Backjump-based Backtracking for Constraint Satisfaction Problems ⋆

Rina Dechter and Daniel Frost

Department of Information and Computer Science,
University of California, Irvine,
Irvine, California, USA 92697-3425

Abstract

The performance of backtracking algorithms for solving finite-domain constraint satisfaction problems can be improved substantially by look-back and look-ahead methods. Look-back techniques extract information by analyzing failing search paths that are terminated by dead-ends. Look-ahead techniques use constraint propagation algorithms to avoid such dead-ends altogether. This survey describes a number of look-back variants including backjumping and constraint recording which recognize and avoid some unnecessary explorations of the search space. The last portion of the paper gives an overview of look-ahead methods such as forward checking and dynamic variable ordering, and discusses their combination with backjumping.

Key words: constraint satisfaction, backtracking, backjumping
PACS:

Email address: `dechter@uci.edu, frost@uci.edu` (Rina Dechter and Daniel Frost).

⋆ This work was supported in part by NSF grant IIS-0086529 and by MURI ONR award N00014-00-1-0617.

1 Introduction

The constraint paradigm is a useful and well-studied framework for expressing many problems of interest in Artificial Intelligence. 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.

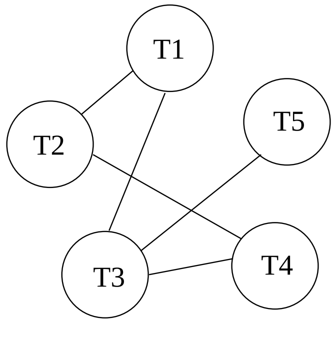
This paper presents a survey of backtacking search for solving constraint satisfaction problems, with an emphasis on look-back enhancements. We provide a detailed exposition of each algorithm, its theoretical underpinnings, and its relationships with similar algorithms. Worst-case bounds on time and space usage are developed for each algorithm. The look-back backjumping schemes are given a fresh exposition through comparison of the three primary variants: Gaschnig's backjumping, graph-based backjumping, and conflict-directed backjumping. The complexity of several algorithms as a function of parameters of the constraint graph are explicated. These include the complexity of backjumping as a function of the depth of the DFS traversal of the constraint graph, the complexity of learning algorithms as a function of the induced width, and the complexity of 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. The exposition is applicable to the general non-binary CSP definition. 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. Section 7 describes how look-back and look-ahead approaches can be integrated, and provides a comparison of selected algorithms and heuristics the paper covers. Finally, in Section 8 we present a brief historical review of the field. Previous surveys on constraint processing as well as on backtracking algorithms can be found in [18,49,46,77,45]; more recent relevant overviews are [13,56,55].

2 The constraint framework

2.1 Definitions

A *constraint satisfaction problem* (CSP) or *constraint network* $P = (X, D, C)$ consists of a set of n variables $X = \{x_1, \ldots, x_n\}$, a set of n finite value domains



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

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

$D = \{D_1, \ldots, D_n\}$, and a set of c constraints or relations $C = \{R_1, \ldots, R_c\}$. Each value domain is a finite set of values, one of which must be assigned to the corresponding variable. A constraint is a relation, defined on a unique subset of the variables, called the constraint's *scope*. Each tuple in the relation denotes a legal combinations of values to the variables. Thus a constraint R_S with scope $S \subset X$ is a subset of the Cartesian product of the domains of the variables in S. Constraints can also be described by mathematical expressions or by computable procedures that indicate valid and invalid assignments. A constraint's *arity* is the number of variables in its scope: a *unary* constraint applies to a single variable; a *binary* constraint has arity two. In a binary CSP all constraints are unary or binary. A *constraint graph* associates a vertex with each variable and has an edge between any two vertices whose associated variables appear in the same constraint's scope.

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 power set of {T1, T2, T3, T4, T5}. Adopting the first approach, the problem's constraint graph is shown in Figure 1. In this case the problem has only unary and binary constraints. The constraint relations are shown on the right of the figure. For example, the tuple (1:00, 2:00) of constraint $R_{\{T1,T2\}}$ indicates that starting task T1 at 1:00 and task T2 at 2:00 is permitted by the definition of the relation between these tasks.

A variable is called *instantiated* when it has been 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 has been instantiated with value a from its domain. When describing an algorithm, $x_i \leftarrow a$ indicates the act of assigning value a to variable x_i. A *partial instantiation* or *partial assignment* to a subset $\{x_1, \ldots, x_i\} \subseteq X$ is a tuple of ordered pairs $((x_1, a_1), ..., (x_i, a_i))$, frequently abbreviated to $(a_1, ..., a_i)$. We also use $\vec{a}_i$ to denote a consecutive set of instantiated variables from x_1 to x_i, and $\vec{a}$ to denote an assignment to an arbitrary subset of variables. The notations can be mixed, e.g., $(\vec{a}_i, x_j = b, x_k = c)$, where $j, k > i$.

Let Y be a set of variables, and let $\vec{a}$ be an instantiation of the variables in Y. Let R_S be a constraint with scope $S \subseteq Y$. We denote by $\vec{a}_S$ the tuple consisting of the values in $\vec{a}$ assigned to variables that are in S. $\vec{a}$ *satisfies* R_S iff $\vec{a}_S \in R_S$; otherwise R_S is *violated* by $\vec{a}$. $\vec{a}$ is *consistent* if $\vec{a}$ satisfies all the applicable constraints, namely, all constraints $R_T, T \subseteq Y$. A consistent partial instantiation is also called a *partial solution.* A *solution* is a consistent instantiation of all the variables. A *nogood* is an assignment of values to an arbitrary subset of variables that is not part of any solution (see also Definition 3). An assignment $x_k = a$ *conflicts with* a partial solution $\vec{a}_i$ if $(\vec{a}_i, x_k = a)$ is not consistent (see also Definition 1).

Example 2. Refering again to the CSP in Figure 1, (T1=2:00, T2=3:00, T3=1:00) is a partial solution, since the relevant constraints (between T1 and T2 and between T1 and T3) are not violated by this partial instantiation. However, this partial solution cannot be extended to include T4. T4=2:00 violates T4's unary constraint, and the other two values conflict with the partial solution: T4=1:00 violates the constraint between T3 and T4 and T4=3:00 violates the constraint between T2 and T4. (T1=3:00, T2=1:00, T3=2:00, T4=3:00, T5=1:00) is a solution to the problem.

2.2 Constraint algorithms

Once a problem of interest has been formulated as a constraint satisfaction problem, a solution can be found with a general purpose constraint algorithm. Constraint satisfaction problems are NP-complete [32]. Many CSP algorithms are based on the principles of search and deduction. In this section we briefly summarize the field of CSP algorithms.

2.2.1 Search based backtracking

The term *search* is used to characterize 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 [62,64]. 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 an uninstantiated variable, thereby extending the current partial solution. It explores the search space in a depth-first manner. If 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 backtracking algorithm was first described more than a century ago, and since then has been reintroduced several times [10].

2.2.2 Deduction based constraint propagation

To solve a problem by deduction is to apply reasoning that transforms 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 as consistency enforcing algorithms [59,48,25]. These procedures transform a constraint network by deducing new constraints, tightening existing constraints, and removing values from variable domains. In general, a consistency enforcing algorithm will make some partial solution of a subnetwork extendible 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. 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 single additional variable [24]. The terms *node-*, *arc-*, and *path-consistency* [48] 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 every single variable that succeeds the $k-1$ variables in the ordering [20]. A problem that is k-consistent for all k is called *globally* consistent.

A variety of algorithms have been developed for enforcing different levels of local consistency, which is also called *constraint propagation* [50,58,14,79,8,20]. For example, arc-consistency algorithms can delete values from the domains of 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 introducing new constraints or nogoods which disallow certain pairs of values. Relational-based consistency enforcing algorithms allow flexible extensions of consistency algorithms that are constraint based rather than variable based [22].

Constraint propagation can be used as a CSP solution procedure. If a problem can be made k-consistent, for all k from 1 to $n-1$, then 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 [14], and so in practice only bounded local consistency enforcing algorithms with $k \leq 3$ are used.

Example 3. 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 incompatible with a unary constraint. Enforcing 2-consistency will cause several other domain values to be removed. For instance, the constraint between T1 and T3 implies 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. As an extreme preprocessing alternative, *adaptive-consistency* [20] is a technique that adjusts the level of enforced consistency on a node by node basis. An ordered constraint graph processed by adaptive-consistency can be solved without backtracking search. The space and time complexity of adaptive-consistency is exponential in a parameter of the ordered graph called w_d^* (see Definition 12). The value of w_d^* can be computed in advance in linear time for a given ordering d. 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 [19].

Varying levels of consistency-enforcing can also be interleaved with the search process. Indeed, this is the primary way consistency enforcing techniques are incorporated into backtracking search and into constraint programming languages [42,70].

2.2.3 Other constraint algorithms

Structure-driven algorithms, which may employ both search and consistency-enforcing components, emerge from studying 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 and therefore a solution to the problem. The basic graph structure that supports tractability is a tree [50]. In particular, enforcing 2-consistency on a tree-structured binary CSP network ensures a solution with no dead-ends along some recognizable ordering of the variables.

A popular class of incomplete algorithms are stochastic methods which typically move in a hill-climbing manner augmented with random steps in the space of complete instantiations [57]. Such techniques are not guaranteed to solve a problem instance. In the CSP community interest in stochastic approaches was sparked by the success of the GSAT algorithm [73] and its variants.

3 Backtracking

3.1 The backtracking algorithm

Backtracking is the primary search algorithm for constraint problems. It traverses the search space of partial instantiations in a depth-first manner. The algorithm maintains a partial solution that denotes a state in the search space. Backtracking has two phases: a forward phase in which the next variable is selected and the current partial solution is extended by assigning a consistent value, if one exists for the next variable; and a backward phase in which, when no consistent solution exists for the current variable, focus returns to the previous variable assigned. Figure 2 describes a basic backtracking algorithm. In this description, BACKTRACKING repeatedly calls the SELECTVALUE subprocedure to find a value for the current variable, x_i, that is consistent with the current partial instantiation, $\vec{a}_{i-i}$. SELECTVALUE, in turn, relies on the CONSISTENT subprocedure, which returns *true* only if current partial solution is consistent with the candidate assignment to the next variable. If SELECTVALUE succeeds in finding a value, BACKTRACKING proceeds to the next variable, x_{i+1}. If SELECTVALUE cannot find a consistent value for x_i, then a *dead-end* occurs, and BACKTRACKING looks for a new value for the previous variable, x_{i-1}. The algorithm terminates when all variables have assignments, or when it has proven that all values of x_1 do not lead to a solution, and thus that the problem is unsolvable. Our presentation of BACKTRACKING stops after a single solution has been found, but it could easily be modified to return all solutions, or a desired number.

The BACKTRACKING procedure employs a series of mutable value domains D'_i such that each $D'_i \subseteq D_i$. D'_i holds the subset of D_i that has not yet been

```
procedure BACKTRACKING
Input: A constraint network P = (X, D, C).
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

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

Fig. 2. The backtracking algorithm.

examined under the current partial instantiation of earlier variables. 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 and CONSISTENT subprocedures are separated from the main BACKTRACKING routine for clarity. Both have access to the local variables and parameters of the main procedure. CONSISTENT handles general binary and non-binary constraints; its implementation, which we do not specify, depends on how constraints are represented by the computer program. The same CONSISTENT subprocedure is used later in the description of other algorithms.

Example 4. Consider the coloring problem in Figure 3. 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, namely partial solutions, are depicted in the figure.

The complexity of CONSISTENT and of SELECTVALUE can be determined, based on the premise that the constraints are stored in tables. Let c be the number of constraints in the problem, and let t be the maximum number of tuples in a constraint. Let m be the maximum size of any domain in D. If the maximum constraint arity is r then $t \leq m^r$. The constraints can be organized to permit finding a tuple of a given constraint in worst-case logarithmic time: $\log t \leq r \log m \leq n \log m$. Since a variable may participate in up to c constraints, the worst-case time complexity of CONSISTENT is $O(c \log t)$ which is also bounded by $O(c\, r \log m)$. SELECTVALUE may invoke CONSISTENT up to m times so the worst-case time complexity of SELECTVALUE is $O(c\, m\, r \log m)$, or $O(c\, m\ \log t)$. For the special case of a binary CSP, it can be practical to store the constraints as a table of boolean values, indexed by the two variables and their values. The tentative instantiation (x_i, a) must then be checked with at most n earlier variables, effectively yielding $O(n)$ complexity for CONSISTENT and $O(nm)$ complexity for SELECTVALUE. If CONSISTENT performs computations other than table lookups, its complexity is of course dependent on the nature of these computations. In summary,

Proposition 1 *For general CSPs with constraints stored in tables, having n variables, c constraints, with constraint arity bounded by r, the number of tuples in a constraint bounded by t, and at most m values for a variable, the time complexity of* CONSISTENT *is $O(c \log t)$ or $O(c\, r \log m)$, and the time complexity of* SELECTVALUE *is $O(c\, m\, r \log m)$ or $O(c\, m \log t)$. For binary CSPs, the complexity of* SELECTVALUE *is $O(n\, m)$.*

3.2 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.

The performance of backtracking can be improved by reducing the size of the search space, which is determined by the algorithm's control strategy, by the constraints' inherent level of local consistency, by the order of variable instantiation, and, when a single solution suffices, by the order in which values are

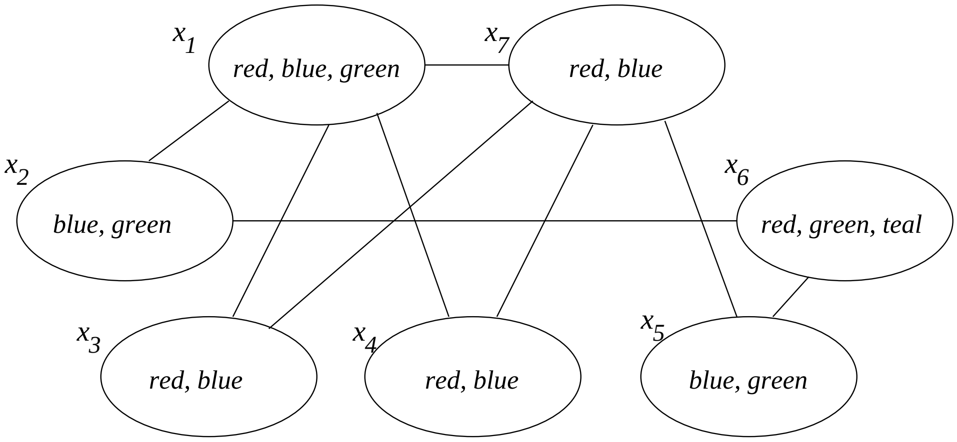


Fig. 3. A coloring problem with variables $(x_1, x_2, \ldots, x_7)$. The domain of each variable is written inside the corresponding node. Each arc represents the constraint that the two variables it connects must be assigned different colors.

assigned to each variable. Addressing 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, which decide which parts of the search space will not be visited. Commonly used pre-processing techniques are arc- and path-consistency algorithms, and heuristic approaches for determining a fixed variable ordering [41,25,21].

Procedures for dynamically improving the pruning power of backtracking can be conveniently classified as *look-back schemes* and *look-ahead 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-back schemes are invoked when the algorithm is preparing to backtrack after encountering a dead-end. These schemes perform two functions:

(1) Deciding how far to backtrack. By analyzing the reasons for the dead-end, irrelevant backtrack points can often be avoided so that the algorithm goes back directly to the source of failure, instead of just to the immediately preceding variable in the ordering. This procedure is often referred to as *backjumping*.
(2) Recording the reasons for the dead-end in the form of new constraints, so that the same conflicts will not arise again later in the search. The terms used to describe this function are *constraint recording* and *learning*.

Look-ahead schemes can be invoked whenever the algorithm is preparing to assign a value to the next variable. Frequently these schemes discover, from a restricted amount of constraint propagation, how the current decisions about variable and value selection will impact future search. This information is used in two ways.

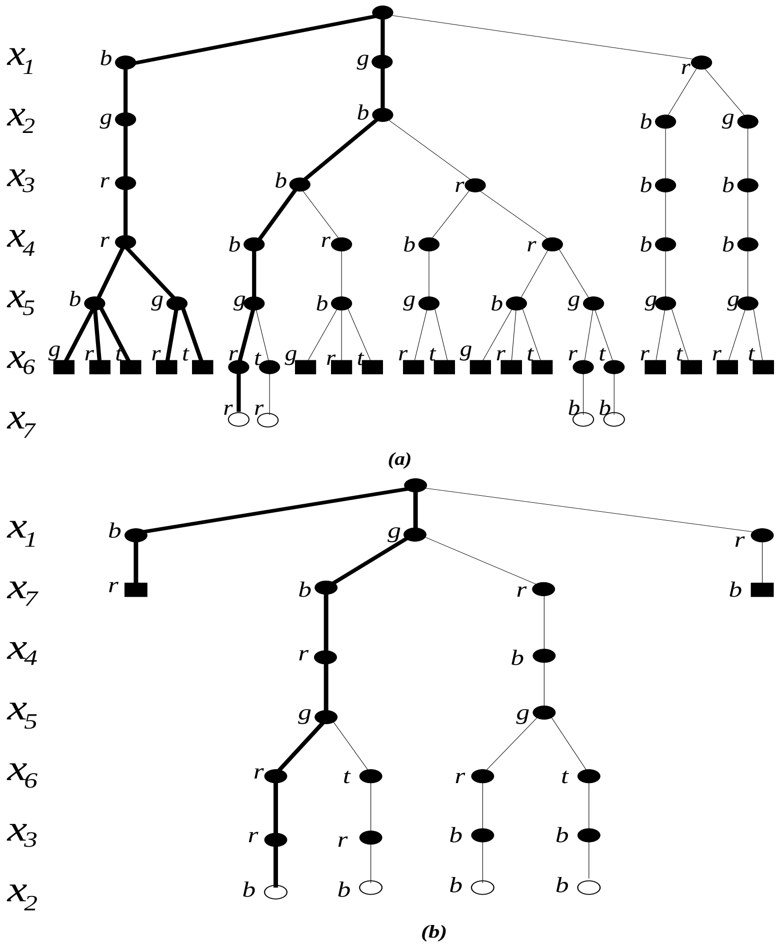


Fig. 4. Backtracking search for the orderings (a) $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$ and (b) $d_2 = x_1, x_7, x_4, x_5, x_6, x_3, x_2$ on the example instance in Figure 3. Intermediate states are indicated by filled ovals, dead-ends by filled rectangles, and solutions by empty ovals. The colors are considered in order (*blue, green, red, teal*), and are abbreviated by their first letters. Thick lines denote the portion of the search space explored by backtracking when stopping after the first solution.

(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. Selecting the variable with least number of values in its domain (after constraint propagation) tends to minimize the size of the search tree [41].

(2) Decide which value to assign to the next variable. 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.

In sections 4 and 5 we will describe in detail several principle look-back schemes, and section 6 will provide an overview of look-ahead methods, including a combination of look-ahead with look-back methods.

4 Backjumping

Backjumping schemes are one of the primary tools for reducing backtracking's undesirable tendency to rediscover the same dead-ends. If a value cannot be found for variable x_i, BACKTRACKING returns 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 encountered 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. Instead, backjumping algorithms can identify the *culprit variable* responsible for the dead-end and then "jump" back immediately to reinstantiate the culprit variable, instead of instantiating the chronologically previous variable repeatedly. Identification of a culprit variable is based on the notion of *conflict sets.*

4.1 Conflict sets

A dead-end state at level i of the search tree indicates that a current partial instantiation $\vec{a}_i = (a_1, ..., a_i)$ conflicts with all values of x_{i+1}. $(a_1, ..., a_i)$ is called a dead-end state and x_{i+1} is called a dead-end variable. 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 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 may retreat 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 $\vec{a}_i = (a_1, \ldots, a_i)$ to the most recent variable x_b such that $\vec{a}_{b-1} = (a_1, \ldots, a_{b-1})$ 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 $\vec{a} = (a_{i_1}, \ldots, a_{i_k})$ be a consistent instantiation of an arbitrary subset of variables, and let x be a variable not yet instantiated. If there is no value b in the domain of x such that $(\vec{a}, x = b)$ is consistent, 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 conflicts with x, $\vec{a}$ is called a* minimal *conflict set of x.*

Definition 2 (leaf dead-end) *Given a variable ordering $d = (x_1, ..., x_n)$, let $\vec{a}_i = (a_1, ..., a_i)$ be a tuple that is consistent. If $\vec{a}_i$ is in conflict with x_{i+1} it is* *called a* leaf dead-end.

Definition 3 (nogood) *Given a problem $P = (X, D, C)$, any partial instantiation $\vec{a}$ that does not appear in any solution of P is called a* nogood. *Minimal nogoods have no nogood subtuples.*

A conflict set is clearly a nogood, but there are nogoods that are not conflict sets of any single variable. Namely, they may conflict with two or more variables simultaneously.

Example 5. Consider the CSP from Figure 3, with the variable ordering $(x_1, x_5, x_4, x_7, x_2, x_3, x_6)$. The partial instantiation $(x_1 = blue, x_5 = blue)$ is not a dead-end, and is not a conflict set of any later variable. However, $(x_1 = blue, x_5 = blue)$ is a nogood, since there is no instantiation of both x_4 and x_7 consistent with these values of x_1 and x_5.

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

Definition 4 (safe jump) *Let $\vec{a}_i = (a_1, ..., a_i)$ be a leaf dead-end state. We say that jumping to x_j, where $j \leq i$, is* safe *if the partial instantiation $\vec{a}_j = (a_1, \ldots, a_j)$ is a nogood, 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 (culprit variable) *Let $\vec{a}_i = (a_1, \ldots, a_i)$ be a leaf dead-end. The* culprit index *relative to $\vec{a}_i$ is defined by $b = \min\{j \leq i \mid \vec{a}_j$ 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 jumping back to x_b is both safe and maximal: safe in that $\vec{a}_b$ cannot be extended to a solution; maximal in that jumping back to an earlier node risks missing a solution.

Proposition 2 *If $\vec{a}_i$ is a leaf dead-end discovered by backtracking, and x_b is* *the culprit variable, then $\vec{a}_b$, is a safe backjump destination, and $\vec{a}_u, u < b$, is not.*

Proof: By definition of a culprit, $\vec{a}_b$ is a conflict set of x_{i+1} and therefore is a nogood. 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 maximality, observe that jumping farther back to an earlier node risks skipping potential solutions. Specifically, if the algorithm jumps to $x_u, u < b$, then by the definition of culprit variable $\vec{a}_u$ is not a conflict set of x_{i+1}, and therefore it may be part of a solution. Note that $\vec{a}_u$ may or may not be a nogood of the original problem, but it is not a nogood of the subproblem defined on $\{x_1 \ldots x_i\}$, and therefore the backjumping algorithm cannot determine whether it is safe, given the information it has. □

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 back-track points exclusively from the constraint graph. It introduces the notion of jumping back at internal dead-ends as well as leaf dead-ends. *Conflict-directed backjumping* combines maximal 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 [34] records some information while generating $\vec{a}_i$, and uses this information to determine the dead-end's culprit variable x_b. Gaschnig's backjumping algorithm is presented in Figure 5; the subprocedure SELECTVALUE-GBJ identifies and records the culprit variable. As originally described by Gaschnig, the algorithm handled binary CSPs. Our version makes a straight-forward extension to high-arity constraints. Each variable x_i has an associated number $latest_i$, which points, when x_i is a dead-end variable, to the *latest* predecessor tested for incompatibility with some possible instantiation of x_i. With a binary CSP, we can say more concretely that, letting $latest_i = b$, there is a binary constraint prohibiting the current instantiation of x_b and (x_i, a'), for some $a' \in D_i$, while (x_i, a') is consistent with $\vec{a}_{b-1}$. If x_i does have at least one consistent value, relative to $\vec{a}_{i-1}$, then x_i is not a dead-end variable, and $latest_i$ is assigned the value $i-1$. GASCHNIG'S-BACKJUMPING jumps from a leaf dead-end $\vec{a}_i$ that is inconsistent with x_{i+1}, back to $x_{latest_{i+1}}$, the culprit since the dead-end variable is x_{i+1}.

Proposition 3 *Gaschnig's backjumping implements only safe and maximal backjumps in leaf dead-ends.*

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

subprocedure 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 > latest_i          (latest_i records the latest)
             latest_i ← k          (variable checked for consistency)
          if CONSISTENT(a⃗_k, x_i = a)
             k ← k + 1
          else
             consistent ← false
       end while
       if consistent
          return a
    end while
    return null                    (no consistent value)
end procedure
```

Fig. 5. Gaschnig's backjumping algorithm.

Proof: Whenever there is a leaf dead-end $\vec{a}_{i-1}$, the algorithm has a partial instantiation $\vec{a}_{i-1} = (a_1, \ldots, a_{i-1})$. Let $j = latest_i$. The algorithm jumps back to 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 $j = latest_i$ 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, as defined in Definition 5. From Proposition 2, it follows that this algorithm is safe and maximal. □

Example 6. For the problem in Figure 3, at the dead-end for x_7 ($x_1 = red$, $x_2 = blue$, $x_3 = blue$, $x_4 = blue$, $x_5 = green$, $x_6 = red$), $latest_7 = 3$, 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 $latest_3 = 2$, the next variable examined will be x_2. This demonstrate the algorithm's ability to backjump on leaf dead-ends. On subsequent dead-ends (at x_3) it goes back to its preceding variable only.

In Gaschnig's backjumping, a jump is made only at a leaf dead-end. If all the children of a node in the search tree lead to dead-ends, as happens with $x_3 = red$ under $x_2 = green$ in Figure 4 (a), the node is called 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. (The graph need not represent a binary CSP.) Whenever a dead-end occurs and a partial solution cannot be extended to the next variable x, the algorithm jumps back to the most recent variable y adjacent to x in the constraint graph; if y has no more values (namely, it is an internal dead-end), 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 $latest_i$ for each consistency test. Graph-based backjumping uses the subset of earlier variables adjacent to x_{i+1} in the constraint graph as an approximation of the minimal conflict set of x_{i+1}. Even when a constraint exists between two variables x_h and x_{i+1}, the particular value currently assigned to x_h may not conflict with any potential value of x_{i+1}.

The importance of graph-based backjumping is that studying algorithms with

```
procedure GRAPH-BASED-BACKJUMPING
Input: A constraint network P = (X, D, C).
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)                 (initialize induced ancestor set)
    while 1 ≤ i ≤ n
       instantiate x_i ← selectValue
       if x_i is null              (no value was returned)
          iprev ← i
          i ← latest in I_i        (backjump)
          I_i ← I_i ∪ I_iprev − {x_i}(merge to update I_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

subprocedure 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 CONSISTENT(a⃗_{i−1}, x_i = a)
          return a
    end while
    return null                    (no consistent value)
end procedure
```

Fig. 6. The graph-based backjumping algorithm.

performance tied to the constraint graph leads to graph-theoretic complexity bounds and thus to graph-based heuristics aimed at reducing these bounds. Such bounds are also applicable to algorithms that use refined run-time information such as Gaschnig's backjumping and conflict-directed backjumping. In particular, we will show that when using a depth-first search ordering of the variables, graph-based backjumping is simple to implement and allows a complexity bound as a function of the depth of the constraint graph's depth-first search spanning tree.

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

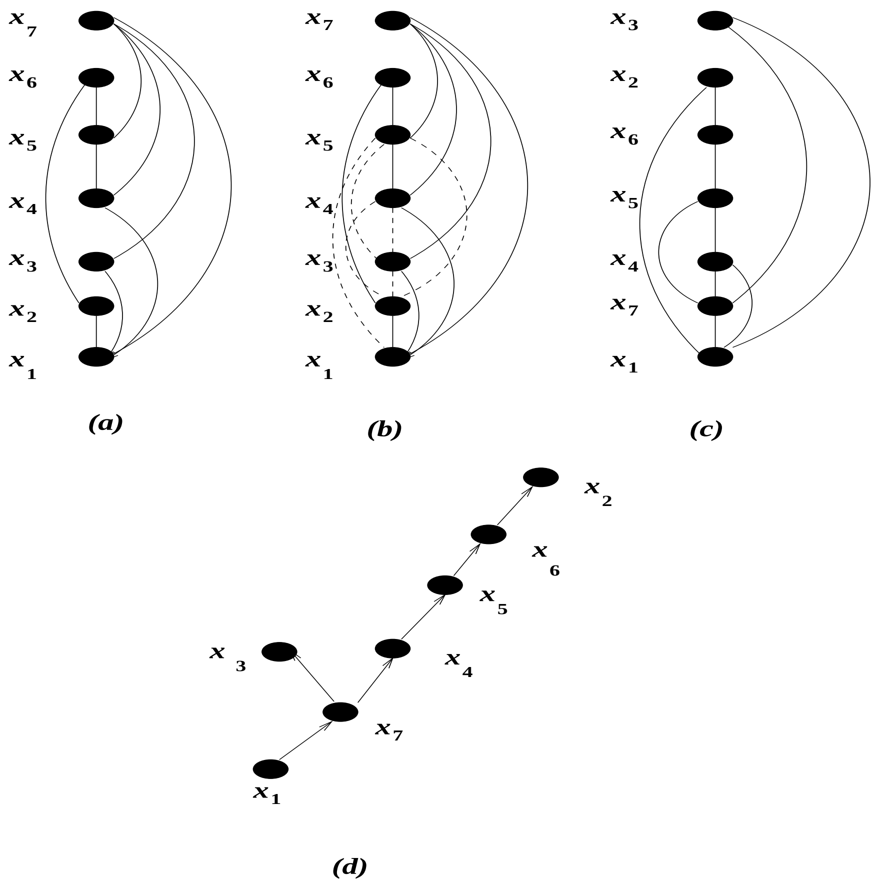


Fig. 7. Ordered constraint graphs on the problem in Figure 3: (a) $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$; (b) the induced graph along d_1; (c) $d_2 = x_1, x_7, x_4, x_5, x_6, x_2, x_3$; (d) a *DFS* spanning tree along ordering d_2.

Definition 6 (ancestors, parent) *Given a constraint graph and an ordering of the nodes 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 $\vec{a}_i = (a_1, \ldots, a_i)$ is a leaf dead-end, we equate $anc(\vec{a}_i)$ with $anc(x_{i+1})$, and $p(\vec{a}_i)$ with $p(x_{i+1})$.*

Example 7. Consider the ordered graph in Figure 7a along the ordering $d_1 = x_1, \ldots, x_7$. 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, jumping back to $p(\vec{a}_i)$ is safe. Moreover, if only graph-based information is utilized, and culprit variables are not computed as in Gaschnig's backjumping, 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 $p(\vec{a}_i)$, as the next example demonstrates.

Example 8. Consider again the constraint network in Figure 3 with ordering $d_1 = x_1, \ldots, x_7$. 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 graph-based culprit variable is determined by the *induced ancestor set* in the current *session*.

Definition 7 (session) *We say that backtracking* invisits x_i *if it processes x_i coming from a variable earlier in the ordering. A* session *of x_i starts upon invisiting x_i and ends when retracting to a variable that precedes x_i. Given a constraint network that is being searched by a backtracking or backjumping algorithm, the* current session *of x_i is the set of variables invisited by the algorithm since the latest* invisit *to x_i.*

Definition 8 (induced ancestors, induced parent) *Let x_i be a variable that is an internal dead-end. Let Y be a subset of the variables consisting of all the dead-ends in the current session of x_i. We denote $anc(Y) = \cup_{y \in Y} anc(y)$. The* induced ancestor set *of x_i relative to Y, $I_i(Y)$, is the union of all Y's ancestors, restricted to variables that precede x_i. Formally,*

$$I_i(Y) = anc(Y) \cap \{x_1, ..., x_{i-1}\}$$

The induced parent of x_i relative to Y, $P_i(Y)$, is the latest variable in $I_i(Y)$.

Theorem 1 *Let $\vec{a}_i$ be a dead-end (leaf or internal), 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 safe variable to jump to.*

Proof: By definition of x_j, all the variables between x_j and x_{i+1} 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 nogoods that caused the dead-end in $\vec{a}_i$, and so they can be skipped.

We next 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 variable in Y that is connected to x_j. We argue that there is no way, based on graph information only, to rule out the possibility that there exists an alternative value of x_j that may lead to a solution. Let x_i be assigned the value 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, and since x_j is an ancestor of y_i, there might be a constraint R, whose scope S contains x_j and y_i, such that the current assignment $\vec{a}_i$ restricted to S cannot be extended to a legal value of y_i. Clearly, had the value of x_j been changed, the current assignment may be extendible to a legal tuple and 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 (when using the graph only), however, that different values of x_j, if attempted, could permit new values of y_i for which a solution might exist. $\square$

Example 9. Consider again the ordered graph in Figure 7a, and let x_4 be a dead-end variable. If x_4 is a leaf dead-end, then $Y = \{\}$, 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 $I_4(\{x_5, x_6\}) = \{x_1, x_2\}$, 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 $I_3(\{x_4, x_5, x_6, x_7\}) = \{x_1, x_2\}$.

Algorithm GRAPH-BASED-BACKJUMPING in Figure 6 implements the principles of graph-based backjumping. It jumps to maximal culprit variables 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.

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 [67]. This algorithm uses the scheme we have outlined for graph-based backjumping but, rather than relying on 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.*

We first define an ordering between constraints. Let $scope(R)$ denote the scope of constraint R.

Definition 9 (earlier constraint) *Given an ordering of the variables in a constraint problem, we say that constraint R is* earlier *than constraint Q if the latest variable in $scope(R)-scope(Q)$ precedes the latest variable in $scope(Q)-scope(R)$.*

For instance, under the variable ordering $(x_1, x_2, \ldots)$, if the scope of constraint R_1 is (x_3, x_5, x_8, x_9) and the scope of constraint R_2 is (x_2, x_6, x_8, x_9), then R_1 is earlier than R_2 because x_5 precedes x_6. Given an ordering of all the variables in X, the *earlier* relation defines a total ordering on the constraints in C.

Definition 10 (earliest minimal conflict set) *For a problem $P = (X, D, C)$ with an ordering of the variables d, let $\vec{a}_i$ be a dead-end tuple whose dead-end variable is x_{i+1}. The* earliest minimal conflict set *of $\vec{a}_i$, denoted $emc(\vec{a}_i)$, can be generated as follows. Consider the constraints in $C = \{R_1, \ldots, R_c\}$ with scopes $\{S_1, \ldots, S_c\}$, in order as defined in Definition 9. For $j = 1$ to c, if there exists $b \in D_{i+1}$ such that R_j is violated by $(\vec{a}_i, x_{i+1} = b)$, but no constraint earlier than R_j is violated by $(\vec{a}_i, x_{i+1} = b)$, then $var\text{-}emc(\vec{a}_i) \leftarrow var\text{-}emc(\vec{a}_i) \cup S_j$. $emc(\vec{a}_i)$ is the subtuple of $\vec{a}_i$ containing just the variable-value pairs where the variable is in $var\text{-}emc(\vec{a}_i)$.*

Definition 11 (jumpback set) *The* jumpback set *J_i of a dead-end $\vec{a}_i$ is defined to include the $var\text{-}emc(\vec{a}_j)$ of all the dead-ends $\vec{a}_j$, $j \geq i$, that occurred in the current session of x_i. Formally,*

$$J_i = \bigcup \{var\text{-}emc(\vec{a}_j) \mid \vec{a}_j \text{ is a dead-end in } x_i\text{'s session}\}$$

$var\text{-}emc(\vec{a}_i)$ plays 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 can be uncovered during search. The set of variables in $var\text{-}emc(\vec{a}_i)$ is a subset of the set of graph-based variables in $anc(x_{i+1})$. The variables in the graph-based $anc(x_{i+1})$ that are not included in $var\text{-}emc(\vec{a}_i)$, either participate only in irrelevant constraints (do not exclude any value of x_{i+1}) relative to the current instantiation $\vec{a}_i$ or, even if relevant, are superfluous, as they rule out values of x_{i+1} that were eliminated by earlier constraints. Consequently, using similar arguments as in the graph-based case, it is possible to show that:

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

Proof (sketch): Let x_j be the latest variable in the jumpback set J_i of a dead-end $\vec{a_i}$. As in the graph-based case, jumping back to a variable later than x_j will not remove some of the nogoods that were active in causing this dead-end, and therefore the same dead-end will recur. To show that we cannot jump back any earlier than x_j we need to show that because we generate the *var-emc* set by looking at earliest constraints first, it is not possible that there exists an alternative set of constraints for which $\vec{a}_i$ is a dead-end and for which the jumpback set yields an earlier culprit variable. Therefore, it is possible that changing the value of x_j will yield a solution, and this solution could be missed if we jumped to an earlier variable. □

Algorithm CONFLICT-DIRECTED-BACKJUMPING is presented in Figure 8. It computes the jumpback sets for each variable, and uses them to determine the variable to which it returns after a dead-end. Therefore,

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

Example 10. Consider the problem of Figure 3 using ordering $d_1 = (x_1, \ldots, x_7)$. Given the dead-end at x_7 and the assignment $\vec{a_6} = (blue, green, red, red, blue, red)$, the *emc* 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 *var-emc* set is $\{x_1\}$, the jumpback set of x_3 includes just x_1, and the algorithm jumps again, this time back to x_1.

4.5 Complexity of backjumping

We will now return to graph-based backjumping and show how graph information can yield graph-based complexity bounds that are relevant to all variants of backjumping.

Although the implementation of graph-based backjumping 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 [23]). The

```
procedure CONFLICT-DIRECTED-BACKJUMPING
Input: A constraint network P = (X, D, C).
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-CBJ
        if x_i is null                 (no value was returned)
            iprev ← i
            i ← index of last variable in J_i     (backjump)
            J_i ← J_i ∪ J_iprev − {x_i}(merge conflict sets)
        else
            i ← i + 1                  (step forward)
            D'_i ← D_i                 (reset mutable domain)
            J_i ← ∅                    (reset conflict set)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, . . . , x_n}
end procedure

subprocedure 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 CONSISTENT(a⃗_k, x_i = a)
                k ← k + 1
            else
                let R_S be the earliest constraint causing the conflict
                add the variables in R_S's scope S, but not x_i, to J_i
                consistent ← false
        end while
        if consistent
            return a
    end while
    return null                        (no consistent value)
end procedure
```

Fig. 8. The conflict-directed backjumping algorithm.

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 maximal 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 11. 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 2 *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 $f(x)$. $\square$

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

Theorem 3 *When graph-based backjumping is performed on a* DFS *ordering of the constraint graph, its complexity is $O(b^m k^{m+1})$ 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_i be a node in the DFS spanning tree whose DFS subtree has depth of m. Let T_i stands for the maximal number of nodes in the search-tree rooted at x_i, namely, T_i is the maximum number of nodes visited in any session of x_i. Since any assignment of a value to x_i generates at most b subtrees of depth $m-1$ or less that can be solved independently, T_i obeys the following recurrence:

$$T_i = k \cdot b \cdot T_{i-1}$$
$$T_0 = k$$

Solving this recurrence yields $T_i = b^m k^{m+1}$. Thus, the worst-case time complexity of graph-based backjumping in terms of number of nodes visited 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 $T_i = O((n/b)k^{m+1})$, since $n = O(b^{m+1})$. $\square$

The bound suggests a graph-based ordering heuristic: use a *DFS* ordering having a minimal depth. Unfortunately, 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 12 (width, tree-width) *Given a graph G over nodes $X = \{x_1, \ldots, x_n\}$, and an ordering $d = x_1, \ldots, x_n$, 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 [1] *of the graph, is identical to the induced width. For more information, see [20].*

Example 12. Consider the graph in Figure 7a ordered along $d_1 = x_1, \ldots, x_7$. 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 an ordering that is also a *DFS* ordering of its induced graph G^*_d and let m^*_d be the *DFS* tree depth.

Theorem 4 *Let m^*_d be the depth of a* DFS *tree traversal of the induced graph*

G_d^* of G. If d is a DFS *ordering of G_d^*, then the complexity of graph-based backjumping using ordering d is* $O(\exp(m_d^*))$.

A proof, that uses somewhat different terminology and derivation, is given in [5].

The virtue of Theorem 4 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 every *DFS* ordering of G is also a *DFS* ordering of its induced graph along d, G_d^* (the added induced arcs are back arcs of the *DFS* tree), *DFS* orderings of G are a subset of all *DFS* orderings of all of G's induced graphs. Thus, this may lead to better orderings having better bounds for backjumping.

4.6 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 techniques for 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 jump back maximally 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 every node and can be exploited to generate more refined conflict sets than in 1-backjumping, namely, conflict sets whose nogoods 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, of course, be studied empirically.

5 Learning Algorithms

The earliest minimal conflict set of Definition 10 is a nogood explicated by search and used to guide backjumping. However, this same nogood may be rediscovered again and again as the algorithm explores different paths in the search space. By making this nogood explicit, in the form of a new constraint, we can make sure that the algorithm will not rediscover it. Doing so may prune the remaining search space. This technique, called constraint recording or *learning*, is behind the learning algorithms described in this section [17].

An opportunity to learn new constraints is presented whenever the backtracking algorithm encounters a dead-end, namely, when the current instantiation $\vec{a}_i = (a_1, \ldots, a_i)$ 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. [2] If $\vec{a}_i$ 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.

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 nonminimal conflict sets to

---

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

[3] We make the assumption that the computer program represents constraints internally by storing the invalid combinations. Thus, increasing the number of stored nogoods increases the size of the data structure and slows down retrieval.

be recorded 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 additional memory to store, and are more expensive to consult than constraints having fewer variables. The algorithm may record a single nogood or multiple nogoods 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 scope size of the constraints recorded, referred to as *bounded learning*. These algorithms exemplify the main alternatives, although there are numerous possible variations.

5.1 Graph-based learning

Graph-based learning uses the same methods as graph-based backjumping to identify a nogood, namely, information on conflicts is derived from the constraint graph alone. Given a leaf dead-end $(a_1, \ldots, a_i)$, the values assigned to the ancestors of x_{i+1} in the graph are identified and included in the conflict set that is recorded.

Example 13. Consider the problem from Figure 3, when searching for all solutions. Figure 9 presents the search space explicated by naive backtracking and by backtracking augmented with graph-based learning. Branches below the cut lines in Figure 9 are generated by the former but not by the latter. Leaf dead-ends are numbered (1) through (10) (only dead-end that appear in the search with learning are numbered). At each dead-end, search with learning can record a new constraint. At dead-end (1), no consistent value exists for x_1. The ancestor set of x_1 is $\{x_2, x_3, x_4, x_7\}$, so graph-based learning records the nogood $(x_2 = \mathit{green}, x_3 = \mathit{blue}, x_4 = \mathit{blue}, x_7 = \mathit{red})$. This nogood reappears later in the search, under the subtree rooted at $x_6 = \mathit{teal}$, and it can be used to prune the search at the dead-end numbered (9). The dead-ends labeled (2) and (4) occur because no consistent value is found for x_7, which has the ancestor set $\{x_3, x_4\}$. The nogoods $(x_3 = \mathit{blue}, x_4 = \mathit{red})$ and $(x_3 = \mathit{red}, x_4 = \mathit{blue})$ are therefore recorded by graph-based learning, in effect creating an "equality" constraint between x_3 and x_4. (The dead-ends at (3) and (5) involve the same nogoods; if graph-based backjumping is used instead of backtracking, these dead-ends will be avoided.) This learned constraint prunes the remaining search four times. The following additional nogoods are recorded by graph-based learning at the indicated dead-ends: (6): $(x_2 = \mathit{green}, x_3 = \mathit{red}, x_4 = \mathit{red}, x_7 = \mathit{blue})$; (7): $(x_4 = \mathit{blue}, x_6 = \mathit{green}, x_7 = \mathit{red})$; (8): $(x_4 = \mathit{red}, x_6 = \mathit{green}, x_7 = \mathit{blue})$; (9): $(x_2 = \mathit{green}, x_3 = \mathit{blue}, x_4 = \mathit{blue})$; (10): $(x_2 = \mathit{green}, x_3 = \mathit{red}, x_4 = \mathit{red})$.

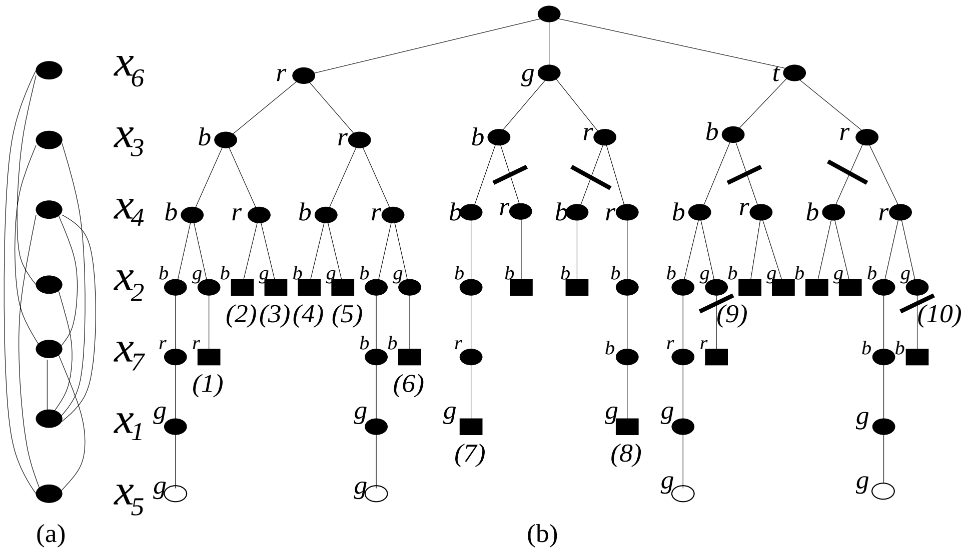


Fig. 9. The search space explicated by backtracking on the CSP from Figure 3, using the variable ordering $(x_6, x_3, x_4, x_2, x_7, x_1, x_5)$ and the value ordering (*blue, red, green, teal*). Part (a) shows the constraint graph; part (b) illustrates the search space. The cut lines in (b) indicate branches not explored when graph-based learning is used.

Note that dead-ends (9) and (10) occur at x_2 with learning, and at x_7 without learning.

The complexity of learning at each dead-end, with graph-based learning, 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 need only add a line (in bold face) to GRAPH-BASED-BACKJUMPING from Fig. 6 after a dead-end is encountered:

instantiate $x_i \leftarrow$ SELECTVALUE
if x_i is null (no value was returned)
  record a constraint prohibiting I_i and corresponding values
  iprev $\leftarrow i$
  (algorithm continues as in Fig. 6)

Recording a new constraint may require adding a new relation R_{c+1} to the list of constraints C. If a constraint with scope I_i already exists, it may only be necessary to remove a value tuple from this constraint.

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 and ordering of Example 13 at the first dead-end, deep learning will record the minimal conflict set $(x_2 = \text{green}, x_3 = blue, x_7 = red)$ (or perhaps $(x_2 = green, x_4 = blue, x_7 = red)$, or both), instead of the non-minimal conflict set including both x_3 and x_4 that is recorded by graph-based learning. Although deep learning is the most informative, 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 of the dead-end variable. The number of such minimal conflict sets will be $\binom{r}{r/2} \cong 2^r$, 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, graph-based learning records the largest size constraints, and deep learning records the smallest ones. As noted for backjumping, the virtues of graph-based learning are mainly theoretical (see Section 5.6); we do not advocate using this algorithm in practice since jumpback learning is always superior. Neither do we recommend using deep learning, because its cost is usually prohibitive.

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* [27] uses this jumpback set, with the values assigned to the variables, as the conflict set to be learned. Because the conflict set is calculated by the underlying backjumping algorithm, the time complexity, at a dead-end, of jumpback learning is only that of storing the conflict set. As with graph-based learning, the modification required to augment CONFLICT-DIRECTED-BACKJUMPING into a learning algorithm is minor: specifying that the conflict set is recorded as a nogood after each dead-end.

instantiate $x_i \leftarrow$ SELECTVALUE-CBJ
if x_i is null (no value was returned)
record a constraint prohibiting J_i and corresponding values
$iprev \leftarrow i$
(algorithm continues as in Fig. 8)

Example 14. For the problem and ordering of Example 13 at the first dead-end, jumpback learning will record the nogood ($x_2 = green, x_3 = blue, x_7 = red$), since that tuple includes the variables in the jumpback set of x_1.

5.4 Bounded learning and relevance 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, the result is 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.

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 13 (i-relevant) *[5] A nogood is i-relevant if it differs from the current partial assignment by at most i variable-value pairs.*

Definition 14 (i'th order relevance bounded learning) *[5] An i'th order relevance bounded learning scheme maintains only those learned nogoods that are i-relevant.*

The *dynamic backtracking* algorithm [38] employs a similar notion of retaining only learned nogoods that are most likely to be consulted in the near future search.

5.5 Nonsystematic randomized backtracking with learning

Learning can be used to make incomplete search algorithms complete, that is, guaranteed to terminate with a solution or with proof that no solution exists. Consider a backtracking-based algorithm that after a fixed number of dead-ends restarts, if it has not terminated normally, with a different, randomly selected, variable and value ordering. Study of this approach has been motivated by observing the performance of incomplete greedy local search algorithms that often outperform traditional backtacking-based algorithms. The problem is that randomization makes the search algorithm incomplete.

Completeness is guaranteed even with randomization, however, when all nogoods discovered are recorded and consulted subsequently in the search, including after randomized restarts. Such randomized, learning-based algorithms are complete because whenever they reach a dead-end they discover and record a *new* conflict-set. Since the number of conflict-sets is finite such algorithms are complete: they are guaranteed to find a solution or prove that no solution exists. In the next subsection we use the same argument to bound the complexity of learning-based algorithms.

5.6 Complexity of backtracking with learning

Graph-based learning yields a useful complexity bound on backtracking's performance parameterized by the induced width w^*, introduced in Definition 12. 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 5 *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 $O((2nm)^{w^*(d)+1})$, where n is the number of variables and m bounds the domain sizes.*

Proof: Graph-based learning has a one-to-one correspondence between dead-ends and conflict sets. Backtracking with graph-based learning along d records conflict sets of size $w^*(d)$ or less, because the dead-end variable will not be connected to more than $w^*(d)$ earlier variables by both original constraints and recorded ones. Therefore the number of dead-ends is bounded by the number of possible nogoods of size $w^*(d)$ or less, yielding a space comlexity of

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

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, $m_d^* \geq w^*(d)$. However, it can be shown [5] 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.

6 Look-ahead Strategies

We now turn our attention to look-ahead methods, which are designed to improve the "forward" phase of backtracking-based algorithms.

6.1 Combining backtracking and constraint propagation

One way CSP search algorithms can combine backtracking and local constraint propagation is by applying a consistency enforcing procedure to the current subproblem. This combination is known as "looking ahead" because the decision to accept or reject a value for the current variable is based on the impact that assignment has when constraint propagation is applied to the set of uninstantiated "future" variables. A partial instantiation may induce constraints on the remaining variables, and making these constraints explicit may reveal useful information that can reduce the amount of backtracking search subsequently required. Of course, actions conditioned on a partial instantiation will have to be retracted if the partial instantiation becomes no longer current due to backtracking.

Example 15. Consider the coloring problem in Figure 3, and assume that variable x_1 is first in the ordering and has been assigned the value *red*. A look-ahead procedure can note that the value *red* in the domains of x_3, x_4, and x_7 is incompatible with the partial instantiation, and provisionally remove 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 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

procedure BACKTRACKING-WITH-LOOKAHEAD
Input: A constraint network $P = (X, D, C)$
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-xxx
       if x_i is null             (no value was returned)
          i ← i − 1               (backtrack)
          reset each D'_k, k > i, to its value before x_i was last instantiated
          remove any constraints added since x_i was last instantiated
       else
          i ← i + 1               (step forward)
    end while
    if i = 0
       return "inconsistent"
    else
       return instantiated values of {x_1, ..., x_n}
end procedure
```

Fig. 10. A common framework for several look-ahead based search algorithms. By replacing SELECTVALUE-XXX with SELECTVALUE-FORWARD-CHECKING the forward checking algorithm is obtained. Similarly, using SELECTVALUE-ARC-CONSISTENCY yields an algorithm that interleaves arc-consistency and search.

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 much smaller search spaces typically result when look-ahead is employed. In general, the stronger the level of constraint propagation being used for look-ahead, 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

The algorithm BACKTRACKING-WITH-LOOKAHEAD in Figure 10 presents a framework for look-ahead algorithms that can be specialized based on the level of constraint propagation, expressed in the specific SELECTVALUE subprocedure employed. BACKTRACKING-WITH-LOOKAHEAD initially sets up all

```
subprocedure 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
      for all values b in D'_k
        if not consistent(a⃗_{i-1}, x_i = a, x_k = b)
          remove b from D'_k
      end for
      if D'_k is empty          (x_i = a leads to a dead-end)
        empty-domain ← true
    end for
    if empty-domain             (don't select a)
      reset each D'_k, i < k ≤ n to status before a was selected
    else
      return a
  end while
  return null                   (no consistent value)
end procedure
```

Fig. 11. The selectValue subprocedure for the forward checking algorithm.

D' sets to be equivalent to the D sets, and the selectValue subprocedure propagates the current instantiation to remove values from the D' sets. Upon backtracking, backtracking-with-lookahead resets D' sets in order to rescind modifications that were contingent on no longer current partial instantiations. (Because the look-ahead selectValue subprocedures we present below will restore the future D' sets after a leaf dead-end, it is only necessary for backtracking-with-lookahead to perform its reset action after internal dead-ends.) Programmers usually use n copies of each D' set, one for each level in the search tree, to permit the reset action to be performed efficiently.

6.3 Forward checking

Forward checking [41], which uses selectValue-forward-checking in Figure 11, does a limited form of constraint propagation. If variables x_1 through x_i have been instantiated, then $n - i$ subproblems can be created by combining the instantiated variables $\vec{a}_i$ with one uninstantiated variable x_u. The only constraints of interest in each subproblem are those in C with a scope that is a subset of $\{x_1, \ldots, x_i\} \cup \{x_u\}$. Enforcing consistency on each subproblem is achieved by removing from x_u's domain any values that conflict with $\vec{a}_i$. Forward checking treats these $n - i$ subproblems independently of each other, removing values from D' sets as necessary. If the domain of one of the future variables becomes empty, then the partial instantiation $\vec{a}_i$ cannot be extended to that variable, and hence $\vec{a}_i$ is not part of a solution. Another value for x_i, the current variable, is therefore considered. If the complexity of CONSISTENT is $O(c \log t)$, where c is the number of constraints and t is the maximum number of tuples per constraint, then SELECTVALUE-FORWARD-CHECKING's complexity is $O(nm^2 c \log t)$, where n is the number of varibles and m is the cardinality of the largest domain. The **while** loop can be executed up to m times, and the outer and inner **for** loops are bounded by n and m, respectively.

An interesting relationship between forward checking and the simplest form of backjumping is:

Proposition 6 *[45] When using the same variable ordering, Gaschnig's backjumping always explores every node explored by forward checking.* □

6.4 Using more look-ahead than forward checking

More work is done at each instantiation by look-ahead algorithms that enforce arc-consistency on the uninstantiated variables after each assignment of a value to the current variable. If a variable's domain becomes empty during the process of enforcing arc-consistency, then the current value is rejected. SELECTVALUE-ARC-CONSISTENCY in Figure 12 implements this approach. The **repeat** ... **until** loop in the subprocedure is essentially an arc-consistency algorithm called AC-1 [48]. There is a long history of arc-consistency algorithms, of which AC-1 is the earliest, perhaps the simplest, and certainly one of the least efficient. Later and more efficient arc-consistency procedures, dubbed AC-2 through AC-7 [48,58,79,65,8,9] can also be used within a SELECTVALUE subprocedure. The optimal time complexity for any arc-consistency procedure is $O(em^r)$, where e is the number of constraints in the subproblem, m is the cardinality of the largest domain, and r is the largest arity of the constraints.

Two algorithms that do more work than forward checking and less work than enforcing arc-consistency at each level in the search are *full looking ahead* and *partial looking ahead* [41]. The full looking ahead algorithm makes a single pass through the future variables; in effect the "**repeat**" and "**until**" in SELECTVALUE-ARC-CONSISTENCY are removed. Partial looking ahead does less work than full looking ahead: in addition to removing the **repeat** loop from SELECTVALUE-ARC-CONSISTENCY, partial looking ahead replaces "**for** all $k, i < k \leq n$" with "**for** all $k, j < k \leq n$". That is, future variables are only compared with those following them.

Although applying arc-consistency was highly successful on a class of vision instances [80], varieties of look-ahead which do more work than forward checking have often been regarded as less useful. This may be due, in part, to the

```
subprocedure selectValue-arc-consistency
  while D'_i is not empty
    select an arbitrary element a ∈ D'_i, and remove a from D'_i
    repeat
    removed-value ← false
      for all j, i < j ≤ n
        for all k, i < k ≤ n
          for each value b in D'_j
            if there is no value c ∈ D'_k such that
                  consistent(a_{i-1}, x_i = a, x_j = b, x_k = c)
              remove b from D'_j
              removed-value ← true
          end for
        end for
      end for
    until removed-value = false
    if any future domain is empty   (don't select a)
      reset each D'_j, i < j ≤ n, to value before a was selected
    else
      return a
  end while
  return null                (no consistent value)
end procedure
```

Fig. 12. The selectValue subprocedure for arc-consistency, based on the AC-1 algorithm.

prematurely negative conclusions about full looking ahead reached in [41]: "The checks of future with future units do not discover inconsistencies often enough to justify the large number of tests required." Subsequent experimentation with larger problems than those used in [41] have justified the value of interleaving arc-consistency enforcing procedures with search.

A search algorithm can also enforce a higher degree of consistency than arc-consistency after each instantiation. Doing so will entail not only deleting values from domains, but adding new constraints (which again may have to be retracted). 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 in constraint propagation is studied in [29,71,4,31]. It is likely that as experiments are conducted with larger and harder problems, look-ahead based on path-consistency will be cost-effective.

We end this section by presenting a relationship between the structure of the

```
subprocedure selectVariable
    m ← min_{i≤j≤n} |D'_j|          (find size of smallest future domain)
    select an arbitrary uninstantiated variable x_k such that |D'_k| = m
    rearrange future variables so that x_k is the ith variable
end subprocedure
```

Fig. 13. The subprocedure SELECTVARIABLE, which employs a heuristic based on the D' sets to choose the next variable to be instantiated.

constraint graph and some forms of look-aheads.

Definition 15 (cycle-cutset) *Given an undirected graph, a subset of nodes in the graph is called a* cycle-cutset *if its removal results in a graph having no cycles.*

Proposition 7 *A constraint problem whose graph has a cycle-cutset of size c can be solved by the partial looking 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 looking 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 $(n-c)k^2$ consistency checks, the complexity follows. For more details see [17]. □

6.5 Using 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 [19]. When using a dynamic variable ordering (DVO), the usual objective, known as "fail first" [41], is to select as the next variable the one that is predicted, by some heuristic, to have the smallest search tree below it. All other factors being equal, the variable with the smallest number of values in its (current) domain will have the fewest subtrees rooted at those values, and therefore the smallest search space below it. A common heuristic is to use the size of the D' sets to determine the next variable. An example is given in the SELECTVARIABLE subprocedure in Figure 13. This routine is particularly simple in that it relies purely on the size of the smallest domain, and breaks ties arbitrarily. More sophisticated DVO schemes have been proposed [37,36,74]. BACKTRACKING-

WITH-LOOKAHEAD can be modified to employ dynamic variable ordering by calling SELECTVARIABLE after the initialization step "$i \leftarrow 1$" and after the forward step "$i \leftarrow i+1$." The FC-CBJ algorithm discussed below illustrates how SELECTVARIABLE should be invoked.

Example 16. Consider again the example in Figure 3. Initially, all variables have domain size of two or three. Suppose the DVO scheme selects x_2 with a domain size of two, and the assigment $x_2 = green$ is made. Forward checking propagation of this choice will remove *green* from the domains of x_1 and x_6. Now all variables have domains of size two; suppose x_7 is selected and assigned *blue*. The impact is to restrict the domains of x_1, x_3, x_4, and x_5 to single values. Now DVO selects x_3, and the only possible value, *red*, is considered. Propagating this assignment to x_1 results in an empty domain, and so $x_3 = red$ is rejected, and the algorithm backtracks to x_7.

The information gleaned during the look-ahead phase can also be used to guide value selection [20,72,35,29]. 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 [29] 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 tentative instantiation and its effects are retracted before the next instantiation is made.) LVO then uses a heuristic function 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 [29].

7 Integration and comparison of algorithms

7.1 Integrating backjumping and look-ahead

Complementary enhancements to backtracking can be integrated into a single procedure. The look-ahead strategies discussed above can be combined with any of the backjumping variants described in Section 4. Additionally, a combined algorithm can employ learning, and the dynamic variable and value ordering heuristics based look-ahead information. One combination is conflict-directed backjumping with forward checking level look-ahead and dynamic variable ordering [27,31]. We present such an integrated algorithm, FC-CBJ, in Figures 14 and 15.

The main procedure of FC-CBJ closely resembles CONFLICT-DIRECTED-BACKJUMPING (Figure 8). Recall that CBJ maintains a jumpback set J for each variable x. SELECTVALUE-CBJ adds earlier, instantiated variables to J_i. Upon reaching a dead-end at x_i, the algorithm jumps back to the latest variable in J_i. When CBJ is combined with look-ahead, the J sets are used in the same way, but they are built in a different manner. While selecting a value for x_i, SELECTVALUE-FC-CBJ puts x_i (and possibly other variables that precede x_i, when non-binary constraints are present) into the J sets of future, uninstantiated variables that have a value in conflict with the value assigned to x_i. On reaching a dead-end at a variable x_j that follows x_i, x_i will be in J_j if x_i, as instantiated, was identified as partially responsible for the dead-end.

FC-CBJ is derived from CONFLICT-DIRECTED-BACKJUMPING by making two modifications. The first is that the D' sets are initialized and reset after a dead-end in the same manner as in BACKTRACKING-WITH-LOOKAHEAD (Figure 10). SELECTVALUE-FC-CBJ is based on look-ahead and relies on the D' being accurate and current. The second modification is the call to SELECTVARIABLE (Figure 13) in the initialization phase and during each step forward. These calls could be removed and the algorithm would revert without other modification to a static variable ordering. But given that look-ahead is being performed for the purposes of rejecting inconsistent values, there is little additional cost in performing SELECTVARIABLE, and in practice this heuristic has been found very effective in reducing the size of the search space.

7.2 Comparison of algorithms

Faced with a variety of backtracking based algorithms and associated heuristics, it is natural to ask which ones are superior in performance. Performance can be assessed by theoretical analysis of worst or average case behavior, or experimentally using benchmark instances or suites of problems, possibly randomly generated. Several criteria are frequently measured: CPU time, size of generated search tree, calls to a common subroutine, such as CONSISTENT.

Figure 16 shows the relationships between several algorithms discussed in this paper, based on worst case performance measured by the size of the search space (which is equivalent to the number of calls to a SELECTVALUE subprocedure).

Figure 17 summarizes the results of experimental comparisons of several backtracking based constraint algorithms. All algorithms here incorporate for-

```
procedure FC-CBJ
Input: A constraint network P = (X, D, C).
Output: Either a solution, or a decision that the network is inconsistent.
   i ← 1                              (initialize variable counter)
   call selectVariable                (determine first variable)
   D'_i ← D_i for 1 ≤ i ≤ n           (copy all domains)
   J_i ← ∅                            (initialize conflict set)
   while 1 ≤ i ≤ n
      instantiate x_i ← selectValue-fc-cbj
      if x_i is null                  (no value was returned)
         iprev ← i
         i ← highest index in J_i     (backjump)
         J_i ← J_i ∪ J_iprev − {x_i}
         reset each D'_k, k > i, to its value before x_i was last instantiated
      else
         i ← i + 1                    (step forward)
         call selectVariable          (determine next variable)
         D'_i ← D_i
         J_i ← ∅
   end while
   if i = 0
      return "inconsistent"
   else
      return instantiated values of {x_1, ..., x_n}
end procedure
```

Fig. 14. The main procedure of the FC-CBJ algorithm.

ward checking level look-ahead and a dynamic variable ordering scheme similar to that described in Figure 13. The names are abbreviated in the table: "FC" refers to forward checking; "FC+AC" refers to forward checking with arc-consistency enforced after each instantiation; "FC-CBJ" refers to conflict-directed backjumping with forward checking; "FC-CBJ+LVO" adds a value ordering heuristic; "FC-CBJ+LRN" is FC-CBJ plus 4th-order jumpback learning; "FC-CBJ+LRN+LVO" is FC-CBJ with both LVO and learning. The columns labeled "Set 1" through "Set 3" report averages from 2000 randomly generated binary CSP instances. All instances had variables with three element value domains, and the number of constraints was selected to generate approximately 50% solvable problems. The number of variables and the number of valid relations per constraint was: Set 1: 200 and 8; Set 2: 300 and 7; Set 3: 350 and 6. Algorithm BT was not run on sets 1 and 2. The rightmost two columns of the figure show results on two specific problems from the Second Dimacs Implementation Challenge [43]: "ssa7552-038" and "ssa7552-158." For more details about the experiments, see [31].

```
subprocedure SELECTVALUE-FC-CBJ
  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
      for all values b in D'_k
        if not CONSISTENT(a⃗_{i-1}, x_i = a, x_k = b)
          let R_S be the earliest constraint causing the conflict
          add the variables in R_S's scope S, but not x_k, to J_k
          remove b from D'_k
      end for
      if D'_k is empty          (x_i = a leads to a dead-end)
        empty-domain ← true
    end for
    if empty-domain             (don't select a)
      reset each D'_k, i < k ≤ n, to status before a was selected
      reset each J_k, i < k ≤ n, to status before a was selected
    else
      return a
  end while
  return null                   (no consistent value)
end subprocedure
```

Fig. 15. The SelectValue subprocedure for FC-CBJ.

Interleaving an arc-consistency procedure with search was generally quite effective in these studies, as was both learning and value ordering. An interesting observation can be made based on the nature of the constraints in each of the three sets of random problems. In Set 1, each constraint between two variables with domains of size three permitted eight of the nine combinations; in Set 2, seven of nine; in Set 3, six of nine. The problems with more restrictive, or "tighter," constraints, had sparser constraint graphs. With the less tight constraints, the difference in performance among the algorithms was much less than on problems with tighter constraints. Enforcing arc-consistency and learning new constraints were much more effective on the sparser graphs with tight constraints. These procedures are able to exploit the local structure in such problems.

The empirical results shown in Figure 17 are only examples of typical experimental comparisons of algorithms. Unfortunately, from these and similar studies it is not necessarily possible to conclude how the algorithms will perform on all problems having different structural properties.

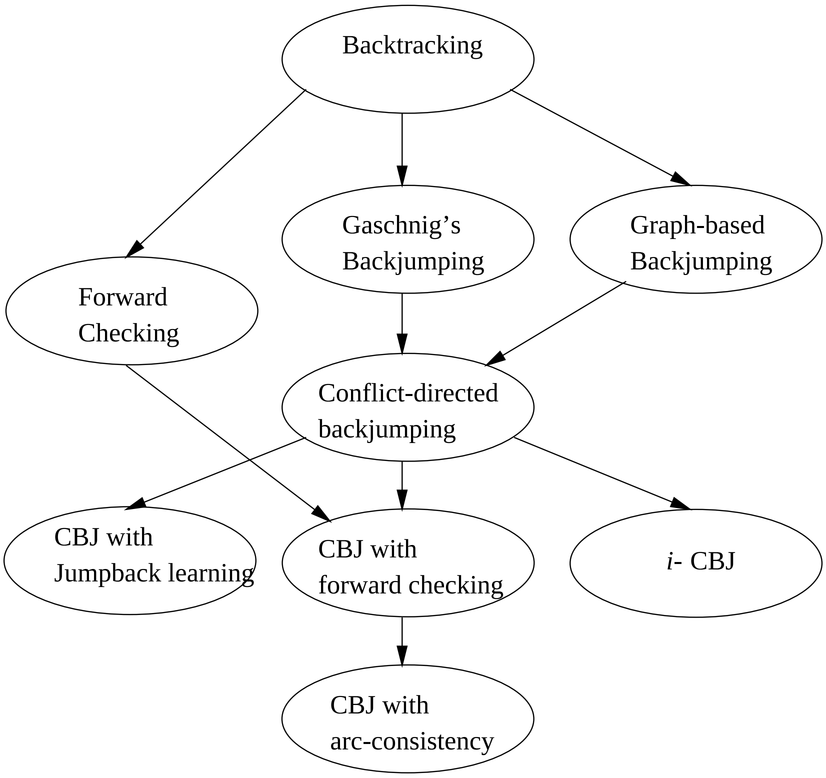


Fig. 16. The relationships of selected backtracking based algorithms. CBJ is an abbreviation for conflict-directed backjumping. An arrow from A to B indicates that on the same problem and with the same variable and value orderings, the number of nodes in A's search tree will be greater than or equal to the number of nodes in B's.

Algorithm	Set 1		Set 2		Set 3		ssa 038		ssa 158	
FC	207	68.5		-		-	46	14.5	52	20.0
FC+AC	40	55.4	1	0.6	1	0.4	4	3.5	18	8.2
FC-CBJ	189	69.2	222	119.3	182	140.8	40	12.2	26	10.7
FC-CBJ+LVO	167	73.8	132	86.8	119	111.8	32	11.0	8	4.5
FC-CBJ+LRN	186	63.4	32	15.6	1	0.5	23	5.5	19	8.6
FC-CBJ+LRN+LVO	160	74.0	26	14.0	1	3.8	16	3.8	13	7.1

Fig. 17. Empirical comparison of six selected CSP algorithms. See text for explanation. In each column of numbers, the first number indicates the number of nodes in the search tree, rounded to the nearest thousand, and final 000 omitted; the second number is CPU seconds.

8 Historical Remarks

Most current work on improving backtracking algorithms for solving constraint satisfaction problems use Bitner and Reingold's formulation of the algorithm [10]. 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 [80]. 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 [59,48,24]. However, Golomb and Baumert [39] may have been the first to informally describe this idea. Following Waltz's work and Montanari's seminal work on constraint networks [59], Mackworth detailed several specific algorithms for node-, arc-, and path-consistency to be be possibly augmented, either before backtracking search or interleaved with backtracing search [48]. Consistency techniques are used in Lauriere's Alice system [47]. Explicit algorithms employing this idea have been given by Gaschnig [34], who described a backtracking algorithm that incorporates arc-consistency; McGregor [54], who described backtracking combined with forward checking, which is a truncated form of arc-consistency; Haralick and Elliott [41], who also added various look-ahead methods; and Nadel [60], who discussed backtracking combined with many variations of partial arc-consistency. Gaschnig [33] has compared Waltz-style look-ahead backtracking with look-back improvements that he introduced, such as backjumping and backmarking. Haralick and Elliot [41] have done a relatively comprehensive study, for the late 1970s, using randomly generated instances with up to 17 variables and n-queens problems with n ranging up to 10. They compared the performances of various look-ahead and look-back methods. 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 the mid-1990s, when larger and more difficult problem classes were tested [71,29,30]. In these studies, forward checking lost its superiority on many problem instances to full looking ahead and other stronger looking-ahead variants. Empirical evaluation of backtracking with dynamic variable ordering on the n-queens problem was reported in [76].

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 [11]. The truth-maintenance systems area also has contributed to improving backtracking. Stallman and Sussman [75] were the first to mention nogood recording, and their idea gave rise to look-back type algorithms, called *dependency-directed backtracking* algorithms, that include both backjumping and nogood recording

[53].

In the context of solving propositional satisfiability, Logemann, Davis and Loveland [16] introduced a backtracking algorithm (DLL) that uses look-ahead 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 [69] and for constraint satisfaction [41,63,61]. The value of look-back improvements for solving propositional satisfiability was initially largely overlooked, as most researchers focused on look-ahead improvements of DLL [15]. This was changed significantly with the work by Bayardo and Schrag in 1997 [7]. They showed that their algorithm *relsat*, which incorporates both learning and backjumping, outperforms many of the best backtracking-based SAT solvers available at the time, on hard benchmarks. Subsequently several smart implementations of look-back based SAT solvers (e.g., Grasp [51]) were developed.

Freuder [25] and Dechter and Pearl [20,17] 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 [20]. Dechter [17] also described the graph-based variant of backjumping, which was followed by conflict-directed backjumping, introduced by Prosser [68]. Other graph-based methods include graph-based learning (i.e., constraint recording) as well as the cycle-cutset scheme [17]. The complexity of these methods is bounded by graph parameters: Dechter and Pearl [20] developed the induced width bound on learning algorithms and Dechter [17] showed that the cycle-cutset size, bounds some look-ahead methods. Frueder and Quinn [26] noted the dependence of backjumping's performance on the depth of the DFS tree of the constraint graph, and Bayardo and Mirankar [6] improved the complexity bound.

Subsequently, as it became clear that many of backtracking's improvements are largely 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 [17] 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 [68], who combined known look-ahead and look-back methods and ranked each combination based on average performance on, primarily, Zebra problems. Dechter and Meiri [19] have evaluated the effect of pre-processing consistency algorithms on backtracking and backjumping.

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 [12] 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 [28,27,29,38,15,5,3]. 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 [2], 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 [44].

The idea of non-systematic complete backtracking was introduced by Makoto Yokoo who was the first to observe that the use of learning in the context of a distributed version of search maintains completeness [81]. This idea caught up recently in the community of SAT-solver developers as well. Many of the current solution methods combine different algorithms or exploit nondeterminizm in the randomized version of backtracking search using either random restarts or randomizing backtrack points [40,66].

Constraint processing techniques have been augmented into *Constraint Logic Programming (CLP)* languages. The inference engines of these languages use a constraint solver as well as the traditional logic programming inference procedures. In addition to employing general constraint techniques such as enforcing arc-consistency, these languages gain efficiency by using a collection of specialized constraint propagation algorithms for frequently used constraints, such as "all-different" [78,42,52].

Acknowledgements

Thanks to Peter van Beek for helpful comments, particularly his useful suggestions on the section covering historical and other perspectives; to Roberto Bayardo and Javier Larrosa for perceptive comments; to Irina Rish for some of the figures; and lastly, to Michelle Bonnice for her dedicated editing of an earlier version of the paper.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] R. Bayardo, Jr. and R. C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *AAAI-97: Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 1997.

[8] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[9] C. Bessière, E. C. Freuder, and J.-C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

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

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

[12] 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.

[13] Xinguang Chen and Peter van Beek. Conflict-Directed Backjumping Revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.

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

[15] 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.

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

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

[18] R. Dechter. Constraint networks. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence, 2nd Edition*, pages 276–285. John Wiley & Sons, 1992.

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

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

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

[22] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, pages 283–308, 1997.

[23] S. Even. Graph algorithms. In *Computer Science Press*, 1979.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] D. H. Frost. *Algorithms and Heuristics for Constraint Satisfaction Problems.* PhD thesis, Information and Computer Science, University of California, Irvine, CA, 1997.

[32] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[33] 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.

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

[35] 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.

[36] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. In *Principles and Practice of Constraint Programming - CP95*, pages 179–193, 1996.

[37] I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *AAAI-96: Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, 1996.

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

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

[40] C. P. Gomez, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998.

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

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

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

[44] 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.

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

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

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

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

[49] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence, 2nd Edition*, pages 285–293. John Wiley & Sons, 1992.

[50] 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.

[51] J. P. Marques-Silva and K. A. Sakalla. Grasp-a search algorithm for propositional satisfiability. *IEEE Transaction on Computers*, pages 506–521, 1999.

[52] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: an Introduction.* MIT Press, 1998.

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

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

[55] I. Miguel and Q. Shen. Solution Techniques for Constraint Satisfaction Problems: Advanced Approaches. *Artificial Intelligence Review*, 15(4):269–293, 2001.

[56] I. Miguel and Q. Shen. Solution Techniques for Constraint Satisfaction Problems: Foundations. *Artificial Intelligence Review*, 15(4):243–267, 2001.

[57] 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.

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

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

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

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

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

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

[64] J. Pearl. *Heuristics: Intelligent Search Strategies.* Addison-Wesley, 1984.

[65] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

[66] S. Prestwich. A hybrid search architecture applied to hard random 3-sat and low autocorrelation binary sequences. *Principles and Practice of Constraint Programming (CP2000)*, pages 337–352, 2000.

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

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

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

[70] Francesca Rossi. Constraint (Logic) Programming: A Survey on Research and Applications. In K. R. Apt, A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints*, pages 40–74. Springer, 1999.

[71] 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.

[72] 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.

[73] 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.

[74] Barbara Smith and Stuart A. Grant. Trying Harder to Fail First. In *Proceedings of the European Conference on AI (ECAI-98)*, pages 249–253, 1998.

[75] 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.

[76] 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.

[77] E. Tsang. *Foundation of Constraint Satisfaction.* Academic Press, 1993.

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

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

[80] 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.

[81] M. Yokoo. Asynchronous weak commitment search for solving distributed constraint satisfaction problems. In *First International Conference on Constraint Programming*, France, 1995.