Backtracking algorithms for constraint satisfaction problems*

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

September 17, 1999

Abstract

Over the past twenty five years many backtracking algorithms have been developed for constraint satisfaction problems. This survey describes the basic backtrack search within the search space framework and then presents a number of improvements developed in the past two decades, 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 is a generalization of propositional logic, in that variables may be assigned values from a set with any number of elements, instead of only TRUE and FALSE. Flexibility in the number of values can improve the ease and naturalness with which interesting problems are modeled.

The contribution of this paper is a survey of several approaches to solving constraint satisfaction problems, focusing on the backtracking algorithm and its variants, which form the basis for many constraint solution procedures. We provide on a careful exposition of each algorithm, its theoretical underpinnings, and its relationship to similar algorithms. Worst-case bounds on time and space usage are developed for each algorithm. In addition to the survey,

*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 paper makes several original contributions in formulation and analysis of the algorithms. In particular the look-back backjumping schemes are given a fresh exposition through comparison of the three primary variants, Gashnig's, graph-based and conflict-directed. We show that each of these backjumping algorithms is optimal relative to its information gathering process. The complexity of several schemes as a function of parameters of the constraint-graph are explicated. Those include backjumping complexity as a function of the depth of the DFS traversal of the constraint graph, learning algorithms as a function of the induced-width, and look-ahead methods such as partial-lookahead as a function of the size of the cycle-cutset of the constraint graph.

The remainder of the paper is organized as follows. Section 2 defines the constraint framework and provides an overview of the basic algorithms for solving constraint satisfaction problems. In Section 3 we present the backtracking algorithm. Sections 4 and 5 survey and analyze look-back methods such as backjumping and learning schemes while Section 6 surveys look-ahead methods. Finally, in Section 7 we present a brief historical review of the field. Previous surveys on constraint processing as well as on backtracking algorithms can be found in [Dec92, Mac92, Kum92, Tsa93, KvB97].

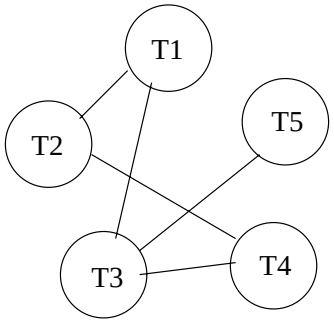
2 The constraint framework

2.1 Definitions

A *constraint network* or *constraint satisfaction problem* (CSP) 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 $R_S \subseteq D_{i_1} \times \cdots \times D_{i_r}$. S is called the scope of R_S. A *nogood* is a particular assignment of values to a subset of variables which is not permitted. In a *binary* constraint network 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 scope.

A variable is called *instantiated* when it is assigned a value from its domain; otherwise it is *uninstantiated*. By $x_i = a$ or by (x_i, a) we denote that variable x_i is instantiated with value a from its domain. A *partial instantiation* or *partial assignment* of a subset of X is a tuple of ordered pairs $((x_1, a_1), ..., (x_i, a_i))$, frequently abbreviated to $(a_1, ..., a_i)$ or $\vec{a_i}$ when the order of the variables is known.

Let Y and S be sets of variables, and let $\vec{y}$ be an instantiation of the variables in Y. We denote by $\vec{y_S}$ the tuple consisting of only the components of $\vec{y}$ that correspond to the variables in S. A partial instantiation $\vec{y}$ satisfies a constraint R_S iff $\vec{y_S} \in R_S$. *[Rina, the next sentence is new.]* $\vec{y}$ is *consistent* if $\vec{y}$ satisfies all constraints $R_T, T \subseteq Y$. A consistent partial instantiation is also called a *partial solution*. A *solution* is an instantiation of all the variables that is consistent.



Unary constraint
T4 $\neq$ 2:00
Binary constraints
T1, T2: {(1:00,2:00), (1:00,3:00), (2:00,1:00)
(2:00,3:00), (3:00,1:00), (3:00,2:00)}
T1, T3: {(2:00,1:00), (3:00,1:00), (3:00,2:00)}
T2, T4: {(1:00,2:00), (1:00,3:00), (2:00,1:00)
(2:00,3:00), (3:00,1:00), (3:00,2:00)}
T3, T4: {(1:00,2:00), (1:00,3:00), (2:00,3:00)}
T3, T5: {(2:00,1:00), (3:00,1:00), (3:00,2:00)}

Figure 1: The constraint graph and constraint relations of the scheduling problem in Example 1.

Example 1. The constraint framework is useful for expressing and solving scheduling problems. Consider the problem of scheduling 5 tasks T1, T2, T3, T4, T5, each of which takes 1 hour to complete. The tasks may start at 1:00, 2:00, or 3:00. Any number of tasks can be executed simultaneously, subject to the restrictions that T1 must start after T3, T3 must start before T4 and after T5, T2 cannot execute at the same time as T1 or T4, and T4 cannot start at 2:00.

With five tasks and three time slots, we can model the scheduling problem by creating five variables, one for each task, and giving each variable the domain {1:00, 2:00, 3:00}. Another equally valid approach is to create three variables, one for each starting time, and to give each of these variables a domain which is the powerset of {T1, T2, T3, T4, T5}. Adopting the first approach, the problem's constraint graph is shown in Figure 1. The constraint relations are shown on the right of the figure.

2.2 Constraint algorithms

Once a problem of interest has been formulated as a constraint satisfaction problem, it can be attacked with a general purpose constraint algorithm. Many CSP algorithms are based on the principles of search and deduction; more sophisticated algorithms often combine both principles. In this section we briefly survey the field of CSP algorithms.

2.2.1 Search - backtracking

The term *search* is used to represent a large category of algorithms which solve problems by guessing an operation to perform or an action to take, possibly with the aid of a heuristic [Nil80, Pea84]. A good guess results in a new state that is nearer to a goal. If the operation does not result in progress towards the goal (which may not be apparent until later in the search), then the operation can be retracted and another guess made.

For CSPs, search is exemplified by the backtracking algorithm. Backtracking search uses the operation of assigning a value to a variable, so that the current partial solution is extended. When no acceptable value can be found, the previous assignment is retracted, which is called a *backtrack*. In the worst case the backtracking algorithm requires exponential time in the number of variables, but only linear space. The algorithm was first described more than a century ago, and since then has been reintroduced several times [BR75].

2.2.2 Deduction - constraint propagation

To solve a problem by deduction is to apply reasoning to transform the problem into an equivalent but more explicit form. In the CSP framework the most frequently used type of deduction is known as constraint propagation or consistency enforcing [Mon74, Mac77, Fre82]. These procedures transform a given constraint network into an equivalent yet more explicit one by deducing new constraints, tightening existing constraints, and removing values from variable domains. In general, a consistency enforcing algorithm will make any partial solution of a subnetwork extendable to some surrounding network by guaranteeing a certain degree of local consistency, defined as follows.

A constraint network is 1-consistent if the values in the domain of each variable satisfy the network's unary constraints (that is, constraints which pertain to a single variable). A network is *k-consistent*, $k \geq 2$, iff given any consistent partial instantiation of any $k-1$ distinct variables, there exists a consistent instantiation of any kth additional variable [Fre78]. The terms *node-*, *arc-*, and *path-consistency* [Mac77] correspond to 1-, 2-, and 3-consistency, respectively. 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]. A problem that is k-consistent for all k is called *globally* consistent.

A variety of algorithms have been developed for enforcing local consistency [MF85, MH86, Coo90, VHDT92, DP87]. For example, arc-consistency algorithms delete certain values from the domains of certain variables, to ensure that each value in the domain of each variable is consistent with at least one value in the domain of each other variable. Path-consistency is achieved by introducting new constraints or nogoods which disallow certain pairs of values.

Constraint propagation can be used as a CSP solution procedure, although doing so is usually not practical. If global consistency can be enforced, then one or more solutions can easily be found in the transformed problem, without backtracking. However, enforcing k-consistency requires in general exponential time and exponential space in k [Coo90], and so in practice only local consistency, with $k \leq 3$, is used.

In Example 1, enforcing 1-consistency on the network will result in the value 2:00 being removed from the domain of T4, since that value is incompatable with a unary constraint. Enforcing 2-consistency will cause several other domain values to be removed. For instance, the constraint between T1 and T3 means that if T1 is scheduled for 1:00, there is no possible time for T3, since it must

occur before T1. Therefore, an arc-consistency algorithm will, among other actions, remove 1:00 from the domain of T1.

Algorithms that enforce local consistency can be performed as a preprocessing step in advance of a search algorithm. In most cases, backtracking will work more efficiently on representations that are as explicit as possible, that is, those having a high level of local consistency. The value of the tradeoff between the effort spent on pre-processing and the reduced effort spent on search has to be assessed experimentally, and is dependent on the character of the problem instance being solved [DM94]. Varying levels of consistency-enforcing can also be interleaved with the search process, and doing so is the primary way consistency enforcing techniques are incorporated into constraint programming languages [JL94].

2.2.3 Other constraint algorithms

In addition to backtracking search and constraint propagation, other approaches to solving constraint problems include *stochastic local search* and *structure-driven* algorithms. Stochastic methods typically move in a hill-climbing manner augmented with random steps in the space of complete instantiations [MJPL90]. In the CSP community interest in stochastic approaches was sparked by the success of the GSAT algorithm [SLM92]. Structure-driven algorithms, which employ both search and consistency-enforcing components, emerge from an attempt to characterize the topology of constraint problems that are tractable. *Tractable classes* of constraint networks are generally recognized by realizing that for some problems enforcing low-level consistency (in polynomial time) guarantees global consistency. The basic graph structure that supports tractability is a tree [MF85]. In particular, enforcing 2-consistency on a tree-structured binary CSP network ensures global consistency along some ordering of the variables.

3 Backtracking

A simple algorithm for solving constraint satisfaction problems is *backtracking*, which traverses the search graph in a depth-first manner. The order of the variables can be fixed in advance or determined at run time. The backtracking algorithm maintains 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; a phase in which the current partial solution is extended by assigning a consistent value, if one exists, to the next variable; 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.

Figure 2 describes a basic backtracking algorithm. As presented in Figure 2, the backtracking algorithm returns at most a single solution, but it can easily be modified to return all solutions, or a desired number. The algorithm employs a series of mutable value domains D'_i such that each $D'_i \subseteq D_i$. D'_i holds the subset

```
procedure BACKTRACKING
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or notification that the network is inconsistent.
    i ← 1                                (initialize variable counter)
    D'_i ← D_i                           (copy domain)
    while 1 ≤ i ≤ n
        instantiate x_i ← selectValue
        if x_i is null                   (no value was returned)
            i ← i − 1                    (backtrack)
        else
            i ← i + 1                    (step forward)
            D'_i ← D_i
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure selectValue
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        if (x_i, a) is consistent with ā_{i−1}
            return a
    end while
    return null                          (no consistent value)
end procedure
```

Figure 2: The backtracking algorithm.

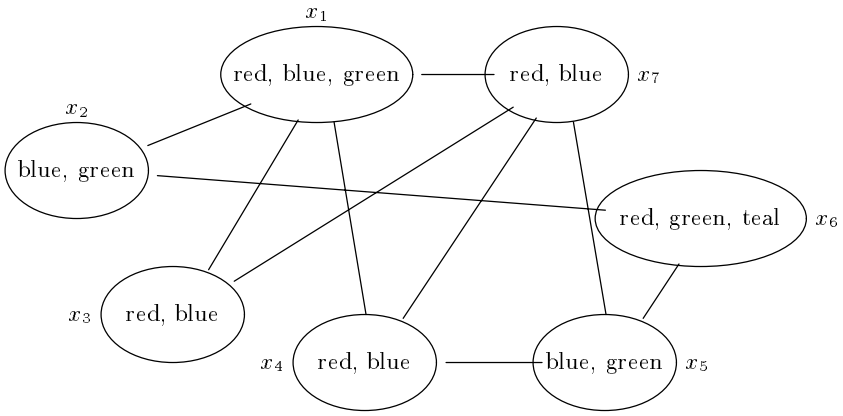


Figure 3: A modified coloring problem.

of D_i that has not yet been examined under the current partial instantiation. The D' sets are not needed if the values can be mapped to a contiguous set of integers that are always considered in ascending order; in this case a single integer can be used as a marker to divide values that have been considered from those that have not. We use the D' sets to describe backtracking for increased generality and to be consistent with the portrayal of more complex algorithms later in the paper.

The SELECTVALUE subprocedure is separated from the main BACKTRACKING routine for clarity. It has access to the current value of all variables in the main procedure. The complexity of SELECTVALUE depends on how the constraints are represented internally. When the CSP is binary, it may be practical to store the constraints in a table in contiguous computer memory; a one bit or one byte flag denotes whether each possible pair of variables with corresponding values is compatible. Accessing an entry in the table is an $O(1)$ operation. Since the compatibility of the current candidate assignment (x_i, a) must be checked with at most n earlier variable-value pairs, the complexity of SELECTVALUE for binary CSPs can be $O(n)$. With non-binary CSPs, checking compatibility is more expensive for two reasons. First, the number of possible constraints is exponential in the number of variables, and so the actual constraints are most likely stored in a list structure. Checking the list will be $O(\log c)$, where c is the number of constraints. Another possibility that bears consideration is that a constraint can be represented not as data but as a procedure. In this case the complexity of SELECTVALUE is of course dependent on the complexity of the procedures it invokes.

Example 2. Consider the coloring problem in Figure 3. 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 4(a) and 4(b), respectively. (Only legal states are depicted in the figure.)

3.1 Improvements to backtracking

Much of the work in constraint satisfaction during the last decade has been devoted to improving the performance of backtracking search. 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].

The performance 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. Commonly used preprocessing techniques are arc- and path-consistency algorithms, and heuristic approaches for determining the variable ordering [HE80, Fre82, DP89].

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 predetermined. Generally, it is advantageous to first instantiate variables that maximally constrain the rest of the search space. Therefore, the most highly constrained variable having the least number of values, 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:

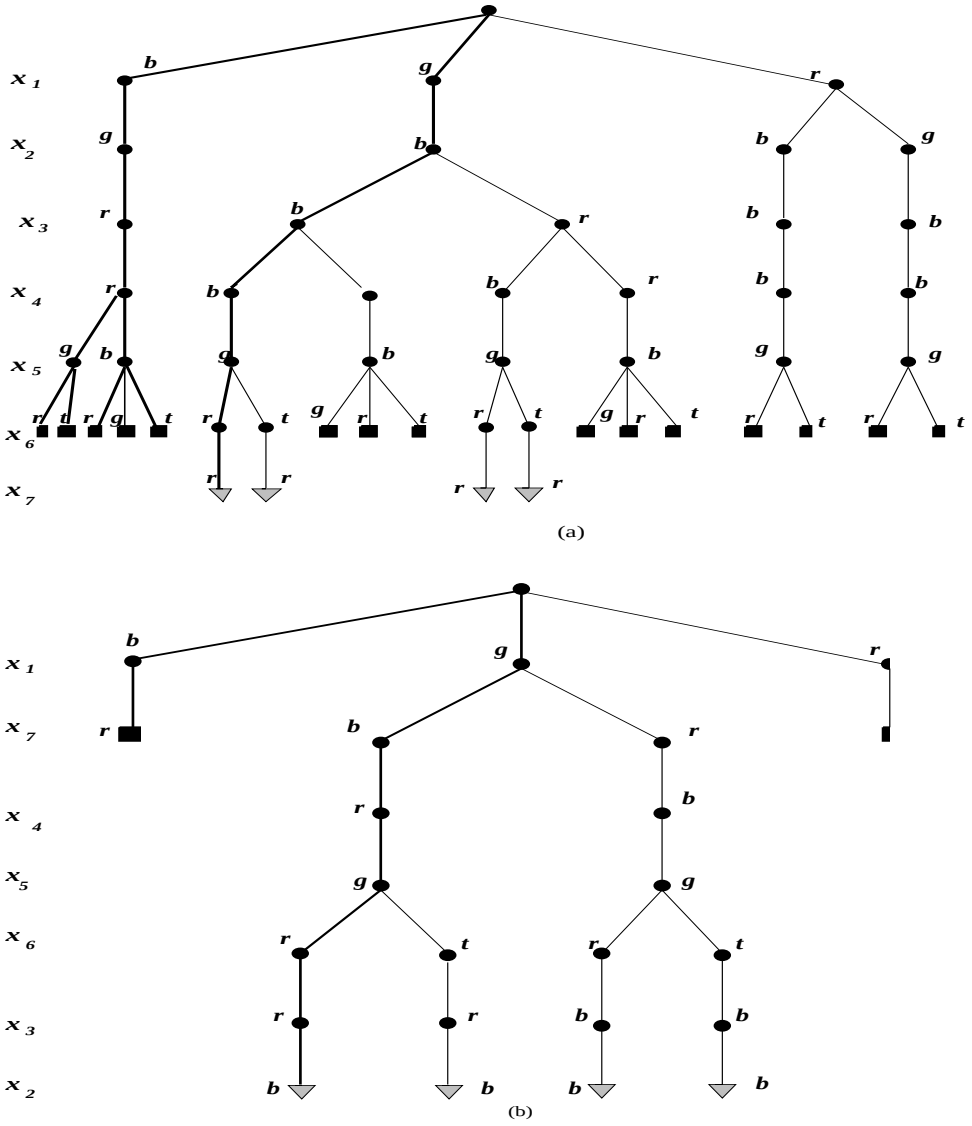


Figure 4: Backtracking search for the orderings (a) $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$ and (b) $d_2 = x_1, x_7, x_4, x_5, x_6, x_3, x_2$ on the example in Figure 3. Intermediate states are indicated by ovals, dead-ends by squares, and solutions by triangles. The colors blue, red, green, and teal are abbreviated by their first letters. Thick lines denote portions of the search space explored by backtracking when using left-to-right ordering and stopping after the first solution.

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*.

In sections 4 and 5 we will describe in detail several principle look-back schemes, while section 6 will focus on look-ahead methods.

4 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 3 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*. For ease of exposition, in the following discussion we assume a fixed ordering of the variables $d = (x_1, \ldots, x_n)$. This restriction can be lifted without affecting correctness, thus allowing dynamic variable orderings in all the algorithms.

4.1 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 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 $\vec{a}_{b-1} = (a_1, ..., a_{b-1})$ contains no conflict sets of the dead-end variable x_{i+1}. As it turns out, identifying this culprit variable is fairly easy.

Definition 1 (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 2 (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 3 (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 4 (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 5 *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 ordered 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, and x_b is the culprit variable, then $\vec{a}_b$, 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.

4.2 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 $\vec{a}$ in the forward phase, the algorithm maintains a pointer $high_i$ for each variable x_i. The pointer identifies the latest variable tested for consistency with x_i and found to be the earliest variable in conflict with a value in D'_i. For example, when no compatible values exist for x_i and if $high_i = 3$, the pointer indicates that $\vec{a_3}$ is a conflict set of x_i. If $\vec{a}_i$ does have a consistent value, then $high_i$ is assigned the value $i - 1$. 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}. Gaschnig's backjumping algorithm is presented in Figure 5.

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

Proof: Whenever there is a leaf dead-end $\vec{a}_{i-1}$, the algorithm has a partial instantiation $\vec{a}_{i-1} = (a_1, ..., a_{i-1})$. Let $j = high_i$. The algorithm jumps back to

```
procedure GASCHNIG'S-BACKJUMPING
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or a decision that the network is inconsistent.
    i ← 1                                   (initialize variable counter)
    D'_i ← D_i                              (copy domain)
    high_i ← 0                              (initialize pointer to culprit)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECTVALUE-GBJ
        if x_i is null                      (no value was returned)
            i ← high_i                      (backjump)
        else
            i ← i + 1
            D'_i ← D_i
            high_i ← 0
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure SELECTVALUE-GBJ
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        consistent ← true
        k ← 1
        while k < i and consistent
            if k > high_i
                high_i ← k
            if a⃗_k conflicts with (x_i, a)
                consistent ← false
            else
                k ← k + 1
        end while
        if consistent
            return a
    end while
    return null                             (no consistent value)
end procedure
```

Figure 5: Gaschnig's backjumping algorithm.

x_j, namely, to the tuple $\vec{a_j}$. Clearly, $\vec{a_j}$ is in conflict with x_i, so we only have to show that $\vec{a_j}$ is minimal. Since $j = high_i$ when the domain of x_i 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_i, and therefore x_j is the culprit variable. ¿From Proposition 1, it follows that this algorithm is safe and optimal. □

Example 3. For the problem in Figure 3, 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 $x_3 = red$ in Figure 4a) 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.

4.3 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 3 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 latest variable in the ordering that precedes x_{i+i} and is connected to x_{i+1} is the culprit variable.

The importance of graph-based backjumping is that studying algorithms with performance tied to the constraint graph leads to graph-theoretic complexity bounds and thus to graph-based heuristics aimed at reducing these bounds.

```
procedure GRAPH-BASED-BACKJUMPING
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or a decision that the network is inconsistent.
    compute anc(x_i) for each x_i        (see Definition 6 in text)
    i ← 1                                (initialize variable counter)
    D'_i ← D_i                           (copy domain)
    I_i ← anc(x_i)                       (copy of anc() that can change)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECTVALUE
        if x_i is null                   (no value was returned)
            i-prev ← i
            i ← highest in I_i           (backjump)
            I_i ← I_i ∪ I_{i-prev} − {x_i}
        else
            i ← i + 1
            D'_i ← D_i
            I_i ← anc(x_i)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure SELECTVALUE                    (same as BACKTRACKING's)
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        if (x_i, a) is consistent with ā_{i-1}
            return a
    end while
    return null                          (no consistent value)
end procedure
```

Figure 6: The graph-based backjumping algorithm.

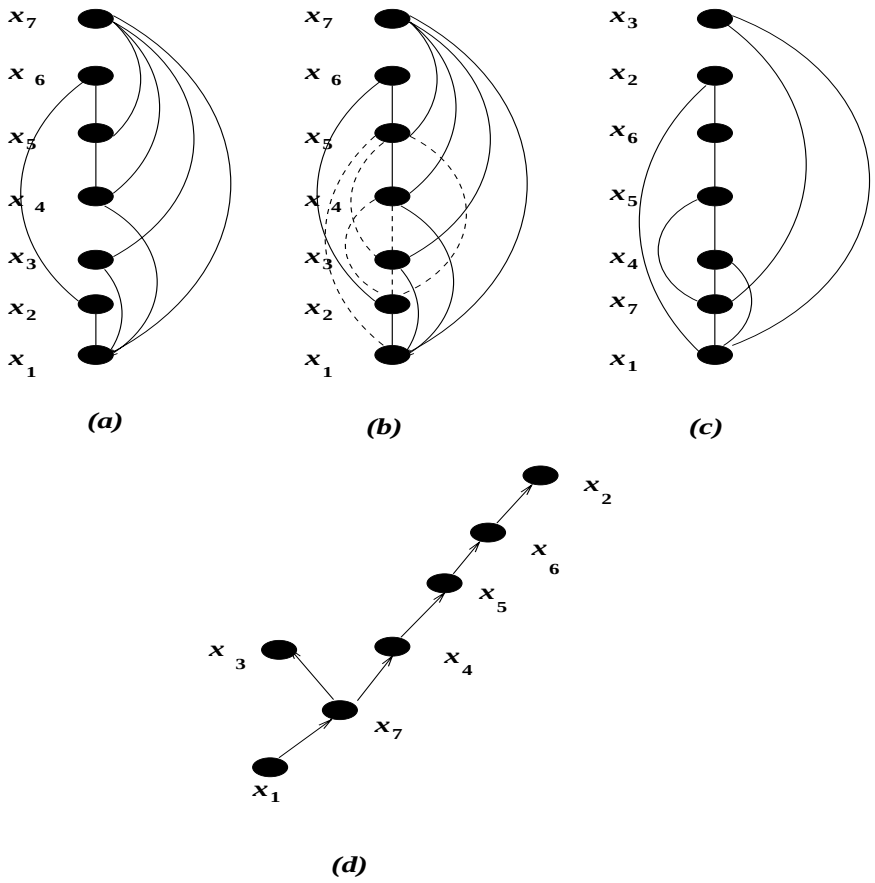


Figure 7: Two ordered constraint graphs on the problem in Figure 3: (a) $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$, (b) the induced graph along d_1, (c) $d_2 = x_1, x_7, x_4, x_5, x_6, x_2, x_3$ (d) A DFS spanning tree along ordering d_2.

Such bounds are also 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 6 (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 4. 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 5. Consider again the constraint network in Figure 3 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. As we will show, the culprit variable is determined by the *induced ancestor set* in the current *session*.

Definition 7 (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 8 (induced ancestors, induced parent) *Given a variable x_j and a subset of variables Y that all succeed x_j in the ordering, 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 $I_j(Y)$.*

Theorem 1 *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 6. 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 $Y = \{\}$, 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 $Y = \{x_5, x_6\}$, 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 $Y = \{x_5, x_7\}$, the corresponding induced parent set $I_4(\{x_5, x_7\}) = \{x_1, x_3\}$, 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 retracts to x_1 since $I_3(\{x_4, x_5, x_7\}) = \{x_1\}$. 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 6 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 2 *Graph-based backjumping implements jumps to optimal culprit variables at both leaf and internal dead-ends, when only graph information is used.*

Proof: Step 3 of the algorithm maintains the set I_i, which is the induced ancestor set of x_i relative to the dead-end variables in its session. Thus, the claim follows from Theorem 1. □

4.3.1 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 7. Consider, once again, the CSP in Figure 3. 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,

Theorem 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 on 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 number of nodes in the 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.

It can be shown that graph-based backjumping can be bounded for a larger class of variable orderings, not only *DFS* ones. To do so a few more graph-based concepts have to be introduced.

Definition 9 (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 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. For more information, see [DP87].*

Example 8. 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.

It can be shown that DFS orderings of induced graphs also allow bounding backjumping's complexity as a function of the depth of a corresponding DFS tree. Let d be a DFS ordering of the induced graph G^* of G, created along an arbitrary ordering, and let m^*_d be the depth of this DFS ordering of G^*.

Theorem 5 *Let* m^*_d *be the depth of a* DFS *tree traversal of some induced graph* G^*. *The complexity of graph-based backjumping using ordering* d *of a constraint problem having a constraint graph* G, *is* $O(exp(m^*_d))$.

A proof, that uses somewhat different terminology and derivation, is given in [BM96].

The virtue of the above Theorem is in allowing a larger set of orderings, each yielding a bound on backjumping's performance as a function of its DFS tree-depth, to be considered. Since it can be shown that every DFS traversal of G is also a DFS traveral of its induced graph along d, G^*_d, DFS orderings are included (in the set of DFS orderings of all induced graph). Thus, this may lead to better orderings having better bounds for backjumping.

4.4 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 next 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 10 (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 11 (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 $\vec{a_j}$, $j \geq i$, 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 $par(\vec{a_i})$ 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 3 *Given a dead-end tuple $\vec{a_i}$, the latest variable in its jumpback set J_i is the culprit, namely, the earliest variable that is safe to jump back to.* □

Algorithm conflict-directed backjumping is presented in Figure 8. It computes the jumpback sets for each variable. In summary,

Proposition 4 *Algorithm conflict-directed backjumping jumps back to the latest variable in the dead-end's jumpback set, and therefore it is optimal.* □.

Example 9. Consider the problem of Figure 3 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.

```
procedure CONFLICT-DIRECTED-BACKJUMPING
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or a decision that the network is inconsistent.
    i ← 1                                (initialize variable counter)
    D'_i ← D_i                           (copy domain)
    J_i ← ∅                              (initialize conflict set)
    while 1 ≤ i ≤ n
        instantiate x_i ← selectVALUE
        if x_i is null                   (no value was returned)
            i-prev ← i
            i ← highest index in J_i     (backjump)
            J_i ← J_i ∪ J_{i-prev} − {x_i}
        else
            i ← i + 1                    (step forward)
            D'_i ← D_i
            J_i ← ∅
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure selectVALUE-CBJ
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        consistent ← true
        k ← 1
        while k < i and consistent
            if a⃗_k conflicts with (x_i, a)
                add x_k to J_i
                consistent ← false
            else
                k ← k + 1
        end while
        if consistent
            return a
    end while
    return null                          (no consistent value)
end procedure
```

Figure 8: The conflict-directed backjumping algorithm.

4.5 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 6, 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 an 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.

5 Learning Algorithms

The earliest minimal conflict set of Definition 10 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. The learning procedure records 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.[1] Yet, when $\vec{a}_i$ contains one or more subsets that are in conflict with x_{i+1}, 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.[2]

With the goal of speeding 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.[3]

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. Constraints involving many variables are less frequently applicable, require more space to store, and are more expensive to consult than constraints having fewer variables. 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.

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

5.1 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 con-

[1] Recording this constraint may be useful if the same initial set of constraints is expected to be queried in the future.

[2] 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.

[3] 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.

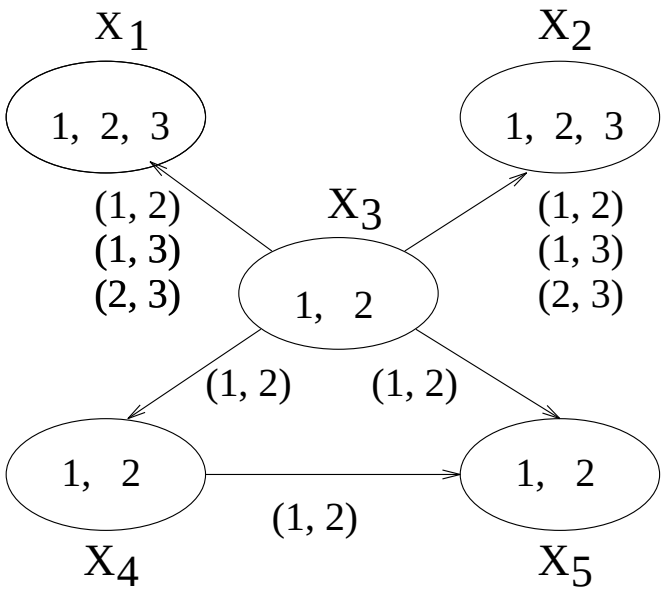


Figure 9: A small CSP. The constraints are: $x_3 < x_1, x_3 < x_2,\ x_3 < x_5,\ x_3 < x_4,\ x_4 < x_5$. The allowed pairs are shown on each arc.

straint graph alone. Given an i-leaf dead-end $(a_1, ..., a_i)$, those subsets of values associated with the ancestors of x_{i+1} are identified and included in the conflict set.

Example 10. Suppose we try to solve the problem in Figure 9 along the ordering $d = x_1, x_2, x_3, x_4, x_5$. 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 (while x_1 and x_2 are not) 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 $n-1$ earlier variables. To augment graph-based backjumping with graph-based learning, we add a line to the based algorithm specifying that a new constraint, or a new tuple in an existing constraint, should be recorded after each dead-end; see Figure 10).

5.2 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 Figure 9, 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 informative,

procedure GRAPH-BASED-BACKJUMPING-LEARNING
Input: A constraint network with variables $\{x_1, \ldots, x_n\}$ and domains $\{D_1, \ldots, D_n\}$.
Output: Either a solution, or a decision that the network is inconsistent.

```
    compute anc(x_i) for each x_i        (see Definition 6 in text)
    i ← 1                                (initialize variable counter)
    D'_i ← D_i                           (make a copy of domain)
    I_i ← anc(x_i)                       (copy of anc() that can change)
    while 1 ≤ i ≤ n
        instantiate x_i ← selectValue
        if x_i is null                   (no value was returned)
            record (a⃗_i)_{I_i} as a nogood
            i-prev ← i
            i ← highest in I_i           (backjump)
            I_i ← I_i ∪ I_{i-prev} − {x_i}
        else
            i ← i + 1
            D'_i ← D_i
            I_i ← anc(x_i)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure
```

Figure 10: The graph-based-backjumping-learning algorithm. The only change is the addition of the line beginning with "record". $(\vec{a_i})_{I_i}$ denotes the projection of $\vec{a}_i$ over the subset of variables in I_i. The SELECTVALUE subprocedure is the same as BACKTRACKING's and GRAPH-BASED-BACKJUMPING's.

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.

5.3 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 11. For the problem in Example 10 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 augments conflict-directed backjumping with learning, is presented in Figure 12. The SELECTVALUE-CBJ subprocedure is not changed and is therefore not repeated in Figure 12, but with learning it must consult all constraints, both original and learned.

5.4 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 11 we present the search space of the problem in Figure 9 explicated by naive backtracking and by backtracking augmented with graph-based second-order learning; all the branches below the cut lines in Figure 11 will be generated by the former but not by the latter.

5.5 Relevance bounded learning

An alternative to bounding the size of learned nogoods is to bound the learning process by discarding nogoods that appear to be no longer relevant, by some measure.

Definition 12 (i-relevant) *[BM96] A nogood is i-relevant if it differs from the current partial assignment by at most i variable-value pairs.*

Definition 13 (i'th order relevance bounded learning) *[BM96] An i'th order relevance bounded learning scheme maintains only those learned nogoods that are i-relevant.*

Ginsberg's *dynamic backtracking* algorithm [Gin93] employs a similar notion of keeping only learned nogoods that are most likely to be consulted in the near future search.

5.6 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

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

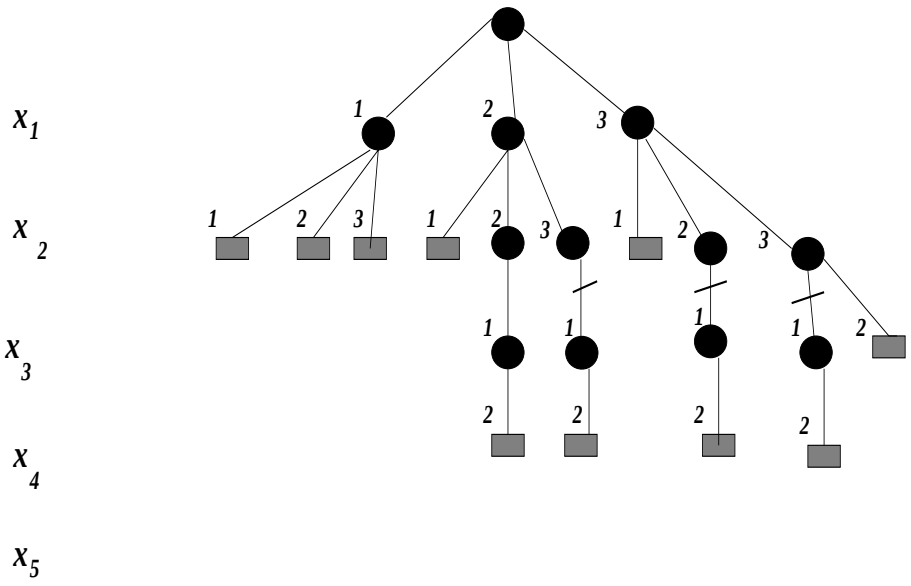


Figure 11: Search space explicated by backtracking, without and with second-order learning.

```
procedure CONFLICT-DIRECTED-BACKJUMPING-LEARNING
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or a decision that the network is inconsistent.
    i ← 1                                  (initialize variable counter)
    D'_i ← D_i                             (copy domain)
    J_i ← ∅                                (initialize conflict set)
    while 1 ≤ i ≤ n
        instantiate x_i ← selectValue
        if x_i is null                     (no value was returned)
            record J_i and corresponding values as a nogood
            i-prev ← i
            i ← highest index in J_i       (backjump)
            J_i ← J_i ∪ J_{i-prev} − {x_i}
        else
            i ← i + 1                      (step forward)
            D'_i ← D_i
            J_i ← ∅
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure
```

Figure 12: The conflict-directed-backjumping-learning algorithm. The only change is the addition of the line beginning with "record".

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 a 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 $m_d^* \leq \log n \cdot w^*(d)$. 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.

5.7 Backmarking

Backmarking [Gas79] is a cacheing-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_i and that x_p is the earliest variable in the ordering which changed its value since the last visit to x_i. Clearly, any testing of values of x_i against variables preceding x_p 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_i can either be recognized immediately as inconsistent or else be tested only against prior instantiations starting from x_p on. Backmarking implements this idea by maintaining two new 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 before those involving later variables.) If $x_i = a_v$ is consistent with all earlier partial instantiations $\vec{a}_j$, $j < i$, then $M_{i,v} = i$. For instance, $M_{10,2} = 4$ 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 with $X_{10} = a_2$. The second table, with elements low_i, records the earliest variable that has 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 13.

```
procedure BACKMARKING
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or notification that the network is inconsistent.
    M_{i,v} ← 0, low_i ← 0 for all i and v      (initialize tables)
    i ← 1                                       (initialize variable counter)
    D'_i ← D_i                                  (copy domain)
    while 1 ≤ i ≤ n
        instantiate x_i ← selectVALUE-BACKMARKING
        if x_i is null                          (no value was returned)
            for all j, i < j ≤ n,               (update low of future variables)
                if i < low_j
                    low_j ← i
            i ← i − 1                           (backtrack)
        else
            i ← i + 1                           (step forward)
            D'_i ← D_i
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure selectVALUE-BACKMARKING
    remove from D'_i all a_v for which M_{i,v} < low_i
    while D'_i is not empty
        select an arbitrary element a_v ∈ D'_i, and remove a_v from D'_i
        consistent ← true
        k ← low_i
        while k < i and consistent
        if a_k conflicts with (x_i, a_v)
            M_{i,v} ← k
            consistent ← false
        else
            k ← k + 1
        end while
        if consistent
            M_{i,v} ← i
            return a_v
    end while
    return null                                 (no consistent value)
end procedure
```

Figure 13: The backmarking algorithm.

Example 12. Consider again the example in Figure 3, and assume backmarking uses ordering d_1. Once the first dead-end is encountered at variable x_7 (see the search space in Figure 4), 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, 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.

6 Look-ahead Strategies

6.1 Combining backtracking and constraint propagation

CSP search algorithms can combine backtracking and local constraint propagation, by applying a consistency enforcing procedure to the uninstantiated variables. This combination is known as "looking ahead." Extending a partial instantiation may induce constraints on the remaining variables, and making these constraints explicit may reduce the amount of backtracking search required. Of course, actions conditioned on a partial instantiation will have to be undone if the partial instantiation becomes no longer current due to backtracking.

Example 13. Consider the coloring problem in Figure 3, and assume that variable x_1 is first in the ordering and is assigned the value *red*. A look-ahead procedure notes that the value *red* in the domains of x_3, x_4, and x_7 is incompatible with the partial instantiation, and provisionally removes those values. A more extensive look-ahead procedure may then note that x_3 and x_7 are connected and are now left with incompatible values; each variable has the domain $\{blue\}$ and the problem, with $x_1 = red$, is therefore not arc-consistent. The

procedure REVISE(m, n)
Input: A constraint network with variables $\{x_1, \ldots, x_n\}$ and domains $\{D_1, \ldots, D_n\}$; a partial instatiation $\vec{a_i}$; and two variable indices m and n.
Output: Modified D'_m.
    for each value a in D'_m
        if there is no value $b \in D'_n$ such that $(\vec{a_i}, x_m = a, x_n = b)$ is consistent
            remove a from D'_m
end procedure

Figure 14: The revise procedure.

implication is that assigning *red* to x_1 will inevitably lead to a dead-end, and thus this assignment should be rejected.

While look-ahead strategies incur an extra cost after each instantiation, they can provide several benefits. First, by removing from each future variable's domain all values that are not consistent with the partial instantiation, they eliminate the need to test values of the current variable for consistency with previous variables. A corollary benefit is that if *all* values of an uninstantiated variable are removed by the look-ahead procedure, then the current instantiation cannot be part of a solution and the algorithm can backtrack. Consequently, dead-ends occur earlier in the search, and users of CSP algorithms often note much smaller search spaces when look-ahead is employed. In general, the stronger the level of constraint propagation, the smaller the search space explored and the higher the computational overhead. Another benefit of look-ahead is that the sizes of the uninstantiated variable domains can be used to guide the selection of the variable and value to choose next; we return to this topic later in the section.

6.2 Look-ahead algorithms

Look-ahead strategies that employ local consistency procedures have the same exponential worst-case time bounds as backtracking. If the look-ahead procedure is based on arc-consistency or a weaker form of consistency, then the space requirements are no more than maintaining the D' sets. The key issue is determining experimentally a cost-effective balance between look-ahead's accuracy and its overhead. We define four levels of look-ahead. Each employs the REVISE [Mac77] subprocedure for enforcing the arc-consistency of the constraint between two variables. Our version of REVISE, in Figure 14, differs from the standard presentation in that in includes the current partial instantiation $\vec{a_i}$ in the test for consistency. This is important when the procedure is used during search.

- *Forward checking* [HE80]. This approach, which is described in Figure 15, does the most limited form of constraint propagation. Forward checking propagates separately the effect of a tentative value selection to each of

```
procedure FORWARD-CHECKING
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or notification that the network is inconsistent.
    D'_i ← D_i for 1 ≤ i ≤ n              (copy all domains)
    i ← 1                                (initialize variable counter)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECTVALUE-FORWARD-CHECKING
        if x_i is null                   (no value was returned)
            reset each D' set to its value before x_i was last instantiated
            i ← i − 1                    (backtrack)
        else
            i ← i + 1                    (step forward)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure SELECTVALUE-FORWARD-CHECKING
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        empty-domain ← false
        for all k, i < k ≤ n
            REVISE(k, i)
            if D'_k is empty             (x_i = a leads to a dead-end)
                empty-domain ← true
        if empty-domain                  (don't select a)
            reset each D'_k, i < k ≤ n to value before a was selected
        else
            return a
    end while
    return null                          (no consistent value)
end procedure
```

Figure 15: The forward checking algorithm.

the future variables. If, as a result of calling REVISE the domain of one of these future variables becomes empty, the value is not selected and the next candidate value is tried.

- *Arc-consistency look ahead.* This category includes backtracking based algorithms which enforce arc-consistency on the uninstantiated variables after each assignment of a value to the current variable. If the uninstantiated variables are not arc-consistent, namely, during the process a variable's domain becomes empty, then the value is rejected. Several arc-consistency enforcing algorithms have been developed; we do not specify which version should be used in Figure 16.

- *Full looking ahead* [HE80]. The additional processing, beyond forward checking, done by full looking ahead is a limited form of arc-consistency, in which each value in the domains of future variables is only once processed and considered for removal. To illustrate how full looking ahead works, suppose there are three future variables, x_{25} with current domain $\{a, b\}$, x_{26} with current domain $\{a, b\}$ and x_{27} with current domain $\{b\}$. Also suppose there is an inequality constraint (as in graph coloring) between x_{25} and x_{26} and between x_{26} and x_{27}. Full looking ahead will process x_{25} and reject neither of its values, since they both have a compatible value in the domain of x_{26}. When full looking ahead processes x_{26}, it removes the value b because there is no allowable match for b in the domain of x_{27}. Arc-consistency would later go back and remove a from x_{25}'s domain, because it no longer has a consistent match in x_{26}'s domain, but full looking ahead does not do this.

- *Directional arc-consistency look-ahead; Partial looking ahead* [HE80]. These approaches perform forward-checking style look-ahead and then do additional arc-consistency checking based on performing REVISE on each future variable and those that follow it in the ordering. Partial looking ahead is outlined in Figure 17. Directional arc-consistency look-ahead, based on [DP87], is identical except for the order in which REVISE is performed between pairs of future variables.

Our list of look-ahead techniques is not meant to be exhaustive. In particular, a search algorithm could enforce a higher degree of consistency than arc-consistency after each instantiation. Although applying arc-consistency was highly successful on a class of vision instances [Wal75], the more extensive varieties of look-ahead have received less attention. This may be due, in part, to the negative conclusions about full looking ahead reached in [HE80]: "The checks of future with future units do not discover inconsistencies often enough to justify the large number of tests required." More recent work has shown that as larger and more difficult problems are experimented with, higher levels of look-ahead become more useful. The balance between overhead and pruning is studied in [FD95, SF94, Bak95, Fro97].

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

```
procedure ARC-CONSISTENCY-LOOKING-AHEAD
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or notification that the network is inconsistent.
    D'_i ← D_i for 1 ≤ i ≤ n              (copy all domains)
    i ← 1                                (initialize variable counter)
    while 1 ≤ i ≤ n
        instantiate x_i ← selectVALUE-ARC-CONSISTENCY
        if x_i is null                   (no value was returned)
            reset each D' set to its value before x_i was last instantiated
            i ← i − 1                    (backtrack)
        else
            i ← i + 1                    (step forward)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure selectVALUE-ARC-CONSISTENCY
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        apply ARC-CONSISTENCY to all uninstantiated variables
        if any future domain is empty   (don't select a)
            reset each D'_k, i < k ≤ n to value before a was selected
        else
            return a
    end while
    return null                          (no consistent value)
end procedure
```

Figure 16: The forward checking algorithm with arc-consistency enforcing after each instantiation. The selectVALUE-FORWARD-CHECKING subprocedure is the same as specified in Figure 15. The ARC-CONSISTENCY subprocedure is not specified; it can remove values from D' sets, and if a D' set becomes empty it informs the main routine that the future variables are not arc-consistent.

```
procedure PARTIAL-LOOK-AHEAD
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or notification that the network is inconsistent.
    D'_i ← D_i for 1 ≤ i ≤ n              (copy all domains)
    i ← 1                                (initialize variable counter)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECTVALUE-PLA
        if x_i is null                   (no value was returned)
            reset each D' set to its value before x_i was last instantiated
            i ← i − 1                    (backtrack)
        else
            i ← i + 1                    (step forward)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure SELECTVALUE-PLA
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        empty-domain ← false
        for all k, i < k ≤ n
            REVISE(k, i)
            if D'_k is empty             (x_i = a leads to a dead-end)
                empty-domain ← true
        for all j, i < j ≤ n             (further look-ahead: compare)
            for all k, j < k ≤ n         (each future variable with later ones)
                REVISE(j, k)
                if D'_j is empty         (x_i = a leads to a dead-end)
                    empty-domain ← true
        if empty-domain                  (don't select a)
            reset each D'_k, i < k ≤ n to value before a was selected
        else
            return a
    end while
    return null                          (no consistent value)
end procedure
```

Figure 17: The partial looking ahead algorithm.

Proposition 5 *[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 6 *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]. □

6.3 Look-ahead for variable and value 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.

When employing variable orderings that are decided dynamically during search, the objective is to select as the next variable the one that most constrains 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 the various levels of look-ahead propagation discussed above. Such methods are called *dynamic variable ordering* (DVO) strategies.

Using forward checking, which does the least amount of look-ahead, has proven cost effective in many empirical studies.[4] We call this weak form of DVO

[4]As far as we know, no empirical testing has been carried out for full looking ahead or partial looking ahead based variable orderings.

```
procedure DVFC
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or notification that the network is inconsistent.
    D'_i ← D_i for 1 ≤ i ≤ n              (copy all domains)
    i ← 1                                (initialize variable counter)
    while 1 ≤ i ≤ n
        instantiate x_i ← selectValue-forward-checking
        if x_i is null                   (no value was returned)
            reset each D' set to its value before x_i was last instantiated
            i ← i − 1                    (backtrack)
        else
            if i < n
                s = min_{i<j≤n} |D'_j|   (find future var with smallest domain)
                rearrange variables so that x_s follows x_i
            i ← i + 1                    (step forward to x_s)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure
```

Figure 18: The DVFC algorithm. It uses the selectValue-forward-checking subprocedure given in Fig. 15.

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 18.

Example 14. Consider again the example in Figure 3. 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.

The information gleaned during the look-ahead phase can also be used to guide value selection [DP87, SF90, Gee92, FD95]. Of course, all look-ahead

algorithms perform a coarse version of value selection when they reject values that are shown to lead to a future dead-end, but a more refined approach that ranks the values of the current variable has been shown to be useful.

The look-ahead value ordering (LVO) algorithm [FD95] is based on forward checking. Instead of accepting the first value for the current variable that is not shown to lead to a dead-end, LVO tentatively instantiates each value of the current variable, and examines the effects of a forward checking style look-ahead on the domains of future variables. (Each instantiation and its effects are retracted before the next instantiation is made.) LVO uses a heuristic to transform this information into a ranking of the values. Experimental results indicate that the cost of performing the additional look-ahead is not justified on smaller and easier problems, but can be extremely useful on particularly hard problems.

6.4 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 $O(e_d \cdot k)$.

7 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 variable 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 [BM95].

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 procedures One of the most usefull constraint techniques is the 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 manuscript, to Irina Rish for

the figures, and lastly, to Michelle Bonnice for her dedicated editing of an earlier version of the paper.

References

[Arn85] S. 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.* PhD thesis, Graduate School of the University of Oregon, Eugene, OR, 1995.

[BM95] R. Bayardo, Jr. and D. P. Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 558–562, 1995.

[BM96] R. Bayardo and D. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI-96: Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 298–304, Portland, OR, 1996.

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

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

[BvR95] F. Bacchus and P. van Run. Dynamic variable ordering in CSPs. In *Principles and Practice of Constraint Programming (CP-95)*, Cassis, France, 1995. Available as Lecture Notes on CS, vol 976, pp 258–277, 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. Cheeseman, B. Kanefsky, and W. Taylor. Where the *really* hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 331–337, 1991.

[Coo90] M. C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41(1):89–95, 1990.

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

[Dec92] R. Dechter. Constraint networks. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence, 2nd Edition*, pages 276–285. John Wiley & Sons, 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, Dec. 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*, 1979.

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

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

[FD95] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 572–578, 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 linear spanning trees to represent constraint satisfaction problems. Technical Report 87-41, University of New Hampshire, Durham, 1987.

[Fre78] E. C. Freuder. Synthesizing constraint expressions. *Communications 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*. PhD thesis, Information and Computer Science, University of California, Irvine, CA, 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 (CSCSI-78)*, 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.

[Gee92] P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, pages 31–35, Vienna, 1992.

[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*. W. H. Freeman, 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 Conference of Artificial Intelligence (IJCAI-94)*, 1994.

[KvB97] G. Kondrak and P. van Beek. A theoretical evaluation 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 S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence, 2nd Edition*, pages 285–293. John Wiley & Sons, 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. *Information Science*, 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*, 25:65–74, 1985.

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

[MJPL90] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 17–24, Boston, Mass., 1990.

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

[Nad89] B. A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 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.

[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*. 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.

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

[SF90] Norman Sadeh and Mark S. Fox. Variable and value ordering heuristics for activity-based job-shop scheduling. In *Proceedings of the Fourth International Conference on Expert Systems in Production and Management*, pages 134–144, 1990.

[SF94] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the European Conference on AI (ECAI-94)*, pages 125–129, Amsterdam, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, 1992.

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

[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 P. H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.