3 Experiments on Large Random Problems

The first set of experiments reported in this chapter were based on the same Model B random problem generator used elsewhere in this thesis. Because the best algorithms from earlier chapters are compared in this experiment, it was possible to test them on larger problems than previously reported. We briefly review the algorithms compared in this chapter:

- BT+DVO. Backtracking with a dynamic variable ordering heuristic.
- BT+DVO+IAC. Backtracking with a dynamic variable ordering heuristic and arc-consistency performed after each instantiation. Arc-consistency is achieved by the AC-3 algorithm, using the AC-DC domain checking technique described in Chapter 4.
- BJ+DVO. Conflict-directed backjumping with a forward checking look-ahead and dynamic variable ordering.
- BJ+DVO+LVO. The BJ+DVO algorithm with the look-ahead value ordering heuristic described in Chapter 5.

Backjumping with DVO and Learning and LVO

Input: *order*

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur to be the index of the next variable, determined according to a VARIABLE-ORDERING-HEURISTIC. Set $P_{cur} \leftarrow \emptyset$.

1A. (Look-ahead value ordering.) Rank the values in D'_{cur} as follows:

   (a) For each value x in D'_{cur}, and for each value v of a future variables $X_i, cur < i \leq n$, determine the consistency of $(\vec{x}_{cur-1}, X_{cur}{=}x, X_i{=}v)$.

   (b) If instantiating a value x leads to an empty domain of some future variable X_f, then add $X_{cur}{=}x$ to P_f and learn by recording P_f as a new constraint, unless its size is greater than *order*.

   (c) Using a heuristic function, compute the rank of x based on the number and distribution of conflicts with future values v.

2. Select a value $x \in D'_{cur}$. Do this as follows:

   (a) If $D'_{cur} = \emptyset$, go to 3.

   (b) Pop the highest ranked value x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.

   (c) (This step can be avoided by caching the results from step 1A.) Examine the future variables $X_i, cur < i \leq n$. For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from $D`_i$ and add X_{cur} to P_i; if D'_i becomes empty, go to (d) (without examining other X_i's).

   (d) Go to 1.

3. (Backjump.) If $P_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent." Otherwise, set $P \leftarrow P_{cur}$; set cur equal to the index of the last variable in P. Set $P_{cur} \leftarrow P_{cur} \cup P - \{X_{cur}\}$. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 8.1: Algorithm BJ+DVO+LRN+LVO, which combines backjumping, dynamic variable ordering, jump-back learning, and look-ahead value ordering.

Parameters	Algorithm	CC	Nodes	CPU
$\langle 200, 3, 0.0592, 0.111\rangle$	BT+DVO	5,871,215	207,183	68.46
	BT+DVO+IAC	23,836,368	40,098	55.44
	BJ+DVO	5,365,467	188,726	69.28
	BJ+DVO+LVO	4,793,417	167,211	73.78
	BJ+DVO+LRN	5,731,244	186,582	63.39
	BJ+DVO+LRN+LVO	5,622,825	159,739	74.00
$\langle 300, 3, 0.0206, 0.222\rangle$	**BT+DVO+IAC**	141,606	632	0.65
	BJ+DVO	2,483,520	222,285	119.26
	BJ+DVO+LVO	1,623,455	131,593	86.76
	BJ+DVO+LRN	419,193	32,221	15.62
	BJ+DVO+LRN+LVO	392,606	25,771	13.98
$\langle 350, 3, 0.0089, 0.333\rangle$	**BT+DVO+IAC**	24,641	494	0.43
	BJ+DVO	1,238,479	182,328	140.81
	BJ+DVO+LVO	969,224	118,854	111.79
	BJ+DVO+LRN	3,727	464	0.46
	BJ+DVO+LRN+LVO	22,688	1,036	3.78

Table 8.1: Comparison of five algorithms on random CSPs with $D = 3$. Each number is the mean of 2,000 solvable and unsolvable instances. The algorithm with the lowest mean CPU seconds in each group is in boldface.

- BJ+DVO+LRN. BJ+DVO with jump-back learning, as defined in Chapter 7. Only nogoods with four or fewer variables are learned, and learning is only performed at leaf dead-ends.
- BJ+DVO+LRN+LVO. BJ+DVO with fourth-order jump-back learning and look-ahead value ordering, as described in the previous section.

Various combinations of parameters N, D, C, and T, were selected, all near the cross-over region where 50% of the problem have solutions. For each set of parameters were generated 2,000 instances and applied the algorithms to each instance. Experiments reported in Chapter 4 demonstrated that on large problems with tight constraints, BT+DVO does not perform well. We therefore applied that algorithm only to problems with relatively loose constraints, specifically parameter combinations $\langle 200, 3, 0.0592, 0.111\rangle$, $\langle 60, 6, 0.4797, 0.111\rangle$, and $\langle 75, 6, 0.1744, 0.222\rangle$. The results of the experiments are reported in Table 8.1 and Table 8.2.

Parameters	Algorithm	CC	Nodes	CPU
$\langle 60, 6, 0.4797, 0.111\rangle$	BT+DVO	24,503,115	412,494	59.72
	BT+DVO+IAC	104,319,923	65,432	130.54
	BJ+DVO	24,228,726	407,253	63.10
	BJ+DVO+LVO	23,904,430	401,131	66.47
	BJ+DVO+LRN	24,368,062	406,332	57.43
	BJ+DVO+LRN+LVO	24,103,544	405,899	65.75
$\langle 75, 6, 0.1744, 0.222\rangle$	BT+DVO	7,766,594	249,603	40.77
	BT+DVO+IAC	18,419,580	16,395	22.22
	BJ+DVO	7,530,726	241,124	42.67
	BJ+DVO+LVO	7,228,548	230,073	44.05
	BJ+DVO+LRN	7,856,321	230,367	42.13
	BJ+DVO+LRN+LVO	7,321,890	231,455	43.79
$\langle 100, 6, 0.0772, 0.333\rangle$	**BT+DVO+IAC**	4,718,685	4,625	5.67
	BJ+DVO	6,248,608	293,922	67.30
	BJ+DVO+LVO	6,581,314	305,121	76.49
	BJ+DVO+LRN	5,979,767	232,780	54.34
	BJ+DVO+LRN+LVO	6,034,538	235,509	61.22
$\langle 125, 6, 0.0395, 0.444\rangle$	**BT+DVO+IAC**	479,228	566	0.60
	BJ+DVO	3,526,619	238,584	66.17
	BJ+DVO+LVO	3,007,791	195,720	62.54
	BJ+DVO+LRN	2,050,232	108,482	28.80
	BJ+DVO+LRN+LVO	1,970,645	102,788	29.45
$\langle 150, 6, 0.0209, 0.555\rangle$	**BT+DVO+IAC**	32,537	111	0.06
	BJ+DVO	3,253,255	359,095	111.70
	BJ+DVO+LVO	1,328,189	124,415	47.08
	BJ+DVO+LRN	339,191	28,056	8.25
	BJ+DVO+LRN+LVO	601,454	25,769	12.87

Table 8.2: Comparison of five algorithms on random problems with D=6. Each number is the mean of 2,000 solvable and unsolvable instances. The algorithm with the lowest mean CPU seconds in each group is in boldface.

We also present graphically some of the data from Tables 8.1 and 8.2, which permits more information than just averages to be conveyed. Fig. 8.2 shows the distribution of consistency checks made and nodes expanded for three algorithms on unsatisfiable problems with parameters $\langle 350, 3, 0.0089, 0.333\rangle$. The distributions are approximated by lognormal curves with estimated μ and σ parameters. Fig. 8.3 is a scatter chart, in which each point indicates the relative performance of BT+DVO+IAC and BJ+DVO+LRN on a problem with parameters $\langle 75, 6, 0.1744, 0.222\rangle$.

8.4 Experiments with DIMACS Problems

The Second Dimacs Implementation Challenge in 1993 [44] collected a set of satisfiability problem instances for the purpose of providing benchmarks for comparison of algorithms and heuristics. We compared our algorithms against six of the problems that were derived from circuit fault analysis. The problems are encoded as Boolean satisfiability problems in conjunctive normal form. Clauses contain from one to six variables.

Each of our six algorithms was applied to these benchmark problems. The results are displayed in Table 8.3 and Table 8.4. The tables show other CPU times on these problems reported in [44]. Dubois *et al.* [21] uses a complete algorithm based on the Davis-Putnam procedure; computer is a Sun SparcStation 10 model 40. Hampson and Kibler [39] use a randomized hill climbing procedure; computer is a Sun SparcStation II. Jaumard *et al.* [43] use a complete Davis-Putnam based algorithm with a tabu search heuristic; computer is a Sun SparcStation 10 model 30. Pretolani's H2R algorithm [67] is based on the Davis-Putnam procedure and uses a pruning heuristic; computer is a Sun SparcStation 2. Resende and Feo [72] present a greedy randomized adaptive search procedure called GRASP-A; the computer used was not reported. Spears [83] uses a simulated annealing based algorithm; computer is a Sun SparcStation 10. Van Gelder and Tsuji [87] use

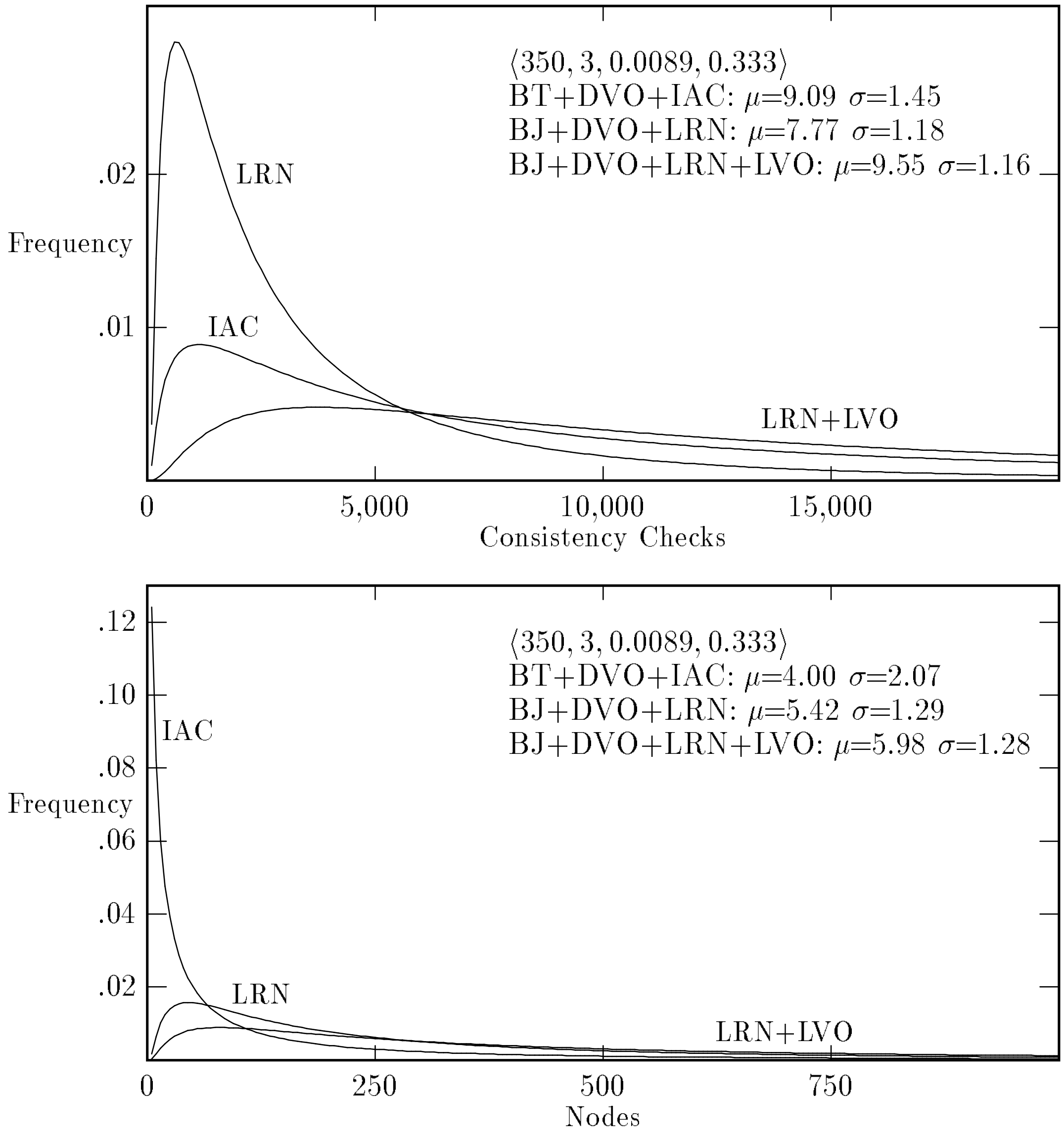


Figure 8.2: Lognormal curves based on unsolvable problems generated from parameters $\langle 350, 3, 0.0089, 0.333\rangle$. The top graph is based on consistency checks, the bottom graph on search space nodes. μ and σ parameters were estimated using the Maximum Likelihood Estimator (see Chapter 3).

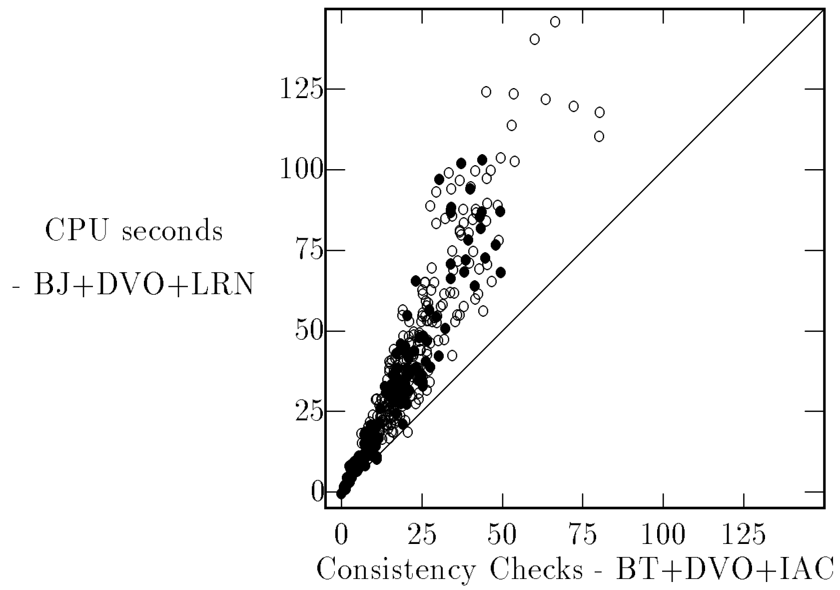


Figure 8.3: Each point (• = has solution, ∘ = no solution) represents one instance from the experiment with $\langle 75, 6, 0.1744, 0.222 \rangle$. Points above the diagonal line required less CPU time with BT+DVO+IAC than with BJ+DVO+LRN.

a complete algorithm that combines search and resolution; computer is a Sun SparcStation 10 model 41.

Among our six algorithms, no clear trend is discernible. BT+DVO+IAC had the best CPU time on three problems, including one tie with BJ+DVO+LRN, and BJ+DVO+LRN was best on two, including the tie. BJ+DVO+LVO and BJ+LVO+LRN+LVO were each best one on problem. On the hardest problem, `ssa2670-141`, we cancelled BT+DVO after 24 CPU hours had passed without the algorithm completing.

8.5 Discussion

The experiments in this chapter confirm and amplify the results reported in earlier chapters. The additional effort expended by enforcing arc-consistency after each instantiation often paid off in the form of a sharply reduced search space and

Problem	Algorithm	CC	Nodes	CPU
ssa0432-003	BT+DVO	51,190	901	0.73
435 variables	BT+DVO+IAC	73,817	512	0.70
1,027 clauses	BJ+DVO	48,811	865	0.81
unsatisfiable	BJ+DVO+LVO	69,100	823	0.92
	BJ+DVO+LRN	52,505	827	0.71
	BJ+DVO+LRN+LVO	59,091	816	0.78
	Dubois			1.40
	Jaumard			9.00
	Pretolani			0.83
	Van Gelder			0.55
	Wallace			499.30
ssa2670-141	BT+DVO			
1,359 variables	BT+DVO+IAC	535,875,109	1,279,009	1,943.51
3,321 clauses	BJ+DVO	173,446,699	7,117,071	2,791.01
unsatisfiable	BJ+DVO+LVO	41,073,083	1,858,408	803.81
	BJ+DVO+LRN	35,689,610	1,036,554	488.25
	BJ+DVO+LRN+LVO	31,854,918	843,099	449.76
	Dubois			2,674.40
	Van Gelder			164.58
ssa7552-038	BT+DVO	755,034	45,796	14.52
1,501 variables	BT+DVO+IAC	1,274,887	3,766	3.51
3,575 clauses	BJ+DVO	687,122	40,008	12.17
satisfiable	BJ+DVO+LVO	578,909	31,899	11.01
	BJ+DVO+LRN	439,755	22,884	5.50
	BJ+DVO+LRN+LVO	398,541	16,001	3.78
	Dubois			1.20
	Pretolani			3.67
	Hampson			152.2
	Resende			8.31
	Van Gelder			1.85

Table 8.3: Comparison of five algorithms on DIMACS problems. The names refer to authors who participated in the DIMACS challenge; references are given in the text. Numbers for our algorithms are all results from single instances. Some CPU times from other authors are averages over multiple randomized runs on the problem.

Problem	Algorithm	CC	Nodes	CPU
`ssa7552-158`	BT+DVO	1,009,736	51,756	19.98
1,363 variables	BT+DVO+IAC	1,863,152	17,938	8.25
3,034 clauses	BJ+DVO	845,991	25,611	10.72
satisfiable	BJ+DVO+LVO	445,172	8,122	4.55
	BJ+DVO+LRN	612,791	19,088	8.64
	BJ+DVO+LRN+LVO	467,890	12,876	7.07
	Dubois			0.80
	Hampson			82.50
	Jaumard			43.00
	Pretolani			2.28
	Resende			2.42
	Van Gelder			1.14
`ssa7552-159`	BT+DVO	883,614	39,110	14.45
1,363 variables	BT+DVO+IAC	1,378,253	4,167	3.20
3,032 clauses	BJ+DVO	674,091	20,093	6.07
satisfiable	BJ+DVO+LVO	691,654	18,987	6.98
	BJ+DVO+LRN	503,122	12,077	3.20
	BJ+DVO+LRN+LVO	563,871	12,890	3.67
	Dubois			0.90
	Hampson			82.30
	Jaumard			6.00
	Pretolani			2.68
	Resende			1.63
	Van Gelder			1.14
`ssa7552-160`	BT+DVO	712,009	35,877	12.92
1,391 variables	BT+DVO+IAC	1,265,887	4,098	3.02
3,126 clauses	BJ+DVO	687,833	22,088	6.28
satisfiable	BJ+DVO+LVO	792,615	23,766	7.14
	BJ+DVO+LRN	453,788	9,745	3.67
	BJ+DVO+LRN+LVO	495,166	10,687	4.50
	Dubois			0.90
	Hampson			86.00
	Jaumard			6.00
	Pretolani			2.80
	Resende			22.79
	Van Gelder			1.44

Table 8.4: Continuation of Table 8.3.

great savings in CPU time. This pattern was most pronounced on problems with tight constraints. Look-ahead value ordering and jump-back learning both tended to improve substantially the efficacy of BJ+DVO. These enhancements too were more effective on problems with tight constraints. Combining jump-back learning and LVO into BJ+DVO+LRN+LVO did not tend to be particularly useful: in only one experiment with random problems, based on parameters $\langle 300, 3, 0.0206, 0.222 \rangle$, was this combination superior in CPU time to the other BJ+DVO variants. On the other hand, BJ+DVO+LRN+LVO showed the best performance on the hardest DIMACS Challenge problem, which suggests that the combination can be useful on problems of sufficient size.

The effectiveness of interleaving arc-consistency was partly contingent on the relatively low cost of performing a consistency check in our program. In the experiment with parameters $\langle 200, 3, 0.0592, 0.111 \rangle$, for instance, BT+DVO+IAC performed about 430,000 consistency checks per CPU second (23,836,368 / 55.44). If consistency checking had been more expensive, the ranking of BT+DVO+IAC relative to the other algorithms might have changed. Indeed, BT+DVO+IAC was not quite such a strong performer on the DIMACS problems, which have a large number on non-binary constraints. In our implementation, binary constraints were stored in a table, which permits fast look-up, while higher order constraints were stored in lists, to which access is slower.

8.6 Conclusions

We have shown that the trends observed in earlier chapters continue to hold both for larger random problems and for a set of satisfiability problems drawn from circuit analysis. Enforcing arc-consistency after each variable instantiation improved the performance of BT+DVO by orders of magnitude on problems with relatively tight constraints. Jump-back learning and look-ahead value ordering, both individually and in combination, substantially improved the performance of BJ+DVO.

Chapter 9

Encoding Maintenance Scheduling Problems as CSPs

9.1 Overview of Chapter

This chapter focusses on a well-studied problem of the electric power industry: optimally scheduling preventative maintenance of power generating units within a power plant. We define a formal model which captures most of the interesting characteristics of these problems, and then show how the model can be cast as a constraint satisfaction problem. Because maintenance scheduling is an optimization problem, we use a series of CSPs with ever-tightening constraints to discover a locally optimal schedule. Empirical results show that applying the learning algorithm from Chapter 7 significantly reduces the CPU time required to solve this series of maintenance scheduling CSPs.

9.2 Introduction

The problem of scheduling off-line preventative maintenance of power generating units is of critical interest to the electric power industry. A typical power plant consists of one or two dozen power generating units which can be individually scheduled for maintenance. Both the required duration of each unit's preventative maintenance and a reasonably accurate estimate of the power demand that the plant will be required to meet throughout the planning period are known in advance. The general purpose of determining a *maintenance schedule* is to determine the duration and sequence of outages of power generating units over a given time period, while minimizing operating and maintenance costs over the planning period, subject to various constraints. A maintenance schedule is often prepared in advance for a year at a time, and scheduling is done most frequently on a week-by-week basis. The power industry generally considers shorter term scheduling, up to a period of one or two weeks into the future, to be a separate problem called "unit commitment."

Computational approaches to maintenance scheduling have been intensively studied since the mid 1970's. Dopazo and Merrill [20] formulated the maintenance scheduling problem as a 0-1 integer linear program. Zurm and Quintana [93] used a dynamic programming approach. Egan [22] studied a branch and bound technique. More recently, techniques such as simulated annealing, artificial neural networks, genetic algorithms, and tabu search [45] have been applied.

This chapter reports the results of applying the constraint processing techniques developed in the earlier chapters to the maintenance scheduling problem. The task is of interest for several reasons. Primary, of course, is the opportunity to take our research results "out of the lab," and to put them to use on problems of substantial economic interest. Within the context of evaluating CSP algorithms, applying the algorithms to maintenance scheduling-based problems provides a testbed of problem instances that have an interesting structure and non-binary constraints. Our preliminary empirical results indicate that algorithms which are superior on random uniform binary CSPs are also superior on maintenance scheduling problems, providing some validation of the empirical approach in the earlier parts of this dissertation.

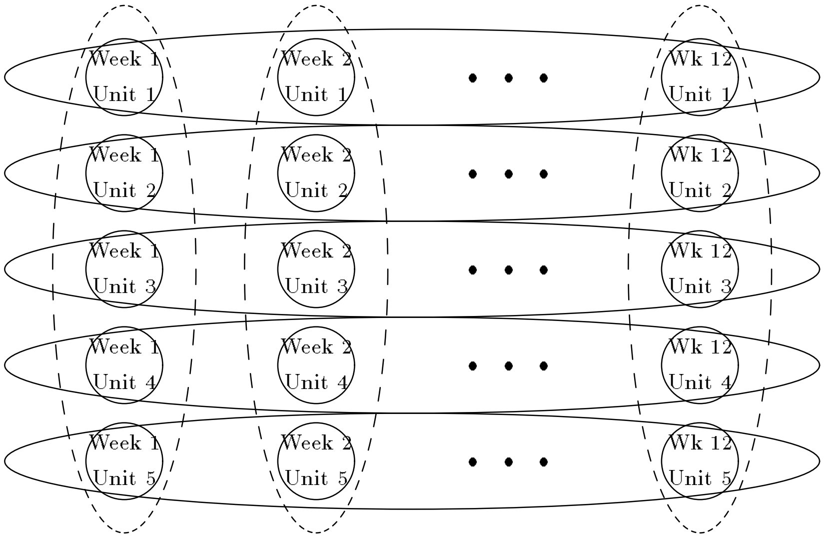


Figure 9.1: A diagrammatic representation of a maintenance scheduling constraint satisfaction problem. Each circle stands for a variable representing the status of one unit in one week. The dashed vertical ovals indicate constraints between all of the units in one week: meeting the minimum power demand and optimizing the cost per week. The horizontal ovals represent constraints on one unit over the entire period: scheduling an adequate period for maintenance.

We also present a new use for learning. The constraint framework we use consists entirely of so-called *hard* constraints, those which must be satisfied for a solution to be valid. Optimization problems sometimes make use of *soft* constraints, which can be partially satisfied. To avoid introducing soft constraints, we approach optimization as solving a series of related CSPs, each consisting solely of hard constraints. The CSPs in the series differ in that the constraint being optimized is tighter in each succeeding problem in the series. The tighter constraints results from a reduced cost bound in the function being optimized. Constraints learned during one instance of the series can be applied again on later instances.

9.3 The Maintenance Scheduling Problem

As a problem for an electric power plant operator, maintenance scheduling must take into consideration such complexities as local holidays, weather patterns, constraints on suppliers and contractors, national and local laws and regulations, and other factors that are germane only to a particular power plant. We have developed a slightly simplified model of the maintenance scheduling problem. Our model is similar to those appearing in most scholarly articles, and follows closely the approach of Yellen and his co-authors [2, 92]. The maintenance scheduling problem can be represented by a rectangular matrix (see Fig. 9.1). Each entry in the matrix represents the status of one generating unit for one week. (Since the time minimum period considered is almost always the week, we will use the terms week and time period interchangeably.) A unit can be in one of three states: ON, OFF, or MAINT.

9.3.1 Parameters

A specific maintenance scheduling problem, in our formulation, is defined by a set of parameters, which are listed in Fig. 9.2. Parameters U, the number of units, and W, the number of weeks, control the size of the schedule. Many power plants have a fixed number of crews which are available to carry out the maintenance; therefore the parameter M specifies the maximum number of units which can be undergoing maintenance at any one time.

In this paragraph and elsewhere in the chapter we adopt the convention of quantifying the subscript i over the number of units, $1 \leq i \leq U$, and the subscript t over the number of weeks, $1 \leq t \leq W$. Several parameters specify the characteristics of the power generating units. The costs involved in preventative maintenance, m_{it}, can vary from unit to unit and from week to week; for instance, hydroelectric units are cheaper to maintain during periods of low water flow. The predicted operating cost of unit i in week t is given by c_{it}. This quantity varies by type of unit and also in response to fuel costs. For example, the fuel costs of nuclear units are low and change little over the year, while oil-fired units are typically more expensive to operate in the winter, when oil prices often increase.

	Input:
U	number of power generating units
W	number of weeks to be scheduled
M	maximum number of units which can be maintained simultaneously
m_{it}	cost of maintaining unit i in period t
c_{it}	operating cost of unit i in period t
k_i	power output capacity of unit i
e_i	earliest maintenance start time for unit i
l_i	latest maintenance start time for unit i
d_i	duration of maintenance for unit i
N	set of pairs of units which cannot be maintained simultaneously
D_t	energy (output) demand in period t
	Output:
x_{it}	status of unit i in period t: ON, OFF or MAINT

Figure 9.2: Parameters which define a specific maintenance scheduling problem.

Parameter k_i specifies the maximum power output of unit i. Most formulations of maintenance scheduling consider this quantity constant over time, although in reality it can fluctuate, particularly for hydro-electric units.

The permissible window for scheduling the maintenance of a unit is controlled by parameters e_i, the earliest starting time, and l_i, the latest allowed starting time. These parameters are often not utilized (that is, e_i is set to 1 and l_i is set to W) because maintenance can be performed at any time. However, the maintenance window can be used to prevent hydro-electric power plants from being maintained during periods of high water flow, or for accommodating holiday and vacation seasons. The duration of maintenance is specified by parameter d_i.

Sometimes the maintenance of two particular units cannot be allowed to overlap, since they both require a particular unique resource, perhaps a piece of equipment or a highly trained crew member. Such incompatible pairs of units are specified in the set $N = \{(i_1, i_2), \ldots, (i_{n-1}, i_n)\}$.

The final input parameter, D_t, is the predicted power demand on the plant in each week t.

The parameters x_{it} are the output of the scheduling procedure, and define the maintenance schedule. x_{it} can take on one of three values:

- ON: unit i is on for week t, can deliver k_i power for the week, and will cost c_{it} to run;
- OFF: unit i is off for week t, will deliver no power and will not result in any cost;
- MAINT: unit i is being maintained for week t, will deliver no power, and will cost m_{it}.

It is worth pointing out that generating units can often be operated at any level of output between zero power and full power, with a corresponding decrease in the cost of operating the unit. Some maintenance scheduling systems schedule all non-maintained units as on, and assume that to meet the demand of each period, units which are not on maintenance pick up power in ascending order of fuel cost [45]. However, determining operating levels is usually not considered part of maintenance scheduling, and a two- or three-value approach such as we have adopted (e.g. ON, OFF, MAINT) is more widely followed.

9.3.2 Constraints

A valid maintenance schedule must meet the following constraints or domain requirements, which arise naturally from the definition and intent of the parameters.

First, the schedule must permit the overall power demand of the plant to be met for each week. Thus the sum of the power output capacity of all units not scheduled for maintenance must be greater than the predicted demand, for each week. Let $z_{it} = 1$ if $x_{it} = \text{ON}$, and 0 otherwise. Then the schedule must satisfy the following inequalities.

$$\sum_i z_{it} k_i \geq D_t \quad \text{for each time period } t \tag{9.1}$$

The second constraint is that maintenance must start and be completed within the prescribed window, and the single maintenance period must be continuous, uninterrupted, and of the desired length. The following conditions must hold true for each unit i.

$$\begin{array}{rll}
\text{(start)} & \text{if } t < e_i \text{ then } x_{it} \neq \textsc{maint} & (9.2) \\
\text{(end)} & \text{if } t \geq l_i + d_i \text{ then } x_{it} \neq \textsc{maint} & (9.3) \\
\text{(continuous)} & \text{if } x_{it_1} = \textsc{maint} \text{ and } x_{it_2} = \textsc{maint} \text{ and } t_1 < t_2 & \\
 & \text{then for all } t, t_1 < t < t_2,\ x_{it} = \textsc{maint} & (9.4) \\
\text{(length)} & \text{if } t_1 = \min_t(x_{it} = \textsc{maint}) \text{ and } t_2 = \max_t(x_{it} = \textsc{maint}) & \\
 & \text{then } t_2 - t_1 + 1 = d_i & (9.5) \\
\text{(existence)} & \exists t \text{ such that } x_{it} = \textsc{maint} & (9.6)
\end{array}$$

The third constraint is that no more than M units can be scheduled for maintenance simultaneously. Let $y_{it} = 1$ if $x_{it} = \textsc{maint}$, and 0 otherwise.

$$\sum_i y_{it} \leq M \quad \text{for each time period } t \tag{9.7}$$

The final constraint on a maintenance schedule is that incompatible pairs of units cannot be scheduled for simultaneous maintenance.

$$\text{if } (i_1, i_2) \in N \text{ and } x_{i_1 t} = \textsc{maint} \text{ then } x_{i_2 t} \neq \textsc{maint} \quad \text{for each time period } t \tag{9.8}$$

After meeting the above constraints, we want to find a schedule which minimizes the maintenance and operating costs during the planning period. Let $w_{it} = m_{it}$ if $x_{it} = \textsc{maint}$, c_{it} if $x_{it} = \textsc{on}$, and 0 if $x_{it} = \textsc{off}$.

$$\text{Minimize} \quad \sum_i \sum_t w_{it} \tag{9.9}$$

Objective functions other than (9.9) can also be used. For example, it may be necessary to reschedule the projected maintenance midway through the planning period. In this case, a new schedule which is as close as possible to the previous schedule may be desired, even if such a schedule does not have a minimal cost.

We have now stated precisely the parameters, constraints and optimization function that define a maintenance scheduling problem.

9.4 Formalizing Maintenance Problems as CSPs

Given the definition of the maintenance scheduling problem presented in the previous section, there exist several ways to encode the problem in the constraint satisfaction framework. Formalizing a maintenance scheduling problem as a constraint satisfaction problem entails deciding on the variables, the domains, and the constraints which will represent the requirements of the problem. The goal of course is to develop a scheme that is conducive to finding a solution – a schedule – speedily. A CSP in general will be easier to solve if it has a smaller number of variables, a smaller number of values per variable, and constraints of smaller arity. (The arity of a constraint is the number of variables it refers to.) Since these three conditions cannot be met simultaneously, compromises must be made to achieve a satisfactory representation as a constraint satisfaction problem.

Our system encodes maintenance scheduling problems as CSPs with $3 \times U \times W$ variables. The variables can be divided into a set of $U \times W$ visible variables, and two $U \times W$ size sets of hidden variables. Of course the distinction between visible and hidden variables is used for explanatory purposes only; the CSP solving program treats each variable in the same way. Each variable has two or three values in its domain. Both binary and higher arity constraints appear in the problem.

We label the visible variables X_{it}; they correspond directly to the output parameters x_{it} of the problem definition. Because i ranges from 1 to U and t ranges from 1 to W, there are $U \times W$ visible variables. Each X_{it} has the domain $\{$ON, OFF, MAINT$\}$, corresponding exactly to the values of x_{it}.

The first set of hidden variables, Y_{it}, signifies the maintenance status of unit i during week t. The domain of each Y variable is {FIRST, SUBSEQUENT, NOT}. $Y_{it} =$ FIRST indicates that week t is the beginning of unit i's maintenance period. $Y_{it} =$ SUBSEQUENT indicates that unit i is scheduled for maintenance during week t and for at least one prior week. If $Y_{it} =$ NOT, then maintenance for unit i is not planned during week t. The following constraint between each X_{it} and Y_{it} variable is required to keep the two variables synchronized (we list the compatible value combinations):

X_{it}	Y_{it}
ON	NOT
OFF	NOT
MAINT	FIRST
MAINT	SUBSEQUENT

The second set of hidden variables, Z_{it}, are boolean variables which indicate whether unit i is producing power output during week t. The two values for each Z variable are {NONE, FULL}, corresponding to no power output and full power output. Binary constraints between each X_{it} and the corresponding Z_{it} variable are as follows, again listing the legal combinations:

X_{it}	Z_{it}
ON	FULL
OFF	NONE
MAINT	NONE

The hidden variables triple the size of the CSP. The reasons for creating them will become clear as we now discuss how the constraints are implemented.

Constraint (9.1) – weekly power demand

Each demand constraint involves the U visible variables that relate to a particular week. The basic idea is to enforce a U-ary constraint between these variables which guarantees that enough of the variables will be ON to meet the power demand for the week. This constraint can be implemented as a table of either compatible or incompatible combinations, or as a procedure which takes as input the U variables and returns TRUE or FALSE. Our implementation uses a table of incompatible combinations. For example, suppose there are four generating units, with output capacities k_1=100, k_2=200, k_3=300, k_4=400. For week 5, the demand D_5=800. The following 4-ary constraint among variables $(Z_{1,5}, Z_{2,5}, Z_{3,5}, Z_{4,5})$ is created (incompatible tuples are listed).

$Z_{1,5}$	$Z_{2,5}$	$Z_{3,5}$	$Z_{4,5}$	comment (output level)
NONE	NONE	NONE	NONE	0
NONE	NONE	NONE	FULL	400
NONE	NONE	FULL	NONE	300
NONE	NONE	FULL	FULL	700
NONE	FULL	NONE	NONE	200
NONE	FULL	NONE	FULL	600
NONE	FULL	FULL	NONE	500
FULL	NONE	NONE	NONE	100
FULL	NONE	NONE	FULL	500
FULL	NONE	FULL	NONE	400
FULL	FULL	NONE	NONE	300
FULL	FULL	NONE	FULL	700
FULL	FULL	FULL	NONE	600

Because the domain size of the Z variables is 2, a U-ary constraint can have as many as $2^U - 1$ tuples. If this constraint were imposed on the X variables directly, which have domains of size 3, there would be 79 tuples $(3^4 - 5)$ instead of 13 $(2^4 - 3)$. This is one reason for creating the hidden Z variables: to reduce the size of the demand constraint.

A relation such as that in the above table may be *projected* onto a subset of its variables, by listing the combinations of values which are restricted to this subset. The relation's projection onto $(Z_{1,5}, Z_{2,5})$, for example, is

$Z_{1,5}$	$Z_{2,5}$
NONE	NONE
NONE	FULL
FULL	NONE
FULL	FULL

Tuples in the new, projected relation which appear with all possible combinations of the remaining variables in the original relation may be recorded as smaller constraints. That is, the binary constraint over the pair $(Z_{1,5}, Z_{2,5})$, allowing $(Z_{1,5}=$ NONE, $Z_{2,5}=$NONE) while the remaining tuples are not allowed, is implied by the 4-ary constraint. It is clearly desirable to recognize these smaller constraints prior to search, and our system does so. In effect, the system notices that if $Z_{1,5}$ is NONE and $Z_{2,5}$ is NONE, then the demand constraint for week 5 cannot be met, whatever the status of the other units.

Constraints (9.2) and (9.3) – earliest and latest maintenance start date

These constraints are easily implemented by removing the value FIRST from the domains of the appropriate Y variables. The removal of a domain value is often referred to as imposing a *unary* constraint.

Constraint (9.4) – continuous maintenance period

To encode this domain constraint in our formalism, we note that if the following three conditions hold, then maintenance will be for a continuous period:

1. There is only one first week of maintenance.
2. Week 1 cannot be a subsequent week of maintenance.
3. Every subsequent week of maintenance must be preceded by a first week of maintenance or a subsequent week of maintenance.

Each of these conditions can be enforced by unary or binary constraints on the Y variables. To enforce condition 1, for every unit i and pair of weeks t_1 and

$t_2, t_1 \neq t_2$, we add the following binary constraint to the CSP (disallowed tuple listed):

Y_{it_1}	Y_{it_2}
FIRST	FIRST

Condition 2 is enforced by a unary constraint removing SUBSEQUENT from the domain of each Y_{i1} variable. Condition 3 is enforced by the following constraint for all $t > 1$ (disallowed tuple listed):

Y_{it-1}	Y_{it}
NOT	SUBSEQUENT

Constraint (9.5) – length of maintenance period

A maintenance period of the correct length cannot be too short or too long. If unit i's maintenance length d_i=1, then too short is not possible (constraint (9.6) prevents non-existent maintenance periods); otherwise, for each unit i, each time period t, and every $t_1, t < t_1 < t + d_i$, the following binary constraint exists to prevent a short maintenance period (disallowed tuple listed):

Y_{it}	Y_{it_1}
FIRST	NOT

To ensure that too many weeks of maintenance are not scheduled, it is only necessary to prohibit a subsequent maintenance week in the first week that maintenance should have ended. This results in the following constraint for each i and t, letting $t_1 = t + d_i$ (disallowed tuple listed):

Y_{it}	Y_{it_1}
FIRST	SUBSEQUENT

Constraint (9.6) – existence of maintenance period

This requirement is enforced by a high arity constraint among the Y variables for each unit. Only the weeks between the earlist start week and the latest start week need be involved. At least one $Y_{it}, e_i \leq t \leq l_i$, must have the value START. It is simpler to prevent them from all having the value NOT, and let constraints (9.4) and (9.5) ensure that a proper maintenance period is established. Thus the $(l_i - e_i + 1)$-arity constraint for each unit i is (disallowed tuple listed):

Y_{il_i}	...	Y_{ie_i}
NOT	NOT	NOT

Constraint (9.7) – no more than M units maintained at once

If M units are scheduled for maintenance in a particular week, constraints must prevent the scheduling of an additional unit for maintenance during that week. Thus the CSP must have $(M + 1)$-ary constraints among the X variables which prevent any $M+1$ from having the value of MAINT in any given week. There will be $\binom{U}{M+1}$ of these constraints for each of the W weeks. They will have the form (disallowed tuple listed):

$X_{i_1 t}$	...	$X_{i_M t}$
MAINT	MAINT	MAINT

Constraint (9.8) – incompatible pairs of units

The requirement that certain units not be scheduled for overlapping maintenance is easily encoded in binary constraints. For every week t, and for every pair of units $(i_1, i_2) \in N$, the following binary constraint is created (incompatible pair listed):

$X_{i_1 t}$	$Y_{i_2 t}$
MAINT	MAINT

Objective function (9.9) – minimize cost

To achieve optimization within the context of our constraint framework, we create a constraint that specifies the total cost must be less that or equal to a set amount. In order to reduce the arity of the cost constraint, we optimize cost by week instead of over the entire planning period. The system therefore achieves a local optimum in that sense and not necessarily a global optimum. Further study is need to assess the trade-offs between constraint size and global optimality.

We implemented the cost constraint as a procedure in our CSP solving program. This procedure is called after each X type variable is instantiated. The input to the procedure is the week, t, of the variable, and the procedure returns TRUE if the total cost corresponding to week t variables assigned ON or MAINT is less than or equal to C_t, a new problem parameter (not referenced in Fig. 9.2) which specifies the maximum cost allowed in period t.

9.4.1 Solution Procedure

In order to achieve a locally optimal schedule, the C_t parameters are initially set to high values, for which it is easy to find an acceptable maintenance schedule. The C_t values are gradually lowered in unison, until a cost level is reached for which no schedule exists. The previously discovered schedule is then reported. To make this process more efficient, we can incorporate jump-back learning, as described in Chapter 7. After the problem with a certain level of C_t is solved successfully, the new constraints recorded by learning are used in subsequent attempts to find a schedule with lower C_t.

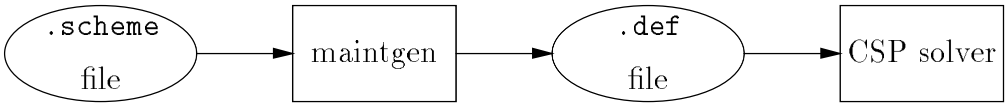
Two improvements to our procedure can be envisioned. Currently, we reduce C_k and stop when no schedule can be found. A more efficient approach is inspired by binary search: set C_k to a high value H (for which a schedule is found quickly), then to a low value L (for which the non-existence of a schedule is found quickly), then to $(H + L)/2$ (for which a schedule may or may not be found), and so on, according to the dictates of binary search. Implementing this strategy requires an enhancement to our learning scheme, for after a round where no schedule is found and C_k must be adjusted upwards, the new constraints recorded in the latest pass must be removed from the database of constraints. Currently our system has no way to distinguish constraints added by learning from those which existed at the beginning of the search. (It may be possible to distinguish between learned constraints that are based on the cost constraint and those that are not. Only the former kind would have to be "forgotten.")

The second needed improvement to our cost optimization procedure is to allow different values of C_k for different weeks. In our current implementation all values of C_k are identical.

9.5 Problem Instance Generator

The previous two sections defined the maintenance scheduling problem as we have formalized it and implemented it in the constraint satisfaction framework. One of our goals is to be able to determine the efficacy of various CSP algorithms and heuristics when applied to Maintenance Scheduling CSPs (MSCSPs). To perform an experimental average-case analysis, we need a source of many MSCSPs. We have therefore developed an MSCSP generator, which can create any number of problems that adhere to a set of input parameters. In this section we define how the generator works.

A flowchart of the overall system is below:



A "scheme" file is an ACII file (with a name usually ending in ".scheme") that defines a class or generic type of MSCSP. Here is an example of a .scheme file:

```
# lines beginning with # are comments
# first line has weeks, units, maximum simultaneous units
4 6 2
#
# next few lines have several points on the demand curve,
# given as week and demand.  Other weeks are interpolated.
0   700
```

```
3  1000
# end this list with EOL
EOL
#
# next line has initial max cost per week, and decrement amount
60000 3000
#
# next line has average unit capacity and standard deviation
200 25
#
# next line has average unit maintenance time and std. dev.
2 1
#
# next line has standard deviation for maintenance costs
1000
#
# next few lines have some points on the maintenance cost curve,
# first number is week, then one column per unit
0 10000 10000 10000 10000 10000 10000
3 13000 16000 19000 10000  7000 10000
#
# next number is standard deviation for operating costs
2000
# next few lines have some points on the operating cost curve,
# first number is week, then one column per unit
0 5000 5000 5000 5000 5000 5000
# the next line specifies the number of incompatible pairs
2
# and that's it!
```

The maintgen program reads in a scheme file and creates one or more specific MSCSP instances in a .def file which can be solved by the CSP solver. The maintgen program reads from the command line a random number generator seed and a number indicating how many individual problems should be written to the .def file created as output. With one scheme file and a seed, any number of MSCSPs can be created; the same instances can easily be recreated later, provided that the same seed is used.

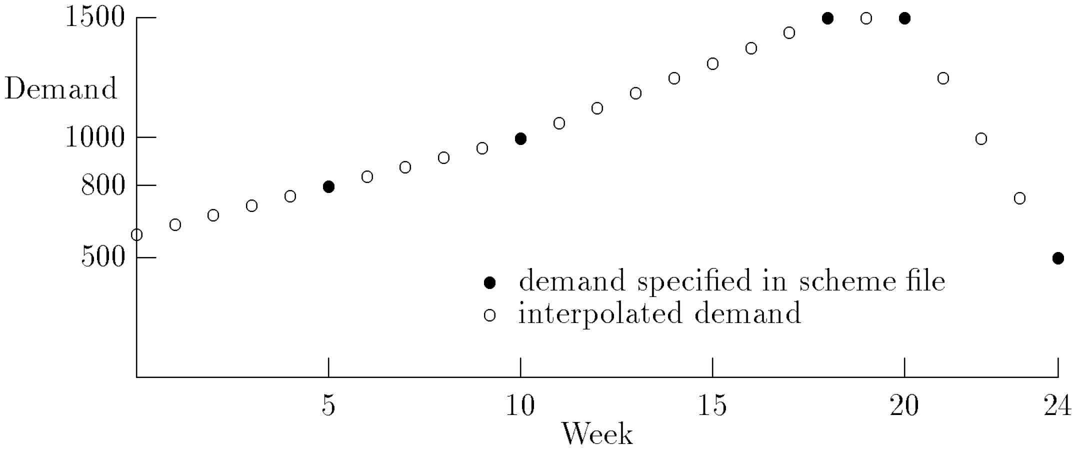


Figure 9.3: Weekly demand generated by the maintgen program when the following (week, demand) points are specified: (5, 800), (10, 10,000), (18, 1,500), (20, 1,500), (24, 500).

The first line (ignoring comment lines which begin with "#") of the scheme file specifies the fundamental size parameters of the MSCSPs which will be generated: the number of weeks W, the number of generating units U, and the number of units which can be maintained at one time M.

The scheme file specifies the demand for any number of weeks. The demand for weeks that are not explicitly specified is computed by a linear interpolation between the surrounding specified weeks. The process is shown diagrammatically in Fig. 9.3. There is no randomness in the demand "curve" that is created based on a scheme file. Note that the weeks are numbered starting from 0, so that in this example the last of the 25 weeks is week #24.

The following line in the scheme file specifies the initial maximum cost per week, and the amount it is to be decremented after each successful search for a schedule.

The characteristics of the units, that is, their output capacities and required maintenance times, are not specified individually in the scheme file. Instead, these values are randomly selected from normal distributions that have the means and standard deviations specified on the following two lines. Currently the earliest and latest maintenance start dates are not specified in the scheme file, and are always set to 0 and $W-1$ in the .def file.

Maintenance costs are specified in the following lines of the scheme file by entering first the standard deviation (1,000 in the example), and then a table which has a "week" column and then one column per unit. As with demand, values for weeks that are not entered are interpolated. However, for maintenance costs there is a random element; the interpolated value is used as the mean, together with the specified standard deviation. Operating costs are defined in the lines following the maintenance costs, with exactly the same structure.

The last piece of information is the number of incompatible pairs of units. The requested number of pairs is created randomly from a uniform distribution of the units.

Here is an example of the .def file created using the scheme file above and the random number seed 12345:

```
# comments begin with #
# first line has weeks W, units U, max-simultaneous M
4 6 2
# demand, one line per week
700
800
900
1000
# next few lines has maximum cost per week.
# Cost must be <= max.
60000 3000
# one line per unit:
# capacity  maint length  earliest maint start  latest maint start
194 1 0 3
171 3 0 3
209 1 0 3
166 1 0 3
219 2 0 3
217 2 0 3
```

```
# maintenance costs, one line per week, one column per unit
11085 10034 9374 8945 10858 10045
11056 11988 13670 10465 9301 10625
12745 14625 15422 10422 8099 7629
12534 15394 21098 9841 6748 9364
# operating costs, one line per week, one column per unit
4284 6857 3847 5050 5145 4998
5987 7352 1967 4635 6152 4635
3746 6475 5151 3988 8172 4131
6152 3436 5475 5600 4366 6070
# incompatible pairs of units (numbering starts from 0)
1 3
2 3
EOL
# and that's it!
```

The def file is in a format which is recognized by our CSP solver. The data in the file follows directly from the scheme file, and corresponds to the parameters in Fig. 9.2. The first line defines parameters W, U, and M. Next are W lines with demand informations, D_t. The following line holds the maximum cost C_t (they are all set to the same value) and the amount by which C_t is decremented.

The subsequent lines of the def file specify, for each unit, the parameters k_i, d_i, e_i, and l_i. Following them is a maintenance cost table with one value for each m_{it} and an operating cost table with one value for each c_{it}. The incompatible pairs of units from which the N set is constructed are listed, one pair per line. "EOL" marks the end of this list.

```
# test4.scheme
# W, U, M
13 15 3
#
# Some weeks on the demand curve.
0  10000
12 13000
EOL
# Initial max cost per week, and decrement amount
110000 5000
#
# Average unit capacity and standard deviation
1000 100
#
# Average unit maintenance time and std. dev.
2 1
#
# Standard deviation for maintenance costs
1000
#
# Some points on the maintenance cost curve,
0 10000 10000 10000 10000 10000 10000 10000 10000 \
  10000 10000 10000 10000 10000 10000 10000
12 10000 10000 10000 10000 10000 10000 10000 10000 \
   10000 12000 12000 12000 12000 12000 12000
#
# Standard deviation for operating costs
1000
# Operating costs
0 5000 6000 4000 5000 6000 4000 5000 6000 \
  4000 5000 4000 6000 5000 5000 8000
# the next line specifies the number of incompatible pairs
5
```

Figure 9.4: The scheme file used to generate MSCSPs.

9.6 Experimental Results

To demonstrate the effectiveness of the constraint processing framework, we present the results of experiments with two sets of MSCSPs. Experiments were based on 100 random MSCSP generated according to the parameters in the scheme file listed in Fig. 9.4, and 100 larger random instances based on a similar scheme file. The smaller problems had 15 units and 13 time periods, creating 585 variables. The larger problems had 20 units and 20 time periods, creating 1200 variables.

9.6.1 Optimization with learning

In the first experiment, we tried to find an optimal schedule for each MSCSP in the smaller and larger sets. We used the iterative cost-bound procedure described above. The results are shown in Fig. 9.5 and Fig. 9.6. BJ+DVO was the base algorithm, with jump-back learning as described in Chapter 7. The learning order was set to six; new constraints above that size were not recorded. We also found schedules using BJ+DVO without learning.

Among the 100 smaller problems, all 100 MSCSPs had schedules at cost bound 85,000 and above. Only 38 had schedules within the 80,000 bound; at 75,000 only four problems were solvable. On the set of 100 larger MSCSPs, schedules were found for all instances at cost bound 120,000 and above. 97 instances had schedules at cost bound 115,000 and 110,000; 11 at cost bound 105,000; and two at cost bound 100,000 and 95,000.

The use of learning improved the performance of the BJ+DVO algorithm on these random maintenance scheduling problems. For instance, on the smaller problems, after finding a schedule with cost bound 95,000, the average number of learned constraints was 214. Tightening the cost bound to 90,000 resulted in over twice as much CPU time needed for BJ+DVO without learning (54.01 CPU seconds compared to 23.28), but only a 71% increase for BJ+DVO with learning (29.41

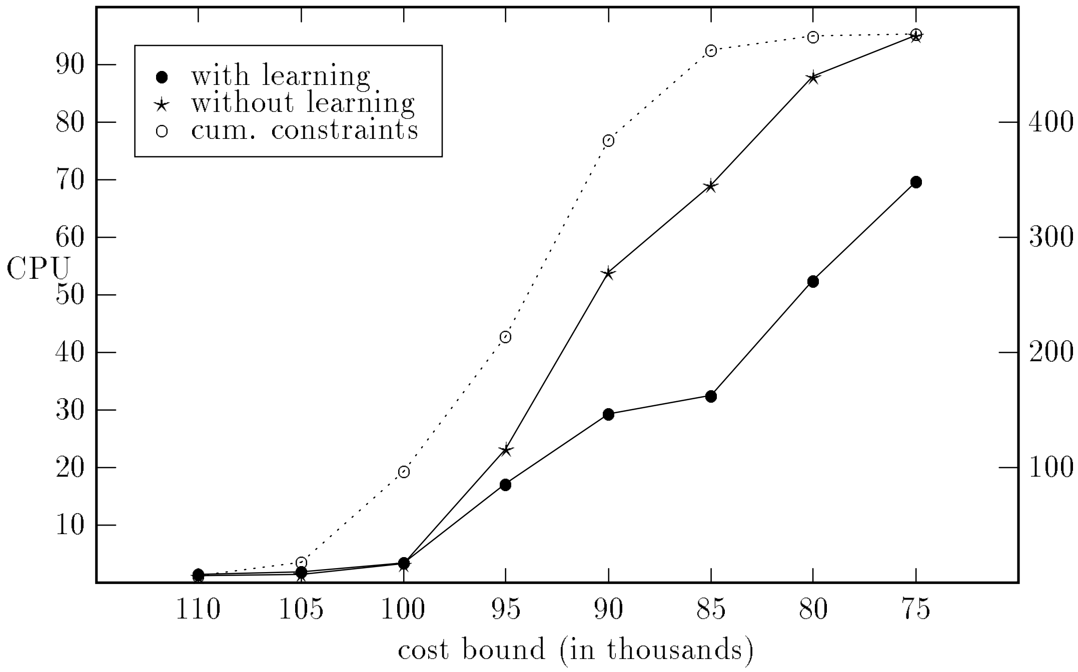


Figure 9.5: Average CPU seconds on 100 small problems (15 units, 13 weeks) to find a schedule meeting the cost bound on the y-axis, using BJ+DVO with learning ($\bullet$) and without learning ($\star$). Cumulative number of constraints learned corresponds to right-hand scale.

compared to 17.20). Learning was less effective on the larger MSCSPs. Although using learning reduced average CPU time, the improvement over BJ+DVO without learning was much less than on the smaller problems.

9.6.2 Comparison of algorithms

The second experiment utilized the same sets of 100 smaller MSCSP instances and 100 larger instances, but we did not try to find an optimal schedule. For the smaller problems we set the cost bound at 85,000 and for the larger problems we set the cost bound at 120,000. Each bound was the lowest level at which schedules could be found for all problems. We used five of the six algorithms compared in Chapter 8 to find a schedule for each problem. BT+DVO was omitted because in

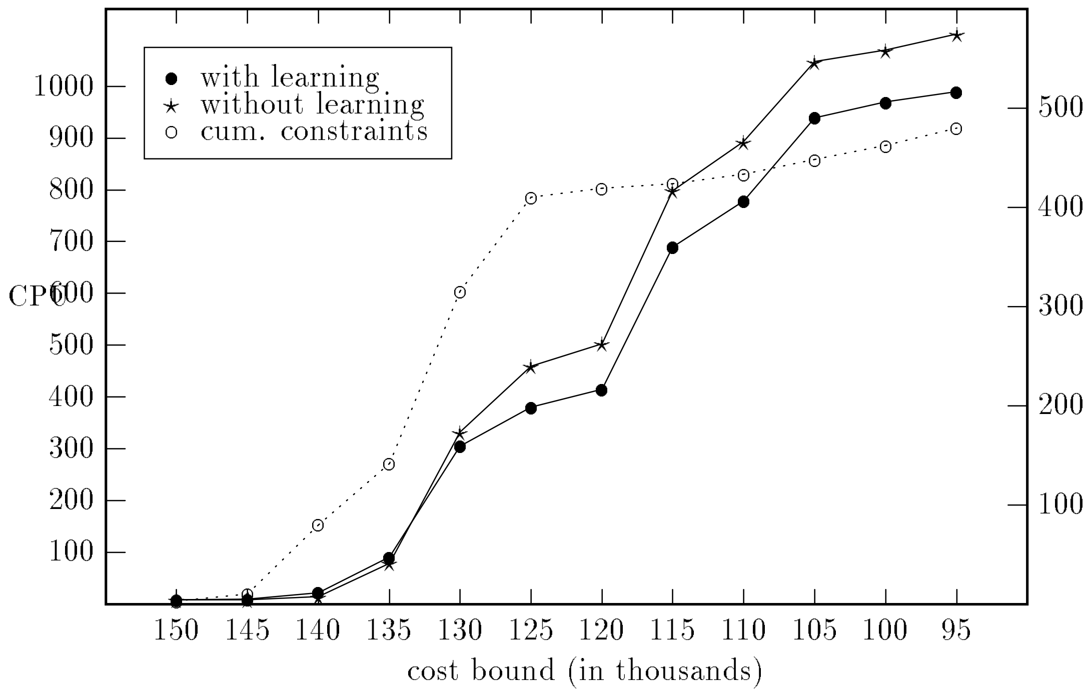


Figure 9.6: Average CPU seconds on 100 large problems (20 units, 20 weeks) to find a schedule meeting the cost bound on the y-axis, using BJ+DVO with learning ($\bullet$) and without learning ($\star$). Cumulative number of constraints learned corresponds to right-hand scale.

preliminary experiments it required several CPU hours per small problem, compared to approximately a minute per problem for the other algorithms. The results are summarized in Table 9.1.

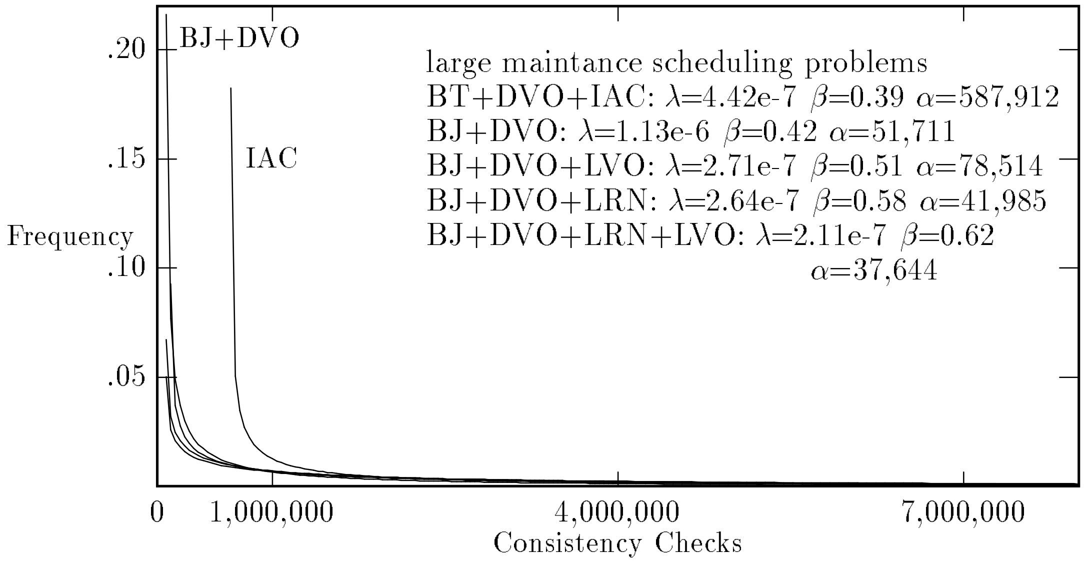
Among the five algorithms, BJ+DVO performed least well on the smaller problems and best on the larger problems, when average CPU time is the criterion. BT+DVO+IAC was the best performer on the smaller problems and the worst on the larger problems. This reversal in effectiveness may be related to the increased size of the higher arity constraints on the larger problems. The high arity constraints, such as those pertaining to the cost bound, the weekly power demand, and the existence of a maintenance period, become looser as the number of units and number of weeks increase. Results from earlier chapters indicated that more look-ahead was effective on problems with tight constraints, and detrimental on problems with loose constraints. Although as similar pattern was observed in

Algorithm	Average CC	Nodes	CPU
100 smaller problems:			
BT+DVO+IAC	315,988	3,761	51.65
BJ+DVO	619,122	8,981	70.07
BJ+DVO+LVO	384,263	5,219	54.48
BJ+DVO+LRN	671,756	8,078	67.51
BJ+DVO+LRN+LVO	476,901	5,085	57.45
100 larger problems:			
BT+DVO+IAC	7,673,173	32,105	694.02
BJ+DVO	2,619,766	28,540	460.42
BJ+DVO+LVO	6,987,091	26,650	469.65
BJ+DVO+LRN	5,892,065	27,342	521.89
BJ+DVO+LRN+LVO	6,811,663	26,402	475.12

Table 9.1: Statistics for five algorithms applied to MSCSPs.

Chapter 4 for backjumping, nevertheless backjumping remains an effective technique on the larger problems. Further experiments are required to determine how the relative efficacy of different algorithms is influenced by factors such as the size of the problem (number of weeks and units) and characteristics such as the homogeneity of the units.

As we have done in earlier chapters, we also summarize our experiments by estimating the parameters of the Weibull distribution. Fig. 9.7 shows plots of Weibull distributions curves, based on data from the experiment with 100 large scheduling problems. Data on consistency checks is represented in the top chart of Fig. 9.7, and data on CPU time in the bottom chart. The procedure, as in earlier chapters, was to estimate the parameters of the Weibull distribution using techniques described in Chapter 3. Each curve on the charts in Fig. 9.7 shows the estimated Weibull distribution for one algorithm. The estimated λ values are between 0.39 and 0.61. This range is similar to what we observed on experiments with random binary problems. These small values of λ denote a large variance in the results, with many problem instances being relatively easy and a few much harder.



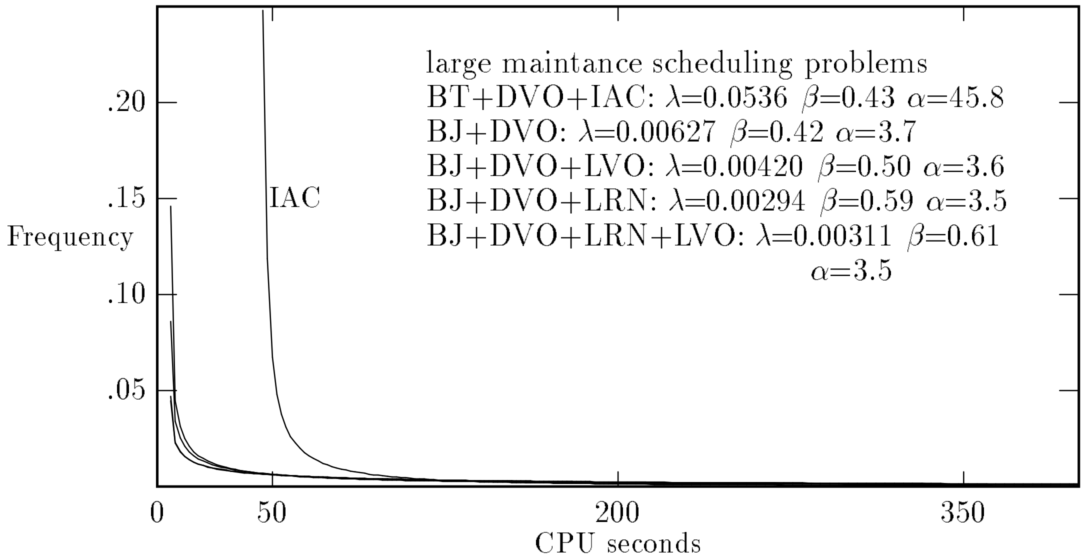


Figure 9.7: Weibull curves based on solvable "large" maintenance scheduling problems generated with whatever. The top chart is based on consistency checks, the bottom chart on CPU seconds. λ, β, and α parameters were estimated using the Modified Moment Estimator (see Chapter 3). The LVO, LRN, and LRN+LVO curves on the top chart are almost indistinguishable, as are all distributions except the one for BT+DVO+IAC in the bottom chart.

9.7 Conclusions

The constraint satisfaction problems derived from the maintenance scheduling needs of the electric power industry are an interesting testbed for the CSP algorithms developed in the earlier chapters. The problems have a mixture of tight binary constraints, such as those that bind the X and Y variables together, and loose high arity constraints, such as those that ensure that at least one maintenance period is scheduled for each unit. Perhaps the most promising algorithm for these problems is learning, which seems from preliminary evidence to be a useful part of performing optimization in the CSP framework. Further studies on larger maintenance scheduling CSPs is required to determine whether one algorithm dominates the others as problem size increases.

A challenging problem that is difficult to formalize is to find the best way to encode the requirements of a problem such as maintenance scheduling into constraints of a CSP. In section 9.4 we discussed some of the trade-offs involved in, for example, adding "hidden" variables in return for a smaller number of tuples in high arity constraints. This is a challenging area for future research that has the potential of greatly impacting the applicability of the constraint satisfaction framework to problems from science and industry.

Chapter 10

Conclusions

10.1 Contributions

The hypothesis underlying this research is that backtracking based search algorithms that incorporate sophisticated "look-ahead" techniques that query, prune, evaluate, and rearrange the future search space can be particularly effective algorithms for constraint satisfaction problems. One premise we have relied on and seen borne out multiple times is that most techniques for solving constraint satisfaction problems are not competitors, but potential allies. On sufficiently large and difficult problems, multiple arrows in the quiver will not be redundant.

The algorithms presented in the work are new combinations of previously existing, well-proven techniques. The BJ+DVO algorithm is based on Prosser's conflict-directed backjumping [68], which is itself a "marriage," as Prosser calls it, of Gaschnig's backjumping [31] and Dechter's graph-based backjumping [15]. The look-ahead technique and dynamic variable ordering heuristic in BJ+DVO are due to Haralick and Elliott [40], but both ideas appear earlier in the literature. Our experiments showed that the components of BJ+DVO work together in a way that is more effective than any of the constituent parts individually.

Look-ahead value ordering processes the variables which have not yet been assigned values in order to choose a value for the current variable. This idea first appeared in Dechter [19], using a different scheme to examine the future variables.

LVO uses the same look-ahead effort already employed by BJ+DVO, allowing the combination BJ+DVO+LVO to benefit twice from its processing of future variables. The LVO heuristic proved in our experiments to be especially helpful on the hardest constraint satisfaction problems.

The new learning algorithm we presented, jump-back learning, takes advantage of the conflict set already maintained by backjumping for search control. This technique makes the additional cost of recording constraints at dead-ends almost negligible. Whether the costs of *accessing* the learned constraints is more or less than the benefit they provide by pruning the search space was investigated empirically. Our experiments showed that when the order of learning (the maximum size of a new constraint) was limited, the benefits of learning usually outweighed the costs. We showed that learning can also play an important role in solving optimization problems in the CSP framework. In the context of maintenance scheduling problems for the electric power industry, optimization was presented as solving a series of CSP decision problems, with cost bound constraints being iteratively adjusted until an optimum is reached. Our experiments idicated that if constraints learned in one iteration are applied again in subsequent attempts to solve the problem, a substantial benefit is realized.

In addition to proposing new combination algorithms, the dissertation provides extensive empirical evaluation of them. Algorithms were run and compared on three types of problems: random binary CSPs, circuit problems encoded as CSPs, and maintenance scheduling problems encoded as CSPs. A distinguishing feature of the research presented in this dissertation is that experiments were performed on larger and harder random CSP instances than have previously been reported. Our empirical results indicated that each of the new algorithm combinations became increasingly more effective as problems became larger and harder. This is good news, because scheduling, planning, optimization, and configuration problems of interest to science and industry are often quite large. Combining multiple techniques will no doubt be necessary to attack these problems as CSPs.

We proposed in Chapter 3 that when backtracking-based algorithms are applied to random binary CSPs, the distribution of sizes of the resulting search trees can be closely approximated by standard continuous probability distributions. It is necessary to segregate solvable and unsolvable problem instances. After doing so, the empirical distribution of unsolvable problems is close to a lognormal distribution, and the empirical distribution of solvable problems is close to a Weibull distributions. We used these distribution functions, along with estimated parameters, to report the results of several experiments in the dissertation.

10.2 Future Work

There are many directions in which this research can be extended. One promising technique for combining multiple algorithms or heuristics is not to integrate them into a single executable unit, but to run several algorithms in tandem, either on parallel computers, or via time-slicing on a single CPU. All algorithms then stop when a single algorithm returns an answer. The potential benefit of this scheme stems from the great variance in the time required to solve instances. This variance has been observed not only over multiple instances drawn from a distribution of problems (as was the focus of Chapter 3), but also over multiple applications of a single algorithm to a single instance, when points of non-determinacy in the algorithm (for instance, variable and value ordering) are decided randomly [77]. It may be possible to extend the analysis of distributions started in Chapter 3 to provide guidance in how much time should be allocated to each time-sliced algorithm. In Chapter 3 the "completion rate" of an algorithm was defined as equivalent to the well-known concept of the hazard rate in reliability theory. It is a measure of the probability of completing the search in the next time unit. Knowledge of the completion rate function can be used in resource-limited situations to suggest an optimum time-bound for an algorithm to process a single instance. Examples would be running multiple algorithms on a single instance in a time-sliced manner, as proposed in [42], and environments where the goal is to complete as many problems as possible in a fixed time period.

An exciting trend in CSP algorithms that has received intense interest recently is the use of randomized or "local search" algorithms. These algorithms essentially guess a solution, and if the guess proves wrong (as of course it almost always does, initially) they are able to follow a gradient in the search space which, it is hoped, leads towards a solution. A critical feature of these algorithms is that if no solution is found after a certain bound on CPU time or subroutine calls, the process is restarted with a new random guess. Several ways to combine randomized search with techniques developed for backtracking have been proposed. Because randomized approaches almost always involve multiple restarts on the same instance, incorporating learning in random algorithms seems particularly promising. For instance, if a new constraint is learned before each restart, then as with backtracking-based search algorithms, the randomized algorithm will be prevented from making the same mistake again.

A third interesting direction for further research is continued study of the correlation between empirical distributions of effort and standard probability distributions from statistics. Perhaps the starting point should be to put the speculations in the last section of Chapter 3 on a more rigorous footing. An explanation of why the lognormal and Weibull distributions approximate the empirical distributions, based on both the mathematical derivation of the distribution functions, and on the known properties of the CSP algorithms and the random problem distributions, might provide a deeper understanding of the search process, might lead to improved algorithms, and might aid in the reporting of experiments with random problems.

Our experiments with backjumping and look-ahead all relied on an amount of look-ahead identical to that of forward checking. However, in Chapter 5 we observed that for some types of problems a stronger amount of look-ahead, such as arc-consistency, was beneficial. We left open the interesting question of whether there exist problem classes for which the combination of backjumping and integrated arc-consistency is effective.

10.3 Final Conclusions

The goal of this dissertation has been to define and evaluate algorithms that combine several effective techniques. Most of the evaluation has been based on random binary constraint satisfaction problems, an approach that has been used in the field for over twenty years. Our interest in accurately summarizing the results of large-scale experiments led to the study of the distribution of problem difficulty. Because many real world problem have distinct structure and non-binary constraints, we also used problems from DIMACS and the electric power industry as a testbed for the algorithms. The conclusion we draw from all the experiments is that no one algorithm is best for all problems, but that by combining several techniques into one algorithm, and by selecting the right algorithm based on the characteristics of the problem instance, a substantial increase in performance can be achieved.

Bibliography

[1] J. Aitchison and J. A. C. Brown. *The Lognormal Distribution.* Cambridge University Press, Cambridge, England, 1960.

[2] T. M. Al-Khamis, S. Vemuri, L. Lemonidis, and J. Yellen. Unit maintenance scheduling with fuel constraints. *IEEE Trans. on Power Systems*, 7(2):933–939, 1992.

[3] Fahiem Bacchus and Paul van Run. Dynamic Variable Ordering In CSPs. In Ugo Montanari and Francesca Rossi, editors, *Principles and Practice of Constraint Programming*, pages 258–275, 1995.

[4] Andrew B. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.* PhD thesis, University of Oregon, Eugene, OR 97403, 1995.

[5] Roberto Bayardo and Daniel Mirankar. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 298–304, 1996.

[6] Christian Bessiére. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[7] Christian Bessière and Jean-Charles Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In Eugene C. Freuder, editor, *Principles and Practice of Constraint Programming – CP96*, pages 61–75, 1996.

[8] James R. Bitner and Edward M. Reingold. Backtrack Programming Techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[9] M. Bruynooghe and L. M. Pereira. Deduction revision by intelligent backtracking. In J. A. Campbell, editor, *Implementation of Prolog*, pages 194–215. Ellis Horwood, 1984.

[10] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the *really* hard problems are. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 331–337, 1991.

[11] A. C. Cohen and B. J. Whitten. *Parameter Estimation in Reliability and Life Span Models*. Marcel Dekker, Inc., New York, 1988.

[12] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27, 1993.

[13] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5:394–397, 1962.

[14] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[15] Rina Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[16] Rina Dechter. Constraint networks. In *Encyclopedia of Artificial Intelligence*, pages 276–285. John Wiley & Sons, 2nd edition, 1992.

[17] Rina Dechter and Itay Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68:211–241, 1994.

[18] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[19] Rina Dechter and Judea Pearl. Tree-clustering schemes for constraint processing. *Artificial Intelligence*, 38:353–366, 1989.

[20] J. F. Dopazo and H. M. Merrill. Optimal Generator Maintenance Scheduling using Integer Programming. *IEEE Trans. on Power Apparatus and Systems*, PAS-94(5):1537–1545, 1975.

[21] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satifiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[22] G. T. Egan. An Experimental Method of Determination of Optimal Maintenance Schedules in Power Systems Using the Branch-and-Bound Technique. *IEEE Trans. SMC*, SMC-6(8), 1976.

[23] Hani El Sakkout, Mark G. Wallace, and E. Barry Richards. An Instance of Adaptive Constraint Propagation. In Eugene C. Freuder, editor, *Principles and Practice of Constraint Programming – CP96*, pages 164–178, 1996.

[24] R. E. Fikes and N. J. Nilsson. A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[25] E. C. Freuder. A sufficient condition for backtrack-free search. *JACM*, 21(11):958–965, 1982.

[26] Daniel Frost and Rina Dechter. Dead-end driven learning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 294–300, 1994.

[27] Daniel Frost and Rina Dechter. In search of the best constraint satisfaction search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 301–306, 1994.

[28] Daniel Frost and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 572–578, 1995.

[29] Daniel Frost, Irina Rish, and Lluís Vila. Summarizing csp hardness with continuous probability distributions. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 327–333, 1997.

[30] John G. Gaschnig. A General Backtrack Algorithm That Eliminates Most Redundant Tests. In *Proceedings of the International Joint Conference on Artificial Intelligence*, page 247, 1977.

[31] John G. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 1979.

[32] Peter Andreas Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In *10th European Conference on Artificial Intelligence*, pages 31–35, 1992.

[33] Richard Génisson and Phillippe Jégou. Davis and Putnam were already checking forward. In *12th European Conference on Artificial Intelligence*, pages 180–184, 1996.

[34] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. In Eugene C. Freuder, editor, *Principles and Practice of Constraint Programming – CP96*, pages 179–193, 1996.

[35] Ian P. Gent and Patrick Prosser. The 50% Point in Constraint-Satisfaction Problems. Technical Report 95/180, Department of Computer Science, University of Strathclyde, 1995.

[36] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[37] Solomon W. Golomb and Leonard D. Baumert. Backtrack Programming. *Communications of the ACM*, 12(4):516–524, 1965.

[38] Carla Gomes, Bart Selman, and Nuno Crato. Heavy-Tailed Distributions in Combinatorial Search. In Gert Smolka, editor, *Principles and Practice of Constraint Programming – CP97*, pages 121–135, 1997.

[39] Steven Hampson and Denis Kibler. Large Plateaus and Plateau Search in Boolean Satisfiability Problems: When to Give Up Searching and Start

Again. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satifiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[40] R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[41] Tad Hogg and Colin P. Williams. The hardest constraint satisfaction problems: a double phase transition (Research Note). *Artificial Intelligence*, 69:359–377, 1994.

[42] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An Economics Approach to Hard Computational Problems. *Science*, 275:51–54, 1997.

[43] Brigette Jaumard, Mihnea Stan, and Jacques Desrosiers. Tabu Search and a Quadratic Relaxation for the Satisfiability Problems. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satifiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[44] David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satifiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, Rhode Island, 1996.

[45] Hyunchul Kin, Yasuhiro Hayashi, and Koichi Nara. An Algorithm for Thermal Unit Maintenance Scheduling Through Combined Use of GA SA and TS. *IEEE Trans. on Power Systems*, 12(1):329–335, 1996.

[46] Donald E. Knuth. Estimating the Efficiency of Backtrack Programs. *Mathmatics of Computation*, 29(129):121–136, 1975.

[47] Grzegorz Kondrak and Peter van Beek. A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[48] R. E. Korf. A program than learns how to solve Rubik's cube. In *Proceedings of the National Conference on Artificial Intelligence*, pages 164–167, 1982.

[49] Alvin C. M. Kwan. Validity of Normality Assumption in CSP Research. In *PRICAI'96: Topics in Artificial Intelligence. Proc. of the 4th Pacific Rim International Conference on Artificial Intelligence*, pages 253–263, 1996.

[50] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[51] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[52] Alan K. Mackworth. The logic of constraint satisfaction. *Artificial Intelligence*, 58:3–20, 1992.

[53] Nancy R. Mann, Ray E. Schafer, and Nozer D. Singpurwalla. *Methods for Statistical Analysis of Reliability and Life Data.* John Wiley and Sons, New York, 1974.

[54] Steven Minton. *Learning Search Control Knowledge: An Explanation-based Approach.* Kluwer Academic Publishers, 1988.

[55] Steven Minton. Qualitative Results Concerning the Utility of Explanation-Based Learning. *Artificial Intelligence*, 42:363–392, 1990.

[56] Steven Minton, Mark D. Johnson, Andrew B. Phillips, and Philip Laird. Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 17–24, 1990.

[57] David Mitchell. Respecting Your Data (I). In *AAAI-94 Workshop on Experimental Evaluation of Reasoning and Search Methods*, pages 28–31, 1994.

[58] David Mitchell, Bart Selman, and Hector Levesque. Hard and Easy Distributions of SAT Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, 1992.

[59] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 25:65–74, 1986.

[60] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[61] Bernard A. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer, 1988.

[62] Bernard A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[63] Bernard A. Nadel. Some applications of the constraint satisfaction problem. Technical report, Wayne State University, 1990.

[64] Wayne Nelson. *Accelerated Testing: Statistical Models, Test Plans, and Data Analyses*. John Wiley & Sons, New York, 1990.

[65] Bernard Nudel. Consistent-Labeling Problems and their Algorithms: Expected-Complexities and Theory-Based Heuristics. *Artificial Intelligence*, 21:135–178, 1983.

[66] Judea Pearl. *Heuristics*. Addison-Wesley, Reading, Mass., 1985.

[67] Daniele Pretolani. Efficiency and Stability of Hypergraph SAT Algorithms. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satifiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[68] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[69] Patrick Prosser. Binary Constraint Satisfaction Problems: Some are Harder than Others. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI 94)*, pages 95–99, 1994.

[70] Patrick Prosser. Mac-cbj: maintaining arc consistency with conflict-directed backjumping. Technical Report 95/177, The University of Strathclyde, Glasgow, Scotland, Dept. of Computer Science, 1995.

[71] Paul Walton Purdom. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence*, 21:117–133, 1983.

[72] Mauricio G. C. Resende and Thomas A. Feo. A GRASP for Satisfiability. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satifiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[73] Irina Rish and Daniel Frost. Statistical analysis of backtracking on inconsistent csps. In Gert Smolka, editor, *Principles and Practice of Constraint Programming – CP97*, pages 150–162r, 1997.

[74] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Principles and Practice of Constraint Programming*, pages 10–20, 1994.

[75] Lothar Sachs. *Applied Statistics: A Handbook of Techniques*. Springer-Verlag, New York, second edition, 1984.

[76] Norman Sadeh and Mark S. Fox. Variable and Value Ordering Heuristics for Activity-based Job-shop Scheduling. In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management*, pages 134–144, 1990.

[77] Bart Selman and Scott Kirkpatrick. Critical behavior in the computational cost of satisfiability testing. *Artificial Intelligence*, 81:273–295, 1996.

[78] Bart Selman, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[79] Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81:17–29, 1996.

[80] Jude W. Shavlik and Thomas G. Dietterich. General aspects of machine learning. In Jude W. Shavlik and Thomas G. Dietterich, editors, *Readings in Machine Learning*, pages 1–10. Morgan Kaufmann, 1990.

[81] Barbara M. Smith. Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI 94)*, pages 100–104, 1994.

[82] Barbara M. Smith and M. E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155–181, 1996.

[83] William M. Spears. Simulated Annealing for Hard Satisfiability Problems. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satifiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[84] R. M. Stallman and G. S. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[85] H. S. Stone and J. M. Stone. Efficient search techniques: An empirical study of the n-queens problem. Technical Report Tech. Rept. RC 12057 (54343), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1986.

[86] Peter van Beek and Rina Dechter. Constraint tightness and looseness versus local and global consistency. *Journal of the ACM*, 44(4):549–566, 1997.

[87] Allen Van Gelder and Yumi K. Tsuji. Satisfiability Testing with More Reasoning and Less Guessing. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satifiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[88] P. Van Hentenryck, Y Deville, and C. M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[89] R. J. Walker. *Combinatorial Analysis (Proceedings of Symposia in Applied Mathematics, Vol. X)*, chapter An enumerative technique for a class of combinatorial problems, pages 91–94. American Mathematical Society, 1960.

[90] David Waltz. Understanding Line Drawings of Scenes with Shadows. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[91] Colin P. Williams and Tad Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:73–117, 1994.

[92] J. Yellen, T. M. Al-Khamis, S. Vemuri, and L. Lemonidis. A decomposition approach to unit maintenance scheduling. *IEEE Trans. on Power Systems*, 7(2):726–731, 1992.

[93] H. H. Zurm and V. H. Quintana. Generator Maintenance Scheduling Via Successive Approximation Dynamic Programming. *IEEE Trans. on Power Apparatus and Systems*, PAS-94(2), 1975.