Intelligent Backtracking On Constraint Satisfaction Problems: Experimental and Theoretical Results

Andrew B. Baker

CIS-TR-95-08
March 1995

Department of Computer and Information Science
University of Oregon

INTELLIGENT BACKTRACKING ON CONSTRAINT SATISFACTION PROBLEMS: EXPERIMENTAL AND THEORETICAL RESULTS

by

ANDREW B. BAKER

A DISSERTATION

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

March 1995

"Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results," a dissertation prepared by Andrew B. Baker in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

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

Dr. Eugene M. Luks, Co-chair of the Examining Committee

Date

Committee in charge:	Dr. Matthew L. Ginsberg, Co-chair
	Dr. Eugene M. Luks, Co-chair
	Dr. James M. Crawford
	Dr. Thomas G. Dietterich
	Dr. William M. Kantor

Accepted by:

Vice Provost and Dean of the Graduate School

Abstract

The Constraint Satisfaction Problem is a type of combinatorial search problem of much interest in Artificial Intelligence and Operations Research. The simplest algorithm for solving such a problem is chronological backtracking, but this method suffers from a malady known as "thrashing," in which essentially the same subproblems end up being solved repeatedly. Intelligent backtracking algorithms, such as backjumping and dependency-directed backtracking, were designed to address this difficulty, but the exact utility and range of applicability of these techniques have not been fully explored. This dissertation describes an experimental and theoretical investigation into the power of these intelligent backtracking algorithms.

We compare the empirical performance of several such algorithms on a range of problem distributions. We show that the more sophisticated algorithms are especially useful on those problems with a small number of constraints that happen to be difficult for chronological backtracking. We also illuminate some issues concerning the distribution of hard and easy constraint problems. It was previously believed that the hardest problems had a small number of constraints, but we show that this result is an artifact of chronological backtracking that does not apply to the more advanced algorithms.

We then study the performance of a particular intelligent backtracking algorithm that has been of much interest: dynamic backtracking. We demonstrate experimentally that for some problems this algorithm can do more harm than good. We discuss the reason for this phenomenon, and we present a modification to dynamic backtracking that fixes the problem.

We also present theoretical worst-case results. It is known that dependency-directed backtracking can solve a constraint satisfaction problem in time exponential in a particular problem parameter known as the "induced width." This algorithm, however, requires about as much space as time; that is, its space requirements also grow exponentially in this parameter. This suggests the question of whether it is possible for a polynomial-space algorithm to be exponential in the induced width. We show that for a large class of constraint satisfaction algorithms, it is not possible. That is, there is a trade-off in intelligent backtracking between space and time.

Acknowledgements

I would like to thank my advisor, Matt Ginsberg, for his wise advice, his technical leadership, and his many acts of assistance and kindness. It is no exaggeration to say that without him this dissertation would never have been written; indeed, it would not even have been started. Matt encouraged me to come to Oregon to finish up my Ph.D. and improve my employment prospects. I was a bit skeptical at first, but everything worked out exactly the way he said it would.

Jimi Crawford has been a source of invaluable technical expertise and of many good ideas; I have benefited greatly from his help. I would also like to thank Eugene Luks, Thomas Dietterich, and William Kantor for their useful comments on this dissertation.

I have enjoyed working with all the other members of the Computational Intelligence Research Laboratory, especially Will Harvey, Tania Bedrax-Weiss, Ari Jónsson, and David Etherington. Laurie Buchanan kept the lab running smoothly, and Betty Lockwood (at the Computer Science Department) helped shepherd me through the various requirements of the Ph.D. program.

This work has been supported by the Air Force Office of Scientific Research under grant number 92-0693 and by ARPA/Rome Labs under grant numbers F30602-91-C-0036 and F30602-93-C-00031. My earlier work at Stanford was supported by a graduate fellowship from the Air Force Office of Scientific Research. I owe many thanks to these funding organizations, and thus ultimately to the American taxpayers.

Finally, I am grateful to my parents for their steadfast love and support through good times and bad. I cannot thank them enough.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Constraint satisfaction problems are a type of combinatorial search problem of much interest in Artificial Intelligence and Operations Research. In a constraint satisfaction problem, or CSP, we are given a set of variables to which values must be assigned, and we are also given a set of constraints that restrict these assignments. The challenge is to assign values to all the variables without violating any of the constraints — or to determine that no such solution is possible [29, 56, 58, 59, 66].

One example of a constraint satisfaction problem is graph (or map) coloring. Here the variables are the vertices (or countries), the values are the available colors, and the constraints require that adjacent variables be colored differently. Other CSPs are important in scheduling, design, temporal reasoning, natural language understanding, scene interpretation, and a slew of other applications.

The simplest algorithm used in practice is backtracking, or depth-first search. It has long been noticed, however, that simple backtracking can suffer from a variety of maladies, referred to as "thrashing," in which time is wasted exploring portions of the search space that cannot possibly contain any solutions. A number of more intelligent backtracking algorithms have been developed with the goal of reducing the amount of thrashing, e.g., backjumping [40], dependency-directed backtracking [79], k-order learning [28], and dynamic backtracking [42]. The exact utility and range of applicability of these techniques, however, have not been fully explored.

On what type of constraint problem is intelligent backtracking most likely to be useful? Is it helpful only on a few pathological examples, or can it actually improve the average performance? What are the disadvantages, if any, of the various techniques? What type of worst-case guarantees can one give? This dissertation is an experimental and theoretical investigation into the power of the various intelligent backtracking algorithms, and it will shed some light on these questions.

The experimental results are in Chapters 2 and 3, and the theoretical results are in Chapters 4 and 5. Sections 1.1 and 1.2 review some basic concepts that are necessary for understanding the dissertation, and Section 1.3 provides an overview of the new results.

1.1 Constraint Satisfaction Problems

A *constraint satisfaction problem*, or *CSP*, consists of a set of variables that must be assigned values from certain domains, subject to a set of constraints. Each constraint specifies allowed combinations of values for some tuple of variables. More formally:

Definition 1.1 *A* constraint satisfaction problem *is defined by a triple* (V, D, C)*. Here,* V *is a finite set of variables, and* D *is a mapping from* V *to the domains; for each variable* $v \in V$*,* D_v *will denote the domain of* v*, that is, the finite set of values that* v *may assume. Finally,* C *is a finite set of constraints; each constraint in* C *is a pair* (W, Q)*, where* $W = (w_1, \ldots, w_k)$ *is a list of variables from* V*, and* $Q \subseteq D_{w_1} \times \cdots \times D_{w_k}$ *is a predicate on these variables. A* solution *to the problem is a total assignment* f *of values to variables, such that for each* $v \in V$*,* $f(v) \in D_v$ *and for each constraint* $((w_1, \ldots, w_k), Q)$*,* $(f(w_1), \ldots, f(w_k)) \in Q$*. If there is no such solution, then we will say that the problem is* unsatisfiable.

We are interested in the *search problem* for a CSP, that is, the problem of finding a solution if one exists, or otherwise returning the answer UNSAT (short for unsatisfiable). There are other problems that one might also study: the decision problem (determining whether or not a solution exists), the enumeration problem (finding all the solutions), or even the counting problem (determining the number of solutions)[70]. There are some obvious relationships between the various problems, but in general, when we refer without further elaboration to a "CSP algorithm," this will mean an algorithm for the search problem.

For any variable v, we will refer to $|D_v|$, the number of allowed values for v, as the *domain size* of that variable. A *Boolean CSP* is one in which the maximum domain size is 2. A constraint that involves k variables, say $((w_1, \ldots, w_k), Q)$, will be called a k-ary constraint; if k is 2, then it will also be called a *binary* constraint. The arity of a CSP is the maximum arity of any of its constraints, and a binary CSP is one with arity 2.

To illustrate these definitions, consider a very small example with three variables.

Example 1.2 *Let V be $\{a, b, c\}$, let $D_a = D_b = D_c = \{1, 2, 3\}$, and let W be*

$$\{((a, b), G), ((b, c), G), ((c, a), G)\},$$

where G is the predicate $\{(x, y) : x > y\}$.

Example 1.2 requires that a, b, and c all be integers from 1 to 3, such that $a > b > c > a$. (Note that this is a binary CSP since each constraint connects only 2 variables.) The CSP, of course, has no solution. If we modified the example by eliminating the constraint $c > a$, then we would have exactly one solution, namely the function f, where $f(a) = 3$, $f(b) = 2$, and $f(c) = 1$.

The computational complexity of a problem is generally discussed in terms of the size of its encoding as a string, so we should say something about the encoding that we are assuming. Consider a problem with n variables and a maximum domain size of a. For each domain, the only real information is the number of values in that domain, and this will require $O(\log a)$ space per variable or $O(n \log a)$ altogether.[1] To encode the k-ary constraint, $((w_1, \ldots, w_k), Q)$, we first use $O(k \log n)$ to list the relevant variables, and we then use

$$\prod_{i=1}^{k} |D_{w_i}|$$

[1]This is the standard notation for representing asymptotic complexity [21]. We say that $f(n) = O(g(n))$ if there are positive constants c and n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$. Similarly, $f(n) = \Omega(g(n))$ if there are positive constants c and n_0 such that $f(n) > cg(n)$ for all $n \geq n_0$.

bits to encode the predicate Q (that is one bit for each combination of values). Thus, if there are m constraints altogether, if a is the maximum domain size, and if the constraints are at most k-ary, then the size of the problem is

$$O(n \log a + m(k \log n + a^k)) \tag{1.1}$$

The details of formula (1.1) are of little importance. The only thing to note is the exponential dependence on k; as long as k is bounded, (1.1) is polynomial in all the parameters, but when k is unbounded for some class of problems, we will have to be more careful: in that case, we might want to represent the constraint as a procedure instead of as a giant table.

It is clear that in its most general form, the constraint satisfaction problem is NP-hard. This is true even for some quite restricted special cases. Consider, for example, the class of Boolean CSPs with maximum arity 3. This subsumes 3-SATISFIABILITY, which is NP-complete [37]. Alternatively, consider the class of binary CSPs with maximum domain size 3. This includes GRAPH 3-COLORABILITY as a special case, and that problem is also known to be NP-complete [37]. If we require all constraints to be binary *and* all domains to be Boolean, then we finally have a tractable problem, 2-SATISFIABILITY, which can in fact be solved in linear time [4]. This case is so restricted, however, that it is of little practical interest, and as we have just noted, relaxing these restrictions even slightly would immediately make the problem NP-hard again.

One thing that might be interesting would be a measure of problem "difficulty", for which the time complexity of the problem would be exponential only in this parameter, regardless of the size of the problem. Thus for any fixed difficulty level, the class of problems of this difficulty would be a tractable problem class. As Dechter and Pearl [30] have noted, the *induced width* is such a parameter, and we will discuss the induced width in some depth in Chapters 4 and 5.

1.2 Backtracking Algorithms

This section will discuss the major systematic search algorithms for constraint satisfaction problems. All of the algorithms make use of *partial assignments*. A partial assignment assigns values to some subset of the variables in the problem, leaving the others unbound. Two special cases are a *total assignment*, which assigns values to all of the variables, and the *null assignment*, which assigns values to none of them. The search algorithms to be described all work as follows. They start with the null assignment, and then extend it incrementally by assigning values to successive variables. If at some point, they determine that the current partial assignment is a dead end, that is, that it cannot be extended to a total assignment that satisfies all of the constraints, then the algorithms "backtrack" by unsetting some of the variables in the current partial assignment. Eventually, they either succeed in finding a solution, or in proving that no solution exists. Where the various algorithms differ is in how quickly they can discover a dead end, and how far they are then able to backtrack.

1.2.1 Exhaustive Search

About the simplest approach imaginable is to generate each of the

$$\prod_{v \in V} |D_v|$$

total assignments in order until one is found that satisfies all of the constraints. It is not a very efficient search strategy, but it will serve as a convenient starting point for our analysis. The algorithm, EXHAUSTIVE-SEARCH, is displayed in Figure 1.1.[2]

The top-level procedure, EXHAUSTIVE-SEARCH-TOP, is called with one argument, P, which should be a CSP of the form (V, D, C), and it returns a solution to the CSP, or UNSAT if no solution exists. It does this by creating an empty assignment, and then calling the recursive procedure EXHAUSTIVE-SEARCH to do the

[2]We use the following conventions in our pseudocode: procedures (like EXHAUSTIVE-SEARCH) and symbolic constants (like UNSAT) are written in small capitals, variables are in italics, and keywords are in boldface; program structure is indicated by indentation alone in order to avoid the clutter of **begin** and **end** statements or other delimiters; finally, the line numbers are not part of the code, but merely serve as convenient labels to facilitate discussion of the algorithms.

```
EXHAUSTIVE-SEARCH-TOP($P$)    {where $P$ is a CSP of the form $(V, D, C)$}
1   $f$ := the null assignment
2   return EXHAUSTIVE-SEARCH($f, P$)

EXHAUSTIVE-SEARCH($f, P$)
1   if $f$ is a total assignment of the variables in $P$
2       if $f$ satisfies the constraints in $P$
3           answer := $f$
4       else
5           answer := UNSAT
6   else
7       $v$ := some variable in $P$ that is not yet assigned a value by $f$
8       answer := UNSAT
9       for each value $x \in D_v$ while answer = UNSAT
10          $f(v) := x$
11          answer := EXHAUSTIVE-SEARCH($f, P$)
12  return answer
```

Figure 1.1: Exhaustive search

work. EXHAUSTIVE-SEARCH takes a partial assignment f as its first argument, and it returns some completion of this assignment that satisfies all of the constraints, or UNSAT if there is no such total assignment. If f is already a total assignment, then the procedure just has to check whether f satisfies the constraints or not. If f is not yet a total assignment, the algorithm chooses an unbound variable v, and then recursively invokes EXHAUSTIVE-SEARCH for each possible value of v. The **for-while** loop on lines 9–11 is set up to terminate as soon as the first solution is found; it would be straightforward to modify the algorithm to instead collect all of the solutions.

The procedure, as described, is nondeterministic because it does not specify which variable is selected on line 7, and nor does it specify the order in which the values are enumerated on line 9. The question of how to make these decisions wisely is actually a very important issue when solving CSPs. One possibility is a *static* variable ordering in which the variables are simply instantiated in some fixed order, say $v_1, v_2, \ldots, v_n$. Similarly, a static value ordering would always enumerate the values in some fixed order. A more sophisticated strategy would be to make these decisions *dynamically* based on the current state of the search. For example, after setting v_1 to some value, one might branch on v_2, but after going back and setting v_1 to some other value, one might then branch on v_3. For the simpleminded EXHAUSTIVE-SEARCH algorithm, this would not make much of a difference, but for some of the more sophisticated search algorithms, there is substantial evidence that dynamic orderings help [23, 43, 49, 87]. We will have something to say on this subject later, but for now, the examples we use to illustrate the various algorithms will all use static orderings.

1.2.2 Backtracking

The most obvious inefficiency in EXHAUSTIVE-SEARCH is that it does not check any of the constraints until it gets to the bottom of the search tree. Consider a problem in which the variables are selected in the (static) order $v_1, v_2, \ldots v_{100}$, and the only possible values for any of the variables are 1 and 2; suppose the value 1 is always tried before the value 2. Suppose further that there is a constraint that v_1 and v_2 cannot both be 1 at the same time. After setting v_1 and v_2 to be 1, the exhaustive search approach will not notice the contradiction until the other 98 variables have been set, and it will have to rediscover this contradiction for each possible completion, or up to 2^{98} times altogether. This type of pathology, in which the search algorithm repeatedly fails for the same reason, is sometimes referred to as "thrashing." It would be more efficient to check the constraints immediately after each variable is assigned a value, and this is what *backtracking* does [8, 12, 17, 46, 55, 84].

The only difference between the backtracking algorithm in Figure 1.2 and the exhaustive-search algorithm in Figure 1.1 is in the location of the consistency check. After assigning a value to a variable on line 7, BACKTRACKING checks at once (on line 8) whether the partial assignment still satisfies the constraints; if it does not, the procedure will move on to the next value immediately. The nature of the consistency check on line 8 is slightly different from that of EXHAUSTIVE-SEARCH. In the earlier procedure, we were checking a total assignment against all of the constraints. Here we are checking a partial assignment, so many of the constraints will not yet be applicable. If we have some constraint (W, Q), where W is a list of variables, and Q is a predicate on these variables, then the constraint will only be applicable if the current partial assignment values all of the variables in W; if any of these variables are not yet bound, then the constraint will be ignored for now. Furthermore, if all the variables in W are bound, but the current variable v is not in W, then the constraint can also be ignored since its consistency must have already been verified at an earlier stage. In other words, instead of checking all of the constraints at the bottom of the tree, the BACKTRACKING routine checks them incrementally on the way down. Backtracking, then, is just a depth-first search of the tree of partial solutions. Sometimes, it is referred to as "chronological" backtracking to distinguish it from the more sophisticated algorithms that we will discuss below.

1.2.3 Backjumping

Backtracking reduces the amount of thrashing, but it does not eliminate it. Consider a slight modification of the above example. Suppose that the variables are again selected in the order $v_1, v_2, \ldots, v_{100}$, and the values are again selected in the order 1 and then 2. Finally, assume that we have the constraint that $v_1 > v_{100}$. If v_1 is set to 1, then there is no possible value for v_{100}, but backtracking will take a long time to

```
BACKTRACKING-TOP($P$)    {where $P$ is a CSP of the form $(V, D, C)$}
1   $f$ := the null assignment
2   return BACKTRACKING($f, P$)

BACKTRACKING($f, P$)
1   if $f$ is a total assignment of the variables in $P$
2       answer := $f$
3   else
4       $v$ := some variable in $P$ that is not yet assigned a value by $f$
5       answer := UNSAT
6       for each value $x \in D_v$ while answer = UNSAT
7           $f(v) := x$
8           if $f$ satisfies the constraints in $P$
9               answer := BACKTRACKING($f, P$)
10  return answer
```

Figure 1.2: Backtracking

learn this. After setting v_1 through v_{99} all to 1, and failing with both values of v_{100}, BACKTRACKING will go back, set $v_{99} = 2$, and then try v_{100} again. Of course, this will not help at all since the contradiction was solely with v_1. BACKTRACKING will have to go through all 2^{98} combinations of values for v_2 through v_{99} before backing up to v_1.

What backtracking lacks is some mechanism for keeping track of the *reasons* for a failure. If one knew which variables were relevant, one could immediately "jump back" to the most recent of these variables instead of backtracking futilely through all the intermediate irrelevant variables. This is the idea behind Gaschnig's *backjumping* algorithm [39, 40]; see Figure 1.3 for the algorithm.[3]

The BACKJUMPING procedure, like BACKTRACKING, takes two arguments, a partial assignment f and a constraint satisfaction problem P, but it now returns two

[3]Actually, Gaschnig's original backjumping [39, 40] was less powerful than the algorithm in Figure 1.3. His algorithm only jumped back from leaf nodes (that is, calls to the procedure for which all the constraint checks failed immediately without any further recursive calls), and then backtracked normally thereafter. Our algorithm corresponds to backjumping as described by Ginsberg [42] and to what Prosser [67] calls "conflict-directed backjumping."

```
BACKJUMPING-TOP($P$)    {where $P$ is a CSP of the form $(V, D, C)$}
1   $f$ := the null assignment
2   ⟨answer, conflict-set⟩ := BACKJUMPING($f$,$P$)
3   return answer

BACKJUMPING($f, P$)
1   if $f$ is a total assignment of the variables in $P$
2       return $\langle f, \emptyset \rangle$
3   else
4       $v$ := some variable in $P$ that is not yet assigned a value by $f$
5       answer := UNSAT
6       conflict-set := $\emptyset$
7       for each value $x \in D_v$
8           $f(v) := x$
9           if $f$ satisfies the constraints in $P$
10              ⟨answer, new-conflicts⟩ := BACKJUMPING($f, P$)
11          else
12              new-conflicts := the set of variables in a violated constraint
13          if answer $\neq$ UNSAT
14              return ⟨answer, $\emptyset$⟩
15          else if $v \notin$ new-conflicts
16              return ⟨UNSAT, new-conflicts⟩
17          else
18              conflict-set := conflict-set $\cup$ (new-conflicts $- \{v\}$)
19      return ⟨UNSAT, conflict-set⟩
```

Figure 1.3: Backjumping

values. The first value has the same meaning as before, namely, it is a total assignment that extends f and solves the problem, or the value UNSAT if f cannot be extended to a solution. The second value is the *reason* for the failure, if there is indeed a failure. (If the first value is not UNSAT, then the second value is irrelevant; by convention, we will set it to $\emptyset$ in that case.) By a reason for a failure we mean a subset of the variables bound by f such that there is provably no solution with these variables having their current values. We will call this set the *conflict set*.

How is the conflict set computed? After the variable v is selected on line 4, the loop on lines 7–18 iterates through all the values in the domain D_v. If the constraint check on line 9 fails for a given value, this means that there is some constraint (W, Q) that prevents v from taking that value; in that case, on line 12, the variable *new-conflicts* is set to W, the set of variables in the constraint. If the constraint check on line 9 succeeds, but the recursive invocation of BACKJUMPING fails, then the *new-conflicts* will be returned by this recursive call. If the current variable is not one of the new conflict variables (this can only happen in the case of the recursive call), then the procedure immediately returns (lines 15–16), and pops up the stack in this fashion until it reaches a relevant variable. If there is never an opportunity to "backjump" in this fashion — and if none of the values lead to a solution — then the *conflict-set* that is updated on line 18 will in the end contain all of the preceding variables that were involved in any of the failures. The conflict set will be returned on line 19.

1.2.4 Dependency-Directed Backtracking

While backjumping is more powerful than simple backtracking, it still cannot avoid all thrashing. Let us suppose, once again, that the variable order is $v_1, v_2, \ldots, v_{100}$, and the allowed values are $\{1, 2\}$. Assume that there is some subset of the constraints involving only the variables $v_{50}, v_{51}, \ldots, v_{100}$ that together imply that v_{50} cannot have the value 1. Assume further that these are the only constraints that are used once v_{50} has been set to 1. So at some point BACKJUMPING is invoked with the partial assignment that assigns values to the variables v_1 through v_{50} (with v_{50} being assigned 1), and after some (possibly) huge amount of search, it returns the values

$\langle$UNSAT, $\{v_{50}\}\rangle$. It has conclusively proven that:

$$v_{50} \neq 1 \tag{1.2}$$

So what does BACKJUMPING do? It sets v_{50} to be 2, and then completely forgets this new piece of information! If the algorithm subsequently backtracks to v_{49}, and then returns to v_{50}, it will have to discover (1.2) all over again, perhaps as many as 2^{49} times altogether.

Dependency-directed backtracking, which was introduced by Stallman and Sussman [79], is designed to address this problem. We will call formulas like (1.2) *nogoods*, and the dependency-directed backtracking routine in Figure 1.4 will accumulate them over the course of the search. The recursive procedure DDB will take three arguments: a partial assignment f, a CSP P, and a set of nogoods Γ; and it will return three values: the answer, the conflict set, and the new set of nogoods (alternatively, the nogood set Γ could just be a global variable).

Each time a backtrack is about to occur, the function MAKE-NOGOOD learns (on line 19) a nogood γ from the current conflict set and the current assignment f. This nogood asserts that it is not possible for all of the variables in the conflict set to simultaneously have their current values. For example, if the conflict set consisted of the variables a, b, c, and if for the current partial assignment f, we had $f(a) = 1$, $f(b) = 2$, and $f(c) = 1$, then the nogood γ would be

$$a \neq 1 \vee b \neq 2 \vee c \neq 3. \tag{1.3}$$

Line 20 adds the new nogood γ to the set Γ, and DDB returns Γ as its third value; this ensures that this work will never have to be repeated. The nogoods get used on line 9 of DDB when the current partial assignment is checked not only against the original constraints of the problem P, but also against the new nogoods in Γ that have been learned so far; and if the clash is with a nogood, then the variables in this nogood will be the new conflict variables on line 12 — for the nogood (1.3), the new conflict variables would be a, b, and c.

```
DEPENDENCY-DIRECTED-BACKTRACKING-TOP($P$)
        {where $P$ is a CSP of the form $(V, D, C)$}
1   $f$ := the null assignment
2   $\Gamma := \emptyset$
3   $\langle$answer, conflict-set, $\Gamma\rangle$ := DDB($f$,$P$,$\Gamma$)
4   return answer

DDB($f$, $P$, $\Gamma$)
1   if $f$ is a total assignment of the variables in $P$
2       return $\langle f, \emptyset, \Gamma\rangle$
3   else
4       $v$ := some variable in $P$ that is not yet assigned a value by $f$
5       answer := UNSAT
6       conflict-set := $\emptyset$
7       for each value $x \in D_v$
8           $f(v) := x$
9           if $f$ satisfies the constraints in both $P$ and $\Gamma$
10              $\langle$answer, new-conflicts, $\Gamma\rangle$ := DDB($f$, $P$, $\Gamma$)
11          else
12              new-conflicts := the set of variables in a violated constraint or nogood
13          if answer $\neq$ UNSAT
14              return $\langle$answer, $\emptyset$,$\Gamma\rangle$
15          else if $v \notin$ new-conflicts
16              return $\langle$UNSAT, new-conflicts,$\Gamma\rangle$
17          else
18              conflict-set := conflict-set $\cup$ (new-conflicts $- \{v\}$)
19      $\gamma$ := MAKE-NOGOOD(conflict-set, $f$)
20      $\Gamma := \Gamma \cup \{\gamma\}$
21      return $\langle$UNSAT, conflict-set, $\Gamma\rangle$
```

Figure 1.4: Dependency-directed backtracking

```
k-ORDER LEARNING(P)
            ⋮
19      if |conflict-set| ≤ k
20          $\gamma$ := MAKE-NOGOOD(conflict-set, f)
21          $\Gamma := \Gamma \cup \{\gamma\}$
22      return ⟨UNSAT, conflict-set, $\Gamma$⟩
```

Figure 1.5: *k*-Order Learning

1.2.5 Generalized Dependency-Directed Backjumping

Dependency-directed backtracking would seem to avoid "thrashing" (although it is hard to say, as we never formally defined the term), but at an enormous cost in memory. Every time the procedure backtracks, a new nogood is learned, making the space complexity of dependency-directed backtracking comparable to its time complexity, and since the constraint satisfaction problem is NP-hard, both complexities may be exponential. With current computing technology, space is a scarcer resource than time, and thus we will often exhaust our memory before we exhaust our patience. For this reason, fancy backtracking procedures that use exponential space are often too expensive for practical use.

We might want to use something in between backjumping and full dependency-directed backtracking. One simple idea would be to retain only those nogoods up to a certain size, say k. That is, if the number of variables in the conflict set is less than or equal to k, then we save the nogood in Γ; otherwise, we discard it. Let us call this approach *k-order learning* [28]. Figure 1.5 shows the new section of this algorithm (the rest is the same as dependency-directed backtracking). For any fixed k, the k-ORDER LEARNING procedure uses only polynomial space since if there are n variables and the maximum domain size is a, then the total number of possible nogoods of size at most k is

$$\sum_{i=0}^{k} \binom{n}{i} a^i.$$

Both k-order learning and full dependency-directed backtracking are special cases

```
GENERALIZED-DDB($P$)
            ⋮
19      $\gamma$ := MAKE-NOGOOD(conflict-set, $f$)
20      $\Gamma$ := UPDATE-NOGOODS($\Gamma, \gamma$)
21      return ⟨UNSAT, conflict-set, $\Gamma$⟩
```

Figure 1.6: Generalized dependency-directed backtracking

of a more general approach, *generalized dependency-directed backtracking.* The new part of GENERALIZED-DDB is presented in Figure 1.6; the crucial step is on line 20 where UPDATE-NOGOODS computes a new nogood set from the old nogood set Γ and the new nogood γ. In general, we will require that

$$\text{UPDATE-NOGOODS}(\Gamma, \gamma) \subseteq \Gamma \cup \{\gamma\}. \tag{1.4}$$

In other words, γ is the only new nogood that may be added, but any number of old nogoods may be deleted. This allows for the possibility of "forgetting" old nogoods that have become irrelevant in order to make room for new ones that are now of more importance.

Of special interest are those generalized dependency-directed backtracking procedures that can guarantee that the nogood set Γ will never exceed some maximum size, where this maximum size is polynomial in the size of the CSP. We will call such a procedure a *polynomial-space dependency-directed backtracking* algorithm; k-order learning is one such algorithm, but obviously there are many other possibilities as well.

1.2.6 Dynamic Backtracking

Let us discuss one last algorithm, Ginsberg's *dynamic backtracking* [42]. The motivation for dynamic backtracking is similar (but not identical) to the motivation for dependency-directed backtracking. Recall the example from the beginning of Section 1.2.3 where after instantiating the variables v_1 through v_{99}, we discovered that the value of v_1 did not not allow any consistent assignment to v_{100}. Backtracking wastes its time trying out new values for variables v_{99} back to v_2 before finally fixing the real problem at v_1. Backjumping is more intelligent since it returns directly to v_1, but it is still not a perfect solution. The problem is that as backjumping pops up the stack, it discards the values for variables v_{99} through v_2. This seems rather silly since for all we know, v_2 through v_{99} may be a completely independent problem from v_1 and v_{100}. It might be better to simply go back and revise the value of v_1 without erasing the work for the intervening variables, and this is exactly what dynamic backtracking does.[4]

Because of this ability to preserve the intermediate work, dynamic backtracking cannot easily be written in the recursive depth-first style that we have been using for the other approaches; it is most naturally expressed as an iterative algorithm. The DYNAMIC-BACKTRACKING procedure is presented in Figure 1.7. Since the algorithm is iterative, it cannot use the stack to keep track of its place in the search space, so it uses a two-dimensional *culprits* array. For each variable v and value x (where x is in the domain D_v), $culprits[v, x]$ will be the set of variables whose current assignments preclude v from being set to x; if $v = x$ is currently allowed, then $culprits[v, x]$ will be set to the constant VALUEALLOWED.

The DYNAMIC-BACKTRACKING procedure begins by initializing the whole *culprits* array to VALUEALLOWED; it then enters its main loop. After choosing an unbound variable v (on line 5), it checks each value of v against the current constraints. If there are values for v that are currently allowed, but would violate constraints, then the *culprits* array is updated (on lines 6–9) to disallow these values and to record for each value the reason why it is disallowed. If there is any value left, then v is set to this value, and DYNAMIC-BACKTRACKING returns to the top of its loop. Otherwise, the algorithm backtracks as follows.

It first computes the *conflict-set* on line 13. As long as all of the variables in this set have their current values, it will be impossible to find any consistent assignment for

[4]Like backjumping, dependency-directed backtracking also discards the values for the intervening variables, but since it is saving all the dependency information in its nogood set, it can quickly recover its position in the search space. One motivation for dynamic backtracking is to try to enjoy some of the advantages of dependency-directed backtracking while using only polynomial space.

v. If *conflict-set* is empty, then the problem is unsatisfiable. Otherwise, DYNAMIC-BACKTRACKING selects the most recently bound variable w in the conflict set and prepares to unbind this variable. This corresponds to backjumping from v to w, but without erasing any intermediate values. The one tricky thing is that w may be one of the culprits for another variable. If for some variable u and value x, we have

$$w \in \mathit{culprits}[u, x],$$

then this indicates that the current assignment for w is one of the reasons that u cannot have the value x. Since we are about to go back and erase the value for w, we have to delete this *culprits* information, and set *culprits*$[u, x]$ back to VALUE-ALLOWED (lines 18–20). We then proceed to erase the value of w (line 22) after recording the justification for the backtrack (line 21). The loop continues until we have either constructed a total assignment or proven the problem to be unsatisfiable.

It may not be immediately obvious that DYNAMIC-BACKTRACKING terminates. Certainly if it terminates, it will return a correct answer, either a solution or UNSAT, but how do we know that it will not loop endlessly? Ginsberg [42] proves that dynamic backtracking will in fact always terminate, but going over the proof here would take us too far afield. We will simply note that the key step in his proof relies on the fact that the backtrack variable w chosen on line 17 is always that variable in the conflict set that was most recently bound; as Ginsberg notes, if we instead chose an arbitrary variable from the conflict set, then dynamic backtracking could get stuck in an infinite loop.

One interesting property that dynamic backtracking shares with full dependency-directed backtracking is that of *compositionality*; if a CSP is the union of several independent subproblems, then dynamic backtracking will solve the CSP in time that is the sum (up to a polynomial factor) of the times required to solve the subproblems individually. By contrast, backtracking and backjumping can take time that is the *product* of the individual times. McAllester [61] calls procedures with this compositionality property "aggressive dependency-directed backtracking" algorithms, and he notes that dynamic backtracking appears to be the first such procedure that is polynomial space.

```
DYNAMIC-BACKTRACKING($P$)    {where $P$ is a CSP of the form $(V, D, C)$}
    $f$ := the null assignment
    for each variable $v$, and each $x \in D_v$
        $culprits[v, x]$ := VALUEALLOWED
    loop until $f$ is a total assignment of the variables in $P$
        $v$ := some variable in $P$ that is not yet assigned a value by $f$
        for each value $x \in D_v$
            if $culprits[v, x]$ = VALUEALLOWED
                if setting $f(v) = x$ would violate a constraint in $P$
                    $culprits[v, x]$ := the variables other than $v$ in that violated constraint
        if there is some $x$ such that $culprits[v, x]$ = VALUEALLOWED
            $f(v) := x$
        else
            $conflict\text{-}set := \bigcup_{x \in D_v} culprits[v, x]$
            if $conflict\text{-}set = \emptyset$
                return UNSAT
            else
                $w$ := the variable in $conflict\text{-}set$ that was most recently bound
                for each variable $u$, and each $x \in D_u$
                    if $w \in culprits[u, x]$
                        $culprits[u, x]$ := VALUEALLOWED
                $culprits[w, f(w)] := conflict\text{-}set - \{w\}$
                Remove the binding of $f(w)$
    return $f$
```

Figure 1.7: Dynamic backtracking

1.3 Overview of the Rest of the Thesis

We have described a number of constraint satisfaction algorithms. The question is how well they perform. There are various ways of answering this question.

One approach is to study the performance empirically on randomly-generated problems. We will take this approach in Chapter 2 using random graph coloring problems as our benchmark. We will be particularly interested in how the various algorithms perform as we vary the number of edges in the graph. If there are only a small number of edges, then there will probably be a solution to the graph coloring problem; furthermore, it will usually be easy to find this solution. If there are too many edges, then there probably will not be a solution, and it will often be fairly easy to prove this. Somewhere in between is the "crossover" point, where around half of the problems have solutions. A number of previous researchers have presented evidence that the hardest constraint satisfaction problems tend to arise near this crossover point. Some recent papers, however, argue that while this is true of the median search cost, the mean search cost has an entirely different behavior. The mean search cost appears to be dominated by a relatively small number of extremely difficult problems, and these problems typically have constraint graphs that are much sparser than those of the problems near the crossover point.

In Chapter 2, we will show that these outliers among the sparse graphs are not fundamentally difficult problems. They are hard for chronological backtracking only because of thrashing, and they are easily handled by more sophisticated search strategies such as backjumping and dependency-directed backtracking. Thus, they are quite different in character from the crossover-point problems, which appear to be difficult for all known search algorithms. We will present our experimental results and discuss their implications.

Chapter 3 will cover some experiments with dynamic backtracking. The point of dynamic backtracking is that the search procedure should be able to return to the source of a difficulty without erasing the intermediate work. For this chapter, we will use randomly-generated propositional satisfiability problems as our benchmark. We will see that dynamic backtracking can sometimes be counterproductive, and in fact can cause an exponential increase in the size of the ultimate search space (compared to backtracking or backjumping). We will discuss the reason for this phenomenon, and we will present a modification to dynamic backtracking that fixes the problem.

Another approach to studying a constraint satisfaction algorithm is to analyze its worst-case performance theoretically. Since constraint satisfaction is NP-hard, one cannot hope to have an algorithm that runs in polynomial time for all problems. We will do our analysis in terms of a certain problem parameter known as the "induced width" [30]. This concept will be discussed in detail in Chapter 4, but for now, the reader should simply think of the induced width as a measure of the complexity of the constraint graph, where the constraint graph is formed by placing edges between any problem variables that share constraints. For a fully connected constraint graph with n variables (that is, for a problem where every variable interacts with every other variable), the induced width will be $n-1$. For a problem where the constraint graph is a simple chain (that is, for a problem where each variable i only interacts with variables $i-1$ and $i+1$), the induced width will be 1. Constraint problems of intermediate complexity will have intermediate induced widths.

It is well known that the time complexity of dependency-directed backtracking is exponential only in the induced width. That is, its complexity can be written as the problem size raised to the power of a function of the induced width. This is better than simply being exponential in the size of the problem. It means that dependency-directed backtracking will run in polynomial time for any problem class of bounded induced width. Unfortunately, the space requirements of dependency-directed backtracking are also exponential in the induced width. As remarked earlier, with present computing technology, space is often a dearer resource than time; for example, a typical workstation can easily write to all of its fast memory in a few minutes. Therefore, there is particular interest in constraint algorithms that use only polynomial space. Now, standard backtracking uses only polynomial space (in fact, it needs only linear space), but it can require time exponential in the size of the problem. These possibilities are displayed in Table 1.1, where x is the size of the problem, w^* is the induced width, c is some constant, and f is an arbitrary function (of course, the minimal values of c and f will not be the same for the different entries

Algorithm	Space	Time
Chronological backtracking	x^c	$Exp(x)$
Dependency-directed backtracking	$x^{f(w^*)}$	$x^{f(w^*)}$
?	x^c	$x^{f(w^*)}$

Table 1.1: Space and time complexity with respect to the induced width

in the table). The question suggested by the last row of Table 1.1 is whether there is a third possibility that combines the advantages of the other two approaches. Is there a polynomial-space CSP algorithm that is exponential only in the induced width?

Our conjecture is that no such algorithm is possible. We will not, however, be able to prove this conjecture in its full generality. For one thing, if the conjecture were true, it would immediately imply that P $\neq$ NP since a polynomial algorithm is trivially exponential only in the induced width. Presumably, then, any reasonable proof of our conjecture would have to assume P $\neq$ NP, and perhaps make other fundamental assumptions as well.

Instead, we will begin in Chapter 4 by examining the various polynomial-space intelligent backtracking algorithms from Section 1.2. We show that none of the following algorithms are exponential only in the induced width: backtracking, backjumping, dynamic backtracking, and k-order learning. Finally, we prove that there is no polynomial-space dependency-directed backtracking procedure (in the sense of Section 1.2.5) that is exponential only in the induced width.

In Chapter 4, we only consider search algorithms that branch on their variables using a fixed, static order. In Chapter 5 we will consider algorithms that use an optimal, dynamic order. That is, at each branch point, they can choose an arbitrary unassigned variable to branch on next, and to give these algorithms every benefit of the doubt, we will assume that they make these decisions optimally. We will show that even with these assumptions, the following algorithms are not exponential only in the induced width: backtracking, backjumping, dynamic backtracking, and k-order learning. Finally, we state (but do not prove) a more general conjecture concerning polynomial-space resolution procedures.

Chapter 6 surveys the CSP literature and discusses related work. Chapter 7 contains concluding remarks and some suggestions for future research.

Chapter 2

Intelligent Backtracking on the Hardest Constraint Problems

2.1 Introduction

In this chapter, we begin our experimental investigation into intelligent backtracking.[1] There has been much recent interest in the empirical study of constraint satisfaction problems, and several important lessons have emerged. These lessons concern the relationships between the number of constraints in a problem, the probability that the problem has a solution, and the typical difficulty of such a problem for a search algorithm [13, 23, 63].

Suppose we randomly generate constraint satisfaction problems of some particular type (e.g., graph coloring). The probability of there being a solution will depend on the distribution used to generate these problems. If the problems only have a few constraints, then they are very likely to have solutions. Such a distribution is often referred to as being *underconstrained.* On the other hand, if there are too many constraints, there probably will not be solutions; these problems are *overconstrained.* Somewhere in the middle is the *crossover point* at which half of the problems are solvable and half are unsolvable. One important lesson is that the transition from the underconstrained region to the overconstrained region is often very abrupt [13, 15, 23].

[1]This chapter is a revised version of [6].

For sufficiently large problems, increasing the number of constraints by even a few percent will drastically reduce the probability of there being a solution.

The second important lesson concerns the difficulty of the problems for a search algorithm. We assume that the task of the search algorithm is either to find a solution if one exists, or otherwise to prove that the problem is unsolvable. Underconstrained problems are typically quite easy; there are so few constraints that it is usually not difficult to find a solution. Overconstrained problems are also relatively easy; there are so many constraints that a backtracking algorithm will be able to quickly rule out all the possibilities. The hardest problems appear to arise at the crossover point. For sufficiently large problems, the crossover-point examples can be orders of magnitude more difficult than those in either the underconstrained or overconstrained regions [13, 23, 63].

In a recent paper, however, Hogg and Williams [51] offer a third lesson. In a careful study of graph coloring, they show that the distribution of problem difficulty is more subtle than previously believed. Although *most* of the underconstrained problems are quite easy, there are a small number of outliers among them that are extremely hard — so hard, in fact, that the peak of the *mean* difficulty (unlike that of the median) appears to be located well into the underconstrained region. Gent and Walsh [41] have reported similar results for the problem of propositional satisfiability.

On the face of it, this phenomenon is somewhat counterintuitive, and neither Hogg and Williams nor Gent and Walsh provide any explanation. In this chapter, we will show that the outliers in the underconstrained region are not fundamentally difficult problems. Rather, they are difficult only for the particular search algorithm used by the above authors: chronological backtracking. Chronological backtracking sometimes suffers from a malady known as "thrashing," in which the search algorithm ends up solving essentially the same subproblem over and over again. Thrashing can be reduced or eliminated by more sophisticated search algorithms such as backjumping [40] and dependency-directed backtracking [79]. Thus, these "hard" problems in the underconstrained region are different in character from the crossover-point problems, which appear to be difficult for all known search algorithms.

The organization of this chapter is as follows. We begin in the next section by reviewing the backtracking experiments from Hogg and Williams [51] that demonstrate the existence of inordinately difficult underconstrained problems. In Section 2.3, we explain what is going wrong with backtracking in these experiments. Our analysis suggests that backjumping should ameliorate the problem, and we will present experimental results confirming this conjecture. It turns out that while backjumping helps a great deal, it still seems to leave room for further improvement. We will see in Section 2.4, however, that dependency-directed backtracking completely eliminates the problem (at least, up to the precision of our experiments). Finally, in Section 2.5, we will discuss the meaning and significance of these findings.

2.2 Graph Coloring and Backtracking

Given a graph and a fixed number of colors, the Graph Coloring Problem is to assign a color to each vertex so that no two adjacent vertices (i.e., vertices joined by an edge) are assigned the same color. Graph coloring is a well-known NP-complete problem [37] that is of some interest in artificial intelligence and operations research; scheduling, for example, can sometimes be reduced to graph coloring. Hogg and Williams [51] used graph coloring for their constraint satisfaction experiments, so in order for our results to be comparable with theirs, we also use graph coloring.

In fact, we follow the experimental setup from [51] as closely as possible. All of the problems are 3-coloring problems, that is, there are three different colors available. The graphs all have 100 vertices. The number of edges will be described in terms of the average degree γ of the graph (i.e., the average number of edges incident on a vertex). If there are n vertices and e edges, then we have $e = \gamma n/2$ (since each edge is incident on two vertices). For our experiments, the average degree of the graph was varied from 1.0 to 7.0 in increments of 0.1, and we generated 50,000 problems at each point (for a total of 3,050,000 problems altogether). Each problem was randomly generated using the uniform distribution over all graphs with the appropriate number of vertices and edges.

Following [51], we use a standard backtracking search (i.e., depth-first search)

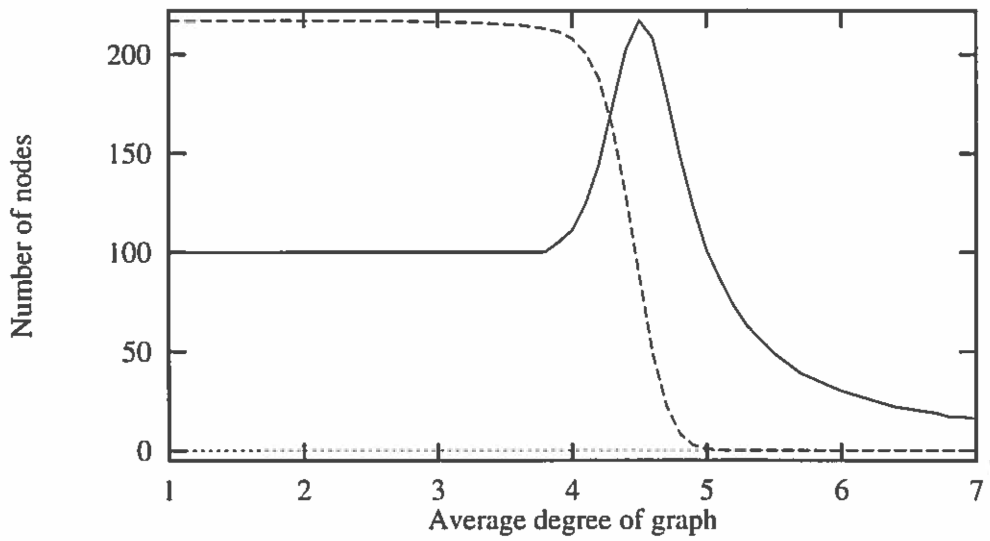

Figure 2.1: The solid curve is the median search cost for backtracking; the dashed curve is the percentage of problems with a solution (scaled to range from 100% on the left to 0% on the right).

guided by the Brélaz heuristic [9, 82]. The Brélaz heuristic is a dynamic variable-ordering rule that chooses the vertex with the smallest number of available colors (or, in other words, with the greatest number of distinctly colored neighbors). Ties are broken by preferring nodes with the most uncolored neighbors, and further ties are broken arbitrarily.[2] No value-ordering heuristic is used; the colors are simply tried in order.

We search in this manner until we find the first solution, or until all possibilities have been exhausted (if the problem is unsolvable). Actually, we never backtrack past the first two vertices; if the initial colors for these two vertices do not work, then there cannot be a solution since graph coloring is symmetric under permutations of the colors. We use the number of nodes explored in this backtracking search tree as our measure of problem difficulty. The results of these experiments are displayed in Figures 2.1–2.5.

Figure 2.1 shows the median search cost of the backtracking algorithm plotted as

[2]We simply choose the first vertex in some arbitrary order. In [51], each tie is broken randomly; this small difference does not seem to have affected our results.

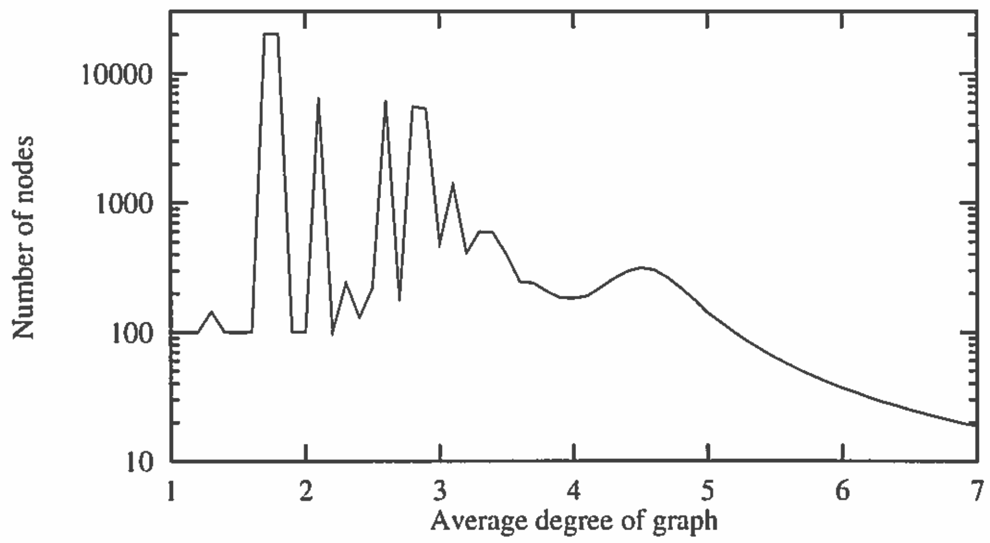

Figure 2.2: The mean search cost for backtracking.

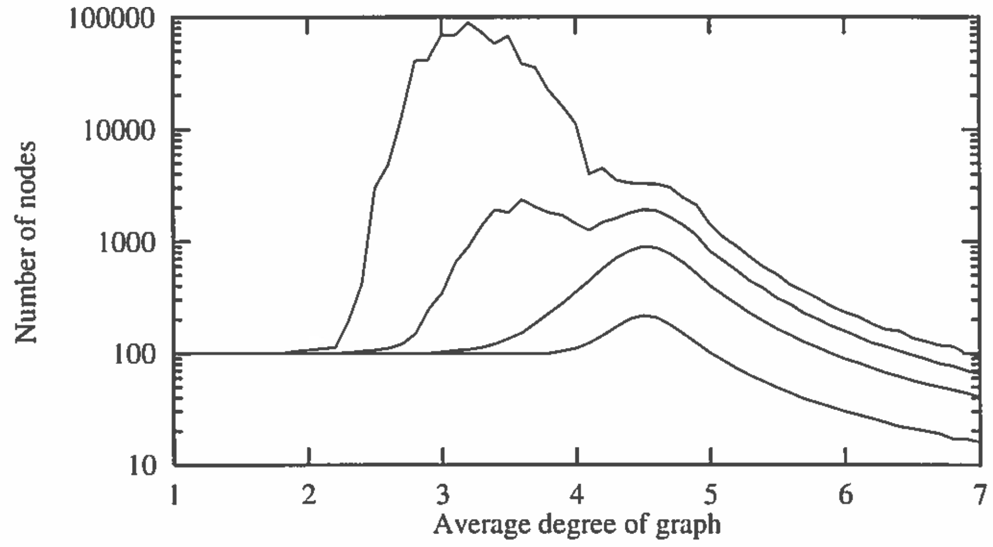

Figure 2.3: Various percentiles of the search cost for backtracking. The lowest curve is the 50% level, followed by the 95%, 99.5%, and 99.95% levels.

a function of the average degree γ of the graph. This data is superimposed on a plot of the fraction of the problems having solutions. The crossover point with respect to γ is somewhere between 4.4 and 4.5, and this is also where the median search cost peaks.[3] This important phenomenon of the hardest constraint satisfaction problems being those at the crossover point has been noted by a number of authors [13, 23, 63]. These authors have gone on to investigate how this behavior scales with increasing problem size; they have shown, among other things, that the difficulty at the crossover point appears to grow exponentially.

Hogg and Williams [51] (and Gent and Walsh [41] for propositional satisfiability), however, took a different approach by instead studying in more depth the results for a fixed problem size. After all, the median is only one particular statistic. One might wonder, for example, what a plot of the mean would look like. The mean search cost is displayed in Figure 2.2, and it is clear that the mean is quite different from the median. While the median peaks at the crossover point, the mean seems to be substantially larger in the underconstrained region. Most of the underconstrained problems are quite easy, but there are also some extremely difficult ones. These sample means are also quite volatile; apparently, 50,000 experiments are not sufficient to accurately estimate the mean.

To give a better sense of the distribution of problem difficulty, Figure 2.3 shows several percentiles: the 50% point (the median), the 95% point (i.e., that number such that 95% of the problems used fewer nodes, and only 5% required more nodes) the 99.5% point, and the 99.95% point. As the percentile is increased, the peak shifts to the left. The median peaks at the crossover point, but the 99.95% point is more than an order of magnitude greater at $\gamma = 3.0$ than it is at $\gamma = 4.5$. This behavior is even more dramatic if we examine the *maximum* at each point (Figure 2.4). The hardest of the 50,000 examples at $\gamma = 4.5$ required only 6,186 nodes, while both $\gamma = 1.7$ and $\gamma = 1.8$ each had a problem that could not be solved in a billion nodes.

Figure 2.5 presents another view of this data. It compares the full distribution

[3]The reader might be curious why the median search cost for the underconstrained problems is exactly 100 nodes, while the overconstrained problems seem to be easier. The answer is that if a problem is solvable, then we have to assign a color to each vertex (and there are 100 vertices here), so even if we never backtrack, we will still need 100 nodes in our search tree.

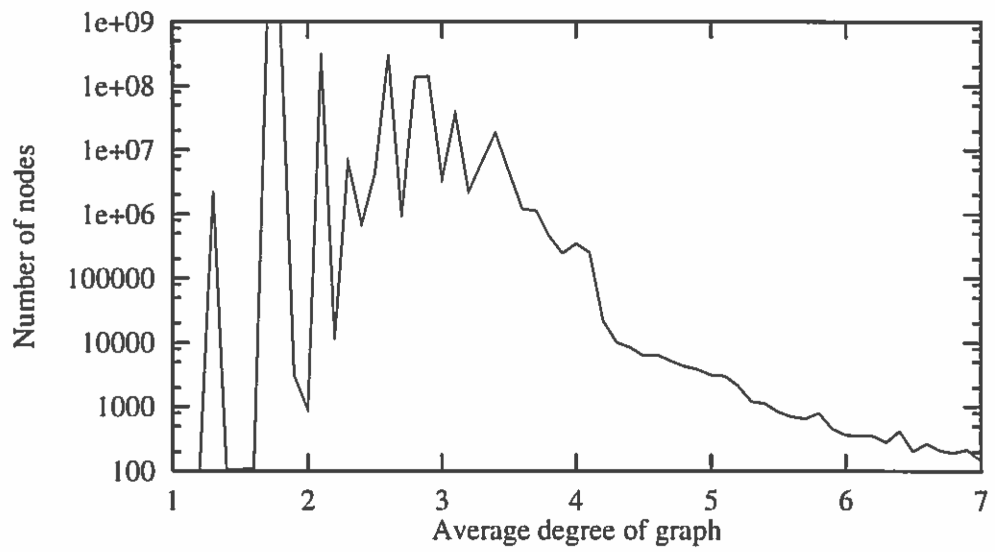

Figure 2.4: The maximum search cost for backtracking (out of the 50,000 samples per data point).

for the underconstrained problems at $\gamma = 3.0$ with that of the crossover problems at $\gamma = 4.5$. For each search cost in Figure 2.5, the ordinate represents the fraction of problems that used at least that number of nodes. We see that the probability of an example being moderately difficult is higher for the crossover problems, but the probability of an example being extremely difficult is higher for the underconstrained problems.

So far we have merely reproduced the results from [51]. We will now discuss what is going wrong with these difficult underconstrained problems, and we will show that properly handled, the problems are not really difficult at all.

2.3 Backjumping

The argument that the crossover-point problems should be the hardest is a fairly intuitive one. If there are too few constraints, then almost any path down the search tree that has not yet hit a dead end (i.e, a constraint violation) will ultimately lead to a solution; hence, we will not have to backtrack very much. If there are too many constraints, then almost any path down the search tree will quickly hit a dead end,

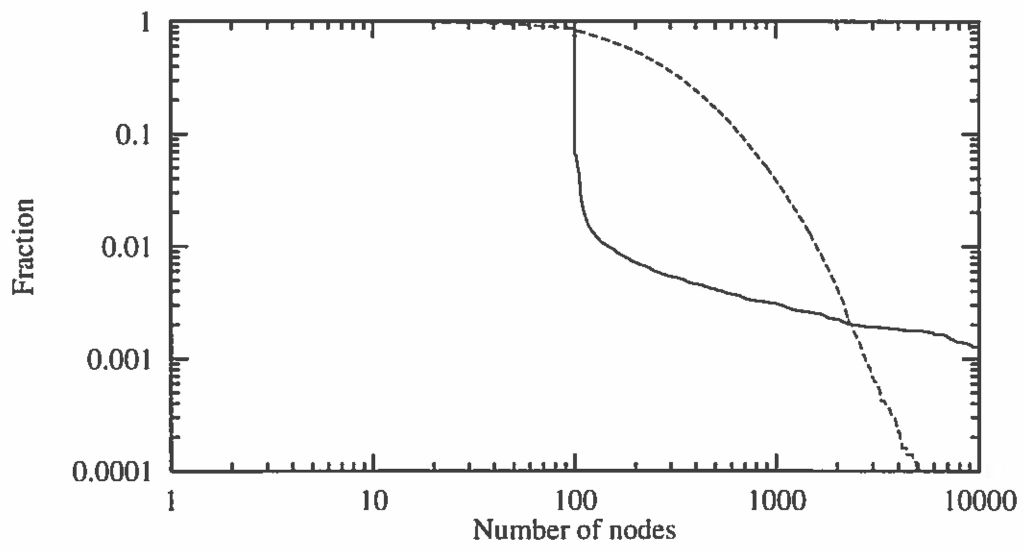

Figure 2.5: The distribution of search costs for backtracking for $\gamma = 3$ (solid curve) and $\gamma = 4.5$ (dashed curve).

and thus it will be easy to exhaustively search the tree. At the crossover point, however, there will be many dead ends that manifest themselves deep in the search tree. In other words, there will be many partial solution that initially seem promising, but for complicated reasons cannot be extended to total solutions.

What about the hard underconstrained problems? Our conjecture is that these problems are difficult solely as a result of *thrashing.* To understand how thrashing might apply to graph coloring, consider an extreme example: the problem of 3-coloring a 100-node graph, with the graph consisting of 96 isolated vertices and 4 vertices connected in a clique, that is, with an edge between every 2 of these 4 vertices. (Since this graph has only 6 edges, it could not have been generated in any of the experiments in this chapter, but it will serve to explain the basic idea.) Suppose that some backtracking program begins by assigning colors to all the isolated vertices, and then attempts to color the clique. It is impossible to color a 4-clique using only 3 colors, so the program would be forced to backtrack. It would assign a new color to the last isolated vertex, and then attempt once again to color the 4-clique, and of course it would fail once again. Continuing in this fashion, the backtracking program would repeatedly fail for the same reason as it proceeded through the different colorings of the isolated vertices. Thus, an easy problem can be very difficult for backtracking.

Now, the Brélaz heuristic would not have any trouble with this *particular* example since the heuristic would always begin with the vertices in the clique instead of the isolated vertices. We cannot, however, expect any heuristic to be perfect. Suppose some graph consisted of two connected components, one of which was 3-colorable and one of which was not. If the colorable component had a maximum degree exceeding that of the non-colorable component, then Brélaz would end up doing the colorable component first, and thrashing would arise just as before.

It should be clear from this discussion why thrashing is a particular problem in the underconstrained region. The backtracking program is wasting its time enumerating all of the solutions to some irrelevant subproblem, and if there are not many constraints, then this subproblem is likely to have many solutions. If the problem were more constrained, then the irrelevant subproblems would tend to have fewer solutions, and thrashing would be less harmful. It also seems likely that even on the underconstrained problems, thrashing will be relatively rare, but that when it occurs, it can be arbitrarily costly. Thus, thrashing is a plausible explanation of the results reported by Hogg and Williams [51] and Gent and Walsh [41]. To show that it is in fact the correct explanation, let us consider some experiments with a more intelligent search algorithm.

One simple method to deal with the particular examples of thrashing discussed above would be to first identify the connected components of the graph and then backtrack on each component separately. It turns out, however, that there are slightly more complicated types of thrashing behavior that would still exist even with this method. It is better, therefore, to proceed in a more systematic fashion. We will use *backjumping* [28, 40, 42, 67].

For the full backjumping algorithm, the reader should consult Figure 1.3 (and the accompanying discussion) in Chapter 1. Here, we will briefly indicate how backjumping applies to graph coloring. With ordinary backtracking, when you have to back up in the search tree, you back up one step and try a new value for the previous variable; that is why it is sometimes referred to as *chronological* backtracking. With backjumping, you instead "jump back" to the last variable that was relevant to the dead end, skipping over the irrelevant intervening variables. For example, if it were determined that one component of the graph is not colorable, then the algorithm would jump all the way back to the beginning of the problem (and declare it unsolvable) since the other components cannot possible be relevant. The variables that are relevant to a dead end are referred to as the *conflict set*, and this set can be computed as follows.

Suppose we have assigned colors to some subset of the vertices, and we have now selected some new vertex v to color. Let η represent the node in the search tree corresponding to this task. If the vertex has no possible color, this means that some adjacent vertex v_1 uses the first color, some adjacent vertex v_2 uses the second color, and some adjacent vertex v_3 uses the third color. The conflict set, in this case, is simply $\{v_1, v_2, v_3\}$. If, on the other hand, none of these colors have been ruled out, then the node η will have three children in the search tree, corresponding to the different ways to color the vertex v. If all these nodes ultimately fail, then the conflict set for η can be obtained by taking the union of the conflict sets of the children, after subtracting out the vertex v. Of course, if v is not present in one of these conflict sets, then this means that the particular color of v was not relevant to the contradiction, and the algorithm will immediately jump back to the most recent vertex that was relevant rather than wasting its time examining the other children of η. The intermediate case (where some, but not all, of the colors are initially disallowed) is equally straightforward.

We ran backjumping on the same examples that were used in Section 2.2. Figure 2.6 shows the results for the median and the 95% level for both backtracking and backjumping. Evidently, backjumping does not help much with the easier problems since thrashing is not all that common. Figure 2.7, however, compares the means for backtracking and backjumping. Here, backjumping makes an enormous difference. The large means that were observed in the underconstrained region with backtracking are no longer present with backjumping.

Figure 2.8 compares the 99.5% points for backtracking and backjumping. This is the number such that 99.5% of the problems required fewer nodes, and only 0.5%

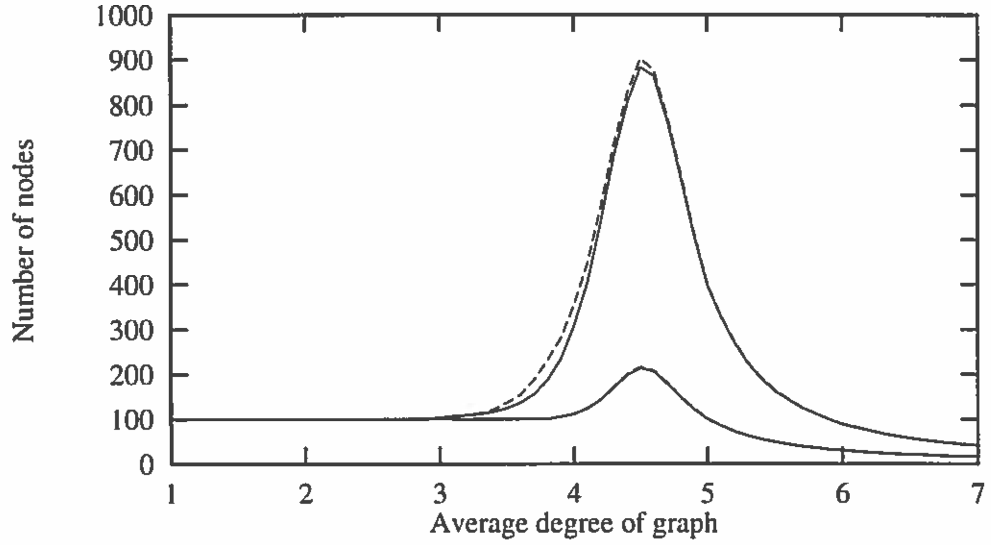

Figure 2.6: The median and 95% level for backjumping (solid curves) and backtracking (dashed curves).

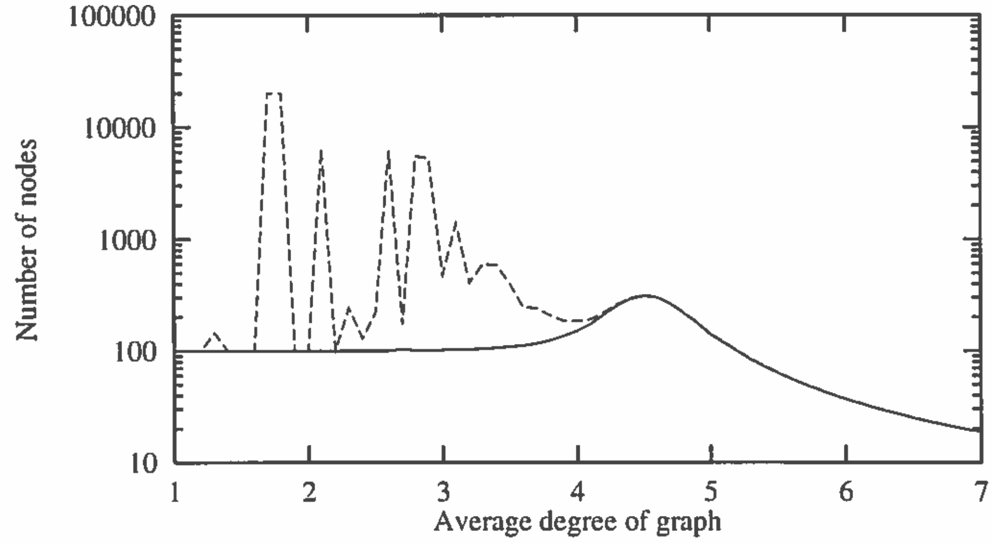

Figure 2.7: The mean for backjumping (solid curve) and backtracking (dashed curve).

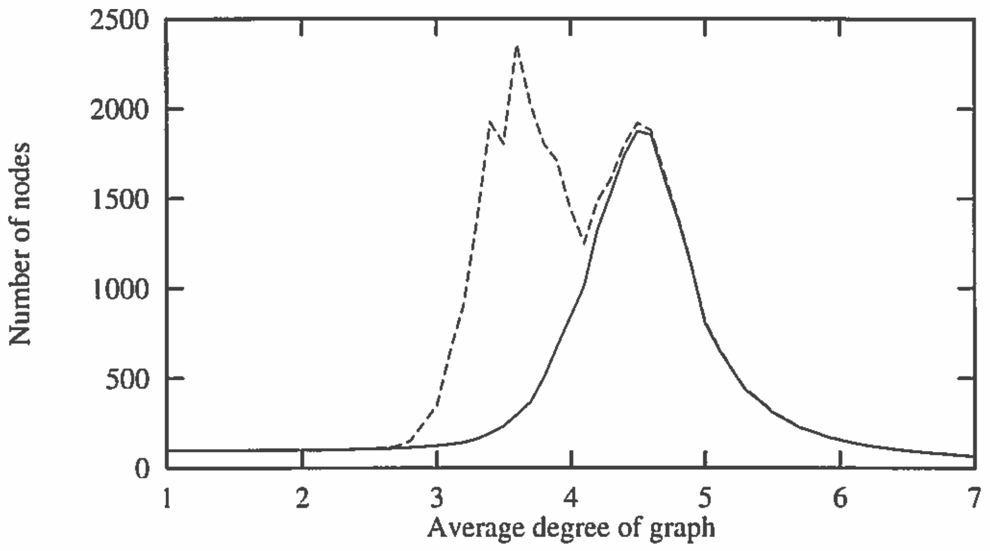

Figure 2.8: The 99.5% level for backjumping (solid curve) and backtracking (dashed curve).

required more. Similarly, Figure 2.9 shows the 99.95% points for both methods. Figures 2.6–2.9 demonstrate that backjumping helps significantly with the hard problems in the underconstrained region, but that it does not have much effect elsewhere. In particular, backjumping does not help much with the hardest problems at the crossover point. This suggests that the hard problems at the crossover point are quite different from the hard underconstrained problems.

While backjumping is a significant improvement over backtracking, it does not completely eliminate the problem of thrashing. Even in Figure 2.9, we can see that the 99.95% curve for backjumping seems similar in some respects to the 99.5% curve for backtracking (see either Figure 2.3 or Figure 2.8). That is, there appears to be a second peak to the left of the crossover point. Figure 2.10 displays the maximum difficulty (out of the 50,000 samples) for each value of γ; although the individual data points are not statistically significant estimates of anything, the graph as a whole strongly suggests that even for backjumping the hardest problems are in the underconstrained region, not at the crossover point.

To further explore this idea, we ran a larger set of backjumping experiments. This time, for each value of γ, we generated a million random problems. Figure 2.11 shows

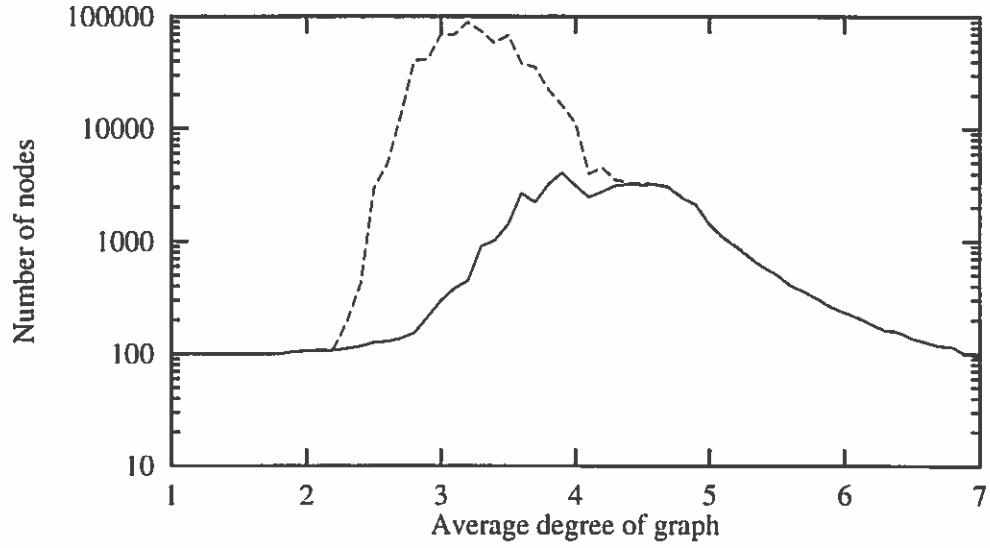

Figure 2.9: The 99.95% level for backjumping (solid curve) and backtracking (dashed curve).

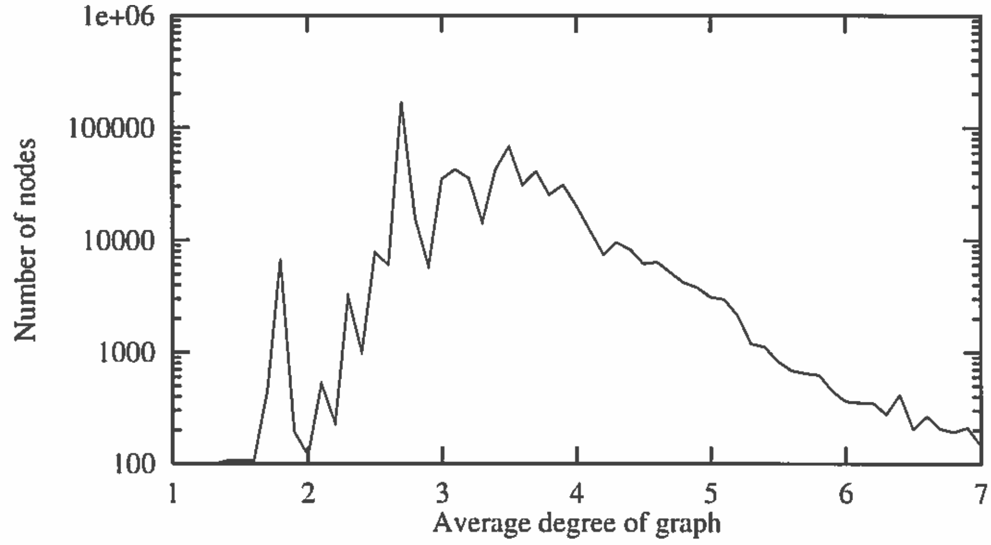

Figure 2.10: The maximum search cost for backjumping (out of the 50,000 samples per data point).

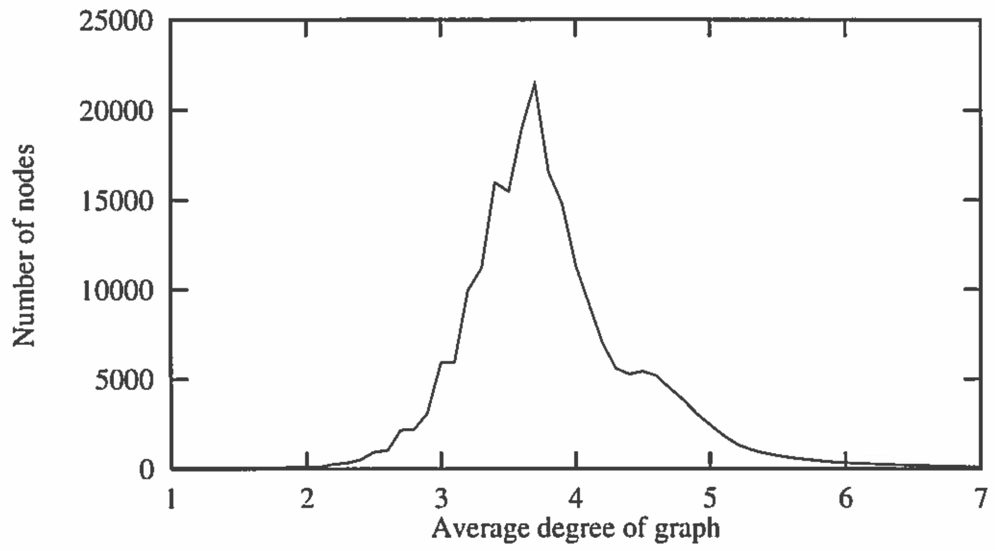

Figure 2.11: The 99.995% Level for Backjumping.

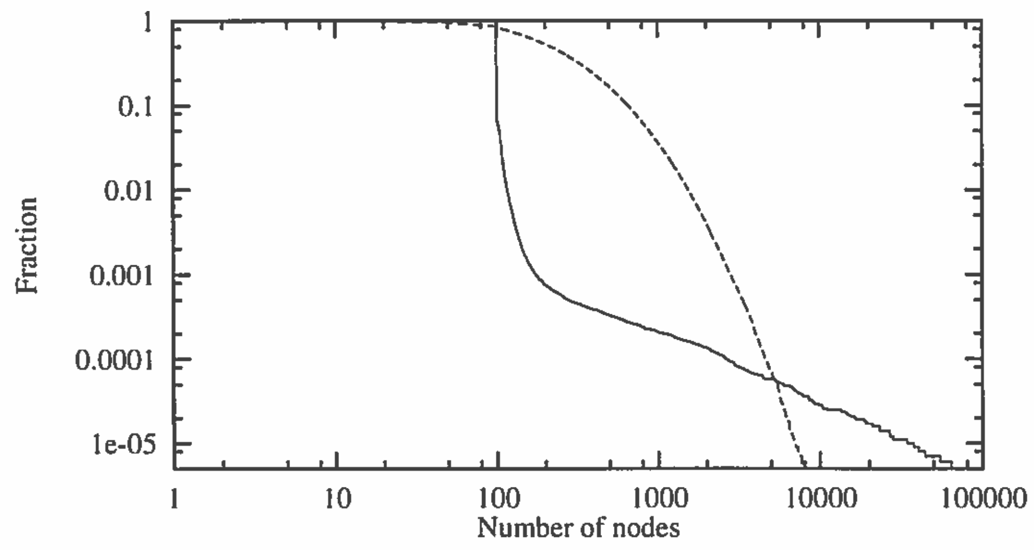

Figure 2.12: The distribution of search costs for backjumping for $\gamma = 3$ (solid curve) and $\gamma = 4.5$ (dashed curve).

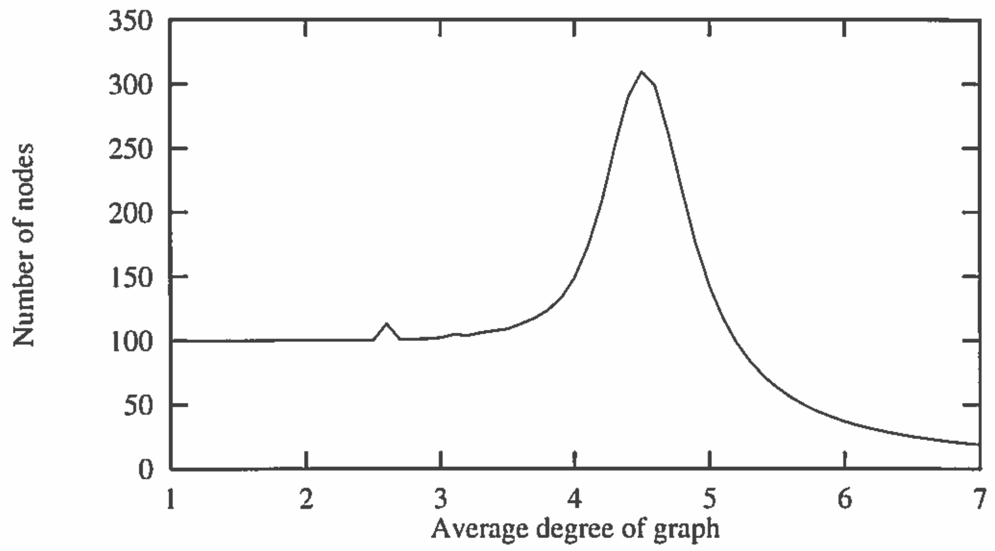

Figure 2.13: The mean search cost for backjumping (computed on the basis of a million samples per data point).

the 99.995% curve computed on the basis of these experiments; it clearly peaks in the underconstrained region. Figure 2.12 shows the full distribution of problem difficulty for these experiments at $\gamma = 3.0$ and $\gamma = 4.5$. It is instructive to compare Figure 2.12 with the analogous graph for backtracking (Figure 2.5). While backjumping has postponed the point at which the sparser problems become harder than the crossover-point problems, it has not altered the qualitative structure of the phenomenon.

Finally, in Figure 2.13, we show the sample means for these larger experiments. With backtracking, there were so many difficult underconstrained problems, that the mean obviously peaked to the left of the crossover point (see Figure 2.2). With backjumping, however, even a million experiments per data point are insufficient to resolve the issue. It is entirely possible that with a billion (or a trillion) experiments per data point, backjumping would begin to exhibit the same phenomenon observed with backtracking. It is also possible that there are just not enough sufficiently hard problems for backjumping, and that the mean continues to peak at the crossover point. In order to decide which is the case, we would have to know (for instance) how far the $\gamma = 3.0$ curve in Figure 2.12 can be extrapolated. It seems likely that the true population mean for backjumping achieves its maximum in the underconstrained region, but we will have to leave this question open for now.

2.4 Dependency-Directed Backtracking

We have seen that almost all of the underconstrained problems that were difficult for backtracking can be solved very easily with backjumping. We have also seen, however, that a few of these problems continue to be very difficult for backjumping. It makes sense, then, to explore even more powerful search algorithms.

When backjumping determines that a node in the search tree is a dead end (perhaps after an extensive search of the node's descendents), it computes the conflict set, the subset of the previously colored vertices whose current colors are relevant to the dead end. Backjumping then jumps back to the last vertex in this conflict set. This is, in fact, the appropriate place to jump back. The problem is that backjumping does not remember this conflict set for future reference. If at some later point in the search, the variables in the conflict set are reassigned their old values, backjumping will have to rediscover the contradiction all over again. In the worst case, this can happen many times.

Dependency-directed backtracking [79] differs from backjumping in that it learns during the course of the search. Each time it jumps back from a dead end, dependency-directed backtracking remembers a *nogood* for future reference. For example, if the conflict set were $\{v_1, v_3, v_7\}$, and the current colors for these vertices were *red*, *green*, and *blue* respectively, then the new nogood would be

$$(v_1 \neq \mathit{red}) \vee (v_3 \neq \mathit{green}) \vee (v_7 \neq \mathit{blue}).$$

This asserts that there is no solution in which these three variables simultaneously have their current values. The new nogood is then treated just like the original problem constraints. At every node, the algorithm will make sure that not only are no two adjacent vertices assigned the same color, but also that none of the nogoods are violated. In this way, dependency-directed backtracking can ensure that it never has to learn the same nogood more than once.[4] For the full dependency-directed

[4]The term "jump-back learning" has also been used to describe this technique [35].

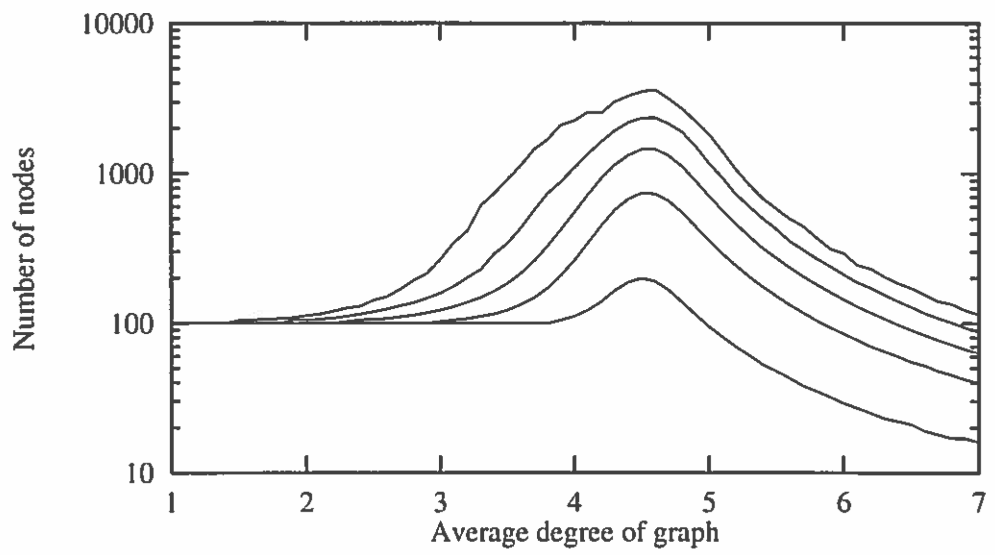

Figure 2.14: The various percentile levels for dependency-directed backtracking: 50%, 95%, 99.5%, 99.95%, and 99.995%.

backtracking algorithm, see Figure 1.4 in Chapter 1.

We ran dependency-directed backtracking on the same randomly generated problems that were used for the larger set of backjumping experiments (a million problems for each value of γ). Figure 2.14 shows the various percentile levels, all of which peak at the crossover point. Figure 2.15 compares the 99.95% level for dependency-directed backtracking with that of backjumping, and Figure 2.16 compares the 99.995% levels for the two algorithms. Dependency-directed backtracking has eliminated the problem that was noted in the last section.

Finally, Figure 2.17 shows the full distribution of search costs for dependency-directed backtracking for $\gamma = 3.0$ and $\gamma = 4.5$; this final graph was computed using an even larger set of experiments, with ten million random problems for each of the two values of γ. Unlike the graphs for backtracking and backjumping (in Figures 2.5 and 2.12 respectively), there is no point at which the sparse problems become more difficult than the crossover-point problems.

These results demonstrate that the phenomenon reported by Hogg and Williams is an artifact of chronological backtracking. The supposedly "hard" underconstrained

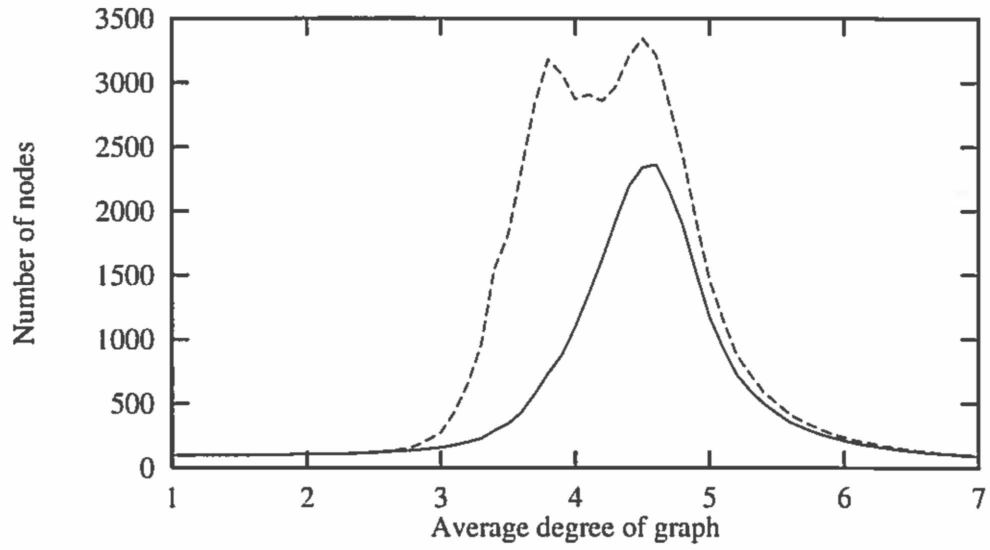

Figure 2.15: The 99.95% level of dependency-directed backtracking (solid curve) and backjumping (dashed curve).

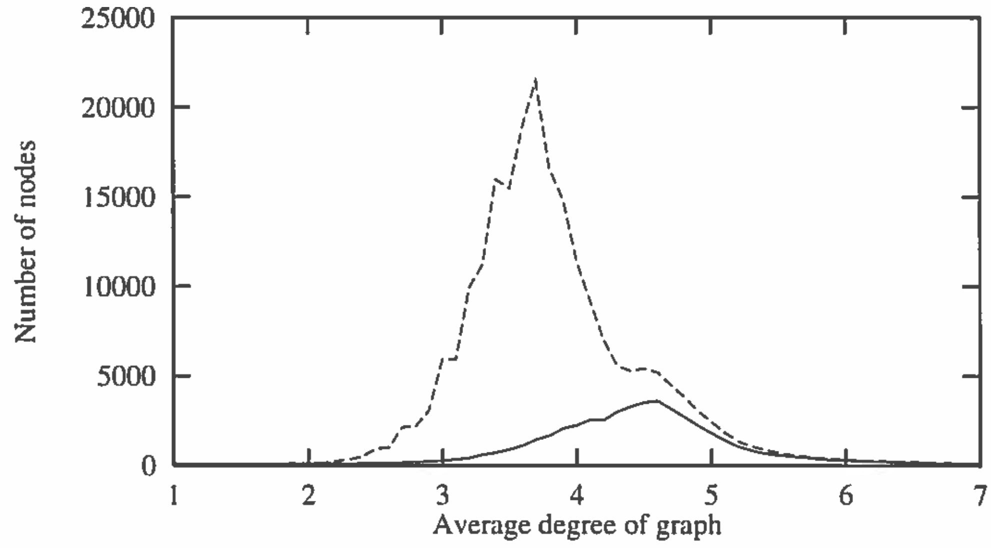

Figure 2.16: The 99.995% level of dependency-directed backtracking (solid curve) and backjumping (dashed curve).

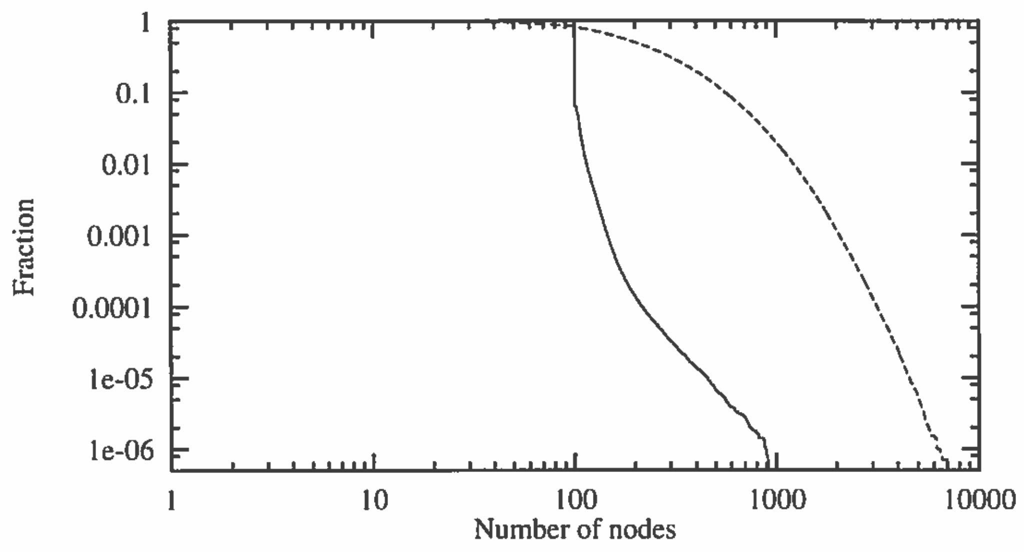

Figure 2.17: The distribution of search costs for dependency-directed backtracking for $\gamma = 3$ (solid curve) and $\gamma = 4.5$ (dashed curve).

problems are not really hard at all, and are actually quite easy for the more sophisticated search strategies. For example, while the two hardest problems for backtracking required more than a billion nodes each, dependency-directed backtracking needed only 78 nodes for one of these problems and 122 nodes for the other.

There remains the question of how well the various algorithms perform in terms of time. This will, of course, depend on implementation details, but we can at least give a general indication. Table 2.1 shows the mean number of nodes, the mean time in seconds, and the mean nodes per second for backtracking, backjumping, and dependency-directed backtracking (DDB). This data was computed using the 3,050,000 problems on which all three of the algorithms were tested. (The algorithms were implemented in C++, and the experiments were performed on a SPARC 10/51.)

Backjumping is taking 41% more time per node here than backtracking, and dependency-directed backtracking is taking another 56% over backjumping. These relationships, however, are not constant over all the problems. To give a better sense of what is going on, Table 2.2 shows this same information for three particular problems, namely the hardest problems for dependency-directed backtracking (out of the 50,000 sampled) at the following values of γ: 3.3, 3.5, and 4.4.

	Nodes	Seconds	Nodes/second
Backtracking	1206	0.046	26482
Backjumping	107	0.006	18781
DDB	101	0.008	12020

Table 2.1: Average performance over the problem set.

γ	Algorithm	Nodes	Seconds	Nodes/second
3.3	Backtracking	291804	11.77	24792
	Backjumping	10010	0.44	22750
	DDB	1225	0.10	12250
3.5	Backtracking	135541	5.63	24075
	Backjumping	68371	3.00	22790
	DDB	2814	0.32	8794
4.4	Backtracking	8312	0.40	20780
	Backjumping	8256	0.42	19657
	DDB	4668	0.87	5366

Table 2.2: Performance on the hardest example for various values of γ.

We see that for the larger problems backjumping is only taking an overhead of around 5% to 10%. The figures in Table 2.1 were somewhat misleading in this respect because most of the problems are so easy. There is a fixed cost in initializing the data structures (and cleaning them up at the end), and backjumping did not have very many nodes over which to amortize this cost.

Dependency-directed backtracking, however, has an increasing overhead as the problems get harder. Every time it backtracks, it learns a new nogood, so as the search trees get larger and larger, dependency-directed backtracking will have to remember more and more nogoods; and every time it moves up or back in the search tree, it will have consult some number of these nogoods. For sufficiently hard problems, dependency-directed backtracking might be too expensive for practical use.

In Chapter 1, we mentioned some algorithms that strike a compromise between backjumping and full dependency-directed backtracking. One useful approach is k-order learning [28]. If the size of the conflict set at a dead end does not exceed k (where k is a constant), then this technique will save the new nogood; otherwise, it will not. For a small k, this will greatly reduce the number of nogoods that have to be learned, and furthermore, it is precisely these short nogoods that are most likely to be useful. We have found empirically that 1-order learning or 2-order learning often captures most of the benefits of full dependency-directed backtracking. Another possibility is to delete nogoods that are no longer "relevant," or likely to be useful. This is the idea behind *dynamic backtracking* [42, 45], although, as we will see in the next chapter, there are some possible hazards with dynamic backtracking if it is used carelessly.

2.5 Summary

This chapter has presented experimental results on the utility and applicability of intelligent backtracking algorithms such as backjumping and dependency-directed backtracking. We have shown that these techniques are especially important on those underconstrained problems that are the hardest for backtracking, but are of less value on the more highly constrained problems.

We have also illuminated some issues concerning the distribution of hard and easy constraint problems. In [51], Hogg and Williams write:

> An interesting open question is whether these hard, sparse graphs have qualitatively different structure than the more densely connected cases associated with the peak in the median.

We have shown that the hard underconstrained problems are in fact quite different from the hard crossover-point problems. The crossover-point problems seem to be fundamentally difficult since their search trees tend to contain a large number of distinct dead ends. The search trees for the hard underconstrained problems, however, tend to contain a small number of distinct dead ends repeated over and over again (which is why they are so easy for backjumping and dependency-directed backtracking).

The experiments in this chapter have all been graph coloring problems (furthermore, the problems have all had 100 vertices and 3 colors). We believe, however, that the lessons are more general and will apply to other constraint satisfaction problems as well.

Chapter 3

The Hazards of Fancy Backtracking

3.1 Introduction

In the last chapter, we saw some of the benefits of intelligent backtracking. We will now turn to a possible hazard. There has been some recent interest in backtracking procedures that can return to the source of a difficulty without erasing the intermediate work. We will show that for some problems it can be counterproductive to do this, and in fact that such "intelligence" can cause an exponential increase in the size of the ultimate search space. We will discuss the reason for this phenomenon, and we will present one way to deal with it.[1]

In particular, we will discuss dynamic backtracking; let us first review the motivation behind that algorithm, using an example from [42]. Suppose we are coloring a map of the United States (subject to the usual constraint that only some fixed set of colors may be used, and adjacent states cannot be the same color).

Let us assume that we first color the states along the Mississippi, thus dividing the rest of the problem into two independent parts. We now color some of the western states, then we color some eastern states, and then we return to the west. Assume further that upon our return to the west we immediately get stuck: we find a western

[1]This chapter is a revised version of [5].

state that we cannot color. What do we do?

Ordinary backtracking would backtrack to the most recent decision, but this would be a state east of the Mississippi and hence irrelevant; the search procedure would only address the real problem after trying every possible coloring for the previous eastern states.

Backjumping [39, 40] is somewhat more intelligent; it would immediately jump back to some state adjacent to the one that we cannot color. In the process of doing this, however, it would erase all the intervening work, i.e., it would uncolor the whole eastern section of the country. This is unfortunate; it means that each time we backjump in this fashion, we will have to start solving the eastern subproblem all over again.

Ginsberg has recently introduced *dynamic backtracking* [42] to address this difficulty. In dynamic backtracking, one moves to the source of the problem without erasing the intermediate work. Of course, simply retaining the *values* of the intervening variables is not enough; if these values turn out to be wrong, we will need to know where we were in the search space so that we can continue the search systematically. In order to do this, dynamic backtracking accumulates nogoods to keep track of portions of the space that have been ruled out.

Taken to an extreme, this would end up being very similar to dependency-directed backtracking [79]. Although dependency-directed backtracking does not save intermediate values, it saves enough dependency information for it to quickly recover its position in the search space. Unfortunately, dependency-directed backtracking saves far too much information. Since it learns a new nogood from every backtrack point, it generally requires an exponential amount of memory — and for each move in the search space, it may have to wade through a great many of these nogoods. Dynamic backtracking, on the other hand, only keeps nogoods that are "relevant" to the current position in the search space. It not only learns new nogoods; it also throws away those old nogoods that are no longer applicable.

Dynamic backtracking, then, would seem to be a happy medium between backjumping and full dependency-directed backtracking. Furthermore, Ginsberg has presented empirical evidence that dynamic backtracking outperforms backjumping on the problem of solving crossword puzzles [42].

Unfortunately, as we will soon see, dynamic backtracking has problems of its own.

The plan of this chapter is as follows. The next section reviews the details of dynamic backtracking. Section 3.3 describes an experiment comparing the performance of dynamic backtracking with that of backtracking and backjumping on a problem class that has become somewhat of a standard benchmark. We will see that dynamic backtracking is *worse* by a factor exponential in the size of the problem. Note that this will not be simply the usual complaint that intelligent search schemes often have a lot of overhead (see, for example, Table 2.2 and the associated discussion towards the end of the previous chapter). Rather, our complaint will be that the effective search space itself becomes larger; even if dynamic backtracking could be implemented without any additional overhead, it would still be far less efficient than the other algorithms.

Section 3.4 contains both our analysis of what is going wrong with dynamic backtracking and an experiment consistent with our view. In Section 3.5, we describe a modification to dynamic backtracking that appears to fix the problem. Concluding remarks for this chapter are in Section 3.6.

3.2 Dynamic Backtracking

We have already presented the dynamic backtracking algorithm in Figure 1.7, but it will be useful to slightly paraphrase that algorithm for the purpose of this chapter. The dynamic backtracking algorithm in Figure 1.7 makes use of a two-dimensional *culprits* array. The meaning of the *culprits* array was that for a variable v and value x, if *culprits*$[v, x]$ is some set of variables, then as long as the these variables have their current values, v cannot have the value x. This array entry can be viewed as a nogood as in dependency-directed backtracking (see the discussion below). In this chapter, we will deal with the nogoods directly, and ignore the lower-level data structures like the *culprits* array. This should help simplify some of the explanations, and it is in keeping with the more recent style of dynamic backtracking work [45].

Like regular backtracking, dynamic backtracking works with partial solutions; a partial solution to a CSP is an assignment of values to some subset of the variables, where the assignment satisfies all of the constraints that apply to this particular subset. The algorithm starts by initializing the partial solution to have an empty domain, and then it gradually extends this solution. As the algorithm proceeds, it will derive new constraints, or "nogoods," that rule out portions of the search space that contain no solutions. Eventually, the algorithm will either derive the empty nogood, proving that the problem is unsolvable, or it will succeed in constructing a total solution that satisfies all of the constraints. We will always write the nogoods in directed form; e.g.,

$$(v_1 = q_1) \wedge \cdots \wedge (v_{k-1} = q_{k-1}) \Rightarrow v_k \neq q_k$$

tells us that variables v_1 through v_k cannot simultaneously have the values q_1 through q_k respectively.

The main innovation of dynamic backtracking (compared to dependency-directed backtracking) is that it only retains nogoods whose left-hand sides are currently true. That is to say that if the above nogood were stored, then v_1 through v_{k-1} would have to have the indicated values (and since the current partial solution has to respect the nogoods as well as the original constraints, v_k would either have some value other than q_k or be unbound). If at some point, one of the left-hand variables were changed, then the nogood would have to be deleted since it would no longer be "relevant." Because of this relevance requirement, it is easy to compute the currently permissible values for any variable. Furthermore, if all of the values for some variable are eliminated by nogoods, then one can resolve these nogoods together to generate a new nogood. For example, assuming that $D_{v_9} = \{1, 2\}$, we could resolve

$$(v_1 = a) \wedge (v_3 = c) \Rightarrow v_9 \neq 1$$

with

$$(v_2 = b) \wedge (v_3 = c) \Rightarrow v_9 \neq 2$$

to obtain

$$(v_1 = a) \wedge (v_2 = b) \Rightarrow v_3 \neq c$$

In order for our partial solution to remain consistent with the nogoods, we would have to simultaneously unbind v_3. This corresponds to backjumping from v_9 to v_3, but without erasing any intermediate work. Note that we had to make a decision about which variable to put on the right-hand side of the new nogood. The rule of dynamic backtracking is that the right-hand variable must always be the one that was most recently assigned a value; this is absolutely crucial, as without this restriction, the algorithm would not be guaranteed to terminate.

The only thing left to mention is how nogoods get acquired in the first place. Before we try to bind a new variable, we will check the consistency of each possible value[2] for this variable with the values of all currently bound variables. If a constraint would be violated, we write the constraint as a directed nogood with the new variable on the right-hand side.

We have now reviewed all the major ideas of dynamic backtracking. Since we have already presented the algorithm once, we will give the algorithm below in a somewhat informal style.

Procedure DYNAMIC-BACKTRACKING

1. Initialize the partial assignment f to have the empty domain, and the set of nogoods Γ to be the empty set. At all times, f will satisfy the nogoods in Γ as well as the original constraints.

2. If f is a total assignment, then return f as the answer. Otherwise, choose an unassigned variable v and for each possible value of this variable that would cause a constraint violation, add the appropriate nogood to Γ.

3. If variable v has some value x that is not ruled out by any nogood, then set $f(v) = x$, and return to step 2.

4. Each value of v violates a nogood. Resolve these nogoods together to generate a new nogood that does not mention v. If it is the empty nogood, then return UNSAT as the answer. Otherwise, write it with its chronologically most recent variable (say, w) on the right-hand side, add this directed nogood to Γ, and

[2] A value is possible if it is not eliminated by a nogood.

call ERASE-VARIABLE(w). If each value of w now violates a nogood, then set $v = w$ and return[3] to step 4; otherwise, return to step 2.

Procedure ERASE-VARIABLE(w)

1. Remove w from the domain of f.

2. For each nogood $\gamma \in \Gamma$ whose left-hand side mentions w, call DELETE-NOGOOD(γ).

Procedure DELETE-NOGOOD(γ)

1. Remove γ from Γ.

Each variable-value pair can have at most one nogood at a given time, so it is easy to see that the algorithm only requires a polynomial amount of memory. In [42], it is proven that dynamic backtracking always terminates with a correct answer.

This is the theory of dynamic backtracking. How well does it do in practice?

3.3 Experiments

To compare dynamic backtracking with backtracking and backjumping, we will use randomly-generated propositional satisfiability problems, or to be more specific, random 3-SAT problems with n variables and m clauses.[4] Since a SAT problem is just a Boolean CSP, the above discussion applies directly. Each clause will be chosen independently using the uniform distribution over the $\binom{n}{3}2^3$ non-redundant 3-literal clauses. It turns out that the hardest random 3-SAT problems appear to arise at the "crossover point" where the ratio of clauses to variables is such that about half the problems are satisfiable [13, 63]; the best current estimate for the location of this crossover point is at $m = 4.24n + 6.21$ [23]. Several recent authors have used these crossover-point 3-SAT problems to measure the performance of their algorithms [23, 77].

[3]This differs from the earlier algorithm in that it automatically includes an obvious variable-ordering heuristic.

[4]Each clause in a 3-SAT problem is a disjunction of three literals. A literal is either a propositional variable or its negation.

In the dynamic backtracking algorithm, step 2 leaves open the choice of which variable to select next; backtracking and backjumping have similar indeterminacies. We used the following variable-selection heuristics:

1. If there is an unassigned variable with one of its two values currently eliminated by a nogood, then choose that variable.

2. Otherwise, if there is an unassigned variable that appears in a clause in which all the other literals have been assigned false, then choose that variable.

3. Otherwise, choose the unassigned variable that appears in the most binary clauses. A binary clause is a clause in which exactly two literals are unvalued, and all the rest are false.[5]

The first heuristic is just a typical backtracking convention, and in fact is intrinsically part of backtracking and backjumping. The second heuristic is unit propagation, a standard part of the Davis-Putnam procedure for propositional satisfiability [26, 27]. The last heuristic is also a fairly common SAT heuristic; see for example [23, 87]. These heuristics choose variables that are highly constrained and constraining in an attempt to make the ultimate search space as small as possible.

For our experiments, we varied the number of variables n from 10 to 60 in increments of 10. For each value of n we generated random crossover-point problems[6] until we had accumulated 100 satisfiable and 100 unsatisfiable instances. We then ran each of the three algorithms on the 200 instances in each problem set. The mean number of times that a variable is assigned a value is displayed in Table 3.1.

Dynamic backtracking appears to be worse than the other two algorithms by a factor exponential in the size of the problem; this is rather surprising. Because of the lack of structure in these randomly-generated problems, we might not expect dynamic backtracking to be significantly better than the other algorithms, but why would it be *worse*? This question is of more than academic interest. Some real-world search problems may turn out to be similar in some respects to the crossword puzzles on

[5]On the very first iteration in a 3-SAT problem, there will not yet be any binary clauses, so instead choose the variable that appears in the most clauses overall.

[6]The numbers of clauses that we used were 49, 91, 133, 176, 218, and 261 respectively.

	Average Number of Assignments		
Variables	Backtracking	Backjumping	Dynamic Backtracking
10	20	20	22
20	54	54	94
30	120	120	643
40	217	216	4,532
50	388	387	31,297
60	709	705	212,596

Table 3.1: A comparison using randomly-generated 3-SAT problems.

which dynamic backtracking does well, while being similar in other respects to these random 3-SAT problems — and as we can see from Table 3.1, even a small "random 3-SAT component" will be enough to make dynamic backtracking virtually useless.

3.4 Analysis

To understand what is going wrong with dynamic backtracking, consider the following abstract SAT example:

$$a \rightarrow x \tag{3.1}$$

$$\Rightarrow \neg a \tag{3.2}$$

$$\neg a \Rightarrow b \tag{3.3}$$

$$b \Rightarrow c \tag{3.4}$$

$$c \Rightarrow d \tag{3.5}$$

$$x \Rightarrow \neg d \tag{3.6}$$

Formula (3.1) represents the clause $\neg a \vee x$; we have written it in the directed form above to suggest how it will be used in our example. The remaining formulas correspond to groups of clauses; to indicate this, we have written them using the double arrow ($\Rightarrow$). Formula (3.2) represents some number of clauses that can be used to prove that a is contradictory. Formula (3.3) represents some set of clauses showing that if a is false, then b must be true; similar remarks apply to the remaining formulas. These formulas will also represent the nogoods that will eventually be learned.

Imagine dynamic backtracking exploring the search space in the order suggested above. First it sets a true, and then it concludes x using unit resolution (and adds a nogood corresponding to (3.1)). Then after some amount of further search, it finds that a has to be false. So it erases a, adds the nogood (3.2), and then deletes the nogood (3.1) since it is no longer "relevant." Note that it does *not* delete the proposition x — the whole point of dynamic backtracking is to preserve this intermediate work.

It will then set a false, and after some more search will learn nogoods (3.3)–(3.5), and set b, c and d true. It will then go on to discover that x and d cannot both be true, so it will have to add a new nogood (3.6) and erase d. The rule, remember, is that the most recently valued variable goes on the right-hand side of the nogood. Nogoods (3.5) and (3.6) are resolved together to produce the nogood

$$x \Rightarrow \neg c \tag{3.7}$$

where once again, since c is the most recent variable, it must be the one that is retracted and placed on the right-hand side of the nogood; and when c is retracted, nogood (3.5) must be deleted also. Continuing in this fashion, dynamic backtracking will derive the nogoods

$$x \Rightarrow \neg b \tag{3.8}$$

$$x \Rightarrow a \tag{3.9}$$

The values of b and a will be erased, and nogoods (3.4) and (3.3) will be deleted.

Finally, (3.2) and (3.9) will be resolved together producing

$$\Rightarrow \neg x \tag{3.10}$$

The value of x will be erased, nogoods (3.6)–(3.9) will be deleted, and the search procedure will then go on to rediscover (3.3)–(3.5) all over again.

By contrast, backtracking and backjumping would erase x before (or at the same time as) erasing a. They could then proceed to solve the rest of the problem without being encumbered by this leftover inference. It might help to think of this in terms of search trees even though dynamic backtracking is not really searching a tree. By

	Average Number of Assignments		
Variables	Backtracking	Backjumping	Dynamic Backtracking
10	77	61	51
20	2,243	750	478
30	53,007	7,210	3,741

Table 3.2: The same comparison as Table 3.1, but with all variable-selection heuristics disabled.

failing to retract x, dynamic backtracking is in a sense choosing to "branch" on x before branching on a through d. This virtually doubles the size of the ultimate search space.

This example has been a bit involved, and so far it has only demonstrated that it is *possible* for dynamic backtracking to be worse than the simpler methods; why would it be worse in the *average* case? The answer lies in the heuristics that are being used to guide the search.

At each stage, a good search algorithm will try to select the variable that will make the remaining search space as small as possible. The appropriate choice will depend heavily on the values of previous variables. Unit propagation, as in equation (3.1), is an obvious example: if a is true, then we should immediately set x true as well; but if a is false, then there is no longer any particular reason to branch on x. After a is unset, our variable-selection heuristic would most likely choose to branch on a variable other than x; branching on x anyway is tantamount to randomly corrupting this heuristic. Now, dynamic backtracking does not really "branch" on variables since it has the ability to jump around in the search space. As we have seen, however, the decision not to erase x amounts to the same thing. In short, the leftover work that dynamic backtracking tries so hard to preserve often does more harm than good because it perpetuates decisions whose heuristic justifications have expired.

This analysis suggests that if we were to eliminate the heuristics, then dynamic backtracking would no longer be defeated by the other search methods. Table 3.2 contains the results of such an experiment. It is important to note that all of the previously listed heuristics (including unit resolution!) were disabled for the purpose of this experiment; at each stage, we simply chose the first unbound variable (using some fixed ordering). For each value of n listed, we used the same 200 random problems that were generated earlier.

The results in Table 3.2 are as expected. All of the algorithms fare far worse than before, but at least dynamic backtracking is not worse than the others. In fact, it is a bit better than backjumping and substantially better than backtracking. So given that there is nothing intrinsically wrong with dynamic backtracking, the challenge is to modify it in order to reduce or eliminate its negative interaction with our search heuristics.

3.5 Solution

We have to balance two considerations. When backtracking, we would like to preserve as much nontrivial work as possible. On the other hand, we do not want to leave a lot of "junk" lying around whose main effect is to degrade the effectiveness of the heuristics. In general, it is not obvious how to strike the appropriate balance. For the propositional case, however, there is a simple modification that seems to help, namely, undoing unit propagation when backtracking.

We will need the following definition:

Definition 3.1 *Let v be a variable (in a Boolean CSP) that is currently assigned a value. A nogood whose conclusion eliminates the other value for v will be said to* justify *this assignment.*

If a value is justified by a nogood, and this nogood is deleted at some point, then the value should be erased as well. Selecting the given value was once a good heuristic decision, but now that its justification has been deleted, the value would probably just get in the way. Therefore, we will rewrite DELETE-NOGOOD as follows, and leave the rest of dynamic backtracking intact:

Variables	Average Number of Assignments		
	Backtracking	Backjumping	Dynamic Backtracking
10	20	20	20
20	54	54	53
30	120	120	118
40	217	216	209
50	388	387	375
60	709	705	672

Table 3.3: The same comparison as Table 3.1, but with dynamic backtracking modified to undo unit propagation when it backtracks.

Procedure DELETE-NOGOOD(γ)

1. Remove γ from Γ.

2. For each variable w justified by γ, call ERASE-VARIABLE(w).

Note that ERASE-VARIABLE calls DELETE-NOGOOD in turn; the two procedures are mutually recursive. This corresponds to the possibility of undoing a cascade of unit resolutions. Like Ginsberg's original algorithm, this modified version is sound and complete, uses only polynomial space, and can solve the union of several independent problems in time proportional to the sum of that required for the original problems.

We ran this modified procedure on the same experiments as before, and the results are in Table 3.3. Happily, dynamic backtracking no longer blows up the search space. It does not do much good either, but there may well be other examples for which this modified version of dynamic backtracking is the method of choice.

How will this apply to non-Boolean problems? First of all, for non-Boolean CSPs, the problem is not quite as dire. Suppose a variable has twenty possible values, all but two of which are eliminated by nogoods. Suppose further that on this basis, one of the remaining values is assigned to the variable. If one of the eighteen nogoods is later eliminated, then the variable will still have but three possibilities and will probably remain a good choice. It is only in the Boolean problems that an assignment can go all the way from being totally justified to totally unjustified with the deletion of a single nogood. Nonetheless, in experiments by Jónsson and Ginsberg it was found that dynamic backtracking often did worse than backjumping when coloring random graphs [52]. Perhaps some variant of our new method would help on these problems. One idea would be to delete a value if it loses a certain number (or percentage) of the nogoods that once supported it.

3.6 Summary

Although we have presented the research in this chapter in terms of Ginsberg's dynamic backtracking algorithm, the implications are much broader. Any systematic search algorithm that learns and forgets nogoods as it moves laterally through a search space will have to address—in some way or another—the problem that we have discussed. The fundamental problem is that when a decision is retracted, there may be subsequent decisions whose justifications are thereby undercut. While there is no *logical* reason to retract these decisions as well, there may be good heuristic reasons for doing so.

On the other hand, the solution that we have presented is not the only one possible, and it is probably not the best one either. Instead of erasing a variable that has lost its heuristic justification, it would be better to keep the value around, but in the event of a contradiction remember to backtrack on this variable instead of a later one. With standard dynamic backtracking, however, we do not have this option; we always have to backtrack on the most recent variable in the new nogood. Ginsberg and McAllester have recently developed *partial-order dynamic backtracking* [45], a variant of dynamic backtracking that relaxes this restriction to some extent, and it might be interesting to explore some of the possibilities that this more general method makes possible.

Perhaps the main purpose of this chapter has been to sound a note of caution with regard to the new search algorithms. Ginsberg claims in one of his theorems that dynamic backtracking "can be expected to expand fewer nodes than backjumping provided that the goal nodes are distributed randomly in the search space" [42]. In the presence of search heuristics, this is false. For example, the goal nodes in *unsatisfiable* 3-SAT problems are certainly randomly distributed (since there are not any goal nodes), and yet standard dynamic backtracking can take orders of magnitude longer to search the space.

Therefore, while there are some obvious benefits to the new backtracking techniques, the reader should be aware that there are also some hazards.

Chapter 4

Backtracking and the Induced Width

In Chapters 2 and 3, we studied the performance of various algorithms experimentally. In this chapter and the next one, we will prove theoretical worst-case results. These results will be stated in terms of a problem parameter known as the *induced width*.

We begin in Section 4.1 with the definition of the induced width. To help motivate this definition, we will discuss Dechter and Pearl's adaptive consistency algorithm. This non-backtracking constraint satisfaction algorithm has the interesting property of being "exponential only in the induced width." This means that its worst-case time complexity can be written as the problem size raised to the power of some function of the induced width. An immediate consequence of this is that the algorithm is polynomial time for problems of bounded induced width. The central project of this chapter will be to examine which of the backtracking algorithms from Chapter 1 share this property; we will be especially interested in the polynomial-space algorithms.

It is well known that dependency-directed backtracking (a non-polynomial-space algorithm) is exponential only in the induced width; we will review this result in Section 4.2. In the subsequent sections of this chapter, we will show that none of the polynomial-space algorithms from Chapter 1 are exponential only in the induced width. In particular, we will prove this for backtracking and backjumping in Section 4.3, for dynamic backtracking in Section 4.4, and for k-order learning and polynomial-space dependency-directed backtracking in Section 4.5. Strictly speaking, we could omit the discussions of backtracking, backjumping, and k-order learning since these algorithms are special cases of polynomial-space dependency-directed backtracking. The special-case arguments, however, are simpler, and might help with some readers' intuitions.

4.1 The Induced Width of a CSP

The induced width of a constraint satisfaction problem (CSP) is defined in terms of the CSP's *constraint graph.* The constraint graph captures the pattern of interaction between the various variables, while abstracting away the actual constraint predicates. It is formed by placing an edge between any two variables that share a constraint. More formally:

Definition 4.1 *The* constraint graph *of a CSP (V, D, C) is the (undirected) graph $G = (V, E)$, where the vertices V are the variables of the CSP, and the edges are*

$$E = \{\{x, y\} : \exists (W, P) \in C, x \in W, y \in W, x \neq y\}.$$

We will also need the notion of an *ordering* for a graph or a CSP.

Definition 4.2 *Consider a graph (V, E) or a CSP (V, D, C), where $|V| = n$. An* ordering *for the graph or CSP is a bijection h from V to $\{1, 2, \ldots, n\}$.*

The ordering h corresponds to the vertices or variables being listed in the order $(h^{-1}(1), h^{-1}(2), \ldots, h^{-1}(n))$. If two variables are connected in the graph, it is sometimes helpful to think of the one that comes first in the ordering as being the "parent" of the second, and the second as being the "child" of the first.

Before defining the induced width, let us define the simpler concept of *width.*

Definition 4.3 *Let $G = (V, E)$ be a graph, and let h be some ordering for this graph. The* width of a vertex v under this ordering *is the number of vertices that are connected to v and precede it in the ordering (i.e., the number of parents of v). The* width of the graph under the ordering *is the maximum width of any of the vertices under the ordering. Finally, the* width of the graph *is its minimum width under any ordering.*

For example, a tree will have width 1, and a fully connected graph with n vertices will have a width of $n - 1$. We also have the following straightforward definitions:

Definition 4.4 *The* width of a CSP under a given ordering *is the width of its constraint graph under that ordering, and the* width of a CSP *is the width of its constraint graph.*

The idea of width was used by Freuder [34] to show that a CSP with width 1 can be solved in linear time using the SOLVE-WIDTH1-PROBLEM algorithm in Figure 4.1. First, the procedure identifies a width-1 ordering h for the problem P; this can be done by performing a preorder tree walk of the constraint graph rooted at an arbitrary vertex. Then, SOLVE-WIDTH1-PROBLEM makes two passes over the variables, first a backward pass (that is, backward with respect to the order h), and then a forward pass. The backward pass is a call to DIRECTIONAL-ARC-CONSISTENCY, which does some constraint propagation on the problem P in order to produce a simpler (but equivalent) problem P'. The forward pass is a call to the backtracking algorithm[1] for P', but because of the constraint propagation phase, the backtracking procedure will never have to backtrack.

The constraint propagation procedure, DIRECTIONAL-ARC-CONSISTENCY, works as follows. It takes as its arguments a CSP P (as usual, the CSP is a triple (V, D, C) of variables, domains, and constraints), and an ordering h for the CSP. It processes the variables in the reverse order $(v_n, \ldots, v_1)$, where v_i abbreviates $h^{-1}(i)$. For each variable v, it inspects the parents of v (on line 2), or in other words, those variables connected to v that precede it in the ordering; for a width-1 problem, the number of parents will always be 0 or 1. It then calls UPDATE-DOMAIN to remove from the domain of the parent any value that is not consistent with at least one value in the domain of the child. This is surely a sound transformation as the value being removed cannot be part of any solution. After this preprocessing has been completed, the new

[1]For the purpose of SOLVE-WIDTH1-PROBLEM, we have extended BACKTRACK-TOP, the top-level call to the backtracking algorithm of Figure 1.2, to take an ordering h as its second argument. The backtracking search will be conducted using the static variable ordering h; that is, on line 4 of BACKTRACKING, the variable v that is selected will always be the first unbound variable in the ordering.

```
SOLVE-WIDTH1-PROBLEM($P$)   {where $P$ is a CSP of the form $(V, D, C)$}
1   $h$ := a width-1 ordering for $P$
2   $P'$ := DIRECTIONAL-ARC-CONSISTENCY($P, h$)
3   return BACKTRACK-TOP($P', h$)

DIRECTIONAL-ARC-CONSISTENCY($P, h$)
1   for $i := n$ down to 1
2       for each variable $v_j$ that is a parent of $v_i$
3           $D_j :=$ UPDATE-DOMAIN($v_j, D_j, v_i, D_i, C$)
4   $P' := (V, D, C)$
5   return $P'$

UPDATE-DOMAIN($u, D_u, v, D_v, C$)
1   for each value $x \in D_u$
2       if there is no value $y \in D_v$ such that setting $u$ to $x$ and $v$ to $y$ would satisfy
         the binary constraint(s) in $C$ between $u$ and $v$
3           $D_u := D_u - \{x\}$
4   return $D_u$
```

Figure 4.1: Freuder's algorithm for a width-1 CSP

problem is *directionally arc consistent* [30]. This means that for any value in the domain of a variable, and for any child of this variable, there will be at least one value in the domain of the child that is consistent with the value of the parent. This is a weaker version of full arc consistency, as defined by Mackworth [58].[2] When backtracking is called on a ordered width-1 problem that is directionally arc consistent, it never has to backtrack. More precisely, if a given value x satisfies the constraints on line 8 of BACKTRACKING, then the recursive call on line 9 must succeed. It is not hard to see that with the appropriate data structures, the entire SOLVE-WIDTH1-PROBLEM procedure will run in time $O(na^2)$, where n is the number of variables and a is the maximum domain size. This is linear in the size of the problem.

Since width-1 CSPs are so easy, one might conjecture that the problems would get progressively more difficult as the width increases. Unfortunately, it turns out that even width-2 problems can be intractable. To see this, note that any binary CSP can be compiled to a width-2 problem by creating a new copy of a variable for each time that the variable is used in a constraint, and then adding a new constraint asserting that the copy must have the same value as the original. If our ordering places the originals before the copies, then the ordering will have width 2. (An alternative transformation proceeds by reifying the constraints.) Therefore, we will not have much use for the width except insofar as it is used to define the more sophisticated notion of induced width.

Actually, SOLVE-WIDTH1-PROBLEM *will* continue to work for any CSP (provided that we replace the width-1 ordering h on line 1 with an arbitrary ordering), but the algorithm will no longer run in linear time. It is instructive to examine what goes wrong. The constraint propagation of DIRECTIONAL-ARC-CONSISTENCY will still make the problem directionally arc consistent, but this property is not as useful for an arbitrary problem as it is for those of width 1. Consider a width-2 constraint graph in which v_3 has two parents, v_1 and v_2. If we have assigned values to v_1 and v_2, then even if the problem is directionally arc consistent, there need not be any consistent assignment for v_3. There must be some value for v_3 that is consistent with the value

[2]Freuder's original algorithm [34] actually used full arc consistency instead of directional arc consistency; we presented the simplified version here because it is both more efficient and easier to understand.

of v_1, and there must be some value for v_3 that is consistent with the value of v_2, but these might be different values; therefore, BACKTRACKING might have to backtrack. If we want to solve the problem without backtracking, then we will have to do more preprocessing than that of DIRECTIONAL-ARC-CONSISTENCY. In particular, we will have to *induce* new constraints between variables like v_1 and v_2. The induced width of a graph takes into account these extra induced edges.

Dechter and Pearl [30] first define the notion of the *induced graph* with respect to a given ordering. This induced graph adds edges to the original graph so that whenever x and y are parents of some vertex, there will also be an edge between x and y. More formally:

Definition 4.5 *Given a graph $G = (V, E)$ and some ordering h, the* induced graph *$G' = (V, E')$ is defined as that graph with the minimal set of edges E' such that $E' \supseteq E$, and such that if $\{x, v\} \in E'$, $\{y, v\} \in E'$, $h(x) < h(v), h(y) < h(v)$, and $x \neq y$, then $\{x, y\} \in E'$.*

The induced graph can be constructed in one pass by processing the vertices in reverse order; that is, first connect the parents of the last vertex, then the parents of the penultimate vertex, etc. It is now straightforward to define the remaining notions that we will need:

Definition 4.6 *The* induced width of a graph under an ordering *is the width under that ordering of the induced graph under that ordering. The* induced width of a graph *is its minimal induced width under any ordering. The* induced width of a CSP under an ordering *is the induced width of its constraint graph under that ordering, and the* induced width of a CSP *is the induced width of its constraint graph.*

There are some other terms that are synonymous with induced width. It is also referred to as the *tree width* or sometimes as the *dimension* of the graph. There is also the notion of a *partial k-tree*; a partial k-tree is simply a graph whose induced width does not exceed k [1]. We use the term "induced width" because of its popularity in the constraint satisfaction literature.

Dechter and Pearl [30] show that given a CSP and an ordering, the CSP can be solved in time exponential only in its induced width under that ordering. Thus, the induced width can be thought of as a measure of problem difficulty. Their algorithm is displayed in Figure 4.2. As with Freuder's algorithm, the CSP is first simplified by a constraint propagation procedure that processes the variables in reverse order; the reduced problem can then be solved without backtracking.

The ADAPTIVE-CONSISTENCY procedure first constructs G, the constraint graph of the problem (line 1). (Note that G is initially the ordinary constraint graph, not the induced graph; the induced graph ends up being constructed over the course of the algorithm.) The variables are processed in reverse order, and a new constraint is added for the parents of each variable v_i (lines 3–5). If there are (say) k parents of v_i, then the new constraint will be a k-ary constraint. The predicate for this constraint, which is computed by the INDUCED-PREDICATE procedure, only allows those k-tuples of values for which there is some value for v_i that is consistent with these values, where consistency here means with respect to both the original constraints and the constraints that we have added. Finally, edges are added between all the parents (line 6–7), and these extra edges will be taken into account when computing the parents of subsequent variables (on line 3). When ADAPTIVE-CONSISTENCY has finished, the CSP can be solved without backtracking.

In short, adaptive consistency is similar to directional arc consistency, but it will ensure a backtrack-free search for any problem, not just for a problem of width 1. For a more general discussion of dynamic programming algorithms for NP-complete graph problems, see the survey paper by Arnborg [1]. Let us now look at the time and space complexity of adaptive consistency. If w^* is the induced width of the CSP with respect to the given ordering, then the parent sets will each have size of at most w^*. Thus, if n is the number of variables, and a is the maximum domain size, the space to store the new predicates is $O(na^{w^*})$. The time to construct these predicates is $O(na^{w^*+1})$ multiplied by whatever time is required to do the constraint check one line 3 of INDUCED-PREDICATE; this latter time will depend on the exact implementation, but it is obviously polynomial in the size of the problem. Since n and a are polynomial in the problem size, and since all the other costs of adaptive consistency are also polynomial in the problem size, it follows that adaptive consistency has both space

```
ADAPTIVE-CONSISTENCY-TOP($P, h$)   {where $P$ is a CSP of the form $(V, D, C)$,
                                        and $h$ is an ordering for that CSP}
1  $P' :=$ ADAPTIVE-CONSISTENCY($P, h$)
2  return BACKTRACK-TOP($P', h$)

ADAPTIVE-CONSISTENCY($P, h$)
1  $G :=$ the constraint graph of the CSP $P$
2  for $i := n$ down to 1
3      $W :=$ the parents of $v_i$ in the graph $G$ under ordering $h$
4      $Q :=$ INDUCED-PREDICATE($W, v_i, P$)
5      $C := C \cup (W, Q)$
6      for all vertices $x, y \in W$, where $x \neq y$
7          Add the edge $\{x, y\}$ to $G$
8  $P' := (V, D, C)$
9  return $P'$

INDUCED-PREDICATE($W, v, P$)
1  $Q :=$ a new $|W|$-ary predicate over the domains of the variables in $W$
2  for each $|W|$-tuple $\omega$ over these domains
3      if there is some value $x \in D_v$ such that setting $v$ to $x$ and $W$ to $\omega$
        would satisfy the constraints
4          $Q(\omega) :=$ TRUE
5      else
6          $Q(\omega) :=$ FALSE
7  return $Q$
```

Figure 4.2: Adaptive consistency

and time complexity of

$$O\left(x^c a^{w^*}\right) \tag{4.1}$$

where x is the problem size, and c is some constant.

The above analysis assumes that we are given an ordering along with the problem. What if the problem happens to have a small-induced-width ordering, but we do not know what it is? Finding the smallest induced width (or an ordering with this induced width) is actually NP-hard [2], but this complexity is also growing exponentially only in the induced width, so the overall cost of adaptive consistency remains the same.

Now, let us be more precise about what it means for an algorithm to be exponential only in the induced width. We have discussed a particular example of such an algorithm, adaptive consistency, with the complexity given in formula (4.1). More generally, we will proceed as follows.

Definition 4.7 *When g is a real-valued function of two arguments, a CSP and an ordering for that CSP, then we will say that g is* exponential only in the induced width *if there is some function f such that g is*

$$O\left(x^{f(w^*)}\right), \tag{4.2}$$

where x is the size of the encoded problem and w^* is the induced width of the problem under the given ordering. If the worst-case running time of a CSP algorithm, when given a CSP and an ordering, is exponential only in the induced width, then we will say that the algorithm is exponential only in the induced width.

Note that f in this definition can be an arbitrary function. In practice, f is usually a linear function, but we do not require this. Perhaps, one might say "exponential only in some function of the induced width" in order to more explicit, but we will stick with the shorter "exponential only in the induced width."

For future reference, let us record the following lemma:

Lemma 4.8 *A CSP algorithm is exponential only in the induced width if and only if it runs in polynomial time on any problem class in which the induced width is bounded.*

Proof. The forward "only if" direction is an immediate consequence of Definition 4.7: for any induced width bound k, let c_k be

$$\max_{w^* \leq k} f(w^*);$$

then from formula (4.2), we know that the running time on a problem class of induced width not exceeding k will be

$$O\left(x^{c_k}\right).$$

This is actually the only part of the lemma that we will need, but in the interest of completeness, we will now establish the other direction as well.

Suppose a CSP algorithm runs in polynomial time on any problem class in which the induced width is bounded. Then, for any k, there are x_k, b_k and c_k, such that for all problems of size $x \geq x_k$, and induced width $w^* \leq k$, the running time will not exceed

$$b_k x^{c_k}.$$

Let us assume that the problem size x is at least 2. Then, this time bound can be written as

$$x^{d_k}$$

(just let $d_k = c_k + \log_x b_k$). To handle the problems of size $x < x_k$ as well, let $f(k)$ be the maximum of d_k and

$$\max_{x < x_k} \log_x t(x),$$

where $t(x)$ is the maximum running time for a problem of size x (and induced width not exceeding k). These maxima exist since there are only a finite number of problems bounded by any fixed size.

It should be clear that f satisfies the conditions of Definition 4.7, provided that we disallow problems of size $x = 1$. To handle this trivial case, we can simply adjust the constant that is permitted by the $O()$ notation of formula (4.2). This completes the proof. □

We are interested in the relation between the space and time requirements of the various backtracking algorithms from Chapter 1. We are particularly interested in the possibility of a polynomial-space algorithm that is exponential only in the induced width. In order to fully specify the various algorithms from Figures (1.2)–(1.7), we should say how they choose the unassigned variable at each stage, and in what order they enumerate the values for that variable. Following Definition 4.7, we will assume that each of the algorithms has been extended to take a variable ordering as an additional argument; and that whenever an unassigned variable needs to be selected, the algorithm will choose the first unassigned variable under that ordering. We will assume that the values are selected in numerical order. These assumptions will allow us to make a fair comparison between adaptive consistency and the various backtracking algorithms. In Chapter 5, we will consider more sophisticated variable and value orderings.

4.2 Dependency-Directed Backtracking

In this section, we give a proof for the following known result:

Proposition 4.9 *Dependency-directed backtracking is exponential only in the induced width.*

The reason in short is that the variables in any nogood must be parents in the induced graph of some other variable, and thus the total number of nogoods is exponential only in the induced width; and by the nature of dependency-directed backtracking, it never has to learn the same nogood twice. To follow the details of the following proof, the reader may wish to refer back to the DDB (dependency-directed backtracking) procedure on page 13.

Proof. Over the course of the search, dependency-directed backtracking will accumulate nogoods in the set Γ. We claim that the variables mentioned in any nogood in Γ, and the variables in any conflict set returned by the DDB procedure, both have the following property: these variables are a subset of the parents of some other variable in the induced graph. As usual, the induced graph here is the induced graph of the CSP under the given ordering. We will prove this claim by induction on the number of times that DDB has returned.

The base case is trivial as Γ is initially empty and the procedure has not yet returned. If DDB returns a solution to the CSP (line 2 or 14), then the induction step is trivial as *conflict-set* is set to $\emptyset$, and Γ is passed through unchanged. If the **for** loop (on lines 7–18) terminates early (on line 16) because the current variable was not relevant to the backjump, then the induction step is also trivial, as both *conflict-set* and Γ are passed through unchanged.

The only interesting case is when the **for** loop exits normally after trying every value for the variable. Let v be the variable over whose values we are iterating. Each time through the loop, *new-conflicts* will be assigned some set of variables (on line 10 or 12), and v will always be one of these variables (since the other cases were covered in the last paragraph). We can see that the other variables in *new-conflicts* must be parents of v in the induced graph by considering the various possibilities: if a constraint is violated, then the other variables in *new-conflicts* will be parents of v in the original graph; and if a nogood is violated or if a recursive call to DDB returns the *new-conflicts* (on line 10), then from the inductive hypothesis, we know that all of these variables (including v) must be parents in the induced graph of some other variable, and thus by the definition of the induced graph, the remaining variables will be induced-graph parents of v. The conflict set that is returned by DDB will be the union of all the sets of *new-conflicts* minus the variable v. This verifies the inductive step for *conflict-set*. The new nogood γ (that is formed on line 19) contains the same variables that are in the conflict set; this completes the inductive argument.

So how many possible nogoods are there? Let n be the number of variables in the CSP, a be the maximum domain size, and w^* be the induced width under the given ordering. Then we can choose a variable in n different ways, and each of its induced-graph parents can either be assigned one of at most a values, or omitted from the nogood. This gives an upper bound of $n(a+1)^{w^*}$ possible nogoods.

When a nogood is learned, the algorithm backjumps up the stack until it reaches a variable in the nogood. If the variables in this nogood are ever instantiated again with the same values, then the nogood will cause a constraint violation (on line 9), preventing DDB from being invoked with this assignment. Therefore, the same nogood will never be learned more than once.

The maximum number of steps between the learning of two nogoods can be bounded in the following way. It takes $O(n)$ steps to backjump up the stack. DDB can then assign at most n variables, doing at most a consistency checks for each one. The time required by the consistency check on line 9 of DDB will depend on the details of the implementation. One might imagine representing the nogoods for the parents of a given variable by a multidimensional array (as in ADAPTIVE-CONSISTENCY), but let us consider instead the straightforward implementation, in which Γ is just a list of nogoods. It might then take time $|\Gamma| = O(n(a+1)^{w^*})$ (multiplied by a polynomial factor) just to search down this list. This gives a total time for dependency-directed backtracking of:

$$O(x^c(a+1)^{2w^*}),$$

where x is the size of the problem, and c is some constant. This satisfies the criterion of formula (4.2); therefore, dependency-directed backtracking is exponential only in the induced width. □

Despite the superficial differences between dependency-directed backtracking and adaptive consistency, the two algorithms are fundamentally quite similar. Both procedures save the solutions to a certain type of subproblem so that these results will not have to be repeatedly recomputed. Adaptive consistency follows the traditional dynamic programming approach of doing the computation from the bottom up, while dependency-directed backtracking follows the "memoization" variant of dynamic programming in which the computation is done top-down, but with the intermediate results being cached [21]. Memoization has the advantage that it does not have to solve the subproblems that never get used. Consider for example, a problem with a huge induced width, but very weak constraints. Adaptive consistency will spend a lot of time computing huge induced relation tables that are of no use, while dependency-directed backtracking (like regular backtracking and backjumping) will probably find a solution very quickly. On the other hand, memoization sometimes requires more overhead than the bottom-up dynamic programming approach.

Actually, dependency-directed backtracking differs in an interesting way from being a straightforward memoized version of adaptive consistency. If v is some variable and $P(v)$ is the set of parents (in the induced graph) of this variable, then if at some point there were no consistent assignment for v, the memoized version of adaptive consistency would in effect make the conflict set equal to $P(v)$ and learn a new nogood based on the current values of all the variables in this set. In dependency-directed backtracking, on the other hand, the conflict set might be a strict subset of $P(v)$, depending on which variables in $P(v)$ were actually involved in the constraint violations; this might permit more backjumping, and it would also lead to the learning of a more general nogood. One might imagine a variant of dependency-directed backtracking that performed a fully static dependency analysis based solely on the structure of the constraint graph; Dechter calls this approach *graph-based dependency-directed backtracking* [28]. Standard dependency-directed backtracking requires more overhead than the graph-based version, but for some problems, it can have more pruning power [35].

Because of the extra flexibility and pruning power of full dependency-directed backtracking, one might speculate that it would be exponential only in the induced width of the problem, regardless of which ordering is used. In other words, suppose the problem has some ordering with a small induced width, but dependency-directed backtracking searches using a completely different ordering. Will this still always yield an efficient search? It will not. This is demonstrated by the example below.

Example 4.10 *Consider the problem class $\{P_n : n \geq 1\}$, where P_n is defined as follows. The variables are $\{X_1, X_2, \ldots, X_{n-1}, X_n, Y_1, Y_2\}$ (so there are $n+2$ variables altogether). For i from 1 to n, the domain of X_i is $\{1, 2\}$; and the domains of Y_1 and Y_2 are $\{1, 2, \ldots, 2n\}$. For i from 1 to n, we have the constraints:*[3]

$$\begin{aligned} X_i = 1 &\Rightarrow Y_1 \neq 2i - 1, \\ X_i = 2 &\Rightarrow Y_1 \neq 2i, \\ X_i = 1 &\Rightarrow Y_2 \neq 2i, \end{aligned}$$

[3]In Definition 1.1, we defined a constraint to be an ordered pair (W, Q), where W is a tuple of variables and Q is a predicate on these variables. Using that notation, the first constraint of this example for $i = 1$ would be written as $((X_1, Y_1), \{(u, v) : u = 1 \Rightarrow v \neq 1\})$. The more concise notation that we use instead is merely a shorthand; it does not mean that we necessarily represent the constraints internally in that particular syntax.

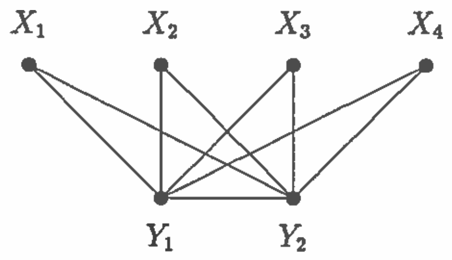

Figure 4.3: The constraint graph of problem P_4 from Example 4.10.

$$X_i = 2 \quad \Rightarrow \quad Y_2 \neq 2i - 1.$$

Finally, we have the constraint:

$$Y_1 = Y_2.$$

Figure 4.3 shows the constraint graph for this example when n is 4.

First of all, note that all of the CSPs in the problem class of Example 4.10 are unsatisfiable. Suppose, for example, we set $Y_1 = Y_2 = 1$. Then there would be no possible value for X_1 because if X_1 is 1, then Y_1 cannot be 1, but if X_1 is 2, then Y_2 cannot be 1. If we instead made $Y_1 = Y_2 = 2$, then X_1 would still have no consistent assignment (for the opposite reason in each case). If we made $Y_1 = Y_2 = 3$, then X_2 would now have no consistent assignment, etc.

As we have just seen, it easy to do this problem if we order the variables Y_1 and Y_2 before any of the X_i variables. Under any such ordering, the problem will have an induced width of 2, and we know from Proposition 4.9 that dependency-directed backtracking under this ordering will run in polynomial time (as would backjumping for this particular example). Suppose, however, that we ordered Y_1 and Y_2 last, *after* all of the X_i variables. The induced width under this ordering would be $n + 1$, so Proposition 4.9 would no longer guarantee polynomial-time performance. Dependency-directed backtracking with this perverse ordering will in fact require time $\Omega(2^n)$ since for each of the 2^n assignments to the X_i variables, there will be no solution, and the conflict set will in each case include all n of the X_i variables; thus the nogood that is learned will never be reused. Since the encoded size of this problem is $O(n^2)$, which is polynomial in n, it follows that dependency-directed backtracking does not run in polynomial time on this example. Therefore, the most we can say is that dependency-directed backtracking is exponential only in the induced width under the ordering that it actually uses.

4.3 Backtracking and Backjumping

In the last section, we gave a positive result (i.e., an upper bound) for dependency-directed backtracking. In the remainder of this chapter, we will present negative results (i.e., lower bounds) for the less memory-intensive algorithms. We will show that none of these other algorithms are exponential only in the induced width. In order to show that an algorithm is not exponential only in the induced width, we will exhibit a sequence of problems of fixed induced width for which the algorithm's running time is nonetheless more than polynomial.

For backtracking and backjumping, it is easy to do this. Here is the counterexample:

Example 4.11 *Consider the problem class $\{P_n : n > 1\}$, where the problem P_n will be defined as follows. The ordered set of variables is $(X_1, X_2, \ldots, X_n)$.*[4] *All of the variables will have the same domain: $\{1, 2, 3\}$. For each i from 1 to $n-1$, there will be a binary constraint:*

$$X_i \neq 3 \Rightarrow X_{i+1} \neq 3,$$

and finally there will be a single unary constraint:

$$X_n = 3.$$

Each of these problems has an induced width of 1 under the given ordering since the only parent of the variable X_i, for $i > 1$, is the variable X_{i-1}. Furthermore, each problem has exactly one solution, namely setting all the variables to 3. The backtracking algorithm, however, will take a long time to discover this solution.

We will make use of the *backtracking search tree* of each problem, which will be defined as follows. Each node in the tree is a partial assignment that assigns values to

[4]Or more verbosely, the set of variables is $\{X_1, X_2, \ldots, X_n\}$, and the ordering is the function h such that $h(X_i) = i$.

an initial subsequence of the variables, i.e., to the variables $X_1, X_2, \ldots, X_k$ for some k where $0 \leq k \leq n$. The root node of the search tree is the null assignment. For any such assignment f where $k < n$, f_i will refer to the new assignment obtained from f by assigning the value i to the next unbound variable (X_{k+1}). If a call to BACKTRACKING with the partial assignment f makes recursive calls to BACKTRACKING, then the children of f are the partial assignments that are passed to these recursive calls; if this initial call does not make any recursive calls, then f is a leaf node.

Since (by assumption) the values are tried in numerical order, the variable X_1 will be assigned the wrong values 1 and 2 before being assigned the correct value 3. Once X_1 has been assigned a wrong value, the backtracking algorithm will have to explore the entire subtree below this value, and this subtree will contain 2^{n-2} leaf nodes. More formally:

Proposition 4.12 *Backtracking is not exponential only in the induced width.*

Proof. For a given partial assignment f in the backtracking search tree of some problem P_n of Example 4.11, let $V(f)$ be the value that BACKTRACKING returns when called with f, and let $L(f)$ be the number of leaves in the subtree under f. Consider a partial assignment f that assigns either a 1 or a 2 to each of the first k variables, where $1 \leq k \leq n-1$. We will show by induction that $V(f)$=UNSAT and $L(f) = 2^{n-1-k}$; this will be by backwards induction on k. The base case is when $k = n-1$; f assigns either the value 1 or 2 to X_{n-1}, and thus there will be no consistent assignment for X_n. Therefore, $V(f)$ =UNSAT and since f is itself a leaf node, $L(f) = 1$. When $1 \leq k < n-1$, f assigns either the value 1 or 2 to X_k, and both 1 and 2 (but not 3) are consistent assignments for X_{k+1}. By the inductive hypothesis then, we know that $V(f_1) = V(f_2) =$ UNSAT and that $L(f_1) = L(f_2) = 2^{n-1-(k+1)}$. Therefore, $V(f) =$ UNSAT, and $L(f) = L(f_1) + L(f_2) = 2^{n-1-k}$, which is what we want. The number of leaf nodes under a given assignment is a lower bound on the total number of calls to BACKTRACKING on this assignment.

The initial call to BACKTRACKING with the empty assignment will first call BACKTRACKING with the assignment of 1 to X_1. By the above paragraph, this will take time $\Omega(2^{n-2})$. The size of the problem is $O(n)$. Therefore, the running time of BACKTRACKING on this problem class is not polynomial. By Lemma 4.8, it then follows that backtracking is not exponential only in the induced width. □

Backjumping will not help at all on Example 4.11. We show below that the conflict set that backjumping uses to determine which past variables contributed to a dead end will always include the last variable that was instantiated; hence, backjumping will search exactly the same tree as backtracking.

Proposition 4.13 *Backjumping is not exponential only in the induced width.*

Proof. Consider the *backjumping search tree* for some problem P_n of Example 4.11; this tree is defined analogously to the backtracking tree, with the children of any node corresponding to the recursive calls of the BACKJUMPING procedure. For a given partial assignment f, let $V(f)$ be the first value returned by BACKJUMPING (either a solution or UNSAT), let $C(f)$ be the conflict set returned, and let $N(f)$ be the number of leaf nodes in the subtree under f. As before, let f be some partial assignment that assigns 1's and 2's to the first k variables, where $1 \leq k \leq n-1$. We claim that $V(f)$=UNSAT, $C(f) = \{X_k\}$, and $L(f) = 2^{n-1-k}$. For the base case of $k = n-1$, the current value of X_{n-1} (either 1 or 2) will rule out the value of 3 for X_n, and the other values for X_n will be ruled out by the unary constraint on X_n. Thus, for such an f, $V(f)$=UNSAT, $C(f) = \{X_{n-1}\}$, and $L(f) = 1$. Now consider an assignment f of length k, where $1 \leq k < n-1$. By the inductive hypothesis, the recursive calls to both f_1 and f_2 will fail, in both cases setting *new-conflicts* to $\{X_{k+1}\}$; since X_{k+1} will be in both of these returned conflicts sets, there will be no opportunity to exit the loop early. The constraint check on f_3 will fail, setting *new-conflicts* to $\{X_k, X_{k+1}\}$. The returned *conflict-set* is obtained by taking the union of these *new-conflicts* sets, and filtering out the branching variable X_{k+1}. So we have: $V(f)$=UNSAT, $C(f) = \{X_k\}$, and $L(f) = L(f_1) + L(f_2) = 2^{n-1-k}$. This gives the same lower bound $\Omega(2^{n-2})$ as for backtracking. Therefore, backjumping is not exponential only in the induced width. □

One odd thing about Example 4.11 is that the reverse of the given variable ordering, that is, the ordering $(V_n, V_{n-1}, \ldots, V_2, V_1)$, would have worked just fine. This will not always be the case, however, as we will see in Chapter 5.

4.4 Dynamic Backtracking

Dynamic backtracking is potentially more powerful than backjumping because it can often preserve dependency information for variables over which it has already backtracked. Consider Example 4.11 from the last section. Suppose dynamic backtracking is working with a partial assignment f that assigns values to the first k variables, with the variable X_k taking the value 1. Like backjumping, dynamic backtracking will, after some amount of search, determine that f cannot be extended to value the variable X_{k+1}, and because of the linear structure of the constraint graph, the only variable in the conflict set would be X_k. Backjumping would just discard this information when it backs up, but dynamic backtracking would remember that

$$\mathit{culprits}[X_k, 1] = \emptyset.$$

The *culprits* entry is the empty set because $X_k = 1$ has been determined to be a dead end regardless of the values of the other variables. Dynamic backtracking will not make the mistake of setting X_k to 1 again. It will go on to learn similar *culprits* entries for all the variables, and it will quickly solve the problem.

In general, dynamic backtracking can solve problems with width-1 orderings in polynomial time. In this respect, it is better than backtracking and backjumping, and the same as Freuder's algorithm and k-order learning when $k = 1$. Unlike the latter two algorithms, however, dynamic backtracking can cache dependency information involving more than one variable, e.g., for some problem it might learn

$$\mathit{culprits}[v, x] = \{w_1, w_2\}, \tag{4.3}$$

meaning that as long as variables w_1 and w_2 have their current values, v cannot take the value x. Unlike full dependency-directed backtracking, however, dynamic backtracking cannot store such dependency information indefinitely; as soon as w_1 or w_2 changes its value, this *culprits* entry must be deleted. If w_1 or w_2 is later changed back to its original value, then (4.3) will have to be derived all over again. Because of this limitation, dynamic backtracking is not exponential only in the induced width.

To demonstrate this fact, we will use the following modification of Example 4.11.

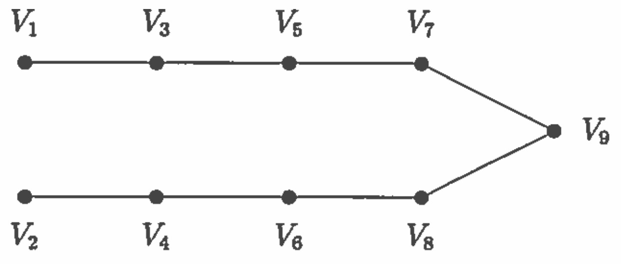

Figure 4.4: The constraint graph of problem P_4 from Example 4.14.

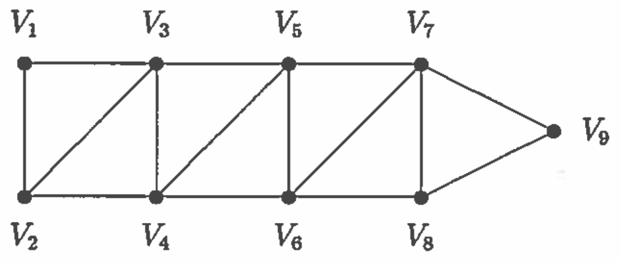

Figure 4.5: The induced graph of problem P_4 from Example 4.14.

Example 4.14 *Consider the problem class $\{P_n : n \geq 1\}$, where the problem P_n will be defined as follows. The ordered set of variables is $(V_1, V_2, \ldots, V_{2n-1}, V_{2n}, V_{2n+1})$ (so the number of variables is $2n+1$). For all i from 1 to $2n$, the domain of V_i is $\{1,2,3\}$; and the domain of V_{2n+1} is $\{1,2\}$. For each i from 1 to $2n-2$, there will be the constraint:*

$$V_i \neq 3 \Rightarrow V_{i+2} \neq 3.$$

Finally, we have the constraints:

$$\begin{aligned} V_{2n-1} \neq 3 &\Rightarrow V_{2n+1} \neq 1, \\ V_{2n} \neq 3 &\Rightarrow V_{2n+1} \neq 2. \end{aligned}$$

Figure 4.4 shows the constraint graph for this example when n=4, and Figure 4.5 shows the corresponding induced graph.

First, we will show that this problem class has a bounded induced width.

Lemma 4.15 *Every problem P_n of Example 4.14 has an induced width of 2 under the given ordering.*

Proof. It is straightforward to prove by induction that for $i > 2$, the only parents of V_i in the induced graph are V_{i-1} and V_{i-2}. □

Note that there are no solutions to this CSP with both V_1 and V_2 simultaneously set to values other than 3. The nature of the ordering ensures that dynamic backtracking will begin by setting both variables to 1. We just have to prove that it will then take exponential time before retracting these assignments. It turns out that on this example, dynamic backtracking does just as poorly as simple backtracking because (1) it never gets a chance to backjump, and (2) each new piece of information in the *culprits* array gets deleted before it can be reused. To follow the details of the following proof, the reader may wish to refer back to the DYNAMIC-BACKTRACKING algorithm on page 18.

Proposition 4.16 *Dynamic backtracking is not exponential only in the induced width.*

Proof. Consider some problem P_n from Example 4.14. Define a tree of partial assignments as follows. The root node is the partial assignment that assigns 1 to both V_1 and V_2. We will say that the root node has a "depth" of 2 (since it assigns two variables). For each node f in the tree of depth d, where $2 \leq d < 2n$, the children of f will be f_1 and f_2; as in the last section, f_i is the partial assignment f extended by assigning i to the first variable that is not bound by f. We will say that these children have depths $d + 1$. It should be apparent that the leaves of this tree are assignments to all the variables except V_{2n+1}, and that there are 2^{2n-2} such leaves. Note that we cannot assume in advance that dynamic backtracking will actually search this tree; after all, dynamic backtracking is not even a tree search algorithm since it can dynamically rearrange the order of past variables. We will show, however, that dynamic backtracking must in fact perform a *full walk* of the tree (a full walk of a tree lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree [21]); this is more than enough to show that dynamic backtracking will have to visit all 2^{2n-2} leaves.

We will monitor the state of the DYNAMIC-BACKTRACKING procedure at the top of its main loop (i.e., when it is about to execute line 5); each time through, we will be interested in the current values of the assignment f and the array *culprits*.

In order to state the appropriate inductive hypothesis, let us define the notion of a trivial *culprits* entry; we will call the following entries *trivial*:

$$\begin{aligned} \mathit{culprits}[v,x] &= \textsc{ValueAllowed} && \text{for any variable } v \text{ and value } x \\ \mathit{culprits}[V_k,3] &= \{V_{k-2}\} && \text{for } 3 \le k \le 2n \\ \mathit{culprits}[V_{2n+1},1] &= \{V_{2n-1}\} \\ \mathit{culprits}[V_{2n+1},2] &= \{V_{2n}\} \end{aligned}$$

In other words, an entry in the *culprits* array is trivial if allows the value, or if it could be obtained directly from one of the original constraints; in either case, it does not cache any useful work.

Our claim, which we will prove by induction, is as follows. Assume that the current partial assignment f assigns the values $(u_1, u_2, \ldots, u_k)$ to the variables $(V_1, V_2, \ldots, V_k)$ respectively, where $k > 2$ and all the u_i are in $\{1,2\}$. Further assume that the *culprits* array has no nontrivial entries for variables that are currently unbound, i.e., for variables $V_{k'}$ where $k' > k$. Then, we claim that by the time that dynamic backtracking unsets the value of V_k, it will have performed a full walk of the subtree with root f (in the graph described above), and that the only new non-trivial entry in the *culprits* array at the end of this process will be

$$\mathit{culprits}[V_k, u_k] = \{V_{k-1}\}.$$

The base case is for a leaf node, that is, where $k = 2n$. There will be no assignment for the variable V_{2n+1} consistent with f; it cannot be 1 because of the current value of V_{2n-1}, and it cannot be 2 because of the current value of V_{2n} (whether because of the constraints or because of trivial *culprits* entries, it does not really matter). The *conflict-set* variable will be set (on line 13) to $\{V_{2n-1}, V_{2n}\}$, and the backtrack variable w will be set (on line 17) to V_{2n}; therefore *culprits*$[V_{2n}, u_{2n}]$ will be set (on line 21) to V_{2n-1}, which is what we want. Finally, since this is a leaf node, the "full walk" claim is fulfilled trivially; this concludes the base case.

Now, for the inductive step where $k < 2n$. The variables up to V_k will have been assigned values, so the next open variable is V_{k+1}. This cannot be 3, either because of a pre-existing entry

$$\mathit{culprits}[V_{k+1}, 3] = \{V_{k-1}\}, \tag{4.4}$$

or because of the problem constraint (which will then be recorded as a trivial *culprits* entry on lines 8–9). If there are no nontrivial *culprits* entries for $V_{k'}$, where $k' > k$, then in particular there will be no entry preventing V_{k+1} from being set to 1. Since there are also no constraints that prevent this, V_{k+1} will be set to 1, and then by the inductive hypothesis, the algorithm will then walk the entire subtree under this augmented assignment before finally unsetting V_{k+1} and learning that

$$\mathit{culprits}[V_{k+1}, 1] = \{V_k\}. \tag{4.5}$$

Then, V_{k+1} will be set 2, and that subtree will be walked, producing the entry

$$\mathit{culprits}[V_{k+1}, 2] = \{V_k\}. \tag{4.6}$$

Now there will be no more allowed values for V_{k+1}. The *conflict-set* will be computed (on line 13) from entries (4.4)–(4.6), yielding $\{V_{k-1}, V_k\}$. Thus, the backtrack variable w will be V_k, causing the deletion of (4.5) and (4.6) (on lines 18–20). The value of $\mathit{culprits}[V_k, u_k]$ will be set to V_{k-1}, and finally V_k will be unbound. This completes the inductive argument.

Dynamic backtracking will start with the null assignment, and it will set V_1 and V_2 to 1 on the first two iterations through the loop. By the argument above, we know that dynamic backtracking will then perform a full walk of the subtree rooted at this assignment, taking time $\Omega(2^{2n-2})$. The size of the problem is $O(n)$, so dynamic backtracking requires more than polynomial time. Since the induced width of the problem class is fixed, it follows that dynamic backtracking cannot be exponential only in the induced width. □

As an aside, Example 4.14 also demonstrates that Freuder's algorithm in Figure 4.1 is not exponential only in the induced width. The directional arc-consistency procedure will accomplish absolutely nothing, as the problem as given is already directionally arc consistent. Backtracking will then have to backtrack over the entire space.

4.5 Polynomial-Space Caching

Before discussing arbitrary polynomial-space dependency-directed backtracking algorithms, let us cover the special case of k-order learning. This idea, remember, was that a new nogood would only be saved if it had k or fewer variables in it. If the induced width does not exceed k, then k-order learning will work just fine:

Proposition 4.17 *Consider a class of problems with induced widths less than or equal to some k (as usual, each problem here is a CSP with an ordering, and the induced width is under that ordering). Then k-order learning will run in polynomial time on these problems.*

Proof. From the proof of Proposition 4.9, we know that dependency-directed backtracking will never learn nogoods that are larger than the induced width under the given ordering. Therefore, if the induced widths do not exceed k, then k-order learning will be equivalent to full dependency-directed backtracking, which is polynomial time on such a problem class. □

Once the induced widths exceed k, however, k-order learning will no longer be tractable. The counterexample that will establish this is a generalization of Example 4.14.

Example 4.18 *Define the problem $P_{m,n}$ as follows, where $m > 0$ and $n > 0$. The ordered set of variables is $(V_1, V_2, \ldots, V_{mn}, V_{mn+1})$. For all i from 1 to mn, the domain of V_i is $\{1, 2, 3\}$; and the domain of V_{mn+1} is $\{1, 2, \ldots, m\}$. For each i from 1 to $m(n-1)$, there will be the constraint:*

$$V_i \neq 3 \Rightarrow V_{i+m} \neq 3.$$

Finally, for j from 1 to m, we have the constraints:

$$V_{m(n-1)+j} \neq 3 \Rightarrow V_{mn+1} \neq j.$$

Figure 4.6 shows the constraint graph for this example when $m = 3$ and $n = 4$.

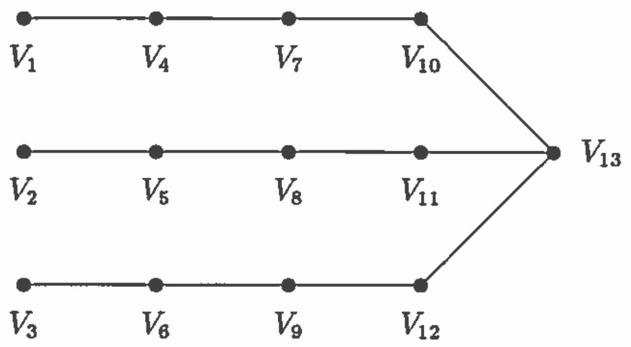

Figure 4.6: The constraint graph of problem $P_{3,4}$ from Examples 4.18 and 4.21.

Note that the problem $P_{m,n}$ has an induced width of m, and further note that it does not have any solutions without at least one of the first m variables being set to 3.

Proposition 4.19 *For any fixed k, the k-order learning algorithm is not exponential only in the induced width.*

Proof. Consider the class of problems $\{P_{m,n} : m = k+1, n \geq 1\}$, where k is the parameter of the k-ORDER LEARNING algorithm. Let f be some partial assignment that assigns the values 1 or 2 to each of the first j variables, where $m \leq j \leq mn$; this assignment cannot be extended to a solution.

When k-ORDER LEARNING is called with this assignment, it will have to examine 2^{mn-j} leaf nodes, and it will then return a conflict set consisting of the last m variables in the partial assignment, that is, the variables $X_{j-m+1}, \ldots, X_j$; since m is greater than k, no new nogood will be learned. This can be proven formally using the same type of inductive argument that was used to establish Proposition 4.13.

Since k-order learning begins by assigning 1 to the first m variables, it must examine at least $2^{m(n-1)}$ leaf nodes altogether. The size of the problem is only $O(mn)$, so k-order learning is not polynomial time on this problem class. The induced width of the problem class is constant; therefore, k-order learning is not exponential only in the induced width. □

We have looked at some special cases of polynomial-space dependency-directed backtracking; now let us consider the general case. As defined on page 15, such a procedure may use an arbitrary method to determine whether or not to cache the new nogood, and it may also choose to delete any set of old nogoods; the only restriction is that the nogood set Γ can never be larger than some polynomial in the size of the problem.[5] Unfortunately, this polynomial-space restriction itself is sufficient to preclude the algorithm from being exponential only in the induced width.

Theorem 4.20 *There is no polynomial-space dependency-directed backtracking algorithm that is exponential only in the induced width.*

In order to prove Theorem 4.20, we will have to modify Example 4.18 slightly. We will keep the same constraint graph, as illustrated by Figure 4.6, but we will expand the domains (for all but the last variable) from $\{1,2,3\}$ to $\{1,2,\ldots,n,n+1\}$, and modify the constraint predicates accordingly.

Example 4.21 *Define the problem $P_{m,n}$ as follows, where $m > 0$ and $n > 0$. The ordered set of variables is $(V_1, V_2, \ldots, V_{mn}, V_{mn+1})$. For all i from 1 to mn, the domain of V_i is $\{1,2,\ldots,n,n+1\}$; and the domain of V_{mn+1} is $\{1,2,\ldots,m\}$. For each i from 1 to $m(n-1)$, there will be the constraint:*

$$V_i \neq n+1 \Rightarrow V_{i+m} \neq n+1.$$

Finally, for j from 1 to m, we have the constraints:

$$V_{m(n-1)+j} \neq n+1 \Rightarrow V_{mn+1} \neq j.$$

We can now prove the theorem:

Proof. A polynomial-space dependency-directed backtracking algorithm must have some constant b such that the maximum cardinality of the nogood set Γ is $O(x^b)$, where x is the size of the problem encoding. We will consider the class of problems

[5] One might object that perhaps some implementation of dependency-directed backtracking could use less memory than the maximum $|\Gamma|$ size by using a clever data structure encoding some collection of m nogoods in less than $O(m)$ space. Our answer to this objection is that if such a representation could actually compress an exponential number of nogoods down to polynomial size, then we would no longer consider it to be a dependency-directed backtracking algorithm.

$\{P_{m,n} : m = 3b+2, n \geq 1\}$ from Example 4.21. These all have induced widths of $3b+2$. The size of the problem encoding is $O(n^3)$ as there are $O(n)$ constraints and each constraint takes $O(n^2)$. Thus the maximum size of Γ is $O(n^{3b}) = O(n^{m-2})$; we will show that this is not large enough for the algorithm to run in polynomial time on this problem class.

Consider a partial assignment f that assigns values to the first j variables, where $m \leq j \leq mn$, where this assignment does not assign the value $n+1$ to any of these variables. We can define some tree of such partial assignments, in which the root node values the first m variables, and each non-leaf node has n children. This tree has $n^{m(n-1)}$ leaf nodes. As before, the *level* of a node is the number of variables to which it assigns values, so the root nodes is at level m. We will call this the "backtracking tree," since it is clear that standard backtracking will have to search all the nodes in this tree.

First, we claim that if the nogood set Γ is initially empty when the dependency-directed procedure is called on the root node, then when the procedure returns from level j, the conflict set and thus the new nogood γ (whether or not it is cached) will contain precisely the last m variables in the current assignment, that is, the variables V_i, where $j-m < i \leq j$. It is straightforward to prove this by mathematical induction.

Now, consider the actual search tree of the polynomial-space algorithm when it is called on the root node with Γ initially empty. We will call this the *pruned tree.* The pruned tree will be a subset of the backtracking tree. We will say that a node in the backtracking tree is *pruned* if it does not appear in the pruned tree. If a node is pruned, but its parent is not pruned, then we will say that the node is *directly pruned.* Let L_i be the set of level i nodes in the backtracking tree, and let M_i be the level i nodes that are directly pruned. We will show that

$$\frac{|M_i|}{|L_i|} = O\left(\frac{1}{n^2}\right). \tag{4.7}$$

Let f be a node at level $i-m$, and let f' be some descendent of f at level i. Suppose f' is about to be directly pruned. From the discussion of the previous paragraph, it follows that the only way that this can happen is if at the time that f' is being considered, the nogood set Γ contains the nogood that rules out the last m values of the assignment f'. But when could this nogood have possibly been acquired? It could only have been acquired at some previous time at which these m variables had the exact same values that they have now. This must have been before f was processed.

Therefore the nogood in question must have been in Γ when f was visited. But Γ can only contain $O(n^{m-2})$ nogoods at any time, and there are n^m level-i descendents of f. Therefore, only $O(1/n^2)$ of these nodes can be directly pruned. This argument applies to every such group of nodes at level i, establishing (4.7).

Finally, consider the set of leaf nodes, L_{mn}. Let Q be the set of pruned leaf nodes. Each such node must have exactly one ancestor (possibly itself) that is directly pruned. Let Q_i be the subset of Q whose directly pruned ancestor is on level i; then $Q = Q_m \cup Q_{m+1} \cup \ldots \cup Q_{mn}$. Given the structure of the tree, we can conclude from (4.7) that for all i,

$$\frac{|Q_i|}{|L_{mn}|} = O\left(\frac{1}{n^2}\right).$$

Therefore,

$$\frac{|Q|}{|L_{mn}|} = O\left(\frac{1}{n}\right).$$

Since there are exponentially many leaf nodes, and only $O(n^{-1})$ of them are being pruned, the algorithm in question does not run in polynomial time on this problem class. The induced width of the problem class is fixed; therefore, we conclude that there is no polynomial-space dependency-directed backtracking procedure that is exponential only in the induced width. □

4.6 Summary

In this chapter, we have analyzed a number of backtracking algorithms to determine which of them have a worst-case time complexity that is exponential only in the induced width. We have shown that full dependency-directed backtracking is exponential only in the induced width, but that none of the polynomial-space algorithms from Chapter 1 have this property. In Chapter 5, we extend these results to cover more sophisticated algorithms.

Chapter 5

Optimal Search Order and the Induced Width

5.1 Introduction

Our most powerful result so far is Theorem 4.20, which proves that there cannot be a polynomial-space dependency-directed backtracking algorithm that is exponential only in the induced width. This result, however, is not as general as we would like. We have assumed that the search algorithm is forced to order the variables using the static ordering that it is given. In fact, the pathological examples from Chapter 4 can be fixed simply by searching the variables in the opposite of the given order! We have also assumed that the values must be enumerated in a fixed order. A more general framework would allow the search algorithm to make these ordering decisions for itself. In this chapter, we will extend some of our previous results to cover this possibility.

We will assume in this chapter that the search algorithms make optimal choices with respect to the variable and value orderings. We make this assumption in order to give the various approaches every benefit of the doubt. If even the version using an optimal ordering cannot solve a problem efficiently, then certainly this result will apply with even greater force to any actual implementation. Of course, for a *satisfiable* CSP, a backtracking algorithm that made optimal choices would have no problem;

it would simply move to a solution immediately by setting all the variables to the appropriate values. For this reason, all of the counterexamples in this chapter will be unsatisfiable problems.

Earlier, when we defined what it meant to be exponential only in the induced width, we were interested in algorithms that took, as arguments, a CSP and a variable ordering. Since we are now assuming that the algorithm will automatically find the best ordering, there is no longer any reason to supply it with an ordering as one of its arguments. Therefore, we can slightly simplify Definition 4.7 for the case of an algorithm that is only given a CSP:

Definition 5.1 *A CSP algorithm that takes one argument, the CSP, is* exponential only in the induced width *if there is some function f such that the worst-case running time of the algorithm is*

$$O\left(x^{f(w^*)}\right)$$

where x is the size of the CSP, and w^* is the induced width of the CSP.

Recall that the induced width of a CSP is its minimal induced width under any ordering. Therefore, Definition 4.7 reduces to Definition 5.1 when a CSP algorithm that is given an ordering simply ignores this ordering.

In the next section, we show that even with an optimal search order, neither backtracking nor backjumping is exponential only in the induced width. Section 5.3 proves the analogous claim for k-order learning. We then note in Section 5.4 that the various constraint algorithms that we have been discussing can be viewed as special cases of resolution theorem proving [71]. This idea is used in Section 5.5 to prove that, even with an optimal search order, neither dynamic backtracking nor any of its recent generalizations [45, 61] is exponential only in the induced width. Finally, in Section 5.6 we present some more general conjectures.

5.2 Optimal Backtracking and Backjumping

The following counterexample will work for both backtracking and backjumping:

Example 5.2 *Consider the problem class $\{P_n : n \geq 1\}$, where the problem P_n will be defined as follows. The variables are $\{X_1, X_2, \ldots, X_n\}$. All of the variables have the same domain: $\{1, 2, \ldots, 2n\}$. For i from 1 to $n-1$, we have the constraint:*

$$X_i \equiv X_{i+1} \pmod{2}.$$

We also have the constraints:

$$X_1 \equiv 0 \pmod{2},$$
$$X_n \equiv 1 \pmod{2}.$$

Each of these examples is an unsatisfiable problem having an induced width of 1. The reason these are hard for backtracking is that the inconsistency is global; in order to prove that there is no solution, information has to be propagated all the way from X_1 to X_n, but backtracking cannot do this without repeatedly solving the same subproblems.

First, let us consider the case where backtracking uses a static variable ordering.

Proposition 5.3 *Backtracking, with an optimal static variable ordering and an optimal value ordering, is not exponential only in the induced width.*

Proof. Consider the problem P_n of Example 5.2, and suppose backtracking uses the variable ordering $(X_{i_1}, X_{i_2}, \ldots, X_{i_n})$. The search tree of partial assignments can be defined as follows. The null assignment is the root node, which is considered to be at level 0; and for any node f at level k (where $0 \leq k < n$), the children of f are those assignments that extend f by assigning a value to the variable $X_{i_{k+1}}$ that is consistent with the values of the previous variables.

Consider the nodes at level $n-1$. These assign values to all of the variables except X_{i_n}. How many such nodes are there? We can consistently assign any even value to any X_j for which $j < i_n$, and we can assign any odd value to any X_j for which $j > i_n$. Thus, there are at least n^{n-1} nodes in level $n-1$ of the search tree. Note that the size of the problem is $O(n^3)$, which is polynomial in n. Hence, the backtracking search tree is exponential in the problem size, and since the problem is unsatisfiable, backtracking will have to search the entire tree regardless of the order it uses to enumerate the values. Therefore, even with an optimal static variable ordering and an optimal value ordering, backtracking is not exponential only in the induced width. □

We now consider the more general case of a dynamic variable ordering. Static orderings require that every path down the tree orders the variables in the same way. In other words, all the nodes at a given level must branch on the same variable. A dynamic ordering, on the other hand, allows this decision to be made individually for each node. More formally, while a static ordering for a set of n variables is a bijection between $\{1, 2, \ldots, n\}$ and the set of variables; a dynamic ordering is a mapping from the set of partial assignments to the set of variables not present in the partial assignment.

The use of a dynamic variable ordering for Example 5.2 can significantly reduce the size of the search space. We can divide the problem in half by first branching on the variable $X_{\lfloor n/2 \rfloor}$. For each even value of this variable, we have created an unsatisfiable subproblem involving only the variables from $X_{\lfloor n/2 \rfloor}$ to X_n; we can then divide this problem in half by branching on $X_{\lfloor 3n/4 \rfloor}$. Similarly, for each odd value of $X_{\lfloor n/2 \rfloor}$, we have created an unsatisfiable subproblem involving only the variables from X_1 to $X_{\lfloor n/2 \rfloor}$, so we can divide *this* problem in half by next branching on $X_{\lfloor n/4 \rfloor}$. By this divide-and-conquer approach, the search space can be reduced from approximately n^n down to approximately $(2n)^{\log_2 n}$. This is still not polynomial, however, and we cannot do any better:

Proposition 5.4 *Backtracking, with optimal variable and value orderings, is not exponential only in the induced width.*

Proof. Define a search tree of partial assignments for the problem P_n of Example 5.2 as follows. The root node is the null assignment. For any node f, let $V(f)$ be the variable that the backtracking algorithm branches on when it is called with f. Then, the children of f are all the assignments that can be obtained from f by assigning some value to the variable $V(f)$ without violating any of the constraints. Obviously, a node will have from 0 to $2n$ children.

Since the problem is unsatisfiable, backtracking will have to examine the entire tree; let us compute a lower bound on the size of this tree. If f assigns values to variables X_i and X_j (where $i < j$), and it assigns an even value to one and an odd value to the other, then we will say that $[i, j]$ is a *bracket* in f. The intuition here is that the inconsistency has been "trapped" between X_i and X_j; in order to prove that f is impossible, one need only branch on the variables of the form X_k, where $i < k < j$. Similarly, if f assigns an odd value to some X_i, then we will say that $[0, i]$ is a bracket, and if f assigns an even value to some X_i, we will say that $[i, n+1]$ is a bracket. Finally, $[0, n+1]$ is a bracket for any assignment. The *difficulty* of a bracket $[a, b]$ is $b - a$, and the difficulty of the node f is the smallest difficulty of any of its brackets.

We claim that if d is the difficulty of f, and $d \geq 2$, then the smallest backtracking tree rooted at f has at least

$$n^{\lceil \log_2 d \rceil - 1} \tag{5.1}$$

leaf nodes; the proof is by induction. First, note that if f violates a constraint, then it must assign either an odd value to X_1, or an even value to X_n, or values of opposite parity to two successive variables. In each case, the difficulty of f must be 1; hence if f has a difficulty of at least 2, then it cannot yet violate any constraints.

For the base case of $d = 2$, formula (5.1) is just $n^0 = 1$, and the tree obviously has this many leaf nodes. Now, consider a node f with difficulty $d > 2$. Regardless of which new variable is selected by $V(f)$, f must have at least n children with difficulties of at least $d' = \lceil d/2 \rceil$. Since d' is at least 2, these children must be consistent with the constraints. We can then apply the inductive hypothesis to conclude that the tree rooted at f must have at least

$$n \left(n^{\lceil \log_2 \lceil d/2 \rceil \rceil - 1} \right) \geq n^{\lceil \log_2 d \rceil - 1}$$

leaf nodes.

Since the root node has a difficulty of $n+1$, any backtracking tree for the problem must have more than $n^{\log_2 n - 1}$ leaf nodes. This is not polynomial in n, and since the problem size is $O(n^3)$, it is not polynomial in the problem size. This completes the proof. □

Now, let us move on to backjumping. It turns out that when you use an optimal dynamic variable ordering, backjumping confers no advantage over backtracking. Backjumping helps to reduce the damage from "mistakes," but this does not matter if you do not make any mistakes:

Lemma 5.5 *For any CSP, the number of nodes in the smallest backtracking tree is the same as the number of nodes in the smallest backjumping tree.*

Proof. With any variable and value ordering, backjumping will obviously do at least as well as backtracking would do with that ordering. Therefore, the smallest backjumping tree will be at least as small as the smallest backtracking tree. We will now show that the converse is true as well.

Note also that if the CSP has a solution, then an optimal version of either procedure will simply select the right value for each variable, and thus this lemma would be true trivially. Assume then that the CSP is unsatisfiable.

Consider the smallest backjumping tree. As usual, the backjumping tree is defined as follows. The root node is the null assignment. For each node f, $V(f)$ is the variable that backjumping chooses to branch on when it is called with f, and the children of f are the partial assignments that are passed to the recursive calls to the backjumping procedure.

Now, imagine doing regular backtracking, selecting for each node f, the same variable $V(f)$ that backjumping selects. The only way that backjumping could outperform this backtracking routine would be if it actually gets to "backjump," i.e., if the BACKJUMPING procedure gets to exit the loop early (on lines 15–16 of the algorithm on page 10). This happens when there is some node f and some child of f, say g, such that when backjumping is called on g, it returns a conflict set that does not include $V(f)$. This would allow backjumping to prune the siblings of g that have not yet been visited. In that case, however, the original backjumping tree could have been improved. Since the variable $V(f)$ was not in the final conflict set, there was no reason to assign it a value, and we could simplify the tree by setting $V(f)$ to be the variable that was the $V(g)$ in the original tree. This contradicts our assumption that the original tree was optimal. Therefore, the best backjumping tree is no better than the best backtracking tree. □

Proposition 5.6 *Backjumping, with optimal variable and value orderings, is not exponential only in the induced width.*

Proof. This is an immediate consequence of Proposition 5.4 and Lemma 5.5. □

5.3 Optimal k-Order Learning

To show that k-order learning with an optimal search order is not exponential only in the induced width, we will use a variant of the well-known pigeonhole problem. The pigeonhole problem involves n pigeons and $n-1$ holes, the problem being to assign each pigeon to a hole such that no two pigeons are assigned the same hole. In constraint satisfaction terms, the pigeons are the variables and the holes are the values. Each of the n variables has $n-1$ possible values, and between each pair of variables, there is an inequality constraint. Obviously, the problem has no solution since there are more pigeons than holes.

It is not hard to see that even full dependency-directed backtracking will have to go through a search tree of size $(n-1)!$ before proving the problem unsolvable. This is not particularly surprising given some results from Haken [47] that we will discuss in Section 6.5. The standard pigeonhole problem, however, is not useful for our current purpose. Since every variable shares a constraint with every other variable, the induced width is $n-1$, and we need an example with a fixed induced width. Therefore, we will modify the pigeonhole problem as follows.

We will think of the pigeons as being arranged along a line, with a hole between each two consecutive pigeons. In other words, pigeon i (for $1 \leq i \leq n$) will be thought of as being at position i along some axis, and hole j (for $1 \leq j \leq n-1$) will be thought of as being at position $j + 1/2$; see Figure 5.1, where the disks are the pigeons, and the circles are the holes. Furthermore, each pigeon will be constrained by a leash of length m (where m is some fixed constant) that prevents it from occupying any

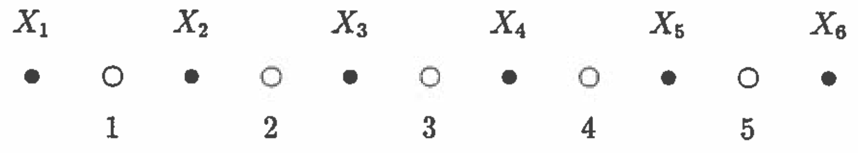

Figure 5.1: The localized pigeonhole problem

hole other than the m nearest holes on either side of it. For example, if m were 2 in Figure 5.1, then pigeon X_1 could only occupy holes 1 or 2, pigeon X_2 could only occupy holes 1, 2, or 3, pigeon X_3 could only occupy holes 1, 2, 3, or 4, pigeon X_4 could only occupy holes 2, 3, 4, or 5, pigeon X_5 could only occupy holes 3, 4, or 5, and pigeon X_6 could only occupy holes 4 or 5. The constraint that two pigeons cannot occupy the same hole remains in effect, but we no longer need constraints between every pair of pigeons. For example, we do not need a constraint between X_1 and X_5 since their domains are disjoint anyway. In general, we only need a constraint between pigeon X_i and X_j if $|i-j| < 2m$. The idea is that we will choose an m that is large enough to defeat the k-order learning algorithm, but then fix this m as the number of pigeons grows.

We need one more complication, however, for technical reasons. Each pigeon, besides being assigned a hole, will also be assigned an "orientation." Each pigeon will have n possible orientations, where the n here is also the number of pigeons. In other words, if there are q holes that some pigeon might occupy, then that pigeon variable will have nq possible values, with the value encoding both the position and the orientation. The constraints will now require that regardless of the orientations, two pigeons cannot occupy the same hole (and regardless of the orientations, the pigeons are free to occupy any two distinct holes that would otherwise be allowed). The orientation, then, is a nuisance parameter that plays no interesting role in the problem besides increasing the domain size. If the domain size were fixed, then even backtracking would be able to solve the problems efficiently using a simple divide-and-conquer approach. The branching factor would be a constant, and the depth of the tree would be $O(\log n)$, giving a total cost that is polynomial. With the orientations, however, the branching factor will now be a multiple of n, giving a total cost that is

$n^{\Omega(\log n)}$, which is not polynomial. This same trick was used in Example 5.2 in the previous section.

More formally, we have:

Example 5.7 *Define the localized pigeonhole problem $P_{m,n}$ (for $m, n \geq 1$) as follows. The variables are $\{X_1, X_2, \ldots, X_n\}$. First, the set of legal holes H_i for variable X_i is:*

$$\{h : i - m \leq h < i + m \text{ and } 1 \leq h < n\}.$$

(The asymmetry between the lower and upper bounds is a consequence of how the pigeons and holes are numbered; see Figure 5.1.) The domain D_i of X_i is then defined as:

$$\{hn + j : h \in H_i \text{ and } 0 \leq j < n\}.$$

For notational convenience, we will define the following abbreviations:

$$\begin{aligned} \mathit{hole}(x) &= \lfloor x/n \rfloor, \\ \mathit{orientation}(x) &= x - n\lfloor x/n \rfloor. \end{aligned}$$

For all $i \neq j$ such $1 \leq i, j \leq n$ and $|i - j| < 2m$, we have the constraint:

$$\text{hole}(X_i) \neq \text{hole}(X_j)$$

Theorem 5.8 *For any fixed k, k-order learning with optimal variable and value orderings is not exponential only in the induced width.*

Proof. Consider the set of problems $\{P_{m,n} : m = k + 1, n \geq 1\}$ from Example 5.7. We have set the "leash length" m for our localized pigeonhole problem to be one more than the order of the learning algorithm. Consider an arbitrary backjumping search tree for some $P_{m,n}$ in this set. Using the argument that was used to establish Proposition 5.6, it is easy to show that this tree must have at least

$$n^{\log_2 n - 1}$$

nodes. We now claim that k-order learning does not do any better than backjumping here. In particular, it will never cache a useful nogood.

Consider some path down the tree of length q that terminates at a backjump point. The path assigns values to the variables $Q = \{X_{i_1}, X_{i_2}, \ldots X_{i_q}\}$. The conflict set will be some subset of Q; a new nogood will be cached only if the size of the conflict set does not exceed k. Suppose that the conflict set is all of Q. Then the nogood (if learned) will be of no use whatsoever; it includes every single variable along the path, so it prunes a portion of the search tree that will never be revisited anyway. Therefore, assume that the conflict set is a strict subset of Q. We will prove that every such conflict set will have a size of at least m, and thus will never be retained by the k-order learning algorithm since k is less than m.

Assume by way of contradiction that the conflict set is a strict subset of Q and has a size less than m, say m'. Since it is a strict subset of Q, we can select some variable X in Q that does not appear in the conflict set. Now, consider the nogood that would be formed by prohibiting the members of the conflict set from having their current values. By the definition of generalized dependency-directed backtracking, this nogood must be a logical consequence of the original problem constraints. Furthermore, it must be derivable without using any of the constraints involving the missing variable X; if a constraint on X were used, it would still be in the conflict set since variables are only removed from the conflict set when all of their values have been branched on, whereas X is above the current backjump point in the search tree. Thus, the nogood has the following two properties: it has a size $m' < m$, and it can be derived without using any constraint of X. We claim, however, that such a nogood cannot exist.

Imagine a modification to our localized pigeonhole problem in which the variable X is omitted, and the m' variables in the conflict set are preassigned their current values. The purported nogood states, in effect, that this problem has no solution. If this problem had a solution, then the nogood would not be a logical consequence of the original constraints of the variables other than X. We now show that the modified problem does in fact have a solution (and thus the purported nogood is not valid). We construct the model as follows: for each pigeon from X_1 up to X_n (except X and those in the conflict set), assign the pigeon to the first unoccupied hole (in numerical order). Since the modified problem has $n-1-m'$ pigeons and holes, there will always be some unoccupied hole; we need to show, however, that the smallest open hole will always be in range of the current pigeon. There are two things that might go wrong: the first open hole might be out of range to the right, or it might be out of range to the left.

The first possibility is easy to disprove. Suppose we are currently trying to place pigeon i (henceforth, we will refer to pigeon i instead of using the more verbose X_i), and the first open hole is out of range to the right. Then, this means that pigeons 1 to $i-1$ together with those in the conflict set (that is, at most $i+m'-1$ pigeons altogether) have collectively occupied holes 1 through $i+m-1$; but this is impossible since $m' < m$.

The other possibility is that the first open hole is out of range to the left. Suppose we are trying to place pigeon i, and that j_1 of the pigeons with indices less than i were among the preassigned pigeons in the conflict set, and that j_2 of the pigeons with indices less than i were omitted from the problem (i.e., j_2 is 1 if the index of X is less than i, and 0 otherwise). We will prove by induction that all holes h, such that $1 \leq h < i - j_1 - j_2$, are already occupied. For the base case, $i = 1$ and $j_1 = j_2 = 0$, so the claim is true trivially. Assume then that the inductive hypothesis is true for i. If i is a pigeon that we have to place, then the leftmost position that it could possibly go to is $i - j_1 - j_2$; therefore, if that position is not currently occupied, it certainly will be after i is placed in a hole. Thus, when we get to pigeon $i+1$, all holes h such that $1 \leq h < (i+1) - j_1 - j_2$ will be occupied, verifying the inductive hypothesis for this case. The other case is that pigeon i is either in the conflict set or is the omitted variable X. In that case, the new $j_1 + j_2$ will be 1 higher than the current one, again verifying the inductive hypothesis. This completes the proof of the inductive lemma. We know, however, that $j_1 \leq m'$ and $j_2 \leq 1$. Therefore, all holes h such that $1 \leq h < i - m' - 1$ will be occupied when we try to place pigeon i. Since $m' + 1 \leq m$, it follows that all holes that are out of range to the left will already be occupied. Together with the previous paragraph, this proves that the leftmost open hole will be in range. The purported nogood is therefore invalid, and hence k-order learning will do no better than backjumping.

Thus, the size of the k-order learning tree is more than polynomial. Since the induced width of the problem set is fixed (at $2m - 1$), it follows that optimally-ordered k-order learning is not exponential only in the induced width. □

5.4 Resolution-Based Methods

In this chapter, for reasons already explained, we are focusing on unsatisfiable examples. The task of a search algorithm, on such an example, is essentially to construct a proof of its unsatisfiability. Before moving on to the analyses in the remainder of this chapter, let us review some material on unsatisfiability proofs.

All of the search algorithms that we have discussed can be viewed as special cases of resolution theorem proving [71]. Resolution is usually presented as an inference rule in predicate calculus or propositional logic, but it can also be used for non-Boolean problems.

In propositional logic, a *literal* is either a variable or its negation; i.e., it is either asserting that the variable is TRUE, or that it is FALSE. A *clause* is a disjunction of literals. For example, if we had the propositional variables p, q, and r, then

$$p \vee \neg q \vee \neg r \tag{5.2}$$

asserts that either p is TRUE, or q is FALSE, or r is FALSE. Resolution is an inference rule that works with two clauses, one containing some variable negated, and the other containing that variable unnegated. Resolution creates a new clause by disjoining all the literals of the two clauses, with the exception of those two literals. So if we had the clause

$$s \vee r \vee \neg t, \tag{5.3}$$

then we could resolve this clause with (5.2) to conclude

$$p \vee \neg q \vee s \vee \neg t.$$

This is a sound inference because r must be either TRUE or FALSE. If r is TRUE, then from (5.2) we can conclude $p \vee \neg q$, and if r is FALSE, from (5.3) we can conclude $s \vee \neg t$. If at some point, we have derived the clauses v and $\neg v$ for some variable v, then we can resolve these two clauses together to get the *empty clause*, or in other words, FALSE. This is a proof that the original set of clauses was inconsistent. Resolution is said to be *refutation complete* because given any set of unsatisfiable clauses, it can ultimately derive the empty clause.

It is straightforward to extend resolution to handle non-Boolean CSPs. If v is some problem variable, and $x \in D_v$ is some value in that variable's domain, then a literal for the CSP will be some expression of the form $v \neq x$. As before, a clause is a disjunction of literals. A resolution step, now, will have to resolve together $|D_v|$ clauses for some variable v. For example, if the domain of v is $\{1, 2, 3\}$, then we can resolve together the clauses

$$(v \neq 1) \vee (a \neq 2),$$
$$(v \neq 2) \vee (b \neq 1),$$
$$(v \neq 3) \vee (c \neq 2)$$

to conclude

$$(a \neq 2) \vee (b \neq 1) \vee (c \neq 2).$$

If we start with the clauses that are direct consequences of individual constraints of some CSP, and we then derive the empty clause by resolution, then we have proven that the CSP is unsatisfiable.

Let us make this a little more formal.

Definition 5.9 *Let (V, D, C) be a CSP. A* literal *is an ordered pair (v, w) where $v \in V$ and $w \in D_v$; it is intended to represent the assertion $v \neq w$. A* clause *is a set of literals that does not repeat any variable; the clause represent the disjunction of its literals. If c is a clause, then $Vars(c)$ is the set of variables that appear in c. If $(v, w) \in c$, then we will define $Val(v, c) = w$. Finally, if $c = \emptyset$, then we will call c the* empty clause.

The intended meaning of a clause is that all of the variable assignments cannot hold simultaneously. For example, the clause $c = \{(x, 1), (y, 2), (z, 3)\}$ represents the formula

$$(x \neq 1) \vee (y \neq 2) \vee (z \neq 3),$$

and we would have $Vars(c) = \{x, y, z\}$, $Val(x, c) = 1$, $Val(y, c) = 2$, etc. A clause is a set because the order of the literals does not matter, and we do not allow a clause to repeat a variable because such a clause would be vacuous since we know in advance that a variable cannot have more than one value.

Next, we need some definitions concerning inference.

Definition 5.10 *For a particular CSP, a clause c will be called a* primitive clause *if the CSP has some constraint that directly entails that clause. More precisely, the CSP must have some constraint (W, Q), where as before, $W = (w_1, w_2, \ldots, w_k)$ is the list of the variables in the constraint, and Q is a predicate on these variables; such that W consists of the variables in the set $Vars(c)$, and*

$$(Val(w_1, c), Val(w_2, c), \ldots, Val(w_k, c)) \notin Q.$$

The other possibility is a clause that is derived by resolution.

Definition 5.11 *Let c be a clause, and let Γ be a set of clauses. We will say that c can be obtained by* resolving *together the clauses in Γ if there is some variable v such that the following conditions all hold:*

- *For all $\gamma \in \Gamma, v \in Vars(\gamma)$.*
- *For all $\gamma_1, \gamma_2 \in \Gamma, Val(v, \gamma_1) \neq Val(v, \gamma_2)$.*
- $|\Gamma| = |D_v|$.
- $c = \bigcup_{\gamma \in \Gamma} \gamma - \{(v, x) : x \in D_v\}$.
- *There is no variable w and values $x_1 \neq x_2$ such that $(w, x_1) \in c$ and $(w, x_2) \in c$.*

Here, v is the variable that is being resolved on, and the first three conditions ensure that for each value $x \in D_v$, (v, x) appears in exactly one clause in Γ. The fourth condition defines the clause c that is the result of the resolution step, and the fifth condition ensures that c is in fact a nonvacuous clause.

We can now give a formal definition of a resolution proof for a CSP:

Definition 5.12 *A* resolution proof *for a CSP is a sequence of clauses $\gamma_1, \gamma_2, \ldots, \gamma_m$ together with a justification function J that maps integers to sets of integers, such that: For all i from 1 to m, either $J(i) = \emptyset$ and γ_i is a primitive clause for the CSP, or $J(i) \subseteq [1, i)$ and γ_i can be obtained by resolving together the clauses in $\{\gamma_j : j \in J(i)\}$.*

If γ_m is the empty clause, then we will call this a resolution proof of unsatisfiability.

It is well known that the various CSP algorithms that we have discussed are essentially constructing resolution proofs. Consider the BACKJUMPING procedure, when it is called with a partial assignment f that cannot be extended to a solution. The conflict set that BACKJUMPING returns is a set of variables that cannot simultaneously have their current values. This can be written as a clause. For example if we had $f(v_i) = x_i$ for $1 \leq i \leq 9$, and the returned conflict set were $\{v_3, v_5, v_9\}$, this would correspond to the clause

$$(v_3 \neq x_3) \vee (v_5 \neq x_5) \vee (v_9 \neq x_9). \tag{5.4}$$

When backjumping loops over the values for some variable v, the returned conflict set will be the union of the conflict sets returned from the recursive calls (or from the direct constraint violations) with the exception of the variable v; this corresponds to resolving these clauses on v. Suppose the variable v_9 from the above example had only the two possible values x_9 and x_9', and after backtracking and setting $f(v_9) = x_9'$, the backjumping procedure again returned UNSAT, with the returned conflict set being $\{v_4, v_5, v_9\}$, corresponding to

$$(v_4 \neq x_4) \vee (v_5 \neq x_5) \vee (v_9 \neq x_9'). \tag{5.5}$$

Then the backjumping call would return the conflict set $\{v_3, v_4, v_5\}$, corresponding to

$$(v_3 \neq x_3) \vee (v_4 \neq x_4) \vee (v_5 \neq x_5). \tag{5.6}$$

Clause (5.6) is the result of resolving the clauses (5.4) and (5.5).

Dependency-directed backtracking is also constructing resolution proofs, but since it saves the clauses that it derives, it can often build shorter proofs than backjumping can. In other words, if a clause gets used several times in the proof, backjumping will have to rederive it each time, while dependency-directed backtracking will only have to derive it once. Dynamic backtracking and the various types of generalized dependency-directed backtracking are intermediate in sophistication between backjumping and full dependency-directed backtracking; sometimes they can reuse a clause, and sometimes they have to rederive it.

Let us record the following well-known facts:

Proposition 5.13 *A CSP is unsatisfiable if and only if there is a resolution proof of its unsatisfiability. The length of the shortest such proof is exponential only in the induced width of the CSP.*

Proof. The soundness of resolution is obvious. The completeness follows from the correspondence between resolution and any of the search algorithms mentioned above. The induced-width property follows from the correspondence between resolution and dependency-directed backtracking, or alternatively, from the correspondence between resolution and adaptive consistency. □

Now that we have reviewed the basics of resolution proofs, we can use these tools to analyze some CSP algorithms. We will define restrictions on resolution corresponding to the types of proofs that these algorithms construct. If we can show that the shortest resolution proof subject to these restrictions is not exponential only in the induced width, then this result will also apply to the running time of the corresponding CSP algorithms, even when these algorithms makes their choices optimally. We will discuss dynamic backtracking in the next section, and polynomial-space dependency-directed backtracking in Section 5.6.

5.5 Dynamic-Backtracking Resolution Proofs

In Chapter 4, we showed that dynamic backtracking using a fixed variable and value ordering is not exponential only in the induced width. We would now like to extend this result to handle an ideal version of dynamic backtracking that uses the best search order. Dynamic backtracking maintains both a partial assignment and an entire array of dependency information, and it is possible that the best search decision at a given stage may depend on both pieces of information. Therefore, specifying the optimal search rule may be difficult.

Furthermore, there are some new versions of dynamic backtracking that differ in subtle but crucial ways from the original algorithm. Specifically, McAllester [61] has introduced *generalized dynamic backtracking* and *partial-order backtracking*, and Ginsberg and McAllester [45] have introduced *partial-order dynamic backtracking*. When standard dynamic backtracking has to backtrack, it always revises the most recently bound variable in the conflict set (this variable is selected on line 17 of the algorithm on page 18). The newer algorithms, however, allow a greater flexibility in this decision; in turn, these algorithms require more bookkeeping to keep track of which backtrack decisions are permissible at any point. Given the variety and complexity of the different dynamic backtracking algorithms, it would be tedious to analyze them case by case. Instead, we will proceed more generally.

The DYNAMIC-BACKTRACKING algorithm that we discussed in Chapter 1 stored its dependency information in a *culprits* array. Suppose f is the current partial assignment, v is a variable, and $x \in D_v$ is some value in the domain of v. Then, if *culprits*$[v, x]$ is not VALUEALLOWED, it will be a set of variables whose current assignments preclude v from being set to the value x. This information can be written declaratively as a clause. For example, if currently $f(v_i) = x_i$, for $1 \leq i \leq 9$, then the entry *culprits*$[v, x] = \{v_3, v_5\}$ corresponds to the clause

$$(v_3 \neq x_3) \vee (v_5 \neq x_5) \vee (v \neq x).$$

It is customary in the dynamic backtracking literature to write this clause in the directed form

$$(v_3 = x_3) \wedge (v_5 = x_5) \Rightarrow v \neq x. \tag{5.7}$$

We will calls these statements, *directed nogoods*, or just nogoods, for short. When dynamic backtracking records a nogood of the form (5.7), it unsets the variable v; conversely, whenever dynamic backtracking retracts an assignment $f(v) = x$, it records the reason for this backtrack with a nogood whose right-hand side is $v \neq x$.

Dynamic backtracking also *deletes* dependency information. If $w \in culprits[v, x]$ and the current value of the variable w is then retracted, then $culprits[v, x]$ is set back to VALUEALLOWED. This corresponds to deleting the associated nogood. Since the only time that the value of w can be retracted is when a new nogood is learned whose right-hand side rules out the current value of w, the deletion rule can be formalized entirely in terms of nogoods with no reference to the current assignment f. The deletion rule of dynamic backtracking is that if a nogood is acquired whose right-hand side is $v \neq x$, then every nogood whose left-hand side contains $v = x$ must be deleted. One might think of this in terms of "relevance." As long as all the assignments in the precondition of a nogood are true, this nogood actually has immediate power; it prevents the variable on the right-hand side from assuming a particular value. If any assignment in the precondition is not currently true, then the nogood is temporarily irrelevant since it does not prevent the assignment in its conclusion. Dynamic backtracking immediately deletes any nogood that has become irrelevant. The problem, of course, is that a nogood that is currently irrelevant may become relevant again, and then dynamic backtracking will have to derive it all over again (this is what happened in Example 4.14, the example that was used to establish Proposition 4.16). Now that we have reviewed the role of nogoods in dynamic backtracking, let us move on to the new material of this section.

We will show in this section that regardless of the search order, neither the original dynamic backtracking algorithm, nor any of its recent variants like partial-order dynamic backtracking, is exponential only in the induced width. In fact, we will show something even stronger. We will define the notion of a *DB-consistent resolution proof*, in which nogoods are deleted using the deletion rule from dynamic backtracking, and then prove that no such procedure can yield unsatisfiability proofs that are exponential only in the induced width. To put it another way, even for problems of bounded induced width, the lengths of the shortest unsatisfiability proofs that dynamic backtracking can construct may grow faster than any polynomial in the size of the problem.

First, we will need a few preliminary definitions.

Definition 5.14 *If c is a clause for some CSP, and $v \in Vars(c)$ is some variable that appears in that clause, then (c, v) will be called a* directed nogood, *or simply a* nogood, *for short. If c is the empty clause, then $(c, \emptyset)$ will be the corresponding nogood, which we will call the* empty nogood.

The nogood is intended to represent the directed version of the clause. For example, the nogood $\gamma = (\{(x,1),(y,2),(z,3)\}, y)$ represents

$$(x = 1) \wedge (z = 3) \Rightarrow y \neq 2. \tag{5.8}$$

We will need a few abbreviations for referring to the components of nogoods, just as we did for clauses.

Definition 5.15 *For a nogood $\gamma = (c, r)$, we will define $Clause(\gamma) = c$, $Rvar(\gamma) = r$, $Vars(\gamma) = Vars(c)$, $Lvars(\gamma) = Vars(c) - \{r\}$, and $Val(v,\gamma) = Val(v,c)$.*

For the nogood γ of (5.8), we would have $Vars(\gamma) = \{x,y,z\}$, $Lvars(\gamma) = \{x,z\}$, $Rvar(\gamma) = y$, $Val(x,\gamma) = 1$, $Val(y,\gamma) = 2$, etc.

The definitions of inference for nogoods are based on the corresponding definitions for clauses.

Definition 5.16 *For a particular CSP, a nogood γ will be called a* primitive nogood *if $Clause(\gamma)$ is a primitive clause for the CSP. If γ is a nogood, and Γ is a set of nogoods, then we will say that γ can be obtained by* resolving *the nogoods in Γ if $Clause(\gamma)$ can be obtained by resolving the clauses $\{Clause(x) : x \in \Gamma\}$ on the variable v, where $v = Rvar(g)$ for all $g \in \Gamma$.*

In other words, we may only resolve a nogood on the variable on its right-hand side. For example, assuming that the domain of z is $\{1, 2\}$, we could resolve

$$x = 1 \Rightarrow z \neq 1$$

with

$$y = 1 \Rightarrow z \neq 2$$

to obtain either

$$x = 1 \Rightarrow y \neq 1$$

or

$$y = 1 \Rightarrow x \neq 1.$$

We are now almost ready to give the definition of a DB-consistent resolution proof. As in dynamic backtracking and partial-order dynamic backtracking, the rule is that if the consequent of the new nogood appears on the left-hand side of an old nogood, then the old nogood must be deleted.

Definition 5.17 *Let γ_1 and γ_2 be nogoods, and let $r = Rvar(\gamma_1)$. Nogood γ_1* deletes *γ_2 if $r \in Lvars(\gamma_2)$ and $Val(r, \gamma_1) = Val(r, \gamma_2)$.*

In a particular proof, of course, γ_2 may have already been deleted by the time that γ_1 is derived (or γ_1 may be derived before γ_2). In any event, if γ_1 deletes γ_2, then γ_2 will not be in memory immediately after γ_1 is derived. A DB-consistent resolution proof is simply a resolution proof in which one uses only nogoods that have not been deleted. More formally:

Definition 5.18 *A* DB-consistent resolution proof *for a CSP is a sequence of nogoods $\gamma_1, \gamma_2, \ldots, \gamma_m$, together with a justification function J that maps integers to sets of integers, that has the following property: For all i from 1 to m, either $J(i) = \emptyset$ and γ_i is primitive; or $J(i)$ is a set of integers in the interval $[1, i)$, such that γ_i can be obtained by resolving together the nogoods $\{\gamma_j : j \in J(i)\}$, and for every $j \in J(i)$, there is no k, $j < k < i$, such that γ_k deletes γ_j.*

If γ_m is the empty nogood, then we will call this a DB-consistent resolution proof of unsatisfiability.

The following lemma provides the rationale for the above definitions.

Lemma 5.19 *If $\gamma_1, \gamma_2, \ldots, \gamma_m$ are the nogoods that are generated (in this order) by the dynamic backtracking algorithm running on some some CSP, then $\gamma_1, \gamma_2, \ldots, \gamma_m$ must be a DB-consistent resolution proof for that CSP. If $f(\rho)$ is the minimal running time of dynamic backtracking with optimal control for an unsatisfiable CSP ρ, and if $g(\rho)$ is the length of the shortest DB-consistent resolution proof of unsatisfiability for ρ, then $f(\rho) = \Omega(g(\rho))$. This result also applies to generalized dynamic backtracking, partial-order backtracking, and partial-order dynamic backtracking.*

Proof. This is obvious for the original dynamic backtracking. We have not described the other algorithms in detail, but the interested reader can easily verify this lemma by referring to [45, 61]. □

We can now state the main result of this section.

Theorem 5.20 *There exists a collection of unsatisfiable CSPs with a constant induced width, such that the lengths of the shortest DB-consistent resolution proofs of unsatisfiability grow faster than any polynomial in the size of the problem. Therefore, the length of the shortest DB-consistent resolution proof of unsatisfiability for a CSP is not exponential only in the induced width.*

Here is the example that will be used to establish the theorem.

Example 5.21 *The problem P_n will be defined as follows. The set of variables is $\{X_1, Y_1, X_2, Y_2, \ldots, X_n, Y_n\}$. The variables will all have the same domain: $\{1, \ldots, 2n\}$. To define the constraints, we will use the abbreviation $M(x, y)$ for the remainder of $x + y$ modulo 2. The constraints will be as follow:*

$$M(X_1, Y_1) = 0, \tag{5.9}$$

$$M(X_n, Y_n) = 1, \tag{5.10}$$

and for $1 \leq i < n$:

$$M(X_i, Y_i) \geq M(X_{i+1}, Y_{i+1}). \tag{5.11}$$

The constraint graph corresponding to P_5 is pictured in Figure 5.2.

All of the problems in this sequence are unsatisfiable, and have a constant induced width (of 3). The maximum domain size is $O(n)$, and there are $O(n)$ constraints with a maximum arity of 4, so the size of the problem is $O(n(n^4)) = O(n^5)$. Since this is polynomial in the parameter n, all we have to show is that the length of the shortest DB-consistent resolution proof of the unsatisfiability of P_n is not polynomial in n.

Note that a resolution proof of unsatisfiability corresponds in an obvious way to a labeled directed acyclic graph with the leaves being labeled by primitive nogoods, and

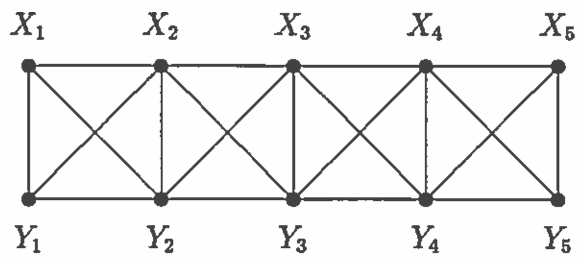

Figure 5.2: The constraint graph of problem P_5 of Example 5.21.

the root node being labeled by the empty nogood. We will refer to this graph as the *proof graph.* For the current example, every node in the proof graph that is labeled by a non-primitive nogood will have exactly $2n$ children, labeled by the nogoods that were resolved together to deduce that nogood. We will show that the smallest proof graph for problem P_n has a super-polynomial number of leaf nodes.

We need to define an appropriate measure of "difficulty" for nogoods. All of the subsequent definitions in this section will be specific to Example 5.21. Let us start with the notion of a *bracket*:

Definition 5.22 *Consider a nogood γ. Let the* left bracket, *b_l, of γ be the maximum i such that $X_i \in Vars(\gamma)$, $Y_i \in Vars(\gamma)$, and*

$$M(Val(X_i,\gamma), Val(Y_i,\gamma)) = 0.$$

If no such i exists, then set b_l to 0. Similarly, the right bracket, *b_r is the minimum i such that $X_i \in Var(\gamma)$, $Y_i \in Vars(\gamma)$, and*

$$M(Val(X_i,\gamma), Val(Y_i,\gamma)) = 1.$$

In this case, if there is no such i, then set b_r to be $n+1$. If $b_l < b_r$, we will say that the nogood γ is orderly, *and we will call $[b_l, b_r]$ the* bracket *of the nogood.*

To give some examples for $n = 8$: the nogood

$$X_3 = 6 \Rightarrow Y_3 \neq 7 \tag{5.12}$$

would have the bracket $[0, 3]$, the nogood

$$X_3 = 6 \Rightarrow Y_3 \neq 6 \tag{5.13}$$

would have the bracket [3,9], the nogood

$$(X_2 = 6) \wedge (Y_2 = 6) \wedge (X_3 = 6) \Rightarrow Y_3 \neq 7 \tag{5.14}$$

would have the bracket [2,3], and the nogood

$$(X_2 = 6) \wedge (Y_2 = 7) \wedge (X_3 = 6) \Rightarrow Y_3 \neq 6$$

would be a disorderly nogood.

The intuition here is that for an orderly nogood, the inconsistency is "trapped" in the bracket; everything outside the bracket is essentially irrelevant. In order to prove the nogood, we would have to somehow reason from one end of the bracket to the other. The question is how hard it is to do this; for this purpose we introduce the notion of a *sub-bracket.*

Definition 5.23 *Let γ be an orderly nogood with bracket $[b_l, b_r]$. We will call the pair (i,j) a* sub-bracket *of γ if $b_l \leq i < j \leq b_r$ and for all k in the open interval (i,j), neither X_k nor Y_k is in $Lvars(\gamma)$. The* difficulty *of this sub-bracket is $j - i$. Finally, we say that a sub-bracket (a,b) is* contained *in another sub-bracket (i,j) when $i \leq a < b \leq j$.*

For example, the nogood

$$(X_1 = 5) \wedge (X_3 = 6) \Rightarrow (Y_3 \neq 7) \tag{5.15}$$

has a bracket of [0,3], and it has sub-brackets of (0,1), (1,2), (2,3), and (1,3), with difficulties of 1, 1, 1, and 2 respectively. Note that there is no requirement that a sub-bracket be maximal (and hence if (i,j) is a sub-bracket for some nogood, and if we have some a and b such that $i \leq a < b \leq j$, then (a,b) will be a sub-bracket as well). Note also that the variable on the right-hand side is used in defining the bracket but not the sub-bracket; thus the nogood

$$(X_3 = 6) \wedge (Y_3 = 7) \Rightarrow (X_1 \neq 5),$$

unlike the nogood (5.15), would have a sub-bracket of (0,3) even though the two nogoods are logically equivalent.

We want to show that a nogood with a sub-bracket of high "difficulty" is indeed difficult to prove. The following two propositions are straightforward.

Proposition 5.24 *Any primitive nogood is orderly, and its only sub-bracket has difficulty 1.*

Proof. A primitive nogood must be directly entailed by one of the original problem constraints. If this constraint is of the form (5.9), then the nogood will rule out some values x_1 and y_1 for X_1 and Y_1 respectively such that

$$x_1 + y_1 \equiv 1 \pmod 2.$$

Thus, the bracket will be $[0, 1]$. Similarly, if it is of the form (5.10), its bracket will be $[n, n+1]$. For the final possibility of a constraint of the form (5.11) for $1 \leq i < n$, the bracket would be $[i, i+1]$. In each case, the only possible sub-bracket will have the same bounds as the bracket, and will thus have a difficulty of 1. ☐

Proposition 5.25 *The empty nogood is orderly, and it has a sub-bracket of difficulty* $n+1$.

Proof. The bracket of the empty nogood is $[0, n+1]$, and $(0, n+1)$ obviously satisfies the requirements for being a sub-bracket. ☐

It is also the case that one step of resolution cannot increase the overall difficulty very much:

Lemma 5.26 *Let γ be an orderly non-primitive nogood that is obtained by resolving together the nogoods in Γ. Assume that γ has some sub-bracket (i, j) of difficulty $d = j - i$. Then at least n elements of Γ are orderly nogoods, each having a sub-bracket contained in (i, j) with a difficulty of at least $d/2$.*

Proof. Let $Rvar(\gamma)$ have index k, i.e., be either X_k or Y_k. We will define the sub-brackets of the desired elements of Γ as follows. If k is not in the open interval (i, j), then (i, j) will be the new sub-bracket; otherwise, choose the larger of the intervals (i, k) or (k, j). Call this new range (a, b). If (a, b) is a sub-bracket for n nogoods in Γ, then the lemma is proven since $b - a \geq (j - i)/2$.

To show that (a, b) is indeed a sub-bracket for a particular nogood $\gamma' \in \Gamma$, we will have to show two things. First, note that when you resolve some γ' with other nogoods to produce γ, you will only eliminate the right-hand variable of γ'. Thus, $Lvars(\gamma') \subseteq Vars(\gamma)$, and so $Lvars(\gamma')$ is disjoint from the open interval (a, b) since $Vars(\gamma)$ is disjoint from this interval.

Second, we have to show that γ' is orderly and that (a, b) is contained in its bracket. Note that all of the elements of Γ will have the same right-hand variable; without loss of generality, let it be X_q. If Y_q is not in $Vars(\gamma)$, then it also cannot be in $Vars(\gamma')$ (since $Lvars(\gamma') \subseteq Vars(\gamma)$ and $Rvar(\gamma') \neq Y_q$). In this case, X_q cannot affect the bracket since its partner Y_q is missing. Since X_q is the only variable in $Vars(\gamma')$ that is not present in $Vars(\gamma)$, it follows that γ' is orderly and that its bracket includes the bracket of γ and hence (a, b) as well.

The remaining possibility is that $Y_q \in Vars(\gamma)$. Note that by the construction of (a, b), it is then impossible for $a < q < b$. So if $q \leq a$, then select those $\gamma' \in \Gamma$ such that $M(Val(X_q, \gamma'), Val(Y_q, \gamma)) = 0$; and if $q \geq b$, then choose those such that $M(Val(X_q, \gamma'), Val(Y_q, \gamma)) = 1$. Exactly half of the $2n$ elements of Γ will satisfy this condition. Any $\gamma' \in \Gamma$ that satisfies this condition will be orderly and will have a bracket that includes (a, b). This completes the proof. $\square$

Lemma 5.26 (together with Propositions 5.24 and 5.25) would be sufficient to show that the resolution proof *tree* is super-polynomial. We are interested, however, in the proof *graph*. The difference is that, in the graph, a node can have more than one parent, allowing work to be reused; but since we are only considering DB-consistent resolution proofs, there are limits to how *much* work can be reused (as we will soon see).

Let g_0 be a proof graph for some example P_n. We will make use of various subgraphs of g_0. Each such subgraph induces a partial order if we assume that children precede their parents. If a subgraph has a single maximal element (i.e., an ancestor of every other node in the subgraph), then we will say that the subgraph is *rooted*:

Definition 5.27 *If g is a subgraph of g_0, we will say that it is a* rooted subgraph *if there is some root node in g that is an ancestor of every other node in g, where ancestorship is defined relative to g.*

Of particular interest are those rooted subgraphs that have the *self-deletion* property:

Definition 5.28 *If g is a rooted subgraph of g_0, then we will say that g is* self-deleting *if every non-root node in g is deleted by at least one of its ancestors in g.*

The intuition here is that once the root node has been derived, the rest of the subgraph will have been deleted. Such a condition will be crucial in limiting the extent to which work can be reused.

Happily, there is a simple condition that will ensure that a subgraph is self-deleting.

Lemma 5.29 *Let g be a rooted subgraph of g_0 with root r. Assume that (i,j) is a sub-bracket for r, but for no other nodes in g. Finally, assume that each node in g has some sub-bracket that is contained in (i,j). Then, g is self-deleting.*

Proof. Let x be a non-root node in g. By assumption, x will have some sub-bracket (a,b) such that $i \leq a < b \leq j$. We claim that $Lvars(x) \not\subseteq Lvars(r)$. Let b_l be the left bracket of x, and let b_r be the right bracket of x; by our definitions, we know that $b_l \leq a$ and $b \leq b_r$. If we had $b_l \leq i$ and $j \leq b_r$ and $Lvars(x)$ disjoint from the open interval (i,j), then (i,j) would be a sub-bracket for x. Since we are assuming that this is not the case, one of these three premises must be false. If the first premise is false, then $i < b_l \leq a$. By the definition of the left bracket, at least two variables in $Vars(x)$ must have indices between i and a; since only one of them can be the right-hand variable $Rvar(x)$, at least one variable in $Lvars(x)$ must be in this range. Similarly, if $b \leq b_r < j$, then some variable in $Lvars(x)$ must be in the range (b,j). So in any of the three cases, $Lvars(x)$ is not disjoint from the open interval (i,j). But by the definition of a sub-bracket, $Lvars(r)$ must be disjoint from (i,j); hence, $Lvars(x) \not\subseteq Lvars(r)$.

Therefore, there must be some variable v in $Lvars(x)$ that is not present in $Lvars(r)$. Consider some path in g from x up to r. Let q be the first node on this path such that $v \notin Lvars(q)$. Since resolution never directly eliminates a variable from the left-hand side of a nogood, v must be the same as $Rvar(q)$, and thus

q deletes x. Since x was an arbitrary non-root node, this shows that the subgraph is self-deleting. □

The self-deletion property can be used in proving that certain subgraphs of g_0 are disjoint.

Lemma 5.30 *Let g_1 and g_2 be two self-deleting rooted subgraphs of g_0 having roots r_1 and r_2 respectively such that the node r_1 does not appear in the graph g_2, and the node r_2 does not appear in the graph g_1. Then g_1 and g_2 are disjoint.*

Proof. Assume by way of contradiction that g_1 and g_2 intersect, i.e., that some node is present in both graphs. Let x be a maximal such node. Then there is a path from x to r_1

$$(x = v_0, v_1, \ldots, v_s = r_1) \tag{5.16}$$

and a path from x to r_2

$$(x = w_0, w_1, \ldots, w_t = r_2) \tag{5.17}$$

such that each contains some node that deletes x. Let v_{i_1} be the minimal such element in the first path, and let w_{i_2} be the minimal such element in the second path. These must be distinct since x is the maximal node that is common to both graphs. Now, if v_{i_1} deletes x, this implies that $Rvar(v_{i_1}) \in Lvars(x)$; similarly, $Rvar(w_{i_2}) \in Lvars(x)$. But since these are the *minimal* such deleters, neither $Rvar(v_{i_1})$ nor $Rvar(w_{i_2})$ can appear on the right-hand side of any v_i such that $0 \leq i < i_1$, or any w_i such that $0 \leq i < i_2$. By the definition of a path in the proof graph, we know that v_0 is resolved with other nogoods to derive v_1, and then v_1 is resolved with other nogoods to derive v_2, and so forth; and the same goes for the other path. We also know that resolution cannot directly eliminate a variable from the left-hand side of a nogood; in other words, if some variable is in $Lvars(v_0)$ but not in $Lvars(v_1)$, then it must be equal to $Rvar(v_1)$, etc. Since $Rvar(v_{i_1})$ and $Rvar(w_{i_2})$ are not on the right-hand sides of any of the nogoods just mentioned, they must be present on all the left-hand sides. Therefore, both v_{i_1} and w_{i_2} not only delete x, but also delete all v_i such that $0 \leq i < i_1$, and all w_i such that $0 \leq i < i_2$. Now, consider the actual proof, and assume without loss of generality, that v_{i_1} is derived before w_{i_2}. At the time that v_{i_1} is derived there will be some maximal i such that w_i has been derived, and i will be less than i_2. But since v_{i_1} deletes this w_i, w_i cannot be used to later derive w_{i+1}. This contradicts our assumption that (5.17) is a path from x to r_2 in the proof graph. Hence, g_1 and g_2 must be disjoint. □

In order to establish a lower bound on the size of the proof graph g_0, we will identify certain special subgraphs.

Definition 5.31 *Let g be a rooted subgraph of g_0 with root r. We will say that g is a (β, d)-*special subgraph *of g_0 if $\beta \geq 1$ and $d \geq 1$, and the following properties all hold:*

1. *The root r has a sub-bracket (i, j) with difficulty $j - i \geq d$.*
2. *No other node in g has the sub-bracket (i, j).*
3. *Every node in g has some sub-bracket that is contained in (i, j).*
4. *Any non-leaf node x in g has, as children, all the children of x in the original proof graph g_0, with the exception of those children that violate condition 3 and at most $n - \beta$ other children.*

This definition is rather complex, and its motivation will not be fully clear until we use it in Lemma 5.33. We will make one quick comment, however. The β in this definition is related to the branching factor of the graph; specifically, we have:

Proposition 5.32 *If g is a (β, d)-special subgraph, then every non-leaf node in g has at least β children.*

Proof. Let x be a non-leaf node in g. It has $2n$ children in the original proof graph g_0. From Lemma 5.26, it is easy to see that at most n of these children will violate condition 3 from Definition 5.31. Since at most $n - \beta$ additional children can be omitted, there must be at least β children remaining. □

We will now proceed to give a lower bound on the size of a special subgraph.

Lemma 5.33 *Let g be a (β, d)-special subgraph of g_0. Then g has at least*

$$\binom{\beta + \lfloor \log_2 d \rfloor - 1}{\lfloor \log_2 d \rfloor} \tag{5.18}$$

leaf nodes.

Proof. Let $f(\beta, d)$ represent the minimal number of leaves in any (β, d)-special subgraph. Let r be the root node of g, and let (i, j) be the sub-bracket referred to in Definition 5.31.

If r is a leaf node, then from Proposition 5.24 and from condition 1 in Definition 5.31, we know that d is 1. In this case, formula (5.18) is equal to 1, and the lemma is true trivially.

If r is not a leaf node, then consider all the descendents of r in g that have a sub-bracket contained in (i, j) with a difficulty of at least $d/2$. We claim that there are at least β such descendents. From Lemma 5.26, we know that at least n of the children of r in the original graph g_0 will satisfy this criterion; certainly, none of these children can violate condition 3 of Definition 5.31. Therefore, from condition 4 of that definition, we know that at least β of these children will be present in g. Hence, there at least β descendents that satisfy the criterion.

Consider some topological sort of these descendents where if node x_1 is a descendent (in g) of node x_2, then x_1 must precede x_2 in the topological sort. We will define subgraphs of g rooted at the first β nodes in this topological sort:

$$x_\beta < x_{\beta-1} < \cdots < x_2 < x_1.$$

Construct subgraphs of g in the following way. Node x_β has some sub-bracket (a, b) such that (a, b) is contained in (i, j), and $b - a \geq d/2$. Let g_β be the largest rooted subgraph of g with root x_β for which all nodes have a sub-bracket contained in (a, b). We claim that g_β is a $(\beta, d/2)$-special subgraph. Condition 1 (with the sub-bracket being (a, b), of course) was the criterion used to select the descendents of r. Condition 2 can be shown by the following argument. If any descendent of x_β had a sub-bracket of (a, b), then this descendent would itself be one of the x_i's since (a, b) is contained in (i, j) and has a difficulty of at least $d/2$. Furthermore, any descendent of x_β must precede it in our topological sort. This cannot be, however, since x_β was defined to be the first x_i in the topological sort. Therefore, no descendent of x_β can have the sub-bracket (a,b); this establishes Condition 2. Condition 3 is also satisfied, as we chose g_β to be the largest subgraph of g rooted at x_β that satisfied this condition. Finally, to establish condition 4, consider an arbitrary non-leaf node x in g_β. At most n children of x in the *original* graph g_0 will violate condition 3 for the sub-bracket (a,b) (see Lemma 5.26). The children of x that violated condition 3 for the old sub-bracket (i,j) (and thus were not present in g) will continue to violate it for the new sub-bracket (a,b). After all, (a,b) is contained in (i,j); if the child does not have a sub-bracket contained in (i,j), then it certainly cannot have one in (a,b). The only other children of x absent from g_β are the (at most) $n-\beta$ additional children of x absent from g. Thus, condition 4 is satisfied as well, and g_β is in fact a $(\beta,d/2)$-special subgraph.

The above analysis has been for the new subgraph g_β. We will now define subgraphs $g_{\beta-1}$, $g_{\beta-2}$, ..., g_1. For any i, such that $\beta > i \geq 1$, we will again have some sub-bracket (a,b) of x_i such that (a,b) is contained in (i,j) and $b-a \geq d/2$. For $\beta > i \geq 1$, let g_i be the largest rooted subgraph of g with root x_i that does not include the nodes $x_{i'}$ for $\beta \geq i' > i$, and for which all nodes have a sub-bracket contained in (a,b). The purpose of filtering out the $x_{i'}$ that precede x_i in the ordering is so that condition 2 will continue to be satisfied. The only consequence of this filtering is that g_i will now be a $(i,d/2)$-special subgraph instead of a $(\beta,d/2)$-special subgraph. To see this, note that a nogood cannot resolve with itself, so each of the earlier $x_{i'}$ can only be a child a single time of any parent.

Now, we can apply Definition 5.31 to the subgraphs g_β, $g_{\beta-1}$, ..., g_1. We know from Lemma 5.29 that these subgraphs must be self-deleting. By the above discussion, the root of any one of these subgraphs cannot appear in any of the other subgraphs. Therefore, by Lemma 5.30, the subgraphs must be completely disjoint. Since the subgraphs are disjoint, the number of leaves in the original graph cannot be less than the sum of the leaves in the new subgraphs. Hence, we have shown:

$$f(\beta,d) \;\geq\; \sum_{i=1}^{\beta} f(i,\lceil d/2\rceil).$$

We also know that

$$f(\beta, 1) \; = \; 1.$$

It will be easier to solve this recurrence if we convert it to an arithmetic reduction. Define $g(\beta, x)$ to be $f(\beta, 2^x)$. Then we have the equations

$$\begin{aligned} g(\beta, x) \; &\geq \; \sum_{i=1}^{\beta} g(i, x-1), \\ g(\beta, 0) \; &= \; 1. \end{aligned}$$

It is straightforward to prove by induction that this is solved by

$$g(\beta, x) \geq \binom{\beta + x - 1}{x}.$$

Hence, we have

$$f(\beta, d) \geq \binom{\beta + \lfloor \log_2 d \rfloor - 1}{\lfloor \log_2 d \rfloor},$$

which is what we want. □

Theorem 5.20 now follows easily:

Proof. Let g_0 be a proof graph for problem P_n of Example 5.21. The root node of g_0 is labeled by the empty nogood, so it has a sub-bracket of $(0, n+1)$. Let r be the minimal node in g_0 with this property. Let g be the largest rooted subgraph of g_0 with root r such that all the nodes are orderly. According to Definition 5.31, g is a $(n, n+1)$-special subgraph. Applying Lemma 5.33, we conclude that g must have at least

$$\binom{n + \lfloor \log_2 (n+1) \rfloor - 1}{\lfloor \log_2 (n+1) \rfloor}$$

leaf nodes, and thus any DB-consistent resolution proof of unsatisfiability for Problem P_n will have at least this many steps. For sufficiently large n, this is more than

$$\binom{n}{\lfloor \log_2 n \rfloor} \; \geq \; \left(\frac{n}{\lfloor \log_2 n \rfloor} \right)^{\lfloor \log_2 n \rfloor}.$$

For sufficiently large n, $\lfloor \log_2 n \rfloor$ is less than (say) $n^{0.01}$, and thus we have the asymptotic lower bound

$$n^{0.99\lfloor \log_2 n \rfloor},$$

which is greater than any polynomial. □

Corollary 5.34 *Regardless of the search order, neither dynamic backtracking nor partial-order dynamic backtracking is exponential only in the induced width.*

Proof. This follows immediately from Theorem 5.20 and Lemma 5.19. □

5.6 Polynomial-Space Resolution

We have proven a number of lower bounds for various polynomial-space CSP algorithms. In each case that we have considered so far, we have shown that the algorithm is not exponential only in the induced width. In this section, we will make some more general conjectures (but we will not prove anything).

We begin with the most obvious conjecture:

Conjecture 5.35 *There is no polynomial-space CSP algorithm that is exponential only in the induced width.*

Recall that for a CSP algorithm to be polynomial space and exponential only in the induced width, we would need a constant c and a function f such that the algorithm uses $O(x^c)$ space and $O(x^{f(w^*)})$ time, where x is the size of the problem and w^* is its induced width. Our conjecture is that there is no such algorithm.

Conjecture 5.35 is at least as strong as the conjecture that P $\neq$ NP. To see this, consider the *CSP Decision Problem*, that is, the problem of determining whether the CSP has a solution or not. This problem is clearly in NP, so if P were equal to NP, then we would have a polynomial-time algorithm for this decision problem. It would be straightforward, however, to use this polynomial-time algorithm for the decision problem to construct a polynomial-time algorithm for the standard search problem of actually finding a solution to the CSP. In other words, if P = NP, then we would have a polynomial-*time* CSP algorithm. A polynomial-time algorithm would be (trivially) exponential only in the induced width, and it would also be polynomial space since you cannot use more space than time. Therefore, Conjecture 5.35 implies P $\neq$ NP. One interesting project would be to try to reduce Conjecture 5.35 to the question of P $\neq$ NP, or perhaps to some other open question or questions.

Let us record an even more general conjecture. Suppose we are only interested in proving unsatisfiability. One might imagine an algorithm that, for any unsatisfiable problem, eventually returns the answer UNSAT, but otherwise need not terminate at all. Our conjecture is that this does not really help.

Conjecture 5.36 *There is no polynomial-space proof method (i.e., nondeterministic machine) for recognizing unsatisfiable CSPs that is exponential only in the induced width.*

The reason that this conjecture is plausible is that the unsatisfiable problems seem to be the hardest ones. For a satisfiable problem, you might get lucky and find a solution very quickly, but for an unsatisfiable problem, you have to prove that there is *no* solution. Just as Conjecture 5.35 would imply that P $\neq$ NP, Conjecture 5.36 would imply that NP $\neq$ co-NP.

Given that these two conjectures seem to be far beyond what we can prove, let us consider a more specific conjecture that still generalizes our earlier results. Instead of allowing an arbitrary proof method as in Conjecture 5.36, we will restrict our attention to resolution proofs.

In Chapter 4, we proved that there is no polynomial-space dependency-directed backtracking algorithm that is exponential only in the induced width. There are a number of ways that one might want to generalize this result. First of all, we assumed that the search was performed using fixed variable and value orderings. Instead, one might want to consider an algorithm that makes the optimal decision at each stage. One might also want to consider alternative methods of constructing resolution proofs besides the depth-first search strategy implicit in dependency-directed backtracking. For example, one might precede (or interleave) the backtracking phase with a constraint propagation phase as in Freuder's algorithm [34] and Dechter and Pearl's adaptive consistency algorithm [30]. In fact, there are a number of such constraint propagation techniques [58], and we will discuss them in detail in Chapter 6. For our purpose here, however, the important thing is that all of these methods are special cases of resolution theorem proving. If we can prove that there is no resolution method that is simultaneously exponential only in the induced width and polynomial space, then we will have also shown that no combination of the above search methods can achieve these two objectives at once.

Let us state this idea formally. First, let us rephrase the previous definition of a resolution proof.

Definition 5.37 *A* resolution proof *for a CSP is a sequence of clauses $\gamma_1, \gamma_2, \ldots, \gamma_m$ together with a sequence of sets of clauses $\Gamma_1, \Gamma_2, \ldots, \Gamma_m$, and a justification function J that maps integers to sets of clauses such that the following properties hold:*

- $\Gamma_1 = \emptyset$.
- *For all i from 1 to m, either $J(i) = \emptyset$ and γ_i is a primitive clause for the CSP, or $J(i) \subseteq \Gamma_i$ and γ_i can be obtained by resolving together the clauses in $J(i)$.*
- *For all i from 1 to $m-1$, $\Gamma_{i+1} \subseteq \Gamma_i \cup \{\gamma_i\}$.*

If γ_m is the empty clause, then we will call this a resolution proof of unsatisfiability.

This definition is equivalent to Definition 5.12 because we can always take Γ_i to be $\{\gamma_j : 1 \leq j < i\}$. (We have also changed the definition of the justification function J to be the set of clauses that are resolved instead of their indices.) Our interest will be in resolution proofs that use only a limited amount of memory; this notion can be formalized by placing restrictions on the Γ_i sets.

Definition 5.38 *Consider a resolution proof whose clause sets are $\Gamma_1, \Gamma_2 \ldots, \Gamma_m$. The* length *of the proof is m, and the* maximum cache size *of the proof is*

$$\max_{1 \leq i \leq m} |\Gamma_i|.$$

A resolution method *is a function that when given an unsatisfiable CSP returns a resolution proof of its unsatisfiability. A resolution method is* exponential only in the induced width *if the lengths of the proofs it returns are exponential only in the induced width. A resolution method is* polynomial space *if the maximum cache sizes of the proofs it returns are bounded by some polynomial in the problem size.*

These important definitions deserve a few comments. First, note that we are not interested here in a resolution method as an *algorithm*; that is, we are not analyzing *its* space and time complexity, but rather that of the proof that it returns. A resolution method is simply an abstract function that returns a resolution proof when given a CSP; it might for example return the shortest such proof that possesses a particular property. If the length of such a proof grows only exponentially in the induced width, then we say that the resolution method is exponential only in the induced width, regardless of whether there is any algorithm that can efficiently discover this proof. This is consistent with the policy in this chapter of giving the various search techniques every benefit of the doubt. Second, note that the polynomial-space property applies to the maximum size of the Γ_i sets — not to the proof as a whole. An actual CSP algorithm has no need to store the entire proof in memory at any time; the maximum cache size, then, is the logical lower bound on how much memory this algorithm will require.

We can now state our conjecture concerning polynomial-space resolution.

Conjecture 5.39 *There is no polynomial-space resolution method that is exponential only in the induced width.*

This is weaker than Conjecture 5.36 because it only applies to resolution proofs rather than an arbitrary proof system.

We believe it likely that Example 5.7, the localized pigeonhole problem that was used to establish the lower bound for optimal k-order learning, will also work for Conjecture 5.39, but for now, we will have to leave this as an open problem.

5.7 Summary

In this chapter, we have studied the worst-case performance of various CSP algorithms. We have shown that k-order learning and dynamic backtracking are not exponential only in the induced width, even when they are allowed an optimal search order. We have also presented a more general conjecture concerning polynomial-space resolution. If this conjecture were proven, it would generalize all the results of this chapter and of Chapter 4.

Chapter 6

Related Work

This chapter discusses the related work on constraint satisfaction problems. We begin in Section 6.1 with a survey of the various backtracking ideas, focusing on those that we have not covered yet. Section 6.2 discusses constraint propagation algorithms; these are procedures, like those of Freuder [34] and Dechter and Pearl [30], that enforce some type of local consistency on the constraint network. Section 6.3 discusses stochastic algorithms for CSPs. Since these procedures cannot prove unsatisfiability, they are really outside the scope of this thesis, but they are sufficiently important that no survey of CSPs would be complete without them.

In this thesis, we have focused on one particular measure of CSP difficulty, the induced width, noting that a CSP class of bounded induced width is tractable. In Section 6.4, we discuss some other measures of CSP tractability.

We proved some results in Chapter 5 concerning the lengths of the shortest unsatisfiability proofs subject to certain restrictions. In Section 6.5, we provide some pointers into the literature on this important topic in computer science.

6.1 Backtracking Algorithms

6.1.1 Historical Notes

Backtracking is such a simple idea that it has undoubtedly been discovered numerous times by various researchers. The term "backtrack" was apparently coined by D.H. Lehmer in the 1950's.[1] The first general exposition of the principle is due to Walker [84]. Golomb and Baumert [46] and Bitner and Reingold [8] provided early surveys of backtracking. Knuth [55] gives a good introduction to backtracking, and he discusses a stochastic method for estimating the size of the backtrack search tree. For a modern survey of backtracking, the reader might consult [12] or [17]. We should note that backtracking is useful for more than constraint satisfaction problems. It is applicable on any search problem for which there is an easily computed predicate on partial solutions that can be used to prune some of those partial solutions that cannot be extended to full solutions. For some interesting examples, see the above references.

6.1.2 Forward Checking

Forward checking is a simple idea that can often speed up a backtracking search [49]; it has also been referred to as "preclusion" [8, 46]. Consider the operation of a standard backtracking algorithm on a binary CSP. Suppose at some point, the variable v has just been assigned a value, but there are other variables that are still unassigned; we will refer to these unassigned variables as "future variables." Standard backtracking will not check the consistency of the new value of v with the values for any future variable until this future variable is assigned a value in the course of the search. Standard backtracking, then, does what one might call backward checking.

Forward checking, on the other hand, does this constraint checking in advance. When v is assigned a value, forward checking removes from the domains of the future variables any values that are inconsistent with the current value of v. If the forward

[1]This is according to [8, 46]; some other sources [17, 55] seem to credit Walker [84], but this might be for the term "backtracking" rather than "backtrack."

checking operation removes all the values from the domain of any future variable, then the search algorithm will backtrack immediately since the current partial assignment must be a dead end. Note that by the time that a variable is processed, its domain will include only those values that are consistent with all the past variables. We have explained forward checking in terms of a binary CSP, but the idea is obviously more general.

There are several perspectives that one might take on forward checking. First, it can eliminate many repetitive consistency checks; in this respect, it is similar to the backmarking idea, which we we will discuss in Section 6.1.6.

Second, it can cause dead-end branches to fail faster. Suppose that we have assigned values to the first 10 variables, and that based on these values, there is no consistent assignment for the 20th variable. Forward checking would fail immediately, while standard backtracking would fail for each combination of values of the intervening variables. Thus, forward checking prunes the search space in a manner analogous to that of backjumping (actually, it prunes the search space in the same way as Gaschnig's original backjumping — see the footnote on page 9).

Third, forward checking can be understood as a simple variable-ordering heuristic. If all of the values for some variable have been eliminated, forward checking, in a sense, chooses to branch on that variable next so that it can immediately backtrack. The more sophisticated variable-ordering heuristics that we used for our experiments in Chapters 2 and 3 included this feature as a special case (see the remarks in the next subsection). The theoretical results in Chapter 5 allowed for an optimal variable order, so they would not be affected by forward checking.

Finally, one can also view forward checking as a primitive constraint propagation algorithm. We will discuss some more sophisticated algorithms of this type in Section 6.2 below.

6.1.3 Heuristics

The order in which variables and values are selected can greatly affect the efficiency of a backtracking search, so there has been much interest in developing heuristics for making these decisions wisely. We will briefly discuss a few of these heuristics.

One important variable-selection heuristic is to select at each point the variable with the fewest number of remaining choices [8]. (Since a variable with zero remaining choices will be most preferred, this heuristic subsumes forward checking.) This principle has been referred to by a number of names, including *cheapest first*, *most-constrained first*, *search rearrangement*, and *dynamic search rearrangement*. The utility of this technique has been verified both theoretically [49, 68] and experimentally [8, 43, 69, 80].

Since the goal is to make the ultimate search tree as small as possible, it makes sense to take into account not only how constrained a variable is, but also how *constraining* it is. One idea in this connection is to prefer variables that participate in as many constraints as possible, or perhaps variables that most restrict the domains of the future variables; one might even look ahead several levels in the search tree in order to make this decision. A number of authors have explored variants of these ideas [23, 49, 56, 69, 87].

The graph coloring experiments in Chapter 2 used the Brélaz heuristic [9, 82], which employs both of the above ideas since it prefers vertices with the fewest number of remaining colors, and breaks ties by preferring vertices with the most neighbors. The satisfiability experiments in Chapter 3 also used a heuristic of this general form; that heuristic preferred propositional variables whose values could be deduced, and to break ties it preferred variables that appeared in the most binary clauses. The examples in Chapter 5 suggest a divide-and-conquer heuristic in which variables are chosen with the goal of breaking the problem into smaller pieces. It might be interesting to explore this idea further.

Once the variable has been selected, one has to decide in which order to try out its different values. Here, the conventional wisdom is that one wants to select first the value that is *least* constraining. This maximizes the chance of finding a solution without backtracking (of course, if the problem is unsatisfiable, it will not make any difference one way or the other). Again, there are a number of variants of the basic idea [30, 43, 49, 53, 78].

6.1.4 Davis-Putnam Procedure

A propositional satisfiability problem is a type of constraint satisfaction problem in which the problem variables only take the values TRUE and FALSE, and the constraints are logical formulas of these variables. Usually, the constraints are assumed to be in clausal form. That is, each constraint is a disjunction of literals, where a literal is either a variable or the negation of a variable.

The well-known *Davis-Putnam procedure* for solving propositional satisfiability problems is just a backtracking algorithm, together with a variant of forward checking known as *unit propagation* [26]. Unit propagation says that if in a given clause, all the literals but one have been assigned the value FALSE,[2] then the remaining literal should be assigned the value TRUE. Since a clause is just a disjunction, this rule of inference is obviously sound. The unit propagation rule is executed repeatedly until it can no longer be applied. Since one unit propagation can trigger another, unit propagation is a more powerful constraint propagation algorithm than forward checking. Alternatively, one can view unit propagation as forward checking together with the cheapest-first heuristic; this heuristic takes a particularly simple form here since the non-empty domains must have sizes of either 1 or 2. Once unit propagation has run to quiescence, the search algorithm needs to select a new variable, and all the usual ideas about variable-selection heuristics apply; see for example [23, 87]. The backtracking algorithm used in the Chapter 3 experiments was a version of the Davis-Putnam procedure.

Let us record two historical notes on the Davis-Putnam procedure. First, the original purpose of the procedure was to prove unsatisfiability of formulas in first-order logic [26, 27]. Since it worked by generating successive ground instances of the first-order formula and then applying the propositional method described above, it was not as efficient for this purpose as the resolution method later proposed by Robinson [71]. Second, what we (and most others) call the "Davis-Putnam procedure" is actually from a paper by Davis, Logemann, and Loveland [26]. An earlier paper by Davis and Putnam [27] described a somewhat different method that is generally not as useful

[2]If a variable v has been assigned the value TRUE, then its negation $\neg v$ is assumed to have the value FALSE, and vice versa.

in practice (although Dechter and Rish [31] argue that there are some problems for which this earlier method is better).

6.1.5 Intelligent Backtracking

Intelligent backtracking procedures are those that determine the reasons for a dead end in order to prune analogous nodes from the search tree. We have already discussed at length the major algorithms of this type. In particular, we have covered backjumping [39, 40], dependency-directed backtracking [79], k-order learning [28], and dynamic backtracking [42]. Let us briefly consider a few additional variants.

Dechter [28] discusses graph-based versions of the various intelligent backtracking routines. The idea is as follows. Suppose some variable v cannot be assigned a value consistent with the previously assigned variables. Normally, we would identify the conflict set of variables that were used in excluding the various values of v. All of the variables in this set must be neighbors of v in the constraint graph since these are the only variables that share constraints with v. The graph-based intelligent backtracking algorithms simply set their conflict set to be *all* the neighbors of v that have been assigned values; this might be a strict superset of the conflict set as determined by the usual dynamic dependency analysis. The advantage of the graph-based approach is that it avoids the extra overhead at each constraint check. The disadvantage is that it may prune fewer nodes from the search tree. The trade-off between these two considerations is essentially an empirical matter.

Bruynooghe has proposed an algorithm called *intelligent backtracking* [11].[3] Bruynooghe's intelligent backtracking is quite similar to dynamic backtracking, with an important difference.

In order to explain this difference, let us review one aspect of dynamic backtracking. Given a conflict set, dynamic backtracking always chooses to backtrack to the variable in this set that was most recently assigned a value. We can define a partial order as follows: for all variables v and w and values x, if $w \in culprits[v, x]$, then

[3]This may be confusing, since we are also using "intelligent backtracking" as a generic term here. To avoid ambiguity, one might refer to this algorithm as "Bruynooghe's intelligent backtracking."

$w < v$. Given the nature of dynamic backtracking, it is clear that this is in fact a partial order, and that the variable to which dynamic backtracking chooses to backtrack will always be a maximal element in this partial order.

Bruynooghe, on the other hand, allows his algorithm to backtrack to *any* variable that is maximal in the partial order. Unfortunately, it is then possible for Bruynooghe's intelligent backtracking procedure to cycle endlessly and never terminate. It is not clear whether Bruynooghe realized this or not, but in any case, his paper [11] presents no termination proof, and it is not hard to construct a counterexample, demonstrating that his procedure does not always terminate.[4] Intelligent backtracking has also been of some interest in the logic programming community [10, 22].

Ginsberg and McAllester have recently introduced *partial-order dynamic backtracking* [45]. This new algorithm allows more backtracking freedom than the original dynamic backtracking algorithm, but less than Bruynooghe's algorithm, and it is guaranteed to terminate.

6.1.6 Backmarking

We will now discuss *backmarking*, an idea due to Gaschnig [38, 40]. Backmarking differs from the above intelligent backtracking routines in that it does not actually prune the search space; it only eliminates redundant consistency checks. Assume that we are searching using a static variable ordering $v_1, v_2, \ldots, v_n$, and that there is a binary constraint (possibly trivial) between each pair of variables. Assume further that we always do these constraint checks using that same static ordering. That is, when we want to assign some value x to some variable (say) v_{20}, we would (with standard backtracking) check the consistency of $v_{20} = x$ with the current assignments for the variables v_1 through v_{19} respectively, stopping at the first constraint violation; if there is no constraint violation, we would proceed to set v_{20} to x.

Backmarking rests on the observation that many of these consistency checks may

[4]Consider a problem with the Boolean variables, A, B, C, and D, such that any values for any two of A, B, C will rule out both values of D. A very unlucky implementation of Bruynooghe's intelligent backtracking might learn $A \Rightarrow B$, and then learn $C \Rightarrow \neg A$ (forgetting the previous nogood), and then learn $B \Rightarrow \neg C$ (forgetting the previous nogood), etc. This example is based on a personal communication from Ginsberg.

have been performed earlier, and would not need to be repeated if we had done some simple bookkeeping. Let us explain this principle by example. Suppose the last time we checked the consistency of $v_{20} = x$, the constraint check first failed with v_{10}. That is, $v_{20} = x$ was consistent with the assignments for the variables v_1 through v_9, but inconsistent with v_{10}. Now, after some backtracking, we have returned to v_{20} again. If in the interim, we have only backtracked as far as (say) v_{12}, then we can reject the assignment of x to v_{20} without checking any constraints; since $12 > 10$, we have not yet changed the value of v_{10}, and the same constraint would fail again. On the other hand, suppose we had backtracked as far as v_7. Then, we might have changed the value of v_{10}, so it is possible that x would be a consistent assignment for v_{20}. Therefore, we would check the constraints of the variables starting with v_7. There is no need to check with the variables v_1 through v_6 because the constraints associated with these variables were satisfied last time, and the values have not yet been changed.

The actual bookkeeping uses two arrays, a one-dimensional array *new* and a two-dimensional array *mark*. For each variable v, the entry $new[v]$ records how far the procedure has backtracked since the last time that v was visited; and for each variable v and value x, $mark[v, x]$ records how far the constraint checking got the last time that $v = x$ was considered.

From our standpoint, backmarking is not all that interesting. As we have already said, it does not prune the actual search space. It may reduce the average time spent per partial assignment, but it does not change the number of partial assignments considered. Therefore, all of the results in this thesis would stand unchanged even if the various algorithms were augmented with some form of backmarking. Furthermore, backmarking depends on the variables being ordered statically; it would have to do more bookkeeping to handle dynamic variable orders. Finally, backmarking is most useful on those problems where every variable shares a constraint with every other variable. In many common problems, however, each variable is involved in only a few constraints, and thus the whole problem that backmarking was invented to solve does not even arise.

6.1.7 Iterative Broadening

Iterative broadening is a general search idea due to Ginsberg and Harvey [44] that is often useful for constraint satisfaction problems. Consider a tree with branching factor b and depth d. In other words, there is one root node at level 0, and each node at level i where $0 \leq i < d$ has b children at level $i+1$. Suppose some subset of the leaf nodes are "solution nodes," and our task is to find a solution or to show that no solution exists. Iterative broadening is an alternative to the standard depth-first strategy for exploring this tree.

With iterative broadening, we first search the tree using an artificial breadth limit of 2. That is, we only consider 2 children of any node (probably those 2 children that our heuristics deem most promising). After backtracking to a given node 2 times, we would ignore its other children, and backtrack from that node to its parent. This amounts to searching a subtree of the original tree containing only 2^d rather than b^d leaf nodes. If this search fails to find a solution, then the process is repeated, but with the breadth cutoff set to 3. If *this* search fails, the breadth cutoff is incremented again, and so on, until either an answer is found, or the until the original tree has been fully searched; for this last search, the breadth cutoff will have to be b, the branching factor of the original tree. Needless to say, the fixed branching factor b and fixed depth d were merely for illustrative purposes; iterative broadening will work just as well for the less regular search trees that arise in (for example) constraint satisfaction problems.

In the worst case, iterative broadening has to eventually search the entire tree, and for unsatisfiable problems, this worst case is unavoidable. Therefore, the worst-case results in this dissertation would still hold even if the search algorithms made use of iterative broadening. Nonetheless, iterative broadening is often a good idea in practice.

The reason that it is a good idea in practice is that often the solutions to a CSP are "clumped." That is, the CSP may have many solutions, but these solutions might not be randomly distributed among the leaf nodes. Perhaps the very first variable selected has some values that lead to no solutions, but other values that easily lead to many solutions. If a standard backtracking algorithm chooses a bad value for this first variable, then it might waste a lot of time searching a huge portion of the space that is devoid of solutions. It cannot revise its value for the first variable until it has exhaustively explored this entire subspace. Iterative broadening, on the other hand, is able to return to the first variable after performing only a partial search of the subspace. In effect, it is determining that this subproblem is "difficult," and it is electing to perform a partial search elsewhere before returning to a more exhaustive search of the original subspace.

Here is another way of looking at the situation. If the solutions are randomly distributed among the leaf nodes, then any order for exploring the leaf nodes is as good as any other. In particular, depth-first search would be just fine. If the solutions are not randomly distributed, however, then we would like the search algorithm to be as "random" as possible. Iterative broadening performs a fairer sampling of the leaf nodes than does depth-first search. Another possibility would be a true stochastic search that actually travels random paths down the tree. This is called *iterative sampling*, and we will discuss this idea later.

6.2 Constraint Propagation

We have mentioned several examples of *constraint propagation* algorithms; these are also known as *local consistency* algorithms. Freuder's algorithm for solving width-1 CSPs in linear time uses a preprocessing routine that ensures directional arc consistency [34], while Dechter and Pearl's adaptive consistency algorithm [30] achieves a higher level of directional consistency. Forward checking [49] and unit propagation [26] are local consistency algorithms that are usually interleaved with a backtracking search. The purpose of all these algorithms is to simplify a CSP by removing impossible values, so that a backtracking algorithm will waste less time thrashing. Let us look at some other constraint propagation algorithms in more detail.

Consider a CSP with the variables ordered $v_1, v_2, \ldots, v_n$. Directional arc consistency ensures that if there is a binary constraint between the variables v_i and v_j where $i < j$, then for every value in the domain of v_i, there will be some value in the domain of v_j that is consistent with it. Full arc consistency requires that this condition holds, even when $i > j$. Directional arc consistency can be achieved in a single pass, and it is very useful if you plan to do a backtracking search using a known static variable order. Arc consistency, however, is much more interesting as a general preprocessing routine. Arc consistency and its generalizations to higher levels of consistency have been the subject of much research.

6.2.1 Survey

One of the earliest papers on arc consistency was by Waltz [85]. He presented an arc consistency algorithm ("Waltz filtering") in the context of a program that interpreted line drawings with shadows. Mackworth [58] discusses three different arc consistency algorithms, AC-1, AC-2, and AC-3. Let us look at these algorithms in some detail.

All of the Mackworth algorithms make use of a REVISE procedure. This procedure takes as arguments two CSP variables, v and w, and it removes from the domain of v any values that are inconsistent with every value in the domain of w. More formally, let D_v be the current domain of v, let D_w be the current domain of w, and let P be the constraint predicate that holds between v and w. Then REVISE updates D_v as follows:

$$D_v := \{x \in D_v : \exists y \in D_w \text{ such that } P(x,y)\}.$$

In a single pass over the problem, AC-1 applies REVISE to every ordered pair of variables that share a binary constraint. A single pass, however, is insufficient to achieve full arc consistency. If any values were deleted from any of the domains during the last pass, then AC-1 has to make another iteration over the problem.

Let n be the number of variables, e be the number of constraints, and a be the maximum domain size. Then AC-1 performs at most $P(na)$ iterations. Each invocation of REVISE takes $O(a^2)$ time, and thus each iteration takes $O(ea^2)$ time. Therefore, the total time complexity of AC-1 is (nea^3). (Since e is $O(n^2)$, this bound is also (n^3a^3), but the earlier bound may be tighter if the constraint graph is sparse.)

AC-1 is rather inefficient. Just because a value has been removed from the domain of a single variable, there is really no reason to process every constraint again. One need only process those constraints involving that variable. Both AC-2 and AC-3 maintain queues containing the edges that need to be processed. The algorithms differ on the details of the queue; AC-3 is simpler, while AC-2 is intended as a reconstruction of Waltz's original algorithm. We will discuss AC-3. The queue is initialized to contain all ordered pairs of variables that share constraints (so if there is a constraint between v and w, both (v, w) and (w, v) will be placed on the queue). Then, as long as the queue is nonempty, AC-3 will remove some arc (v, w) from the queue, and call REVISE on this pair of variables. If this causes any values to be removed from the domain of v, then AC-3 adds to the queue any edges of the form (u, v) where $u \neq w$. When the queue is finally empty, full arc consistency will have been achieved. The maximum number of times that an arc (u, v) can be processed is $a + 1$, since it is on the queue initially, and it will be added to the queue only when a value has been deleted from the domain of v. The time to process a constraint a single time remains $O(a^2)$, and there are only $2e$ directed edges (since each constraint is considered in both orders). Therefore, the time complexity of AC-3 is $O(ea^3)$, which is better than the $O(nea^3)$ of AC-1.

Consider a constraint satisfaction problem that is arc consistent. The problem might still be unsatisfiable, and even if the problem has a solution, it still might be difficult to discover this solution. Therefore, it is sometimes useful to achieve a higher level of local consistency than arc consistency.

For some CSP, consider $k - 1$ variables $v_1, v_2, \ldots, v_{k-1}$, and suppose we have an assignment of values to these variables such that this assignment satisfies all the constraints among these $k - 1$ variables. Now consider some other variable v_k. It might be impossible to extend this partial solution to value this additional variable without violating any constraints. If it is always possible to extend a solution in this fashion, then we will say, following Freuder, that the problem is *k-consistent* [33]. If the problem is k'-consistent for all k' such that $1 \leq k' \leq k$, then the problem is *strongly k-consistent*. Arc consistency is synonymous with strong 2-consistency. The special case of strong 3-consistency is also of interest, and it is known as *path consistency* [58, 65]. The concept of path consistency was introduced by Montanari [65], and Mackworth [58] discusses two algorithms for path consistency, PC-1 and PC-2, having complexities $O(n^5a^5)$ and $O(n^3a^5)$ respectively. Mackworth and Freuder [60] discuss the complexities of the various algorithms from [58] in more detail.

Mohr and Henderson [64] point out that Mackworth's AC-3, with its time complexity of $O(ea^3)$ is not optimal, and they present an improved arc consistency algorithm, AC-4, having a time complexity $O(ea^2)$, at the cost of using more space. They also propose a new path consistency algorithm, PC-3, which turned out to be unsound; Han and Lee [48] point this out, and present a working algorithm PC-4. This path consistency algorithm has a time and space complexity of $O(n^3a^3)$. Cooper [19] extends these ideas to produce an optimal k-consistency algorithm.

Two other arc consistency papers deserve comment. Hentenryck, Deville, and Teng [50] present AC-5, a "generic" arc-consistency algorithm, which can be instantiated to produce either AC-3 or AC-4. AC-5 is particularly useful when the constraints take a particular form, for example, functional dependencies; AC-5 can then avoid the overhead that would be involved in treating the constraints as arbitrary predicates.

Finally, Bessière [7] discusses AC-6, a compromise between AC-3 and AC-4. It keeps the optimal worst-case time complexity of AC-4, while avoiding AC-4's extra space requirements. In the average case, AC-6 seems to outperform both AC-3 and AC-4, at least according to the data presented in [7].

6.2.2 Discussion

From the perspective of this dissertation, the important observation is that all of these local consistency procedures are performing resolution. Consider, for example, the basic operation of arc consistency. Suppose that there is a binary constraint between the variables u and v, and that both variables have a domain of $\{1, 2, 3\}$; further suppose that for every value of u, the constraint prohibits v from being 2. Then we can resolve together the following clauses (which follow directly from the constraint and hence are primitive in the sense of Definition 5.10)

$$(u \neq 1) \vee (v \neq 2),$$
$$(u \neq 2) \vee (v \neq 2),$$
$$(u \neq 3) \vee (v \neq 2)$$

to conclude

$$v \neq 2. \tag{6.1}$$

If we had some other variable w, with the constraints

$$(v \neq 1) \vee (w \neq 1),$$
$$(v \neq 3) \vee (w \neq 1),$$

then we could resolve these clauses with (6.1) to conclude that $w \neq 1$. Arc consistency is simply a sequence of these resolution steps. The same principle applies to path consistency and to higher levels of k-consistency.

Therefore, Conjecture 5.39 (if true) would ensure that we cannot use any such local consistency procedure to construct a polynomial-space CSP algorithm that is exponential only in the induced width. This would be true regardless of whether we first preprocessed the CSP using the constraint propagation procedure and then performed backtracking, or if we instead interleaved the two operations.

6.3 Nonsystematic Methods

This thesis has been limited to *systematic* algorithms. A systematic CSP algorithm is guaranteed to find a solution if one exists and otherwise prove that the problem is unsatisfiable. Many problems, however, are so large and so difficult that this guarantee is of only theoretical interest. Suppose we have (say) 10 hours of computation time available, but the search space is so huge that even the best of the known systematic algorithms would take centuries to search it exhaustively. If the problem is unsatisfiable, we will not be able to prove it so within the 10-hour time limit, and even if there are solutions, we will not necessarily find one since there is only enough time to examine a small subset of the possibilities. It therefore makes sense to consider algorithms that are good at exploring the portions of the search space most likely to contain solutions, even when these algorithms are not systematic. The *local search* procedures are one such category of algorithms.

6.3.1 Local Search

Local search procedures work in a completely different way from the algorithms that we have discussed so far. Such a procedure typically starts by assigning random values to all of the variables. Unless this initial assignment is a solution, it will violate some (nonzero) number of constraints. The local search procedure then incrementally modifies the current assignment by changing the value of one variable at a time in an attempt to reduce the number of violated constraints. The goal is to eventually reduce the number of violated constraints to zero.

These local search procedures are similar to the numerical optimization routines that try to maximize a function by following local gradients. Just as with these numerical procedures, the CSP local search procedures can get stuck at local minima. Therefore, they often have provisions for making random uphill moves or restarting. The local search procedures have no guarantee of finding a solution in any given amount of time, and of course they cannot prove unsatisfiability. On the other hand, they can sometimes solve problems that are too difficult for any known systematic method. We will discuss two local search procedures, *Min-Conflicts* [62] and *GSAT* [77].

Min-Conflicts is a local search procedure that was developed by Minton and his colleagues [62]. The hill-climbing step of Min-Conflicts works as follows. It first identifies the set of variables involved in conflicts, that is, the set of variables that participate in violated constraints. It then selects a random variable from this set. Finally, it assigns this variable a new value that minimizes the number of other variables that are inconsistent with it. This process is repeated until a solution is found. Min-Conflicts has been used successfully to solve large-scale scheduling problems [62]. It has also been used to solve instances of the n-queens problem, the problem of placing n queens on an n by n chessboard so that no two queens attack each other; Min-Conflicts can solve this problem for $n = 1,000,000$ in a few minutes [62], while the best backtracking methods have trouble for even $n = 1000$.

GSAT is a local search procedure specifically designed for propositional problems by Selman and his colleagues [77]. It assumes that the problem is presented in clausal form. GSAT begins by generating a random truth assignment for the problem variables. Then for each variable, GSAT considers the effect of "flipping" this variable, that is, of changing its value from TRUE to FALSE, or vice versa. GSAT flips the variable that give the greatest increase in the total number of satisfied clauses, with ties being broken randomly. This iterative improvement step is repeated a maximum of *Max-Flips* times, where *Max-Flips* is a parameter provided to GSAT. If no solution has been found after *Max-Flips* flips, GSAT starts over with a new randomly generated truth assignment. GSAT repeats this entire process a maximum of *Max-Tries* times (where *Max-Tries* is another parameter), after which it gives up, returning "no solution found" as its answer. Obviously, Minton's Min-Conflicts algorithms could also be embedded in an outer loop of this kind.

Selman and Kautz have made a number of interesting improvements to the basic GSAT algorithm [74, 75, 76]. One new feature associates a "weight" with each clause. When the version of GSAT with weights computes the utility of flipping a variable, it uses the sum of the weights of the satisfied clauses as its criterion, instead of simply the number of such clauses. The weights are all initialized to 1, but clauses that are often unsatisfied will have their weights increased gradually. Another improvement allows more randomness in order to escape from local minima. With probability p, this version chooses a random variable in any unsatisfied clause instead of choosing the variable using the usual greedy heuristic. This has some of the flavor of simulated annealing [54], but using a fixed "temperature" (determined by p) instead of an annealing schedule.

GSAT has also been extended to handle non-clausal formulas [72]. In other words, instead of being given a conjunction of disjunctions of literals, this version is given an arbitrary propositional formula. It computes the number of clauses that would be violated if this formula were converted to clausal form. This calculation can be done efficiently even when the conversion itself would cause an exponential blowup.

6.3.2 Iterative Sampling

Local search is not the only stochastic approach to constraint satisfaction problems. *Iterative sampling* is another idea [57]. Iterative sampling is a tree search algorithm that travels random paths down the tree. That is, it starts at the root, and then moves to a random child of this node. It continues to move to a random child of its current node until it reaches a goal node or a dead end. If iterative sampling hits a dead end, it starts over again at the top of the tree. In the context of a CSP, the root node is the empty assignment, and the children of a given node correspond to the partial solution being extended by assigning the various values to the next variable. Iterative sampling might also do some type of forward checking or constraint propagation. In this case, the children will correspond to only those values that are consistent with the prior values, modulo the constraint propagation algorithm.

Iterative sampling can be a good idea for the same reason that iterative broadening (see Section 6.1.7 above) can be a good idea. If the solutions to a CSP are clumped, then standard backtracking risks wasting its time exploring a portion of the search space devoid of solutions, even when there are many solutions elsewhere. Iterative sampling avoids this problem by randomly sampling the possible solutions. On one type of scheduling problem, iterative sampling has been shown to outperform both backtracking and GSAT [24].

6.4 Tractable Classes of CSPs

In Chapters 4 and 5 of this dissertation, we have analyzed CSP algorithms with respect to one particular measure of problem difficulty, the induced width. CSPs with a fixed induced width can be solved in polynomial time provided that the algorithm is willing to let its memory usage be exponential in this induced width. In this section we discuss some other tractable CSPs. We divide them into three broad categories: restrictions on the constraint graph, restrictions on the constraint predicates, and more complicated restrictions.

6.4.1 Tractable Constraint Graphs

There are other interesting parameters of a graph besides its induced width. One such parameter is the *bandwidth* [14]. Consider a graph $G = (V, E)$ with an ordering

h, where V is the set of vertices, E is the set of edges and h is a bijection between V and the set $\{1, 2, \ldots, n = |V|\}$. The bandwidth of a vertex $v \in V$ under the ordering h is the maximum value of $|h(v) - h(w)|$ for any edge $\{v, w\} \in E$. In other words, the bandwidth of a vertex is the maximum distance between that vertex and any other vertex that is connected to it, where distance is defined in terms of the ordering h. The bandwidth of the graph under an ordering is the maximum bandwidth of any of its vertices under that ordering. The bandwidth of a graph is its minimum bandwidth under any ordering.

Zabih [86] discusses the bandwidth in the context of constraint satisfaction problems. Let $B(G, h)$ be the bandwidth of a graph G under the ordering h, and let $W(G, h)$ be the induced width of the graph under that ordering. Zabih [86] notes that for any graph G and ordering h,

$$W(G, h) \leq B(G, h). \tag{6.2}$$

To see this, consider some arbitrary vertex v. If w is a parent of v (that is, a vertex connected to v that precedes v under the ordering h), then by the definition of the bandwidth, we must have $|h(v) - h(w)| \leq B(G, h)$. Since w is a parent of v, we must therefore have

$$h(v) - B(G, h) \leq h(w) < h(v).$$

Since the edges that are added when constructing the induced graph will only connect the parents of some vertex, it follows by an inductive argument, that the induced graph must also have this property. Therefore, v can have at most $B(G, h)$ parents in the induced graph, establishing (6.2). Since the induced width cannot exceed the bandwidth, a problem class with a limited bandwidth will also have a limited induced width, and will therefore be tractable.

Seidel [73] defined the notion of the *front length*. Consider a graph $G = (V, E)$ and ordering h, and let v be some vertex in the graph. The front length of v under the ordering is the cardinality of the set

$$\{w : h(w) < h(v), \exists x \in V, h(v) \leq h(x), \{w, x\} \in E\}. \tag{6.3}$$

This set consists of all the vertices before v (under the ordering) that are connected to vertices after or equal to v in the ordering. The front length of G under the ordering

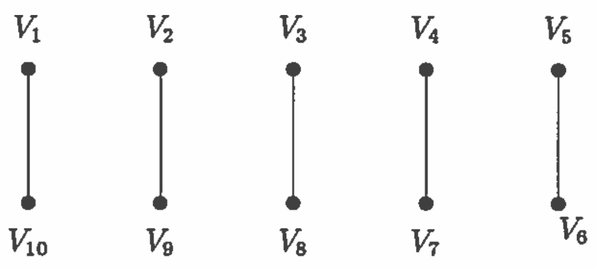

Figure 6.1: Bandwidth vs. front length vs. induced width

h, which we will write as $F(G,h)$, is the maximum front length of any of its vertices under the ordering,[5] and the front length of the graph is its minimum front length under any ordering.

Seidel [73] introduced these concepts in order to present a CSP dynamic programming algorithm similar in some respects to Dechter and Pearl's algorithm. For our purposes, we need only note that

$$W(G,h) \leq F(G,h) \leq B(G,h). \tag{6.4}$$

Zabih [87] proves the second part of this inequality, but both parts are straightforward. Consider the set (6.3) for some vertex v in the graph. For any w in this set, we must have that $h(v) - B(G,h) \leq h(w) < h(v)$ since w is connected to some vertex x, where $h(x) \geq h(v)$. Thus the size of the set (6.3) is bounded by $B(G,h)$; this establishes the second part of (6.4). To prove the first part, note that every parent of v in the induced graph must either be a parent of v in the original graph, or the parent of some vertex x that follows v in the original graph (that is, $h(x) > h(v)$). But all vertices of this type will be in the set (6.3); this establishes the first part.

To see that the inequalities of (6.4) can sometimes be strict, consider the example in Figure 6.1, where, as indicated, the variables are ordered clockwise starting from the upper left. The induced width under this ordering is 1, the front length is 5, and the bandwidth is 9. On the other hand, for a completely connected graph with n vertices, all three parameters would be $n-1$.

Because the induced width is bounded by the front length and the bandwidth, the

[5]Actually, Seidel [73] refers to an ordering as an "invasion," the set (6.3) as a "front" in this invasion, and the front length under the ordering as the "front length of the invasion."

latter two measures also define tractable classes of CSPs; but since the induced width can be strictly less than the other two, it is the more informative parameter. Figure 6.1 illustrated this point, but since it is based on a suboptimal ordering, perhaps another example would be useful. Consider a CSP whose constraint graph is a tree. The induced width is 1, but the bandwidth and front length are not bounded. Therefore, the induced width identifies more tractable CSPs than the other two measures.

How do the bandwidth and the front length relate to the theoretical results in this thesis? Note that the counterexamples of Chapters 4 and 5 have bandwidths identical to their induced widths. That is, while it is possible in principle for the induced width to be bounded while the bandwidth diverges, we did not in practice take advantage of this freedom. Therefore, we have really proven stronger results that we have claimed. We have really shown, for example, that there is no polynomial-space dependency-directed backtracking procedure that is exponential only in the *bandwidth*.

There is another graph parameter that is sometimes of interest, the *cycle-cutset size* [28]. We know that CSPs with tree-structured constraint graphs can be solved in linear time. Suppose that a constraint graph is not a tree, but that it can be made so by deleting a few of its vertices (along with the edges incident on these vertices). We will call this set of vertices a *cycle cutset*, or simply a *cutset* for short. The CSP can be solved efficiently by first assigning consistent values to the variables in the cycle cutset, and then applying the tree algorithm to the rest of the problem; this process might have to be repeated for each combination of values for these variables, or a total of a^c times altogether, where a is the maximum domain size, and c is the size of the cutset. We will say that the cycle-cutset size of a graph is the size of its smallest cutset (it is NP-hard to compute this parameter).

How does the cycle-cutset size c relate to the induced width? If we order the vertices in the cutset first, followed by the other vertices in some tree-like order, then the induced width under this ordering cannot exceed $c + 1$ since each vertex has at most one parent in the tree, and at most c parents in the cutset. Therefore, the induced width is at most $c + 1$, and we see that the induced width is at least as informative (in terms of identifying tractable problems) as the cycle-cutset size. On the other hand, the cycle-cutset size is not necessarily bounded by the induced width. In fact, it is not even bounded by the bandwidth. In some of the counterexamples of Chapters 4 and 5, the bandwidth is fixed, but the cycle-cutset size grows linearly with the problem. Therefore, our theorems do not apply to the cycle-cutset size; it is quite possible for a polynomial-space CSP algorithm to be exponential only in the size of the cutset (we have just sketched such an algorithm in the last paragraph).

6.4.2 Tractable Constraints

So far, we have examined only those tractable CSPs that can be characterized based purely on the topology of the constraint graph, irrespective of the actual constraint predicates. We will now consider the opposite extreme, namely, conditions on the constraint predicates sufficient to ensure tractability.

One simple example is 2-SATISFIABILITY (also know as 2-SAT, or 2-CNF); this was mentioned briefly in Chapter 1. In this example, all of the variables are propositional (that is, they take only two values, conventionally TRUE and FALSE). The constraints are all binary clauses; equivalently, we can let the constraints be arbitrary binary constraints between these propositional variables since any constraint can be represented by at most 4 binary clauses. It is well known that 2-SAT can be solved in linear time [4].

Cooper *et al.* [20] generalize this idea to an arbitrary binary CSP that is not necessarily propositional. Consider two variables, v and w, that have a constraint between them, with P being the actual constraint relation. Let D_v be the domain of v, and let D_w be the domain of w. For each value $x \in D_v$, let $h(x)$ be the values from D_w that are consistent with x, that is, the set

$$\{y \in D_w : (x, y) \in P\}.$$

We say that the constraint from v to w is a *directed 0/1/all constraint* if for all $x \in D_v$, $h(x)$ is either the empty set, or a one-element set, or the entire set D_w. If the same condition also applies with the role of w and v reverse, then we will say that the constraint is a *0/1/all constraint.* If every constraint is a 0/1/all constraint, then we will say that the CSP is a *0/1/all problem*. Cooper *et al.* [20] show that any 0/1/all problem can be solved in polynomial time.

Consider the effect of assigning a value x to some variable v in a 0/1/all problem. What will this let us conclude about the value of some other variable w? There are only three possibilities. If $h(v) = \emptyset$, then we have derived a contradiction. If $h(v)$ is a singleton, then we can propagate this value to the variable w. Finally, if $h(v)$ is just the set D_w, then we have gained no information at all. The algorithm from [20] works as follows. First, it selects an unbound variable; we will call this variable a "choice point." Then, the algorithm assigns some value to the variable and performs the constraint propagation indicated above; if this propagation leads to a contradiction, then the next value is tried for the choice point until a consistent value is found. If there is no consistent value for the choice point, then the problem must be unsatisfiable, so the algorithm can return this fact. If there is some consistent value for the choice point, the algorithm proceeds to choose a new unassigned variable as the next choice point, and it repeats the above process, and so on. The key insight of the algorithm is that if the new choice point yields a contradiction, then the problem does not have to backtrack to the previous choice point — it can simply declare the problem to be unsatisfiable. This follows from the nature of a 0/1/all problem: after a new variable assignment has been constraint-propagated to quiescence, it has no effect on the variables that remain unassigned. One can think of this algorithm as combining unit propagation with backjumping. The backjumping just takes a particularly trivial form, since if a choice point fails, the conflict set will always be empty.

The original paper [20] actually presents a slightly more general definition of a 0/1/all problem than the one that we have given here. Essentially, its definition is equivalent to assuming that the problem is 0/1/all by our definition after arc consistency has been established. One might wonder whether there are other tractable constraints beside the 0/1/all constraints. Let S be a set of binary constraints that includes some constraint that is not a 0/1/all constraint (in the expanded sense of the original paper). Cooper *et al* [20] prove that provided S is closed under certain operations (permutations of the values, restrictions on the domains, composition, and intersection), the set of binary CSPs with constraints from S is not a tractable problem class. Thus, in some sense, the set of 0/1/all problems is the largest tractable class that can be obtained by restricting the nature of the constraints.

6.4.3 More Tractable Problems

A propositional satisfiability problem is a CSP in which the variables take the values TRUE and FALSE, and the constraints are clauses. Each clause is a disjunction of literals, where each literal is either a propositional variable or its negation. A literal that is simply a variable will be said to be *positive*, and a literal that is the negation of a variable will be said to be *negative*. A *Horn clause* is a clause that contains at most one positive literal. For example, the clause

$$\neg p \vee q \vee \neg r \tag{6.5}$$

is a Horn clause, while

$$p \vee q \vee \neg r$$

is not a Horn clause. It is often convenient to write Horn clauses in a directed form. For example, (6.5) is equivalent to the rule

$$p \wedge r \Rightarrow q. \tag{6.6}$$

It is well known that satisfiability problems in which all the constraints are Horn clauses are tractable, and can in fact be solved in linear time [32]. Furthermore, the algorithm that accomplishes this is quite simple; it just performs unit propagation starting from the original set of clauses. That is, if there is any clause that is simply a single positive literal, then the variable of that clause is set to TRUE. Then, if there is any rule whose precondition variables have all been set TRUE, then this rule's conclusion is set TRUE; for example if p and r had been set TRUE in (6.6), then q would be set to TRUE. This process continues until no more rules are applicable. Now, there might be a clause consisting entirely of negative literals. In the directed form, this clause would have an empty conclusion. If all of the precondition variables of such a clause have been set TRUE, then the original theory must be unsatisfiable.

If this never happens, the search algorithm can complete the satisfying assignment by setting all the unassigned variables to FALSE. It is not hard to see that the assignment constructed in this way must satisfy all of the clauses.

Just as one can start with the set of CSPs with tree-structure constraint graphs, which can be solved in linear time, and then define successive supersets (CSPs of induced width no more than 2,3,4,...) that are progressively more difficult, one can define a similar hierarchy starting with the Horn clauses. Let H be the set of satisfiability problems in which all the clauses are Horn; as discussed above, any such problem can be solved in linear time. Arvind and Biswas [3] identify a set of problems S_0, where S_0 is a strict superset of H, that can be handled in quadratic time. Gallo and Scutellà [36] define a sequence of classes of satisfiability problems $\Gamma_0, \Gamma_1, \Gamma_2, \ldots$ with the following properties:

$$\begin{aligned}
\Gamma_0 &= H, \\
\Gamma_1 &= S_0, \\
\Gamma_k &\subset \Gamma_{k+1} \text{ for } k \geq 0, \\
\bigcup_{k=0}^{\infty} \Gamma_k &= \text{the set of all satisfiability problems,}
\end{aligned}$$

such that any problem in Γ_k can be solved in $O(xn^k)$ time, where x is the size of the problem and n is the number of variables.

Thus, this Gallo-Scutellà parameter k is in some sense analogous to the induced width. For any fixed value of the parameter, the problem is tractable; and for all satisfiability problems, the time complexity is exponential in this parameter. Interestingly, the space complexity of the Gallo and Scutellà algorithm is also $O(xn^k)$, and hence growing exponentially in k. Besides this fact, there does not appear to be any obvious relation between k and the induced width. The induced width, of course, applies to all constraint satisfaction problems, while the Gallo-Scutellà parameter applies only to propositional problems. Furthermore, this new parameter seems less intuitive than the induced width. It is easy to visualize in terms of the pattern of connectivity among the variables why a constraint problem might have a small induced width, but it is harder to understand the Gallo-Scutellà parameter this way.

We will mention one more technique for identifying tractable problems, Dalal's idea of *intricacy* [25]. Intricacy is defined for arbitrary formulas in first-order predicate calculus, not just for constraint satisfaction problems, and it is far too complicated a notion to discuss here in detail. The intricacy has a number of interesting properties, however. When applied to CSPs, it takes into account both the topology of the constraint network and the semantics of the constraints, and it appears to subsume many of the other tractability ideas that we have discussed. Therefore, the intricacy may be able to identify more tractable problems then any of the other methods. Dalal [25] provides a decision procedure whose time and space complexities are both exponential onlyin the intricacy. Given the results of this dissertation, it is not surprising that Dalal's method is so space intensive. Since the intricacy is always less than the induced width [25], any class of problems with a bounded induced width will also have a bounded intricacy. Therefore, the induced-width results in Chapters 4 and 5 apply to intricacy as well. For example, we know from Theorem 4.20 that there cannot be a polynomial-space dependency-directed algorithm that is exponential only in the intricacy.

6.5 Proof Methods

In Chapter 5, we were able to bound the performance of certain types of CSP algorithms by making a connection with theorem proving. For unsatisfiable problems, a search algorithm must in effect construct unsatisfiability proofs. To analyze a particular type of search algorithm, we should identify the type of proof that it is constructing; if proofs of this kind are exponentially long, then the CSP algorithm will require exponential time as well. The current section puts the results from Chapter 5 in a broader context.

The complexity of different proof systems has been of some interest in the theoretical computer science community. Generally, this work has been presented in terms of propositional logic. Given a valid propositional formula (i.e., a tautology), how long is its shortest proof? Obviously, this will depend on which *proof system* is being used. Cook and Reckhow [18] define the general notion of a proof system as follows.

Definition 6.1 *A* proof system *is a function F from the set Σ^* of strings on some finite alphabet Σ onto the set of valid propositional formulas, such that F can be computed in polynomial time. If $F(w) = A$, then we will say that w is a* proof *of A in the system.*

Resolution proofs, for example, easily fit into this framework since the proof can be regarded as a sequence of symbols, and if the string w is not a legitimate proof, then we can just define $F(w)$ to be some arbitrary tautology, say $p \vee \neg p$.

One question is whether there is any proof system F such that every tautology has a polynomial-length proof in this system. This is equivalent to the question of whether NP = co-NP, one of the outstanding open problems of computer science [18]. It is known, however, that resolution is *not* such a proof system [16, 47, 83]. To prove a tautology using resolution, one converts its negation to clausal form, and then derives a contradiction via some number of resolution steps. Haken [47] used the pigeonhole problem to show that there is a sequence of unsatisfiable sets of clauses whose shortest resolution proofs are growing faster than any polynomial in the size of the clause set. Building on this work and on the work of Urquhart [83], Chvátal and Szemerédi [16] were able to prove that if one generates random unsatisfiability problems (using the appropriate distribution), almost all of these problems will require exponentially long proofs.

In this dissertation, on the other hand, we have only considered examples that have short resolution proofs (since the induced width has been bounded). We have shown that these examples no longer have short proofs when we move to a weaker proof systems such as DB-consistent resolution. Cook and Reckhow [18] also study the relationships between various proof systems. A proof system F_1 is said to be at least as powerful as system F_2 if for every formula A and proof ρ_2 in the second system (such that $F_2(\rho_2) = A$), there is some proof ρ_1 in the first system (such that $F_1(\rho_1) = A$), such that $|\rho_1| \leq Poly(|\rho_2|)$, where $Poly$ is some polynomial. In other words, the proofs in the F_1 system are just as short as those in the F_2 system, up to a polynomial. Two proof systems are equivalent in power if each is as powerful as the other. If F_1 is at least as powerful as F_2, but not vice versa, then F_1 is more powerful than F_2; this means that there are proofs using the first system that blow up exponentially upon being translated into the second system. Cook and Reckhow [18] give a partial order (in terms of power) of a number of proof systems. Tree resolution, i.e., resolution when clauses cannot be reused, is less powerful than standard resolution [81]. It is not hard to see that tree resolution basically corresponds to the proofs that are constructed by backtracking or backjumping. The Cook and Reckhow paper, however, does not consider the other cases that we have discussed of proof systems weaker than resolution, namely, k-order learning, DB-consistent resolution, and polynomial-space resolution.

Cook and Reckhow [18] also discuss several proof systems that are more powerful than resolution: Frege systems, natural deduction, Gentzen with cut, and extended resolution. From our standpoint, extended resolution is the most interesting of these. Extended resolution, which was invented by Tseitin [81], augments resolution with the ability to define new propositions. Specifically, if the proposition α does not appear anywhere in the theory, then we may add the clausal version of the formula

$$\alpha \equiv (\neg p \vee \neg q). \tag{6.7}$$

for arbitrary propositions p and q; this can be repeated any number of times as long as a new α is used each time. Since (6.7) is just a definition, it will not affect the satisfiability of the original theory. For some problems, extended resolution can be used to write proofs that are exponentially shorter than the corresponding resolution proofs. If one could prove Conjecture 5.39 concerning polynomial-space resolution, it might be interesting to then investigate the power of *polynomial-space extended resolution* (using the CSP analog of extended resolution). The type of counterexample that we have used in this dissertation, however, will be of little use in proving lower bounds for this new proof method. For instance, in Example 5.7, we could define a new variable whose value is based on whether a particular pigeon is in a given hole or not, thus abstracting away the orientations.

Chapter 7

Conclusion

7.1 Contributions

This dissertation has presented experimental and theoretical results on certain constraint satisfaction algorithms.

In Chapter 2, we studied the utility of backjumping and dependency-directed backtracking on a range of problems. We showed that these advanced search techniques were most useful on those problems with sparse constraint graphs that were the hardest for chronological backtracking. These results deepened our understanding of the distribution of hard and easy constraint problems. There was previous speculation that the underconstrained problems that were very hard for backtracking were, in some sense, fundamentally difficult problems. Hogg and Williams [51] suggested that these problems "represent the most challenging cases that Nature supplies," and Gent and Walsh [41] described these problems as being "on the knife edge between satisfiability and unsatisfiability" and perhaps even being the source of the "hardness of many NP-hard problem classes." On the contrary, we have shown that these "hard" underconstrained problems are usually quite easy for the intelligent backtracking algorithms.

Chapter 3 identified a hazard of one particular intelligent backtracking algorithm, dynamic backtracking. We showed that there can be a negative interaction between this backtracking method and the heuristics that are used to order the search. The basic problem is that when dynamic backtracking jumps back to an early choice point, it tries to preserve as many intermediate decisions as possible; but since these decisions were originally made on the basis of the earlier decision, they may no longer be appropriate in the new context. We presented a modification to dynamic backtracking that addresses this problem. This chapter was an object lesson that seemingly well-motivated search techniques can sometimes do more harm than good.

Chapters 4 and 5 proved theoretical results on the various algorithms. It was previously known that dependency-directed backtracking is exponential only in the induced width, or in other words, polynomial-time for any problem class of bounded induced width. Full dependency-directed backtracking, however, can require a huge amount of memory, so there was interest in investigating the properties of the simpler intelligent backtracking algorithms. We proved in Chapter 4 that there is no polynomial-space dependency-directed backtracking algorithm that is exponential only in the induced width, assuming that this algorithm searches using a given variable ordering. In Chapter 5, we showed that, even using an optimal search order, neither k-order learning nor dynamic backtracking is exponential only in the induced width. We also presented a more general conjecture concerning polynomial-space resolution.

The lessons of this dissertation can be summarized as follows. From Chapter 2, we learned that intelligent backtracking can be very useful on average, even when it only helps on a small number of very difficult problems. From Chapter 3, we learned that some intelligent backtracking methods can be hazardous because they interfere with the efficacy of the search heuristics; fancy backtracking is important, but so are heuristics. Finally, from Chapters 4 and 5 we learned that even though the polynomial-space methods seem to do very well in practice, they can do quite poorly in the worst case; the induced-width guarantee that one can give for full dependency-directed backtracking does not apply to the polynomial-space methods.

7.2 Future Work

There are a number of directions in which one might want to extend this work. From a theoretical perspective, the outstanding open problem is Conjecture 5.39. A proof of this conjecture would extend and unify the other results that we have presented. It would also be interesting to explore the relationship between Conjectures 5.35 and 5.36 and other open problems in computer science. Furthermore, we need a better understanding of the relationships between the various classes of tractable CSPs. For example, is there any set of constraint graphs of *unbounded* induced width that nonetheless ensures tractability for the underlying constraint problems?

The obvious project on the experimental side would be bigger and better experiments. We have looked at medium-sized graph coloring and propositional satisfiability problems. It would be interesting to see how our results change for larger problems or for other types of constraint problems. The more interesting questions, however, concern the probability distributions that were used. After fixing the number of variables and constraints for a given type of CSP, we always used the uniform distribution over the various possible problems. This was in keeping with most of the work in this field, and it was a logical place to start, but one can also consider more complicated distributions. The uniform distribution tends to generate problems with very little structure, that is, any variable is equally likely to be connected with any other variable. Alternative distributions can be used in order to generate problems with more structure. There is reason to think that intelligent backtracking will be even more useful on problems of this kind [31, 45]. Another way of looking at the situation is as follows. With a uniform distribution, a small percentage of the underconstrained problems were very difficult for backtracking; an alternative distribution might generate these same difficult problems with a much higher frequency. It would be useful to characterize this notion of "structure" more precisely, and to study its relationship with the utility of intelligent backtracking techniques.

Finally, it will be important to understand the relevance of our results for real-world constraint satisfaction problems. These problems are likely to have more structure than the uniformly-distributed random problems, so the above discussion is applicable. If we can determine what types of structure actually arise in practice, then we might be able to use this information to design even better intelligent backtracking algorithms.

Bibliography

[1] Stefan Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability — a survey. *BIT*, 25:2–23, 1985.

[2] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8:277–284, 1987.

[3] V. Arvind and S. Biswas. An $O(n^2)$ algorithm for the satisfiability problem of a subset of propositional sentences in CNF that includes all Horn sentences. *Information Processing Letters*, 24:67–69, 1987.

[4] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.

[5] Andrew B. Baker. The hazards of fancy backtracking. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 288–293, 1994.

[6] Andrew B. Baker. Intelligent backtracking on the hardest constraint problems. *Journal of Artificial Intelligence Research*, 1995. To appear.

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

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

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

[10] M. Bruynooghe and L.M. Pereira. Deduction revision by intelligent backtracking. In J.A. Campbell, editor, *Implementations of Prolog*, pages 194–215. Halsted Press, New York, 1984.

[11] Maurice Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12:36–39, 1981.

[12] Maurice Bruynooghe and Raf Venken. Backtracking. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence (2nd Edition)*, pages 84–88. Wiley, New York, 1992.

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

[14] P.Z. Chinn, J. Chvátalová, A.K. Dewdney, and N.E. Gibbs. The bandwidth problems for graphs and matrices — a survey. *Journal of Graph Theory*, 6(3):223–254, 1982.

[15] V. Chvátal and B. Reed. Mick gets some (the odds are on his side). In *Proceedings of the Thirty-Third Annual Symposium on Foundations of Computer Science*, pages 620–627, 1992.

[16] Vašek Chvátal and Endre Szemerédi. Many hard examples for resolution. *Journal of the Association for Computing Machinery*, 35:759–768, 1988.

[17] Jacques Cohen. Non-deterministic algorithms. *Computing Surveys*, 11(2):79–94, 1979.

[18] Stephen Cook and Robert Reckkhow. On the lengths of proofs in the propositional calculus. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pages 135–148, 1974.

[19] Martin C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989.

[20] Martin C. Cooper, David A. Cohen, and Peter G. Jeavons. Characterizing tractable constraints. *Artificial Intelligence*, 65:347–361, 1994.

[21] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[22] P.T. Cox. Finding backtrack points for intelligent backtracking. In J.A. Campbell, editor, *Implementations of Prolog*, pages 216–233. Halsted Press, New York, 1984.

[23] James M. Crawford and Larry 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.

[24] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1092–1097, 1994.

[25] Mukesh Dalal. Tractable deduction in knowledge representation systems. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 393–402, 1992.

[26] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[27] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[28] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[29] Rina Dechter. Constraint networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence (2nd Edition)*, pages 276–285. Wiley, New York, 1992.

[30] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[31] Rina Dechter and Irina Rish. Directional resolution: The Davis-Putnam procedure, revisited. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 134–145, 1994.

[32] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[33] Eugene Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21:958–966, 1978.

[34] Eugene Freuder. A sufficient condition for backtrack-free search. *Journal of the Association for Computing Machinery*, 29:24–32, 1982.

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

[36] Giorgio Gallo and Maria Grazia Scutellà. Polynomially solvable satisfiability problems. *Information Processing Letters*, 29:221–227, 1988.

[37] Michael R. Garey and Davis S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[38] John Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, page 457, 1977.

[39] John Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Conference of the Canadian Society for Computational Studies of Intelligence*, Toronto, 1978.

[40] John Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1979. Available as Technical Report CMU-CS-79-124.

[41] Ian P. Gent and Toby Walsh. Easy problems are sometimes hard. *Artificial Intelligence*, 70:335–345, 1994.

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

[43] Matthew L. Ginsberg, Michael Frank, Michael P. Halpin, and Mark C. Torrance. Search lessons learned from crossword puzzles. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 210–215, 1990.

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

[45] Matthew L. Ginsberg and David A. McAllester. GSAT and dynamic backtracking. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 226–237, 1994.

[46] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *Journal of the Association for Computing Machinery*, 12(4):516–524, 1965.

[47] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[48] Ching-Chih Han and Chia-Hoang Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36:125–130, 1988.

[49] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[50] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[51] Tad Hogg and Colin P. Williams. The hardest constraint problems: A double phase transition. *Artificial Intelligence*, 69:359–377, 1994.

[52] Ari K. Jónsson and Matthew L. Ginsberg. Experimenting with new systematic and nonsystematic search procedures. In *Proceedings of the AAAI Spring Symposium on AI and NP-Hard Problems*, 1993.

[53] L.V. Kale. A perfect heuristic for the n non-attacking queens problem. *Information Processing Letters*, 34(4):173–178, 1990.

[54] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1982.

[55] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–136, 1975.

[56] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–34, 1992.

[57] Pat Langley. Systematic and nonsystematic search strategies. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, pages 145–152, 1992.

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

[59] Alan K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence (2nd Edition)*, pages 285–293. Wiley, New York, 1992.

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

[61] David A. McAllester. Partial order backtracking. Unpublished manuscript, available by anonymous ftp from `ftp.ai.mit.edu:/pub/users/dam/dynamic.ps`, 1993.

[62] Steven Minton, Mark D. Johnston, Andrew B. Philips, 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.

[63] 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.

[64] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[65] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[66] Bernard A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[67] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[68] Paul Walton Purdom, Jr. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[69] Paul Walton Purdom, Jr., Cynthia A. Brown, and Edward L. Robertson. Backtracking with multi-level dynamic search rearrangment. *Acta Informatica*, 15:99–113, 1981.

[70] Igor Rivin and Ramin Zabih. An algebraic approach to constraint satisfaction problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 284–289, 1989.

[71] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

[72] Roberto Sebastiani. Applying GSAT to non-clausal formulas. *Journal of Artificial Intelligence Research*, 1:309–314, 1994.

[73] Raimund Seidel. A new method for solving constraint satisfaction problems. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 338–342, 1981.

[74] Bart Selman and Henry Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 290–295, 1993.

[75] Bart Selman and Henry A. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 46–51, 1993.

[76] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *Proceedings of the Second DIMACS Challenge Workshop on Cliques, Coloring, and Satisfiability*, Rutgers University, 1993.

[77] 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.

[78] Stephen F. Smith and Cheng-Chung Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 139–144, 1993.

[79] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[80] H.S. Stone and J.M. Stone. Efficient search techniques — an empirical study of the n-queens problem. Technical Report Technical Report RC 12057 (#54343), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1986.

[81] G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125. Consultants Bureau, New York, 1968. Translated from Russian.

[82] Jonathan S. Turner. Almost all k-colorable graphs are easy to color. *Journal of Algorithms*, 9:63–82, 1988.

[83] A. Urquhart. Hard examples for resolution. *Journal of the Association for Computing Machinery*, 34:209–219, 1987.

[84] R.J. Walker. An enumerative technique for a class of combinatorial problems. In *Combinatorial Analysis (Proceedings of Symposium in Applied Mathematics, Vol X)*, pages 91–94, Providence, Rhode Island, 1960. American Mathematical Society.

[85] D.L. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, New York, 1975. First published as MIT Technical Report AI271, 1972.

[86] Ramin Zabih. Some applications of graph bandwidth to constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 46–51, 1990.

[87] Ramin Zabih and David McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 155–160, 1988.