Backtracking algorithms for constraint satisfaction problems – a tutorial survey*

Rina Dechter and Daniel Frost
Department of Information and Computer Science
University of California, Irvine
Irvine, California, USA 92697-3425
{dechter,frost}@ics.uci.edu

April 20, 1998

Abstract

Over the past twenty years a number of backtracking algorithms for constraint satisfaction problems have been developed. This survey describes the basic backtrack search within the search space framework and then presents a number of improvements including look-back methods such as backjumping, constraint recording, backmarking, and look-ahead methods such as forward checking and dynamic variable ordering.

1 Introduction

Constraint networks have proven successful in modeling mundane cognitive tasks such as vision, language comprehension, default reasoning, and abduction, as well as specialized reasoning tasks including diagnosis, design, and temporal and spatial reasoning. The constraint paradigm can be considered a generalization of propositional logic, in that variables may be assigned values from a set with any number of elements not just TRUE and FALSE. This flexibility in the number of values can improve the ease and naturalness with which interesting problems are modeled.

The goal of this survey is to give a brief overview of several approaches to solving constraint satisfactions problems, and then to focus on the backtracking algorithm and its variants, which form the basis for many constraint solution procedures. The algorithms are presented abstractly, in that considerations of data structures and optimizations are avoided. Instead we are trying to elucidate

*This work was partially supported by NSF grant IRI-9157636, Air Force Office of Scientific Research grant AFOSR F49620-96-1-0224, Rockwell MICRO grants ACM-20775 and 95-043

the essence of each algorithm, and to show the relationship of each algorithm to the others.

Two general approaches to solving CSPs are search and deduction. Each is based on the idea of solving a hard problem by transforming it into an easier one. Search works in general by guessing an operation to perform, possibly with the aid of a heuristic [Nil80, Pea84]. A good guess results in a new state that is nearer to a goal. For CSPs, search is exemplified by the backtracking algorithm which uses two phases. The first extends a partial solution by assigning a value to one more variable. When no acceptable value can be found, the second operation, which is to *backtrack* or undo an assignment operation, is performed. The backtracking algorithm is time exponential, but requires only linear space.

Deduction in the CSP framework is known as constraint propagation or consistency enforcing [Mon74, Mac77, Fre82]. In this approach the problem is transformed into an equivalent but more explicit form. Because consistency enforcing algorithms can require exponential time and exponential space in the number of variables, they are usually applied to subsets of the variables. For example, path-consistency algorithms ensure that any consistent solution to a two-variable subnetwork is extendible to any third variable. In general, i-consistency algorithms ensure that any consistent instantiation of $i-1$ variables can be extended to a consistent value of any ith variable. A problem that is i-consistent for all i is called *globally* consistent.

In addition to backtracking search and constraint propagation, two other approaches are stochastic local search and structure-driven algorithms. Stochastic methods move in a hill-climbing manner augmented with random steps in the space of complete instantiations [SML90]. In the CSP community it is mostly recognized by the name GSAT [SLM92]. Structure-driven algorithms cut across both search and consistency-enforcing algorithms. These techniques emerged from an attempt to characterize the topology of constraint problems that are tractable. *Tractable classes* were generally recognized by realizing that enforcing low-level consistency (in polynomial time) guarantees global consistency for some problems. The basic graph structure that supports tractability is a tree [MF85]. In particular, enforcing 2-consistency on a tree-structured network ensures global consistency along some ordering.

Solving a constraint satisfaction problem is usually taken to mean finding one or more ways to assign a value to each variable, such that no constraint is violated. Backtracking, which traverses the space of partial solutions in a depth-first manner, is the standard search procedure for solving constraint satisfaction problems. The algorithm was first described more than a century ago, and since then has been reintroduced several times [BR75]. The backtracking algorithm typically considers the variables in some specified order. Starting with the first variable, the algorithm assigns a provisional value to each variable in turn as long as each assigned value is found to be consistent with values assigned in the past. When the algorithm encounters a variable for which none of the domain values is consistent with previous assignments, a *dead-end* occurs and backtracking

takes place. That is, the value assigned to the variable immediately preceding the dead-end variable is changed, and the search continues. The algorithm halts either when the required number of solutions has been found, or when it can be concluded that no solutions, or no more solutions, exist.

Much of the work in constraint satisfaction during the last decade has been devoted to improving the performance of backtracking search. Because the problem is known to be NP-complete [GJ79], polynomial variants of backtracking are unlikely. In fact, it can be shown that all backtracking algorithms are exponential in the worst-case. Nevertheless, the average performance of *naive* backtracking can be improved tremendously by augmenting it with simple heuristics.

The efficiency of backtracking can be improved by reducing the size of its *expanded* search space, which is determined both by the size of the *underlying* search space, and by the algorithm's control strategy. The size of the underlying search space depends on the way the constraints are represented (e.g. on the level of local consistency), the order of variable instantiation, and, when one solution suffices, the order in which values are assigned to each variable. Using these factors, researchers have developed procedures of two types: those employed before performing the search, thus bounding the size of the underlying search space; and those used dynamically *during* the search and that decide which parts of the search space will not be visited.

The pre-processing procedures include the *constraint propagation* or *local consistency* algorithms, such as arc-consistency, path-consistency, and i-consistency. Choosing the level of consistency that should be enforced on the network is not simple. In most cases, backtracking will work more efficiently on representations that are as explicit as possible, namely, those having a high level of local consistency. However, because the complexity of enforcing i-consistency is exponential in i, there is often a tradeoff between the effort spent on pre-processing and the effort spent on search [DM94]. Whether the tradeoff is ultimately beneficial depends on the character of the problem instance being solved.

Along with local consistency levels, the parameter that most affects the size of the underlying search space is *variable ordering*. Several heuristic orderings have been developed, [HE80, Fre82, DP89], all following the intuition that tightly constrained variables should come first.

The procedures for dynamically improving the pruning power of backtracking can be conveniently classified as *look-ahead schemes* and *look-back schemes*, in accordance with backtracking's two main phases of going forward to assemble a solution and going back in case of a dead-end. *Look-ahead* schemes can be invoked whenever the algorithm is preparing to assign a value to the next variable. The essence of these schemes is to discover from a restricted amount of constraint propagation how the current decisions about variable and value selection will restrict future search. Once a certain amount of forward constraint propagation is complete the algorithm can use the results to:

1. Decide which variable to instantiate next, if the order is not predeter-

mined. Generally, it is advantageous to first instantiate variables that maximally constrain the rest of the search space. Therefore, the most highly constrained variable is usually selected.

2. Decide which value to assign to the next variable when there is more than one candidate. Generally, when searching for a single solution an attempt is made to assign a value that maximizes the number of options available for future assignments.

Look-back schemes are invoked when the algorithm is preparing the backtracking step after encountering a dead-end. These schemes perform two functions:

1. Deciding how far to backtrack. By analyzing the reasons for the dead-end, irrelevant backtrack points can often be avoided so that the algorithm goes back directly to the source of failure, instead of just to the immediately preceding variable in the ordering. This procedure is often referred to as *backjumping*.

2. Recording the reasons for the dead-end in the form of new constraints, so that the same conflicts will not arise again later in the search. The terms used to describe this function are *constraint recording* and *learning*.

After providing the necessary definitions in Section 2, we embed constraint solving within the search space paradigm (Section 3) and present the basic backtracking algorithm (Section 4). Sections 5 through 7 describe in detail several algorithms, each one improving a different aspect of backtracking search. This collection of algorithms characterizes the main approaches presented in the literature, but is by no means complete. In section 8 we highlight more recent research which focus on combining the different improvement ideas into hybrid algorithms and present such hybrids. The final section provides historical remarks. Previous surveys on constraint processing as well as on backtracking algorithms can be found in [Dec92, Mac92, Kum92, Tsa93, KvB97].

2 Definitions

A *constraint network* $\mathcal{R}$ is a set of n variables $X = \{x_1, \ldots, x_n\}$, a set of value domains D_i for each variable x_i, and a set of constraints or relations. Each value domain is a finite set of values, one of which must be assigned to the corresponding variable. A *constraint* R_S over $S \subseteq X$ is a subset of the cartesian product of the domains of the variables in S. If $S = \{x_{i_1}, \ldots, x_{i_r}\}$, then $R_S \subseteq D_{i_1} \times \cdots \times D_{i_r}$. A *binary constraint network* is a constraint network in which all constraints are defined over pairs of variables. A *constraint graph* associates each variable with a node and connects any two nodes whose variables appear in the same constraint.

A variable is called *instantiated* when it is assigned a value from its domain, otherwise it is *uninstantiated*. By $x_i = a_i$ or by (x_i, a_i) we denote that variable x_i is instantiated with value a_i from its domain. An *instantiation* of all the variables in X is an n-tuple $a = (a_1, \ldots, a_n)$ representing an assignment of $a_i \in D_i$ to each x_i, $1 \leq i \leq n$.

We denote an assignment of a subset of variables, also called a *partial assignment*, by a tuple of ordered pairs $((x_1, a_1), ..., (x_1, a_i))$, frequently abbreviated to $(a_1, ..., a_i)$. Let Y and S be sets of variables, and let y be an instantiation of the variables in Y. We denote by y_S the tuple consisting of only the components of y that correspond to the variables in S. A partial instantiation is *consistent* if it satisfies all of the constraints that have no uninstantiated variables. Formally, an instantiation y is *consistent* if for all $S_i \subseteq Y$, such that if there is a constraint R_{S_i} $y_{S_i} \in R_{S_i}$. A consistent partial instantiation is also called a *partial solution*. A *solution* is an instantiation of all the variables that is *consistent*.

We next briefly discuss the concept of i-consistency that is central to constraint processing. Mackworth [Mac77] defined three properties of networks which characterize the local consistency of networks: *node-*, *arc-*, and *path-consistency*. Node-, arc-, and path-consistency correspond to 1-, 2-, and 3-consistency, respectively. Freuder [Fre78] generalized these to k-consistency. Dechter and Pearl [DP87] introduce the notion of *directional consistency*.

A network is 1-consistent if the values in the domain of each variable satisfy the network's unary constraints. A network is *k-consistent*, $k \geq 2$, iff given any partial instantiation of any $k-1$ distinct variables which is consistent, there exists an instantiation of any kth variable such that the k values taken together are consistent [Fre78]. Given an ordering of the variables, the network is *directional* k-consistent iff any subset of $k-1$ variables is k-consistent relative to variables that succeed the $k-1$ variables in the ordering [DP87].

When the network does not possess the desired level of local consistency, local-consistency and directional local-consistency algorithms can be applied. Since enforcing i-consistency is exponential in i, usually only low levels of local consistency are enforced. In particular, varying levels of arc-consistency can be embeded dynamically into search. The last decade had seen an extensive study of i-consistency (and in particular, arc-consistency) algorithms [MF85, MH86, Coo89, VHDT92, DP87]. The reader interested in a fuller description of consistency enforcing algorithms may consult the above references.

3 The Search Space

In this paper, we focus on solving constraint satisfaction problems with algorithms that are systematic and that work by extending a consistent partial solution into a full solution. Such algorithms can be viewed as traversing a state space graph whose nodes (the states) are *consistent* partial instantiations and whose arcs represent operators transforming states to states. An operator takes

a consistent partial solution and augments it with an instantiation of an additional variable that does not conflict with prior assignments. Algorithms that use a fixed and predetermined variable ordering can be regarded as traversing a search tree in which the states are restricted to prefixes of variables along that ordering. Consider the n-queens problem, in which the goal is to place n queens on an $n \times n$ chessboard such that no two queens are on the same row, column, or diagonal. If we try to solve the problem by placing a queen on each row in some order, the states at the ith level of the search tree are legal positionings of queens on the first i rows. An operator applied to such a state assigns a queen on the $(i+1)$th row to a column that does not conflict with queens on the first i rows.

The notion of a search space and a search graph facilitates many types of problem-solving activities, not exclusively constraint satisfaction. We first present a brief review of search terminology and then make the appropriate specialization for our domain. A *state search space* is generally defined by four elements: a set of states, S, a set of operators O that map states to states, an initial state $s_0 \in S$, and a set of goal states $S_g \subseteq S$. The fundamental task is to find a solution, namely, a sequence of operators that transform the initial state into a goal state. Classic examples are the 8-puzzle, the traveling salesperson and the n-queens problem[1]. In the 8-puzzle, for instance, states represent any puzzle configuration. The initial state is the starting puzzle configuration, the goal state is the specified final configuration, and the set of operators are the moves which slide a tile into the open space.

The state search space can be represented by a directed graph, called the *search graph*, where the nodes represent states and where there is a directed arc $s_i \rightarrow s_j$ iff there is an operator transforming s_i to s_j. In graph terms, a solution is a directed path from the node representing the initial state to a goal node. *Terminal* or *leaf nodes* are nodes lacking outward directed arcs. Goal nodes represent solutions, and non-goal terminal nodes represent dead-ends. Any search algorithm for finding a solution can be understood as a traversal algorithm looking for a solution path in a search graph. For details about search spaces and general search algorithms, see [Nil78, Pea84].

We now define the search space of a constraint satisfaction problem when the variables are considered in a fixed ordering. In this case, the search graph is a tree.

Definition 1 (ordered search space) *Given a constraint network $\mathcal{R}$ and a variable ordering $d = x_1, \ldots, x_n$, in the ordered search space of a constraint satisfaction problem a state is any consistent partial instantiation $((x_1, a_1), \ldots, (x_j, a_j))$, $0 \leq j \leq n$, an operator maps a legal state $(a_1, \ldots, a_j)$ to another legal state $(a_1, \ldots, a_j, a_{j+1})$, the initial state is the empty instantiation, and goal states are complete instantiations.*

[1] For descriptions of these problems see [Pea84].

Notice that the above definition differs from some appearing in the literature in that it does not include the deadend variable and its possible failing values in the search space.

Example 1 *Consider the constraint network having four variables* z, x, y, l. *The domains are* $D_x = \{2,3,4\}$, $D_y = \{2,3,4\}$, $D_l = \{2,5,6\}$ *and* $D_z = \{2,3,5\}$. *There is a constraint between* z *and each of the variables* x, y, l *that requires the value assigned to* z *to divide the value for* x, y *and* l. *The constraint graph of this problem is depicted in Figure 1a; its search space graph along the ordering* $d_1 = z, x, y, l$ *is given in Figure 1b and along the ordering* $d_2 = x, y, l, z$, *in Figure 1c.*

Circled and filled nodes in Figure 1 denotes legal states. Terminal legal states that are solutions are denoted by filled triangles, while terminal deadends are denoted by filled boxes. The additional hollow boxes connected by broken lines in Figure 1b and Figure 1c represent illegal states that correspond to instantiation attempts that fail. These illegal states will sometimes be depicted in the search graph since they express problem solving activity (see section 3.2). In fact, various search spaces definition include those leaf nodes as legal states in the search space.

Figure 1 illustrates how ordering affects the size of the search space. Clearly, any traversal algorithm, whether depth-first or breadth-first, is likely to expand fewer nodes when given a smaller search space to explore. Since the search space includes all solutions, one way to assess whether its size is excessive is by counting the number of dead-end leaves. Ordering d_1, for instance, renders a search graph with only one dead-end leaf, whereas the search graph for ordering d_2 has 18 dead-end leaves (note that all the hollow nodes are illegal states).

When the variable ordering is left to the search algorithm rather than fixed in advance of search, the search space should include all possible orderings of variables. Thus, an unordered search space has as its states any consistent partial instantiation. The unordered search space looks undesirable at first since it may not effectively bound the search. Nevertheless, as we shall see, its flexibility frequently leads to a comparatively small *explored* graph.

3.1 Consistency level

Another factor affecting the search space is the level of local consistency, or explicitness of the constraints. The search space shrinks if we change the representation of a problem by using a tighter (but still equivalent) set of constraints.

Example 2 *Consider again the constraint network of Example 1. The network is not arc-consistent. For instance, the value 5 of* z *does not divide any of* x*'s values. Thus, applying arc-consistency results in the deletion of the value 5 from the domains of* z *and* l. *Consequently, with the resulting arc-consistency network the search space along ordering* d_1 *will no longer include the dead-end*

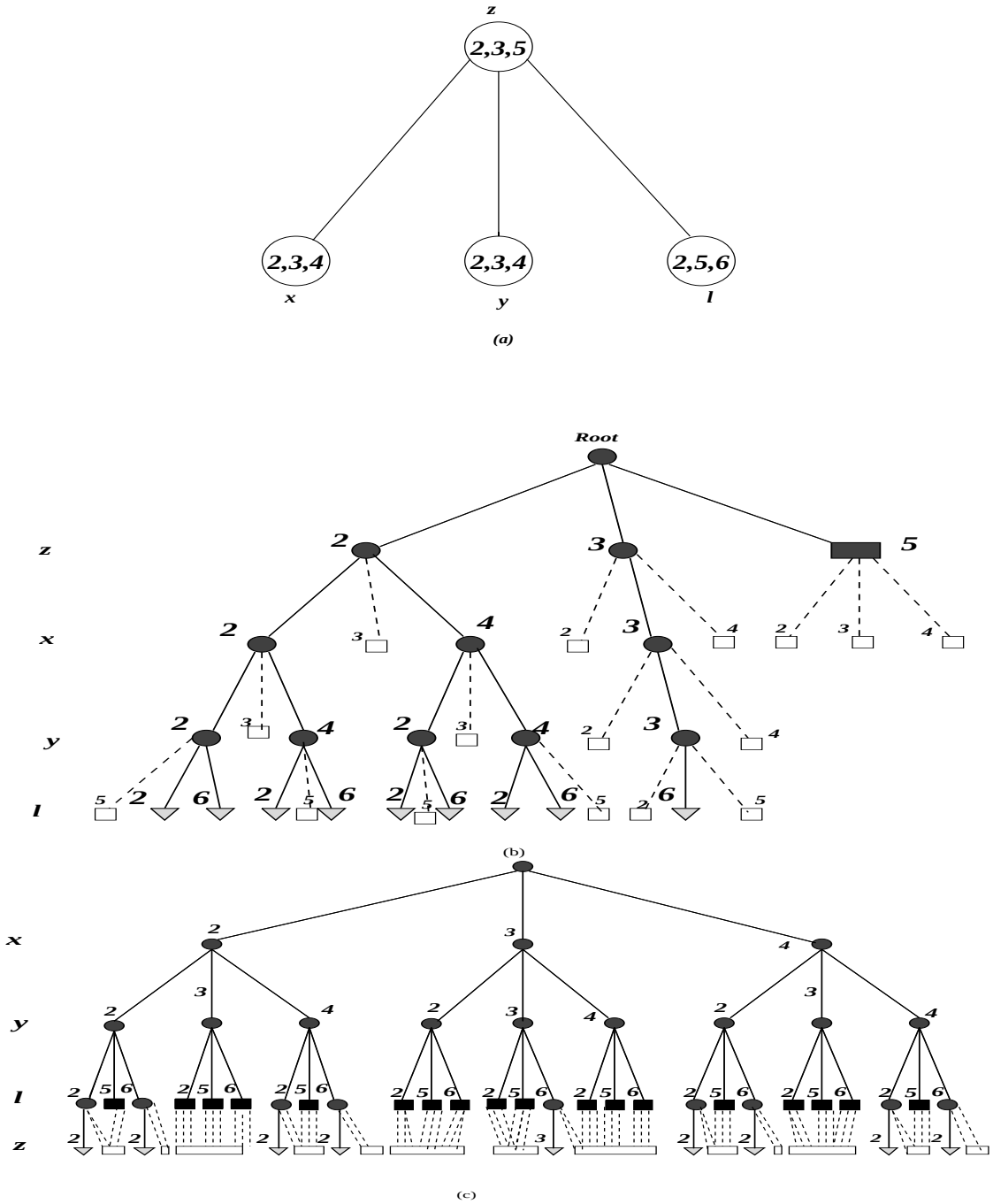


Figure 1: Constraint graph (a) and its search space for ordering $d_1 = z, x, y, l$ (b) and ordering $d_2 = x, y, l, z$ (c). Hollow nodes and bars express illegal states that may be considered but will be rejected as states.

at node $z = 5$, which emanates from the root in Figure 1a. Similarly, nine dead-end leaves (all corresponding to l=5) are eliminated from the search space along ordering d_2 thanks to arc-consistency. The network R is also not path-consistent. For instance, the assignment $(x = 2, y = 3)$ is consistent, but it cannot be extended to a value of z. To enforce path-consistency a collection of binary constraints are introduced. We get a network called $path(R)$ that includes the set of constraints

$$
\begin{aligned}
R_{zx} &= \{(2,2)(2,4)(3,3)\} \\
R_{zy} &= \{(2,2)(2,4)(3,3)\} \\
R_{zl} &= \{(2,2)(2,6)(3,6)\} \\
R_{xy} &= \{(2,2)(2,4)(4,2)(4,4)(3,3)\} \\
R_{xl} &= \{(2,2)(2,6)(4,2)(4,6)(3,6)\} \\
R_{yl} &= \{(2,2)(2,6)(4,2)(4,6)(3,6)\},
\end{aligned}
$$

and whose set of solutions is

$$
\begin{aligned}
&sol(z,x,y,l) = \{(2,2,2,2)(2,2,2,6)(2,2,4,2)(2,2,4,6) \\
&(2,4,2,2)(2,4,2,6)(2,4,4,2)(2,4,4,6)(3,3,3,6)\}
\end{aligned}
$$

The search graphs of $path(R)$ and R along ordering d_2 are compared in Figure 2b. Note the amount of pruning of the search space that results from tightening the representation.

Indeed, a tighter and more explicit representation of the same solution set generally results in a smaller underlying search space. In summary:

Theorem 1 *Let R be a network tighter than R', where both represent the same set of solutions for a given constraint network. Then, for any ordering d, any path appearing in the search graph derived from R also appears in the search graph derived from R'.* □

The above discussion suggests that one should make the representation of a problem as explicit as possible before searching for a solution. This observation needs two qualifications, however. First, applying a local consistency algorithm is costly and its cost may not always be offset by greater efficiency of the subsequent search. Second, while the size of the search space is smaller when the representation is tighter, a tighter representation normally possesses many more explicit constraints than a looser one. As a result, the former requires many more constraint checks (that is, testing if an assignment satisfies a single constraint) than the latter per *node generation*.

3.2 Node generation

Given a state s in the search space, *node generation* refers to the generation of a new child node of s, and *node expansion* refers to the task of generating all

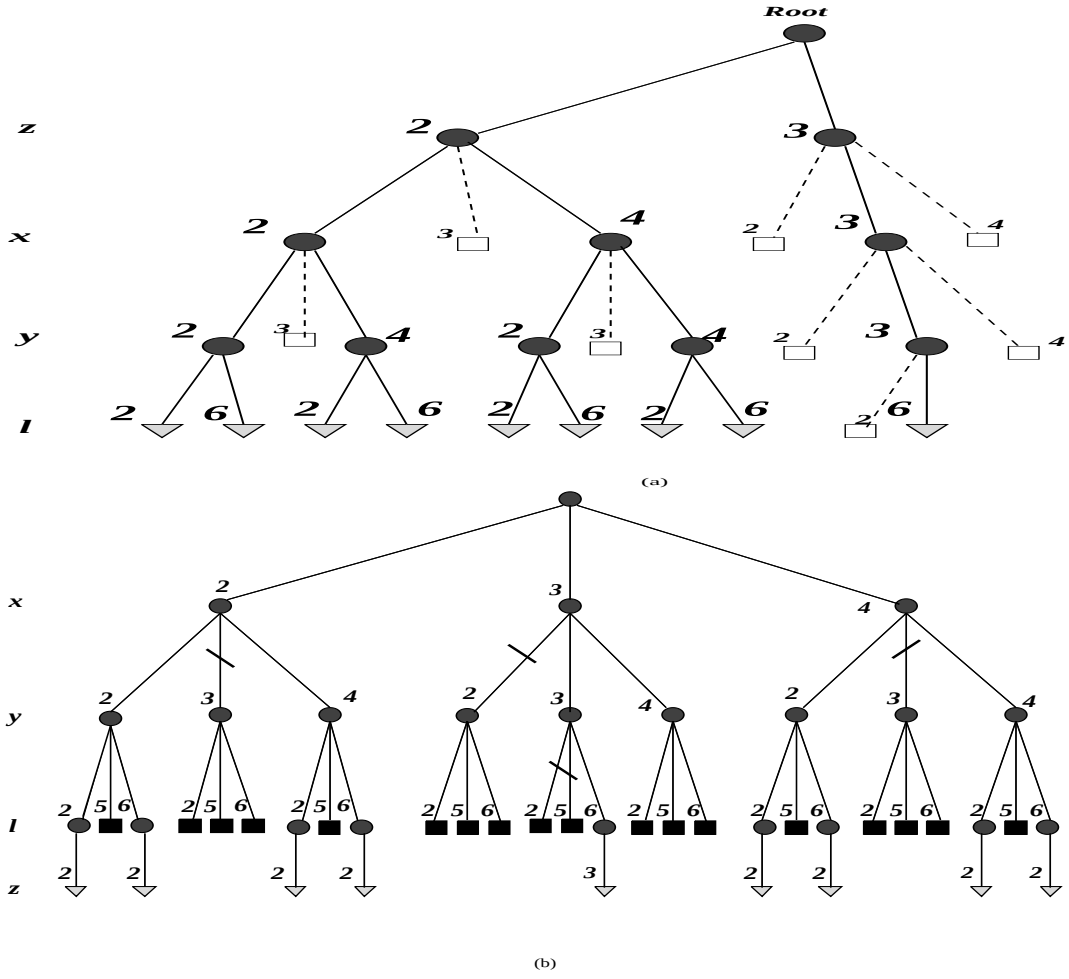


Figure 2: (a) Search space with ordering d_1 after arc-consistency, (b) Search space for ordering d_2 of Example 1 with reductions effects by enforcing path-consistency marked with slashes.

child nodes of s. Generating a new child node of a given state in the search space consists of extending a partial solution with a consistent value assignment to the next variable. This requires that the candidate value of the next variable be tested against prior value assignments. If the constraints are binary, we will never have more than $O(n)$ consistency checks per node generation. However, in the general case, the number of constraints may be very large; it may be $O(n^{r-1})$ when r bounds the constraints' arity. Therefore, whether one or a large number of constraints needs to be tested at each node generation, especially when the constraint arity is not bounded, may make a considerable difference to the efficiency of the search algorithm.

Example 3 *Consider again the network R in Example 1. When generating the search space for R along ordering d_1, exactly one constraint is tested for each new node generation. In contrast, when using $path(R)$ which has an explicit constraint for every pair of variables, each node generated at level 1 of the search tree requires one constraint check, at level 2 each node requires two constraint checks, and three checks are required for each node generated at level 3. Overall, fewer constraint tests (around 20) are performed when generating the whole search tree for R along ordering d_1, while many more constraint checks (around 40) are required for $path(R)$ using ordering d_1. When generating the search graph for R along ordering d_2, the first three levels of the tree require no constraint checks (there are no explicit constraints between variables x, y, l), but generating the fourth level in the search tree may require three constraint checks per node, yielding between 45-81 constraint tests depending on the order of constraint testing. When using $path(R)$ along ordering d_2, constraint checks are performed in each of the first three levels in the tree (one per node in the first level, two per node in the second, three per node in the third). This search graph has only 9 nodes at level 4, each requiring three tests. For ordering d_2, enforcing path-consistency pays off: the maximum number of tests before path-consistency is enforced is higher (around 80), while afterwards, it is reduced considerably (to about 50). So, while path-consistency reduces the overall number of constraint tests along ordering d_2, it increases the overall number of tests required along ordering d_1.*

Observe that following the application of arc-consistency, the search space along ordering d_1 contains solution paths only. This is an extremely desirable state of affairs. Any algorithm that searches such a space is guaranteed to find a solution in linear time (provided that the cost of node generation is bounded). When this is the case, we call the search space *backtrack-free*.

Definition 2 (backtrack-free network) *A network R is said to be* backtrack-free *along ordering d, if in the network's ordered search graph, every leaf node is a solution node.*

Generating the whole search space is unnecessary especially if we seek just one solution. In this case, the tradeoff between the cost of pruning and the saving

on search may be extremely unbalanced, since the expense of pre-processing is more likely to be recouped over the generation of all solutions.

Viewing a search algorithm as the exploration of a path in a graph is useful for analysis and design. One should not forget, however, that a typical search algorithm does not have the explicit search graph, nor would a typical search algorithm normally retain the entire graph that it explicates during search, because the explored search space may require exponential space. In the following section, we focus on a particular class of search algorithms called *backtracking*, all of which are restricted to polynomial space.

4 Backtracking

The simplest algorithm for solving constraint satisfaction problems is *backtracking*, which traverses the search graph in a depth-first manner. It is often assumed that the variables are examined in a fixed ordering. The backtracking algorithm maintains and operates on a partial solution that denotes a state in the algorithm's search space. Backtracking has three phases: a forward phase in which the next variable in the ordering is selected and called *current*; a phase in which the current partial solution is extended by assigning a consistent value to the current variable, if one exists; and a backward phase in which, when no consistent value exists for the current variable, focus returns to the variable prior to the current variable. All the variations of backtracking described in forthcoming subsections are designed to either return a single solution if one exists or determine that the network is inconsistent.

Figure 3 describes a basic backtracking algorithm. In addition to its fixed value domain D_i, each variable x_i maintains a mutable value domain D'_i such that $D'_i \subseteq D_i$. D'_i holds the subset of D_i that has not yet been examined under the current instantiation. Initially, all variables are uninstantiated.

We denote by $\vec{a_i}$ the subtuple of consecutive values $(a_1, ..., a_i)$ for a given ordering of the variables $x_1, ..., x_i$. We denote by $\vec{a}$ an arbitrary subtuple of values. Step 2c in the algorithm (see Figure 3) is implemented by performing *consistency checks* between $x_{cur} = a$ and all past assignments $x_i = a_i, 1 \leq i < cur$. The algorithm tests whether the tuple $\vec{a}_{cur-1}$, if extended by $x_{cur} = a$, is consistent; if the constraints are binary, the algorithm tests whether the pairs $(x_i = a_i, x_{cur} = a)$ are allowed by the binary constraints $R_{i,cur}$. Because consistency checking is performed frequently, a count of the number of consistency checks is a common measure of the overall cost of the algorithm. The following example demonstrates backtracking and its dependence on value ordering of the domains.[2]

[2]Since a consistency check is the most basic atomic operation performed in all backtracking algorithms, one should try to reduce consistency checks. One way is first to test whether the two variables are constrained and if not, no checking is performed.

Backtracking
Input: A constraint network R and an ordering of the variables $d = \{x_1, ..., x_n\}$.
Output: Either a solution if one exists or a decision that the network is inconsistent.

0. (Initialize.) $cur \leftarrow 0$.
1. (Step forward.) If x_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, $cur \leftarrow cur + 1$. Set $D'_{cur} \leftarrow D_{cur}$.
2. (Choose a value.) Select a value $a \in D'_{cur}$ that is consistent with all previously instantiated variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$ (x_{cur} is a dead-end), go to Step 3.
   (b) Select a from D'_{cur} and remove it from D'_{cur}.
   (c) For each constraint defined on x_1 through x_{cur} test whether it is violated by $\vec{a}_{cur-1}$ and $x_{cur} = a$. If it is, go to Step 2a.
   (d) Instantiate $x_{cur} \leftarrow a$ and go to Step 1.
3. (Backtrack step.) If x_{cur} is the first variable, exit with "inconsistent". Otherwise, set $cur \leftarrow cur - 1$. Go to Step 2.

Figure 3: Algorithm backtracking

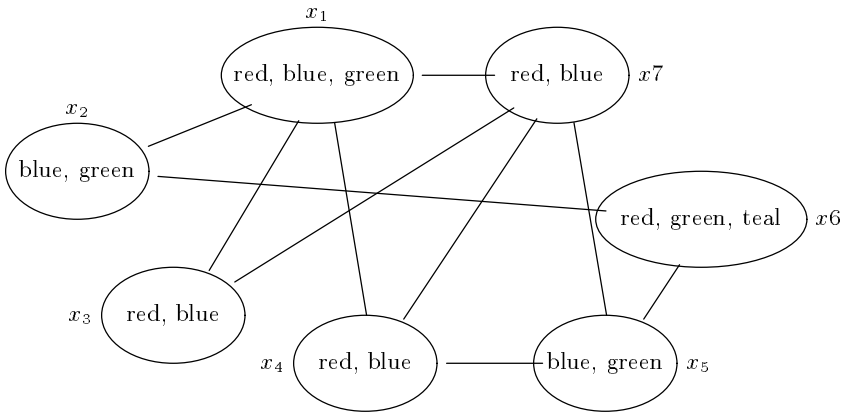


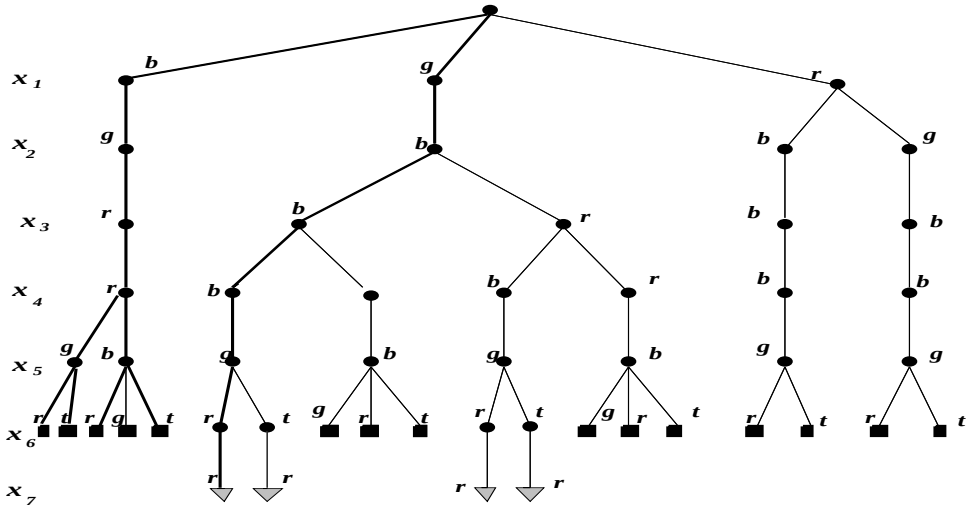
Figure 4: A modified coloring problem.

Example 4 *Consider the network in Example 1. Using ordering* $d_1 = (z, x, y, l)$ *and assuming that the values in the domains are ordered increasingly relative to the natural numbers, the algorithm performs a depth-first left-to-right search exploring the search space depicted in Figure 1b. In this case, because the first path to a leaf is a solution path, the algorithm will go forward and explicate only this path. When ordering* d_2 *is employed, the algorithm traverses the search space depicted in Figure 1c. If traversed from left to right, the algorithm again will find a solution and will have no need to go back. However, if the domains are searched in reverse order (corresponding to a depth-first, right-to-left search), the algorithm will instantiate* $x = 4, y = 4, l = 6$*. As these variables have no constraints among them, consistency checks are never performed for node generation. When variable* z *becomes the current variable, the algorithm first tests the value* $z = 5$ *against* $x = 4$*. Since there is a conflict, the algorithm will test* $z = 3$ *against* $x = 4$*, fail again, and finally test* $z = 2$ *against each past instantiation. Because there is no conflict, the solution* $x = 4, y = 4, l = 6, z = 2$ *is returned. Plainly, although the search spaces for the two orderings differs dramatically, a lucky ordering of the domain may result in a very effective search if we are searching for one solution.*

Example 5 *Consider the example in Figure 4, which is a modified coloring problem. The domain of each node is written inside the node. Note that not all nodes have the same domain. Arcs join nodes that must be assigned different colors. Assume backtracking search for a solution using two possible orderings:* $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$ *and* $d_2 = x_1, x_7, x_4, x_5, x_6, x_3, x_2$*. The search spaces along orderings* d_1 *and* d_2*, as well as those portions explicated by backtracking from left to right, are depicted in Figure 5a and 5b, respectively. (Only legal states are depicted in the figure.)*

Backtracking usually suffers from thrashing, namely, rediscovering the same inconsistencies and same partial successes during search. Efficient cures for such behavior in all cases are unlikely, since the problem is NP-hard [GJ79]. However, there are some simple heuristics that can provide convenient remedies in a large number of cases. Some of these heuristics involve limited exploration of the future search space so as to increase the chance of a good current decision, while others involve a limited amount of learning, which entails exploiting information already collected during search. Most remedies cures trade the cost of implementing the cure off against the amount of pruning in search.

Many of the improvements to backtracking presented here will focus on pruning the size of the search space while keeping the cost of node generation bounded. Sections 5 and 6 deal with look-back improvements; look-ahead strategies are discussed in section 7.



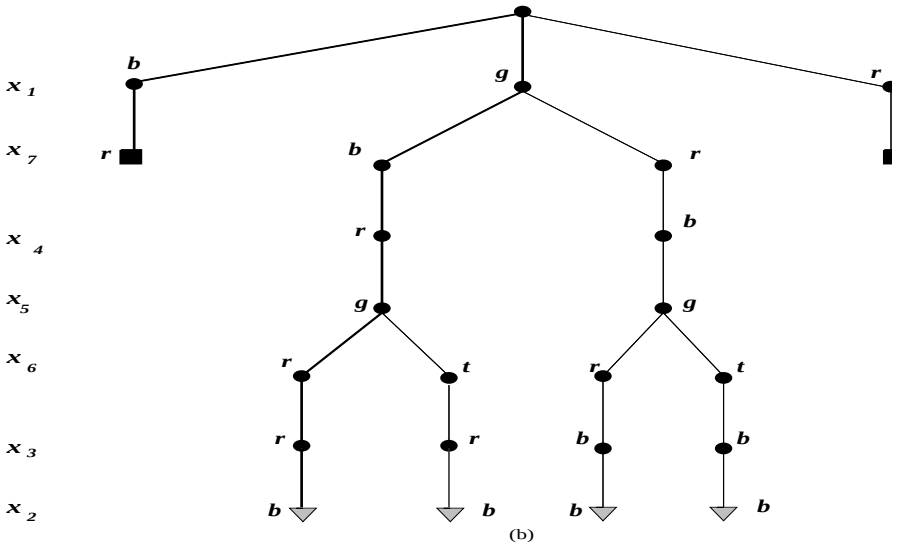


Figure 5: Backtracking search for the orderings (a) $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$ and (b) $d_2 = x_1, x_7, x_4, x_5, x_6, x_3, x_2$ on the example in Figure 4. The fat lines denote the portions explored by backtracking when using left-to-right ordering.

5 Backjumping

Backjumping schemes are one of the primary tools for reducing backtracking's unfortunate tendency to rediscover the same dead-ends. A dead-end occurs if x_i has no consistent values left, in which case the backtracking algorithm will go back to x_{i-1}. Suppose a new value for x_{i-1} exists but there is no constraint between x_i and x_{i-1}. A dead-end will be reached at x_i for each value of x_{i-1} until all values of x_{i-1} have been exhausted. For instance, the problem in Figure 4 will have a dead-end at x_7 given the assignment $(x_1 = red, x_2 = blue, x_3 = blue, x_4 = blue, x_5 = green, x_6 = red)$. Backtracking will then return to x_6 and instantiate it as $x_6 = teal$, but the same dead-end will be encountered at x_7. We can finesse this situation by identifying the *culprit variable* responsible for the dead-end and then jumping back immediately to reinstantiate the culprit variable, instead of repeatedly instantiating the chronologically previous variable. Identification of a culprit variable in backtracking is based on the notion of *conflict sets*.

5.1 Conflict sets

A dead-end state at level i indicates that a current partial instantiation $\vec{a_i} = (a_1, ..., a_i)$ conflicts with x_{i+1}. $(a_1, ..., a_i)$ is called a dead-end state and x_{i+1} is called a dead-end variable. Namely, backtracking generated the consistent tuple $\vec{a_i} = (a_1, ..., a_i)$ and tried to extend it to the next variable, x_{i+1}, but failed: no value of x_{i+1} was consistent with all the values in $\vec{a_i}$.

The subtuple $\vec{a}_{i-1} = (a_1, ..., a_{i-1})$ may also be in conflict with x_{i+1}, and therefore going back to x_i and changing its value will not always resolve the dead-end at variable x_{i+1}. In general, a tuple $\vec{a_i}$ that is a leaf dead-end at level i may contain many subtuples that are in conflict with x_{i+1}. Any such partial instantiation will not be part of any solution. Backtracking's control strategy often retreats to a subtuple $\vec{a_j}$ (alternately, to variable x_j) without resolving all or even any of these conflict sets. As a result, a dead-end at x_{i+1} is guaranteed to recur. Therefore, rather than going to the previous variable, the algorithm should jump back from the dead-end state at $\vec{a_i} = (a_1, ..., a_i)$ to the most recent variable x_b such that $\vec{x}_{b-1} = (a_1, ..., a_{b-1})$ contains no conflict sets of the dead-end variable x_{i+1}. As it turns out, identifying this culprit variable is fairly easy.

Definition 3 (conflict set) *Let $\vec{a} = (a_1, ..., a_i)$ be a consistent instantiation, and let x be a variable not yet instantiated. If no value in the domain of x is consistent with $\vec{a}$, we say that $\vec{a}$ is a* conflict set *of x, or that $\vec{a}$ conflicts with variable x. If, in addition, $\vec{a}$ does not contain a subtuple that is in conflict with x, $\vec{a}$ is called a* minimal *conflict set of x.*

Definition 4 (i-leaf dead-ends) *Given an ordering $d = x_1, ..., x_n$, then a tuple $\vec{a_i} = (a_1, ..., a_i)$ that is consistent but is in conflict with x_{i+1} is called an* i-leaf dead-end state.

Definition 5 (no-good) *Any partial instantiation $\vec{a}$ that does not appear in any solution is called a* no-good. Minimal *no-goods have no no-good subtuples.*

A conflict set is clearly a no-good, but there are no-goods that are not conflict sets of any single variable. Namely, they may conflict with two or more variables.

Whenever backjumping discovers a dead-end, it tries to jump as far back as possible without skipping potential solutions. These two issues of *safety* in jumping and *optimality* in the magnitude of a jump need to be defined relative to the *information status* of a given algorithm. What is safe and optimal for one style of backjumping may not be safe and optimal for another, especially if they are engaged in different levels of information gathering. Next, we will discuss two styles of backjumping, Gaschnig's and graph-based, that lead to different notions of safety and optimality when jumping back.

Definition 6 (safe jump) *Let $\vec{a_i} = (a_1, ..., a_i)$ be an i-leaf dead-end state. We say that x_j, where $j \leq i$, is* safe *if the partial instantiation $\vec{a_j} = (a_1, ..., a_j)$ is a no-good, namely, it cannot be extended to a solution.*

In other words, we know that if x_j's value is changed no solution will be missed.

Definition 7 *Let $\vec{a_i} = (a_1, ..., a_i)$ be an i-leaf dead-end. The* culprit index *relative to $\vec{a_i}$ is defined by $b = \min\{j \leq i |\ \vec{a_j}$ conflicts with $x_{i+1}\}$. We define the* culprit variable *of $\vec{a_i}$ to be x_b.*

We use the notions of culprit tuple $\vec{a_b}$ and culprit variable x_b interchangeably. By definition, $\vec{a_b}$ is a conflict set that is minimal relative to prefix tuples, namely, those associated with a prefix subset of the variables. We claim that x_b is both safe and optimal: safe in that $\vec{a_b}$ cannot be extended to a solution; optimal in that jumping back to an earlier node risks missing a solution. Essentially, if the algorithm fails to retract as far back as x_b, it is guaranteed to wander in the search space rooted at $\vec{a_b}$ unnecessarily, but if it retracts further back than x_b, the algorithm may exclude a portion of the search space in which there is a solution.

Proposition 1 *If $\vec{a_i}$ is an i-leaf dead-end discovered by backtracking, then when x_b is the culprit variable, $\vec{a_b}$, is an optimal and safe backjump destination.*

Proof: By definition of a culprit, $\vec{a_b}$ is a conflict set of x_{i+1} and therefore is a no-good. Consequently, jumping to x_b and changing the value a_b of x_b to another consistent value of x_b (if one exists) will not result in skipping a potential solution. To prove optimality, we need to show that jumping farther back to an earlier node risks skipping potential solutions. Specifically, if the algorithm jumps to x_{b-j}, then by definition $\vec{a}_{b-j}$ is not a conflict set of x_{i+1}, and therefore it may be part of a solution. Note that $\vec{a}_{b-j}$ may be a no-good,

but the backtracking algorithm cannot determine whether it is without testing it further. □.

Computing the culprit variable of $\vec{a_i}$ is relatively simple since at most i subtuples need to be tested for consistency with x_{i+1}. Moreover, it can be computed during search by gathering some basic information while assembling $\vec{a_i}$. Procedurally, the culprit variable of a dead-end $\bar{a_i} = (a_1, ..., a_i)$ is the most recent variable whose assigned value renders inconsistent the last remaining value in the domain of x_{i+1} not ruled out by prior variables.

Next we present three variants of backjumping. *Gaschnig's backjumping* implements the idea of jumping back to the culprit variable only at leaf dead-ends. *Graph-based Backjumping* extracts information about irrelevant backtrack points exclusively from the constraint graph. Although its strategy for jumping back at leaf dead-ends is less than optimal, it introduces the notion of jumping back at internal dead-ends as well as leaf dead-ends. *Conflict-directed backjumping* combines optimal backjumps at both leaf and internal dead-ends.

5.2 Gaschnig's backjumping

Rather than wait for a dead-end $\vec{a_i}$ to occur, Gaschnig's backjumping [Gas79] records some information while generating $\vec{a_i}$, and uses this information to determine the dead-end's culprit variable x_b. The algorithm uses a marking technique whereby each variable maintains a pointer to the *latest* predecessor found incompatible with any of the variable's values. While generating $\vec{a_i}$ in the forward phase, the algorithm maintains an array of pointers $high_j, 1 \leq j \leq n$, for each variable x_j. These pointers indicate the variable most recently tested for consistency with x_j and found to be in conflict with a value of x_j that was not conflicted with before. In other words, when x_j is a dead-end variable and $high_j = 3$, the pointer indicates that $\vec{a_3}$ is a conflict set of x_j. If $\vec{a}_{j-1}$ is not a dead-end, then $high_j$ arbitrarily, is assigned the value $j - 1$. The algorithm jumps from a leaf dead-end $\vec{a_i}$ that is inconsistent with x_{i+1}, back to $x_{high_{i+1}}$, its culprit since the dead-end variable is x_{i+1}. The algorithm is presented in Figure 6.

Proposition 2 *Gaschnig's backjumping implements only safe and optimal backjumps in leaf dead-ends.*

Proof: Whenever there is a leaf dead-end $\vec{a}_{cur-1}$, the algorithm has a partial instantiation $\vec{a}_{cur-1} = (a_1, ..., a_{cur-1})$. Let $j = high_{cur}$. The algorithm jumps back to x_j, namely, to the tuple $\vec{a_j}$. Clearly, $\vec{a_j}$ is in conflict with x_{cur}, so we only have to show that $\vec{a_j}$ is minimal. Since $j = high_{cur}$ when the domain of x_{cur} is exhausted, and since a dead-end did not happen previously, any earlier $\vec{a_k}$ for $k < j$ is not a conflict set of x_{cur}, and therefore x_j is the culprit variable. From Proposition 1, it follows that this algorithm is safe and optimal. □

Gaschnig's backjumping
Input: A constraint network R and an ordering of the variables $d = \{x_1, ..., x_n\}$.
Output: Either a solution if one exists or a decision that the network is inconsistent.

0. (Initialize.) $cur \leftarrow 0$.
1. (Step forward.) If x_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, $cur \leftarrow cur + 1$. Set $D'_{cur} \leftarrow D_{cur}$. Set $high_{cur} \leftarrow 0$.
2. Select a value $a \in D'_{cur}$ that is consistent with all previous variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Select a value a from D'_{cur} and remove it from D'_{cur}.
   (c) For $1 \leq i \leq cur$ (in ascending order) do,
   if $i > high_{cur}$, then set $high_{cur} \leftarrow i$; if $\vec{a_i}$ conflicts with $x_{cur} = a$, then go to 2(a).
   (d) Instantiate $x_{cur} \leftarrow a$ and go to 1.
3. (Backjumping step) If $high_{cur} = 0$ (there is no previous variable which shares a constraint with x_{cur}), exit with "inconsistent". Otherwise, select variable $x_{high_{cur}}$; call it x_{cur}. Go to 2.

Figure 6: Gaschnig's backjumping.

Example 6 *For the problem in Figure 4, at the dead-end for* x_7 ($x_1 = red$, $x_2 = blue$, $x_3 = blue$, $x_4 = blue$, $x_5 = green$, $x_6 = red$), $high_7 = 3$, *because* $x_7 = red$ *was ruled out by* $x_1 = red$, $blue$ *was ruled out by* $x_3 = blue$, *and no later variable had to be examined. On returning to* x_3, *the algorithm finds no further values to try* ($D'_3 = \emptyset$). *Since* $high_3 = 2$, *the next variable examined will be* x_2. *This demonstrate the algorithm's ability to backjump on leaf dead-ends. On subsequent dead-ends (in* x_3) *it goes back to its preceding variable only.*

In Gaschnig's backjumping, a jump happens only at leaf dead-ends. If all the children of a node in the search tree lead to dead-ends (as happens with x_3 in Figure 5a) the node is termed an *internal dead-end*. Algorithm *graph-based backjumping* implements jumps at internal dead-ends as well as at leaf dead-ends.

5.3 Graph-based backjumping

Graph-based backjumping extracts knowledge about possible conflict sets from the constraint graph exclusively. Whenever a dead-end occurs and a solution cannot be extended to the next variable x, the algorithm jumps back to the most recent variable y *connected* to x in the constraint graph; if y has no more values, the algorithm jumps back again, this time to the most recent variable z connected to x *or* y; and so on. The second and any further jumps are jumps at *internal dead-ends*. By using the precompiled information encoded in the graph, the algorithm avoids computing $high_i$ during each consistency test. This, however, is a programming convenience only. The cost of computing $high_i$ at each node is small, and overall computing $high_i$ at each node is less expensive than the loss of using only the information in the graph. Information retrieved from the graph is less precise since, even when a constraint exists between two variables x and y, the particular value currently being assigned to y may not conflict with any potential value of x. For instance, assigning $blue$ to x_2 in the problem of Figure 4 has no effect on x_6, because $blue$ is not in x_6's domain. Since graph-based backjumping does not maintain domain value information, it fills in this gap by assuming the worst: it assumes that the subset of variables connected to x_{i+1} is a minimal conflict set of x_{i+1}. Under this assumption, the most recent variable connected to x_{i+1} is the culprit variable.

We mention graph-based backjumping here primarily because algorithms with performance tied to the constraint graph lead to graph-theoretic bounds and thus to graph-based heuristics aimed at reducing these bounds. Such bounds are applicable to algorithms that use refined run-time information such as Gaschnig's backjumping and conflict-directed backjumping.

We now introduce some graph terminology which will be used ahead.

Definition 8 (ancestors,parent) *Given a constraint graph and an ordering* d, *the* ancestor set *of variable* x, *denoted* $anc(x)$, *is the subset of the variables*

that precede and are connected to x. The parent *of x, denoted $p(x)$, is the most recent (or latest) variable in $anc(x)$. If $\vec{a_i} = (a_1, ..., a_i)$ is a leaf dead-end, we define $anc(\vec{a_i}) = anc(x_{i+1})$, and $p(\vec{a_i}) = p(x_{i+1})$.*

Example 7 *Consider the ordered graph in Figure 7a along the ordering $d_1 = x_1, ..., x_7$. In this example, $anc(x_7) = \{x_1, x_3, x_4, x_5\}$ and $p(x_7) = x_5$. The parent of the leaf dead-end $\vec{a_6} = (blue, green, red, red, blue, red)$ is x_5, which is the parent of x_7.*

It is easy to show that if $\vec{a_i}$ is a leaf dead-end, $p(\vec{a_i})$ is safe. Moreover, if only graph-based information is utilized, it is unsafe to jump back any further. When facing an internal dead-end at $\vec{a_i}$, however, it may not be safe to jump to its parent $p(\vec{a_i})$.

Example 8 *Consider again the constraint network in Figure 4 with ordering $d_1 = x_1, ..., x_7$. In this ordering, x_1 is the parent of x_4. Assume that a dead-end occurs at node x_5 and that the algorithm returns to x_4. If x_4 has no more values to try, it will be perfectly safe to jump back to its parent x_1. Now let us consider a different scenario. The algorithm encounters a dead-end leaf at x_7, so it jumps back to x_5. If x_5 is an internal dead-end, control is returned to x_4. If x_4 is also an internal dead-end, then jumping to x_1 is unsafe now, since if we change the value of x_3 perhaps we could undo the dead-end at x_7 that started this latest retreat. If, however, the dead-end variable that initiated this latest retreat was x_6, it would be safe to jump as far back as x_2 upon encountering an internal dead-end at x_4.*

Clearly, when encountering an internal dead-end, it matters which node initiated the retreat. The culprit variable is determined by the *induced ancestor set* in the current *session*.

Definition 9 (session) *Given a constraint network that is being searched by a backtracking algorithm, the* current session *of x_i is the set of variables processed by the algorithm since the latest* invisit *to x_i. We say that backtracking* invisits *x_i if it processes x_i coming from a variable earlier in the ordering. The session starts upon invisiting x_i and ends when retracting to a variable that precedes x_i.*

Definition 10 (induced ancestors, induced parent) *Given a variable x_j and a subset of variables Y that all succeed x_j, the* induced ancestor set *of x_j relative to Y, denoted $I_j(Y)$, contains all nodes x_k, $k < j$, such that there is a path of length one or more from x_j to x_k that may go through nodes in Y. Let the* induced parent *of x_j relative to Y, denoted $P_j(Y)$, be the latest variable in $I_j(Y)$.*

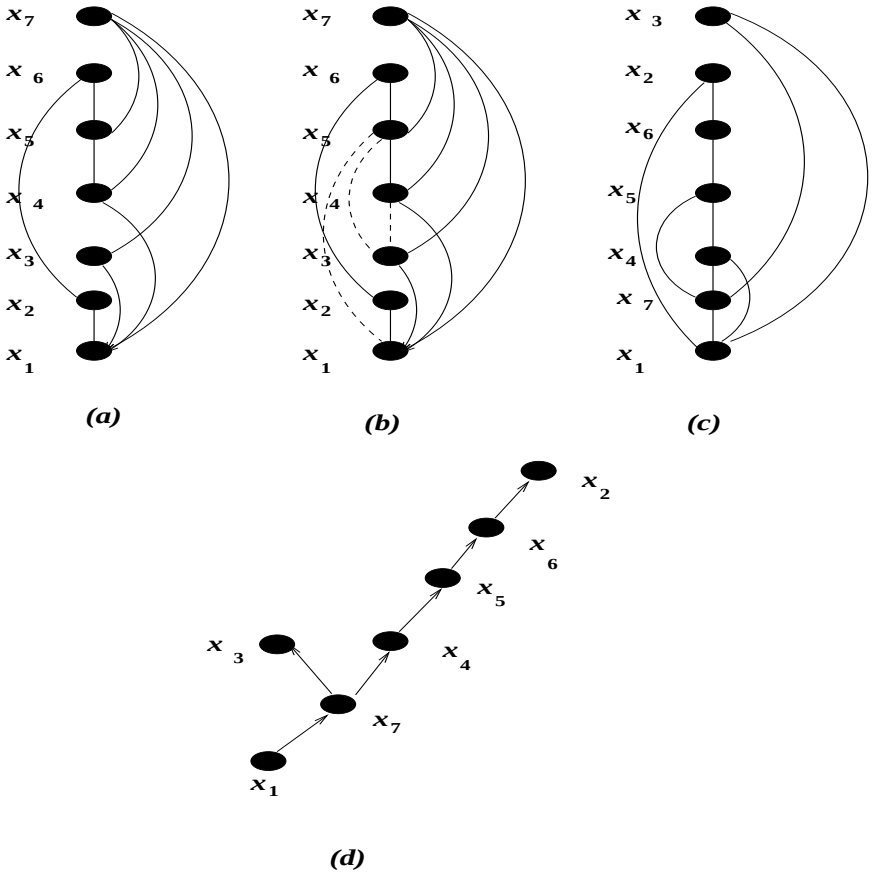


Figure 7: Two ordered constraint graphs on the example in Figure 4 (a) $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$, (b) the induced graph along d_1, (c) $d_2 = x_1, x_7, x_4, x_5, x_6, x_3, x_2$ (d) A DFS spanning tree along ordering d_2.

Theorem 2 *Let $\vec{a_i}$ be a dead-end (internal or leaf), and let Y be the set of dead-end variables (leaf or internal) in the current session of x_i. If only graph information is used, $x_j = P_i(Y)$ is the earliest (and therefore safe) culprit variable.*

Proof: By definition of x_j, all the variables between x_{i+1} and x_j do not participate in any constraint with any of the dead-end variables Y in x_i's current session. Consequently, any change of value to any of these variables will not perturb any of the no-goods that caused this dead-end and so they can be skipped.

To prove optimality, we need to show that if the algorithm jumped to a variable earlier than x_j, some solutions might be skipped. Let y_i be the first dead-end in Y that added x_j to the induced ancestor set of x_i. We argue that there is no way to rule out the possibility that there exists an alternative value of x_j that may lead to a solution. Let x_i equal a_i at the moment a dead-end at y_i occurred. Variable y_i is either a leaf dead-end or an internal dead-end. If y_i is a leaf dead-end, then x_j is an ancestor of y_i. Clearly, had the value of x_j been different, the dead-end at y_i may have not occurred. Therefore the possibility of a solution with an alternative value to x_j was not ruled out. In the case that y_i is an internal dead-end, it means that there were no values of y_i that were both consistent with $\vec{a_j}$ and that could be extended to a solution. It is not ruled out, however, that different values of x_j, if attempted, could permit new values of y_i for which a solution might exist. □

Example 9 *Consider again the ordered graph in Figure 7a, and let x_4 be a dead-end variable. If x_4 is a leaf dead-end, then $Y = \{x_4\}$, and x_1 is the sole member in its induced ancestor set $I_4(Y)$. The algorithm may jump safely to x_1. If x_4 is an internal dead-end with $Y = \{x_4, x_5, x_6\}$, the induced ancestor set of x_4 is $I_4(\{x_4, x_5, x_6\}) = \{x_1, x_2\}$, and the algorithm can safely jump to x_2. However, if $Y = \{x_7\}$, the corresponding induced parent set $I_4(\{x_7\}) = \{x_1, x_3\}$, and upon encountering a dead-end at x_4, the algorithm should retract to x_3. If x_3 is also an internal dead-end the algorithm retract to x_1 since $I_3(\{x_4, x_7\}) = \{x_1\}$. If, however, $Y = \{x_5, x_6, x_7\}$, when a dead-end at x_4 is encountered (we could have a dead-end at x_7, jump back to x_5, go forward and jump back again at x_6, and another jump at x_5) then $I_4(\{x_5, x_6, x_7\}) = \{x_1, x_2, x_3\}$, the algorithm retracts to x_3, and if it is a dead-end it will retract further to x_2, since $I_3(\{x_4, x_5, x_6, x_7\}) = \{x_1, x_2\}$.*

The algorithm in Figure 8 incorporates jumps to the optimal culprit variable at both leaf and internal dead-ends. For each variable x_i, the algorithm maintains x_i's induced ancestor set I_i relative to the dead-ends in x_i's current session. We summarize:

Theorem 3 *Graph-based backjumping implements jumps to optimal culprit variables at both leaf and internal dead-ends, when only graph information is used.*

Graph-based Backjumping
Input: A constraint network R and ordering $d = \{x_1, ..., x_n\}$.
Output: Either a solution if one exists or a decision that the network is inconsistent.

0. (Initialize ancestor sets.) Compute $anc(x_i)$ for each variable. Set $I_i \leftarrow anc(x_i)$.
1. (Step forward.) If x_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable in the ordering. Set $D'_{cur} \leftarrow D_{cur}$.
2. Select a value $a \in D'_{cur}$ that is consistent with all previous variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop a from D'_{cur}.
   (c) For every constraint defined on x_1 through x_{cur} test whether it is violated by $\vec{a}_{cur-1}$ and $x_{cur} = a$. If it is, go to 2(a).
   (d) Instantiate $x_i \leftarrow a$ and go to 1.
3. (Backjump.) If $I_{cur} = \emptyset$ (there is no variable in the induced ancestor set), exit with "inconsistent". Otherwise, set $I_{temp} \leftarrow I_{cur}$, Set cur equal to the index of the last variable in I_{cur}. Set $I_{cur} \leftarrow I_{cur} \cup I_{temp} - \{x_{cur}\}$. For all $j > cur$, set $I_j \leftarrow anc(x_j)$. Go to 2.

Figure 8: Algorithm graph-based backjumping.

Proof: Step 3 of the algorithm maintains the set I_{cur}, which is the induced ancestor set of x_{cur} relative to the dead-end variables in its session. Thus, the claim follows from Theorem 2. □

5.3.1 Using depth-first ordering

Although the implementation of the optimal graph-based backjumping scheme requires, in general, careful maintenance of each variable's induced ancestor set, some orderings facilitate a particularly simple rule for determining the variable to jump to.

Given a graph, a depth-first search (*DFS*) ordering is one that is generated by a *DFS* traversal of the graph. This traversal ordering results also in a *DFS spanning tree* of the graph which includes all and only the arcs in the graph that were traversed in a forward manner. The depth of a *DFS* spanning tree is the number of levels in that tree created by the *DFS* traversal (see [Eve79]). The arcs in a *DFS* spanning tree are directed towards the higher indexed node. For each node, its neighbor in the *DFS* tree preceding it in the ordering is called its *DFS parent*.

If we use graph-based backjumping on a *DFS* ordering of the constraint graph, finding the optimal graph-based back-jump destination requires following a very simple rule: if a dead-end (leaf or internal) occurs at variable x, go back to the *DFS* parent of x.

Example 10 *Consider, once again, the example of Figure 4. A* DFS *ordering:* $d_2 = (x_1, x_7, x_4, x_5, x_6, x_2, x_3)$ *and its corresponding* DFS *spanning tree are given in Figure 7c,d. If a dead-end occurs at node* x_3*, the algorithm retreats to its* DFS *parent, which is* x_7*.*

In summary,

Proposition 3 *Given a DFS ordering of the constraint graph, if* $f(x)$ *denotes the* DFS *parent of* x*, then, upon a dead-end at* x*,* $f(x)$ *is* x*'s graph-based culprit variable for both leaf and internal dead-ends.*

Proof: Given a *DFS* ordering and a corresponding *DFS* tree we will show that if there is a dead-end at x (internal or leaf) $f(x)$ is the latest amongst all the induced ancestors of x. Clearly, $f(x)$ always appear in the induced ancestor set of x since it is connected to x and since it precedes x in the ordering. It is also the most recent one since all the variables that appear in x's session must be its descendents in the *DFS* subtree rooted at x. Let y be a dead-end variable in the session of x. It is easy to see that y's ancestors that precede x must lie of the path from the root to x and therefore they either coincide with $f(x)$, or appear before $f(x)$. □

We can now present the first of two graph-related bounds on the complexity of backjumping.

Theorem 4 *When graph-based backjumping is performed on a* DFS *ordering of the constraint graph, its complexity is* $O(exp(b^m k^{m+1}))$ *steps, where* b *bounds the branching degree of the* DFS *tree associated with that ordering,* m *is its depth and* k *is the domain size.*

Proof: Let x_m be a node in the DFS spanning tree whose DFS subtree has depth of m. Let T_m stand for the maximal search-tree rooted at x_m, namely, it is the maximum number of nodes visited in any session of x_m. Since any assignment of a value to x_m generates at most b subtrees of depth $m-1$ or less that can be solved independently, T_m obeys the following recurrence:

$$\begin{aligned} T_m &= k \cdot b \cdot T_{m-1} \\ T_0 &= k \end{aligned}$$

Solving this recurrence yields $T_m = b^m k^{m+1}$. Thus, the worst-case time complexity of graph-based backjumping is $O(b^m k^{m+1})$. Notice that when the tree is balanced (namely, each internal node has exactly two child nodes) the bound can be improved to $T_m = O((n/b)k^{m+1})$, since $n = O(b^{m+1})$. □.

The bound suggests a graph-based ordering heuristic: use a *DFS* ordering having a minimal depth. Unfortunately, like many other graph parameters we will encounter, finding a minimal depth DFS tree is NP-hard. Nevertheless, knowing what we should be minimizing may lead to useful heuristics.

We will now show that graph-based backjumping can be bounded for any variable ordering, not only a *DFS* one. For that a few more graph-based concepts have to be introduced.

Definition 11 (width, tree-width) *Given a graph* (G) *over nodes* $X = \{x_1, ..., x_n\}$, *and an ordering* $d = x_1, ..., x_n$, *the* width *of a node in the ordered graph is the number of its earlier neighbors. The* width of an ordering *is the maximal width of all its nodes along the ordering, and the* width of the graph *is the minimum width over all its orderings. The* induced ordered graph *of* G, *denoted* G_o^* *is the ordered graph obtained by recursively connecting all earlier neighbors of* x_i *going in reverse order of* o. *The* induced width *of this ordered graph, denoted* $w^*(o)$, *is the maximal number of earlier neighbors each node has in* G_o^*. *The minimal induced width over all the graph's orderings is the* induced width w^*. *A related well known parameter, called the* tree-width [Arn85] *of the graph, equals the induced-width plus one.*

Example 11 *Consider the graph in Figure 7a ordered along* $d_1 = x_1, ..., x_7$. *The width of this ordering is 4 since this is the width of node* x_7. *On the other hand the width of* x_7 *in the ordering* $d_2 = x_1, x_7, x_4, x_5, x_6, x_2, x_3$ *is just 1. and the width of ordering* d_2 *is just 2 (Figure 7c). The induced graph along* d_1 *is given in Figure 7b. The added arcs, connecting earlier neighbors while going from* x_7 *towards* x_1, *are denoted by broken lines. Note that the induced width of node* x_5 *changes from 1 to 4. The induced width of ordering* d_1 *remains 4.*

Given a graph G and an ordering d, it can be shown that d is a DFS ordering of its induced graph G_d^*. Let m_d^* be the depth of this DFS ordering of G_d^*.

Theorem 5 *Given an ordering d, let m_d^* be the depth of the DFS tree of the induced graph G_d^* along d. The complexity of graph-based backjumping using ordering d is $O(exp(m_d^*))$.*

For a proof see [BM96].

5.4 Conflict-directed backjumping

The two ideas, jumping back to a variable that, *as instantiated*, is in conflict with the current variable, and jumping back at internal dead-ends, can be integrated into a single algorithm, the *conflict-directed backjumping* algorithm [Pro93a]. This algorithm uses the scheme we have outlined for graph-based backjumping but, rather than using graph information, exploits information gathered during search. For each variable, the algorithm maintains an induced *jumpback set*.

Given a dead-end tuple $\vec{a_i}$, we define the jumpback set of $\vec{a}_i$ (or of x_{i+1}) as the variables participating in $\vec{a}_i$'s *earliest minimal conflict set*. Conflict-directed backjumping includes a variable in the jumpback set if its current value conflicts with a value of the current variable which was not in conflict with any earlier variable assignment.

Definition 12 (earliest minimal conflict set) *Let $\vec{a_i}$ be a dead-end tuple whose dead-end variable is x_{i+1}. We denote by $emc(\vec{a_i})$ the* earliest minimal conflict set *of $\vec{a_i}$ and by $par(\vec{a_i})$ the set of variables appearing in $emc(\vec{a_i})$. Formally, the emc is generated by selecting its members from $\vec{a_i}$ in increasing order. Assume that $(a_{i_1}, ..., a_{i_j})$ were the first j members selected. Then, the first value appearing after a_{i_j} in $\vec{a_i}$ that is inconsistent with a value of x_{i+1} that was not ruled out by $(a_{i_1}, ..., a_{i_j})$, will be included in emc.*

Definition 13 (jumpback set) *The* jumpback set *of a dead-end $\vec{a}_i$ is defined to include the $par(\vec{a_j})$ of all the dead-ends $\vec{a_j}$, $j > i$, that occurred in the current session of x_i. Formally,*

$$J_i = \bigcup \{par(\vec{a_j}) | \vec{a_j} \ dead-end \ in \ x_i's \ session\}$$

The variables $par(\vec{x_i})$ play the role of ancestors in the graphical scheme while J_i plays the role of induced ancestors. However, rather than being elicited from the graph, they are dependent on the particular value instantiation and are uncovered during search. Consequently, using the same arguments as in the graph-based case, it is possible to show that:

Proposition 4 *Given a dead-end tuple $\vec{a_i}$, the latest variable in its jumpback set J_i is the earliest variable that is safe to jump back to.* □

Conflict-directed backjumping
Input: A constraint network R and an ordering $d = \{x_1, ..., x_n\}$.
Output: Either a solution if one exists or a decision that the network is inconsistent.

0. (Initialize) $cur \leftarrow 0$.
1. (Step forward.) If x_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, $cur \leftarrow cur + 1$. Set $D'_{cur} \leftarrow D_{cur}$. Set $J_{cur} \leftarrow \emptyset$.
2. Select a value $a \in D'_{cur}$ that is consistent with all previous variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop a from from D'_{cur}.
   (c) For $1 \leq i < cur$ in ascending order do,
   if $\vec{a_i}$ conflicts with $x_{cur} = a$ then add x_i to J_{cur} and go to 2(a).
   (d) Instantiate $x_{cur} \leftarrow a$ and go to 1.
3. (Backjump.) If $J_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent". Otherwise, set $J_{temp} \leftarrow J_{cur}$, set cur equal to the last variable in J_{temp}. Set $J_{cur} \leftarrow J_{cur} \cup J_{temp} - \{x_{cur}\}$. Reinitialize: for all $j > cur$, set $J_j \leftarrow \emptyset$. Go to 2.

Figure 9: Algorithm conflict-directed backjumping.

Algorithm conflict-directed backjumping is presented in Figure 9. It computes the jumpback sets for each variable. In summary,

Proposition 5 *Algorithm conflict-directed backjumping jumps back to the latest variable in the dead-end's jumpback set, and therefore it is optimal.* □.

Example 12 *Consider the problem of Figure 4 using ordering* $d_1 = x_1, ..., x_7$. *Given the dead-end at* x_7 *and the assignment* $\vec{a_6} = (blue, green, red, red, blue, red)$, *the jumpback set is* $(x_1 = blue, x_3 = red)$ *since it accounts for eliminating all the values of* x_7. *Therefore, algorithm conflict-directed backjumping jumps to* x_3. *Since* x_3 *is an internal dead-end whose own emc set is* $\{x_1\}$, *the jumpback set of* x_3 *includes just* x_1, *and the algorithm jumps again, back to* x_1.

5.5 i-Backjumping

The notion of a conflict set is based on a simple restriction: we identify conflicts of a single variable only. What if we lift this restriction so that we can look a little further ahead? For example, when backtracking instantiates variables in its forward phase, what happens if it instantiates two variables at the same time?

In Section 7, we will discuss various attempts at looking ahead. However, at this point, we wish to mention a very restricted type of look-ahead that can be incorporated naturally into backjumping. We define a set of parameterized backjumping algorithms, called *i-backjumping* algorithms, where i indexes the number of variables consulted in the forward phase. All algorithms use jumping back optimally at both leaf and internal dead-ends, as follows. Given a fixed ordering of the variables, instantiate them one at a time as does conflict-directed backjumping; note that conflict-directed backjumping is *1-backjumping*. However, when selecting a new value for the next variable, make sure the new value is both consistent with past instantiation, and consistently extendable by the next $i-1$ variables. This computation will be performed at any node and can be exploited to generate more refined conflict sets than in 1-backjumping, namely, conflict sets whose no-goods conflict with i future variables. This leads to the concept of *level-i conflict sets*. A tuple $\vec{a_j}$ is a level-i conflict set if it is not consistently extendable by the next i variables. Once a dead-end is identified by i-backjumping, its associated conflict set is a level-i conflict set. The algorithm can assemble the earliest level-i conflict set and jump to the latest variable in this set exactly as done in 1-backjumping. The balance between computation overhead at each node and the savings on node generation should be studied empirically.

6 Learning Algorithms

The earliest minimal conflict set of Definition 12 is a no-good explicated by search and is used to focus backjumping. However, this same no-good may be rediscovered again and again while the algorithm explores different paths in the search space. By making this no-good explicit, in the form of a new constraint, we can make sure that the algorithm will not rediscover it and, moreover, that it may be used for pruning the search space. This technique, called *constraint recording*, is behind the learning algorithms described in this section [Dec90].

By *learning* we mean recording potentially useful information as that information becomes known to the problem-solver during the solution process. The information recorded is deduced from the input and involves neither generalization nor errors. An opportunity to learn (or infer) new constraints is presented whenever the backtracking algorithm encounters a dead-end, namely, when the current instantiation $\vec{a}_i = (a_1, ..., a_i)$ is a conflict set of x_{i+1}. Had the problem included an explicit constraint prohibiting this conflict set, the dead-end would never have been reached. At a dead-end, learning means recording a new constraint that makes explicit an incompatibility that already existed implicitly in a given set of variable assignments. There is no point, however, in recording at this stage the conflict set $\vec{a}_i$ itself as a constraint, because under the backtracking control strategy the current state will not recur.[3] Yet, when $\vec{a}_i$ contains

[3]Although recording this constraint may be useful if the same initial set of constraints is

one or more subsets that are in conflict with x_i, recording these smaller conflict sets as constraints may prove useful in the continued search; future states may contain these conflict sets, and they exclude larger conflict sets as well.[4]

In order to speed up search, the target of learning is to identify conflict sets that are as small as possible, namely, minimal. As noted above, one obvious candidate is the earliest minimal conflict set, which is identified anyway for conflict-directed backjumping. Alternatively, if only graph information is used, the graph-based conflict set could be identified and recorded. Another (extreme) option is to learn and record *all* the minimal conflict sets associated with the current dead-end.

In learning algorithms, the savings from possibly reducing the amount of search by finding out earlier that a given path cannot lead to a solution must be balanced against the costs of processing at each node generation a more extensive database of constraints.[5]

Learning algorithms may be characterized by the way they identify smaller conflict sets. Learning can be *deep* or *shallow*. Deep learning records only the minimal conflict sets. Shallow learning allows recording of nonminimal conflict sets as well. Learning algorithms may also be characterized by how they bound the arity of the constraints recorded. For example, the algorithm may record a single no-good or multiple no-goods per dead-end, and it may allow learning at leaf dead-ends only or at internal dead-ends as well. Constraints involving many variables are less frequently applicable, require more space to store, and are more expensive to consult than constraints having fewer variables.

We present three types of learning: graph-based learning, deep learning, and jumpback learning. Each of these can be further restricted by the maximum arity of the constraints recorded, referred to as *bounded learning*. These cover the main alternatives, although there are numerous possible variations, each of which may be suitable for a particular class of instances.

6.1 Graph-based learning

Graph-based learning uses the same methods as graph-based backjumping to identify a no-good, namely, information on conflicts is derived from the constraint graph alone. Given an i-leaf dead-end $(a_1, ..., a_i)$, values associated with the ancestors of x_{i+1} are identified and included in the conflict set.

Example 13 *Suppose we try to solve the problem in Figure 10 along the or-*

expected to be queried in the future.

[4] The type of learning discussed here can be viewed as *explanation-based learning*, in which learning can be done by recording an explanation or a proof for some concept of interest. The target concept in this case is a no-good whose proof is the conflict set, and the algorithm records a summary of this proof.

[5] We make the assumption that the computer program represents constraints internally by storing the invalid combinations. Thus, increasing the number of no-goods through learning will increase the size of the data structure and slow down retrieval.

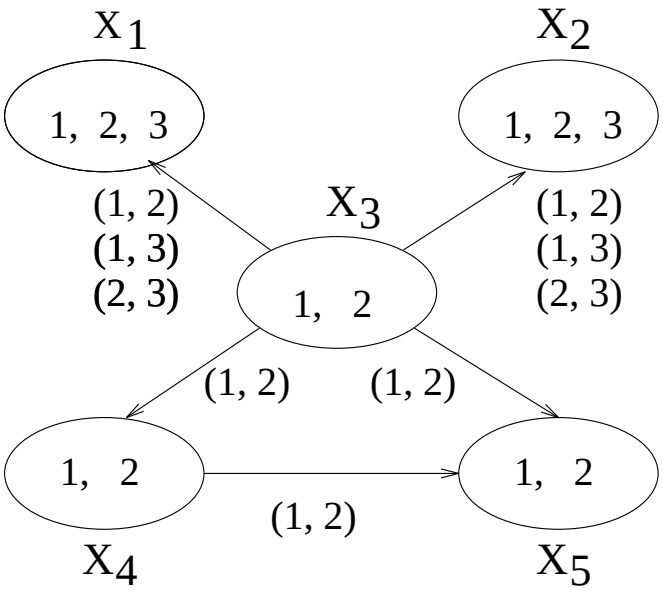


Figure 10: A small CSP. The constraints are: $x_3 < x_1, x_3 < x_2,\ x_3 < x_5,\ x_3 < x_4,\ x_4 < x_5$. The allowed pairs are shown on each arc.

Graph-based backjumping-learning
Input: A constraint network R and an ordering $d = \{x_1, ..., x_n\}$.
Output: Either a solution if one exists or a decision that the network is inconsistent.

3. If $I_{cur} = \emptyset$ (there is no variable in the induced ancestor set), exit with "inconsistent". Otherwise,
(learning:) set $I_{temp} \leftarrow I_{cur}$ and record a no-good tuple over I_{temp}, $\Pi_{I_{temp}}(a_1, ..., a_{cur})$, then
(backjump.) set cur equal to the index of the last variable in I_{cur}. Set $I_{cur} = I_{cur} \cup I_{temp} - \{x_{cur}\}$. For all $j > cur$ set $I_j \leftarrow anc(x_j)$. Go to 2.

Figure 11: Algorithm graph-based backjumping-learning

dering $d = x_1, x_2, x_3, x_4, x_5$. After instantiating $x_1 = 2, x_2 = 2, x_3 = 1, x_4 = 2$, the dead-end at x_5 will cause graph-based learning to record the conflict set $(x_3 = 1, x_4 = 2)$ since x_3 and x_4 are both connected with x_5 and since this is a conflict set discovered by graph-based backjumping.

The complexity of learning at each dead-end, in this case, is $O(n)$, since each variable is connected to at most $n - 1$ other variables. To augment graph-based backjumping with graph-based learning, we change the last step of graph-based backjumping (Figure 8) so that it has two components: backjumping and learning (see Figure 11). Note that $\Pi_I(a_1, ..., a_i)$ denotes the projection of the subtuple, $(a_1...a_i)$ over the subset of variables in I.

6.2 Deep learning

Identifying and recording only minimal conflict sets constitutes *deep learning*. Discovering all minimal conflict sets means acquiring all the possible information out of a dead-end. For the problem in Example 13, deep learning will record the minimal conflict set $(x_4 = 2)$ instead of the nonminimal conflict set recorded by graph-based learning. Although this form of learning is the most accurate, its cost is prohibitive if we want all minimal conflict sets and in the worst case, exponential in the size of the initial conflict set. If r is the cardinality of the graph-based conflict set, we can envision a worst case where all the subsets of size $r/2$ are minimal conflict sets. The number of such minimal conflict sets will be

$$\left(\begin{array}{c} r \\ \frac{1}{2}r \end{array} \right) \cong 2^r,$$

which amounts to exponential time and space complexity at each dead-end. Discovering *all* minimal conflict sets can be implemented by enumeration: first, recognize all conflict sets of one element; then, all those of two elements; and so on. In general, given that all minimal conflict sets of size $1, ..., i - 1$ were

recognized, find all the size-i conflict sets that do not contain any smaller conflict sets. Deep learning is a backtracking strategy that may require exponential space.

6.3 Jumpback learning

To avoid the explosion in time and space of full deep learning one may settle for identifying just one conflict set, minimal relative to prefix conflict sets. The obvious candidate is the *jumpback set* for leaf and internal dead-ends as it was explicated by conflict-directed backjumping. *Jumpback learning* [FD94a] uses this jumpback set as the conflict set. Because the conflict set is calculated by the underlying backjumping algorithm, the complexity of computing the conflict set is constant.

Example 14 *For the problem in Example 13, jumpback learning will record* $(x_3 = 1, x_4 = 2)$ *as a new constraint upon a dead-end at* x_5*. The algorithm selects these two variables because it first looks at* $x_3 = 1$ *and notes that* x_3 *conflicts with* $x_5 = 1$*. Then, proceeding to* $x_4 = 2$*, the algorithm notes that* x_4 *conflicts with* $x_5 = 2$*. At this point, all values of* x_5 *have been ruled out, so the conflict set is complete. It will record the same conflict set as graph-based learning.*

In general, graph-based learning records the largest size constraints, and deep learning records the smallest. The virtues of graph-based learning are mainly theoretical; we do not advocate using this algorithm in practice since jumpback learning is always more powerful. Neither do we recommend using deep learning, because its cost is usually prohibitive. Algorithm backjumping-learning, which combines conflict-directed backjumping with learning, is presented in Figure 13. Only the third step of the algorithm is given, as the first two steps are identical to conflict-directed backjumping. Note that the algorithm should consult all its constraints, old and new, in its various steps.

6.4 Bounded learning

Each learning algorithm can be compounded with a restriction on the size of the conflicts learned. When conflict sets of size greater than i are ignored, we get i-order graph-based learning, i-order jumpback learning, or i-order deep learning. When restricting the arity of the recorded constraint to i, the *bounded learning* algorithm has an overhead complexity that is time and space exponentially bounded by i.

In Figure 12 we present the search space of the problem in Figure 10 explicated by naive backtracking and by backtracking augmented with graph-based second-order learning; all the branches below the cut lines in Figure 12 will be generated by the former but not by the latter.

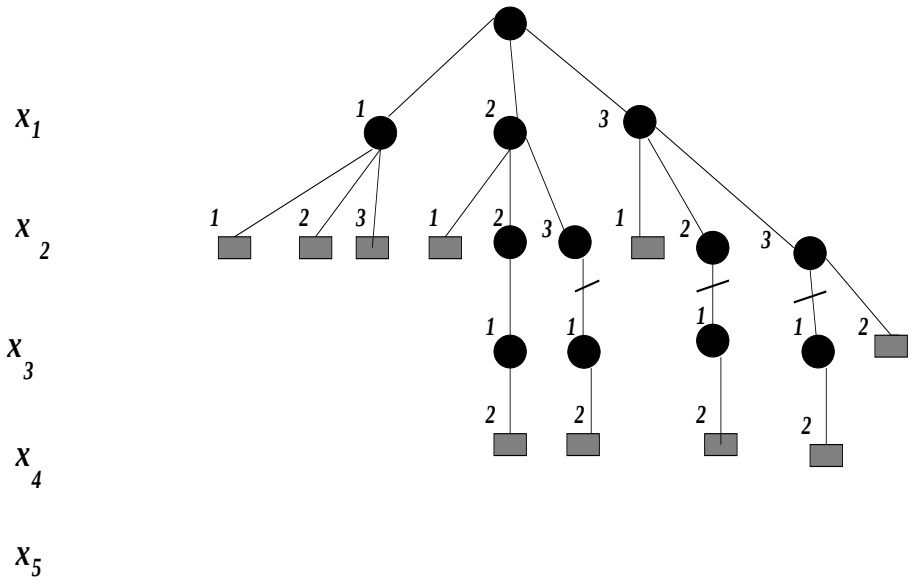


Figure 12: Search space explicated by backtracking, without and with second-order learning.

Backjump-learning
Input: A constraint network R. An ordering $d = \{x_1, ..., x_n\}$.
Output: a solution if exists, or a decision that the network is inconsistent.

3 If $J_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent". Otherwise, (Learning:) Let $J_{temp} \leftarrow J_{cur}$. Record a nogood constraint over J_{temp}: $\Pi_{J_{temp}}(a_1, ..., a_{cur})$, and
(Backjumping:) set *cur* equal to the index of the last variable in J_{cur}. Set $J_{cur} \leftarrow J_{cur} \cup J_{temp} - \{x_{cur}\}$. For all $j > cur$, set $J_j \leftarrow \emptyset$. Go to 2.

Figure 13: Algorithm backjump-learning (step 3)

6.5 Complexity of backtracking with learning

We will now show that graph-based learning yields a useful complexity bound on the backtracking algorithm's performance parameterized by the induced width w^*. Graph-based learning is the most conservative learning algorithm (when excluding arity restrictions) so its complexity bound will be applicable to all the corresponding variants of learning discussed here.

Theorem 6 *Let d be an ordering of a constraint graph, and let $w^*(d)$ be its induced width. Any backtracking algorithm using ordering d with graph-based learning has a space complexity of $O((nk)^{w^*(d)+1})$ and a time complexity of $O((2nk)^{w^*(d)+1})$, where n is the number of variables and k bounds the domain sizes.*

Proof: Graph-based learning has a one-to-one correspondence between dead-ends and conflict sets. It is easy to see that backtracking with graph-based learning along d records conflict sets of size $w^*(d)$ or less. Therefore the number of dead-ends is bounded by

$$\sum_{i=1}^{w^*(d)} \binom{n}{i} k^i = O((nk)^{w^*(d)+1})$$

This gives the space complexity. Since deciding that a dead-end occurred requires testing all constraints defined over the dead-end variable and at most $w^*(d)$ prior variables, at most $O(2^{w^*(d)})$ constraints are checked per dead-end, yielding a time complexity bound of $O((2nk)^{w^*(d+1)})$. □

Recall that the time complexity of graph-based backjumping is bounded by $O(exp(m_d^*))$, where m_d^* is the depth of the DFS tree of the corresponding ordered induced graph, while the algorithm requires only linear space. Clearly, $m_d^* \geq w^*(d)$. It can be shown [BM96] that for any graph $m_d^* \leq logn \cdot w^*(d)$. Therefore, to reduce the time bound of graph-based backjumping by a factor of

$\log n$, we need to invest $O(exp(w^*(d))$ in space, augmenting backjumping with learning.

6.6 Backmarking

Backmarking [Gas79] is a caching-type algorithm that focuses on lowering the cost of node generation. It reduces the number of consistency checks required to generate each node, without trying to prune the search space itself.

By keeping track of where consistency checks failed in the past, backmarking can eliminate the need to repeat, unnecessarily, previously performed checks. We describe the algorithm as an improvement to a naive backtracking algorithm.

Recall that a backtracking algorithm moves either forward or backward in the search space. Suppose that the current variable is x_{cur} and that x_i is the earliest variable in the ordering which changed its value since the last visit to x_{cur}. Clearly, any testing of values of x_{cur} against variables preceding x_i will produce the same results: If failed against earlier instantiations, it will fail again; if it succeeded earlier, it will succeed again. Therefore, by maintaining the right information from earlier parts of the search, values in the domain of x_{cur} can either be recognized immediately as inconsistent or else be tested only against prior instantiations starting from x_i on. Backmarking implements this idea by maintaining two additional tables. First, for each variable x_i and for each of its values a_v, backmarking remembers the earliest prior variable x_p such that the current partial instantiation $\vec{a}_p$ conflicted with $x_i = a_v$. This information is maintained in a table with elements $M_{i,v}$. (Note that this assumes that constraints involving earlier variables are tested first.) If $x_i = a_v$ is consistent with all earlier partial instantiations $\vec{a}_j$, $j < i$, then $M_{i,v} = i$. For instance, $M_{10,a_2} = 4$ means that $\vec{a}_4$ as instantiated was found inconsistent with $x_{10} = a_2$ and that $\vec{a}_1, \vec{a}_2$, and $\vec{a}_3$ did not conflict. The second table, with elements low_i, records the earliest variable that changed value since the last time x_i was instantiated. This information is put to use at every step of node generation. If $M_{i,v}$ is less than low_i, then the algorithm knows that the variable pointed to by $M_{i,v}$ did not change and that $x_i = a_v$ will fail again when checked against $\vec{a}_{M_{i,v}}$, so no further consistency checking is needed at this node. If $M_{i,v}$ is greater than or equal to low_i, then $x_i = a_v$ is consistent with $\vec{a}_j$, for all $j < low_i$, and those checks can be skipped. The algorithm is presented in Figure 14.

Example 15 *Consider again the example in Figure 4, and assume backmarking uses ordering* d_1. *Once the first dead-end is encountered at variable* x_7 *(see the search space in Figure 5), table* M *has the following values:* $M(1, blue) = 1, M(2, green) = 2, M(3, blue) = 1, M(3, red) = 3\ M(4, blue) = 1, M(4, red) = 4, M(5, blue) = 5, M(6, green) = 2, M(6, red) = 6, M(7, red) = 3, M(7, blue) = 1$. *Upon backtracking from* x_7 *to* x_6, $low(7) = 6$ *and* $x_6 = teal$ *is assigned. When trying to instantiate a new value for* x_7, *the algorithm notices that the* M *values are smaller than* $low(7)$ *for both values of* x_7 *(red, blue) and, consequently,*

Backmarking
Input: A constraint network R and an ordering $d = \{x_1, ..., x_n\}$.
Output: Either a solution if one exists or a decision that the network is inconsistent.

0. (Initialize tables) Set all $M_{i,v} \leftarrow 0$, $low_i \leftarrow 0$.
1. (Step forward.) If x_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set $cur \leftarrow cur+1$. Set $D'_{cur} \leftarrow D_{cur}$.
2. Select a value $a \in D'_{cur}$ which is consistent with all previous variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop a_v from D'_{cur}. (v is the index of the domain value popped.)
   (c) If $M_{cur,v} < low_{cur}$, then go to 2(a).
   (d) In order, $low_{cur} \leq i < cur$ for each variable x_i do,
   if $\vec{a}_i$ as instantiated conflicts with $x_{cur} = a$, then set $M_{cur,v} \leftarrow i$ and go to Step 2(a).
   (e) Instantiate $x_{cur} \leftarrow a_v$, set $M_{cur,v} \leftarrow cur$, and go to 1.
3. (Backtrack.) If x_{cur} is the first variable, exit with "inconsistent". Otherwise, for all x_i after x_{cur+1}, if $cur < low_i$, then set $low_i \leftarrow cur$. Set $low_{cur} \leftarrow cur-1$. Set $cur \leftarrow cur - 1$. Go to 2.

Figure 14: Algorithm backmarking.

both values can be determined to be inconsistent without performing any other consistency checks.

Theorem 7 *The search space explored by backtracking and backmarking is identical when given the same variable ordering and value ordering. The cost of node expansion by backmarking is always equal to or smaller than the cost of node expansion by backtracking.*

Proof: Clearly, although backmarking does no pruning of the search space, it does replace a certain number of consistency checks with table look-ups and table updating. The cost of table updating for each node generation is constant. The overall extra space required is $O(n \cdot k)$, where n is the number of variables and k the number of values. For each new node generated, one table look-up may replace $O(n)$ consistency tests. □

All of the enhancements to backtracking introduced so far are compatible with backmarking as long as the variable ordering remains fixed.

7 Look-ahead Strategies

Look-ahead strategies perform a limited amount of forward reasoning (or inference) so that decisions regarding which variable to select next and what value of the selected variable to try next are done in a more informed way. In variable selection, we seek to control the size of the remaining search space. In value selection, we seek a value that is most likely to lead to a consistent solution. Such decisions are accomplished through constraint propagation, namely, path- or arc-consistency. Since such computations are carried out at each state, they need to be limited in order to be cost effective. In the following sections, we describe look-ahead strategies based on applying a limited amount of arc-consistency.

7.1 Look-ahead strategies for value selection

The main goal of value selection strategies is to evaluate the incremental effect of a candidate value assignment on each future variable, namely, to perform a limited amount of forward inference in the hope of being able to determine whether the value in question leads to a dead-end if added to the current partial assignment. If it does, it should not be selected.

Look-ahead methods based on arc-consistency range from applying full arc-consistency at each value selection (a strategy that we term *full look-ahead*) to using a very restricted amount of arc-consistency (a strategy known as *forward-checking*). Many intermediate levels of constraint propagation can be defined. The key issue is determining a cost-effective balance between look-ahead's accuracy and its overhead.

The *full look-ahead* algorithm [HE80][6] performs at each node in the search space, one full cycle of arc-consistency[7]. An arc-consistency algorithm is applied to the problem instantiated with the current assignments and the tentative value being considered. If, as a result, one of the future domains becomes empty, the tentative value will lead to a dead-end and it should be excluded. If none of the future domains becomes empty, the tentative value will be selected. More sophisticated procedures may use this analysis to prioritize value selection. The full look-ahead strategy is highly successful on a class of vision instances [Wal75], but it is sometime too costly; although provably effective in pruning the search space, its overhead sometimes exceeds its savings. This observation leads to value selection methods based on weaker forms of constraint propagation. We present two such algorithms, *partial look-ahead* and *forward-checking* , next.

Algorithm partial look-ahead uses only *directional arc-consistency* at each state. In other words, for each candidate value, it processes each constraint just once and the effect of shrunken domains is propagated. The procedure is given in Figure 15. At step 2c, first, the domain of each variable is shrunk so that each of its remaining values is consistent with the current partial assignment $\vec{a}_{cur}$, and then for each pair of future variables x_i, x_j, the domain of x_j is restricted so that it is compatible with the partial assignment $(\vec{a}_{cur}, x_i = a_i)$ for some value a_i of x_i, when $i \leq j$. Algorithm forward-checking uses an even more limited form of constraint propagation which is sometime more cost effective. In forward-checking, the effect of a tentative value selection is propagated *separately* to each of the future variables. Here, at step 2c, only part of partial look-ahead is executed. Namely, each future variable x_j, where $j > cur$, is tested relative to the current (tentative) value assignment *only*. If, as a result of forward-checking or partial-looking-ahead, the domain of one of these future variables becomes empty, this value is not selected and the next candidate value is tried. Algorithm forward-checking is presented in Figure 16.

Clearly, algorithms that utilize look-ahead strategies reach dead-ends earlier in the search than algorithms that do not use such strategies.

Example 16 *Consider the problem in Figure 4. If we use algorithm forward-checking along the ordering $d_1 = x_1, ..., x_7$, instantiating $x_1 = red$ reduces the domains of x_3, x_4, and x_7 but none of them becomes empty. Subsequently, instantiating $x_2 = blue$ does not affect any future domain. Next, variable x_3 has only blue in its domain, and its selection causes the domain of x_7 to become empty. Therefore, $x_3 = blue$ is rejected, and x_3 becomes a dead-end variable. If we use algorithm partial look-ahead, selecting $x_1 = red$ leaves only blue for x_3, and when x_7 is tested for consistency with $x_1 = red$, none of its values is consistent. Therefore, $x_1 = red$ will be rejected by partial look-ahead, right then and there.*

[6] In [HE80], the term indicates just one pass on each arc of the arc-consistency algorithm.
[7] We assume familiarity with basic algorithms for enforcing arc-consistency [MF85].

Partial look-ahead
Input: A constraint network R and an ordering $d = \{x_1, ..., x_n\}$.
Output: Either a solution if one exists or a decision that the network is inconsistent.

0. (Initialize) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. If x_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set $cur \leftarrow cur + 1$.
2. Select a value $a \in D'_{cur}$ that is consistent with at least one remaining value of each future variable. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop a from D'_{cur}.
   (c) Tentatively instantiate $x_{cur} \leftarrow a$.
   i. For $i = cur + 1, ..., n$, tentatively remove all v in D'_i that conflict with $\vec{a}_{cur}$.
   ii. For $i = cur + 1, ..., n$, For $j = i + 1, n$, do:
   Tentatively remove all v in D'_j without a value consistent with $\vec{a}_{cur}$ and $x_i \leftarrow a_i$ for some consistent value a_i of x_i. If doing steps i and ii so results in $D'_j = \emptyset$ for some j, then restore the D'_i's to their values before step 2(c) was begun and go to 2(a).
   (d) Instantiate $x_{cur} \leftarrow a$ and go to 1.
3. (Backtrack step.) If there is no previous variable, exit with "inconsistent". Otherwise, set $cur \leftarrow cur - 1$. Reset all D' sets to the way they were before x_{cur} was last instantiated. Go to 2.

Figure 15: Algorithm partial look-ahead.

Forward-checking
Input: A constraint network R and an ordering $d = \{x_1, ..., x_n\}$.
Output: Either a solution if one exists or a decision that the network is inconsistent.

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

Figure 16: Algorithm forward-checking.

Full look-ahead, which is not explicitly specified here, prunes larger portions of the search space than does partial look-ahead, which, in turn, prunes more than forward-checking does. In general, the stronger the level of constraint propagation, the smaller the search space explored and the higher the computational overhead. Nevertheless, empirical evaluations often find forward-checking to be the most cost effective of the three look-ahead strategies on smaller and easier problems. As larger and more difficult problems are experimented with, higher levels of look-aheads become more useful. The balance between overhead and pruning is the focus of many current investigations [FD95, SF94, Bak95]. Examples exist for which increasing constraint propagation incrementally results in exponential pruning and, vice versa, look-ahead's having an exponential overhead may not cause any pruning.

We next present a relationship between forward-checking and the simplest form of backjumping.

Proposition 6 *[KvB97] When using the same variable ordering Gaschnig's backjumping always explores every node explored by forward-checking.* □.

We end this section by presenting a relationship between the structure of the constraint graph and some forms of look-aheads.

Definition 14 (cycle-cutset) *Given an undirected graph, a subset of nodes in the graph is called a* cycle-cutset *if its removal results in a graph having no cycles.*

Proposition 7 *A constraint problem whose graph has a cycle-cutset of size c can be solved by partial look-ahead algorithm in time of $O((n-c)\cdot k^{c+2})$.*

Proof: Once a variable is instantiated, the flow of interaction through this variable is terminated. This can be expressed graphically by deleting the corresponding variable from the constraint graph. Therefore, once a set of variables that forms a cycle-cutset is instantiated, the remaining problem can be perceived as a tree. A tree can be solved by directional arc-consistency, and therefore partial look-ahead performing directional arc-consistency at each node is guaranteed to solve the problem if the cycle-cutset variables initiate the search ordering. Since there are k^c possible instantiations of the cutset variables, and since each remaining tree is solved in $(n-c)k^2$, the complexity follows. For more details see [Dec90]. □

7.2 Look-ahead strategies for variable selection

Variable ordering has a tremendous effect on the size of the search space. Empirical and theoretical studies have shown that there are several effective static orderings that result in smaller search spaces [DM94]. In particular, the *min-width ordering* and the *max-cardinality ordering*, both of which use information

from the constraint graph, are quite effective. The min-width heuristic orders the variables from last to first by selecting, at each stage, a variable in the constraint graph that connects to the minimal number of variables that have not been selected yet. The max-cardinality heuristic selects an arbitrary first variable, and then, in each successive step, selects as the next a variable connected to a maximal set of the variables already selected.

In the rest of this section, we focus on variable orderings that are decided dynamically during search. The objective is to select the next variable that constrains the most the remainder of the search space. Given a current partial solution $\vec{a_i}$, we wish to determine the domain values for each future variable that are consistent with $\vec{a_i}$ and likely to lead to a solution. The fewer such candidates, the stronger the selected variable's expected pruning power. We may estimate the domain sizes of future variables using various levels of look-ahead propagation.

As in the case of value selection, we have a *full look-ahead variable ordering* strategy that performs full arc-consistency at each value selection, tentatively prunes the domains of future variables, and subsequently selects a variable with the smallest domain. We also have a *partial look-ahead variable ordering* strategy that performs only directional arc-consistency at each decision point and then selects the variable with the smallest domain. Finally, as in forward-checking, the weakest level of constraint propagation may be incorporated. All such methods are called *dynamic variable ordering* (DVO) strategies.

The last alternative has proven particularly cost effective in many empirical studies.[8] We call this weakest form of DVO *dynamic variable forward-checking* (DVFC), since it is based on the forward-checking level of constraint propagation. Given a state $\vec{a_i} = (a_1, ..., a_i)$, the algorithm updates the domain of each future variable, D'_j, to include only values consistent with $\vec{a_i}$. Then, a variable with a domain of minimal size is selected. If any future variable has an empty domain, it is moved to be next in the ordering, and a dead-end will occur when the next variable becomes the current variable. The algorithm is described in Figure 17.

Example 17 *Consider again the example in Figure 4. Initially, all variables have domain size of 2 or more. We pick x_7 whose domain size is 2, and choose value x_7 = blue. Propagating this choice to each future variable restricts the domains of x_3, x_4, and x_5 to single values. We select x_3, assign it its only possible value,* red*, and propagate this assignment, which adds variable x_1 to the variables with a singleton domain. We choose x_1 and its only consistent value, red. At this point, after propagating this choice, we see that x_4 has an empty domain. We select this variable, recognize it as a dead-end and backtrack.*

[8] As far as we know, no empirical testing has been carried out for either full look-ahead or partial look-ahead variable orderings.

Algorithm DVFC
Input: A constraint network R.
Output: Either a solution if one exists or a decision that the network is inconsistent.

0. (Initialize) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$. Set $cur \leftarrow 1$. Set x_1 to the variable participating in the most constraints.
1. (Step forward.) If $cur = n$, then all variables have value assignments; exit with this solution.
2. Select a value $a \in D'_{cur}$. Do this as follows.
   (a) If $D'_{cur} = \emptyset$, reset the domains of all future variables to their original domains and go to 4.
   (b) Pop a from D'_{cur} and instantiate $x_{cur} \leftarrow a$.
3. Choose the next variable as follows:
   (a) Examine all uninstantiated variables x_i. For each, remove all v in D'_i that conflict with $\vec{a}_{cur-1}$ and $x_{cur} = a$.
   (b) Set $cur \leftarrow cur + 1$. Set an uninstantiated variable x_i with the smallest remaining domain D'_i to x_{cur}. Go to 1.
4. (Backtrack step.) If there is no previous variable, exit with "inconsistent". Otherwise, set $cur \leftarrow cur - 1$. Go to 2.

Figure 17: Algorithm DVFC.

Since forward-checking for value selection and DVFC use the same level of constraint propagation, using both in one algorithm is natural. It is interesting to note, however, that once an algorithm implements variable ordering decisions with a given amount of constraint propagation, the corresponding value selection decision is accomplished with no extra effort. For example, DVFC already accomplishes forward-checking. To see this, assume that DVFC has selected x as the next variable. Next, a value in the domain of x consistent with past values is selected for this variable. DVFC will now propagate the effect of this *committed* value selection to all future variables. If the domain of one of these future variables becomes empty, the associated variable is selected since it has the most restricted domain, a dead-end occurs, and the algorithm returns to x and chooses its next value. This cycle of value selection and retraction is exactly what would have happened if we had done forward-checking once a variable is selected. In conclusion:

Proposition 8 *[BvR95] Algorithm DVFC performs forward-checking for value selection as well as variable selection.* □

7.3 Implementation and complexity

The cost of node expansion when implementing a look-ahead strategy can be controlled if certain information is cached and maintained. One possibility is to maintain for each variable a table containing viable values relative to the partial solution currently being assembled. When testing a new value of the current variable or after committing to a value for the current variable, the tables will tentatively be updated, once constraint propagation has been applied. This strategy requires an additional $O(n \cdot k)$ space, and the cost of node generation will be $O(e_d \cdot k)$, where e_d bounds the number of constraints mentioning each variable and k is the domain size. Still, whenever a dead-end occurs, the algorithm has to recompute the tables associated with the state in the search to which the algorithm retracted. This operation may require $O(n \cdot e_d \cdot k)$ consistency tests.

A more sophisticated approach is to keep a table of pruned domains for each variable and for each level in the search tree. This results in additional space of $O(n^2 \cdot k)$. Upon reaching a dead-end, the algorithm jumps back to a particular node (and a particular level in the search tree) and uses the tables maintained at that level. Consequently, the only cost of nodes generation is that of updating the table. The time complexity of this operation is bounded by $O(e_d \cdot k)$.

8 Hybrids of Backjumping with look-ahead Approaches

The idea of combining roughly orthogonal techniques into a single constraint algorithm is an old one, dating back at least to Gaschnig's DEEB algorithm

[Gas79]. Because solving constraint problems is an NP-hard task, it is reasonable to conjecture that any polynomial technique, or combination of polynomial techniques, will be useful on some problems. In this section we briefly survey ways that backjumping and one or more other constraint algorithms or heuristics can be combined, focussing on recent work. The techniques we discuss are look-ahead algorithms auch as dynamic variable ordering, dynamic value ordering and backmarking. The combination of backjumping and learning was discussed in Section 6.

8.1 Look-ahead Algorithms

In Section 7 several "look-ahead" algorithms were described that do a limited amount of consistency enforcing after each instantiation. The most frequently used look-ahead algorithm is forward checking. Backjumping and forward checking each make a tradeoff, doing extra work at one phase of the search in order to reduce the amount of work required later. Forward checking does more work early in the search than does backtracking, since each instantiation of an early variable requires checking consistency with each value of each later variable. This extra work, however, frequently leads to a significantly reduced search space.

Prosser [Pro93b] proposes a "BJ-FC" algorithm that combines conflict-directed backjumping and forward checking and uses a fixed variable ordering. In his experiments the combined algorithm is usually superior to either backjumping or forward checking alone.

Frost and Dechter [FD94b] describe "BJ+DVFC", (which was called there BJ+DVO) an algorithm which combines conflict-directed backjumping with the variable ordering DVFC. It can also be views as extending BJ-FC with dynamic variable ordering (see Figure 18). Step 1 of BJ+DVFC utilizes the variable ordering heuristic described explicitly in Figure 19, which is the one discussed in section 7.2. Namely, it selects the future variable with the smallest remaining domain.

The presence of a forward checking style look-ahead mechanism affects how the J's (ancestor sets) in conflict-directed backjumping are updated in Step 2. In backjumping, earlier variables are added to J_{cur}. In BJ+DVFC, x_{cur} is added to the ancestor set of a future variable. This reflects the fact that pure look-back algorithms (such as backjumping) compare the current variable with earlier variables, while an algorithm having a look-ahead component, such as BJ+DVFC, removes values from the domains of future variables.

Another change to Step 2 in conflict-directed backjumping marks a departure from the earlier versions of the backjumping algorithm. In the earlier algorithms, if a value a from D'_{cur} was found incompatible, either by a look back or a look forward check with other variables, there was a "go to (a)" step which continued the search with the next value of the current variable. In contrast, BJ+DVFC's step 2 always proceeds to step 1 after assigning a value to x_{cur}. If that value

Backjumping with DVFC

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$. Set $J_i \leftarrow \emptyset$ for $1 \leq i \leq n$.

1. (Step forward.) If x_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of a variable, selected according to a VARIABLE-ORDERING-HEURISTIC (see Fig. 19). Set $J_{cur} \leftarrow \emptyset$.

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

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

   (b) Pop a from D'_{cur} and instantiate $x_{cur} \leftarrow a$.

   (c) Examine the future variables $x_i, cur < i \leq n$. For each a in D'_i, if $x_i = a$ conflicts with $\vec{a}_{cur}$ then remove a from D'_i and add x_{cur} to J_i; if D'_i becomes empty, go to (d) (without examining other x_i's).

   (d) Go to 1.

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

Figure 18: The BJ+DVFC algorithm.

causes the domain of a future variable x_e to become empty, then the variable ordering heuristic will select x_e to be the next variable, and after step 2 (a) (for x_e with an empty domain) is executed, control will go to the backjump step, step 3. The backjump from x_e will of course be a step back to the immediately preceeding variable, since it was the instantiation of that variable which caused x_e to have an empty domain. Thus in BJ+DVFC, a backjump from a leaf dead-end is always to the immediately preceeding variable. The fact that selecting to be next a variable with an empty domain makes Gaschnig's backjumping – that is, backjumping from leaves in the search tree – redundant was noted in [BvR95] and [KvB94].

Step 3 of BJ+DVFC is identical to the step 3 of the conflict-directed backjumping algorithm in Figure 9. The most recent variable in the J_{cur} set is identified, and that variable becomes current, with its parent set being merged with the parent set of the dead-end variable.

BJ+DVFC combines conflict-directed backjumping, forward checking-style look-ahead, and dynamice variable ordering. Experimental results in [FD94b, Fro97] indicate that the additional overhead of this more complex algorithm pays off, in comparison to backjumping or forward checking alone, on sufficiently

VARIABLE-ORDERING-HEURISTIC

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

Figure 19: The variable ordering heuristic used by BJ+DVO.

difficult problem instances.

It is also be possible to combine backjumping with a look-ahead algorithm that enforces a greater degree of consistency. For instance, arc-consistency can be enforced after each variable instantiation [Pro95]. Will the combination be effective? Observe that if a high enough degree of consistency is enforced at each node in the search tree, then no search at all is necessary. In contrast, look-back methods cannot subsume look-ahead methods in the same way, because they only operate on a subset of the variables. In practice, however, performing a very limited amount of look-ahead usually seems to be more cost-effective.

8.2 Backjumping + Backmarking

Backjumping can easily be enhanced by integrating it with backmarking. All that is required is to modify backjumping to maintain and consult the two backmarking tables. See Figure 20.

8.3 Backjumping and Value Ordering

If a constraint satisfaction problem has a solution, knowing the right value for each variable would enable a solution to be found in a backtrack-free manner. Value-ordering heuristics are based on the principle of first trying the values which are more likely to lead to a consistent solution. As with variable-ordering heuristics, a value ordering can be static or dynamic, and a dynamic approach can be based on information derived during a look-ahead phase, if the algorithm includes one. Frost and Dechter [FD95] describe an algorithm that takes this approach, combining a "Look-ahead Value Ordering" heuristic that runs in conjunction with the BJ+DVFC algorithm. The essential idea is to estimate the probability that a value of the current variable is part of a solution by checking

Backjumping plus backmarking
Input: A constraint network R. An ordering $d = \{x_1, ..., x_n\}$.
Output: a solution if exists, or a decision that the network is inconsistent.

0. $M_{i,v} = low_i \leftarrow 0 \; \forall i, 1 \leq i \leq n, \forall v$.
1. If $cur = n$, then all variables have value assignements; exit with this solution. Otherwise, set $cur \leftarrow cur + 1$. Set $D'_{cur} \leftarrow D_{cur}$. Set $J_{cur} = \emptyset$.
2. Select a value $a \in D'_{cur}$ which is consistent with all previous variables. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop a_v from D'_{cur}.
   (c) If $M_{cur,v} < low_{cur}$, then go to (a).
   (d) Examine in order the previous variables x_i, $low_{cur} \leq i < cur$; if $\vec{a}_i$ as instantiated conflicts with $x_{cur} = a_v$ then set $M_{cur,v} \leftarrow i$, add x_i to J_{cur} and go to (a).
   (e) Instantiate $x_{cur} \leftarrow a_v$, set $M_{cur,v} \leftarrow cur$, and go to 1.
3. (Backjump.) If $J_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent". Otherwise, set $J_{temp} \leftarrow J_{cur}$. Set cur equal to the index of the last variable in J_{temp}. Set $J_{cur} \leftarrow J_{cur} \cup J_{temp} - \{x_{cur}\}$. Set $low_{cur+1} \leftarrow cur$. For all x_i after x_{cur+1}, if $cur < low_i$ then set $low_i = cur$. Go to 2.

Figure 20: Algorithm backjumping plus backmarking

its compatibility with the values of all future variables. This is done by processing, in forward-checking style, the search tree under each value of the current variable, and counting the number of values of future variables that would be eliminated. The value of the current variable which conflicts with the fewest future values is selected for instantiation.

An algorithm that ranks values in a similar spirit was developed by Dechter and Pearl [DP87]. The Advised Backtracking (ABT) algorithm uses an estimate of the number of solutions in each subproblem to choose which value to instantiate next.

9 Historical Remarks

Most current work on improving backtracking algorithms for solving constraint satisfaction problems use Bitner and Reingold's formulation of the algorithm [BR75]. One of the early and still one of the most influential ideas for improving backtracking's performance on constraint satisfaction problems was introduced by Waltz [Wal75]. Waltz demonstrated that often, when constraint propagation in the form of arc-consistency is applied to a two-dimensional line-drawing interpretation, the problem can be solved without encountering any dead-ends. This led to the development of various consistency-enforcing algorithms such as arc-, path- and k-consistency [Mon74, Mac77, Fre78]. However, Golomb and Baumert [GB65] may have been the first to informally describe this idea. Following Waltz's work and Montanari's seminal work on constraint networks [Mon74], Mackworth [Mac77] proposed interleaving backtracking with more general local consistency algorithms, and consistency techniques are used in Lauriere's Alice system [Lau78]. Explicit algorithms employing this idea have been given by Gaschnig [Gas79], who described a backtracking algorithm that incorporates arc-consistency; McGregor [McG79], who described backtracking combined with forward-checking, which is a truncated form of arc-consistency; Haralick and Elliott [HE80], who also added various look-ahead methods; and Nadel [Nad89], who discussed backtracking combined with many variations of partial arc-consistency. Gaschnig [Gas78] has compared Waltz-style look-ahead backtracking with look-back improvements that he introduced, such as backjumping and backmarking. In his empirical evaluation, he showed that on n-queen problems and on randomly generated problems of small size, backmarking appears to be the superior method. Haralick and Elliot [HE80] have done a relatively comprehensive study of look-ahead and look-back methods, in which they compared the performance of the various methods on n-queen problems and on randomly generated instances. Based on their empirical evaluation, they concluded that forward-checking, the algorithm that uses the weakest form of constraint propagation, is superior. This conclusion was maintained until very recently; when larger and more difficult problem classes were tested [SF94, FD95, FD96]. Empirical evaluation of backtracking with dynamic vari-

able ordering on the n-queen problem, was reported by [SS86]. Forward-checking lost its superiority on many problem instances, to full look-ahead and other stronger looking-ahead, variants. In the context of solving propositional satisfiability, Logemann, Davis and Loveland [DLL62] introduced a backtracking algorithm that uses look-aheads for variable selection in the form of *unit resolution*, which is similar to arc-consistency. To date, this algorithm is perceived as one of the most successful procedures for that task. Analytical average-case analysis for some backtracking algorithms has been pursued for satisfiability [Pur83] and for constraint satisfaction [HE80, Nud83, Nad90].

Researchers in the logic-programming community have tried to improve a backtracking algorithm used for interpreting logic programs. Their improvements, known under the umbrella name *intelligent backtracking*, focused on a limited amount of backjumping and constraint recording [Bru81]. The truth-maintenance systems area also has contributed to improving backtracking. Stallman and Sussman [SS77] were the first to mention no-good recording, and their idea gave rise to look-back type algorithms, called *dependency-directed backtracking* algorithms, that include both backjumping and no-good recording [McA90].

Later, Freuder [Fre82] and Dechter and Pearl [DP87, Dec90] introduced graph-based methods for improving both the look-ahead and the look-back methods of backtracking. In particular, *advice generation*, a look-ahead value selection method that prefers a value if it leads to more solutions as estimated from a tree relaxation, was proposed [DP87]. Dechter [Dec90] also described the graph-based variant of backjumping, which was followed by conflict-directed backjumping [Pro93b]. Other graph-based methods include graph-based learning (i.e., constraint recording) as well as the cycle-cutset scheme [Dec90]. The complexity of these methods is bounded by graph parameters: Dechter and Pearl [DP87] developed the induced-width bound on learning algorithms and Dechter [Dec90] showed that the cycle-cutset size, bounds some look-ahead methods. Frueder and Quinn [FQ87] noted the dependence of backjumping's performance on the depth of the DFS tree of the constraint graph, and Bayardo and Mirankar [BM96] improved the complexity bound. They also observed that with learning the time complexity of graph-based backjumping can be reduced by a factor of $\log n$ at an exponential space cost in the induced-width [RBM95].

Subsequently, as it became clear that many of backtracking's improvements are orthogonal to one another (i.e., look-back methods and look-ahead methods), researchers have more systematically investigated various hybrid schemes in an attempt to exploit the virtues in each method. Dechter [Dec90] evaluated combinations of graph-based backjumping, graph-based learning, and the cycle-cutset scheme, emphasizing the additive effect of each method. An evaluation of hybrid schemes was carried out by Prosser [Pro93b], who combined known look-ahead and look-back methods and ranked each combination based on average performance on, primarily, Zebra problems. Dechter and Meiri [DM94] have evaluated the effect of pre-processing consistency algorithms on backtracking

and backjumping. Before 1993, most of the empirical testing was done on relatively small problems (up to 25 variables), and the prevalent conclusion was that only low-overhead methods are cost effective.

With improvements in hardware and recognition that empirical evaluation may be the best way to compare the various schemes, has come a substantial increase in empirical testing. After Cheeseman, Kanefsky, and Taylor [CKT91] observed that randomly generated instances have a phase transition from easy to hard, researchers began to focus on testing various hybrids of algorithms on larger and harder instances [FD94b, FD94a, FD95, Gin93, CA93, BM96, Bak94]. In addition, closer examination of various algorithms uncovered interesting relationships. For instance, as already noted, dynamic variable ordering performs the function of value selection as well as variable selection [BvR95], and when the order of variables is fixed, forward-checking eliminates the need for backjumping in leaf nodes, as is done in Gaschnig's backjumping. [KvB94].

Recently, constraint processing techniques were augmented into the *Constraint Logic Programming (CLP)* languages. The inference engine of these languages uses a constraint solver as well as the traditional logic programming inference poroceures One of the most usefull constraint techniques is use of arc-consistency in look-ahead search [VH89, JL94].

Acknowledgements

Thanks to Peter van Beek for helpful comments, particularly his useful suggestions on the section covering historical and other perspectives; to Roberto Bayardo for commenting on a final version of this manuscripts, to Irina Rish for the figures, and lastly, to Michelle Bonnice for her dedicated editing of the final version.

References

[Arn85] S.A. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, 25:2–23, 1985.

[Bak94] A. B. Baker. The hazards of fancy backtracking. In *Proceedings of National Conference of Artificial Intelligence (AAAI-94)*, 1994.

[Bak95] A. B. Baker. Intelligent backtracking on constraint satisfaction problems: experimental and theoretical results. Technical report, Phd thesis, Graduate school of the university of Oragon, Oragon, 1995.

[BM96] R. Bayardo and D. Miranker. A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem.

In *AAAI-96: Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 298–304, 1996.

[BR75] J. R. Bitner and E. M. Reingold. Backtracking programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[Bru81] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[BvR95] F. Bacchus and P. van Run. Dynamic variable ordering in csps. In *Principles and Practice of Constraints Programming (CP-95)*, Cassis, France, 1995.

[CA93] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. In *AAAI-93: Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27, 1993.

[CKT91] P. Cheesman, B. Kanefsky, and W. Taylor. Where the *really* hard problems are. In *International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 331–337, 1991.

[Coo89] M. C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989.

[Dec90] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[Dec92] R. Dechter. Constraint networks. *Encyclopedia of Artificial Intelligence*, pages 276–285, 1992.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[DM94] R. Dechter and I. Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68:211–241, 1994.

[DP87] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[DP89] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, pages 353–366, 1989.

[Eve79] S. Even. Graph algorithms. In *Computer Science Press, Rockville, MD*, 1979.

[FD94a] D. Frost and R. Dechter. Dead-end driven learning. In *AAAI-94: Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 294–300, 1994.

[FD94b] D. Frost and R. Dechter. In search of best search: An empirical evaluation. In *AAAI-94: Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 301–306, Seattle, WA, 1994.

[FD95] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 572–578, August 1995.

[FD96] D. Frost and R. Dechter. Looking at full look-ahead. In *Proceedings of the Second International Conference on Constraint Programming (CP-96)*, 1996.

[FQ87] E. C. Freuder and M. J. Quinn. The use of lineal spanning trees to represent constraint satisfaction problems. Technical Report 87-41, University of New Hampshire, Durham, 1987.

[Fre78] E. C. Freuder. Synthesizing constraint expressions. *Communication of the ACM*, 21(11):958–965, 1978.

[Fre82] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[Fro97] D. H. Frost. Algorithms and heuristics for constraint satisfaction problems. Technical report, Phd thesis, Information and Computer Science, University of California, Irvine, Irvine, California, 1997.

[Gas78] 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, Ont., 1978.

[Gas79] J. Gaschnig. Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, 1979.

[GB65] S. Golomb and L. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 1965.

[Gin93] M.L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[GJ79] M. R Garey and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness. In *W. H. Freeman and Company, New York*, 1979.

[HE80] M. Haralick and G. L. Elliot. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[JL94] J. Jaffar and J. Lassez. Constraint logic programming: A survey. *Journal of Logic Programming*, 19(20):503–581, 1994.

[Kum92] V. Kumar. Algorithms for constraint satisfaction problems: a survey. *AI magazine*, 13(1):32–44, 1992.

[KvB94] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. In *Proceedings of International Joint Confererence of Artificial Intelligence (IJCAI-94)*, 1994.

[KvB97] G. Kondrak and P. van Beek. A theoretical valuation of selected algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[Lau78] J. L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1), 1978.

[Mac77] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Mac92] A. K. Mackworth. Constraint satisfaction. *In Encyclopedia of Artificial Intelligence*, pages 285–293, 1992.

[McA90] D. A. McAllester. Truth maintenance. In *AAAI-90: Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1109–1116, 1990.

[McG79] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inform. Sci.*, 19:229–250, 1979.

[MF85] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.

[MH86] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Mon74] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7(66):95–132, 1974.

[Nad89] B. A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–299, 1989.

[Nad90] B. A. Nadel. Some applications of the constraint satisfaction problem. In *AAAI-90: Workshop on Constraint Directed Reasoning Working Notes*, Boston, Mass., 1990.

[Nil78] N. Nillson. In *Introduction to Artificial Intelligence*. Tioga Publishing, 1978.

[Nil80] N. J. Nillson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, Ca, 1980.

[Nud83] B. Nudel. Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics. *Artificial Intelligence*, 21:135–178, 1983.

[Pea84] J. Pearl. Heuristics: Intelligent search strategies. In *Addison-Wesley*, 1984.

[Pro93a] P. Prosser. Forward checking with backmarking. Technical Report AISL–48–93, University of Strathclyde, 1993.

[Pro93b] P. Prosser. Hybrid algorithms for constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299, 1993.

[Pro95] Patrick Prosser. Mac-cbj: maintaining arc consistency with conflict-directed backjumping. Number 95/77, 1995.

[Pur83] P. W. Purdom. Search rearangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[RBM95] Jr. R. Bayardo and D. P. Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *Fourteenth International Joint Conference on Artificial Intelligence(Ijcai95)*, pages 558–562, 1995.

[SF94] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI-94*, pages 125–129, Amsterdam, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[SML90] A.B. Philips S. Minton, M.D. Johnston and P. Laired. Solving large scale constraint satisfaction and scheduling problems using heuristic repair methods. In *National Conference on Artificial Intelligence (AAAI-90)*, pages 17–24, Anaheim, CA, 1990.

[SS77] M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[SS86] S. Stone and J.M. Stone. Efficient search techniques-an empirical study of the n-queen problem. In *Technical report RC (#54343) IBM T.J. Watson*, 1986.

[Tsa93] E. Tsang. *Foundation of Constraint Satisfaction*. Academic press, 1993.

[VH89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[VHDT92] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Wal75] D Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, 1975.