Chapter 5

General search strategies: Look-ahead

No matter how much we *reason* about a problem, after some consideration we are left with choices, and the only way to proceed is *trial and error* or *guessing and testing* (if the guess is consistent with previous choices and how it affects future choices.) That is, we must search the space of possible choices. So, if we have 12 guests for dinner, we may have already determined out the seats of the host and hostess, and that Jill, who will help with serving, must sit next to the hostess. And we know that then, Jill's ex-husband must not sit next to her nor face her. So who will sit next to Jill? We have 8 possible guests, lets choose one arbitrarily and see how it plays out with the rest of the constraints.

Indeed, most CSP algorithms employ either search or deduction principles; more sophisticated algorithms often combine both. The term *search* characterizes a large category of algorithms which solve problems by guessing the next operation to perform, sometimes with the aid of a heuristic. A good guess results in a new state that is nearer to the goal. If the operation does not result in progress towards the goal (this may not be apparent until later in the search), then the step can be retracted and another operation can be tried.

For CSPs, search is epitomized by the backtracking algorithm. Backtracking search for CSPs extends a current partial solution by assigning values to variables. Starting with the first variable, the algorithm assigns a provisional value to each variable in turn, making sure that each assigned value is consistent with values assigned thus far before proceeding to the next variable. When the algorithm encounters a variable for which no domain value is consistent with previous assignments, a *dead-end* occurs. At this point, *backtracking* takes place; that is, the value assigned to the variable immediately preceding the dead-end variable is changed – if possible – and the search continues. The algorithm halts when either the required number of solutions has been found or when it can be concluded that no solution, or no more solutions, exist. Backtracking requires only linear space, but in the worst case it requires time exponential in the number of variables.

Much of the work in constraint satisfaction during the last decade has been devoted to improving the performance of backtracking search. The performance of backtracking can be improved by reducing the size of its *explored* search space, which is determined both by the size of the *underlying* search space and by the algorithm's control strategy. The size of the underlying search space depends on the level of local consistency possessed by the problem, and on the order of variable instantiation. Using these factors, two types of improvement procedures have emerged: those that bound the size of the underlying search space and are employed *before* performing the search, and those used dynamically *during* the search that decide which parts of the search space will not be visited. Commonly used pre-processing techniques are arc- and path-consistency algorithms, as well as heuristic approaches that determine the variable ordering.

For the remainder of this chapter, we will take a closer look at search spaces for CSPs, at backtracking in general, and at *look-ahead* schemes that improve basic backtracking. *Look-back* improvement schemes will be covered in Chapter 6.

5.1 The search space

A *state search space* is generally defined by four elements: a set S of states, a set O of operators that map states to states, an initial state $s_0 \in S$, and a set $S_g \subseteq S$ of goal states. The fundamental task is to find a solution, namely, a sequence of operators that transforms the initial state into a goal state. The search space can be effectively represented by a directed *search graph*, where each node represents a state and where a directed arc from s_i to s_j means that there is an operator transforming s_i into s_j. Terminal or leaf nodes are nodes lacking outwardly directed arcs. Goal nodes represent solutions, and non-goal terminal nodes represent dead-ends. Any search algorithm for finding a solution can thus be understood as a traversal algorithm looking for a solution path in a search graph. For details about search spaces and general search algorithms, see [220, 225].

Search algorithms for CSPs can be viewed as traversing a state space graph whose nodes (the states) are consistent partial instantiations and whose arcs represent operators that take a partial solution $(< x_1, a_1 >, \ldots, < x_j, a_j >)$, $1 \leq j \leq n$, and augment it with an instantiation of an additional variable that does not conflict with prior assignments, yielding $(< x_1, a_1 >, \ldots, < x_j, a_j >, < x_{j+1}, a_{j+1} >)$. The *initial state* is the empty instantiation, and *goal states* are full solutions. Variables are processed in some order, which may be either specified prior to search or determined dynamically during search. To demonstrate concepts of search-space we use a simple and small problem example so that the visual depiction of the search-space stays feasible.

Example 5.1.1 Consider a constraint network $\mathcal{R}$ having four variables x, y, l, z with domains $D_x = \{2, 3, 4\}$, $D_y = \{2, 3, 4\}$, $D_l = \{2, 5, 6\}$ and $D_z = \{2, 3, 5\}$. There is a constraint between z and each of the other variables. These constraints require that the value assigned to z evenly divides the values assigned to x, y, and l. The constraint graph of this problem is depicted in Figure 5.1a. State space graphs for the problem along the orderings $d_1 = (z, x, y, l)$ and $d_2 = (x, y, l, z)$ are given in Figure 5.1b and c. Filled nodes in Figure 5.1b and c denote legal states. Light colored ovals denote goal states, black ovals denote intermediary states and black boxes denote dead-ends. The hollow boxes connected by broken lines represent illegal states that correspond to failed instantiation attempts. These illegal states are sometimes depicted in the search graph because they express problem-solving activity. (In fact, some search space definitions include these leaf nodes as legal states in the search space.) □

Next, we look at two factors that contribute to the size of a search space: variable ordering and consistency level.

5.1.1 Variable ordering

As illustrated in Figure 5.1b and c, a constraint network may have different search spaces, depending on the variable ordering. The search graph for ordering d_1 includes 20 legal states, whereas the search graph for ordering d_2 includes 48 legal states. Since the search space includes all solutions, one way to assess whether its size is excessive is by counting the number of dead-end leaves. Ordering d_1, for instance, renders a search graph with only one dead-end leaf, while the search graph for ordering d_2 has 18 dead-end leaves.

When variable ordering is unrestricted, the ordering of the variables is left to the search algorithm rather than fixed in advance, and the underlying search space includes all possible orderings of variables. Because of its large size, the unordered search space may seem undesirable and appear unable to effectively bound the search. Nevertheless, as we shall see, its flexibility frequently leads to a comparatively small *explored* graph.

5.1.2 Consistency level

The second factor affecting the search space of a CSP is the level of local consistency, which shrinks the search space as we change the representation of a problem to an equivalent, but tighter set of constraints.

Example 5.1.2 Consider again the constraint network in Example 5.1.1. As given, the network is not arc-consistent. The value 5 of z, for instance, does not evenly divide any of x's values. Applying arc-consistency to the network would result in the deletion of the value 5 from the domains of z and l. Consequently, in the resulting arc-consistent

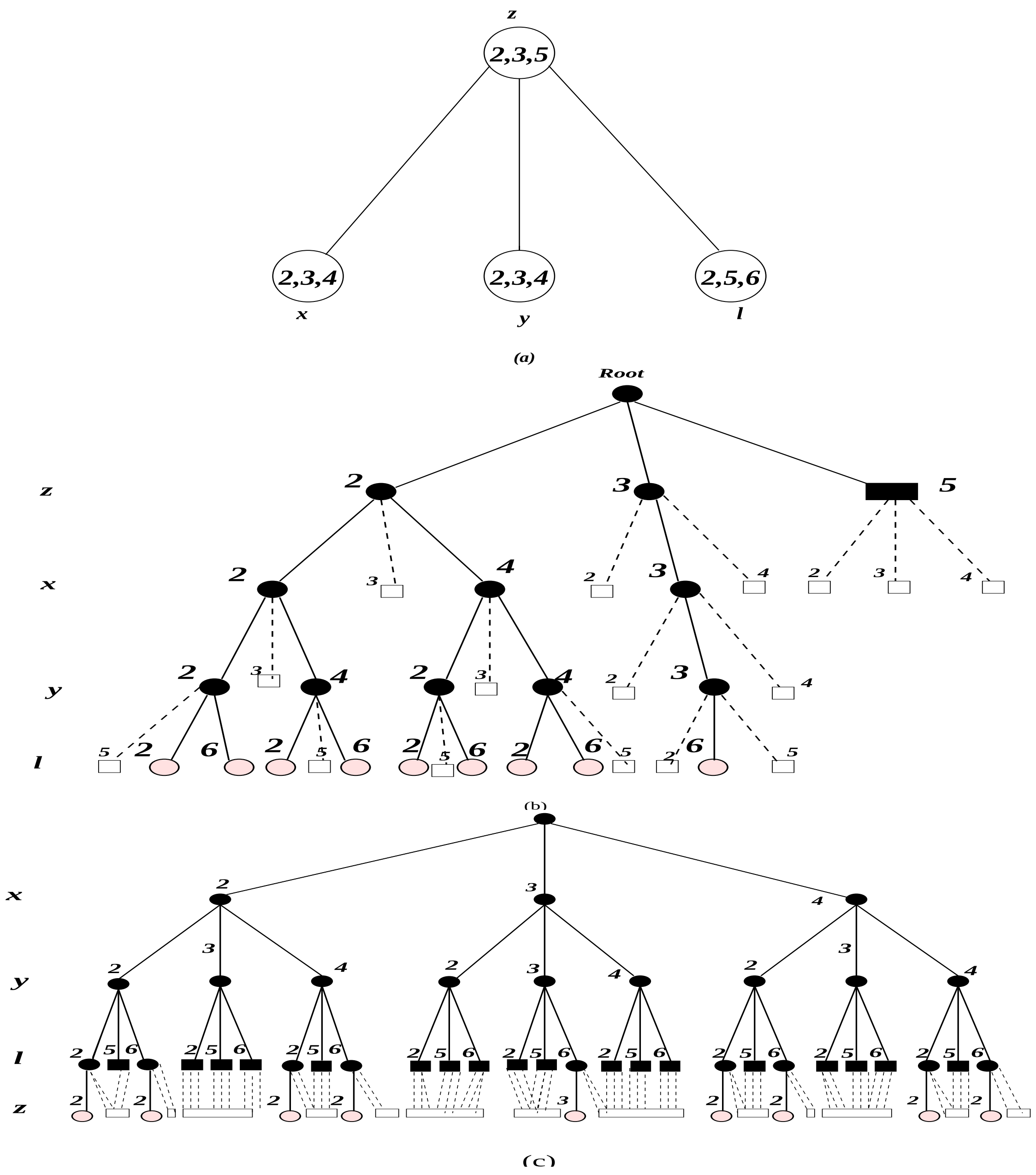


Figure 5.1: (a) A constraint graph, (b) its search space along ordering $d_1 = (z, x, y, l)$, and (c) its search space along ordering $d_2 = (x, y, l, z)$. Hollow nodes and bars in the search space graphs represent illegal states that may be considered, but will be rejected. Numbers next to the nodes represent value assignments.

network the search space along ordering d_1 will no longer include the dead-end at node $z = 5$, which emanates from the root in Figure 5.1b. The new search space is shown in Figure 5.2a. Similarly, enforcing arc-consistency would eliminate nine dead-end leaves (all corresponding to removing $< l, 5 >$) from the search space along ordering d_2. The network, as given, is also not path-consistent. For instance, the assignment $(< x, 2 >, < y, 3 >)$ is consistent, but cannot be extended to any value of z. To enforce path-consistency, we introduce a collection of binary constraints and we get a path-consistent network $\mathcal{R}'$ which includes the following set of constraints:

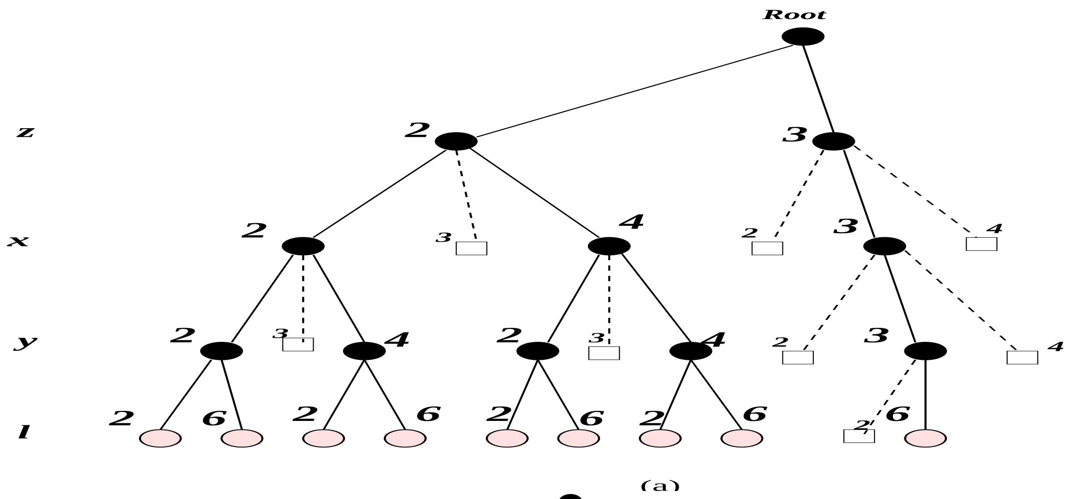
$$\begin{array}{l}
R'_{zx} = \{(2,2),(2,4),(3,3)\} \\
R'_{zy} = \{(2,2),(2,4),(3,3)\} \\
R'_{zl} = \{(2,2),(2,6),(3,6)\} \\
R'_{xy} = \{(2,2),(2,4),(4,2),(4,4),(3,3)\} \\
R'_{xl} = \{(2,2),(2,6),(4,2),(4,6),(3,6)\} \\
R'_{yl} = \{(2,2),(2,6),(4,2),(4,6),(3,6)\}.
\end{array}$$

The search graphs of $\mathcal{R}'$ and $\mathcal{R}$ along ordering d_2 are compared in Figure 5.2b. Note the amount of pruning of the search space (indicated by slash marks across the pruned branches) that results from tightening the representation. □

Theorem 5.1.3 *Let $\mathcal{R}'$ be a tighter network than $\mathcal{R}$, where both represent the same set of solutions. For any ordering d, any path appearing in the search graph derived from $\mathcal{R}'$ also appears in the search graph derived from $\mathcal{R}$.* □

The above discussion suggests that one should make the representation of a problem as explicit as possible by inference before searching for a solution. However, too much thought process in the form of local consistency algorithm is costly, and its cost may not always be offset by the smaller search space. Tighter representations normally include many more explicit constraints than looser ones, and may require many additional constraint checks (each testing if a given assignment satisfies a single constraint). If the constraints are binary, we will never have more than $O(n)$ consistency checks per state generation. However, in the general case, the number of constraints may be very large, as high as $O(n^{r-1})$ when r bounds the constraints' arity.

Example 5.1.4 To observe the tradeoff associated with preprocessing by path-consistency, lets continue Example 5.1.2. When generating the search space for $\mathcal{R}$ along ordering d_1, exactly one constraint is tested for each new node generated. In contrast, when using $\mathcal{R}'$, which has an explicit constraint for every pair of variables, each node generated at level 1 of the search tree requires one constraint check; at level 2 each node requires



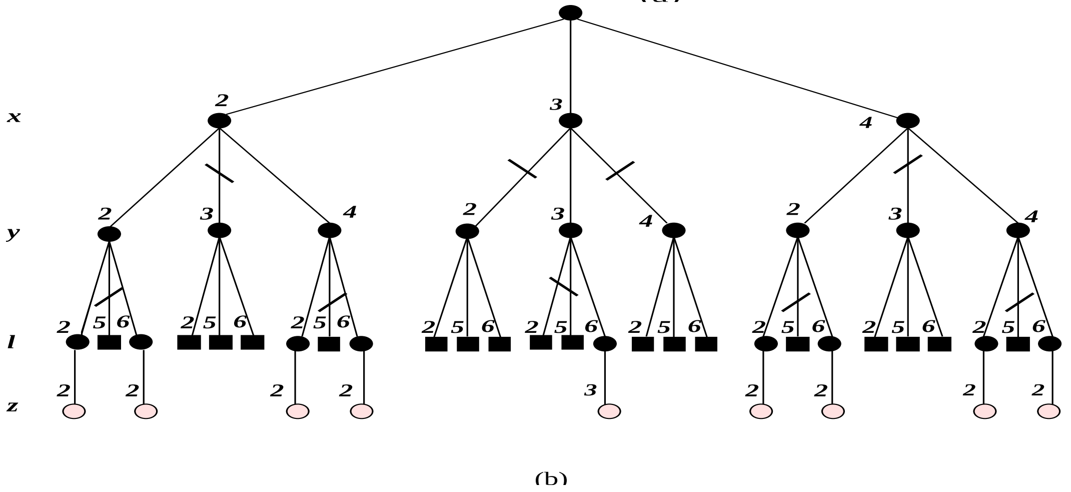


Figure 5.2: (a) Search space Example 5.1.1 with ordering d_1 after arc-consistency. (b) Search space for ordering d_2 with reduction effects from enforcing path-consistency marked with slashes.

two constraint checks; and three checks are required for each node generated at level 3. Overall, using order d_1, fewer constraint tests, around 20, are performed when generating the whole search tree for R, while many more constraint checks, around 40, are required for $\mathcal{R}'$. When generating the search graph for $\mathcal{R}$ along ordering d_2 (Figure 5.2b), on the other hand, the first three levels of the tree require no constraint checks (there are no explicit constraints between variables x, y, l), but generating the fourth level in the search tree may require as many as three constraint checks per node, yielding between 45-81 constraint tests depending on the order of constraint testing. When using $\mathcal{R}'$ along ordering d_2, constraint checks are performed in each of the first three levels in the tree (one per node in the first level, two per node in the second, three per node in the third). This search graph has only 9 nodes at level 4, each requiring three tests. Thus, for ordering d_2, enforcing path-consistency pays off: the maximum number of required tests is higher, around 80, before enforcing path-consistency, whereas afterwards, it is reduced considerably, to about 50.

□

Finally, observe that after the application of arc-consistency, the search space along ordering d_1 in Figure 5.2a contains solution paths only. This is an extremely desirable state of affairs since any depth-first search of such a space is now guaranteed to find a solution in linear time (provided that the cost of state generation is bounded). Next, we redefine this *backtrack-free* search space.

Definition 5.1.5 (backtrack-free network) *A network R is said to be* backtrack-free *along ordering d if every leaf node in the corresponding search graph is a solution.*

Let's refresh our memory on relevant notation. We denote by $\vec{a_i}$ the subtuple of consecutive values $(a_1, ..., a_i)$ for a given ordering of the variables $x_1, ..., x_i$.

5.2 Backtracking

The *backtracking* algorithm traverses the search space of partial instantiations in a depth-first manner. Backtracking has two phases. The first is a forward phase during which the variables are selected in sequence, and a current partial solution is extended by assigning a consistent value, if one exists, for the next variable. The second phase is a backward one in which, when no consistent solution exists for the current variable, the algorithm returns to the previous variable assigned. Figure 5.4 describes a basic backtracking algorithm. The BACKTRACKING procedure employs a series of mutable value domains D'_i, where D'_i holds the subset of D_i that has not yet been examined under the current partial instantiation of earlier variables.

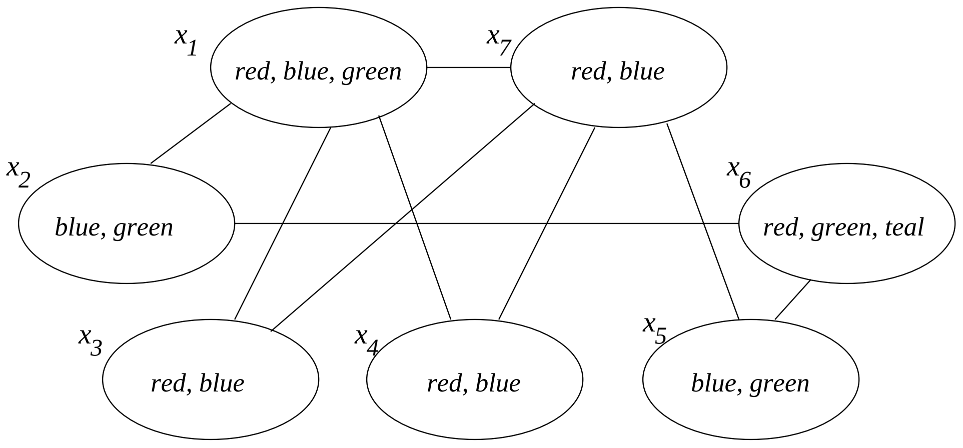


Figure 5.3: A coloring problem with variables $(x_1, x_2, \ldots, x_7)$. The domain of each variable is written inside the corresponding node. Each arc represents the constraint that the two variables it connects must be assigned different colors.

Example 5.2.1 Consider the graph-coloring problem in Figure 5.3. Assume a 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 5.5(a) and 5.5(b), respectively. Only legal states, namely partial solutions, are depicted in the figure. □

In its 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, $\vec{a}_{i-1}$. This subprocedure describes the work associated with generating the next node in the search space. SELECTVALUE, in turn, relies on the CONSISTENT subprocedure, which returns *true* only if the 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, 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 consistent assignments, or when it has proven that all values of x_1 cannot 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 be easily modified to return all or a desired number of solutions.

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.) For simplicity's sake, our examples involve binary con-

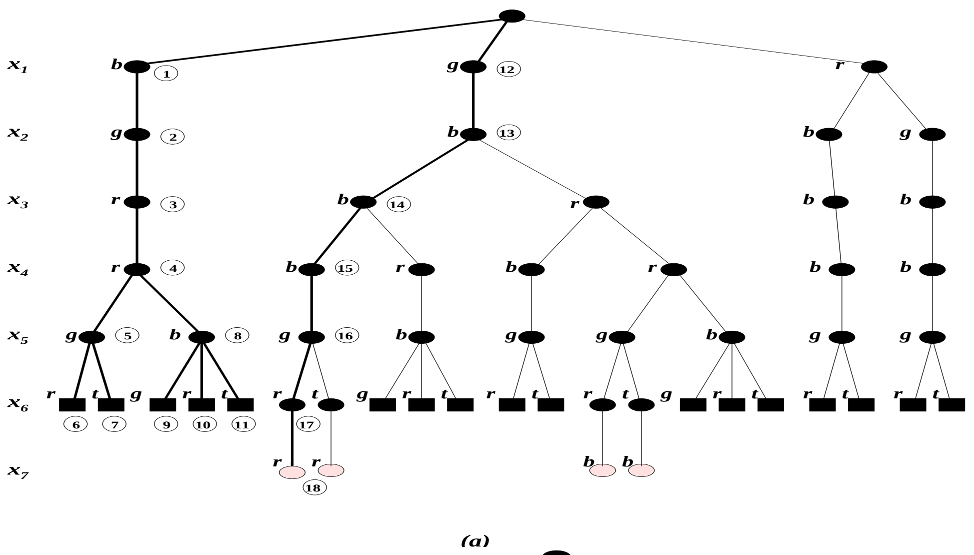
```
procedure BACKTRACKING
Input: A constraint network $\mathcal{R} = (X, D, C)$.
Output: Either a solution, or notification that the network is inconsistent.

    $i \leftarrow 1$                          (initialize variable counter)
    $D'_i \leftarrow D_i$                     (copy domain)
    while $1 \leq i \leq n$
        instantiate $x_i \leftarrow$ SELECTVALUE
        if $x_i$ is null                      (no value was returned)
            $i \leftarrow i - 1$              (backtrack)
        else
            $i \leftarrow i + 1$              (step forward)
            $D'_i \leftarrow D_i$
    end while
    if $i = 0$
        return "inconsistent"
    else
        return instantiated values of $\{x_1, \ldots, x_n\}$
end procedure

subprocedure SELECTVALUE   (return a value in $D'_i$ consistent with $\vec{a}_{i-1}$)

    while $D'_i$ is not empty
        select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$
        if CONSISTENT($\vec{a}_{i-1}, x_i = a$)
            return $a$
    end while
    return null                               (no consistent value)
end procedure
```

Figure 5.4: The backtracking algorithm.



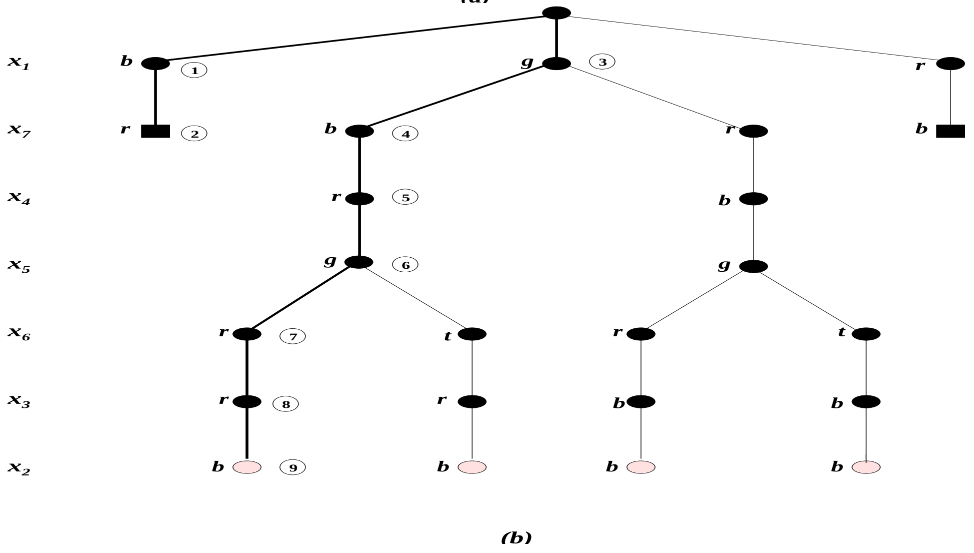


Figure 5.5: Backtracking search for the orderings (a) $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$ and (b) $d_2 = x_1, x_7, x_4, x_5, x_6, x_3, x_2$ on the example instance in Figure 5.3. Intermediate states are indicated by filled ovals, dead-ends by filled rectangles, and solutions by grey ovals. The colors are considered in order (*blue, green, red, teal*), and are denoted by first letters. Bold lines represent the portion of the search space explored by backtracking when stopping after the first solution. Circled numbers indicate the order in which nodes are expanded.

straints, but the description is completely general. CONSISTENT handles general binary and non-binary constraints; its implementation, which is not specified, depends on how constraints are represented by the computer program. Next, we analyze its complexity.

5.2.1 Complexity of extending a partial solution

We can determine the complexities of CONSISTENT and SELECTVALUE by conceptualizing the constraints as stored in tables. Let e be the number of constraints in the problem, and let t be the maximum number of tuples in a constraint. Let k be the maximum size of any domain in D. If the maximum constraint arity is r then $t \leq k^r$. The constraints can be organized to permit finding a tuple of a given constraint in worst-case logarithmic time: $\log t \leq r \log k \leq n \log k$. Since a variable may participate in up to e constraints, the worst-case time complexity of CONSISTENT is $O(e \log t)$ which is also bounded by $O(e\, r \log k)$. SELECTVALUE may invoke CONSISTENT up to k times so the worst-case time complexity of SELECTVALUE is $O(e\, k\, r \log k)$, or $O(e\, k\, \log t)$.

In general we will assume that when given a partial solution $\vec{a}_i$, CONSISTENT tests the consistency of a specific assignment to x_{i+1} by looking at the relevant constraints in an arbitrary ordering. Let e_i be the number of constraints that are defined over x_i. Clearly, the ordering of contraint testing can have an impact on performance and rejecting an inconsistent extension can cost between 1 to e_{i+1} constraint checks. Therefore, the best-case performance of SELECTVALUE is $O(k \log t)$.

For the special case of a binary CSP, the tentative instantiation $< x_i, a >$ must then be checked with at most n earlier variables, effectively yielding $O(n)$ complexity for CONSISTENT and $O(nk)$ complexity for SELECTVALUE. Of course, if CONSISTENT performs computations other than table lookups, its complexity is dependent on the nature of these computations. In summary;

Proposition 5.2.2 *For general CSPs having n variables, e constraints, a constraint arity bounded by r and number of tuples in a constraint bounded by t, and at most k domain values, the time complexity of* CONSISTENT *is $O(e \log t)$ or $O(e\, r \log k)$, and the time complexity of* SELECTVALUE *is $O(e\, k\, r \log k)$ or $O(e\, k \log t)$. For binary CSPs, the complexity of* SELECTVALUE *is $O(n\, k)$.*

5.2.2 Improvements to backtracking

Backtracking frequently suffers from *thrashing*: repeatedly rediscovering the same inconsistencies and partial successes during search. An efficient cure for thrashing in all cases is unlikely, since the problem is NP-complete. Still, backtracking performance can be improved by employing preprocessing algorithms that reduce the size of the *underlying* search space, and by dynamically improving the algorithm's control strategy during search. The procedures for dynamically improving the pruning power of backtracking during search are split into *look-ahead schemes* and *look-back schemes*, as they relate to backtracking's two main phases: going forward to assemble a solution and going back in case of a dead-end.

Look-ahead schemes can be invoked whenever the algorithm is preparing to assign a value to the next variable. These schemes attempt to discover, from a restricted amount of inference, that is, constraint propagation, how current decisions about variable and value selection will restrict future search. Once a certain amount of forward constraint propagation is complete, the algorithm can use the information to:

1. Decide which variable to instantiate next, if the order is not predetermined. It is generally advantageous to first instantiate those variables that maximally constrain the rest of the search space. Therefore, the most highly constrained variable having the least number of viable values is usually selected.

2. Decide which value to assign to the next variable. When searching for a single solution, an attempt is made to assign the value that maximizes the number of options available for future assignments.

Look-back schemes are invoked when the algorithm prepare to backtrack after encountering a dead-end. These schemes perform two functions:

1. Decide how deep to backtrack. By analyzing the reasons for the dead-end, irrelevant backtrack points can often be avoided so that the algorithm goes directly back to the source of failure. This procedure is often referred to as *backjumping.*

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

We return to look-back schemes in Chapter 6. The remainder of this chapter focuses on look-ahead algorithms. But before we proceed, let us conclude this section by looking at one of the earliest procedures for improving the state generation cost, *backmarking*, which does not quite fit as either a look-ahead or look-back scheme.

Backmarking

By keeping track of where past consistency tests failed, *backmarking* can eliminate the need to repeat previously performed checks.

```
procedure BACKMARKING
Input: A constraint network $\mathcal{R} = (X, D, C)$,
Output: A solution or conclusion that the network is inconsistent.
    $M_{i,v} \leftarrow 0, low_i \leftarrow 0$ for all $i$ and $v$      (initialize tables)
    $i \leftarrow 1$                           (initialize variable counter)
    $D'_i \leftarrow D_i$                          (copy domain)
    while $1 \leq i \leq n$
        instantiate $x_i \leftarrow$ SELECTVALUE-BACKMARKING
        if $x_i$ is null                   (no value was returned)
            for all $j, i < j \leq n$,     (update low of future variables)
                if $i < low_j$
                    $low_j \leftarrow i$
            $low_i \leftarrow i - 1$
            $i \leftarrow i - 1$               (backtrack)
        else
            $i \leftarrow i + 1$               (step forward)
            $D'_i \leftarrow D_i$
    end while
    if $i = 0$
        return "inconsistent"
    else
        return instantiated values of $\{x_1, \ldots, x_n\}$
end procedure

procedure SELECTVALUE-BACKMARKING
    remove from $D'_i$ all $a_v$ for which $M_{i,v} < low_i$
    while $D'_i$ is not empty
        select an arbitrary element $a_v \in D'_i$, and remove $a_v$ from $D'_i$
        $consistent \leftarrow true$
        $k \leftarrow low_i$
        while $k < i$ and $consistent$
        if not CONSISTENT($\vec{a_k}, x_i = a_v$)
            $M_{i,v} \leftarrow k$
            $consistent \leftarrow false$
        else
            $k \leftarrow k + 1$
        end while
        if $consistent$
            $M_{i,v} \leftarrow i$
            return $a_v$
    end while
    return null                    (no consistent value)
end procedure
```

Figure 5.6: The backmarking algorithm.

Recall that a backtracking algorithm moves either forward or backward in the search space. Suppose that the current variable is x_i and that x_p is the earliest variable in the ordering whose value has changed since the last visit to x_i. Clearly, any testing of constraints whose scope includes x_i and those variables preceding x_p will produce the same results as in the earlier visit to x_i. If it failed against earlier instantiations, it will fail again; if it succeeded earlier, it will succeed again. By maintaining the right information from earlier parts of the search, therefore, values in the domain of x_i can either be immediately recognized as inconsistent, or else the only constraints to be tested are those involving instantiations starting from x_p.

Backmarking acts on this insight by maintaining two new tables. First, for each variable x_i and for each of its values a_v, backmarking remembers the earliest prior variable x_p such that the current partial instantiation $\vec{a}_p$ conflicted with $x_i = a_v$. This information is maintained in a table with elements $M_{i,v}$. (Note: this approach assumes that constraints involving earlier variables are tested before those involving later variables.) If $x_i = a_v$ is consistent with all earlier partial instantiations $\vec{a}_j$, $j < i$, then $M_{i,v} = i$. The second table, containing elements low_i, records the earliest variable that changed value since the last time x_i was instantiated. This information is put to use at every step of node generation. If $M_{i,v}$ is less than low_i, then the algorithm knows that the variable pointed to by $M_{i,v}$ did not change and that $x_i = a_v$ will fail again when checked against $\vec{a}_{M_{i,v}}$, so no further consistency checking is needed for rejecting $x_i = a_v$. If $M_{i,v}$ is greater than or equal to low_i, then $x_i = a_v$ is consistent with $\vec{a}_{low_i}$, and the relevant consistency checks can be skipped. The algorithm is presented in Figure 5.6.

Example 5.2.3 Consider again the example in Figure 5.3, and assume that backmarking uses ordering $d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$. Once the algorithm encounters the first dead-end at variable x_7 with the assignment $(< x_1, blue >, < x_2, green >, < x_3, red >, < x_4, red >, < x_5, green > \;, < x_6, red >)$ (see the search space in Figure 5.5a), table M has the following values: $M(1, blue) = 1, M(2, green) = 2, M(3, blue) = 1, M(3, red) = 3$ $M(4, blue) = 1, M(4, red) = 4, M(5, blue) = 5, M(6, green) = 2, M(6, red) = 6, M(7, red) = 3, M(7, blue) = 1$. Upon backtracking from x_7 to x_6, $low(7) = 6$ and $x_6 = teal$ is assigned. Then, when trying to instantiate a new value for x_7, the algorithm notices that the M values are smaller than $low(7)$ for both values of x_7 (red, blue) and, consequently, both values are determined inconsistent without performing any other consistency checks. □

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

5.3 Look-ahead strategies

Look-ahead schemes are invoked whenever the algorithm is preparing to select the next variable or the next value. These schemes seek to discover, from a restricted amount of consistency-enforcing, how current decisions about variable and value selection will affect future search. Specifically, the decision to accept or reject a value of the current variable is based on the impact that assignment has when constraint propagation is applied to the uninstantiated "future" variables. Once a certain amount of reasoning via constraint propagation is completed, the algorithm can use the results to decide which *variable* to instantiate next, if the order is not predetermined, and which *value* to assign to the next variable.

Example 5.3.1 Consider again the coloring problem in Figure 5.3. Assume that variable x_1 is first in the ordering and that it is assigned the value *red*. A look-ahead procedure notes that the value *red* in the domains of x_3, x_4 and x_7 is incompatible with the current partial instantiation $< x_1, red >$ and thus provisionally removes those values. A more extensive look-ahead procedure may then note that x_3 and x_7 are connected and are thereby left with incompatible values; each variable has the domain $\{blue\}$ and the problem with $x_1 = red$ is therefore not arc-consistent. The scheme concludes that assigning *red* to x_1 will inevitably lead to a dead-end, cannot be part of a solution and should be rejected. The same is true for $x_1 = blue$. So, for this level of constraint propagation, both partial solutions $< x_1 = red >$ or $< x_1 = blue >$ will be determined as dead-ends and the corresponding branches below them will be pruned in Figure 5.5. □

Look-ahead strategies incur an extra cost for each instantiation request, but they can provide important benefits. For example, if *all* values of an uninstantiated variable are removed by looking-ahead, then the current instantiation cannot be part of a solution and the algorithm knows to backtrack. Consequently, dead-ends occur earlier in the search and much smaller portions of the search space need be explored. In general, the stronger the level of constraint propagation used for look-ahead, the smaller the search space explored and the higher the computational overhead.

Finally, by removing from the domains of each future variable all values inconsistent with the current partial instantiation via constraint propagation, we eliminate the need to test the consistency of values of the current variable with previous variables.

Look-ahead strategies rarely yield better worst-case performance guarantees over naive backtracking. The key challenge is to strike a cost-effective balance between look-ahead's effectiveness and its overhead.

Algorithm GENERALIZED-LOOKAHEAD in Figure 5.7 presents a framework for look-ahead algorithms that can be specialized based on the level of constraint propagation, as

```
procedure GENERALIZED-LOOKAHEAD
Input: A constraint network $\mathcal{R} = (X, D, C)$
Output: Either a solution, or notification that the network is inconsistent.

   $D'_i \leftarrow D_i$ for $1 \leq i \leq n$       (copy all domains)
   $i \leftarrow 1$                         (initialize variable counter)
   while $1 \leq i \leq n$
      instantiate $x_i \leftarrow$ SELECTVALUE-XXX
      if $x_i$ is null                   (no value was returned)
         $i \leftarrow i - 1$                  (backtrack)
         reset each $D'_k, k > i$, to its value before $x_i$ was last instantiated
      else
         $i \leftarrow i + 1$                  (step forward)
   end while
   if $i = 0$
      return "inconsistent"
   else
      return instantiated values of $\{x_1, \ldots, x_n\}$
end procedure
```

Figure 5.7: 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 interweaves arc-consistency and search.

```
procedure SELECTVALUE-FORWARD-CHECKING
    while $D'_i$ is not empty
        select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$
        empty-domain ← false
        for all $k$, $i < k \leq n$
            for all values $b$ in $D'_k$
                if not CONSISTENT($\vec{a}_{i-1}, x_i = a, x_k = b$)
                    remove $b$ from $D'_k$
            end for
            if $D'_k$ is empty          ($x_i = a$ leads to a dead-end)
                empty-domain ← true
        if empty-domain                 (don't select $a$)
            reset each $D'_k, i < k \leq n$ to value before $a$ was selected
        else
            return $a$
    end while
    return null                         (no consistent value)
end procedure
```

Figure 5.8: The SELECTVALUE subprocedure for the forward checking algorithm.

expressed in the specific SELECTVALUE subprocedure employed. GENERALIZED-LOOKAHEAD initially sets all the tentative domains (the D' sets) to equal the original domains (the D sets), and the SELECTVALUE subprocedure propagates the current instantiation to remove values from the D' sets. At this stage, the algorithm may also record inferred constraints on future variables, though, to bound the overhead of constraint propagation, most look-ahead methods modify future domains only. Upon backtracking, GENERALIZED-LOOKAHEAD resets D' sets in order to rescind modifications that were contingent on partial instantiations no longer applicable. As we see in our upcoming implementation discussion, n copies of each D' set – one for each level in the search tree – are usually maintained in order to permit the reset action to be performed efficiently.

We will define four levels of look-ahead that are based on arc-consistency. We will focus first on their use for value selection strategies.

5.3.1 Look-ahead algorithms for value selection

Forward-Checking

The first of our four look-ahead algorithms, *Forward-checking*, produces the most limited form of constraint propagation during search. It propagates the effect of a tentative value selection to each future variable, *separately*. If the domain of one of these future variables becomes empty, the value under consideration is not selected and the next candidate value is tried. Forward-checking uses the SELECTVALUE-FORWARD-CHECKING subprocedure, presented in Figure 5.8. Specifically, if variables x_1 through x_{i-1} have been instantiated, and if x_i is the current variable to be assigned, then for the tentative partial solution $\vec{a}_i = (\vec{a}_{i-1}, x_i = a_i)$, $n - i$ subproblems can be created by combining $\vec{a}_i$ with one uninstantiated variable x_u. The only constraints of interest in each subproblem are those whose scope includes x_i, x_u and a subset of $\{x_1, \ldots, x_{i-1}\}$. Enforcing consistency on a subproblem is achieved by removing from x_u's domain any value that conflict with $\vec{a}_i$ for some relevant constraint. Forward-checking treats these $n-i$ subproblems independently of each other, removing values from the domains of future variables namely, from the D' sets – as necessary. If the domain of one of the future variables x_u becomes empty, then the partial instantiation $\vec{a}_i$ cannot be extended consistently to x_u, and therefore $< x_i, a_i >$ is rejected and another value for x_i is considered. Notice that forward-checking is by no means restricted to binary constraints. Given a partial solution, all constraints having at most 2 variables that are not assigned, (one of which is the current variable) can affect this look-ahead.

Complexity of SelectValue-forward-checking. If the complexity of a constraint check is $O(1)$, then SELECTVALUE-FORWARD-CHECKING's complexity is $O(ek^2)$, where k is the cardinality of the largest domain and e is the number of constraints. How do we arrive at this figure? For a given tentative value of x_i the procedure tests the consistency of $\vec{a}_i$ against each of the k values of each future variable x_u, each involving e_u constraints, yielding $O(e_u k)$ tests. Summing over all future variables yields $O(ek)$ tests because $e = \sum_u e_u$. Finally, the procedure cycles through all the values of x_i, resulting in $O(ek^2)$ constraint checks.

Example 5.3.2 Consider again the coloring problem in Figure 5.3. In this problem, instantiating $x_1 = red$ reduces the domains of x_3, x_4 and x_7. Instantiating $x_2 = blue$ does not affect any future variable. The domain of x_3 includes only *blue* , and selecting that value causes the domain of x_7 to be empty, so $x_3 = blue$ is rejected and x_3 is determined to be a dead-end. See Figure 5.9. □

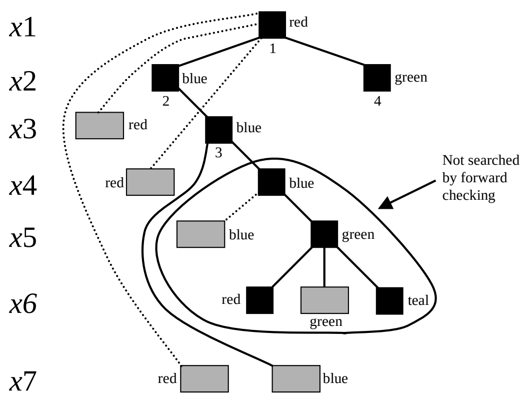


Figure 5.9: Part of the search space explored by forward-checking in the example in Figure 5.3. Only the search space below $x_1 = red$ and $x_2 = blue$ is drawn. Dotted lines connect values with future values that are filtered out.

Arc-consistency look-ahead

Arc-consistency look-ahead algorithms enforce full arc-consistency on all uninstantiated variables following each tentative value assignment to the current variable. Clearly, this look-ahead method does more work at each instantiation than does forward-checking. If a variable's domain becomes empty during the process of enforcing arc-consistency, then the current candidate value is rejected. SELECTVALUE-ARC-CONSISTENCY in Figure 5.10 implements this approach. The **repeat** ... **until** loop in the subprocedure is essentially the arc-consistency algorithm AC-1 when some of the variables are instantiated. More efficient arc-consistency procedures can also be used within a SELECTVALUE subprocedure.

Complexity of SelectValue-arc-consistency. The general optimal time complexity for any arc-consistency procedure is $O(ek^2)$, where e is the number of constraints in the subproblem, and k is the cardinality of the largest domain. Since arc-consistency may need to be applied for each value of the current variable, SELECTVALUE-ARC-CONSISTENCY yields worst-case bound of $O(ek^3)$ checks, if an optimal arc-consistency algorithm is used. As presented, however, SelectValue-arc-consistency has worse complexity bounds since it uses the AC-1 algorithm. Its analysis is left as an exercise.

Algorithm arc-consistency look-ahead is sometimes called *real full-look-ahead* to be contrasted with the full and partial look-ahead methods presented ahead. A popular variant of full arc-consistency, is *Maintaining arc-consistency (MAC)* which performs full arc-consistency after each domain value is rejected. In other words, assume that we select a value for the current variable x which has 4 values $\{1, 2, 3, 4\}$. First, tentatively assign $x = 1$ and apply full arc-consistency. If this choice causes an empty domain, $x = 1$ is rejected, and the domain of x reduces to $\{2, 3, 4\}$. Now apply full arc-consistency again

```
subprocedure SELECTVALUE-ARC-CONSISTENCY

    while $D'_i$ is not empty
        select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$
        repeat
        removed-value ← false
            for all $j, i < j \leq n$
                for all $k, i < k \leq n$
                    for each value $b$ in $D'_j$
                        if there is no value $c \in D'_k$ such that
                                CONSISTENT($\vec{a}_{i-1}, x_i = a, x_j = b, x_k = c$)
                            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 $D'_j, i < j \leq n$, to value before $a$ was selected
        else
            return $a$
    end while
    return null                    (no consistent value)
end procedure
```

Figure 5.10: The SELECTVALUE subprocedure for arc-consistency, based on the AC-1 algorithm.

with this reduced domain. If it causes an empty domain, conclude that the problem has no solutions. Otherwise, (following the propagation by arc-consistency) resume value selection with the next variable (any other variable can be selected at this point). We can view MAC as if it searches a virtual binary tree, were at any value selection point, the choice is between a single value $x = a$ from its domain to the current variable, and between assigning any other value to this variable ($x = \neg a$). This distinction when associated with full arc-consistency at each such choice $\{a, \neg a\}$ sometimes yields more effective pruning. It also allows richer variable orderings strategies (see exercises).

Full and Partial look-ahead

Two algorithms that do more work than forward checking and less work than enforcing full arc-consistency at each state are *full looking ahead* and *partial looking ahead*. The full looking ahead algorithm makes a single pass through the future variables; in effect the "**repeat**" and "**until**" lines in selectValue-arc-consistency are removed.

Partial looking ahead (PLA) does less work than full looking ahead. It applies directional arc-consistency to the future variables. 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 variables following them.

Example 5.3.3 Consider the problem in Figure 5.3 using the same ordering of variables and values as in Figure 5.9. Partial-look-ahead starts by considering $x_1 = red$. Applying directional arc-consistency from x_1 towards x_7 will first shrink the domains of x_3, x_4 and x_7, (when processing x_1), as was the case for forward-checking. Later, when directional arc-consistency processes x_4 (with its only value, "blue") against x_7 (with its only value, "blue"), the domain of x_4 will become empty, and the value "red" for x_1 will be rejected. Likewise, the value $x_1 = blue$ will be rejected. Therefore, the whole tree in Figure 5.9 will not be visited if either partial-look-ahead or the more extensive look-ahead schemes are used. With this level of look-ahead only the subtree below $x_1 = green$ will be expanded.
□

Complexity of partial look-ahead. SelectValue-partial-look-ahead applies directional arc-consistency to k subproblems, each requiring in the worst-case, $O(ek^2)$ complexity. The total complexity is therefore bounded by $O(ek^3)$. As this bound is identical to that of optimal arc-consistency look-ahead, it is clear that it does not reflect well the actual complexity. Notice, however, that SelectValue-partial-look-ahead will not require a complex implementation of arc-consistency.

Dynamic value orderings (LVO)

We can use the information gathered from constraint propagation also to rank-order the promise of non-rejected values by estimating their likelihood to lead to a solution. Such a *look-ahead value ordering* (LVO) algorithm can be based on forward checking or any higher level of constraint propagation. Rather than just accepting the current variable's first value 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 or arc-consistency style look-ahead on the domains of future variables. (Each instantiation and its effects are retracted before the next instantiation is made.) LVO's strategies can vary by the amount of look-ahead performed and also by the heuristic measure they use to transform the propagated information into a ranking of the values. We list several popular LVO heuristics next.

One, called min-conflicts (MC), chooses the value which removes the smallest number of values from the domains of future variables. Namely, it considers each value in the domain of the current variable, and associates with it the total number of values in the domains of future variables with which it conflicts but which are consistent with the current partial assignment. The current variable's values are then selected in increasing order of this count.

A second heuristic is inspired by the intuition that a subproblem that includes variables having small domains (e.g., a single valued domains) is more likely to be inconsistent. Note that each of the search trees rooted at an instantiation of x_i is a different subproblem. The max-domain-size (MD) heuristic prefers the value in the current variable that creates the largest *minimum* domain size in the future variables. For example, if after instantiating x_i with value a the $min_{j \in \{i+1,...,n\}}|D'_j|$ is 2 and with $x_i = b$ the minimum is 1, then a will be preferred.

A third heuristic attempts to estimate the number of solutions in each potential subproblem. The ES (estimates solutions) computes an upper bound on the number of solutions by multiplying together the domain sizes of each future variable, after values incompatible with the candidate value of the current variable have been removed. The value that leads to the highest upper bound on the number of solutions is selected first.

Finally, another useful value ordering heuristic is to prefer the most recent value assigned to x_i (when it was last assigned, and since retracted) in order to capitalize on previous assembled partial solutions.

Experimental results [102] indicate that the cost of performing the additional look-ahead, while not justified on smaller and easier problems, can certainly be valuable on consistent large and hard problems. In particular the MC heuristic emerged as best among the above measures on a variety of randomly generated instances.

5.3.2 Look-ahead for variable ordering

Dynamic variable orderings (DVO)

The ordering of variables has a tremendous impact on the size of the search space. Empirical and theoretical studies have shown that certain fixed orderings are generally effective at producing smaller search spaces. In particular, the *min-width ordering* and *max-cardinality ordering* introduced in Chapter 4 both of which use information from the constraint graph, are quite effective.

When dynamically determining variable ordering during search, one common heuristic, known as "fail-first", is to select as the next variable the one likely to constrain the

```
subprocedure selectVariable

    $m \leftarrow \min_{i \leq j \leq n} |D'_j|$          (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 $i$th variable
end subprocedure
```

Figure 5.11: The subprocedure SELECTVARIABLE, which employs a heuristic based on the D' sets to choose the next variable to be instantiated.

remainder of the search space, the most. All other factors being equal, the variable with the smallest number of viable values in its (current) domain will have the fewest subtrees rooted at those values, and therefore the smallest search space below it. Given a current partial solution $\vec{a_i}$, we wish to determine the domain values for each future variable that are consistent with $\vec{a_i}$. We may estimate the domain sizes of future variables using the various levels of look-ahead propagation discussed above. Such methods are called *dynamic variable ordering* (DVO) strategies. An example is given in the SELECTVARIABLE subprocedure in Figure 5.11. GENERALIZED-LOOKAHEAD can be modified to employ dynamic variable ordering by calling SELECTVARIABLE after the initialization step "$i \leftarrow 1$" and after the forward step "$i \leftarrow i + 1$."

For clarity we explicitly specify DVO with forward-checking. Forward checking, which performs the least amount of look-ahead, has proven cost effective in many empirical studies.[1] We call this weak form of DVO *dynamic variable forward-checking* (DVFC). In DVFC, given a state $\vec{a_i} = (a_1, ..., a_i)$, the algorithm updates the domain of each future variable to include only values consistent with $\vec{a_i}$. Then, a variable with a domain of minimal size is selected. If any future variable has an empty domain, it is placed next in the ordering, and a dead-end will occur when this next variable becomes the current variable. This algorithm is described in Figure 5.12.

Example 5.3.4 Consider again the example in Figure 5.3. Initially, all variables have domain size of 2 or more. DVFC picks x_7, whose domain size is 2, and the value $< x_7, blue >$. Forward-checking propagation of this choice to each future variable restricts the domains of x_3, x_4 and x_5 to single values, and reduces the size of x_1's domain by one. DVFC selects x_3 and assigns it its only possible value, *red.* Subsequently, forward-checking causes variable x_1 to also have a singleton domain. The algorithm chooses x_1 and its only consistent value, *green.* After propagating this choice, we see that x_4 has one value, *red*; it is selected and assigned the value. Then x_2 can be selected and assigned its only

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

consistent value, *blue*. Propagating this assignment does not further shrink any future domain. Next, x_5 can be selected and assigned *green*. The solution is then completed, without dead-ends, by assigning *red* or *teal* to x_6. □

Notice that $DVFC$ accomplishes also the same value pruning associated with FORWARD-CHECKING. The information gleaned during the look-ahead phase can also be used to guide value selection. 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 can be useful.

Randomized backtracking

Since value and variable orderings can have a dramatic effect on the performance of backtracking algorithm with huge variances even on a single instance, and since finding a heuristic that will work well in all cases is unlikely, an alternative strategy explores randomness in values and variables selection. For example, when the algorithm needs to select a variable, it can use a variable selection heuristics such as min-domain, but breaks ties randomly. Similarly, a value may be selected randomly, or a random tie breaking rule can be employed on top of a popular value selection heuristic. In order to capitalize on the huge performance variance exhibited by randomized backtracking algorithms, it is common to restart the randomized algorithm after being aborted due to a self-imposed time cutoff. When running a sequence of such randomized backtracking algorithms with increasing cutoff times, the resulted scheme is guaranteed to find a solution if one exists or to report that the problem is inconsistent.

5.3.3 The cycle-cutset effect

We next present a relationship between the structure of the constraint graph and some forms of look-ahead. We will see more such properties in other chapters as well.

Definition 5.3.5 (cycle-cutset) *Given an undirected graph, a subset of nodes in the graph is a* cycle-cutset *if its removal results in a graph having no cycles.*

Proposition 5.3.6 *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 graphically expressed by deleting the corresponding variable from the constraint graph. Therefore, once the set of variables that forms a cycle-cutset is instantiated, the remaining problem can be perceived as a tree. As noted in Chapter

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

Figure 5.12: The DVFC algorithm. It uses the selectValue-forward-checking subprocedure given in Fig. 5.8.

4, 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 once 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 above-proposed complexity follows. $\square$

5.3.4 An implementation issue

The cost of node expansion when implementing a look-ahead strategy can be controlled if certain information is cached and maintained. One possibility is to maintain for each variable a table containing viable domain values relative to the partial solution currently being assembled. When testing a new value of the current variable, or after committing to a value for the current variable, the tables will tentatively be updated following the application of constraint propagation. This strategy requires an additional $O(n \cdot k)$ space. However, whenever a dead-end occurs, the algorithm has to recompute the tables associated with the search state to which the algorithm retracted. This may require n times the node generation cost that depends on the level of look-ahead used.

Another approach is to maintain a table of pruned domains for each variable and for each level in the search tree, which results in additional space of $O(n^2 \cdot k)$. Upon reaching a dead-end, the algorithm retracts to the previous level and uses the tables maintained at that level. Consequently, the only cost of node generation is the usual partial solution extension and that of updating the table.

5.3.5 Extensions to stronger look-ahead's

The collection of look-ahead techniques we've described is not meant to be exhaustive. For example, a search algorithm could enforce a degree of consistency higher than arc-consistency following each instantiation. Doing so entails not only deleting values from domains, but also adding new constraints. More recent work [64] has shown that as larger and more difficult problems are considered, higher levels of look-ahead become more useful. Therefore, it is likely that as experiments are conducted with larger and harder problems, look-ahead based on path-consistency becomes cost-effective.

Look-ahead algorithms can also be extended to handle some generalized definitions of consistency, such as relational consistency described in Chapter 8. Those definitions focus on constraints rather than on variables and are particularly well suited for the non-binary case. The "generalized arc-consistency" condition defined in Chapter 3 is particularly appropriate since it reduces domains only. We rewrite the definition here

Definition 5.3.7 (generalized arc-consistency) *Given a constraint $C = (R, S)$ and a variable $x \in S$, a value $a \in D_x$ is supported in C if there is a tuple $t \in R$ such that*

```
DPLL(φ)
Input: A cnf theory φ
Output: A decision of whether φ is satisfiable.
1. Unit_propagate(φ);
2. If the empty clause is generated, return(false);
3. Else, if all variables are assigned, return(true);
4. Else
5.        Q = some unassigned variable;
6.        return( DPLL( φ ∧ Q) ∨
                  DPLL(φ ∧ ¬Q) )
```

Figure 5.13: The DPLL Procedure.

$t[x] = a$. t is then called a support for $< x, a >$ in C. C is arc-consistent if for each variable x, in its scope and each of its values, $a \in D_x$, $< x, a >$ has a support in C. A CSP is arc-consistent if each of its constraints is arc-consistent.

The reader can try to extend SelectValue-arc-consistency and forward-checking to the generalized arc-consistency definition. (See exercises).

5.4 Satisfiability: Look-ahead in backtracking

The backtracking algorithm and its advances are applicable to the special case of propositional satisfiability. The most well known, and still one of the best variation of backtracking for this representation, is known as the *Davis-Putnam, Logemann and Loveland procedure (DPLL).* [60]. This is a backtracking algorithm applied to a CNF theory (a set of clauses) augmented with unit-propagation as the look-ahead method (see Figure 3.16 in Chapter 3.) This level of look-ahead is akin to applying a type of full arc-consistency (i.e., relational arc-consistency) at each node.

Figure 5.13 presents DPLL. Unlike our general exposition of search algorithms we define this one recursively. As usual, the algorithm systematically searches the space of possible truth assignments of propositional variables. Step 1 applies unit propagation (line 1), accomplishing some level of dynamic variable and value selection automatically within unit-propagation. Additional value and variable selection can be done at step 5 of Figure 5.13.

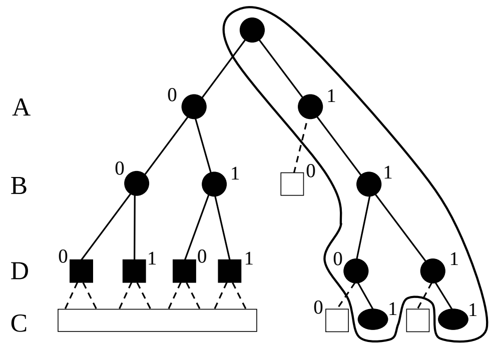


Figure 5.14: A backtracking search tree along the variables A, B, D, C for a CNF theory $\varphi = \{(\neg A \vee B), (\neg C \vee A), (A \vee B \vee D), C\}$. Hollow nodes and bars in the search tree represent illegal states, black ovals represent solutions. The enclosed area corresponds to DPLL with unit-propagation.

Example 5.4.1 Consider the party problem with the following rules:if Alex attends the party then Bill will as well. If Chris attends then Alex will, and if both Alex and Bill do not attend then David will. Assume we also know that Chris did not attend to the party. The cnf theory for this story is: $\varphi = \{(\neg A \vee B), (\neg C \vee A), (A \vee B \vee D), (C)\}$. The search tree of our party example along ordering A, B, D, C is shown in Figure 5.14. We see that there are two consistent scenarios. Suppose we now apply the DPLL algorithm. Applying unit propagation will resolve with unit clause C, yielding unit clause A, which in turn is resolved, yielding the unit clause B. Unit propagation terminates and we have three variable assignments. In step 5 the algorithm will select the only variable D. Since both of its truth assignments are consistent, we have two solutions. In this case the algorithm encountered no dead-ends. The portion of the search-space explored is marked is enclosed in Figure 5.14. □

Higher levels of look-ahead using stronger resolution-based inference can naturally be considered here. Different levels of bounded resolution, restricted by the size of the generated resolvents, or by the size of the participating clauses can be considered for propagation at each node in the search tree. The level of such propagation that is cost-effective is not clear, leading to the common tradeoff between more inference at each node vs exploration of fewer nodes. Clearly, when problems are larger and harder, more intensive look-ahead is likely to become cost-effective, as already observed for general constraints. One possible extension of unit-propagation is to apply path-consistency at each node. In the context of CNF theories, path-consistency is equivalent to applying resolution between clauses of length 2 at the most. Indeed, adding such *binary clause reasoning* has recently produced very effective versions of look-ahead on top of DPLL.

Finally, look-ahead methods can also be used for variable ordering, as we saw in the general case. A useful heuristic for variable ordering that has been reported for these DPLL-based algorithms is the *2-literal clause heuristic*. This heuristic suggests preferring a variable that would cause the largest number of unit propagations, where the number of possible unit propagations is approximated by the number of 2-literal clauses in which the variable appears. The augmented algorithm was demonstrated to significantly outperform a DPLL that doesn't employ this heuristic.

5.5 Summary

This chapter introduces backtracking search for solving constraint satisfaction problems and focused on enhancements that use look-ahead algorithms. The primary variants of look-ahead, forward-checking, full-look-ahead, partial-look-ahead and arc-consistency look-ahead were introduced and their use for variable and value selection was discussed. The complexity overheads of these methods was analyzed and the worst-case complexity bound as function of the problem's cycle-cutset was presented. Finally the chapter presented the DPLL-based backtracking search with look-ahead for satisfiability.

5.6 Chapter notes

Most current work on improving backtracking algorithms for solving constraint satisfaction problems use Bitner and Reingold's formulation of the algorithm [28]. One of the early and still one of the influential ideas for improving backtracking's performance on constraint satisfaction problems was introduced by Waltz [291]. Waltz demonstrated that when constraint propagation in the form of arc-consistency is applied to a two-dimensional line-drawing interpretation, the problem can often be solved without encountering any dead-ends. Golomb and Baumert [117] may have been the first to informally describe this idea. Consistency techniques for look-ahead are used in Lauriere's Alice system [184]. Explicit algorithms employing this idea have been given by Gaschnig [110], who described a backtracking algorithm that incorporates arc-consistency; McGregor [207], who described backtracking combined with forward checking; Haralick and Elliott [124], who added the various look-ahead methods described in this chapter (forward-checking, full look-ahead and partial look-ahead); and Nadel [216], who discussed backtracking combined with many partial and full arc-consistency variations. Gaschnig [109] has compared Waltz-style look-ahead backtracking (which corresponds to arc-consistency look-ahead) with look-back improvements that he introduced, such as backjumping and backmarking. Haralick and Elliot [124] have done a relatively comprehensive study of look-ahead and look-back methods at the time, in which they compared the performance of the various methods on n-queens problems and on randomly generated instances. Based on their empirical evaluation, they concluded that forward checking, the algorithm that uses the weakest form of constraint propagation, is superior. This conclusion was maintained for two decades until larger and more difficult problem classes were tested by Sabin and Freuder and and by Frost and Dechter [247, 102, 103]. In these later studies, forward checking lost its superiority on many problem instances to arc-consistency looking-ahead, MAC and full looking ahead and other stronger looking-ahead variants. Various Value ordering heuristics (MC,MD, ES) were tested empirically in [102]. Learning schemes for value ordering heuristics are also investigated [245]. Empirical evaluation of backtracking with dynamic variable ordering on the n-queens problem was reported by Stone [267]. More sophisticated DVO schemes have been investigated by Gent et. al. and Smith [134, 133, 264], focusing on computational tradeoff of various such heuristics. Following the introduction of graph-based methods [96, 77] Dechter and Pearl [77] introduced *advice generation*, a look-ahead value selection method that prefers a value if it leads to more solutions based on a tree-approximation of the problem at each node. Dechter introduced the cycle-cutset scheme [65] and its complexity bound for search as a function of the cycle-cutset size. The balance between overhead and pruning is studied in [102, 247, 16, 104].

Since 1997, many solution methods combine different algorithms, or exploit nondeterminism in the randomized version of backtracking search using either random restarts or randomizing backtrack points [39, 229]. This ideas were described initially by [192] and followed up by extensive empirical investigations [39, **?**, 119].

In the context of solving propositional satisfiability, the most notable development was Logemann, Davis and Loveland [60] backtracking search algorithm (called DPLL) that uses look-ahead in the form of *unit resolution*, which is similar to arc-consistency. Interestingly, the algorithm was preceded by a complete resolution algorithm called Davis-Putnam algorithm [61], which was confused for a long time with the search-based algorithm presented in 1962. Such confusions are still occurring today.

The *2-literal clause heuristic* was proposed by Crawford et. al. [55]. For at least one decade this algorithm was perceived as one of the most successful DPLL version for satisfiability. It provided the key idea for many subsequent heuristics for variable selection, namely, estimating the variable whose assignment will lead to the greatest amount of forward propagation. Another popular DPLL based algorithm named SATO [296] incorporated a sophisticated data structure called *watch-list* that allows a quick recognition of clauses that became unit clauses.

Methods for incorporating stronger resolutions for constraint propagation, were also tried, and often failed to be cost-effective. One such approach that is focused on the binary clauses resolution was recently demonstrated by Bacchus [13] to be effective if augmented with additional heuristics.

In the past ten years, the basic DPLL procedure was improved dramatically using both look-ahead methods described in this chapter and look-back methods, to be described in the next chapter. Sophisticated data-structures and engineering ideas contributed to the effectiveness of these schemes considerably. We elaborate more in Chapter 6.

Analytical average-case analysis for some backtracking algorithms has also been pursued for satisfiability [232] and for constraint satisfaction [124, 221, 217].

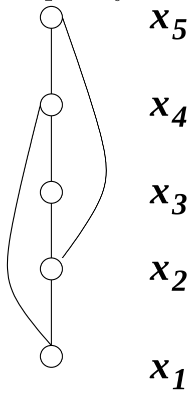
5.7 Exercises

1. The two primary search algorithms for exploring any search space are depth-first search (DFS) and Best-first search (BFS). These algorithms have complementary properties. For instance, when BFS terminates, it finds an optimal solution though it may require exponential space to do so. DFS requires only linear space, but may not terminate on infinite search spaces.

   Why are DFS-based algorithms the algorithms of choice for exploring constraint satisfaction search spaces?

2. Extend the forward-checking and arc-consistency look-ahead (ACLH) algorithm to exploit the generalized arc-consistency definition.

3. Let G be a constraint graph of a problem having 5 variables x_1, x_2, x_3, x_4, x_5 with arcs (x_1, x_2), (x_2, x_3) (x_3, x_4) (x_4, x_1) (x_4, x_5) (x_5, x_2). Assume the domains of all variables are $\{1, 2, 3\}$. Assume inequality constraints.



   For a problem having the constraint graph G and using ordering $d_1 = x_1, x_2, x_3, x_4, x_5$, confirm, reject or answer – and justify your answer:

   (a) Will backmarking help when searching this problem?

   (b) Apply forward-checking

(c) Apply arc-consistency look-ahead

(d) Apply Partial look-ahead

4. Consider the crossword puzzle in example 2.12 formulated using the dual-graph representation. Using the ordering $x_1, x_2, x_5, x_6, x_4, x_3$, show a trace, whose length is limited by a 1 page description, for each of the following algorithm:

   (a) Backtracking

   (b) Backmarking

   (c) Forward-checking

   (d) DVFC

   (e) Partial look-ahead

   (f) Bound the complexity of partial look-ahead as a function of the constraint graph.

5. Analyze the complexity of SELECTVALUE-ARC-CONSISTENCY when using (a) AC-3 (b) AC-4, for arc-consistency.

6. Analyze the overhead of each of the following algorithm: backtracking, backmarking, forward-checking, partial look-ahead and arc-consistency look-ahead. Assume that you are maintaining a table of current domains at each level of the search tree. Analyze each node generation separately and also analyze the complexity of generating a path along the search tree.

7. Apply the following algorithms to the 5-queens problem. Show only the search space generated for finding all solutions.

   (a) Forward-checking,

   (b) Dynamic variable ordering, DVFC,

   (c) Arc-consistency look-ahead,

   (d) Maintaining arc-consistency (MAC).

8. Implement DVO algorithms for 3 levels of look-ahead: forward-checking, arc-consistency and partial look-ahead. Run these algorithms on various random and real benchmarks, and compute three performance measures: the size of the search space (or number of dead-ends), the number of consistency checks, and CPU time. Present the comparison of your experiments.

9. We say that a problem is backtrack-free at level k of the search if a backtracking algorithm, once it reaches this level, is guaranteed to be backtrack-free. Assume that we apply DPC as a look-ahead propagation at each level of the search.

   (a) Can you identify a level for which DPC is backtrack-free?

   (b) Can you bound the complexity of DPC as a function of a width-2 cutset (a cutset is a width-2 if its removal from the graph yields a graph having induced-width of 2.)

10. Describe algorithm DPLL augmented with a look-ahead scheme that does bounded resolution such that any two clause creating whose size is bounded by 2 are carried out at each step.

11. A set of literals that reduces a cnf theory to a Horn theory is called a literal-Horn cutset. A set of propositions such that any of its truth assignment, is a literal Horn-cutset, is called a Horn cutset.

    (a) Assume that a CNF theory has a Horn-cutset of size h. Can you bound the complexity of DPLL as a function of h?

    (b) Suggest an algorithm for finding a small size Horn-cutset?

12. The Golomb ruler problem asks to place a set of n markers $x_1 > ... > x_n$ on the integer line, (assigning a positive integer to each marker) such that the distances between any two markers are all different, and such that the *length of the ruler*, namely the assignment to $x_n - x_1$, is the smallest possible.

    (a) Provide two ways of formulating the Golomb ruler as a constraint satisfaction problem. Assume that you know the optimal length l and you have to provide values for the markers. Demonstrate your formulation on a problem of small size (5 variables).

    (b) Discuss the solution of the problem by *Look-ahead methods* such as:

       i. DVFC

       ii. Forward-checking

       iii. Full-arc-consistency look-ahead

       iv. Partial look-ahead, as well as *look-back methods* such as backjumping and learning.

    (c) Propose two algorithms to apply and test for the problem.

13. The *combinatorial auction* problem: There is a set of items $S = \{a_1, ..., a_n\}$ and a set of m bids $B = \{b_1, ..., b_m\}$ Each bid is $b_i = (S_i, r_i)$ where $S_i \subseteq S$ and $r_i = r(b_i)$ is the cost to be paid for bid b_i. The task is to find a subset of bids $B_i \subseteq B$ s.t. any two bids in B_i do not share an item and to maximize for every $B_i \subseteq B$, $R(B_i)$ defined by: $R(B_i) = \sum_{b \in B_i} r(b)$. Assume knowledge of the value of the optimal solution.

    Answer items (a), (b), and (c) of the previous question.

14. The *Rain* problem: Given a communication network modeled as a graph $G = (N, E)$, where the set of nodes are processing nodes and the links are bidirectional communication links, each link e is associated with *bandwidth capacity* $c(e)$. A *demand* is a communication-need between a pair of nodes $d_u = (x_u, y_u, \beta_u)$ where x_u is the source y_u is the destination and β_u is the required bandwidth. Given a set of demands $\{d_1, ..., d_m\}$, the task is assign a route (a simple path) from source to destination for each demand such that the total capacity used over each link is below the available bandwidth. Assume again knowledge of the value of an optimal solution.

    Answer questions (a), (b), and (c) of the previous question.

Chapter 6

General search strategies: Look-back

We have now looked at the basic backtracking algorithm as it applies to CSPs, and at some schemes developed to forsee and avoid some future traps during the algorithm's *forward* phase (Chapter 5). This chapter introduces another class of improvement schemes which attempt to counteract backtracking's propensity for *thrashing*, or repeatedly rediscovering the same inconsistencies and partial successes during search. These schemes are invoked when the algorithm gets stuck in a dead-end and prepares to backtrack. Collectively, these are called *look-back* schemes, and they can be divided into two main categories. The aim of the first type is to improve upon the standard policy of retreating just one step backwards when encountering a dead-end. By analyzing the reasons for the dead-end, irrelevant backtrack points can often be avoided so that the algorithm jumps back directly to the source of failure, instead of just to the immediately preceding variable in the ordering. The aim of the second type of look-back scheme, called *constraint recording* or *no-good learning*, is to record 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. In this chapter, we will take a deep look at several algorithms that exemplify some of the many ways in which these two categories of improvement schemes can be implemented.

6.1 Conflict set analysis

Backjumping schemes are one of the primary tools for reducing backtracking's unfortunate tendency to repeatedly rediscover the same dead-ends. A dead-end occurs if x_i has no consistent values left relative to the current partial solution, in which case the backtracking algorithm will go back to x_{i-1}. Suppose that a new value for x_{i-1} exists and that there is no constraint whose scope includes x_i and x_{i-1}. In this case, x_{i-1} cannot be a cause for the dead-end at x_i, and a dead-end will again be reached at x_i, for each value of x_{i-1}, until all values of x_{i-1} have been exhausted. For instance, returning to our coloring

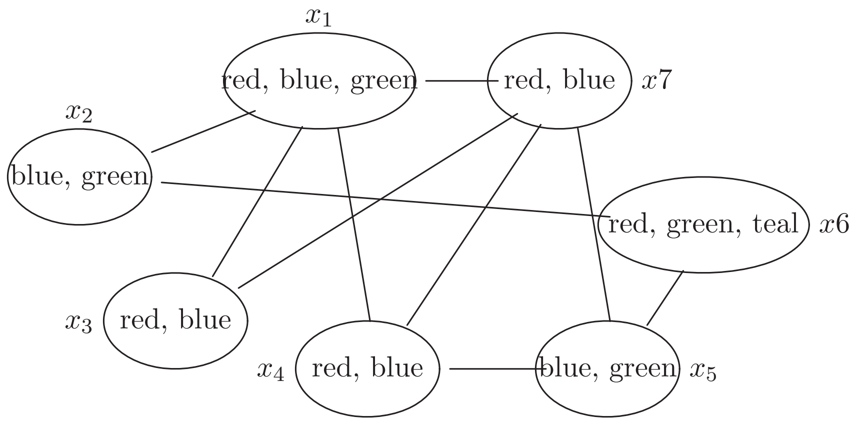


Figure 6.1: A modified coloring problem.

problem (reproduced here as Figure 6.1), if we apply backtracking along the ordering $d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$, we will encounter 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 because the value assigned at x_6 was not relevant to the cause of the dead-end at x_7.

We can ameliorate this situation by identifying a *culprit variable* responsible for the dead-end, and then immediately jump back and reinstantiate the culprit variable, instead of repeatedly instantiating the chronologically previous variable. Identification of culprit variables in backtracking is based on the notion of *conflict sets*. For ease of exposition, in the following discussion we assume a fixed ordering of the variables $d = (x_1, \ldots, x_n)$. This restriction can be lifted without affecting correctness, thus allowing dynamic variable orderings in all the algorithms.

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

The subtuple $\vec{a}_{i-1} = (a_1, ..., a_{i-1})$ may also be in conflict with x_{i+1}, and therefore going back to x_i only 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 state 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 normal control strategy often retreats to a subtuple $\vec{a_j}$ (alternately, to variable x_j) without resolving all or even any of these conflict sets. Therefore, rather than going to the previous variable, the algorithm should jump back from the dead-end state at $\vec{a_i} = (a_1, ..., a_i)$ to the most recent variable x_b such that $\vec{a}_{b-1} = (a_1, ..., a_{b-1})$ contains no conflict sets of the dead-end variable x_{i+1}. As it turns out, identifying this culprit variable is fairly easy. We next give a few definitions to capture the necessary concepts.

Definition 6.1.1 (conflict set) *Let $\bar{a} = (a_{i_1}, ..., a_{i_k})$ 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 $(\bar{a}, x = b)$ is consistent, we say that $\bar{a}$ is a* conflict set *of x, or that $\bar{a}$ conflicts with variable x. If, in addition, $\bar{a}$ does not contain a subtuple that is in conflict with x, $\bar{a}$ is called a* minimal *conflict set of x.*

Definition 6.1.2 (leaf dead-end) *Let $\vec{a_i} = (a_1, ..., a_i)$ be a consistent tuple. If $\vec{a_i}$ is in conflict with x_{i+1}, it is called a* leaf dead-end.

Definition 6.1.3 (no-good) *Given a network $\mathcal{R} = (X, D, C)$, any partial instantiation $\bar{a}$ that does not appear in any solution of $\mathcal{R}$ is called a* no-good. Minimal *no-goods have no no-good subtuples.*

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

Example 6.1.4 For the problem in Figure 6.1, the tuple $(< x_1, red >, < x_2, blue >, < x_3, blue >, < x_4, blue >, < x_5, green >, < x_6, red >)$ is a conflict set relative to x_7 because it cannot be consistently extended to any value of x_7. It is also a leaf dead-end. Notice that the assignment $(< x_1, blue >, < x_2, green >, < x_3, red >)$ is a no-good that is not a conflict set relative to any single variable. □

Whenever backjumping discovers a dead-end, it seeks to jump back as far as possible without skipping potential solutions. While the issue of *safety* in jumping can be made algorithm independent, the *maximality* in the magnitude of a jump needs to be defined relative to the information recorded by the algorithm. What is maximal for one style of backjumping may not be maximal for another, especially if they are engaged in different levels of information gathering.

Definition 6.1.5 (safe jump) *Let $\vec{a_i} = (a_1, ..., a_i)$ be a leaf dead-end state. We say that x_j, where $j \leq i$, is* safe *if the partial instantiation $\vec{a_j} = (a_1, ..., a_j)$ is a no-good, namely, it cannot be extended to a solution.*

In other words, we know that if x_j's value is changed from a_j to another value we will never explore again any solution that starts with $\vec{a_j} = (a_1, ..., a_j)$. But since $\vec{a_j}$ is a no-good, no solution will be missed.

Next we present three styles of backjumping. *Gaschnig's backjumping* jumps back to the culprit variable only at leaf dead-ends. *Graph-based backjumping* extracts information about irrelevant backtrack points exclusively from the constraint graph and jumps back at non-leaf (internal) dead-ends as well. *Conflict-directed backjumping* combines maximal backjumps at both leaf and internal dead-ends, but is not restricted to graph information alone.

6.2 Gaschnig's backjumping

We first define the notion of culprit that is used in Gaschnig's backjumping.

Definition 6.2.1 (culprit variable) *Let $\vec{a_i} = (a_1, ..., a_i)$ be a leaf dead-end. The* culprit index *relative to $\vec{a_i}$ is defined by $b = \min\{j \leq i | \ \vec{a_j} \ conflicts \ with \ x_{i+1}\}$. We define the* culprit variable *of $\vec{a_i}$ to be x_b.*

We use the notions of culprit tuple $\vec{a_b}$ and culprit variable x_b interchangeably. By definition, $\vec{a_b}$ is a conflict set that is minimal relative to a prefix tuple, namely, that tuple associated with a prefix subset of the ordered variables. We claim that x_b is both safe and maximal. That is,

Proposition 6.2.2 *If $\vec{a_i}$ is a leaf dead-end and x_b is its culprit variable, then $\vec{a_b}$ is a safe backjump destination and $\vec{a_j}$, $j < b$ is not.*

Proof: By definition of a culprit, $\vec{a_b}$ is a conflict set of x_{i+1} and is therefore a no-good. Consequently, jumping to x_b and changing the value a_b of x_b to another consistent value of x_b (if one exists) will not result in skipping a potential solution. On the other hand, it is easy to construct a case where backing up further than x_b skips solutions (left as an exercise). $\square$

Computing the culprit variable of $\vec{a_i}$ is relatively simple since at most i subtuples need to be tested for consistency with x_{i+1}. Moreover, it can be computed during search by gathering some basic information while assembling $\vec{a_i}$. Alternative description is that the culprit variable of a dead-end state $\vec{a_i} = (a_1, ..., a_i)$ is the most recent variable in $\{x_1, ..., x_i\}$, whose assigned value in $\vec{a_i}$ renders inconsistent the last remaining values in the domain of x_{i+1} not ruled out by prior variable-value assignments.

Rather than waiting for a dead-end $\vec{a_i}$ to occur, Gaschnig's backjumping uses information recorded while generating $\vec{a_i}$ to determine the dead-end's culprit variable x_b. The algorithm uses a marking technique whereby each variable maintains a pointer to the *most recently instantiated* predecessor found incompatible with any of the variable's values. While generating a partial solution in the forward phase, the algorithm maintains a

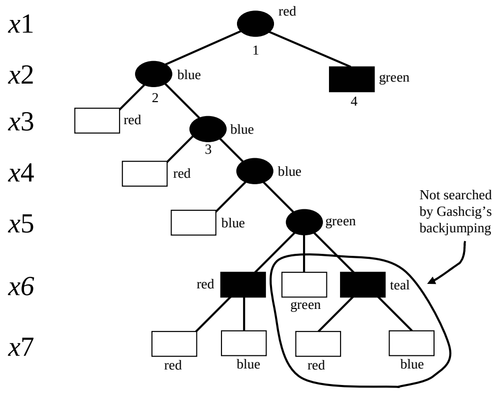


Figure 6.2: Portion of the search space explored by Gaschnig's backjumping, on the example network in Figure 6.1 under $x_1 = red$. The nodes circled are explored by backtracking but not by Gaschnig's backjumping. Notice that unlike previous examples we explicitly display leaf dead-end variables although they are not legal states in the search space.

pointer $latest_j$ for each variable x_j. The pointer identifies the most recent variable tested for consistency with x_j and found to have a value in conflict with a new value in D'_j. For example, if no compatible values exist for x_j and if $latest_j = 3$, the pointer indicates that $\vec{a_3}$ is a conflict set of x_j. If x_j does have a consistent value, then $latest_j$ is assigned the value $j-1$ (which allows to backtrack chronologically in non-leaf dead-ends). The algorithm jumps from a dead-end variable x_{i+1} (or, a leaf dead-end $\vec{a_i}$) back to $x_{latest_{i+1}}$, its culprit. Gaschnig's backjumping algorithm is presented in Figure 6.3.

Example 6.2.3 Consider the problem in Figure 6.1 and the order d_1. At the dead-end for x_7 that results from the partial instantiation ($< x_1, red >, < x_2, blue >, < x_3, blue >, < x_4, blue >, < x_5, green >, < x_6, red >$), $latest_7 = 3$, because $x_7 = red$ was ruled out by $< x_1, red >$, $x_7 = blue$ 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. Thus we see the algorithm's ability to backjump at leaf dead-ends. On subsequent dead-ends, as in x_3, it goes back to its preceding variable only. An example of the algorithm's practice of pruning the search space is given in Figure 6.2. □

Proposition 6.2.4 *Gaschnig's backjumping implements only safe and maximal backjumps in leaf dead-ends.*

Proof: Whenever there is a leaf dead-end x_i, the algorithm has a partial instantiation of $\vec{a}_{i-1} = (a_1, ..., a_{i-1})$. We only need to show that $latest_i$ is the culprit of x_i. Let $j = latest_i$.

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

procedure SELECTVALUE-GBJ
    while $D'_i$ is not empty
        select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$
        $consistent \leftarrow true$
        $k \leftarrow 1$
        while $k < i$ and $consistent$
            if $k > latest_i$
                $latest_i \leftarrow k$
            if not CONSISTENT($\vec{a_k}, x_i = a$)
                $consistent \leftarrow false$
            else
                $k \leftarrow k + 1$
        end while
        if $consistent$
            return $a$
    end while
    return null                         (no consistent value)
end procedure
```

Figure 6.3: Gaschnig's backjumping algorithm.

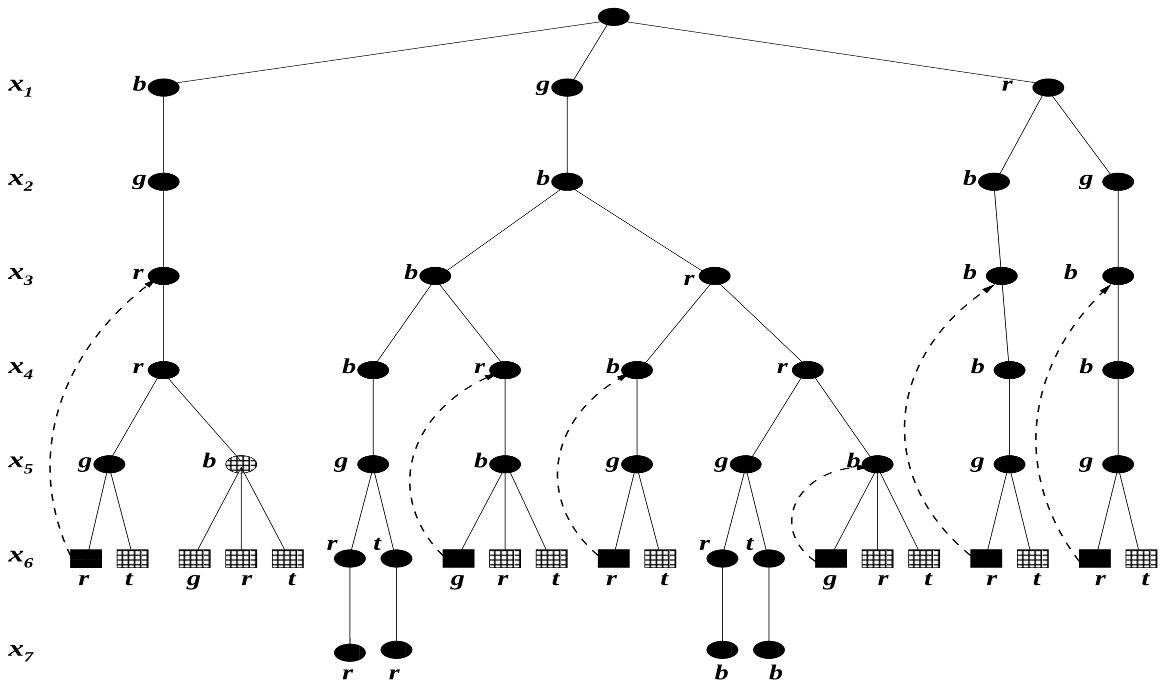


Figure 6.4: The search space for the graph coloring problem from Figure 6.1. Dashed arrows show the jumps performed at leaf dead-ends by Gaschnig's backjumping. Cross-hatched nodes are those avoided by this technique.

$\vec{a_j}$ is clearly in conflict with x_i, so we only have to show that $\vec{a_j}$ is minimal relative to prefix tuples. Since $j = latest_i$ when the domain of x_i is exhausted, and since a dead-end did not previously happen, 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 by Definition 6.2.1. From Proposition 6.2.2, it follows that this algorithm is safe and maximal. □

6.3 Graph-based backjumping

If a backtracking algorithm jumps back to a previous variable x_j from a leaf dead-end, and if x_j has no more candidate values to instantiate, then x_j is termed an *internal dead-end variable*, and $\vec{a}_{j-1}$ is an *internal dead-end state.*

Example 6.3.1 In Figure 6.4, all of the backjumps illustrated lead to internal dead-ends, except for the jump back to $(\langle x_1, green\rangle, \langle x_2, blue\rangle, \langle x_3, red\rangle, \langle x_4, blue\rangle)$, because this is the only case where another value exists in the domain of the culprit variable. □

In Gaschnig's backjumping, a jump of more than one level happens only at leaf dead-ends. At internal dead-ends, the algorithm takes the most conservative step of jumping back only one level. Algorithm *graph-based backjumping* implements jumps of more than one level at both internal dead-ends and leaf dead-ends.

Graph-based backjumping extracts knowledge about possible conflict sets from the structure of the constraint graph exclusively (that is, it recognizes variables, and the presence or absence of constraints between variables, but it does not use information about the domains of the variables or the nature of the constraints). Whenever a dead-end occurs and a solution cannot be extended to the next variable x, the algorithm jumps back to the most recent variable y that is *connected* to x in the constraint graph; if y has no more values, the algorithm jumps back again, this time to the most recent variable z connected to x *or* y; and so on. The second and any further jumps are jumps at *internal dead-ends*. By using the precompiled information encoded in the graph, the algorithm avoids computing $latest_i$ during each consistency test.

Graph-based backjumping uses the subset of earlier variables adjacent to x_{i+1} as an approximation of a minimal conflict set of x_{i+1}. It is an approximation because even when a constraint exists between two variables x_u and x_{i+1}, the particular value currently being assigned to x_u may not conflict with any potential value of x_{i+1}. For instance, assigning *blue* to x_2 in our graph coloring problem of Figure 6.1 has no effect on x_6 because *blue* is not in x_6's domain. Since graph-based backjumping does not maintain domain value information, it fills in this gap by assuming the worst: that the subset of variables connected to x_{i+1} is a minimal conflict set of x_{i+1}. Under this assumption, the latest variable in the ordering that precedes x_{i+1} and is connected to x_{i+1} is the culprit variable.

Studying graph-based backjumping is important because algorithms with performance tied to the constraint graph allow us to determine graph-theoretic complexity bounds, and thus to develop 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 (described in a subsequent section).

We now introduce some graph terminology which will be used in the following pages:

Definition 6.3.2 (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 $\vec{a_i} = (a_1, ..., a_i)$ is a dead-end, we equate $anc(\vec{a_i})$ with $anc(x_{i+1})$, and $p(\vec{a_i})$ with $p(x_{i+1})$.*

Example 6.3.3 The graph of Figure 6.1 is given in Figure 6.6a ordered along $d_1 = (x_1, ..., x_7)$. In this example, $anc(x_7) = \{x_1, x_3, x_4, x_5\}$ and $p(x_7) = x_5$. The parent of the leaf dead-end $\vec{a_6} = (blue, green, red, red, blue, red)$ is x_5, which is the parent of x_7. □

```
procedure GRAPH-BASED-BACKJUMPING
Input: A constraint network $\mathcal{R} = (X, D, C)$
Output: Either a solution, or a decision that the network is inconsistent.
    compute $anc(x_i)$ for each $x_i$ (see Definition 6.3.2 in text)
    $i \leftarrow 1$                      (initialize variable counter)
    $D'_i \leftarrow D_i$                 (copy domain)
    $I_i \leftarrow anc(x_i)$             (copy of anc() that can change)
    while $1 \leq i \leq n$
        instantiate $x_i \leftarrow$ SELECTVALUE
        if $x_i$ is null                  (no value was returned)
            $iprev \leftarrow i$
            $i \leftarrow$latest index in $I_i$   (backjump)
            $I_i \leftarrow I_i \cup I_{iprev} - \{x_i\}$
        else
            $i \leftarrow i + 1$
            $D'_i \leftarrow D_i$
            $I_i \leftarrow anc(x_i)$
    end while
    if $i = 0$
        return "inconsistent"
    else
        return instantiated values of $\{x_1, \ldots, x_n\}$
end procedure

procedure SELECTVALUE             (same as BACKTRACKING's)
    while $D'_i$ is not empty
        select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$
        if CONSISTENT($\vec{a}_{i-1}, x_i = a$)
            return $a$
    end while
    return null                       (no consistent value)
end procedure
```

Figure 6.5: The graph-based backjumping algorithm.

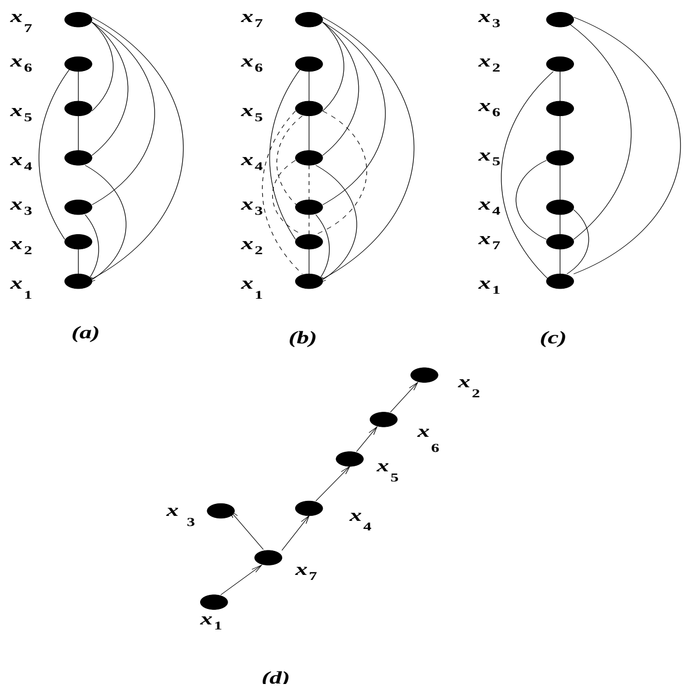


Figure 6.6: Several ordered constraint graphs of the problem in Figure 6.1: (a) along ordering $d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$, (b) the induced graph along d_1, (c) along ordering $d_2 = (x_1, x_7, x_4, x_5, x_6, x_2, x_3)$, and (d) a DFS spanning tree along ordering d_2.

It is easy to show that if $\vec{a_i}$ is a leaf dead-end, $p(\vec{a_i})$ is safe. Moreover, if only graph-based information is utilized and if culprit variables are not compiled as in Gaschnig's backjumping, it is unsafe to jump back any further. When facing an internal dead-end at x_i, however, it may not be safe to jump to its parent $p(x_i)$, as we see in the next example.

Example 6.3.4 Consider again the constraint network in Figure 6.1 with ordering $d_1 = x_1, ..., x_7$. In this ordering, x_1 is the parent of x_4. Assume that a dead-end occurs at node x_5 and that the algorithm returns to x_4. If x_4 has no more values to try, it will be perfectly safe to jump back to its parent x_1. Now let us consider a different scenario. The algorithm encounters a leaf dead-end 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 now unsafe, since changing the value of x_3 could theoretically 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, if we encounter an internal dead-end at x_4. □

Clearly, when encountering an internal dead-end, it matters which node initiated the retreat. The following few definitions identify the graph-based culprit variable via the *induced ancestor set* in the current *session*.

Definition 6.3.5 (session) *We say that backtracking* invisits *x_i if it processes x_i coming from a variable earlier in the ordering. The session of x_i starts upon the invisiting of x_i and ends when retracting to a variable that precedes x_i. At a given state of the search where variable x_i is already instantiated, the* current session *of x_i is the set of variables processed by the algorithm since the most recent* invisit *to x_i. The current session of x_i includes x_i and therefore the session of a leaf dead-end variable has a single variable.*

Definition 6.3.6 (relevant dead-ends) *The relevant dead-ends of x_i's session, denoted $r(x_i)$, are defined recursively as follows. The relevant dead-ends in the session of a leaf dead-end x_i, is just x_i. If x_i is a variable to which the algorithm retracted from x_j, then the relevant-dead-ends of x_i's session are the union of its current relevant dead-ends and the ones inherited from x_j, namely, $r(x_i) = r(x_i) \cup r(x_j)$.*

The above definition of relevant dead-ends seems involved. However this complication is necessary if we want to allow the biggest graph-based backjumping possible.

Definition 6.3.7 (induced ancestors, graph-based culprit) *Let x_i be a dead-end variable. Let Y be a subset of the variables that includes all the relevant dead-ends in x_i's current session. 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, $I_i(Y) = \cup_{y \in Y} anc(y) \cap \{x_1, ..., x_{i-1}\}$. The induced parent of x_i relative to Y, $P_i(Y)$, is the latest variable in $I_i(Y)$. We call $P_i(Y)$ the graph-based culprit of x_i.*

Theorem 6.3.8 *Let $\vec{a}_i$ be a dead-end (leaf or internal) and let Y be the set of relevant dead-end variables in the current session of x_i. If only graph information is used, the graph-based culprit, $x_j = P_i(Y)$, is the earliest variable to which it is safe to jump for graph-based backjumping.*

Proof (sketch): By definition of x_j, all the variables between x_j and the dead-end variable do not participate in any constraint with any of the relevant dead-end variables Y in x_i's current session. Consequently, any change of value to any of these variables will not perturb any of the no-goods that caused the dead-end in $\vec{a}_i$, and so jumping to x_j is safe.

Next we argue that if the algorithm had jumped to a variable *earlier* than x_j, some solutions might have been skipped. Let y_i be the first (relative to the ordering) relevant dead-end variable in Y, that is connected to x_j, encountered by backtracking. 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 for which y_i may not lead to a dead-end. If y_i is a leaf dead-end, and since x_j is an ancestor of y_i, there exists 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 tuple of R. Clearly, had the value of x_j been changed, the current assignment may have been extendible to a legal tuple of R and the dead-end at y_i may not have occurred. If y_i is an internal dead-end, there were no values of y_i that were both consistent with $\vec{a}_j$ and could be extended to a solution. It is not ruled out (when using the graph only), that different values of x_j, if attempted, could permit new values of y_i for which a solution might exist. The reader is asked (see exercises) to construct a counter example to make the above intuitive argument exact. □

Example 6.3.9 Consider again the ordered graph in Figure 6.6a, and let x_4 be a dead-end variable. If x_4 is a leaf dead-end, then $Y = \{x_4\}$, and x_1 is the sole member in its induced ancestor set $I_4(Y)$. The algorithm may jump safely to x_1. If x_4 is an internal dead-end with $Y = \{x_4, x_5, x_6\}$, the induced ancestor set of x_4 is $I_4(\{x_4, x_5, x_6\}) = \{x_1, x_2\}$, and the algorithm can safely jump to x_2. However, if $Y = \{x_4, x_5, x_7\}$, the corresponding induced parent set $I_4(\{x_4, x_5, x_7\}) = \{x_1, x_3\}$, 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 $I_3(\{x_3, x_4, x_5, x_7\}) = \{x_1\}$. If, however, $Y = \{x_4, x_5, x_6, x_7\}$, 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 yet again at x_5), then $I_4(\{x_4, x_5, x_6, x_7\}) = \{x_1, x_2, x_3\}$. The algorithm then retracts to x_3, and if it is a dead-end it will retract further to x_2, since $I_3(\{x_3, x_4, x_5, x_6, x_7\}) = \{x_1, x_2\}$. □

Algorithm GRAPH-BASED-BACKJUMPING is given in Figure 6.5. It can be shown that it jumps back at leaf and internal dead-ends as far as graph-based information allows.

Namely, for each variable x_i, the algorithm maintains x_i's induced ancestor set I_i relative to the relevant dead-ends in x_i's current session.

6.4 Conflict-directed backjumping

The two backjumping ideas we have discussed, 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. This algorithm uses the scheme we have already 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 $\vec{a}_i$ as the variables participating in $\vec{a}_i$'s *earliest minimal conflict set* of all relevant dead-ends in its session. We first define an ordering between constraints. Let *scope*(R) denote the scope of constraint R.

Definition 6.4.1 (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 6.4.2 (earliest minimal conflict set) *For a network $\mathcal{R} = (X, D, C)$ with an ordering of the variables d, let $\vec{a}_i$ be a leaf 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 6.4.1. 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-emc$(\vec{a}_i) \leftarrow$ var-emc$(\vec{a}_i) \cup S_j$. emc$(\vec{a}_i)$ is the subtuple of $\vec{a}_i$ containing just the variable-value pairs of var-emc$(\vec{a}_i)$. Namely, emc$(\vec{a}_i) = \vec{a}_i[var - emc(\vec{a}_i)]$.*

Definition 6.4.3 (jumpback set) *The* jumpback set *J_{i+1} of a leaf dead-end x_{i+1} is its var-emc$(\vec{a}_i)$. The jump-back set of an internal state $\vec{a}_i$ includes all the var-emc$(\vec{a}_j)$ of all the relevant dead-ends $\vec{a}_j$ $j \geq i$, that occurred in the current session of x_i. Formally, $J_i = \bigcup\{$var-emc$(\vec{a}_j) \mid \vec{a}_j$ is a relevant dead-end in x_i's session$\}$*

The definition of relevant dead-ends is exactly the same as in the graph-based case, when replacing *anc*$(\vec{a}_i)$ by var-emc$(\vec{a}_i)$. In other words, *var-emc*$(\vec{a}_i)$ plays the role that ancestors play in the graphical scheme, while J_i plays the role of induced ancestors. However, rather than being elicited from the graph, these elements are dependent on the particular value instantiation and can be uncovered during search. The variables *var-emc*$(\vec{a}_i)$ is a subset of the graph-based $anc(x_{i+1})$. The variables in $anc(x_{i+1})$ that are not included in *var-emc*$(\vec{a}_i)$ either participate only in non-affecting constraints (do not exclude any value of x_{i+1}) relative to the current instantiation $\vec{a}_i$ or, even if the constraints are potentially affecting, they are superfluous, as they rule out values of x_{i+1} eliminated by earlier constraints. Consequently, using similar arguments as we employed in the graph-based case, it is possible to show that:

Proposition 6.4.4 *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 no-goods that were active in causing this dead-end, and therefore the same dead-end will recur. To show that we cannot jump back any farther than x_j we must 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 that this solution might be missed if we jumped to an earlier variable. □

Algorithm CONFLICT-DIRECTED-BACKJUMPING is presented in Figure 6.7. It computes the jumpback sets for each variable and uses them to determine the variable to which it returns after a dead-end.

Example 6.4.5 Consider the problem of Figure 6.1 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. □

6.5 Complexity of backjumping

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

```
procedure CONFLICT-DIRECTED-BACKJUMPING
Input: A constraint network $\mathcal{R} = (X, D, C)$.
Output: Either a solution, or a decision that the network is inconsistent.

    $i \leftarrow 1$                          (initialize variable counter)
    $D'_i \leftarrow D_i$                        (copy domain)
    $J_i \leftarrow \emptyset$                         (initialize conflict set)
    while $1 \leq i \leq n$
        instantiate $x_i \leftarrow$ SELECTVALUE-CBJ
        if $x_i$ is null                  (no value was returned)
            $iprev \leftarrow i$
            $i \leftarrow$ index of last variable in $J_i$    (backjump)
            $J_i \leftarrow J_i \cup J_{iprev} - \{x_i\}$(merge conflict sets)
        else
            $i \leftarrow i + 1$                  (step forward)
            $D'_i \leftarrow D_i$                  (reset mutable domain)
            $J_i \leftarrow \emptyset$                   (reset conflict set)
    end while
    if $i = 0$
        return "inconsistent"
    else
        return instantiated values of $\{x_1, \ldots, x_n\}$
end procedure

subprocedure SELECTVALUE-CBJ

    while $D'_i$ is not empty
        select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$
        $consistent \leftarrow true$
        $k \leftarrow 1$
        while $k < i$ and $consistent$
            if CONSISTENT($\vec{a}_k, x_i = a$)
                $k \leftarrow k + 1$
            else
                $R_S \leftarrow$ the earliest constraint causing the conflict,
                add the variables in $R_S$'s scope $S$ excluding $x_i$, to $J_i$
                $consistent \leftarrow false$
        end while
        if $consistent$
            return $a$
    end while
    return null                     (no consistent value)
end procedure
```

Figure 6.7: The conflict-directed backjumping algorithm.

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 which 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 [92]). The arcs in a *DFS* spanning tree are directed towards the higher indexed node. For each node, its neighbor in the *DFS* tree preceding it in the ordering is called its *DFS parent*.

If we use graph-based backjumping on a *DFS* ordering of the constraint graph, finding the graph-based culprit 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 6.5.1 Consider, once again, the CSP in Figure 6.1. 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 6.6c,d. If a dead-end occurs at node x_3, the algorithm retreats to its *DFS* parent, which is x_7. $\square$

Theorem 6.5.2 *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 earliest safe 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 the induced ancestors of x. Clearly, $f(x)$ always appears 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 latest in the induced ancestors since all the relevant dead-ends that appear in x's session must be its descendents in the *DFS* subtree rooted at x. Let y be a relevant dead-end variable in the DFS subtree rooted at x. Because in a DFS tree of any graph the only non-tree arcs are back-arcs (connecting a node to one of its ancestors in the tree [92]) y's ancestors that precede x must therefore lie on the path (in the DFS spanning tree) from the root to x. 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 6.5.3 *When graph-based backjumping is performed on a* DFS *ordering of the constraint graph, the number of nodes visited is bounded by $O((b^m k^{m+1}))$, where b bounds the branching degree of the* DFS *tree associated with that ordering, m is its depth and k is the domain size. The time complexity (measured by the number of consistency checks) is $O(ek(bk)^m)$, where e is the number of constraints.*

Proof: Let x_i be a node in the DFS spanning tree whose DFS subtree has depth of $m-i$. We associate backjumping search along the DFS tree order with a AND-OR search space tree (rather than the usual tree search space) as follows. The AND-OR tree has two types of nodes: variable nodes are the OR nodes and value nodes are the AND nodes. The variable and value nodes are alternating. The root of the (AND-OR) search tree is the root variable of the DFS tree. It has k value child nodes, each corresponds to a possible consistent value. From each of these value nodes emanate b variable nodes, one for each of its child variables in the DFS tree. From each of these variable nodes, value child nodes emanate, each corresponding to a value consistent with the assignment on the path from the root, and so on. In general, any new value nodes that emanate from a variable node must be consistent with the assignment along the path to the root of the AND-OR search tree. The reader should convince himself that backjumping traverses a portion of this AND-OR search tree and therefore its size bounds the number of value tested for consistency with a partial instantiation. Let T_i stand for the number of nodes in the AND-OR search subtree rooted at node x_i (at level $m-i$ of the DFS spanning tree). Since any assignment of a value to x_i generates at most b independently solvable subtrees of depth $m-i+1$ or less, T_i obeys the following recurrence:

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

Solving this recurrence yields $T_m = b^m k^{m+1}$. Thus, the worst-case number of nodes visited by graph-based backjumping is $O(b^m k^{m+1})$. Since creating each value node requires e consistency tests we get $O(ek(bk)^m)$ consistency tests overall. Notice that when the DFS tree is balanced (when, each internal node has exactly b child nodes) the bound can be improved to $T_m = O((n/b)k^{m+1})$, since $n = O(b^{m+1})$. $\Box$

The above 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. Still, knowing what we should be minimizing may lead to useful heuristics.

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 a *DFS* ordering of an induced graph (G^*, d_1), and let m^*_d be the *DFS* tree depth. It can be shown that graph-based backjumping, if applied along ordering d, has the same backjump rule we saw on a DFS ordering of the original graph.

Proposition 6.5.4 *Let d be a DFS order of an induced-graph of a given constraint problem, then jumping back to its parent in this DFS tree is safe for graph-based backjumping.*

Theorem 6.5.5 *If d is a* DFS *ordering of (G^*, d_1) for some ordering d_1, having depth m^*_d, then the complexity of graph-based backjumping using ordering d is $O(\exp(m^*_d))$.*

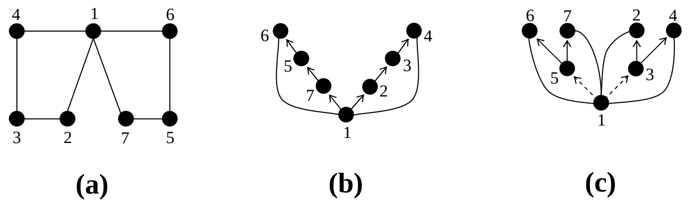


Figure 6.8: A graph (a), its DFS ordering (b), a DFS ordering of its induced-graph (c)

A proof, that uses somewhat different terminology and derivation, is given in [234]. The virtue of Theorem 6.5.5 is in allowing a larger set of orderings to be considered, each yielding a bound on backjumping's performance as a function of its *DFS* tree-depth. 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 which may lead to orderings having better bounds.

Example 6.5.6 Consider the graph G displayed in Figure 6.8a. Ordering $d_1 = 1, 2, 3, 4, 7, 5, 6$ is a DFS ordering having the smallest depth of 3. Consider now the induced-graph along the same ordering, which has the two added arcs: $\{(1, 3), (1, 5)\}$. A DFS ordering of the induced graph can be provided by: $d = 1, 3, 4, 2, 5, 6, 7$ that has a spanning tree depth of only 2. Consequently, graph-based backjumping is $O(exp(4)$ along d_1 and $O(exp(3))$ along d. Notice that while $f(x)$ is a safe backjump point, it may not be maximal; if going forward from 1 to 3, we realize a dead-end at 3 (imagine the domain of x_3 is empty to begin with), we should jump back earlier than 1 and conclude that the problem is inconsistent. □

6.6 *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? This may lead to special types of hybrids between look-ahead and look-back.

We can 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, i-backjumping instantiates them one at a time, as does conflict-directed backjumping. However, when selecting a new value for the next variable, the algorithm makes sure the new value is both consistent with past instantiation, and consistently extendible by the next $i-1$ variables (which will require revising the CONSISTENT procedure). Note that conflict-directed backjumping is *1-backjumping.* This computation will be performed at every node and can be exploited to generate conflict sets that conflict with i future variables, yielding *level-i conflict sets.* A tuple $\vec{a}_j$ is a level-i conflict set if it is not consistently extendible 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 is done in 1-backjumping. The balance between computation overhead at each node and the savings on node pruning should, of course, be considered. (For elaboration see exercises).

6.7 Learning Algorithms

The earliest minimal conflict set of Definition 6.4.2 is a no-good explicated during search and used to guide backjumping. However, this same no-good may be rediscovered as the algorithm explores different paths in the search space. By making this no-good explicit in the form of a new constraint we can make sure that the algorithm will not rediscover it, since it will be available for consistency testing. Doing so may prune some subtrees in the remaining search space. This technique, called constraint recording or *learning*, is the foundation of the learning algorithms described in this section.

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. There is no point, however, in recording the conflict set $\vec{a}_i$ itself as a constraint at this stage, because under the backtracking control strategy the current state will not recur.[1] However, 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 (i.e., minimal). As noted above, one obvious candidate is the earliest minimal conflict set, which is already identified 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.

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

In learning algorithms, the savings yielded from a potentially pruned search (by finding out in advance that a given path cannot lead to a solution) must be balanced against the overhead of processing at each node generation a more extensive database of constraints.[2]

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

We next present three primary 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.

6.7.1 Graph-based learning

Graph-based learning uses the same methods as graph-based backjumping to identify a no-good; information on conflicts is derived from the constraint graph alone. Given a leaf dead-end $\vec{a}_i = (a_1, \ldots, a_i)$, the values assigned to the ancestors of x_{i+1} are identified and included in the recorded conflict set. In internal dead-ends the induced-ancestor set will be considered.

Example 6.7.1 Consider the problem from Figure 6.1, when searching for *all solutions* in the ordering $d = x_6, x_3, x_4, x_2, x_7, x_1, x_5$. Figure 6.9b presents the search space explicated by naive backtracking and by backtracking, augmented with graph-based learning. Branches below the cut lines in Figure 6.9 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 no-good $(< x_2, green >, < x_3, blue >, < x_4, blue >, < x_7, red >)$. This no-good reappears later in the search, under the subtree rooted at $x_6 = teal$, and it can be used to prune the

[2]We make the assumption that the computer program represents constraints internally by storing the invalid combinations. Thus, increasing the number of stored no-goods increases the size of the data structure and slows down retrieval.

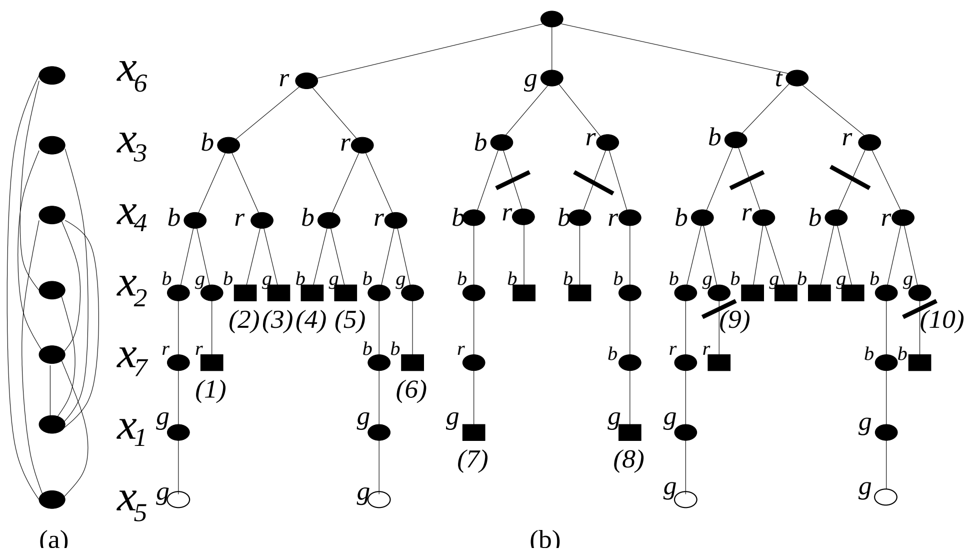


Figure 6.9: The search space explicated by backtracking on the CSP from Figure 6.1, using the variable ordering $(x_6, x_3, x_4, x_2, x_7, x_1, x_5)$ and the value ordering (*blue, red, green, teal*). Part (a) shows the ordered constraint graph, part (b) illustrates the search space. The cut lines in (b) indicate branches not explored when graph-based learning is used.

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 no-goods $(< x_3, blue >, < x_4, red >)$ and $(< x_3, red >, < x_4, blue >)$ 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 no-goods; if graph-based backjumping is used instead of backtracking, these dead-ends will be avoided however.) This learned constraint prunes the remaining search four times. The following additional no-goods are also recorded by graph-based learning at the indicated dead-ends: (6) with dead-end variable x_1: $(< x_2, green >, < x_3, red >, < x_4, red >, < x_7, blue >)$; (7) with dead-end variable x_5: $(< x_4, blue >, < x_6, green >, < x_7, red >)$; (8), dead-end variable x_5: $(< x_4, red >, < x_6, green >, < x_7, blue >)$; (9), dead-end variable x_7: $(< x_2, green >, < x_3, blue >, < x_4, blue >)$; (10), dead-end variable x_7: $(< x_2, green >, < x_3, red >, < x_4, red >)$. Note that dead-ends (9) and (10) occur at x_2 with learning, and at x_7 without learning. □

To augment graph-based backjumping with graph-based learning, we need only add a line (in bold face) to GRAPH-BASED-BACKJUMPING after a dead-end is encountered, as shown in Figure 6.5.

```
procedure GRAPH-BASED-BACKJUMP-LEARNING


    instantiate $x_i \leftarrow$ SELECTVALUE
    if $x_i$ is null                  (no value was returned)
        record a constraint prohibiting $\vec{a}_{i-1}[I_i]$.
        $iprev \leftarrow i$
        (algorithm continues as in Fig. 6.5)
```

Figure 6.10: Graph-based backjumping learning, modifying CBJ.

Recording a new constraint may require adding a new relation to the list of constraints C. If a constraint with scope I_i already exists, it may only be necessary to remove a tuple from this constraint. The overhead 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.

6.7.2 Deep vs shallow learning

Identifying and recording only minimal conflict sets constitutes *deep learning*. Not insisting on minimal conflict sets is "shallow" learning. Discovering all minimal conflict sets means acquiring all the possible information out of a dead-end. For the problem and ordering of Example 6.7.1 at the first dead-end, deep learning will record the minimal conflict set of x_1 (x_2 = 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 sets including both x_3 and x_4 that are recorded by graph-based learning. 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. Although deep learning is the most informative approach, its cost is prohibitive especially 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 may be $\binom{r}{r/2} \cong 2^r$, which amounts to exponential time and space complexity at each dead-end.

6.7.3 Jumpback learning

To avoid the explosion in time and space of full deep learning, we may settle for identifying just one conflict set, minimal relative to prefix conflict sets. The obvious candidate is

```
procedure CONFLICT-DIRECTED-BACKJUMP-LEARNING

    instantiate $x_i \leftarrow$ SELECTVALUE-CBJ
    if $x_i$ is null                    (no value was returned)
        record a constraint prohibiting $\vec{a}_{i-1}[J_i]$ and corresponding values
        $iprev \leftarrow i$
        (algorithm continues as in Fig. 6.7)
```

Figure 6.11: Conflict-directed bakjump-learning, modifying CBJ.

the *jumpback set* for leaf and internal dead-ends, as was explicated by conflict-directed backjumping. *Jumpback learning* uses this jumpback set, with the values assigned to the variables, as the conflict set to be learned. Because this conflict set is calculated by the underlying backjumping algorithm, the overhead in time complexity, at a dead-end, is only the time it takes to store the conflict set. As with graph-based learning, the modification required to augment CONFLICT-DIRECTED-BACKJUMPING with a learning algorithm is minor: specify that the conflict set is recorded as a no-good after each dead-end, as described in Figure 6.11.

Example 6.7.2 For the problem and ordering of Example 6.7.1 at the first dead-end, jumpback learning will record the no-good $(x_2 = \mathit{green}, x_3 = \mathit{blue}, x_7 = \mathit{red})$, since that tuple includes the variables in the jumpback set of x_1. □

In general, graph-based learning records the constraints with largest scopes, and deep learning records the smallest ones. As noted for backjumping, the virtues of graph-based learning are mainly theoretical (see Section 6.7.6); using this algorithm in practice is not advocated since jumpback learning is always superior. Nor is the use of deep learning recommended, because its overhead cost is usually prohibitive.

6.7.4 Bounded 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 only.

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

Definition 6.7.3 (i-relevant) *A no-good is i-relevant if it differs from the current partial assignment by at most i variable-value pairs.*

Definition 6.7.4 (i'th order relevance-bounded learning) *An i'th order relevance-bounded learning scheme maintains only those learned no-goods that are i-relevant and discard the rest.*

6.7.5 Non-systematic randomized backtrack-learning

Learning can be used to make incomplete search algorithms complete, that is, make them guaranteed to terminate with a solution, or with a proof that no solution exists. Consider a randomized backtracking-based algorithm (randomizing value or variable selection) that, after a fixed number of dead-ends, restarts with a different, randomly selected, variable or value ordering, or one that allows unsafe backjumping points. Study of such algorithms has been motivated by observing the performance of incomplete stochastic local search algorithms that often outperform traditional backtacking-based algorithms (see Chapter 7).

Often randomization, restarts or unsafe backjumping make the search algorithm incomplete. However, completeness can be reintroduced if all no-goods discovered are recorded and consulted. 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 guaranteed to find a solution. This same argument allows bounding the complexity of learning-based algorithms, discussed in the next subsection.

6.7.6 Complexity of backtracking with learning

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

Theorem 6.7.5 *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 $O(n \cdot (k)^{w^*(d)})$ and a time complexity of $O(n^2 \cdot (2k)^{w^*(d)+1})$, where n is the number of variables and k bounds the domain sizes.*

Proof: Graph-based learning has a one-to-one correspondence between dead-ends and conflict sets. 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 no-goods of size $w^*(d)$ or less. Moreover, a dead-end at a particular variable x can occur at most $k^{w^*(d)}$ times after which point constraints are learned excluding all possible assignments of its induced parents. So the total number of dead-ends for backtracking with learning is $O(n \cdot k^{w^*(d)})$, yielding space complexity of $O(n \cdot k^{w^*(d)})$. Since the total number of values considered between successive dead-ends is at most $O(kn)$, the total number of values considered during backtracking with learning is $O(kn \cdot n \cdot k^{w^*(d)}) = O(n^2 \cdot k^{w^*(d)+1})$. Since each value requires testing all constraints defined over the current variable, and at most $w^*(d)$ prior variables, at most $O(2^{w^*(d)})$ constraints are checked per value test, yielding a time complexity bound of $O(n^2 (2k)^{w^*(d)+1})$. $\square$

Recall that the time complexity of graph-based backjumping can be bounded by $O(exp(m^*_d))$, where m^*_d is the depth of a DFS tree of an ordered induced graph, while the algorithm requires only linear space. Note that $m^*_d \geq w^*(d)$. However, it can be shown [17] 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 $k^{\log n}$, we need to invest $O(exp(w^*(d))$ in space, augmenting backjumping with learning. In Chapter 10 we show an alternative argument for this time-space tradeoff.

6.8 Look-back techniques for satisfiability

Incorporating backjumping and learning into a backtrack-based algorithm for satisfiability requires figuring out how to compute the jumpback conflict set for clausal constraints and for bi-valued domains.

For simplicity's sake assume a CNF-based backtracking algorithm like DPLL, but initially without any unit-propagation. Here we do not distinguish between the variables in the jumpback set and the associated no-goods uncovered. Note that any no-good is a clause. Note also that a partial solution can be described by a set of literals. Thus, we define the jumpback set as a conflict set called a J-clause, which is determined as follows: whenever the algorithm encounters a leaf dead-end of variable x, there must be a clause that forces the literal x and another clause that forces $\neg x$ relative to the current partial solution. We say that clause $(\alpha \vee x)$ forces x relative to a current partial solution σ, if all the literals in α are reversed (negated) in σ. Let $(\alpha \vee x)$ and $(\beta \vee \neg x)$ be two such forcing clauses. By resolving these two clauses we generate a new no-good clause $J_x = (\alpha \vee \beta)$. Moreover, had we identified the two earliest clauses (see definition 6.4.1) that forbid x

and $\neg x$, their resolvent would have been the earliest conflict set of x.

In the case of internal dead-ends the J-clauses are updated as we saw for jumpback sets earlier. Let x be a leaf dead-end and let y be the most recent literal in σ that appears in the clause J_x (let us assume w.l.g. that y appears positively in σ). Therefore, $J_x = (\theta \vee \neg y)$. If $\neg y$ is consistent with the current prefix of σ truncated below y, y inherits J_x's conflict set[3]. That is: $J_y \leftarrow resolve(J_x, J_y \vee y)$ (note that J_y may be empty here) and search continues forward. Otherwise, if $\neg y$ is not consistent with the prefix of σ below y, there must exist a clause $(\delta \vee y)$ which forces y. The algorithm then resolves $J_x = (\theta \vee \neg y)$ with $(\delta \vee y)$, yielding a new conflict set of y, $J_y = (\theta \vee \delta)$. The algorithm now jumps back to the recent variable in J_y records J_y as a conflict and so on. Algorithm SAT-CBJ-LEARN given in Figure 6.8, maintains these earliest J_i conflict sets for each variable. Backjumping is implemented by jumping to the most recent variable in these J sets, while learning records these as clauses.

Notice that many details are hidden in the procedure CONSISTENT. Clearly the order by which clauses are tested can have a significant effect on what is learned. Our representation tests earlier clauses before later ones to allow a deep backjump. There are numerous variations of learning in the context of SAT formulas. At any dead-end different clauses can be learned and the type of clauses learned is indeed the focus of several of the new improved learning schemes (e.g., Grasp). The scheme proposed here favors learning early clauses.

The analysis and intricacy in learning grows when unit propagation is allowed inside CONSISTENT. One can then distinguish between forced assignments (due to unit propagation) and choice-based assignments. An efficient implementation of learning in the presence of unit propagation can benefit from specialized efficient data-structures (e.g., watch lists as proposed in [296]) that recognize quickly unit clauses. Indeed a sequence of learning algorithms for SAT with increased levels of implementation sophistication are being developed, often yielding remarkable leaps in performance. In the algorithm we presented, the preference for unit variable in the variable selection (step 3) will have the effect of unit propagation as in DPLL.

6.9 Integration and comparison of algorithms

6.9.1 Integrating backjumping and look-ahead

Complementary enhancements to backtracking can be integrated into a single procedure. The look-ahead strategies discussed in Chapter 5 can be combined with any of the

[3]Note that we abuse notation denoting by x both a variable and a literal over x. The meaning should be clear from the context

```
procedure SAT-CBJ-LEARN
Input: A CNF theory $\varphi$ over a set of variables $X = \{x_1, ..., x_n\}$, $\forall\ i\ D_i = \{l_i, \neg l_i\}$.
Output: Either a solution, or a decision that the formula is unsatisfiable.
1. $J_i \leftarrow \emptyset$
2.    While $1 \le i \le n$
3.       Select the next variable: $x_i \in X$, $X \leftarrow X - \{x_i\}$, prefer variables appearing in
      unit clauses.
4.       instantiate $x_i \leftarrow$ SELECTVALUE-SAT-CBJ-LEARN.
5.       If $x_i$ is null (no value returned), then
6.          add $J_i$ to $\varphi$        (learning)
7.          $iprev \leftarrow i$
8.       $i \leftarrow$ index of last variable in $J_i$, $\sigma[i]$ its $\sigma$ value    ($backjump$)
9.       $J_i \leftarrow resolve(J_i \vee l_i, J_{iprev})$        (merge conflict sets)
10.       else,
11.          $i \leftarrow i + 1$, $D_i \leftarrow \{l_i, \neg l_i\}$      (go forward)
12.       $J_i \leftarrow \emptyset$  (reset conflict set)
13.   Endwhile
14.   if $i = 0$ Return "inconsistent"
15.   else, return the set of literals $\sigma$
end procedure

subprocedure SELECTVALUE-SAT-CBJ-LEARN
1. While $D_i$ is not empty
2.    $l_i \leftarrow$ select $x_i \in D_i$, remove $l_i$ from $D_i$.
3.    consistent $\leftarrow$ $true$
4.     If CONSISTENT($\sigma \cup l_i$) then return $\sigma \leftarrow \sigma \cup \{l_i\}$.
5.     else, determine the earliest clause $(\theta \vee \neg l_i)$ forcing $\neg l_i$.
6.     $J_i \leftarrow J_j \vee \theta (= resolve(J_i \vee l_i, \theta \vee \neg l_i)$
7. Endwhile
8. Return $x_i \leftarrow$ null  (no consistent value)
end procedure subprocedure SELECTVALUE-SAT-CBJ-LEARN
```

Figure 6.12: Algorithm SAT-CBJ-LEARN

backjumping variants. Additionally, a combined algorithm can employ learning, and the dynamic variable and value ordering heuristics based on look-ahead information. One possible combination is conflict-directed backjumping with forward-checking level look-ahead and dynamic variable ordering. We present such an integrated algorithm, FC-CBJ, in Figures 6.13.

The main procedure of FC-CBJ closely resembles CONFLICT-DIRECTED-BACKJUMPING (Figure 6.7). 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, pruned some values of x_j.

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 GENERALIZED-LOOKAHEAD (Figure 5.7), since 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 5.11) in the initialization phase and during each forward step. These calls could be removed and the algorithm would revert without other modification to a static variable ordering. But because look-ahead is being performed for the purposes of rejecting inconsistent values, there is little additional cost in performing SELECTVARIABLE. In practice, this heuristic has been found most effective in reducing the size of the search space.

Apparently, there is some inverse relationship between look-ahead and look-back schemes; the more you look-ahead, the less you need to look-back. This can be shown at least for forward-checking, and for the simplest form of backjumping.

Proposition 6.9.1 *When using the same variable ordering, Gaschnig's backjumping always explores every node explored by forward-checking.* $\Box$. *For a proof see exercises.*

6.9.2 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 a theoretical analysis of worst or average case behavior, or determined empirically. Figure 6.14 shows the partial order relationships between several algorithms discussed in

```
procedure FC-CBJ
Input: A constraint network $\mathcal{R} = (X, D, C)$.
Output: Either a solution, or a decision that the network is inconsistent.

    $i \leftarrow 1$                          (initialize variable counter)
    call selectVariable                       (determine first variable)
    $D'_i \leftarrow D_i$ for $1 \le i \le n$ (copy all domains)
    $J_i \leftarrow \emptyset$                (initialize conflict set)
    while $1 \le i \le n$
        instantiate $x_i \leftarrow$ selectValue-fc-cbj
        if $x_i$ is null                      (no value was returned)
            $iprev \leftarrow i$
            $i \leftarrow$ latest index in $J_i$   (backjump)
            $J_i \leftarrow J_i \cup J_{iprev} - \{x_i\}$
            reset each $D'_k, k > i$, to its value before $x_i$ was last instantiated
        else
            $i \leftarrow i + 1$              (step forward)
            call selectVariable               (determine next variable)
            $D'_i \leftarrow D_i$
            $J_i \leftarrow \emptyset$
    end while
    if $i = 0$, return "inconsistent"
    else , return instantiated values of $\{x_1, \ldots, x_n\}$
end procedure
subprocedure selectValue-fc-cbj
    while $D'_i$ is not empty
        select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$
        $empty\text{-}domain \leftarrow false$
        for all $k$, $i < k \le n$
            for all values $b$ in $D'_k$
                if not consistent($\vec{a}_{i-1}, x_i = a, x_k = b$)
                    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 $D'_k$
            endfor
            if $D'_k$ is empty                (x_i = a leads to a dead-end)
                $empty\text{-}domain \leftarrow true$
        endfor
        if $empty\text{-}domain$              (don't select $a$)
            reset each $D'_k$ and $j_k, i < k \le n$, to status before $a$ was selected
        else , return $a$
    end while
    return null                               (no consistent value)
end subprocedure
```

Figure 6.13: The main procedure of the FC-CBJ algorithm.

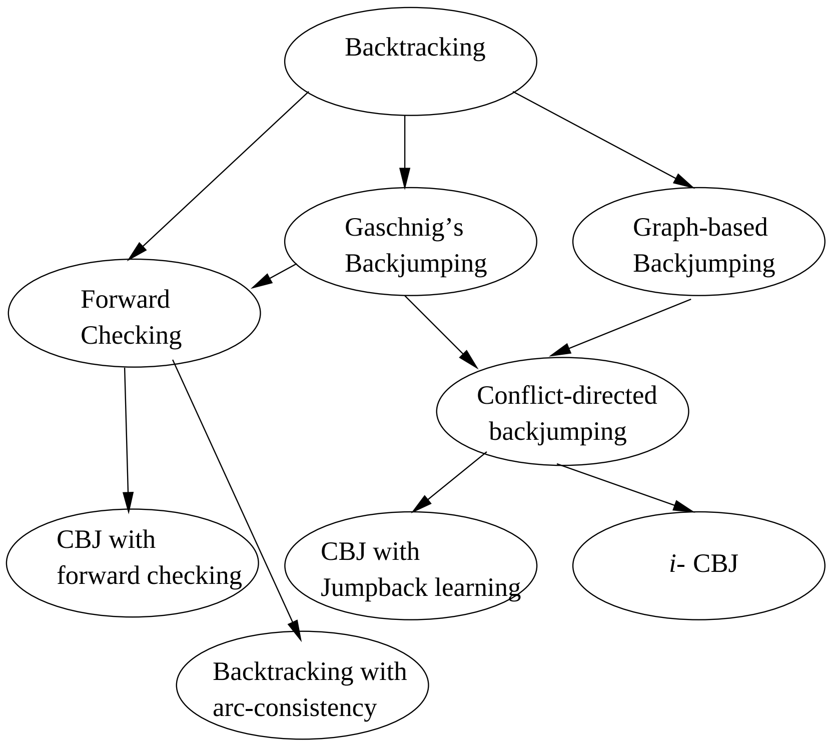

Backtracking
Gaschnig's Backjumping
Graph-based Backjumping
Forward Checking
Conflict-directed backjumping
CBJ with forward checking
CBJ with Jumpback learning
i- CBJ
Backtracking with arc-consistency

Figure 6.14: The relationships of selected backtracking-based algorithms. An arrow from A to B indicates that on the same problem and with the same variable and value orderings, the nodes in A's search tree will contain the nodes in B's.

this chapter, based on their search space. Algorithm A is superior to B (an arrow from A to B) if the search space of A is contained in the search-space of B.

The study of algorithms for constraint satisfaction problems has often relied upon experimentation to compare the relative merits of different algorithms or heuristics. Worst-case analysis, the primary analytical performance evaluation method, has focussed largely on characterizing tractable classes of problems. However, simple backtracking search, although exponential in the worst case, can have good average performance. The lack of effective worst-case analysis of backtrack search makes empirical evaluation mandatory.

Initially, empirical evaluations have been based on simple benchmark problems, such as the N-Queens puzzle, and on small randomly generated problem instances. In the 1990s, the experimental side of the field has blossomed, due to the increasing power of inexpensive computers and the identification of the "phase-transition" phenomenon (describe next), which has enabled hard random problems to be generated easily.

For an appropriate empirical studies, researchers have to face the question of problem distributions, and benchmarks. It is clear that multiple types of benchmarks and multiple ways of presenting the results are needed to be able to present a reliable picture. Benchmarks should include, individual instances that come from various applications, parameterized random problems, and application-based parameterized problems. Several criteria are frequently measured: CPU time, size of generated search tree, or calls to a common subroutine, such as CONSISTENT. The general goal of such studies is to identify a small number of algorithms that are dominating, namely observed to be superior on some class of problems.

Individual instances as benchmarks: The merit of this approach is that it is close, if not identical, to the underlying goal of all research: to solve real problems. If the benchmark problems are interesting, then the results of such a comparison are likely to be interesting. The drawback of using benchmarks of this type is that it is often impossible to extrapolate the results. Algorithm A may beat algorithm B on one benchmark and lose on another.
Random problems A contrasting technique for evaluating or comparing algorithms is to run the algorithm on artificial, synthetic, parameterized, randomly generated data. Since a practically unlimited supply of such random problems is easily generated, it is possible to run an algorithm on a large number of instances, thereby minimizing sampling error. Moreover, because the problem generator is controlled by several parameters, the experimenter can observe the possibly changing efficacy of the algorithm as one or more of the parameters change.
Application-based random problems The idea is to identify a problem domain (e.g., job shop scheduling) that can be used to define parameterized problems having a specific structure, and to generate instances by randomly generating values for the problem's parameters. This approach combines the virtues of the two approaches above: it focuses on problems that are related to real applications and it allows generating many instances for the purpose of statistical validity of the results.

The phase-transition

The phase transition phenomenon was studied extensively in the context of satisfiability. The theory of NP-completeness is based on worst-case complexity. The fact that 3-SAT is NP-complete implies (assuming the $NP \neq P$) merely that any algorithm for 3-SAT will take an exponential time for some problem instances. To understand the behavior of algorithms in practice, average-case complexity is more appropriate. For this we need to supply a probability distribution on formulas for each input length. Initially, experiments were conducted using random clauses length formulas. These were generated using a parameter p as follows. For each new clause of the m clauses, include each of the 2n literals in the new clause with probability p. It was shown however that DPLL solves these kinds of problems in polynomial average time (for a survey see [50]). Subsequently it was found that the fixed-length formulas took exponential time on the average and therefore were perceived as a more appropriate benchmark. Fixed length formulas are generated by selecting a constant number m of clauses, uniformly at random from the set of all possible clauses of a given length k. The resulting distribution is called random k-SAT. When investigating the performance of DPLL on these problems it was observed that when the number of clauses is small most instances are satisfiable and very easily solved, when the number of clauses is very large, most instances can be determined unsatisfiable quickly as well. In between, as the number of clauses grow, difficulty in solution grows up to a peak and then goes down monotonically. If c is the ratio of clauses to variables $C = m/n$, it was shown that for 3-SAT the peak occures around the ratio 4.2 (see Figure 6.15). Also between these ratios, the probability of satisfiability shifts smoothly from near 1 to near 0. It is intriguing that the peak in difficulty occurs near the ratio where about half of the formulas were satisfiable. The same pattern of hardness was found with some other algorithms and also for larger values of k, but with the transition at higher ratios, and the peak difficulty for DPLL being much greater [50].

The phase transition observation had a significant practical implication. It allowed researchers to test their algorithms on hard problem instances, sampled from the phase-transition, thus deliberately testing the average performance of hard instances.

A similar phenomenon was observed for binary CSPs. To generate binary random CSPs over N variables, of fixed length, a fixed number C of constraints are generated uniformly at random from the set of N(N-1)/2 constraints, and for each constraint, T pairs of values out of the possible K^2 values are selected uniformly at random as nogoods. Given a fixed number of variables, N, values K, and tightness T, the value of C in the phase transition of any backtrack search could be determined empirically while varying the number of constraints from small to large. Therefore, empirical studies, first determine the value of C that corresponds to the phase transition and subsequently generate instances from this region.

Some empirical evidence

Figure 6.16 demonstrates results of some typical experiments comparing several backtracking algorithms. All algorithms here incorporate forward checking level look-ahead and a dynamic variable ordering scheme similar to that described in Figure 5.11. The names are abbreviated in the table: "FC" refers to backtracking with forward-checking and dynamic variable ordering; "FC+AC" refers to forward-checking with arc-consistency enforced after each instantiation; "CBJ" refers to FC and conflict-directed backjumping; "FC+CBJ+LVO" adds a value ordering heuristic called LVO; "FC+CBJ+LRN" is CBJ plus 4th-order jumpback learning; "FC+CBJ+LRN+LVO" is CBJ with both LVO and learning. The columns labeled "Set 1" through "Set 3" report averages from 2000 ran-

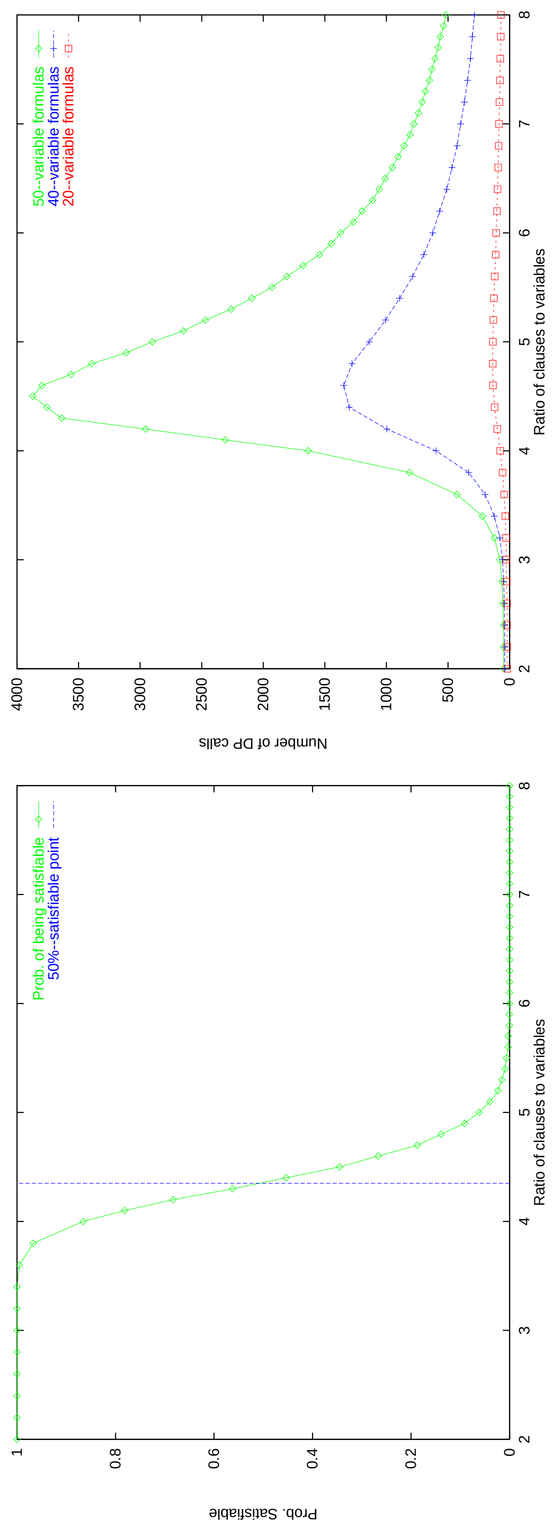


Figure 6.15: The phase transition for 3-SAT

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

Figure 6.16: 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.

domly 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 tuples per constraint was: Set 1: 200 and 8; Set 2: 300 and 7; Set 3: 350 and 6. The rightmost two columns of the figure show results on two specific problems from the Second Dimacs Implementation Challenge [156]: "ssa7552-038" and "ssa7552-158." For more details about the experiments, see [104].

These results show that interleaving an arc-consistency procedure with search was generally quite effective in these studies, as was combining 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. The problems with more restrictive, or "tighter," constraints, had sparser constraint graphs. With the looser constraints, the difference in performance among the algorithms was much less than on problems with tighter constraints. The arc-consistency enforcing and constraint-learning procedures were much more effective on the sparser graphs with tight constraints. These procedures are able to exploit the local structure in such problems. We also see that FC+AC prune the search space most effectively.

The empirical results shown in Figure 6.16 are only examples of typical experimental comparisons of algorithms. Unfortunately, it is not possible to conclude from these and similar studies, how the algorithms will perform on all problems having different structural properties.

6.10 Summary

This chapter described the primary look-back methods, backjumping and learning, for improving backtracking. Both graph-based approaches for backjumping and learning, as well as constraint-based approaches were presented and analyzed. We showed that backjumping's time complexity can be bounded exponentially as a function of the depth of a DFS tree traversing any of its induced graph, while its space complexity is linear. We also showed that the time and space complexity of learning during search can be bounded exponentially by the induced width. We also P explicitly present a look-back algorithm for satisfiability incorporated with DPLL. Some discussion of algorithms that combine look-ahead and look-back improvements were presented, and a comparison of the various algorithms using a partial order over their search-spaces was given.

6.11 Chapter notes

Truth-maintenance systems was the earliest area to contribute to the look-back aspect of backtracking. Stallman and Sussman [265] were the first to mention no-good recording, and their idea gave rise to look-back type algorithms, called *dependency-directed backtracking*, that include both backjumping and no-good recording [206]. Their work was followed by Gaschnig [110] who introduced the backjumping algorithm and also coined the name. Researchers in the logic-programming community were also among the earliest to try and improve a backtracking algorithm used for interpreting logic programs. Their improvements, known under the umbrella name *intelligent backtracking*, has focused on the basic principled ideas for a limited amount of backjumping and constraint recording [34, 242, 35, 54].

Later, following the introduction of graph-based methods [96, 77] Dechter [65] described the graph-based variant of backjumping, which was followed by Prosser's conflict-directed backjumping [230]. She also introduced learning no-goods into backtracking that includes graph-based learning [65]. Dechter and Pearl [77] identified the induced-width bound on learning algorithms. Frueder and Quinn [97] noted the dependence of backjumping's performance on the depth of the DFS tree of the constraint graph, and Bayardo and Miranker [234] improved the complexity bound. They also observed that with (relevance-based) learning, the time complexity of graph-based backjumping can be reduced by a factor of $k^{\log n}$ at an exponential space cost in the induced-width [234]. Ginsberg introduced *Dynamic backtracking* algorithm [114] which employs a similar notion of keeping only relevant learned no-goods that are most likely to be consulted in the near-future search. Dynamic backtracking uses an interesting variable reordering strategy that allow exploiting good partial solutions constructed in earlier subtrees. The usefulness of the DFS ordering for distributed execution of backjumping was also shown [48].

Subsequently, as it became clear that many of backtracking's improvements are orthogonal to one another (i.e., look-back methods and look-ahead methods), researchers have more systematically investigated various hybrid schemes in an attempt to exploit the virtues in each method. Dechter [65] evaluated combinations of graph-based backjumping, graph-based learning and the cycle-cutset scheme. An evaluation of hybrid schemes was carried out by Rosiers and Bruynooghe [242] on coloring and cryptarithmetic problems, who combined dynamic variable orddering and some limited look-back, and Prosser [230], who combined known look-ahead and look-back methods and ranked each combination based on average performance, primarily on Zebra problems. Dechter and Meiri [74] have evaluated the effect of pre-processing by directional consistency algorithms on backtracking and backjumping.

Before 1993, most of the empirical testing was conducted on relatively small problems (up to 25 variables), and the prevalent conclusion was that only low-overhead methods are cost effective. With improvements in hardware and recognition that empirical evaluation may be the best way to compare the various schemes, a substantial increase in empirical testing has been realized. After Cheeseman, Kanefsky, and Taylor [43] 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 [101, 100, 102, 114, 55, 17, 15]. 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 [14], 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 [169].

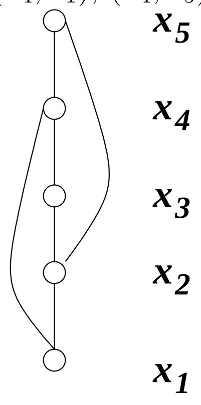
The value of look-back improvements for solving propositional satisfiability was initially largely overlooked, when most algorithms focused on look-ahead improvements of DPLL [55, 296]. This was changed significantly with the work by Bayardo and Schrag in 1997 [157]. They showed that their algorithm *relsat*, which incorporates both learning and backjumping outperformed many of the best look-ahead-based SAT solvers based on hard benchmarks available at the time. Subsequently, several variants of learning were proposed for SAT solvers (e.g., Grasp [201]) all incorporating an efficient data structure called *watch-list* proposed in the DPLL solver SATO [296], followed by even more intricate engineering ideas in learning-based solvers such as Chaff [193]. We should note that these learning-based techniques work well especially on structured problems while they normally are inferior to DPLL based procedures on uniformly random instances. In particular, backjump-learning based SAT solvers performed extremely well for planning instances and for symbolic model verification problems [3].

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 [294] and was tested for SAT by [239]. This idea recently caught up in the community of SAT-solver developers as well.

Recently, constraint processing techniques have been incorporated into the *Constraint Logic Programming (CLP)* languages. The inference engine of these languages uses a constraint solver as well as the traditional logic programming inference procedures. One of the most useful constraint techniques included in those languages is the use of various levels of arc-consistency in look-ahead search [283, 141]. It remains to be seen whether various styles of look-back methods can also be incorporated in a cost-effective manner.

6.12 Exercises

1. Let G be a constraint graph of a problem having 5 variables x_1, x_2, x_3, x_4, x_5 with arcs (x_1, x_2), (x_2, x_3), (x_3, x_4), (x_4, x_1), (x_4, x_5), (x_5, x_2).



For a problem having the constraint graph G and using ordering $d_1 = (x_1, x_2, x_3, x_4, x_5)$, answer, confirm or reject:

(a) Graph-based backjumping will always behave exactly as backtracking.

(b) Gaschnig's backjumping will always behave like backtracking.

(c) Conflict-directed backjumping and Gaschnig's backjumping are always identical on G.

(d) Graph-based learning over G and ordering d_1 will never record constraints of size greater than two.

(e) If a leaf dead-end occurs at x_5, what is the induced-ancestor set $I_5(\{x_5\}$? what is the induced ancestors $I_3(x_3, x_4)$? $I_3(x_3, x_4, x_5)$?

(f) Propose a better ordering for graph-based backjumping. Justify your answer.

(g) If a leaf dead-end occurs at x_5 and another internal dead-end at x_4, what is the conflict-set recorded by graph-based learning?

   (h) How would your answer on all previous questions be different if constraint (x_2, x_3) is omitted?

2. Complete the proof of Proposition 6.2.2, namely construct an example showing that backing up further from the culprit in leaf dead-ends skips solutions.

3. Prove the claim that: "For leaf dead-ends, jumping back to the parent will not cause any solution to be missed."

4. Complete the proof of Theorem 6.3.8 by constructing a counter example the proves that it is not safe to back up more than the graph-based culprit when using graph-based backjumping.

5. Let m_d be the depth of a DFS tree of an induced graph along some ordering d_1. Let d be the ordering. a) prove that graph-based backjumping always jumps to the parent in the DFS tree or to an earlier variable. (b) Prove that backjumping along d is bounded exponentially by m_d.

6. Describe backjump and graph-based learning algorithms for SAT that are incorporated in DPLL with unit propagation look-ahead.

7. Consider the crossword puzzle in Figure 6.18 as descibed in Chapter 2, formulated using the dual-graph representation. Using the ordering $(x_1, x_2, x_5, x_6, x_4, x_3)$, show a trace, whose length is limited to a 1-page description, or provide the required answer, for each of the following algorithms:

   (a) graph-based backjumping

   (b) graph-based backjumping-learning

   (c) jumpback-learning

   (d) Bound the complexity of graph-based backjumping on the crossword puzzle along the ordering above.

   (e) Bound the complexity of graph-based backjumping-learning using that ordering.

8. Consider the graph in Figure 6.17. Provide a DFS ordering of the graph and bound the complexity of solving any problem having that graph.

9. Precisely define algorithm i-backjumping by defining its i-CONSISTENT procedure.

10. Suppose we try to solve the problem in Figure 6.19 along the ordering $d = (x_1, x_2, x_3, x_4, x_5)$.

   (a) Show the trace and no-goods recorded by graph-based learning.

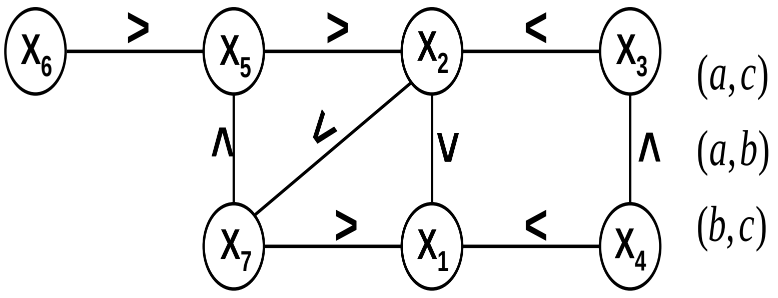


Figure 6.17: A constraint graph

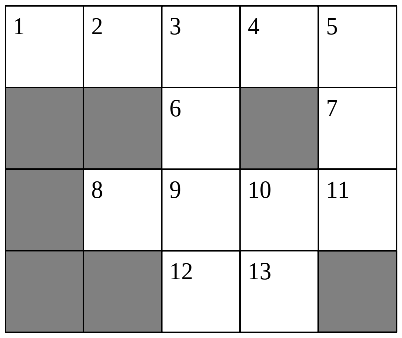


Figure 6.18: A crossword puzzle

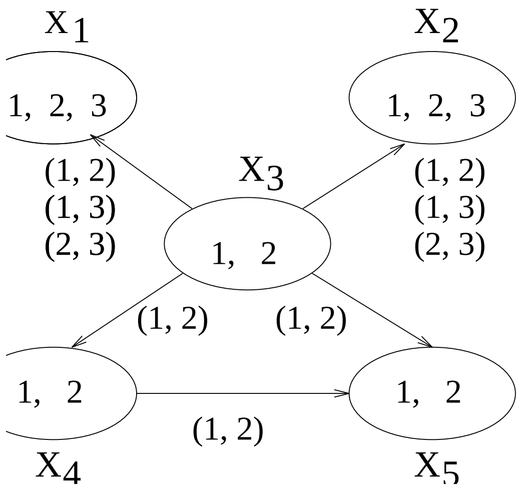


Figure 6.19: A small CSP. The constraints are: $x_3 < x_1, x_3 < x_2,\ x_3 < x_5,\ x_3 < x_4,\ x_4 < x_5$. The allowed pairs are shown on each arc.

   (b) Discuss the performance of deep learning on that problem

   (c) Show how jumpback learning will behave upon a dead-end at x_5.

11. Prove that learning can provide a time complexity bound that is smaller by a factor of $n^{log_2 k}$ of backjumping

12. a) Prove the correctness of FC-CBJ.
    b) Analyze the overhead at each node of SAT-CBJ-LEARN.

13. Prove proposition 6.9.1that [170] When using the same variable ordering, Gaschnig's backjumping always explores every node explored by forward-checking.

14. The Golomb ruler problem was described in Chapter 5. Discuss the relevance and effect of backjumping and learning for this problem.

15. Propose a learning and backjumping algorithm for the *combinatorial auction* problem presented in Chapter 5. Trace the performance of the algorithm on a small problem instance. To what extent you think graph-based backjumping and conflict-directed backjumping are effective for this clas of problems. Do the same for the Rain problem.