Handbook of Constraint Programming
Edited by F. Rossi, P. van Beek and T. Walsh

Chapter 4

Backtracking Search Algorithms

Peter van Beek

There are three main algorithmic techniques for solving constraint satisfaction problems: backtracking search, local search, and dynamic programming. In this chapter, I survey backtracking search algorithms. Algorithms based on dynamic programming [15]—sometimes referred to in the literature as variable elimination, synthesis, or inference algorithms—are the topic of Chapter 7. Local or stochastic search algorithms are the topic of Chapter 5.

An algorithm for solving a constraint satisfaction problem (CSP) can be either complete or incomplete. Complete, or systematic algorithms, come with a guarantee that a solution will be found if one exists, and can be used to show that a CSP does not have a solution and to find a provably optimal solution. Backtracking search algorithms and dynamic programming algorithms are, in general, examples of complete algorithms. Incomplete, or non-systematic algorithms, cannot be used to show a CSP does not have a solution or to find a provably optimal solution. However, such algorithms are often effective at finding a solution if one exists and can be used to find an approximation to an optimal solution. Local or stochastic search algorithms are examples of incomplete algorithms.

Of the two classes of algorithms that are complete—backtracking search and dynamic programming—backtracking search algorithms are currently the most important in practice. The drawbacks of dynamic programming approaches are that they often require an exponential amount of time and space, and they do unnecessary work by finding, or making it possible to easily generate, all solutions to a CSP. However, one rarely wishes to find all solutions to a CSP in practice. In contrast, backtracking search algorithms work on only one solution at a time and thus need only a polynomial amount of space.

Since the first formal statements of backtracking algorithms over 40 years ago [30, 57], many techniques for improving the efficiency of a backtracking search algorithm have been suggested and evaluated. In this chapter, I survey some of the most important techniques including branching strategies, constraint propagation, nogood recording, backjumping, heuristics for variable and value ordering, randomization and restart strategies, and alternatives to depth-first search. The techniques are not always orthogonal and sometimes combining two or more techniques into one algorithm has a multiplicative effect (such as

combining restarts with nogood recording) and sometimes it has a degradation effect (such as increased constraint propagation versus backjumping). Given the many possible ways that these techniques can be combined together into one algorithm, I also survey work on comparing backtracking algorithms. The best combinations of these techniques result in robust backtracking algorithms that can now routinely solve large, hard instances that are of practical importance.

4.1 Preliminaries

In this section, I first define the constraint satisfaction problem followed by a brief review of the needed background on backtracking search.

Definition 4.1 (CSP). *A* constraint satisfaction problem *(CSP) consists of a set of* variables, $X = \{x_1, \ldots, x_n\}$*; a set of* values, $D = \{a_1, \ldots, a_d\}$*, where each variable* $x_i \in X$ *has an associated finite domain* $dom(x_i) \subseteq D$ *of possible values; and a collection of* constraints.

Each constraint C is a relation—a set of tuples—over some set of variables, denoted by $vars(C)$. The size of the set $vars(C)$ is called the *arity* of the constraint. A *unary* constraint is a constraint of arity one, a *binary* constraint is a constraint of arity two, a *non-binary* constraint is a constraint of arity greater than two, and a *global* constraint is a constraint that can be over arbitrary subsets of the variables. A constraint can be specified *intensionally* by specifying a formula that tuples in the constraint must satisfy, or *extensionally* by explicitly listing the tuples in the constraint. A *solution* to a CSP is an assignment of a value to each variable that satisfies all the constraints. If no solution exists, the CSP is said to be inconsistent or unsatisfiable.

As a running example in this survey, I will use the 6-queens problem: how can we place 6 queens on a 6×6 chess board so that no two queens attack each other. As one possible CSP model, let there be a variable for each column of the board $\{x_1, \ldots, x_6\}$, each with domain $dom(x_i) = \{1, \ldots, 6\}$. Assigning a value j to a variable x_i means placing a queen in row j, column i. Between each pair of variables x_i and x_j, $1 \leq i < j \leq 6$, there is a constraint $C(x_i, x_j)$, given by $(x_i \neq x_j) \wedge (|i - j| \neq |x_i - x_j|)$. One possible solution is given by $\{x_1 = 4, x_2 = 1, x_3 = 5, x_4 = 2, x_5 = 6, x_6 = 3\}$.

The satisfiability problem (SAT) is a CSP where the domains of the variables are the Boolean values and the constraints are Boolean formulas. I will assume that the constraints are in conjunctive normal form and are thus written as clauses. A literal is a Boolean variable or its negation and a clause is a disjunction of literals. For example, the formula $\neg x_1 \vee x_2 \vee x_3$ is a clause. A clause with one literal is called a unit clause; a clause with no literals is called the empty clause. The empty clause is unsatisfiable.

A backtracking search for a solution to a CSP can be seen as performing a depth-first traversal of a *search tree*. The search tree is generated as the search progresses and represents alternative choices that may have to be examined in order to find a solution. The method of extending a node in the search tree is often called a *branching strategy*, and several alternatives have been proposed and examined in the literature (see Section 4.2). A backtracking algorithm *visits* a node if, at some point in the algorithm's execution, the node is generated. Constraints are used to check whether a node may possibly lead to a solution of the CSP and to prune subtrees containing no solutions. A node in the search tree is a *deadend* if it does not lead to a solution.

The naive backtracking algorithm (BT) is the starting point for all of the more sophisticated backtracking algorithms (see Table 4.1). In the BT search tree, the root node at level 0 is the empty set of assignments and a node at level j is a set of assignments $\{x_1 = a_1, \ldots, x_j = a_j\}$. At each node in the search tree, an uninstantiated variable is selected and the branches out of this node consist of all possible ways of extending the node by instantiating the variable with a value from its domain. The branches represent the different choices that can be made for that variable. In BT, only constraints with no uninstantiated variables are checked at a node. If a constraint check fails—a constraint is not satisfied—the next domain value of the current variable is tried. If there are no more domain values left, BT backtracks to the most recently instantiated variable. A solution is found if all constraint checks succeed after the last variable has been instantiated.

Figure 4.1 shows a fragment of the backtrack tree generated by the naive backtracking algorithm (BT) for the 6-queens problem. The labels on the nodes are shorthands for the set of assignments at that node. For example, the node labeled 25 consists of the set of assignments $\{x_1 = 2, x_2 = 5\}$. White dots denote nodes where all the constraints with no uninstantiated variables are satisfied (no pair of queens attacks each other). Black dots denote nodes where one or more constraint checks fail. (The reasons for the shading and dashed arrows are explained in Section 4.5.) For simplicity, I have assumed a static order of instantiation in which variable x_i is always chosen at level i in the search tree and values are assigned to variables in the order $1, \ldots, 6$.

4.2 Branching Strategies

In the naive backtracking algorithm (BT), a node $p = \{x_1 = a_1, \ldots, x_j = a_j\}$ in the search tree is a set of assignments and p is extended by selecting a variable x and adding a branch to a new node $p \cup \{x = a\}$, for each $a \in dom(x)$. The assignment $x = a$ is said to be *posted* along a branch. As the search progresses deeper in the tree, additional assignments are posted and upon backtracking the assignments are retracted. However, this is just one possible branching strategy, and several alternatives have been proposed and examined in the literature.

More generally, a node $p = \{b_1, \ldots, b_j\}$ in the search tree of a backtracking algorithm is a set of *branching constraints*, where b_i, $1 \leq i \leq j$, is the branching constraint posted at level i in the search tree. A node p is extended by adding the branches $p \cup \{b^1_{j+1}\}, \ldots, p \cup \{b^k_{j+1}\}$, for some branching constraints b^i_{j+1}, $1 \leq i \leq k$. The branches are often ordered using a heuristic, with the left-most branch being the most promising. To ensure completeness, the constraints posted on all the branches from a node must be mutually exclusive and exhaustive.

Usually, branching strategies consist of posting unary constraints. In this case, a variable ordering heuristic is used to select the next variable to branch on and the ordering of the branches is determined by a value ordering heuristic (see Section 4.6). As a running example, let x be the variable to be branched on, let $dom(x) = \{1, \ldots, 6\}$, and assume that the value ordering heuristic is lexicographic ordering. Three popular branching strategies involving unary constraints are the following.

1. *Enumeration.* The variable x is instantiated in turn to each value in its domain. A branch is generated for each value in the domain of the variable and the constraint $x = 1$ is posted along the first branch, $x = 2$ along the second branch, and so

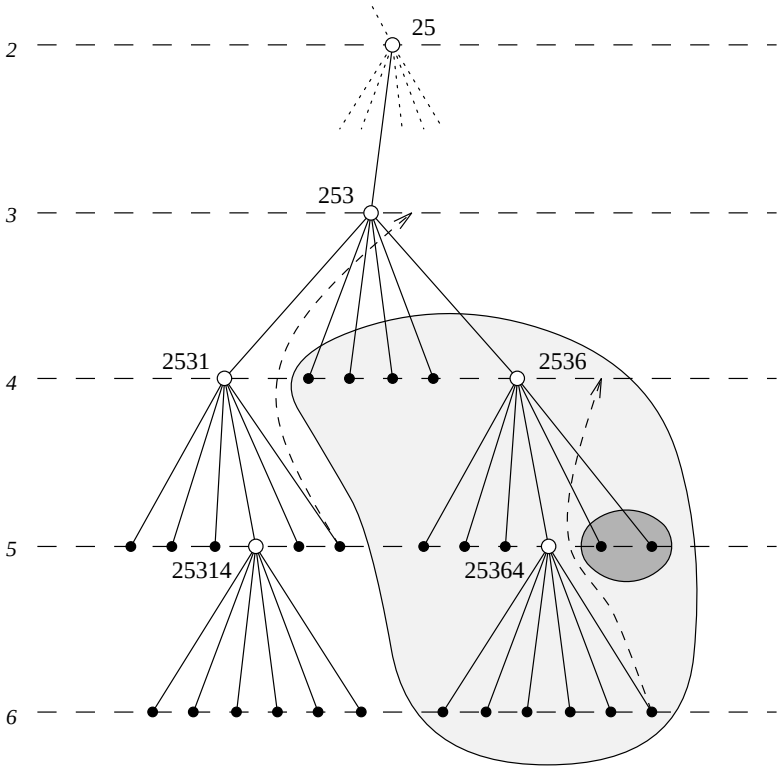

Figure 4.1: A fragment of the BT backtrack tree for the 6-queens problem (from [79]).

on. The enumeration branching strategy is assumed in many textbook presentations of backtracking and in much work on backtracking algorithms for solving CSPs. An alternative name for this branching strategy in the literature is d-way branching, where d is the size of the domain.

2. *Binary choice points.* The variable x is instantiated to some value in its domain. Assuming the value 1 is chosen in our example, two branches are generated and the constraints $x = 1$ and $x \neq 1$ are posted, respectively. This branching strategy is often used in constraint programming languages for solving CSPs (see, e.g., [72, 123]) and is used by Sabin and Freuder [116] in their backtracking algorithm which maintains arc consistency during the search. An alternative name for this branching strategy in the literature is 2-way branching.

3. *Domain splitting.* Here the variable is not necessarily instantiated, but rather the choices for the variable are reduced in each subproblem. For ordered domains such as in our example, this could consist of posting a constraint of the form $x \leq 3$ on one branch and posting $x > 3$ on the other branch.

The three schemes are, of course, identical if the domains are binary (such as, for example, in SAT).

Table 4.1: Some named backtracking algorithms. Hybrid algorithms which combine techniques are denoted by hyphenated names. For example, MAC-CBJ is an algorithm that maintains arc consistency and performs conflict-directed backjumping.

BT	Naive backtracking: checks constraints with no uninstantiated variables; chronologically backtracks.
MAC	Maintains arc consistency on constraints with *at least* one uninstantiated variable; chronologically backtracks.
FC	Forward checking algorithm: maintains arc consistency on constraints with *exactly* one uninstantiated variable; chronologically backtracks.
DPLL	Forward checking algorithm specialized to SAT problems: uses unit propagation; chronologically backtracks.
MC$_k$	Maintains strong k-consistency; chronologically backtracks.
CBJ	Conflict-directed backjumping; no constraint propagation.
BJ	Limited backjumping; no constraint propagation.
DBT	Dynamic backtracking: backjumping with 0-order relevance-bounded nogood recording; no constraint propagation.

Branching strategies that consist of posting non-unary constraints have also been proposed, as have branching strategies that are specific to a class of problems. As an example of both, consider job shop scheduling where we must schedule a set of tasks $t_1, \ldots, t_k$ on a set of resources. Let x_i be a finite domain variable representing the starting time of t_i and let d_i be the fixed duration of t_i. A popular branching strategy is to order or serialize the tasks that share a resource. Consider two tasks t_1 and t_2 that share the same resource. The branching strategy is to post the constraint $x_1 + d_1 \leq x_2$ along one branch and to post the constraint $x_2 + d_2 \leq x_1$ along the other branch (see, e.g., [23] and references therein). This continues until either a deadend is detected or all tasks have been ordered. Once all tasks are ordered, one can easily construct a solution to the problem; i.e., an assignment of a value to each x_i. It is interesting to note that, conceptually, the above branching strategy is equivalent to adding auxiliary variables to the CSP model which are then branched on. For the two tasks t_1 and t_2 that share the same resource, we would add the auxiliary variable O_{12} with $dom(O_{12}) = \{0, 1\}$ and the constraints $O_{12} = 1 \iff x_1 + d_1 \leq x_2$ and $O_{12} = 0 \iff x_2 + d_2 \leq x_1$. In general, if the underlying backtracking algorithm has a fixed branching strategy, one can simulate a different branching strategy by adding auxiliary variables. Thus, the choice of branching strategy and the design of the CSP model are interdependent decisions.

There has been further work on branching strategies that has examined the relative power of the strategies and proposed new strategies. Van Hentenryck [128, pp.90–92] examines tradeoffs between the enumeration and domain splitting strategies. Milano and van Hoeve [97] show that branching strategies can be viewed as the combination of a value ordering heuristic and a domain splitting strategy. The value ordering is used to rank the domain values and the domain splitting strategy is used to partition the domain into two or

more sets. Of course, the set with the most highly ranked values will be branched into first. The technique is shown to work well on optimization problems.

Smith and Sturdy [121] show that when using chronological backtracking with 2-way branching to find all solutions, the value ordering can have an effect on the efficiency of the backtracking search. This is a surprise, since it is known that value ordering has no effect under these circumstances when using d-way branching. Hwang and Mitchell [71] show that backtracking with 2-way branching is exponentially more powerful than backtracking with d-way branching. It is clear that d-way branching can be simulated by 2-way branching with no loss of efficiency. Hwang and Mitchell show that the converse does not hold. They give a class of problems where a d-way branching algorithm with an optimal variable and value ordering takes exponentially more steps than a 2-way branching algorithm with a simple variable and value ordering. However, note that the result holds only if the CSP model is assumed to be fixed. It does not hold if we are permitted to add auxiliary variables to the CSP model.

4.3 Constraint Propagation

A fundamental insight in improving the performance of backtracking algorithms on CSPs is that local inconsistencies can lead to much thrashing or unproductive search [47, 89]. A *local inconsistency* is an instantiation of some of the variables that satisfies the relevant constraints but cannot be extended to one or more additional variables and so cannot be part of any solution. (Local inconsistencies are nogoods; see Section 4.4.) If we are using a backtracking search to find a solution, such an inconsistency can be the reason for many deadends in the search and cause much futile search effort. This insight has led to:

(a) the definition of conditions that characterize the level of local consistency of a CSP (e.g., [39, 89, 102]),

(b) the development of constraint propagation algorithms—algorithms which enforce these levels of local consistency by removing inconsistencies from a CSP (e.g., [89, 102]), and

(c) effective backtracking algorithms for finding solutions to CSPs that maintain a level of local consistency during the search (e.g., [31, 47, 48, 63, 93]).

A generic scheme to maintain a level of local consistency in a backtracking search is to perform constraint propagation at each node in the search tree. Constraint propagation algorithms remove local inconsistencies by posting additional constraints that rule out or remove the inconsistencies. When used during search, constraints are posted at nodes as the search progresses deeper in the tree. But upon backtracking over a node, the constraints that were posted at that node must be retracted. When used at the root node of the search tree—before any instantiations or branching decisions have been made—constraint propagation is sometimes referred to as a preprocessing stage.

Backtracking search integrated with constraint propagation has two important benefits. First, removing inconsistencies during search can dramatically prune the search tree by removing many deadends and by simplify the remaining subproblem. In some cases, a variable will have an empty domain after constraint propagation; i.e., no value satisfies the unary constraints over that variable. In this case, backtracking can be initiated as there

is no solution along this branch of the search tree. In other cases, the variables will have their domains reduced. If a domain is reduced to a single value, the value of the variable is forced and it does not need to be branched on in the future. Thus, it can be much easier to find a solution to a CSP after constraint propagation or to show that the CSP does not have a solution. Second, some of the most important variable ordering heuristics make use of the information gathered by constraint propagation to make effective variable ordering decisions (this is discussed further in Section 4.6). As a result of these benefits, it is now standard for a backtracking algorithm to incorporate some form of constraint propagation.

Definitions of local consistency can be categorized in at least two ways. First, the definitions can be categorized into those that are constraint-based and those that are variable-based, depending on what are the primitive entities in the definition. Second, definitions of local consistency can be categorized by whether *only* unary constraints need to be posted during constraint propagation, or whether posting constraints of higher arity is sometimes necessary. In implementations of backtracking, the domains of the variables are represented extensionally, and posting and retracting unary constraints can be done very efficiently by updating the representation of the domain. Posting and retracting constraints of higher arity is less well understood and more costly. If only unary constraints are necessary, constraint propagation is sometimes referred to as domain filtering or domain pruning.

The idea of incorporating some form of constraint propagation into a backtracking algorithm arose from several directions. Davis and Putnam [31] propose unit propagation, a form of constraint propagation specialized to SAT. Golomb and Baumert [57] may have been the first to informally describe the idea of improving a general backtracking algorithm by incorporating some form of domain pruning during the search. Constraint propagation techniques were used in Fikes' REF-ARF [37] and Lauriere's Alice [82], both languages for stating and solving CSPs. Gaschnig [47] was the first to propose a backtracking algorithm that enforces a precisely defined level of local consistency at each node. Gaschnig's algorithm used d-way branching. Mackworth [89] generalizes Gaschnig's proposal to backtracking algorithms that interleave case-analysis with constraint propagation (see also [89] for additional historical references).

Since this early work, a vast literature on constraint propagation and local consistency has arisen; more than I can reasonably discuss in the space available. Thus, I have chosen two representative examples: arc consistency and strong k-consistency. These local consistencies illustrate the different categorizations given above. As well, arc consistency is currently the most important local consistency in practice and has received the most attention so far, while strong k-consistency has played an important role on the theoretical side of CSPs. For each of these examples, I present the definition of the local consistency, followed by a discussion of backtracking algorithms that maintain this level of local consistency during the search. I do not discuss any specific constraint propagation algorithms. Two separate chapters in this Handbook have been devoted to this topic (see Chapters 3 & 6). Note that many presentations of constraint propagation algorithms are for the case where the algorithm will be used in the preprocessing stage. However, when used during search to maintain a level of local consistency, usually only small changes occur between successive calls to the constraint propagation algorithm. As a result, much effort has also gone into making such algorithms incremental and thus much more efficient when used during search.

When presenting backtracking algorithms integrated with constraint propagation, I present the "pure" forms of the backtracking algorithms where a uniform level of local

consistency is maintained at each node in the search tree. This is simply for ease of presentation. In practice, the level of local consistency enforced and the algorithm for enforcing it is specific to each constraint and varies between constraints. An example is the widely used all-different global constraint, where fast algorithms are designed for enforcing many different levels of local consistency including arc consistency, range consistency, bounds consistency, and simple value removal. The choice of which level of local consistency to enforce is then up to the modeler.

4.3.1 Backtracking and Maintaining Arc Consistency

Mackworth [89, 90] defines a level of local consistency called arc consistency[1]. Given a constraint C, the notation $t \in C$ denotes a tuple t—an assignment of a value to each of the variables in $vars(C)$—that satisfies the constraint C. The notation $t[x]$ denotes the value assigned to variable x by the tuple t.

Definition 4.2 (arc consistency)**.** *Given a constraint C, a value $a \in dom(x)$ for a variable $x \in vars(C)$ is said to have a* support *in C if there exists a tuple $t \in C$ such that $a = t[x]$ and $t[y] \in dom(y)$, for every $y \in vars(C)$. A constraint C is said to be* arc consistent *if for each $x \in vars(C)$, each value $a \in dom(x)$ has a support in C.*

A constraint can be made arc consistent by repeatedly removing unsupported values from the domains of its variables. Note that this definition of local consistency is constraint-based and enforcing arc consistency on a CSP means iterating over the constraints until no more changes are made to the domains. Algorithms for enforcing arc consistency have been extensively studied (see Chapters 3 & 6). An optimal algorithm for an arbitrary constraint has $O(rd^r)$ worst case time complexity, where r is the arity of the constraint and d is the size of the domains of the variables [101]. Fortunately, it is almost always possible to do much better for classes of constraints that occur in practice. For example, the all-different constraint can be made arc consistent in $O(r^2d)$ time in the worst case.

Gaschnig [47] suggests maintaining arc consistency during backtracking search and gives the first explicit algorithm containing this idea. Following Sabin and Freuder [116], I will denote such an algorithm as MAC[2]. The MAC algorithm maintains arc consistency on constraints with *at least* one uninstantiated variable (see Table 4.1). At each node of the search tree, an algorithm for enforcing arc consistency is applied to the CSP. Since arc consistency was enforced on the parent of a node, initially constraint propagation only needs to be enforced on the constraint that was posted by the branching strategy. In turn, this may lead to other constraints becoming arc inconsistent and constraint propagation continues until no more changes are made to the domains. If, as a result of constraint propagation, a domain becomes empty, the branch is a deadend and is rejected. If no domain is empty, the branch is accepted and the search continues to the next level.

[1] Arc consistency is also called domain consistency, generalized arc consistency, and hyper arc consistency in the literature. The latter two names are used when an author wishes to reserve the name arc consistency for the case where the definition is restricted to binary constraints.

[2] Gaschnig's DEEB (Domain Element Elimination with Backtracking) algorithm uses d-way branching. Sabin and Freuder's [116] MAC (Maintaining Arc Consistency) algorithm uses 2-way branching. However, I will follow the practice of much of the literature and use the term MAC to denote an algorithm that maintains arc consistency during the search, regardless of the branching strategy used.

As an example of applying MAC, consider the backtracking tree for the 6-queens problem shown in Figure 4.1. MAC visits only node 25, as it is discovered that this node is a deadend. The board in Figure 4.2a shows the result of constraint propagation. The shaded numbered squares correspond to the values removed from the domains of the variables by constraint propagation. A value i is placed in a shaded square if the value was removed because of the assignment at level i in the tree. It can been seen that after constraint propagation, the domains of some of the variables are empty. Thus, the set of assignments $\{x_1 = 2, x_2 = 5\}$ cannot be part of a solution to the CSP.

When maintaining arc consistency during search, any value that is pruned from the domain of a variable does not participate in any solution to the CSP. However, not all values that remain in the domains necessarily are part of some solution. Hence, while arc consistency propagation can reduce the search space, it does not remove all possible deadends. Let us say that the domains of a CSP are *minimal* if each value in the domain of a variable is part of some solution to the CSP. Clearly, if constraint propagation would leave only the minimal domains at each node in the search tree, the search would be backtrack-free as any value that was chosen would lead to a solution. Unfortunately, finding the *minimal domains* is at least as hard as solving the CSP. After enforcing arc consistency on individual constraints, each value in the domain of a variable is part of some solution to the constraint considered in isolation. Finding the minimal domains would be equivalent to enforcing arc consistency on the *conjunction* of the constraints in a CSP, a process that is worst-case exponential in n, the number of variables in the CSP. Thus, arc consistency can be viewed as approximating the minimal domains.

In general, there is a tradeoff between the cost of the constraint propagation performed at each node in the search tree, and the quality of the approximation of the minimal domains. One way to *improve* the approximation, but with an increase in the cost of constraint propagation, is to use a stronger level of local consistency such as a singleton consistency (see Chapter 3). One way to *reduce* the cost of constraint propagation, at the risk of a poorer approximation to the minimal domains and an increase in the overall search cost, is to restrict the application of arc consistency. One such algorithm is called forward checking. The forward checking algorithm (FC) maintains arc consistency on constraints with *exactly* one uninstantiated variable (see Table 4.1). On such constraints, arc consistency can be enforced in $O(d)$ time, where d is the size of the domain of the uninstantiated variable. Golomb and Baumert [57] may have been the first to informally describe forward checking (called preclusion in [57]). The first explicit algorithms are given by McGregor [93] and Haralick and Elliott [63]. Forward checking was originally proposed for binary constraints. The generalization to non-binary constraints used here is due to Van Hentenryck [128].

As an example of applying FC, consider the backtracking tree shown in Figure 4.1. FC visits only nodes 25, 253, 2531, 25314 and 2536. The board in Figure 4.2b shows the result of constraint propagation. The squares that are left empty as the search progresses correspond to the nodes visited by FC.

Early experimental work in the field found that FC was much superior to MAC [63, 93]. However, this superiority turned out to be partially an artifact of the easiness of the benchmarks. As well, many practical improvements have been made to arc consistency propagation algorithms over the intervening years, particularly with regard to incrementality. The result is that backtracking algorithms that maintain full arc consistency during the search are now considered much more important in practice. An exception is the widely

used DPLL algorithm [30, 31], a backtracking algorithm specialized to SAT problems in CNF form (see Table 4.1). The DPLL algorithm uses unit propagation, sometimes called Boolean constraint propagation, as its constraint propagation mechanism. It can be shown that unit propagation is equivalent to forward checking on a SAT problem. Further, it can be shown that the amount of pruning performed by arc consistency on these problems is equivalent to that of forward checking. Hence, forward checking is the right level of constraint propagation on SAT problems.

Forward checking is just one way to restrict arc consistency propagation; many variations are possible. For example, one can maintain arc consistency on constraints with various numbers of uninstantiated variables. Bessière et al. [16] consider the possibilities. One could also take into account the size of the domains of uninstantiated variables when specify which constraints should be propagated. As a third alternative, one could place *ad hoc* restrictions on the constraint propagation algorithm itself and how it iterates through the constraints [63, 104, 117].

An alternative to restricting the *application* of arc consistency—either by restricting which constraints are propagated or by restricting the propagation itself—is to restrict the *definition* of arc consistency. One important example is bounds consistency. Suppose that the domains of the variables are large and ordered and that the domains of the variables are represented by intervals (the minimum and the maximum value in the domain). With bounds consistency, instead of asking that each value $a \in dom(x)$ has a support in the constraint, we only ask that the minimum value and the maximum value each have a support in the constraint. Although in general weaker than arc consistency, bounds consistency has been shown to be useful for arithmetic constraints and global constraints as it can sometimes be enforced more efficiently (see Chapters 3 & 6 for details). For example, the all-different constraint can be made bounds consistent in $O(r)$ time in the worst case, in contrast to $O(r^2d)$ for arc consistency, where r is the arity of the constraint and d is the size of the domains of the variables. Further, for some problems it can be shown that the amount of pruning performed by arc consistency is equivalent to that of bounds consistency, and thus the extra cost of arc consistency is not repaid.

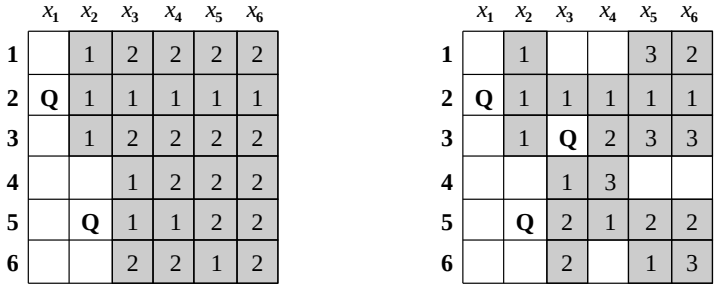

Figure 4.2: Constraint propagation on the 6-queens problem; (a) maintaining arc consistency; (b) forward checking.

4.3.2 Backtracking and Maintaining Strong k-Consistency

Freuder [39, 40] defines a level of local consistency called strong k-consistency. A set of assignments is *consistent* if each constraint that has *all* of its variables instantiated by the set of assignments is satisfied.

Definition 4.3 (strong k-consistency)**.** *A CSP is k-consistent if, for any set of assignments $\{x_1 = a_1, \ldots, x_{k-1} = a_{k-1}\}$ to $k-1$ distinct variables that is consistent, and any additional variable x_k, there exists a value $a_k \in dom(x_k)$ such that the set of assignments $\{x_1 = a_1, \ldots, x_{k-1} = a_{k-1}, x_k = a_k\}$ is consistent. A CSP is strongly k-consistent if it is j-consistent for all $j \leq k$.*

For the special case of binary CSPs, strong 2-consistency is the same as arc consistency and strong 3-consistency is also known as path consistency. A CSP can be made strongly k-consistent by repeatedly detecting and removing all those inconsistencies $t = \{x_1 = a_1, \ldots, x_{j-1} = a_{j-1}\}$ where $1 \leq j < k$ and t is consistent but cannot be extended to some j^{th} variable x_j. To remove an inconsistency or nogood t, a constraint is posted to the CSP which rules out the tuple t. Enforcing strong k-consistency may dramatically increase the number of constraints in a CSP, as the number of new constraints posted can be exponential in k. Once a CSP has been made strongly k-consistent any value that remains in the domain of a variable can be extended to a consistent set of assignments over k variables in a backtrack-free manner. However, unless $k = n$, there is no guarantee that a value can be extended to a solution over all n variables. An optimal algorithm for enforcing strong k-consistency on a CSP containing arbitrary constraints has $O(n^k d^k)$ worst case time complexity, where n is the number of variables in the CSP and d is the size of the domains of the variables [29].

Let MC$_k$ be an algorithm that maintains strong k-consistency during the search (see Table 4.1). For the purposes of specifying MC$_k$, I will assume that the branching strategy is enumeration and that, therefore, each node in the search tree corresponds to a set of assignments. During search, we want to maintain the property that any value that remains in the domain of a variable can be extended to a consistent set of assignments over k variables. To do this, we must account for the current set of assignments by, conceptually, modifying the constraints. Given a set of assignments t, only those tuples in a constraint that agree with the assignments in t are selected and those tuples are then projected onto the set of uninstantiated variables of the constraint to give the new constraint (see [25] for details). Under such an architecture, FC can be viewed as maintaining one-consistency, and, for binary CSPs, MAC can be viewed as maintaining strong two-consistency.

Can such an architecture be practical for $k > 2$? There is some evidence that the answer is yes. Van Gelder and Tsuji [127] propose an algorithm that maintains the closure of resolution on binary clauses (clauses with two literals) and gives experimental evidence that the algorithm can be much faster than DPLL on larger SAT instances. The algorithm can be viewed as MC$_3$ specialized to SAT. Bacchus [2] builds on this work and shows that the resulting SAT solver is robust and competitive with state-of-the-art DPLL solvers. This is remarkable given the amount of engineering that has gone into DPLL solvers. So far, however, there has been no convincing demonstration of a corresponding result for general CSPs, although efforts have been made.

4.4 Nogood Recording

One of the most effective techniques known for improving the performance of backtracking search on a CSP is to add implied constraints. A constraint is *implied* if the set of solutions to the CSP is the same with and without the constraint. Adding the "right" implied constraints to a CSP can mean that many deadends are removed from the search tree and other deadends are discovered after much less search effort.

Three main techniques for adding implied constraints have been investigated. One technique is to add implied constraints by hand during the modeling phase (see Chapter 11). A second technique is to automatically add implied constraints by applying a constraint propagation algorithm (see Section 4.3). Both of the above techniques rule out local inconsistencies or deadends *before* they are encountered during the search. A third technique, and the topic of this section, is to automatically add implied constraints *after* a local inconsistency or deadend is encountered in the search. The basis of this technique is the concept of a nogood, due to Stallman and Sussman [124][3].

Definition 4.4 (nogood)**.** *A nogood is a set of assignments and branching constraints that is not consistent with any solution.*

In other words, there does not exist a solution—an assignment of a value to each variable that satisfies all the constraints of the CSP—that also satisfies all the assignments and branching constraints in the nogood. If we are using a backtracking search to find a solution, each deadend corresponds to a nogood. Thus nogoods are the cause of all futile search effort. Once a nogood for a deadend is discovered, it can be ruled out by adding a constraint. Of course, it is too late for this deadend—the backtracking algorithm has already refuted this node, perhaps at great cost—but the hope is that the constraint will prune the search space in the future. The technique, first informally described by Stallman and Sussman [124], is often referred to as nogood or constraint recording.

As an example of a nogood, consider the 6-queens problem. The set of assignments $\{x_1 = 2, x_2 = 5, x_3 = 3\}$ is a nogood since it is not contained in any solution (see the backtracking tree shown in Figure 4.1 where the node 253 is the root of a failed subtree). To rule out the nogood, the implied constraint $\neg(x_1 = 2 \wedge x_2 = 5 \wedge x_3 = 3)$ could be recorded, which is just $x_1 \neq 2 \vee x_2 \neq 5 \vee x_3 \neq 3$ in clause form.

The recorded constraints can be checked and propagated just like the original constraints. In particular, since nogoods correspond to constraints which are clauses, forward checking is an appropriate form of constraint propagation. As well, nogoods can be used for backjumping (see Section 4.5). Nogood recording—or discovering and recording implied constraints during the search—can be viewed as an adaptation of the well-known technique of adding caching (sometimes called memoization) to backtracking search. The idea is to cache solutions to subproblems and reuse the solutions instead of recomputing them.

The constraints that are added through nogood recording could, in theory, have been ruled out *a priori* using a constraint propagation algorithm. However, while constraint propagation algorithms which add implied unary constraints are especially important, the

[3] Most previous work on nogood recording implicitly assumes that the backtracking algorithm is performing *d*-way branching (only adding branching constraints which are assignments) and drops the phrase "*and branching constraints*" from the definition. The generalized definition and descriptions used in this section are inspired by the work of Rochart, Jussien, and Laburthe [113].

algorithms which add higher arity constraints often add too many implied constraints that are not useful and the computational cost is not repaid by a faster search.

4.4.1 Discovering Nogoods

Stallman and Sussman's [124] original account of discovering nogoods is embedded in a rule-based programming language and is descriptive and informal. Bruynooghe [22] informally adapts the idea to backtracking search on CSPs. Dechter [33] provides the first formal account of discovering and recording nogoods. Dechter [34] shows how to discover nogoods using the static structure of the CSP.

Prosser [108], Ginsberg [54], and Schiex and Verfaillie [118] all independently give accounts of how to discover nogoods dynamically during the search. The following definition captures the essence of these proposals. The definition is for the case where the backtracking algorithm does not perform any constraint propagation. (The reason for the adjective "jumpback" is explained in Section 4.5.) Recall that associated with each node in the search tree is the set of branching constraints posted along the path to the node. For d-way branching, the branching constraints are of the form $x = a$, for some variable x and value a; for 2-way branching, the branching constraints are of the form $x = a$ and $x \neq a$; and for domain splitting, the branching constraints are of the form $x \leq a$ and $x > a$.

Definition 4.5 (jumpback nogood)**.** *Let $p = \{b_1, \ldots, b_j\}$ be a deadend node in the search tree, where b_i, $1 \leq i \leq j$, is the branching constraint posted at level i in the search tree. The* jumpback nogood *for p, denoted $J(p)$, is defined recursively as follows.*

1. *p is a leaf node. Let C be a constraint that is not consistent with p (one must exist);*

$$J(p) = \{b_i \mid vars(b_i) \cap vars(C) \neq \emptyset, 1 \leq i \leq j\}.$$

2. *p is not a leaf node. Let $\{b^1_{j+1}, \ldots, b^k_{j+1}\}$ be all the possible extensions of p attempted by the branching strategy, each of which has failed;*

$$J(p) = \bigcup_{i=1}^{k} (J(p \cup \{b^i_{j+1}\}) - \{b^i_{j+1}\}).$$

As an example of applying the definition, consider the jumpback nogood for the node 25314 shown in Figure 4.1. The set of branching constraints associated with this node is $p = \{x_1 = 2, x_2 = 5, x_3 = 3, x_4 = 1, x_5 = 4\}$. The backtracking algorithm branches on x_6, but all attempts to extend p fail. The jumpback nogood is given by,

$$\begin{aligned} J(p) &= (J(p \cup \{x_6 = 1\}) - \{x_6 = 1\}) \cup \cdots \cup (J(p \cup \{x_6 = 6\}) - \{x_6 = 6\}), \\ &= \{x_2 = 5\} \cup \cdots \cup \{x_3 = 3\}, \\ &= \{x_1 = 2, x_2 = 5, x_3 = 3, x_5 = 4\}. \end{aligned}$$

Notice that the order in which the constraints are checked or propagated directly influences which nogood is discovered. In applying the above definition, I have chosen to check the constraints in increasing lexicographic order. For example, for the leaf node $p \cup \{x_6 = 1\}$, both $C(x_2, x_6)$ and $C(x_4, x_6)$ fail—i.e., both the queen at x_2 and the queen at x_4 attack the queen at x_6—and I have chosen $C(x_2, x_6)$.

The discussion so far has focused on the simpler case where the backtracking algorithm does not perform any constraint propagation. Several authors have contributed to our understanding of how to discover nogoods when the backtracking algorithm does use constraint propagation. Rosiers and Bruynooghe [114] give an informal description of combining forward checking and nogood recording. Schiex and Verfaillie [118] provide the first formal account of nogood recording within an algorithm that performs forward checking. Prosser's FC-CBJ [108] and MAC-CBJ [109] can be viewed as discovering jumpback nogoods (see Section 4.5.1). Jussien, Debruyne, and Boizumault [75] give an algorithm that combines nogood recording with arc consistency propagation on non-binary constraints. The following discussion captures the essence of these proposals. The key idea is to modify the constraint propagation algorithms so that, for each value that is removed from the domain of some variable, an eliminating explanation is recorded.

Definition 4.6 (eliminating explanation)**.** *Let $p = \{b_1, \ldots, b_j\}$ be a node in the search tree and let $a \in dom(x)$ be a value that is removed from the domain of a variable x by constraint propagation at node p. An* eliminating explanation *for a, denoted $\mathsf{expl}(x \neq a)$, is a subset (not necessarily proper) of p such that $\mathsf{expl}(x \neq a) \cup \{x = a\}$ is a nogood.*

The intention behind the definition is that $\mathsf{expl}(x \neq a)$ is sufficient to account for the removal of a. As an example, consider the board in Figure 4.2a which shows the result of arc consistency propagation. At the node $p = \{x_1 = 2, x_2 = 5\}$, the value 1 is removed from $dom(x_6)$. An eliminating explanation for this value is $\mathsf{expl}(x_6 \neq 1) = \{x_2 = 5\}$, since $\{x_2 = 5, x_6 = 1\}$ is a nogood. An eliminating explanation can be viewed as the left-hand side of an implication which rules out the stated value. For example, the implied constraint to rule out the nogood $\{x_2 = 5, x_6 = 1\}$ is $\neg(x_2 = 5 \wedge x_6 = 1)$, which can be rewritten as $(x_2 = 5) \Rightarrow (x_6 \neq 1)$. Similarly, $\mathsf{expl}(x_6 \neq 3) = \{x_1 = 2, x_2 = 5\}$ and the corresponding implied constraint can be written as $(x_1 = 2 \wedge x_2 = 5) \Rightarrow (x_6 \neq 3)$.

One possible method for constructing eliminating explanations for arc consistency propagation is as follows. Initially at a node, a branching constraint b_j is posted and arc consistency is enforced on b_j. For each value a removed from the domain of a variable $x \in vars(b_j)$, $\mathsf{expl}(x \neq a)$ is set to $\{b_j\}$. Next constraint propagation iterates through the constraints re-establishing arc consistency. Consider a value a removed from the domain of a variable x during this phase of constraint propagation. We must record an explanation that accounts for the removal of a; i.e., the reason that a does not have a support in some constraint C. For each value b of a variable $y \in vars(C)$ which could have been used to form a support for $a \in dom(x)$ in C but has been removed from its domain, add the eliminating explanation for $y \neq b$ to the eliminating explanation for $x \neq a$; i.e. $\mathsf{expl}(x \neq a) \leftarrow \mathsf{expl}(x \neq a) \cup \mathsf{expl}(y \neq b)$. In the special case of arc consistency propagation called forward checking, it can be seen that the eliminating explanation is just the variable assignments of the instantiated variables in C.

The jumpback nogood in the case where the backtracking algorithm performs constraint propagation can now be defined as follows.

Definition 4.7 (jumpback nogood with constraint propagation)**.** *Let $p = \{b_1, \ldots, b_j\}$ be a deadend node in the search tree. The* jumpback nogood *for p, denoted $J(p)$, is defined recursively as follows.*

1. *p is a leaf node. Let x be a variable whose domain has become empty (one must exist), where $dom(x)$ is the original domain of x;*

$$J(p) = \bigcup_{a \in dom(x)} \textsf{expl}(x \neq a).$$

2. *p is not a leaf node. Same as Definition 4.5.*

Note that the jumpback nogoods are not guaranteed to be the minimal nogood or the "best" nogood that could be discovered, even if the nogoods are locally minimal at leaf nodes. For example, Bacchus [1] shows that the jumpback nogood for forward checking may not give the best backjump point and provides a method for improving the nogood. Katsirelos and Bacchus [77] show how to discover *generalized* nogoods during search using either FC-CBJ or MAC-CBJ. Standard nogoods are of the form $\{x_1 = a_1 \wedge \cdots \wedge x_k = a_k\}$; i.e., each element is of the form $x_i = a_i$. Generalized nogoods also allow conjuncts of the form $x_i \neq a_i$. When standard nogoods are propagated, a variable can only have a value pruned from its domain. For example, consider the standard nogood clause $x_1 \neq 2 \vee x_2 \neq 5 \vee x_3 \neq 3$. If the backtracking algorithm at some point makes the assignments $x_1 = 2$ and $x_2 = 5$, the value 3 can be removed from the domain of variable x_3. Only indirectly, in the case where all but one of the values have been pruned from the domain of a variable, can propagating nogoods cause the value of a variable to be forced; i.e., cause an assignment of a value to a variable. With generalized nogoods, the value of a variable can also be forced directly which may lead to additional propagation.

Marques-Silva and Sakallah [92] show that in SAT, the effects of Boolean constraint propagation (BCP or unit propagation) can be captured by an implication graph. An implication graph is a directed acyclic graph where the vertices represent variable assignments and directed edges give the reasons for an assignment. A vertex is either positive (the variable is assigned true) or negative (the variable is assigned false). Decision variables and variables which appear as unit clauses in the original formula have no incoming edges; other vertices that are assigned as a result of BCP have incoming edges from vertices that caused the assignment. A contradiction occurs if a variable occurs both positively and negatively. Zhang et al. [139] show that in this scheme, the different cuts in the implication graph which separate all the decision vertices from the contradiction correspond to the different nogoods that can be learned from a contradiction. Zhang et al. show that some types of cuts lead to much smaller and more powerful nogoods than others. As well, the nogoods do not have to include just branching constraints, but can also include assignments that are forced by BCP. Katsirelos and Bacchus [77] generalize the scheme to CSPs and present the results of experimentation with some of the different clause learning schemes.

So far, the discussion on discovering nogoods has focused on methods that are tightly integrated with the search process. Other methods for discovering nogoods have also been proposed. For example, many CSPs contain symmetry and taking into account the symmetry can improve the search for a solution. Freuder and Wallace [43] observe that a symmetry mapping applied to a nogood gives another nogood which may prune additional parts of the search space. For example, the 6-queens problem is symmetric about the horizontal axis and applying this symmetry mapping to the nogood $\{x_1 = 2, x_2 = 5, x_3 = 3\}$ gives the new nogood $\{x_1 = 5, x_2 = 2, x_3 = 4\}$.

Junker [74] shows how nogood discovery can be treated as a separate module, independent of the search algorithm. Given a set of constraints that are known to be inconsistent,

Junker gives an algorithm for finding a small subset of the constraints that is sufficient to explain the inconsistency. The algorithm can make use of constraint propagation techniques, independently of those enforced in the backtracking algorithm, but does not require modifications to the constraint propagation algorithms. As an example, consider the backtracking tree shown in Figure 4.1. Suppose that the backtracking algorithm discovers that node 253 is a deadend. The set of branching constraints associated with this node is $\{x_1 = 2, x_2 = 5, x_3 = 3\}$ and this set is therefore a nogood. Recording this nogood would not be useful. However, the subsets $\{x_1 = 2, x_2 = 5\}$, $\{x_1 = 2, x_3 = 3\}$, and $\{x_2 = 5, x_3 = 3\}$ are also nogoods. All can be discovered using arc consistency propagation. Further, the subsets $\{x_2 = 5\}$ and $\{x_3 = 3\}$ are also nogoods. These are not discoverable using just arc consistency propagation, but are discoverable using a higher level of local consistency. Clearly, everything else being equal, smaller nogoods will lead to more pruning. On CSPs that are more difficult to solve, the extra work involved in discovering these smaller nogoods may result in an overall reduction in search time.

While nogood recording is now standard in SAT solvers, it is currently not widely used for solving general CSPs. Perhaps the main reason is the presence of global constraints in many CSP models and the fact that some form of arc consistency is often maintained on these constraints. If global constraints are treated as a black box, standard methods for determining nogoods quickly lead to saturated nogoods where all or almost all the variables are in the nogood. Saturated nogoods are of little use for either recording or for backjumping. The solution is to more carefully construct eliminating explanations based on the semantics of each global constraint. Katsirelos and Bacchus [77] present preliminary work on learning small generalized nogoods from arc consistency propagation on global constraints. Rochart, Jussien, and Laburthe [113] show how to construct explanations for two important global constraints: the all-different and stretch constraints.

4.4.2 Nogood Database Management

An important problem that arises in nogood recording is the cost of updating and querying the database of nogoods. Stallman and Sussman [124] propose recording a nogood at each deadend in the search. However, if the database becomes too large and too expensive to query, the search reduction that it entails may not be beneficial overall. One method for reducing the cost is to restrict the size of the database by including only those nogoods that are most likely to be useful. Two schemes have been proposed: one restricts the nogoods that are recorded in the first place and the other restricts the nogoods that are kept over time.

Dechter [33, 34] proposes i^{th}-order size-bounded nogood recording. In this scheme a nogood is recorded only if it contains at most i variables. Important special cases are 0-order, where the nogoods are used to determine the backjump point (see Section 4.5) but are not recorded; and 1-order and 2-order, where the nogoods recorded are a subset of those that would be enforced by arc consistency and path consistency propagation, respectively. Early experiments on size-bounded nogood recording were limited to 0-, 1-, and 2-order, since these could be accommodated without moving beyond binary constraints. Dechter [33, 34] shows that 2-order was the best choice and significantly improves BJ on the Zebra problem. Schiex and Verfaillie [118] show that 2-order was the best choice and significantly improves CBJ and FC-CBJ on the Zebra and random binary problems. Frost and Dechter [44] describe the first non-binary implementation of nogood recording

and compare CBJ with and without unrestricted nogood recording and 2-, 3-, and 4-order size-bounded nogood recording. In experiments on random binary problems, they found that neither unrestricted nor size-bounded dominated, but adding either method of nogood recording led to significant improvements overall.

In contrast to restricting the nogoods that are recorded, Ginsberg [54] proposes to record all nogoods but then delete nogoods that are deemed to be no longer relevant. Assume a d-way branching strategy, where all branching constraints are an assignment of a value to a variable, and recall that nogoods can be written in the form,

$$((x_1 = a_1) \wedge \cdots \wedge (x_{k-1} = a_{k-1})) \Rightarrow (x_k \neq a_k).$$

Ginsberg's dynamic backtracking algorithm (DBT) always puts the variable that has most recently been assigned a value on the right-hand side of the implication and only keeps nogoods whose left-hand sides are currently true (see Table 4.1). A nogood is considered irrelevant and deleted once the left-hand side of the implication contains more than one variable-value pair that does not appear in the current set of assignments. When all branching constraints are of the form $x = a$, for some variable x and value a, DBT can be implemented using $O(n^2 d)$ space, where n is the number of variables and d is the size of the domains. The data structure maintains a nogood for each variable and value pair and each nogood is $O(n)$ in length.

Bayardo and Miranker [10] generalize Ginsberg's proposal to i^{th}-order relevance-bounded nogood recording. In their scheme a nogood is deleted once it contains more than i variable-value pairs that do not appear in the current set of assignments. Subsequent experiments compared unrestricted, size-bounded, and relevance-bounded nogood recording. All came to the conclusion that unrestricted nogood recording was too expensive, but differed on whether size-bounded or relevance-bounded was better. Baker [7], in experiments on random binary problems, concludes that CBJ with 2-order size-bounded nogood recording is the best tradeoff. Bayardo and Schrag [11, 12], in experiments on a variety of real-world and random SAT instances, conclude that DPLL-CBJ with 4-order relevance-bounded nogood recording is best overall. Marques-Silva and Sakallah [92], in experiments on real-world SAT instances, conclude that DPLL-CBJ with 20-order size-bounded nogood recording is the winner.

Beyond restricting the size of the database, additional techniques have been proposed for reducing the cost of updating and querying the database. One of the most important of these is "watch" literals [103]. Given a set of assignments, the nogood database must tell the backtracking search algorithm whether any nogood is contradicted and whether any value can be pruned from the domain of a variable. Watch literals are a data structure for greatly reducing the number of nogoods that must be examined to answer these queries and reducing the cost of examining large nogoods.

With the discovery of the watch literals data structure, it was found that recording very large nogoods could lead to remarkable reductions in search time. Moskewicz et al. [103] show that 100- and 200-order relevance-bounded nogood recording with watch literals, along with restarts and a variable ordering based on the recorded nogoods, was significantly faster than DPLL-CBJ alone on large real-world SAT instances. Katsirelos and Bacchus [77] show that unrestricted generalized nogood recording with watch literals was significantly faster than MAC and MAC-CBJ alone on a variety of CSP instances from planning, crossword puzzles, and scheduling.

4.5 Non-Chronological Backtracking

Upon discovering a deadend, a backtracking algorithm must retract some previously posted branching constraint. In the standard form of backtracking, called chronological backtracking, only the most recently posted branching constraint is retracted. However, backtracking chronologically may not address the reason for the deadend. In non-chronological backtracking, the algorithm backtracks to and retracts the closest branching constraint which bears some responsibility for the deadend. Following Gaschnig [48], I refer to this process as *backjumping*[4].

Non-chronological backtracking algorithms can be described as a combination of (i) a strategy for discovering and using nogoods for backjumping, and (ii) a strategy for deleting nogoods from the nogood database.

4.5.1 Backjumping

Stallman and Sussman [124] were the first to informally propose a non-chronological backtracking algorithm—called dependency-directed backtracking—that discovered and maintained nogoods in order to backjump. Informal descriptions of backjumping are also given by Bruynooghe [22] and Rosiers and Bruynooghe [114]. The first explicit backjumping algorithm was given by Gaschnig [48]. Gaschnig's backjumping algorithm (BJ) [48] is similar to BT, except that it backjumps from deadends. However, BJ only backjumps from a deadend node when all the branches out of the node are leaves; otherwise it chronologically backtracks. Dechter [34] proposes a graph-based backjumping algorithm which computes the backjump points based on the static structure of the CSP. The idea is to jump back to the most recent variable that shares a constraint with the deadend variable. The algorithm was the first to also jump back at internal deadends.

Prosser [108] proposes the conflict-directed backjumping algorithm (CBJ), a generalization of Gaschnig's BJ to also backjump from internal deadends. Equivalent algorithms were independently proposed and formalized by Schiex and Verfaillie [118] and Ginsberg [54]. Each of these algorithms uses a variation of the jumpback nogood (Definition 4.5) to decide where to safely backjump to in the search tree from a deadend. Suppose that the backtracking algorithm has discovered a non-leaf deadend $p = \{b_1, \ldots, b_j\}$ in the search tree. The algorithm must backtrack by retracting some branching constraint from p. Chronological backtracking would choose b_j. Let $J(p) \subseteq p$ be the jumpback nogood for p. Backjumping chooses the largest i, $1 \leq i \leq j$, such that $b_i \in J(p)$. This is the backjump point. The algorithm jumps back in the search tree and retracts b_i, at the same time retracting any branching constraints that were posted after b_i and deleting any nogoods that were recorded after b_i.

As examples of applying CBJ and BJ, consider the backtracking tree shown in Figure 4.1. The light-shaded part of the tree contains nodes that are skipped by Conflict-Directed Backjumping (CBJ). The algorithm discovers a deadend after failing to extend node 25314. As shown earlier, the jumpback nogood associated with this node is $\{x_1 = 2, x_2 = 5, x_3 = 3, x_5 = 4\}$. CBJ backtracks to and retracts the most recently posted branching constraint, which is $x_5 = 4$. No nodes are skipped at this point. The remaining

[4] Backjumping is also referred to as intelligent backtracking and dependency-directed backtracking in the literature.

two values for x_5 also fail. The algorithm has now discovered that 2531 is a deadend node and, because a jumpback nogood has been determined for each branch, the jumpback nogood of 2531 is easily found to be $\{x_1 = 2, x_2 = 5, x_3 = 3\}$. CBJ backjumps to retract $x_3 = 3$ skipping the rest of the subtree. The backjump is represented by a dashed arrow. In contrast to CBJ, BJ only backjumps from deadends when all branches out of the deadend are leaves. The dark-shaded part of the tree contains two nodes that are skipped by Backjumping (BJ). Again, the backjump is represented by a dashed arrow.

In the same way as for dynamic backtracking (DBT), when all branching constraints are of the form $x = a$, for some variable x and value a, CBJ can be implemented using $O(n^2d)$ space, where n is the number of variables and d is the size of the domains. The data structure maintains a nogood for each variable and value pair and each nogood is $O(n)$ in length. However, since CBJ only uses the recorded nogoods for backjumping and constraints corresponding to the nogoods are never checked or propagated, it is not necessary to actually store a nogood for each value. A simpler $O(n^2)$ data structure, sometimes called a conflict set, suffices. The conflict set stores, for each variable, the union of the nogoods for each value of the variable.

CBJ has also been combined with constraint propagation. The basic backjumping mechanism is the same for all algorithms that perform non-chronological backtracking, no matter what level of constraint propagation is performed. The main difference lies in how the jumpback nogood is constructed (see Section 4.4.1 and Definition 4.7). Prosser [108] proposes FC-CBJ, an algorithm that combines forward checking constraint propagation and conflict-directed backjumping. An equivalent algorithm as independently proposed and formalized by Schiex and Verfaillie [118]. An informal description of an algorithm that combines forward checking and backjumping is also given by Rosiers and Bruynooghe [114]. Prosser [109] proposes MAC-CBJ, an algorithm that combines maintaining arc consistency and conflict-directed backjumping. As specified, the algorithm only handles binary constraints. Chen [25] generalizes the algorithm to non-binary constraints.

Many experiments studies on conflict-directed backjumping have been reported in the literature. Many of these are summarized in Section 4.10.1.

4.5.2 Partial Order Backtracking

In chronological backtracking and conflict-directed backjumping, it is assumed that the branching constraints at a node $p = \{b_1, \ldots, b_j\}$ in the search tree are totally ordered. The total ordering is the order in which the branching constraints were posted by the algorithm. Chronological backtracking then always retracts b_j, the last branching constraint in the ordering, and backjumping chooses the largest i, $1 \leq i \leq j$, such that b_i is in the jumpback nogood.

Bruynooghe [22] notes that this is not a necessary assumption and proposes partial order backtracking. In partial ordering backtracking the branching constraints are considered initially unordered and a partial order is induced upon jumping back from deadends. Assume a d-way branching strategy, where all branching constraints are an assignment of a value to a variable. When jumping back from a deadend, an assignment $x = a$ must be chosen from the jumpback nogood and retracted. Bruynooghe notes that backjumping must respect the current partial order, and proposes choosing *any* assignment that is maximal in the partial order. Upon making this choice and backjumping, the partial order must now be further restricted. Recall that a nogood $\{x_1 = a_1, \ldots, x_k = a_k\}$ can be written in

the form $((x_1 = a_1) \wedge \cdots \wedge (x_{k-1} = a_{k-1})) \Rightarrow (x_k \neq a_k)$. The assignment $x = a$ chosen to be retracted must now appear on the right-hand side of any nogoods in which it appears. Adding an implication restricts the partial order as the assignments on the left-hand side of the implication must come before the assignment on the right-hand side. And if the retracted assignment $x = a$ appears on the left-hand side in any implication, that implication is deleted and the value on the right-hand side is restored to its domain. Deleting an implication relaxes the partial order. Rosiers and Bruynooghe [114] show, in experiments on hard (non-binary) word sum problems, that their partial order backtracking algorithm was the best choice over algorithms that did forward checking, backjumping, or a combination of forward checking and backjumping. However, Baker [7] gives an example (the example is credited to Ginsberg) showing that, because in Bruynooghe's scheme *any* assignment that is maximal in the partial order can be chosen, it is possible for the algorithm to cycle and never terminate.

Ginsberg proposes [54] the dynamic backtracking algorithm (DBT, see Table 4.1). DBT can be viewed as a formalization and correction of Bruynooghe's scheme for partial order backtracking. To guarantee termination, DBT always chooses from the jumpback nogood the most recently posted assignment and puts this assignment on the right-hand side of the implication. Thus, DBT maintains a total order over the assignments in the jumpback nogood and a partial order over the assignments not in the jumpback nogood. As a result, given the same jumpback nogood, the backjump point for DBT would be the same as for CBJ. However, in contrast to CBJ which upon backjumping retracts any nogoods that were posted after the backjump point, DBT retains these nogoods (see Section 4.4.2 for further discussion of the nogood retention strategy used in DBT). Ginsberg [54] shows, in experiments which used crossword puzzles as a test bed, that DBT can solve more problems within a fixed time limit than a backjumping algorithm. However, Baker [7] shows that relevance-bounded nogood recording, as used in DBT, can interact negatively with a dynamic variable ordering heuristic. As a result, DBT can also degrade performance—by an exponential amount—over an algorithm that does not retain nogoods such as CBJ.

Dynamic backtracking (DBT) has also been combined with constraint propagation. Jussien, Debruyne, and Boizumault [75] show how to integrate DBT with forward checking and maintaining arc consistency, to give FC-DBT and MAC-DBT, respectively. As with adding constraint propagation to CBJ, the main difference lies in how the jumpback nogood is constructed (see Section 4.4.1 and Definition 4.7). However, because of the retention of nogoods, there is an additional complexity when adding constraint propagation to DBT that is not present in CBJ. Consider a value in the domain of a variable that has been removed but its eliminating explanation is now irrelevant. The value cannot just be restored, as there may exist another relevant explanation for the deleted value; i.e., there may exist several ways of removing a value through constraint propagation.

Ginsberg and McAllester [56] propose an algorithm called partial order dynamic backtracking (PBT). PBT offers more freedom than DBT in the selection of the assignment from the jumpback nogood to put on the right-hand side of the implication, while still giving a guarantee of correctness and termination. In Ginsberg's DBT and Bruynooghe's partial order algorithm, deleting an implication relaxes the partial order. In PBT, the idea is to retain some of the partial ordering information from these deleted implications. Now, choosing any assignment that is maximal in the partial order is correct. Bliek [18] shows that PBT is not a generalization of DBT and gives an algorithm that does generalize both PBT and DBT. To date, no systematic evaluation of either PBT or Bliek's generalization

have been reported, and no integration with constraint propagation has been reported.

4.6 Heuristics for Backtracking Algorithms

When solving a CSP using backtracking search, a sequence of decisions must be made as to which variable to branch on or instantiate next and which value to give to the variable. These decisions are referred to as the variable and the value ordering. It has been shown that for many problems, the choice of variable and value ordering can be crucial to effectively solving the problem (e.g., [5, 50, 55, 63]).

A variable or value ordering can be either static, where the ordering is fixed and determined prior to search, or dynamic, where the ordering is determined as the search progresses. Dynamic variable orderings have received much attention in the literature. They were proposed as early as 1965 [57] and it is now well-understood how to incorporate a dynamic ordering into an arbitrary tree-search algorithm [5].

Given a CSP and a backtracking search algorithm, a variable or value ordering is said to be *optimal* if the ordering results in a search that visits the fewest number of nodes over all possible orderings when finding one solution or showing that there does not exist a solution. (Note that I could as well have used some other abstract measure such as the amount of work done at each node, rather than nodes visited, but this would not change the fundamental results.) Not surprisingly, finding optimal orderings is a computationally difficult task. Liberatore [87] shows that simply deciding whether a variable is the first variable in an optimal variable ordering is at least as hard as deciding whether the CSP has a solution. Finding an optimal value ordering is also clearly at least as hard since, if a solution exists, an optimal value ordering could be used to efficiently find a solution. Little is known about how to find optimal orderings or how to construct polynomial-time approximation algorithms—algorithms which return an ordering which is guaranteed to be near-optimal (but see [70, 85]). The field of constraint programming has so far mainly focused on heuristics which have no formal guarantees.

Heuristics can be either application-independent, where only generic features common to all CSPs are used, or application-dependent. In this survey, I focus on application-independent heuristics. Such heuristics have been quite successful and can provide a good starting point when designing heuristics for a new application. The heuristics I present leave unspecified which variable or value to choose in the case of ties and the result is implementation dependent. These heuristics can often be dramatically improved by adding additional features for breaking ties. However, there is no one best variable or value ordering heuristic and there will remain problems where these application-independent heuristics do not work well enough and a new heuristic must be designed.

Given that a new heuristic is to be designed, several alternatives present themselves. The heuristic can, of course, be hand-crafted either using application-independent features (see [36] for a summary of many features from which to construct heuristics) or using application-dependent features. As one example of the latter, Smith and Cheng [122] show how an effective heuristic can be designed for job shop scheduling given deep knowledge of job shop scheduling, the CSP model, and the search algorithm. However, such a combination of expertise can be scarce.

An alternative to hand-crafting a heuristic is to automatically adapt or learn a heuristic. Minton [98] presents a system which automatically specializes generic variable and value

ordering heuristics from a library to an application. Epstein et al. [36] present a system which learns variable and value ordering heuristics from previous search experience on problems from an application. The heuristics are combinations from a rich set of primitive features. Bain, Thornton, and Sattar [6] show how to learn variable ordering heuristics for optimization problems using evolutionary algorithms.

As a final alternative, if only relatively weak heuristics can be discovered for a problem, it has been shown that the technique of randomization and restarts can boost the performance of problem solving (see Section 4.7). Cicirello and Smith [27] discuss alternative methods for adding randomization to heuristics and the effect on search efficiency. Hulubei and O'Sullivan [70] study the relationship between the strength of the variable and value ordering heuristics and the need for restarts.

4.6.1 Variable Ordering Heuristics

Suppose that the backtracking search is attempting to extend a node p. The task of the variable ordering heuristic is to choose the next variable x to be branched on.

Many variable ordering heuristics have been proposed and evaluated in the literature. These heuristics can, with some omissions, be classified into two categories: heuristics that are based primarily on the domain sizes of the variables and heuristics that are based on the structure of the CSP.

Variable ordering heuristics based on domain size

When solving a CSP using backtracking search interleaved with constraint propagation, the domains of the unassigned variables are pruned using the constraints and the current set of branching constraints. Many of the most important variable ordering heuristics are based on the current domain sizes of the unassigned variables.

Definition 4.8 (remaining values)**.** *Let $rem(x \mid p)$ be the number of values that remain in the domain of variable x after constraint propagation, given a set of branching constraints p.*

Golomb and Baumert [57] were the first to propose a dynamic ordering heuristic based on choosing the variable with the smallest number of values remaining in its domain. The heuristic, hereafter denoted dom, is to choose the variable x that minimizes,

$$rem(x \mid p),$$

where x ranges over all unassigned variables. Of course, the heuristic makes sense no matter what level of constraint propagation is being performed during the search. In the case of algorithms that do not perform constraint propagation but only check constraints which have all their variables instantiated, define $rem(x \mid p)$ to contain only the values which satisfy all the relevant constraints. Given that our backtracking search algorithm is performing constraint propagation, which in practice it will be, the dom heuristic can be computed very efficiently. The dom heuristic was popularized by Haralick and Elliott [63], who showed that dom with the forward checking algorithm was an effective combination.

Much effort has gone into understanding this simple but effective heuristic. Intriguingly, Golomb and Baumert [57], when first proposing dom, state that from an information-theoretic point of view, it can be shown that on average choosing the variable with the smallest domain size is more efficient, but no further elaboration is provided. Haralick and Elliott [63] show analytically that dom minimizes the depth of the search tree, assuming a simplistic probabilistic model of a CSP and assuming that we are searching for all solutions using a forward checking algorithm. Nudel [105], shows that dom is optimal (it minimizes the number of nodes in the search tree) again assuming forward checking but using a slightly more refined probabilistic model. Gent et al. [52] propose a measure called kappa whose intent is to capture "constrainedness" and what it means to choose the most constrained variable first. They show that dom (and dom+deg, see below) can be viewed as an approximation of this measure.

Hooker [66], in an influential paper, argues for the scientific testing of heuristics—as opposed to competitive testing—through the construction of empirical models designed to support or refute the intuition behind a heuristic. Hooker and Vinay [67] apply the methodology to the study of the Jeroslow-Wang heuristic, a variable ordering heuristic for SAT. Surprisingly, they find that the standard intuition, that "a [heuristic] performs better when it creates subproblems that are more likely to be satisfiable," is refuted whereas a newly developed intuition, that "a [heuristic] works better when it creates simpler subproblems," is confirmed. Smith and Grant [120] apply the methodology to the study of dom. Haralick and Elliott [63] proposed an intuition behind the heuristic called the fail-first principle: "to succeed, try first where you are most likely to fail". Surprisingly, Smith and Grant find that if one equates the fail-first principle with minimizing the depth of the search tree, as Haralick and Elliott did, the principle is refuted. In follow on work, Beck et al. [14] find that if one equates the fail-first principle with minimizing the number of nodes in the search tree, as Nadel did, the principle is confirmed. Wallace [132], using a factor analysis, finds two basic factors behind the variation in search efficiency due to variable ordering heuristics: immediate failure and future failure.

In addition to the effort that has gone into understanding dom, much effort has gone into improving it. Brélaz [20], in the context of graph coloring, proposes a now widely used generalization of dom. Let the degree of an unassigned variable x be the number of constraints which involve x and at least one other unassigned variable. The heuristic, hereafter denoted dom+deg, is to choose the variable with the smallest number of values remaining in its domain and to break any ties by choosing the variable with the highest degree. Note that the degree information is dynamic and is updated as variables are instantiated. A static version, where the degree information is only computed prior to search, is also used in practice.

Bessière and Régin [17] propose another generalization of dom. The heuristic, hereafter denoted dom/deg, is to divide the domain size of a variable by the degree of the variable and to choose the variable which has the minimal value. The heuristic is shown to work well on random problems. Boussemart et al. [19] propose to divide by the weighted degree, hereafter denoted dom/wdeg. A weight, initially set to one, is associated with each constraint. Every time a constraint is responsible for a deadend, the associated weight is incremented. The weighted degree is the sum of the weights of the constraints which involve x and at least one other unassigned variable. The dom/wdeg heuristic is shown to work well on a variety of problems. As an interesting aside, it has also been shown empirically that arc consistency propagation plus the dom/deg or the dom/wdeg heuristic can

reduce or remove the need for backjumping on some problems [17, 84].

Gent et al. [50] propose choosing the variable x that minimizes,

$$rem(x \mid p) \prod_{C} (1 - t_C),$$

where C ranges over all constraints which involve x and at least one other unassigned variable and t_C is the fraction of assignments which do not satisfy the constraint C. They also propose other heuristics which contain the product term in the above equation. A limitation of all these heuristics is the requirement of an updated estimate of t_C for each constraint C as the search progresses. This is clearly costly, but also problematic for intensionally represented constraints and non-binary constraints. As well, the product term implicitly assumes that the probability a constraint fails is independent, an assumption that may not hold in practice.

Brown and Purdom [21] propose choosing the variable x that minimizes,

$$rem(x \mid p) + \min_{y \neq x} \left\{ \sum_{a \in rem(x|p)} rem(y \mid p \cup \{x = a\}) \right\},$$

where y ranges over all unassigned variables. The principle behind the heuristic is to pick the variable x that is the root of the smallest 2-level subtree. Brown and Purdom show that the heuristic works better than dom on random SAT problems as the problems get larger. However, the heuristic has yet to be thoroughly evaluated on hard SAT problems or general CSPs.

Geelen [49] proposes choosing the variable x that minimizes,

$$\sum_{a \in dom(x)} \prod_{y} rem(y \mid p \cup \{x = a\}),$$

where y ranges over all unassigned variables. The product term can be viewed as an upper bound on the number of solutions given a value a for x, and the principle behind the heuristic is said to be to choose the most "constrained" variable. Geelen shows that the heuristic works well on the n-queens problem when the level of constraint propagation used is forward checking. Refalo [111] proposes a similar heuristic and shows that it is much better than dom-based heuristics on multi-knapsack and magic square problems. Although the heuristic is costly to compute, Refalo's work shows that it can be particularly useful in choosing the first, or first few variables, in the ordering. Interestingly, Wallace [132] reports that on random and quasigroup problems, the heuristic does not perform well.

Freeman [38], in the context of SAT, proposes choosing the variable x that minimizes,

$$\sum_{a \in dom(x)} \sum_{y} rem(y \mid p \cup \{x = a\}),$$

where y ranges over all unassigned variables. Since this is an expensive heuristic, Freeman proposes using it primarily when choosing the first few variables in the search. The principle behind the heuristic is to maximize the amount of propagation and the number of variables which become instantiated if the variable is chosen, and thus simplify the remaining problem. Although costly to compute, Freeman shows that the heuristic works well on

hard SAT problems when the level of constraint propagation used is unit propagation, the equivalent of forward checking. Malik et al. [91] show that a truncated version (using just the first element in $dom(x)$) is very effective in instruction scheduling problems.

Structure-guided variable ordering heuristics

A CSP can be represented as a graph. Such graphical representations form the basis of structure-guided variable ordering heuristics. Real problems often do contain much structure and on these problems the advantages of structure-guided heuristics include that structural parameters can be used to bound the worst-case of a backtracking algorithm and structural goods and nogoods can be recorded and used to prune large parts of the search space. Unfortunately, a current limitation of these heuristics is that they can break down in the presence of global constraints, which are common in practice. A further disadvantage is that some structure-guided heuristics are either static or nearly static.

Freuder [40] may have been the first to propose a structure-guided variable ordering heuristic. Consider the constraint graph where there is a vertex for each variable in the CSP and there is an edge between two vertices x and y if there exists a constraint C such that both $x \in vars(C)$ and $y \in vars(C)$.

Definition 4.9 (width)**.** *Let the vertices in a constraint graph be ordered. The width of an ordering is the maximum number of edges from any vertex v to vertices prior to v in the ordering. The width of a constraint graph is the minimum width over all orderings of that graph.*

Consider the static variable ordering corresponding to an ordering of the vertices in the graph. Freuder [40] shows that the static variable ordering is backtrack-free if the level of strong k-consistency is greater than the width of the ordering. Clearly, such a variable ordering is within an $O(d)$ factor of an optimal ordering, where d is the size of the domain. Freuder [40] also shows that there exists a backtrack-free static variable ordering if the level of strong consistency is greater than the width of the constraint graph. Freuder [41] generalizes these results to static variable orderings which guarantee that the number of nodes visited in the search can be bounded *a priori*.

Dechter and Pearl [35] propose a variable ordering which first instantiates variables which cut cycles in the constraint graph. Once all cycles have been cut, the constraint graph is a tree and can be solved quickly using arc consistency [40]. Sabin and Freuder [117] refine and test this proposal within an algorithm that maintains arc consistency. They show that, on random binary problems, a variable ordering that cuts cycles can out perform dom+deg.

Zabih [136] proposes choosing a static variable ordering with small bandwidth. Let the n vertices in a constraint graph be ordered 1, ..., n. The bandwidth of an ordering is the maximum distance between any two vertices in the ordering that are connected by an edge. The bandwidth of a constraint graph is the minimum bandwidth over all orderings of that graph. Intuitively, a small bandwidth ordering will ensure that variables that caused the failure will be close by and thus reduce the need for backjumping. However, there is currently little empirical evidence that this is an effective heuristic.

A well-known technique in algorithm design on graphs is divide-and-conquer using graph separators.

Definition 4.10 (separator)**.** *A separator of a graph is a subset of the vertices or the edges which, when removed, separates the graph into disjoint subgraphs.*

A graph can be recursively decomposed by successively finding separators of the resulting disjoint subgraphs. Freuder and Quinn [42] propose a variable ordering heuristic based on a such a recursive decomposition. The idea is that the separators (called cutsets in [42]) give groups of variables which, once instantiated, decompose the CSP. Freuder and Quinn also propose a special-purpose backtracking algorithm to correctly use the variable ordering to get additive behavior rather than multiplicative behavior when solving the independent problems. Huang and Darwiche [69] show that the special-purpose backtracking algorithm is not needed; one can just use CBJ. Because the separators are found prior to search, the pre-established variable groupings never change during the execution of the backtracking search. However, Huang and Darwiche note that within these groupings the variable ordering can be dynamic and any one of the existing variable ordering heuristics can used. Li and van Beek [86] present several improvements to this divide-and-conquer approach. So far the divide-and-conquer approach has been shown to be effective on hard SAT problems [69, 86], but there has as yet been no systematic evaluation of the approach on general CSP problems.

As two final structure-guided heuristics, Moskewicz et al. [103], in their Chaff solver for SAT, propose that the choice of variable should be biased towards variables that occur in recently recorded nogoods. Jégou and Terrioux [73] use a tree-decomposition of the constraint graph to guide the variable ordering.

4.6.2 Value Ordering Heuristics

Suppose that the backtracking search is attempting to extend a node p and the variable ordering heuristic has chosen variable x to be branched on next. The task of the value ordering heuristic is to choose the next value a for x. The principle being followed in the design of many value ordering heuristics is to choose next the value that is most likely to succeed or be a part of a solution. Value ordering heuristics have been proposed which are based on either estimating the number of solutions or estimating the probability of a solution, for each choice of value a for x. Clearly, if we knew either of these properties *exactly*, then a perfect value ordering would also be known—simply select a value that leads to a solution and avoid a value that does not lead to a solution.

Dechter and Pearl [35] propose a static value ordering heuristic based on approximating the number of solutions to each subproblem. An approximation of the number of solutions is found by forming a tree relaxation of the problem, where constraints are dropped until the constraint graph of the CSP can be represented as a tree. Counting all solutions to a tree-structured CSP is polynomial and thus can be computed exactly. The values are then ordered by decreasing estimates of the solution counts. Followup work [76, 94, 131] has focused on generalizing the approach to dynamic value orderings and on improving the approximation of the number of solutions (the tree relaxation can provide a poor estimate of the true solution count) by using recent ideas from Bayesian networks. A limitation of this body of work is that, while it compares the number of solutions, it does not take into account the size of the subtree that is being branched into or the difficultly or cost of searching the subtree.

Ginsberg et al. [55], in experiments which used crossword puzzles as a test bed, propose the following dynamic value ordering heuristic. To instantiate x, choose the value $a \in dom(x)$ that maximizes the *product* of the remaining domain sizes,

$$\prod_y rem(y \mid p \cup \{x = a\}),$$

where y ranges over all unassigned variables. Ginsberg et al. show that the heuristic works well on crossword puzzles when the level of constraint propagation used is forward checking. Further empirical evidence for the usefulness of this heuristic was provided by Geelen [49]. Geelen notes that the product gives the number of possible completions of the node p and these completions can be viewed in two ways. First, assuming that every completion is equally likely to be a solution, choosing the value that maximizes the product also maximizes the probability that we are branching into a subproblem that contains a solution. Second, the completions can be viewed as an upper bound on the number of solutions to the subproblem. Frost and Dechter [46] propose choosing the value that maximizes the *sum* of the remaining domain sizes. However, Geelen [49] notes that the product differentiates much better than summation. In the literature, the product heuristic is sometimes called the "promise" heuristic and the summation heuristic is sometimes called the "min-conflicts" heuristic—as it was inspired by a local search heuristic of the same name proposed by Minton et al. [99].

4.7 Randomization and Restart Strategies

It has been widely observed that backtracking algorithms can be brittle on some instances. Seemingly small changes to a variable or value ordering heuristic, such as a change in the ordering of tie-breaking schemes, can lead to great differences in running time. An explanation for this phenomenon is that ordering heuristics make mistakes. Depending on the number of mistakes and how early in the search the mistakes are made (and therefore how costly they may be to correct), there can be a large variability in performance between different heuristics. A technique called randomization and restarts has been proposed for taking advantage of this variability.

The technique of randomization and restarts within backtracking search algorithms goes back at least to the PhD work of Harvey [64]. Harvey found that periodically restarting a backtracking search with different variable orderings could eliminate the problem of "early mistakes". This observation led Harvey to propose randomized backtracking algorithms where on each run of the backtracking algorithm the variable or the value orderings are randomized. The backtracking algorithm terminates when either a solution has been found or the distance that the algorithm has backtracked from a deadend exceeds some fixed cutoff. In the latter case, the backtracking algorithm is restarted and the search begins anew with different orderings. Harvey shows that this randomize and restart technique gives improved performance over a deterministic backtracking algorithm on job shop scheduling problems. Gomes et al. [60, 61, 62] have done much to popularize and advance the technique through demonstrations of its wide applicability, drawing connections to closely related work on Las Vegas algorithms, and contributions to our understanding of when and why restarts help.

In the rest of this section, I first survey work on the technique itself and then survey work that addresses the question of when do restarts help. For more on the topic of randomization and restart strategies see, for example, the survey by Gomes [58].

4.7.1 Algorithmic Techniques

The technique of randomization and restarts requires a method of adding randomization to a deterministic backtracking algorithm and a restart strategy, a schedule or method for deciding when to restart.

Randomization

Several possible methods of adding randomization to backtracking algorithms have been proposed in the literature. Harvey [64] proposes randomizing the *variable* ordering. Gomes et al. [61, 62] propose randomizing the variable ordering heuristic either by randomized tie breaking or by ranking the variables using an existing heuristic and then randomly choosing a variable from the set of variables that are within some small factor of the best variable. They show that restart strategies with randomized variable orderings lead to orders of magnitude improvement on a wide variety of problems including both SAT and CSP versions of scheduling, planning, and quasigroup completion problems. Cicirello and Smith [27] discuss alternative methods for adding randomization to heuristics and the effect on search efficiency. Other alternatives are to choose a variable with a probability that is proportional to the heuristic weight of the variable or to randomly pick from among a suite of heuristics. One pitfall to be aware of is that the method of adding randomization to the heuristic must give enough different decisions near the top of the search tree. Harvey [64] proposes randomizing the *value* ordering so that each possible ordering is equally likely. As well, all the options listed above for randomizing variable orderings are also options for value orderings. Zhang [138] argues that randomizing a heuristic can weaken it, an undesirable effect. Prestwich [106] and Zhang [138] propose a random backwards jump in the search space upon backtracking. Although effective, this has the consequence that the backtracking algorithm is no longer complete.

Restart strategies

A restart strategy $S = (t_1, t_2, t_3, ...)$ is an infinite sequence where each t_i is either a positive integer or infinity. The idea is that the randomized backtracking algorithm is run for t_1 steps. If no solution is found within that cutoff, the algorithm is run for t_2 steps, and so on. A *fixed cutoff* strategy is a strategy where all the t_i are equal. Various restart strategies have been proposed.

Luby, Sinclair, and Zuckerman [88] (hereafter just Luby) examine restart strategies in the more general setting of Las Vegas algorithms. A Las Vegas algorithm is a randomized algorithm that always gives the correct answer when it terminates, however the running time of the algorithm varies from one run to another and can be modeled as a random variable. Let $f(t)$ be the probability that a backtracking algorithm $\mathcal{A}$ applied to instance x stops after taking exactly t steps. Let $F(t)$ be the cumulative distribution function of f; i.e., the probability that $\mathcal{A}$ stops after taking t or fewer steps. $F(t)$ is sometimes referred to as the runtime distribution of algorithm $\mathcal{A}$ on instance x. The tail probability is the

probability that $\mathcal{A}$ stops after taking more than t steps; i.e., $1 - F(t)$, which is sometimes referred to as the survival function. Luby shows that, given full knowledge of the runtime distribution, the optimal strategy is given by $S_{t^*} = (t^*, t^*, t^*, \ldots)$, for some fixed cutoff t^*. Of course, the runtime distribution is not known in practice. For the case where there is no knowledge of the runtime distribution, Luby shows that a universal strategy given by $S_u = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, \ldots)$ is within a log factor of the optimal strategy S_{t^*} and that this is the best performance that can be achieved up to a constant factor by any universal strategy. Further, Luby proves that, no matter what the runtime distribution of the original algorithm $\mathcal{A}$, if we apply $\mathcal{A}$ using restart strategy S_{t^*} or S_u, the tail probability of the restart strategy is small as it decays exponentially.

To use a restart strategy in practice, one must decide what counts as a primitive operation or step in the computation. Several methods have been used in the literature. Harvey [64] uses a fixed cutoff strategy which restarts the backtracking algorithm when the distance that the algorithm has backtracked from a deadend exceeds some fixed cutoff. Richards [112] restarts at every deadend, but maintains completeness by first recording a nogood so the deadend is not revisited. Gomes et al. [61] use a fixed cutoff strategy that restarts the backtracking algorithm when the number of backtracks exceeds some fixed cutoff. Kautz et al. [78, 115] use the number of nodes visited by the backtracking algorithm. For a fixed cutoff strategy, one must also decide what cutoff to use. So far it appears that good cutoffs are specific to an instance. Thus one must perform some sort of trial-and-error search for a good cutoff value. However, van Moorsel and Wolter [130] observe that for some runtime distributions a wide range of cutoffs perform well. They further observe that it is often safer to make the cutoff too large rather than too small. For the universal strategy, one does not need to decide the cutoff. However, it has been reported that the universal strategy is slow in practice as the sequence increases too slowly (e.g., [61, 78, 115]). Note that this does not contradict the fact that the universal strategy is within a log factor of optimal, since this is an asymptotic result and ignores constant factors. However, it may also be noted that one can scale the universal strategy $S_u = (s, s, 2s, \ldots)$, for some scale factor s, and possibly improve performance while retaining the optimality guarantee.

Walsh [134] proposes a universal strategy $S_g = (1, r, r^2, \ldots)$, where the restart values are geometrically increasing, and shows that values of r in the range $1 < r < 2$ work well on the problems examined. The strategy has the advantage that it increases more quickly than the universal strategy but avoids the search for a cutoff necessary for a fixed cutoff strategy. Although it appears to work well in practice, unfortunately the geometric strategy comes with no formal guarantees for its worst-case performance. It can be shown that the expected runtime of the geometric strategy can be arbitrarily worse than that of the optimal strategy.

Kautz et al. [78, 115] (hereafter just Kautz) observe that Luby makes two assumptions when proving the optimality of S_{t^*} that may not hold in practice. The assumptions are (i) that successive runs of the randomized algorithm are statistically independent and identically distributed, and (ii) that the only feasible observation or feature is the length of a run. As an example of where the first assumption may be false, consider the case where the current instance is drawn from one of two distributions but we do not know which. The failure to find a solution in previous runs can change our belief about the runtime distribution of the current instance. To show that the second assumption may be false, Kautz shows that a Bayesian model based on a rich set of features can with sufficient accuracy predict the runtime of the algorithm on the current instance. Kautz removes these assumptions and

proposes context-sensitive or dynamic restart strategies. In one set of experiments, Kautz shows that a dynamic strategy can do *better* than the static optimal strategy S_{t^*}. The strategy uses a Bayesian model to predict whether a current run of the algorithm will be either "long" or "short", and restarts if the prediction is "long".

Van Moorsel and Wolter [130] consider a case that often arises in practice where a solution is useful only if it is found within some deadline; i.e., we are given a deadline c and we may run the restart strategy until a total of c steps of the algorithm have been executed. Van Moorsel and Wolter consider restart strategies that maximize the probability the deadline is met.

4.7.2 When Do Restarts Help?

The question of when and why the technique of randomization and restarts is useful has been addressed from two angles: For what kinds of runtime distributions are restarts helpful and what are the underlying causes for these runtime distributions.

Runtime distributions for which restarts are useful

In the case where restarts are helpful on satisfiable instances, Gomes et al. [61, 62] show that probability distributions with heavy-tails can be a good fit to the runtime distributions of backtracking algorithms with randomized heuristics. A heavy-tailed distribution is one where the tail probability or survival function (see above) decays polynomially; i.e., there is a significant probability that the backtracking algorithm will run for a long time. For unsatisfiable instances, Gomes et al. [61] report that in their experiments on random quasigroup completion problems, heavy-tailed behavior was not found and that restarts were consequently not helpful on these problems. As an interesting aside, Baptista and Marques-Silva [8] show experimentally that—because of synergy between the techniques—a backtracking algorithm that incorporates nogood recording can benefit from randomization and restarts when solving unsatisfiable instances.

Hoos [68] notes that restarts will not only be effective for heavy tails, but that its effectiveness depends *solely* on there existing some point where the cumulative runtime distribution is increasing slower than the exponential distribution. It is at this point, where the search is stagnating, that a restart would be helpful.

Van Moorsel and Wolter [129] provide necessary and sufficient conditions for restarts to be helpful. Their work can be seen as a formalization of Hoos' insight and its extension from one restart to multiple restarts. Let T be a random variable which models the runtime of a randomized backtracking algorithm on an instance and let $E[T]$ be the expected value of T. Under the assumption that successive runs of the randomized algorithm are statistically independent and identically distributed, Van Moorsel and Wolter show that *any* number of restarts using a fixed cutoff of t steps is better than just letting the algorithm run to completion if and only if,

$$E[T] < E[T - t \mid T > t]$$

holds; i.e., if and only if the expected runtime of the algorithm is less than the expected remaining time to completion given that the algorithm has run for t steps. Van Moorsel and Wolter also show that if a single restart improves the expected runtime, multiple restarts

perform even better, and unbounded restarts performs best. For what kinds of distributions does the above condition hold? Restarts will be most effective (the inequality will be greatest) for heavy-tailed distributions, where the tail decays polynomially, but Van Moorsel and Wolter observe that the condition also hold for some distributions where the tail decays exponentially. For other exponentially decaying distributions, restarting will be strictly worse than running the algorithm to completion. Zhan [137] shows that this is not an isolated case and for many problems restarts can be harmful. For pure exponential distributions, the condition is an equality and restarts will be neither helpful or harmful.

Underlying causes for these runtime distributions

Various theories have been postulated for explaining why restarts are helpful; i.e., why do runtime distributions arise where restarts are helpful. It is superficially agreed that an explanation for this phenomenon is that ordering heuristics make mistakes which require the backtracking algorithm to explore large subtrees with no solutions. However, the theories differ in what it means for an ordering heuristic to make a mistake.

Harvey [64] defines a mistake as follows.

Definition 4.11 (value mistake)**.** *A mistake is a node in the search tree that is a nogood but the parent of the node is not a nogood.*

When a mistake is made, the search has branched into a subproblem that does not have a solution. The result is that the node has to be refuted and doing this may require a large subtree to be explored, especially if the mistake is made early in the tree. In this definition, value ordering heuristics make mistakes, variable ordering heuristics do not. However, changing the variable ordering can mean either that a mistake is not made, since the value ordering is correct for the newly chosen variable, or that any mistake is less costly to correct. Harvey constructs a probabilistic model to predict when a restart algorithm will perform better than its deterministic counterpart. With simplifying assumptions about the probability of a mistake, it is shown that restarts are beneficial when the mistake probability is small. Clearly, the definition, and thus the probabilistic model on which it depends, only applies if a CSP has a solution. Therefore, the theory does not explain when restarts would be beneficial for unsatisfiable problems.

As evidence in support of this theory, Hulubei and O'Sullivan [70] consider the distribution of refutation sizes to correct mistakes (the size of the subtrees that are rooted at mistakes). They show that when using a poor value ordering in experiments on quasigroup completion problems, heavy-tailed behavior was observed for every one of four different high-quality variable ordering heuristics. However, the heavy-tailed behavior disappeared when the same experiments were performed but this time with a high-quality value ordering heuristic in place of the random value ordering.

Williams, Gomes, and Selman [135] (hereafter just Williams) define a mistake as follows.

Definition 4.12 (backdoor mistake)**.** *A mistake is a selection of a variable that is not in a minimal backdoor, when such a variable is available to be chosen.*

A backdoor is a set of variables for which there exists value assignments such that the simplified problem (such as after constraint propagation) can be solved in polynomial time. Backdoors capture the intuition that good variable and value ordering heuristics simplify

the problem as quickly as possible. When a mistake is made, the search has branched into a subproblem that has not been as effectively simplified as it would have been had it chosen a backdoor variable. The result is that the subproblem is more costly to search, especially if the mistake is made early in the tree. In this definition, variable ordering heuristics make mistakes, value ordering heuristics do not. Williams constructs a probabilistic model to predict when heavy-tailed behavior will occur but there will exist a restart strategy that will have polynomial expected running time. With simplifying assumptions about the probability of a mistake, it is shown that both of these occur when the probability of a mistake is sufficiently small and the size of the minimal backdoor is sufficiently small. The theory can also explain when restarts would be beneficial for unsatisfiable problems, through the notion of a strong backdoor. However, the theory does not entirely account for the fact that a random value ordering together with a restart strategy can remove heavy-tail behavior. In this case the variable ordering remains fixed and so the probability of a mistake also remains unchanged.

Finally, some work contributes to our understanding of why runtime distributions arise where restarts are helpful while remaining agnostic about the exact definition of a mistake. Consider the probability distribution of refutation sizes to correct mistakes. It has been shown both empirically on random problems and through theoretical, probabilistic models that heavy-tails arise in the case where this distribution decays exponentially as the size of the refutation grows [24, 59]. In other words, there is an exponentially decreasing probability of making a costly (exponentially-sized) mistake.

4.8 Best-First Search

In the search tree that is traversed by a backtracking algorithm, the branches out of a node are assumed to be ordered by a value ordering heuristic, with the left-most branch being the most promising (or at least no less promising than any branch to the right). The backtracking algorithm then performs a depth-first traversal of the search tree, visiting the branches out of a node in left-to-right order. When a CSP instance is unsatisfiable and the entire search tree must be traversed, depth-first search is the clear best choice. However, when it is known or it can safely be assumed that a CSP instance is satisfiable, alternative search strategies such as best-first search become viable. In this section, I survey discrepancy-based search strategies, which can be viewed as variations on best-first search.

Harvey and Ginsberg [64, 65] were the first to propose a discrepancy-based search strategy, in an algorithm called limited discrepancy search. A discrepancy is the case where the search does not follow the value ordering heuristic and does not take the left-most branch out of a node. The idea behind limited discrepancy search is to iteratively search the tree by increasing number of discrepancies, preferring discrepancies that occur near the root of the tree. This allows the search to recover from mistakes made early in the search (see Definition 4.11). In contrast, with backtracking (or depth-first) search, mistakes made near the root of the tree can be costly to discover and undo. On the i^{th} iteration, the limited discrepancy algorithm visits all leaf nodes with up to i discrepancies in the path from the root to the leaf. The algorithm terminates when a solution is found or the iteration is exhausted. Limited discrepancy search is called iteratively with $i = 0, 1, \ldots, k$. If $k \geq n$, where n is the depth of the search tree, the algorithm is complete; otherwise it is

incomplete. Harvey and Ginsberg show both theoretically and experimentally that limited discrepancy search can be better than depth-first search on satisfiable instances when a good value ordering heuristic is available.

Korf [80] proposes a modification to the limited discrepancy algorithm so that it visits fewer duplicate nodes on subsequent iterations. On the i^{th} iteration, Korf's algorithm visits all leaf nodes with *exactly* i discrepancies in the path from the root to the leaf. However, to achieve these savings, Korf's algorithm prefers discrepancies deeper in the tree. Korf notes that limited discrepancy search can be viewed as a variation on best-first search, where the cost of a node p is the number of discrepancies in the path from the root of the search tree to p. In best-first search, the node with the lowest cost is chosen as the next node to be extended. In Harvey and Ginsberg's proposal, ties for lowest cost are broken by choosing a node that is *closest* to the root. In Korf's proposal, ties are broken by choosing a node that is *farthest* from the root.

Walsh [133] (and independently Meseguer [95]), argues that value ordering heuristics tend to be less informed and more prone to make mistakes near the top of the search tree. Walsh proposes depth-bounded discrepancy search, which biases search to discrepancies near the top of the tree, but visits fewer redundant nodes than limited discrepancy search. On the i^{th} iteration, the depth-bounded discrepancy search algorithm visits all leaf nodes where all discrepancies in the path from the root to the leaf occur at depth i or less. Meseguer [95] proposes interleaved depth-first search, which also biases search to discrepancies near the top of the tree. The basic idea is to divide up the search time on the branches out of a node using a variation of round-robin scheduling. Each branch—or more properly, each subtree rooted at a branch—is searched for a given time-slice using depth-first search. If no solution is found within the time slice, the search is suspended and the next branch becomes active. Upon suspending search in the last branch, the first branch again becomes active. This continues until either a solution is found or all the subtrees have been exhaustively searched. The strategy can be applied recursively within subtrees.

Meseguer and Walsh [96] experimentally compare backtracking algorithms using traditional depth-first search and the four discrepancy-based search strategies described above. On a test bed which consisted of random binary, quasigroup completion, and number partitioning CSPs, it was found that discrepancy-based search strategies could be much better than depth-first search. As with randomization and restarts, discrepancy-based search strategies are a way to overcome value ordering mistakes made early in the search.

4.9 Optimization

In some important application areas of constraint programming such as scheduling, sequencing and planning, CSPs arise which have, in addition to constraints which must be satisfied, an objective function f which must be optimized. Without loss of generality, I assume in what follows that the goal is to find a solution which minimizes f and that f is a function over all the variables of the CSP. I also assume that a variable c has been added to the CSP model and constrained to be equal to the objective function; i.e., $c = f(X)$, where X is the set of variables in the CSP. I call this the objective constraint.

To solve optimization CSPs, the common approach is to find an optimal solution by solving a sequence of CSPs; i.e., a sequence of satisfaction problems. Several variations have been proposed and evaluated in the literature. Van Hentenryck [128] proposes what

can be viewed as a constraint-based version of branch-and-bound. Initially, a backtracking search is used to find any solution p which satisfies the constraints. A constraint is then added to the CSP of the form $c < f(S)$ which excludes solutions that are not better than this solution. A new solution is then found for the augmented CSP. This process is repeated until the resulting CSP is unsatisfiable, in which case the last solution found has been proven optimal. Baptiste, Le Pape, and Nuijten [9] suggest iterating on the possible values of c by either (i) iterating from the smallest value in $dom(c)$ to the largest until a solution is found, (ii) iterating from largest to smallest until a solution is no longer found, or (iii) performing binary search. Each time, of course, we are solving a satisfaction problem using a backtracking search algorithm. For these approaches to be effective, it is important that constraint propagation techniques be applied to the objective constraint. For example, see [9, Chapter 5] for propagation techniques for objective constraints for several objective functions that arise in scheduling.

4.10 Comparing Backtracking Algorithms

As this survey has indicated, many improvements to backtracking have been proposed and there are many ways that these techniques can be combined together into one algorithm. In this section, I survey work on comparing the performance of backtracking algorithms. The work is categorized into empirical and theoretical approaches. Both approaches have well-known advantages and disadvantages. Empirical comparisons allow the comparison of any pair of backtracking algorithms, but any conclusion about which algorithm is better will always be weak since it must be qualified by the phrase, "on the instances we examined". Theoretical comparisons allow categorical statements about the relative performance of some pairs of backtracking algorithms, but the requirement that any conclusion be true for all instances means that statements cannot be made about every pair of algorithms and the statements that can be made must sometimes be necessarily weak.

When comparing backtracking algorithms, several performance measures have been used. For empirical comparisons, of course runtime can be used, although this requires one to be sure that one is comparing the underlying algorithms and not implementation skill or choice of programming language. Three widely used performance measures that are implementation independent are number of constraint checks, backtracks, and nodes visited.

4.10.1 Empirical Comparisons

Early work in empirical comparisons of backtracking algorithms was hampered by a lack of realistic or hard test problems (e.g., [21, 48, 63, 93, 108, 114]). The experimental test bed often consisted of only toy problems—the ubiquitous n-queens problem first used in 1965 [57] was still being used as a test bed more than 20 years later [125]—and simple random problems. As well, often only CSPs with binary constraints were experimented upon. The focus on simple, binary CSPs was sometimes detrimental to the field and led to promising approaches being prematurely dismissed.

The situation improved with the discovery of hard random problems that arise at a phase transition and the investigation of alternative random models of CSPs (see [51] and references therein). Experiments could now be performed which compared the algorithms

on the hardest problems and systematically explored the entire space of random problems to see where one algorithm bettered another (e.g., [17, 45, 126]). Unfortunately, most of the random models lack any structure or realism. The situation was further improved by the realization that important applications of constraint programming are often best modeled using global constraints and other non-binary constraints, and the construction and subsequent wide use of a constraint programming benchmark library [53].

In the remainder of this section, I review two representative streams of experiments: experiments that examine what level of constraint propagation a backtracking algorithm should perform and experiments that examine the interaction between several techniques for improving a backtracking algorithm. Many other experiments—such as those performed by authors who have introduced a new technique and then show that the technique works better on a selected set of test problems—are reported elsewhere in this survey.

Experiments on level of constraint propagation

Experiments have examined the question of what level of local consistency should be maintained during the backtracking search. Consider the representative set of experiments summarized in Table 4.2. Gaschnig [47] originally proposed interleaving backtracking search with arc consistency. Early experiments which tested this proposal concluded that an algorithm that maintained arc consistency during the search was not competitive with forward checking [48, 63, 93].

This view was maintained for about fifteen years until it was challenged by Sabin and Freuder. Sabin and Freuder [116], using hard random problems, showed that MAC could be much better than forward checking. More recently, with an increasing emphasis on applying constraint programming in practice, has come an understanding of the importance of global constraints and other intensionally represented non-binary constraints for modeling real problems. With such constraints, special purpose constraint propagation algorithms are developed and the modeler has a choice of what level of constraint propagation to enforce. It is now generally accepted that the choice of level of constraint propagation depends on the application and different choices may be made for different constraints within the same CSP.

Table 4.2: *Experiments on constraint propagation: MAC vs FC.*

	Faster?	Comment
McGregor (1979) [93]	FC	3 $\times$ faster
Haralick & Elliott (1980) [63]	FC	3 $\times$ faster
Sabin & Freuder (1994) [116]	MAC	much better
Bacchus & van Run (1995) [5]	FC	3–20 $\times$ faster
Bessière & Régin (1996) [17]	MAC	much better
Larrosa (2000) [81]	both	much better

Experiments on the interaction between improvements

Experiments have examined the interaction of the quality of the variable ordering heuristic, the level of local consistency maintained during the backtracking search, and the addition of backjumping techniques such as conflict-directed backjumping (CBJ) and dynamic backtracking (DBT). Unfortunately, these three techniques for improving a backtracking algorithm are not entirely orthogonal. Consider the representative set of experiments summarized in Table 4.3. These experiments show that, if the variable ordering is fixed and the level of constraint propagation is forward checking, conflict-directed backjumping is an effective technique. However, it can also be observed in previous experimental work that as the level of local consistency that is maintained in the backtracking search is increased and as the variable ordering heuristic is improved, the effects of CBJ are diminished [5, 17, 107, 108]. For example, it can be observed in Prosser's [108] experiments that, given a static variable ordering, increasing the level of local consistency maintained from none to the level of forward checking, diminishes the effects of CBJ. Bacchus and van Run [5] observe from their experiments that adding a dynamic variable ordering (an improvement over a static variable ordering) to a forward checking algorithm diminishes the effects of CBJ. In their experiments the effects are so diminished as to be almost negligible and they present an argument for why this might hold in general. Bessière and Régin [17] observe from their experiments that simultaneously increasing the level of local consistency even further to arc consistency and further improving the dynamic variable ordering heuristic diminishes the effects of CBJ so much that, in their implementation, the overhead of maintaining the data structures for backjumping actually slows down the algorithm. They conjecture that when arc consistency is maintained and a good variable ordering heuristic is used, "CBJ becomes useless". All of the above experiments were on small puzzles—the Zebra problem and n-queens problem—and on random CSPs which lacked any structure.

In contrast, in subsequent experiments on both random and real-world *structured* CSPs, backjumping was found to be a useful technique. Jussien, Debruyne, Boizumault [75] present empirical results that show that adding dynamic backtracking to an algorithm that maintains arc consistency can greatly improve performance. Chen and van Beek [26] present empirical results that show that, although the effects of CBJ may be diminished, adding CBJ to a backtracking algorithm that maintains arc consistency can still provide orders of magnitude speedups. Finally, CBJ is now a standard technique in the best backtracking algorithms for solving structured SAT problems [83].

Table 4.3: *Experiments on backjumping: FC vs FC-CBJ.*

	Faster?	Comment
Rosiers and Bruynooghe (1987) [114]	FC-CBJ	never worse
Prosser (1993) [108]	FC-CBJ	three times better
Frost & Dechter (1994) [45]	FC-CBJ	somewhat better
Bacchus & van Run (1995) [5]	FC-CBJ	slightly
Smith & Grant (1995) [119]	FC-CBJ	sometimes much better
Bayardo & Schrag (1996, 1997) [11, 12]	FC-CBJ	much better

4.10.2 Theoretical Comparisons

Worst-case analysis and average-case analysis are two standard theoretical approaches to understanding and comparing algorithms. Unfortunately, neither approach has proven generally successful for comparing backtracking algorithms. The worst-case bounds of backtracking algorithms are always exponential and rarely predictive of performance, and the average-case analyses of backtracking algorithms have, by necessity, made simplifying and unrealistic assumptions about the distribution of problems (e.g., [63, 105, 110]).

Two alternative approaches that have proven more successful for comparing algorithms are techniques based on proof complexity and a methodology for constructing partial orders based on characterizing properties of the nodes visited by an algorithm.

Proof complexity and backtracking algorithms

Backtracking algorithms can be compared using techniques from the proof complexity of resolution refutation proofs. The results that can be proven are of the general form: Given any CSP instance, algorithm $\mathcal{A}$ with an optimal variable and value ordering never visits fewer and can visit exponentially more nodes when applied to the instance than algorithm $\mathcal{B}$ with an optimal variable and value ordering. The optimal orderings are relative to the algorithms and thus $\mathcal{A}$ and $\mathcal{B}$ may use different orderings. I begin by briefly explaining resolution refutation proofs and proof complexity, followed by an explanation of some results of applying proof complexity techniques to the study of backtracking algorithms for CSPs.

The resolution inference rule takes two premises in the form of clauses $(A \vee x)$ and $(B \vee \neg x)$ and gives the clause $(A \vee B)$ as a conclusion. The two premises are said to be resolved and the variable x is said to be resolved away. Resolving the two clauses x and $\neg x$ gives the empty clause. Given a set of input clauses $\mathcal{F}$, a resolution proof or derivation of a clause C is a sequence of applications of the resolution inference rule such that C is the final conclusion and each premise in each application of the inference rule is either a clause from $\mathcal{F}$ or a conclusion from a previous application of the inference rule. A resolution proof that derives the empty clause is called a refutation proof, as it shows that the input set of clauses $\mathcal{F}$ is unsatisfiable.

A resolution proof of a clause C can be viewed as a directed acyclic graph (DAG). Each leaf node in the DAG is labeled with a clause from $\mathcal{F}$, each internal node is labeled with a derived clause that is justified by resolving the clauses of its two parents, and there is a single node with no successors and the label of that node is C. Many restrictions on the form of the proof DAG have been studied. For our purposes, one will suffice. A tree resolution proof is a resolution proof where the DAG of inferences forms a tree. The size of a proof is the number of nodes (clauses) in the proof DAG.

Proof complexity is the study of the size of the *smallest* proof a method can produce [28]. It is known that the smallest tree resolution refutation proof to show a set of clauses $\mathcal{F}$ is unsatisfiable can be exponentially larger than the smallest unrestricted resolution refutation proof and can never be smaller (see [13] and references therein). To see why tree proofs can be larger, note that if one wishes to use a derived clause elsewhere in the proof it must be re-derived. To see why tree proofs can never be smaller, note that every tree resolution proof is also an unrestricted resolution proof.

Why is resolution refutation proof complexity interesting for the study of backtracking algorithms? The search tree that results from applying a complete backtracking algorithm to an unsatisfiable CSP can be viewed as a resolution refutation proof. As an example of the correspondence, consider the backtracking tree that results from applying BT to the SAT problem which consists of the set of clauses $\{a \vee b \vee c, a \vee \neg c, \neg b, \neg a \vee c, b \vee \neg c\}$. Each leaf node is labeled with the clause that caused the failure, interior nodes are labeled by working from the leaves to the root applying the resolution inference rule, and the root will be labeled with the empty clause. Thus, proof complexity addresses the question of the size of the smallest possible backtrack tree; i.e., the size of the backtrack tree *if one assumes optimal variable and value ordering heuristics*.

The connection between backtracking algorithms for SAT and resolution has been widely observed and it is known that DPLL-based algorithms on unsatisfiable instances correspond to tree resolution refutation proofs. Baker [7] shows how to generalize this correspondence to the backtracking algorithm BT for general CSPs, when BT is using d-way branching. Mitchell [100], using earlier work by de Kleer [32], shows how to generalize this correspondence when BT is using 2-way branching.

Beame, Kautz, and Sabharwal [13] (hereafter Beame) use proof complexity techniques to investigate backtracking algorithms with nogood recording. Let DPLL be a basic backtracking algorithm for SAT, let DPLL+nr be DPLL with a nogood recording scheme (called FirstNewCut) added, and let DPLL+nr+restarts be DPLL with nogood recording and infinite restarts added. Beame shows that the smallest refutation proofs using DPLL can be exponentially longer than the smallest refutation proofs using DPLL+nr. In other words, DPLL with an optimal variable and value ordering never visits fewer and can visit exponentially more nodes than DPLL with nogood recording and an optimal variable and value ordering. Beame also shows that DPLL+nr+restarts is equivalent to unrestricted resolution if the learned nogoods are retained between restarts. It is an open question whether DPLL+nr is equivalent to unrestricted resolution or falls strictly between unrestricted resolution and tree resolution proofs.

Hwang and Mitchell [71] use proof complexity techniques to investigate backtracking algorithms with different branching strategies. Let BT-2-way be a basic backtracking algorithm for general CSPs using 2-way branching, and let BT-d-way be a backtracking algorithm using d-way branching. Hwang and Mitchell show that BT-d-way with an optimal variable and value ordering never visits fewer and can visit exponentially more nodes than BT-2-way with an optimal variable and value ordering.

Although a powerful technique, a limitation of the proof complexity framework is that it cannot be used to distinguish between some standard improvements to the basic chronological backtracking algorithm. For example, consider the four combinations of adding or not adding unit propagation and conflict-directed backjumping to DPLL. When using an optimal variable and value ordering each algorithm visits exactly the same number of nodes. Similar results hold for adding conflict-directed backjumping, dynamic backtracking, or forward checking to BT [7, 26, 100].

A partial order on backtracking algorithms

Backtracking algorithms can be compared by formulating necessary and sufficient conditions for a search tree node to be visited by each backtracking algorithm. These characterizations can then be used to construct a partial order (or hierarchy) on the algorithms

according to two standard performance measures: the number of nodes visited and the number of constraint checks performed.

The results that can be proven are of the general form: Given any CSP instance and any variable and value ordering, algorithm $\mathcal{A}$ with the variable and value ordering never visits more nodes (and may visit fewer) when applied to the instance than algorithm $\mathcal{B}$ with the same variable and value ordering. In other words, algorithm $\mathcal{A}$ dominates algorithm $\mathcal{B}$ when the performance measure is nodes visited. A strong feature of this approach is that the results still hold ($\mathcal{A}$ still dominates $\mathcal{B}$), even if the CSP model used by both algorithms is the model that is best from algorithm $\mathcal{B}$'s point of view and even if the variable and value ordering used by both algorithms are the orderings that are best (optimal) from algorithm $\mathcal{B}$'s point of view.

Kondrak and van Beek [79] introduce the general methodology and give techniques and definitions that can be used for characterizing backtracking algorithms. Using the methodology, they formulate necessary and sufficient conditions for several backtracking algorithms including BT, BJ, CBJ, FC, and FC-CBJ. As an example of a necessary condition, it can be shown that if FC visits a node, then the parent of the node is 1-consistent (see Definition 4.3). As an example of a sufficient condition, it can be shown that if the parent of a node is 1-consistent, then BJ visits the node. The necessary and sufficient conditions can then be used to order the two backtracking algorithms. For example, to show that FC dominates BJ in terms of nodes visited, we show that every node that is visited by FC is also visited by BJ. The necessary condition for FC is used to deduce that the parent of the node is 1-consistent. Since the parent of the node is 1-consistent, the sufficient condition for BJ can then be used to conclude that BJ visits the node.

Chen and van Beek [26] extend the partial ordering of backtracking algorithms to include backtracking algorithms and their CBJ hybrids that maintain levels of local consistency beyond forward checking, including the algorithms that maintain arc consistency. To analyze the influence of the level of local consistency on the backjumping, Chen and van Beek use the notion of *backjump level*. Informally, the level of a backjump is the distance, measured in backjumps, from the backjump destination to the farthest deadend. By classifying the backjumps performed by a backjumping algorithm into different levels, CBJ is weakened into a series of backjumping algorithms which perform limited levels of backjumps. Let BJ$_k$ be a backjumping algorithm which backjumps if the backjump level j is less than or equal to k, but chronologically backtracks if $j > k$. BJ$_n$ is equivalent to CBJ, which performs unlimited backjumps, and BJ$_1$ is equivalent to Gaschnig's [48] BJ, which only does first level backjumps. Recall that the maintaining strong k-consistency algorithm (MC$_k$) enforces strong k-consistency at each node in the backtrack tree, where MC$_1$ is equivalent to FC and on binary CSPs MC$_2$ is equivalent to MAC. MC$_k$ can be combined with backjumping, namely MC$_k$-CBJ. Chen and van Beek show that an algorithm that maintains strong k-consistency never visits more nodes than a backjumping algorithm that is allowed to backjump at most k levels. Thus, as the level of local consistency that is maintained in the backtracking search is increased, the less that backjumping will be an improvement.

Figure 4.3 shows a partial order or hierarchy in terms of the size of the backtrack tree for BJ$_k$, MC$_k$, and MC$_k$-CBJ. If there is a path from algorithm $\mathcal{A}$ to algorithm $\mathcal{B}$ in the figure, $\mathcal{A}$ never visits more nodes than $\mathcal{B}$. For example, for all variable orderings, MC$_k$ never visits more nodes than BJ$_j$, for all $j \leq k$.

Bacchus and Grove [3] observe that the partial orderings with respect to nodes visited

can be extended to partial orderings with respect to constraint checks, or other measures of the amount of work performed at each node. For example, on binary CSPs the MAC algorithm can perform $O(n^2d^2)$ work at each node of the tree, where n is the number of variables and d is the size of the domain, whereas the FC algorithm can perform $O(nd)$ work. Thus, one can conclude that on binary CSPs MAC can be at most $O(nd)$ times slower in the worst case (when the two algorithms visit the same nodes). The partial orderings with respect to nodes and constraint checks are consistent with and explain some of the empirical results reported in the literature (e.g., see the experiments reported in Tables 4.2 & 4.3).

Besides the relationships that are shown explicitly, it is important to note the ones that are implicit in the hierarchy. If there is *not* a path from algorithm $\mathcal{A}$ to algorithm $\mathcal{B}$ in the hierarchy, $\mathcal{A}$ and $\mathcal{B}$ are incomparable. To show a pair of algorithms $\mathcal{A}$ and $\mathcal{B}$ are incomparable, one needs to find a CSP and a variable ordering on which $\mathcal{A}$ is better than $\mathcal{B}$, and one on which $\mathcal{B}$ is better than $\mathcal{A}$. Examples have been given that cover all the incomparability results [4, 26, 79]. Some of the more surprising results include: CBJ and FC-CBJ are incomparable [79], CBJ and MC$_k$ are incomparable for any fixed $k < n$ in that each can be exponentially better than the other [4], and MAC-CBJ and FC-CBJ and more generally MC$_k$-CBJ and MC$_{k+1}$-CBJ are incomparable for any fixed $k < n$ in that each can be exponentially better than the other [26].

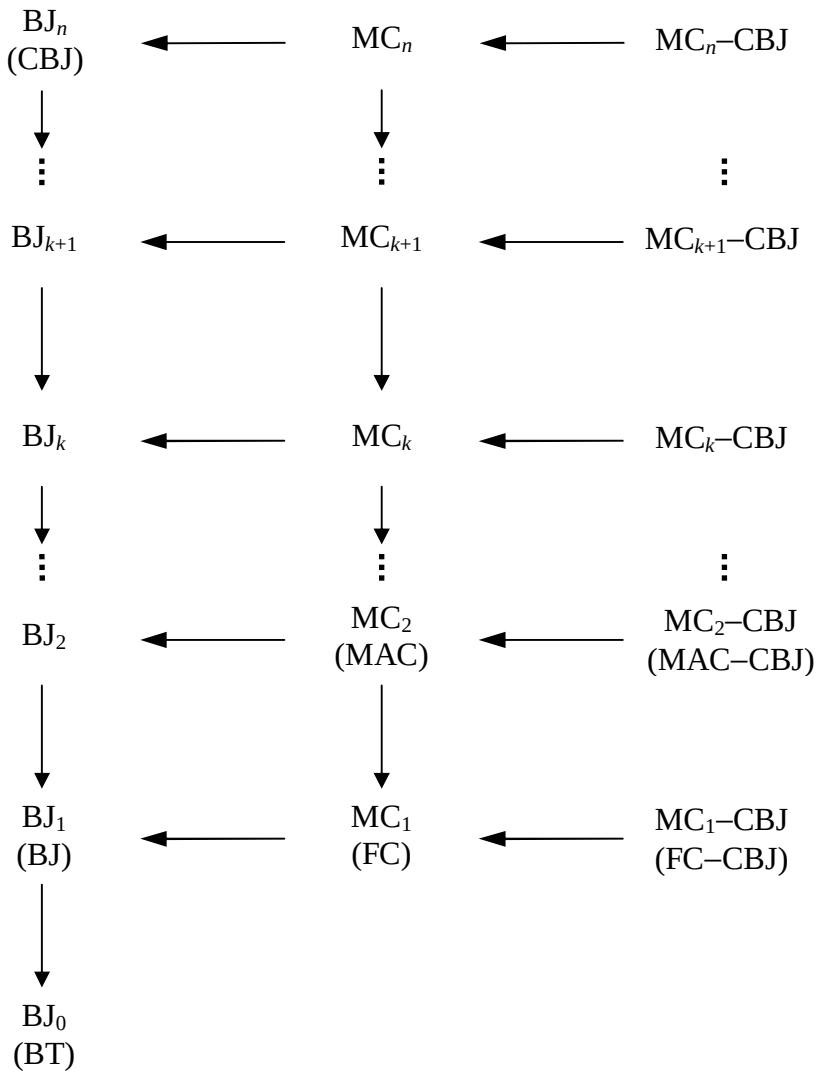

Figure 4.3: A hierarchy for BJ$_k$, MC$_k$, and MC$_k$-CBJ in terms of the size of the backtrack tree (adapted from [26, 79]). On binary CSPs, MC$_2$ is equivalent to MAC and MC$_2$-CBJ is equivalent to MAC-CBJ.

Acknowledgements

I would like to thank Fahiem Bacchus, Xinguang Chen, Grzegorz Kondrak, Dennis Manchak, and Jonathan Sillito for many interesting discussions and collaborations in the past on backtracking algorithms. I would also like to thank Christian Bessière and George Katsirelos for helpful comments on an early draft of this survey. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

Bibliography

[1] F. Bacchus. Extending forward checking. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 35–51, Singapore, 2000.

[2] F. Bacchus. Enhancing Davis Putman with extended binary clause reasoning. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 613 – 619, Edmonton, 2002.

[3] F. Bacchus and A. Grove. On the forward checking algorithm. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 292–309, Cassis, France, 1995.

[4] F. Bacchus and A. Grove. Looking forward in constraint satisfaction algorithms. Unpublished manuscript, 1999.

[5] F. Bacchus and P. van Run. Dynamic variable ordering in CSPs. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 258–275, Cassis, France, 1995.

[6] S. Bain, J. Thornton, and A. Sattar. Evolving variable-ordering heuristics for constrained optimisation. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming*, pages 732–736, Sitges, Spain, 2005.

[7] A. B. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, University of Oregon, 1995.

[8] L. Baptista and J. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 489–494, Singapore, 2000.

[9] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer, 2001.

[10] R. J. Bayardo Jr. and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 298–304, Portland, Oregon, 1996.

[11] R. J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 46–60, Cambridge, Mass., 1996.

[12] R. J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-

world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 203–208, Providence, Rhode Island, 1997.

[13] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. of Artificial Intelligence Research*, 22:319–351, 2004. URL `http://www.jair.org`.

[14] J. C. Beck, P. Prosser, and R. J. Wallace. Trying again to fail first. In *Recent Advances in Constraints, Lecture Notes in Artificial Intelligence, Vol. 3419*. Springer-Verlag, 2005.

[15] U. Bertelè and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, 1972.

[16] C. Bessière, P. Meseguer, E. C. Freuder, and J. Larrosa. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141:205–224, 2002.

[17] C. Bessière and J.-C. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 61–75, Cambridge, Mass., 1996.

[18] C. Bliek. Generalizing partial order and dynamic backtracking. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 319–325, Madison, Wisconsin, 1998.

[19] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 146–150, Valencia, Spain, 2004.

[20] D. Brélaz. New methods to color the vertices of a graph. *Comm. ACM*, 22:251–256, 1979.

[21] C. A. Brown and P. W. Purdom Jr. An empirical comparison of backtracking algorithms. *IEEE PAMI*, 4:309–315, 1982.

[22] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12:36–39, 1981.

[23] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 369–383, Santa Margherita Ligure, Italy, 1994.

[24] H. Chen, C. Gomes, and B. Selman. Formal models of heavy-tailed behavior in combinatorial search. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 408–421, Paphos, Cyprus, 2001.

[25] X. Chen. *A Theoretical Comparison of Selected CSP Solving and Modeling Techniques*. PhD thesis, University of Alberta, 2000.

[26] X. Chen and P. van Beek. Conflict-directed backjumping revisited. *J. of Artificial Intelligence Research*, 14:53–81, 2001. URL `http://www.jair.org`.

[27] V. A. Cicirello and S. F. Smith. Amplification of search performance through randomization of heuristics. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 124–138, Ithaca, New York, 2002.

[28] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *J. Symbolic Logic*, 44:36–50, 1979.

[29] M. C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989.

[30] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. ACM*, 5:394–397, 1962.

[31] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.

[32] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 290–296, Detroit, 1989.

[33] R. Dechter. Learning while searching in constraint satisfaction problems. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 178–183, Philadelphia, 1986.

[34] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[35] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[36] S. L. Epstein, E. C. Freuder, R. J. Wallace, A. Morozov, and B. Samuels. The adaptive constraint engine. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 525–540, Ithaca, New York, 2002.

[37] R. E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[38] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[39] E. C. Freuder. Synthesizing constraint expressions. *Comm. ACM*, 21:958–966, 1978.

[40] E. C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29:24–32, 1982.

[41] E. C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32: 755–761, 1985.

[42] E. C. Freuder and M. J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1076–1078, Los Angeles, 1985.

[43] E. C. Freuder and R. J. Wallace. Generalizing inconsistency learning for constraint satisfaction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 563–571, Montréal, 1995.

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

[45] D. Frost and R. Dechter. In search of the best search: An empirical evaluation. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 301–306, Seattle, 1994.

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

[47] J. Gaschnig. A constraint satisfaction method for inference making. In *Proceedings Twelfth Annual Allerton Conference on Circuit and System Theory*, pages 866–874, Monticello, Illinois, 1974.

[48] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian*

Conference on Artificial Intelligence, pages 268–277, Toronto, 1978.

[49] P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 31–35, Vienna, 1992.

[50] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 179–193, Cambridge, Mass., 1996.

[51] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. *Constraints*, 6(4):345–372, 2001.

[52] I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 246–252, Portland, Oregon, 1996.

[53] I. P. Gent and T. Walsh. CSPlib: A benchmark library for constraints. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, pages 480–481, Alexandria, Virginia, 1999.

[54] M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1: 25–46, 1993. URL `http://www.jair.org`.

[55] M. L. Ginsberg, M. Frank, M. P. Halpin, and M. C. Torrance. Search lessons learned from crossword puzzles. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 210–215, Boston, Mass., 1990.

[56] M. L. Ginsberg and D. A. McAllester. GSAT and dynamic backtracking. In *Proceedings of the Second Workshop on Principles and Practice of Constraint Programming*, pages 243–265, Rosario, Orcas Island, Washington, 1994.

[57] S. Golomb and L. Baumert. Backtrack programming. *J. ACM*, 12:516–524, 1965.

[58] C. Gomes. Randomized backtrack search. In M. Milano, editor, *Constraint and Integer Programming: Toward a Unified Methodology*, pages 233–292. Kluwer, 2004.

[59] C. Gomes, C. Fernández, B. Selman, and C. Bessière. Statistical regimes across constrainedness regions. *Constraints*, 10:317–337, 2005.

[60] C. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 121–135, Linz, Austria, 1997.

[61] C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. of Automated Reasoning*, 24:67–100, 2000.

[62] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, Wisconsin, 1998.

[63] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[64] W. D. Harvey. *Nonsystematic backtracking search*. PhD thesis, Stanford University, 1995.

[65] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 607–613, Montréal, 1995.

[66] J. N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1:

33–42, 1996.

[67] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[68] H. H. Hoos. Heavy-tailed behaviour in randomised systematic search algorithms for SAT. Technical Report TR-99-16, UBC, 1999.

[69] J. Huang and A. Darwiche. A structure-based variable ordering heuristic for SAT. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1167–1172, Acapulco, Mexico, 2003.

[70] T. Hulubei and B. O'Sullivan. Search heuristics and heavy-tailed behaviour. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming*, pages 328–342, Sitges, Spain, 2005.

[71] J. Hwang and D. G. Mitchell. 2-way vs. d-way branching for CSP. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming*, pages 343–357, Sitges, Spain, 2005.

[72] ILOG S. A. ILOG Solver 4.2 user's manual, 1998.

[73] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.

[74] U. Junker. QuickXplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 167–172, 2004.

[75] N. Jussien, R. Debruyne, and B. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 249–261, Singapore, 2000.

[76] K. Kask, R. Dechter, and V. Gogate. Counting-based look-ahead schemes for constraint satisfaction. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, pages 317–331, Toronto, 2004.

[77] G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 390–396, Pittsburgh, 2005.

[78] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 674–681, Edmonton, 2002.

[79] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[80] R. E. Korf. Improved limited discrepancy search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 286–291, Portland, Oregon, 1996.

[81] J. Larrosa. Boosting search with variable elimination. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 291–305, Singapore, 2000.

[82] J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[83] D. Le Berre and L. Simon. Fifty-five solvers in Vancouver: The SAT 2004 competition. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004)*, pages 321–344, Vancouver, 2004. Available as: Springer Lecture Notes in Computer Science 3542, 2005.

[84] C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of the Sixteenth IEEE International Conference on Tools with Artificial Intelligence*, pages 549–557, Boca Raton, Florida, 2004.

[85] C.-M. Li and S. Gérard. On the limit of branching rules for hard random unsatisfiable 3-SAT. *Discrete Applied Mathematics*, 130:277–290, 2003.

[86] W. Li and P. van Beek. Guiding real-world SAT solving with dynamic hypergraph separator decomposition. In *Proceedings of the Sixteenth IEEE International Conference on Tools with Artificial Intelligence*, pages 542–548, Boca Raton, Florida, 2004.

[87] P. Liberatore. On the complexity of choosing the branching literal in DPLL. *Artificial Intelligence*, 116:315–326, 2000.

[88] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. In *Proceedings of the Second Israel Symposium on the Theory of Computing and Systems*, Jerusalem, 1993.

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

[90] A. K. Mackworth. On reading sketch maps. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 598–606, Cambridge, Mass., 1977.

[91] A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. Technical Report CS-2005-19, School of Computer Science, University of Waterloo, 2005.

[92] J. Marques-Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, Calif., 1996.

[93] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inform. Sci.*, 19:229–250, 1979.

[94] A. Meisels, S. E. Shimony, and G. Solotorevsky. Bayes networks for estimating the number of solutions to a CSP. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 185–190, Providence, Rhode Island, 1997.

[95] P. Meseguer. Interleaved depth-first search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1382–1387, Nagoya, Japan, 1997.

[96] P. Meseguer and T. Walsh. Interleaved and discrepancy based search. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 239–243, Brighton, UK, 1998.

[97] M. Milano and W. J. van Hoeve. Reduced cost-based ranking for generating promising subproblems. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 1–16, Ithaca, New York, 2002.

[98] S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1:7–44, 1996.

[99] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–206, 1992.

[100] D. G. Mitchell. Resolution and constraint satisfaction. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*,

pages 555–569, Kinsale, Ireland, 2003.

[101] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656, Munchen, Germany, 1988.

[102] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci.*, 7:95–132, 1974.

[103] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of 39th Design Automation Conference*, Las Vegas, 2001.

[104] B. A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5: 188–224, 1989.

[105] B. Nudel. Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics. *Artificial Intelligence*, 21:135–178, 1983.

[106] S. Prestwich. A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 337–352, Singapore, 2000.

[107] P. Prosser. Domain filtering can degrade intelligent backtracking search. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 262–267, Chambèry, France, 1993.

[108] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[109] P. Prosser. MAC-CBJ: Maintaining arc consistency with conflict-directed backjumping. Research Report 177, University of Strathclyde, 1995.

[110] P. W. Purdom Jr. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[111] P. Refalo. Impact-based search strategies for constraint programming. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, pages 557–571, Toronto, 2004.

[112] E. T. Richards. *Non-systematic Search and No-good Learning*. PhD thesis, Imperial College, 1998.

[113] G. Rochart, N. Jussien, and F. Laburthe. Challenging explanations for global constraints. In *CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS'03)*, pages 31–43, Kinsale, Ireland, 2003.

[114] W. Rosiers and M. Bruynooghe. Empirical study of some constraint satisfaction algorithms. In P. Jorrand and V. Sgurev, editors, *Artificial Intelligence II, Methodology, Systems, Applications, Proc. AIMSA'86*, pages 173–180. North Holland, 1987.

[115] Y. Ruan, E. Horvitz, and H. Kautz. Restart policies with dependence among runs: A dynamic programming approach. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 573–586, Ithaca, New York, 2002.

[116] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 125–129, Amsterdam, 1994.

[117] D. Sabin and E. C. Freuder. Understanding and improving the MAC algorithm. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 167–181, Linz, Austria, 1997.

[118] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3:1–15, 1994.

[119] B. M. Smith and S. A. Grant. Sparse constraint graphs and exceptionally hard problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 646–651, Montréal, 1995.

[120] B. M. Smith and S. A. Grant. Trying harder to fail first. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 249–253, Brighton, UK, 1998.

[121] B. M. Smith and P. Sturdy. Value ordering for finding all solutions. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 311–316, Edinburgh, 2005.

[122] S. F. Smith and C. Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 139–144, Washington, DC, 1993.

[123] G. Smolka. The OZ programming model. In *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 324–343, 1995.

[124] 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.

[125] H. S. Stone and J. M. Stone. Efficient search techniques—an empirical study of the N-queens problem. *IBM J. Res. and Develop.*, 31:464–474, 1987.

[126] E. P. K. Tsang, J. E. Borrett, and A. C. M. Kwan. An attempt to map the performance of a range of algorithm and heuristic combinations. In *Proceedings of the AI and Simulated Behaviour Conference*, pages 203–216, 1995.

[127] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, pages 559–586. American Mathematical Society, 1996.

[128] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[129] A. P. A. van Moorsel and K. Wolter. Analysis and algorithms for restart. In *Proceedings of the IEEE Quantitative Evaluation of Systems (QEST 2004)*, pages 195–204, Enschede, The Netherlands, 2004.

[130] A. P. A. van Moorsel and K. Wolter. Meeting deadlines through restart. In *Proceedings of the 12th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems*, pages 155–160, Dresden, Germany, 2004.

[131] M. Vernooy and W. S. Havens. An evaluation of probabilistic value-ordering heuristics. In *Proceedings of the Australian Conference on AI*, pages 340–352, Sydney, 1999.

[132] R. J. Wallace. Factor analytic studies of CSP heuristics. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming*, pages 712–726, Sitges, Spain, 2005.

[133] T. Walsh. Depth-bounded discrepancy search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1388–1393, Nagoya, Japan, 1997.

[134] T. Walsh. Search in a small world. In *Proceedings of the Sixteenth International*

Joint Conference on Artificial Intelligence, pages 1172–1177, Stockholm, 1999.

[135] R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1173–1178, Acapulco, Mexico, 2003.

[136] R. Zabih. Some applications of graph bandwidth to constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 46–51, Boston, Mass., 1990.

[137] Y. Zhan. Randomisation and restarts, 2001. MSc thesis, University of York.

[138] H. Zhang. A random jump strategy for combinatorial search. In *Proceedings of International Symposium on AI and Math*, Fort Lauderdale, Florida, 2002.

[139] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285, San Jose, Calif., 2001.