Chapter 6

General search strategies: Look-back

We have looked at the basic backtracking algorithm as it applies to CSPs and at some schemes developed to counteract backtracking's propensity for *thrashing*, or repeatedly rediscovering the same inconsistencies and the same partial successes during search. In Chapter 5, we introduced schemes that seek to improve the performance of backtracking during its *forward* phase. Here we introduce a class of improvement schemes which are invoked when the algorithm is preparing the backtracking step *after* encountering a dead-end. Collectively they 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 always retreating only one step backwards upon encountering a dead-end. By analyzing the reasons for the dead-end, however, irrelevant backtrack points can often be avoided so that the algorithm goes back directly to the source of failure, instead of just to the immediately preceding variable in the ordering. The aim of the second type of look-back scheme 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. The terms used to describe this type of algorithm are *constraint recording* and *no-good learning*. In this chapter, we will take a deeper 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}. Then 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}

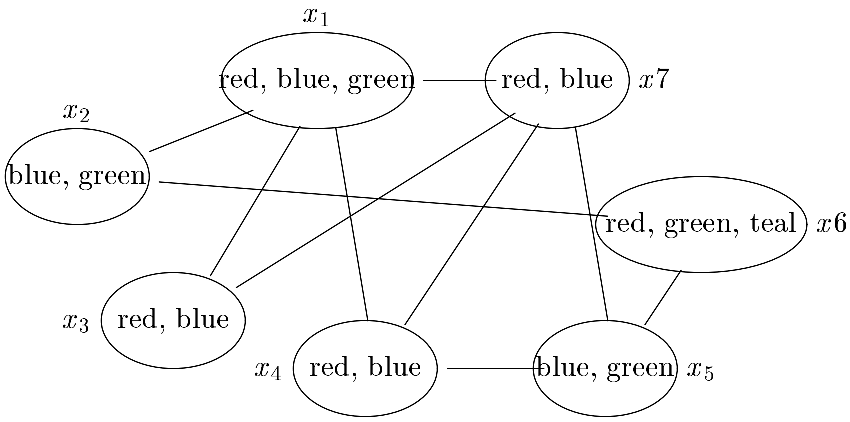


Figure 6.1: A modified coloring problem.

until all values of x_{i-1} have been exhausted. For instance, to continue with our coloring 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 have a dead-end at x_7 given the assignment $(< x_1, red >, < x_2, blue >, < x_3, blue >, < x_4, blue >, < x_5, green >, < x_6, red >)$. Backtracking will then return to x_6 and instantiate it as $x_6 = teal$, but the same dead-end will be encountered at x_7 because the value assigned at x_6 was not the cause of the dead-end at x_7.

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

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 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.

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

Definition 6.1.2 (leaf dead-end) *Given a variable ordering $d = (x_1, ..., x_n)$, let $\vec{a_i} = (a_1, ..., a_i)$ be a tuple that is consistent. If $\vec{a_i}$ is in conflict with x_{i+1} it is called a* leaf dead-end.

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

A conflict set is clearly a no-good, but there are no-goods that are not conflict sets of any single variable. Namely, they may conflict with two or more variables 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 should jump as far back as possible without skipping potential solutions. Intuitively, these two issues of *safety* in jumping and *maximality* in the magnitude of a jump need to be defined relative to the *information status* of a given algorithm. What is safe and maximal for one style of backjumping may not be safe and maximal for another, especially if they are engaged in different levels of information gathering. in the remainder of this section we will discuss the notion of safely when jumping back that are relevant to any style of backjumping.

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, no solution will be missed.

Definition 6.1.6 (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, those associated with a prefix subset of the ordered variables. We claim that x_b is both safe and maximal: safe in that $\vec{a_b}$ cannot be extended to a solution; maximal in that jumping back to an earlier node risks missing a solution.

Proposition 6.1.7 *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 therefore is a no-good. Consequently, jumping to x_b and changing the value a_b of x_b to another consistent value of x_b (if one exists) will not result in skipping a potential solution. To prove maximality observe that if the algorithm jumps to x_{b-j}, then by definition $\vec{a}_{b-j}$ is not a conflict set of x_{i+1}, and it may therefore be part of a solution. Note that $\vec{a}_{b-j}$ may, in fact, be a nogood of the original problem, but since it is not a nogood of the subproblem defined on $\{x_1, ...x_i\}$, backtracking algorithm cannot determine whether it is safe given the information it has. $\Box$.

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

Next we present three variants of backjumping. *Gaschnig's backjumping* implements the idea of jumping back to the culprit variable only at leaf dead-ends. *Graph-based backjumping* extracts information about irrelevant backtrack points exclusively from the constraint graph. It introduces the notion of jumping back at internal dead-ends as well as at leaf dead-ends. *Conflict-directed backjumping* combines maximal backjumps at both leaf and internal dead-ends which are not based on graph information alone.

6.2 Gaschnig's backjumping

Rather than waiting for a dead-end $\vec{a_i}$ to occur, Gaschnig's backjumping records some information while generating $\vec{a_i}$ and uses this information to determine the dead-end's

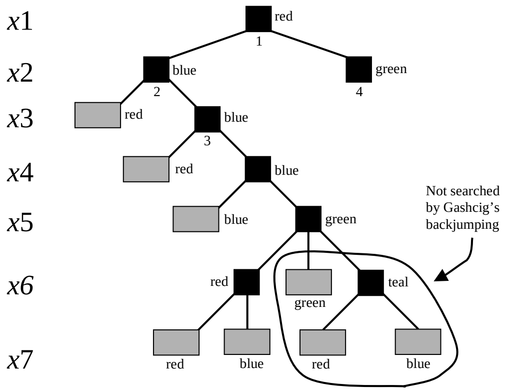


Figure 6.2: The search space explored by Gaschnig's backjumping, on the example network in Figure 6.1. The nodes surrounded by the circle are explored by backtracking but not by Gaschnig's backjumping.

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 $\vec{a}$ in the forward phase, the algorithm maintains a pointer $latest_i$ 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, when 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 $i-1$. The algorithm jumps from a leaf dead-end $\vec{a_i}$ that is inconsistent with x_{i+1}, back to $x_{latest_{i+1}}$, its culprit since the dead-end variable is x_{i+1}. Gaschnig's backjumping algorithm is presented in Figure 6.3.

Example 6.2.1 For the problem in Figure 6.1 and continuing to use 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 $x_1 = red$, $blue$ was ruled out by $x_3 = blue$, and no later variable had to be examined. On returning to x_3, the algorithm finds no further values to try ($D'_3 = \emptyset$). Since $latest_3 = 2$, the next variable examined will be x_2. This demonstrates 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 its pruning the search space is given in Figure 6.2. □

Proposition 6.2.2 *Gaschnig's backjumping implements only safe and maximal backjumps in leaf dead-ends.*

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

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

Figure 6.3: Gaschnig's backjumping algorithm.

Proof: Whenever there is a leaf dead-end $\vec{a}_{i-1}$, the algorithm has a partial instantiation $\vec{a}_{i-1} = (a_1, ..., a_{i-1})$. Let $j = latest_i$. The algorithm jumps back to x_j, namely, to the tuple $\vec{a_j}$. Clearly, $\vec{a_j}$ is in conflict with x_i, so we only have to show that $\vec{a_j}$ is minimal. Since $j = latest_i$ when the domain of x_i is exhausted, and since a dead-end did not happen previously, any earlier $\vec{a_k}$ for $k < j$ is not a conflict set of x_i, and therefore x_j is the culprit variable as defined by definition 6.1.6. From Proposition 6.1.7, it follows that this algorithm is safe and maximal. $\square$

In Gaschnig's backjumping, a jump happens only at leaf dead-ends. If all the children of a node in the search tree lead to dead-ends, as happens with $(< x_1, blue >, < x_2, green >, < x_3, red >)$ in Figure 5.5a, the node is termed an *internal dead-end*. Algorithm *graph-based backjumping* , described next, implements jumps at internal dead-ends as well as at leaf dead-ends.

6.3 Graph-based backjumping

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 have 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 to a minimal conflict set. 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 the 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: it assumes that the subset of variables connected to x_{i+1} is a minimal conflict set of x_{i+1}. Under this assumption, the latest variable in the ordering that precedes x_{i+1} and is connected to x_{i+1} is the culprit variable.

The importance of graph-based backjumping is that studying algorithms with performance tied to the constraint graph leads to graph-theoretic complexity bounds and thus to graph-based heuristics aimed at reducing these bounds. Such bounds are also applicable to algorithms that use refined run-time information such as Gaschnig's backjumping

```
procedure GRAPH-BASED-BACKJUMPING
Input: A constraint network 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 6.3.1 in text)
    i ← 1                            (initialize variable counter)
    D'_i ← D_i                       (copy domain)
    I_i ← anc(x_i)                   (copy of anc() that can change)
    while 1 ≤ i ≤ n
       instantiate x_i ← SELECTVALUE
       if x_i is null                (no value was returned)
          iprev ← i
          i ←latest index in I_i        (backjump)
          I_i ← I_i ∪ I_iprev − {x_i}
       else
          i ← i + 1
          D'_i ← D_i
          I_i ← anc(x_i)
    end while
    if i = 0
       return "inconsistent"
    else
       return instantiated values of {x_1, ..., x_n}
end procedure

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

Figure 6.4: The graph-based backjumping algorithm.

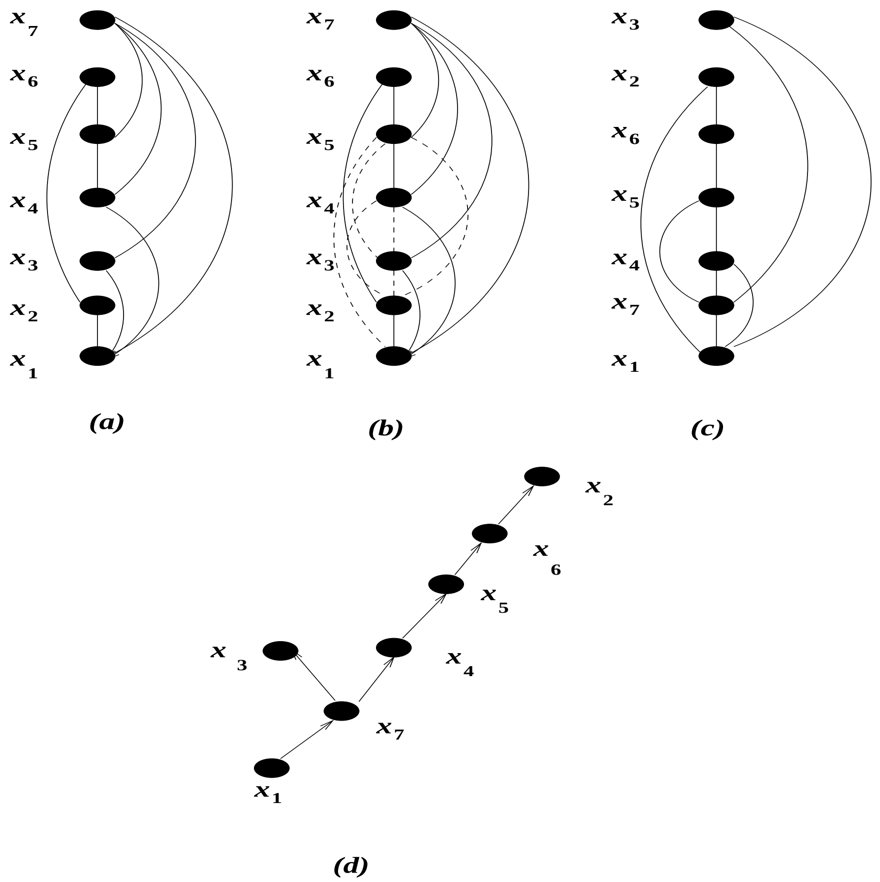


Figure 6.5: Several ordered constraint graphs of the problem in Figure 6.1: (a) along ordering $d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$, (b) the induced graph along d_1, (c) along ordering $d_2 = (x_1, x_7, x_4, x_5, x_6, x_2, x_3)$, and (d) a DFS spanning tree along ordering d_2.

and conflict-directed backjumping (described in a subsequent section).

Througout this chapter we will introduce graph terminology as it become relevant to the discussion.

Definition 6.3.1 (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 leaf dead-end, we equate $anc(\vec{a_i})$ with $anc(x_{i+1})$, and $p(\vec{a_i})$ with $p(x_{i+1})$.*

Example 6.3.2 Consider the ordered graph in Figure 6.5a along the ordering $d_1 = (x_1, ..., x_7)$. In this example, $anc(x_7) = \{x_1, x_3, x_4, x_5\}$ and $p(x_7) = x_5$. The parent of the leaf dead-end $\vec{a_6} = (blue, green, red, red, blue, red)$ is x_5, which is the parent of x_7. □

It is easy to show that if $\vec{a_i}$ is a leaf dead-end, $p(\vec{a_i})$ is safe. Moreover, if only graph-based information is utilized 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 $\vec{a_i}$, however, it may not be safe to jump to its parent $p(\vec{a_i})$ as the next example demonstrates.

Example 6.3.3 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 if we change the value of x_3, perhaps we could undo the dead-end at x_7 that started this latest retreat. If, however, the dead-end variable that initiated this latest retreat was x_6, it would be safe to jump as far back as x_2 upon encountering an internal dead-end at x_4. □

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

Definition 6.3.4 (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 invisiting 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, then the* current session *of x_i is the set of variables processed by the algorithm since the most recent* invisit *to x_i.*

Definition 6.3.5 (induced ancestors, induced parent) *Let x_i be a variable that is an internal dead-end. Let Y be a subset of the variables consisting of all the deadends in the current session of x_i. We denote $anc(Y) = \cup_{y \in Y} anc(y)$. The* induced ancestor set *of x_i relative to Y, $I_i(Y)$, is the union of all Y's ancestors, restricted to variables that precede x_i. Formally, $I_i(Y) = anc(Y) \cap \{x_1, ..., x_{i-1}\}$. The induced parent of x_i relative to Y, $P_i(Y)$, is the latest variable in $I_i(Y)$.*

Theorem 6.3.6 *Let $\vec{a}_i$ be a dead-end (leaf or internal), and let Y be the set of dead-end variables (leaf or internal) in the current session of x_i. If only graph information is used, $x_j = P_i(Y)$ is the earliest for which it is safe to jump for graph-based backjumping.*

Proof: By definition of x_j, all the variables between x_j and x_{i+1} do not participate in any constraint with any of the dead-end variables Y in x_i's current session. Consequently, any change of value to any of these variables will not perturb any of the nogoods that caused the dead-end in $\vec{a}_i$, and so they can be skipped.

We next show that if the algorithm jumped to a variable *earlier* than x_j, some solutions might be skipped. Let y_i be the first dead-end variable in Y that is connected to x_j. We argue that there is no way, based on graph information only, to rule out the possibility that there exists an alternative value of x_j that may lead to a solution. Let x_i be assigned the value a_i at the moment a dead-end at y_i occurred. Variable y_i is either a leaf dead-end or an internal dead-end. If y_i is a leaf dead-end, and since x_j is an ancestor of y_i, there might be a constraint R, whose scope S contains x_j and y_i, such that the current assignment $\vec{a}_i$ restricted to S cannot be extended to a legal value of y_i. Clearly, had the value of x_j been changed, the current assignment may be extendible to a legal tuple and the dead-end at y_i may have not occurred. Therefore the possibility of a solution with an alternative value to x_j was not ruled out. In the case that y_i is an internal dead-end, it means that there were no values of y_i that were both consistent with $\vec{a}_j$ and that could be extended to a solution. It is not ruled out (when using the graph only), however, that different values of x_j, if attempted, could permit new values of y_i for which a solution might exist. □

Example 6.3.7 Consider again the ordered graph in Figure 6.5a, and let x_4 be a dead-end variable. If x_4 is a leaf dead-end, then $Y = \{\}$, and x_1 is the sole member in its induced ancestor set $I_4(Y)$. The algorithm may jump safely to x_1. If x_4 is an internal dead-end with $Y = \{x_5, x_6\}$, the induced ancestor set of x_4 is $I_4(\{x_5, x_6\}) = \{x_1, x_2\}$, and the algorithm can safely jump to x_2. However, if $Y = \{x_5, x_7\}$, the corresponding induced parent set $I_4(\{x_5, x_7\}) = \{x_1, x_3\}$, and upon encountering a dead-end at x_4, the algorithm should retract to x_3. If x_3 is also an internal dead-end the algorithm retracts to x_1 since $I_3(\{x_4, x_5, x_7\}) = \{x_1\}$. If, however, $Y = \{x_5, x_6, x_7\}$, when a dead-end at x_4 is encountered (we could have a dead-end at x_7, jump back to x_5, go forward and jump back again at x_6, and another jump at x_5) then $I_4(\{x_5, x_6, x_7\}) = \{x_1, x_2, x_3\}$, the algorithm retracts to x_3, and if it is a dead-end it will retract further to x_2, since $I_3(\{x_4, x_5, x_6, x_7\}) = \{x_1, x_2\}$. □

Algorithm GRAPH-BASED-BACKJUMPING in Figure 6.4 implements the principles of graph-based backjumping. It jumps back as far as as graph-based information allows at both leaf and internal dead-ends. For each variable x_i, the algorithm maintains x_i's induced ancestor set I_i relative to the dead-ends in x_i's current session.

6.4 Conflict-directed backjumping

The two ideas, jumping back to a variable that, *as instantiated*, is in conflict with the current variable, and jumping back at internal dead-ends, can be integrated into a single algorithm, the *conflict-directed backjumping* algorithm. This algorithm uses the scheme we have outlined for graph-based backjumping but, rather than relying on graph information, exploits information gathered during search. For each variable, the algorithm maintains an induced *jumpback set*. Given a dead-end tuple $\vec{a}_i$, we define next the *jumpback set* of $\vec{a}_i$ (or of x_{i+1}) as the variables participating in $\vec{a}_i$'s *earliest minimal conflict set*. We first define an ordering between constraints. Let *scope*(R) denote the scope of constraint R.

Definition 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 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 of a dead-end $\vec{a}_i$ is defined to include the var-emc$(\vec{a}_j)$ of all the dead-ends $\vec{a}_j$, $j \geq i$, that occurred in the current session of x_i. Formally, $J_i = \bigcup\{$var-emc$(\vec{a}_j) \mid \vec{a}_j$ is a dead-end in x_i's session$\}$*

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

Proposition 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 nogoods that were active in causing this dead-end, and therefore the same dead-end will recur. To show that we cannot jump back any earlier than x_j we need to show that because we generate the $var - emc$ set by looking at earliest constraints first, it is not possible that there exists an alternative set of constraints for which $\vec{a_i}$ is a dead-end and for which the jumpback set yields an earlier culprit variable. Therefore, it is possible that changing the value of x_j will yield a solution, and this solution could be missed if we jumped to an earlier variable. □

Algorithm CONFLICT-DIRECTED-BACKJUMPING is presented in Figure 6.6. It computes the jumpback sets for each variable, and uses them to determine the variable to which it returns after a dead-end. Therefore,

Proposition 6.4.5 *Algorithm conflict-directed backjumping jumps back to the latest variable in the dead-end's jumpback set, and is therefore safe and maximal.* □.

Example 6.4.6 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 show 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 ← 1                          (initialize variable counter)
   D'_i ← D_i                     (copy domain)
   J_i ← ∅                        (initialize conflict set)
   while 1 ≤ i ≤ n
      instantiate x_i ← selectValue-cbj
      if x_i is null              (no value was returned)
         iprev ← i
         i ← index of last variable in J_i     (backjump)
         J_i ← J_i ∪ J_iprev − {x_i}(merge conflict sets)
      else
         i ← i + 1                (step forward)
         D'_i ← D_i               (reset mutable domain)
         J_i ← ∅                  (reset conflict set)
   end while
   if i = 0
      return "inconsistent"
   else
      return instantiated values of {x_1, ..., x_n}
end procedure

subprocedure selectValue-cbj

   while D'_i is not empty
      select an arbitrary element a ∈ D'_i, and remove a from D'_i
      consistent ← true
      k ← 1
      while k < i and consistent
         if consistent(ā_k, x_i = a)
            k ← k + 1
         else
            let R_S be the earliest constraint causing the conflict
            add the variables in R_S's scope S, but not x_i, to J_i
            consistent ← false
      end while
      if consistent
         return a
   end while
   return null                    (no consistent value)
end procedure
```

Figure 6.6: 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 the variable to jump to. Given a graph, a depth-first search (*DFS*) ordering is one that is generated by a *DFS* traversal of the graph. This traversal ordering results also in a *DFS spanning tree* of the graph which includes all and only the arcs in the graph that were traversed in a forward manner. The depth of a *DFS* spanning tree is the number of levels in that tree created by the *DFS* traversal (see [48]). 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 earliest safe graph-based back-jump destination requires following a very simple rule: if a dead-end (leaf or internal) occurs at variable x, go back to the *DFS* parent of x.

Example 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 6.5c,d. If a dead-end occurs at node x_3, the algorithm retreats to its *DFS* parent, which is x_7. □

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 all the induced ancestors of x. Clearly, $f(x)$ always appear in the induced ancestor set of x since it is connected to x and since it precedes x in the ordering. It is also the latest one since all the variables that appear in x's session must be its descendents in the *DFS* subtree rooted at x. Let y be a dead-end variable in the session of x. Because of properties of a DFS tree of any graph, ensuring that the only non-tree arcs are back-arcs, connecting a node to one of its ancestors in the tree [48], y's ancestors that precede x must therefore lie on the path from the root to x and therefore they either coincide with $f(x)$, or appear before $f(x)$. □

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

Theorem 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 (measure 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 i. Let T_i stands for the maximal number of nodes in the search-tree rooted at x_i, namely, T_i is the maximum number of nodes visited in any session of x_i. Since any assignment of a value to x_i generates at most b subtrees of depth $i-1$ or less that can be solved independently, 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 time complexity of graph-based backjumping is $O(b^m k^{m+1})$. Notice that when the tree is balanced (namely, each internal node has exactly two child nodes) the bound can be improved to $T_m = O((n/b)k^{m+1})$, since $n = O(b^{m+1})$. $\Box$

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

It can be shown that *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. Namely, the induced-parent is always the parent in that DFS tree of the induced-graph. The algorithm is bounded exponentially by the depth of such trees as well.

Proposition 6.5.4 *Let d be a DFS order of an induced-graph of a given constraint problem, than the induced-parent of backjumping at each deadend is its parent in the DFS tree.*

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 $O(\exp(m_d^*))$.*

A proof, that uses somewhat different terminology and derivation, is given in [132]. The virtue of Theorem 6.5.5 is in allowing a larger set of orderings, each yielding a bound on backjumping's performance as a function of its *DFS* tree-depth, to be considered. Since every *DFS* ordering of G is also a *DFS* ordering of its induced graph along d, (G^*, d) (the added induced arcs are back arcs of the *DFS* tree), *DFS* orderings of G are a subset of all *DFS* orderings of all of G's induced graphs.

Example 6.5.6 Consider the graph G having the following arcs:
$\{(1,2)(2,3)(3,4)(4,1)(1,7)(7,5)(5,6)(1,6)\}$ displayed in Figure 6.7. Ordering $d_1 = 1,2,3,4,7,5,6$

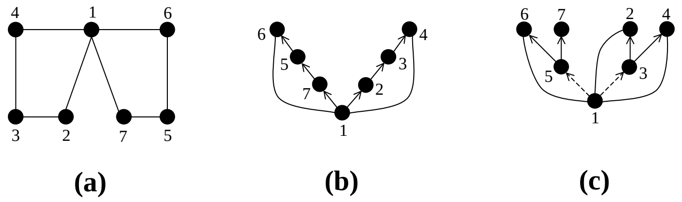


Figure 6.7: A graph (a), its DFS ordering (b), a DFS ordering of its induced-graph (c)

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 2 only. Consequently, graph-based backjumping is $O(exp(4)$ along d_1 and $O(exp(3))$ along d. □

6.5.1 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 define a set of parameterized backjumping algorithms, called *i-backjumping* algorithms, where i indexes the number of variables consulted in the forward phase. All algorithms jump back maximally at both leaf and internal dead-ends, as follows. Given an ordering of the variables, instantiate them one at a time as does conflict-directed backjumping; note that conflict-directed backjumping is *1-backjumping*. However, when selecting a new value for the next variable, make sure the new value is both consistent with past instantiation, and consistently extendible by the next $i-1$ variables (which will require revising CONSISTENT procedure). This computation will be performed at every node and can be exploited to generate more refined conflict sets than in 1-backjumping, namely, conflict sets that conflict with i future variables. This leads to the concept of *level-i conflict sets*. A tuple $\vec{a}_j$ is a level-i conflict set if it is not consistently 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 done in 1-backjumping. The balance between computation overhead at each node and the savings on node generation should, of course, be studied. (For elaboration see exercises).

6.6 Learning Algorithms

The earliest minimal conflict set of Definition 6.4.2 is a nogood explicated during search and used to guide backjumping. However, this same nogood may be rediscovered again and again as the algorithm explores different paths in the search space. By making this nogood explicit, in the form of a new constraint, we can make sure that the algorithm will not rediscover it, since it will be available for consistency testing. Doing so may prune the remaining search space. This technique, called constraint recording or *learning*, is behind 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 at this stage the conflict set $\vec{a}_i$ itself as a constraint, because under the backtracking control strategy the current state will not recur.[1] 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, namely, minimal. As noted above, one obvious candidate is the earliest minimal conflict set, which is identified anyway for conflict-directed backjumping. Alternatively, if only graph information is used, the graph-based conflict set could be identified and recorded. Another (extreme) option is to learn and record *all* the minimal conflict sets associated with the current dead-end.

In learning algorithms, the savings from possibly pruning search by finding out earlier 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 records only the minimal conflict sets. Shallow learning allows non-minimal conflict sets to be recorded as well. Learning algorithms may also be characterized by how they bound the arity of the constraints recorded. Constraints involving many variables are less frequently applicable, require additional memory to store, and are more expensive to consult than constraints having fewer variables. The algorithm may record a single nogood or multiple nogoods per dead-

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

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

end, and it may allow learning at leaf dead-ends only or at internal dead-ends as well.

We present three types of learning: graph-based learning, deep learning, and jump-back 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.

6.6.1 Graph-based learning

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

Example 6.6.1 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 6.8b presents the search space explicated by naive backtracking and by backtracking augmented with graph-based learning. Branches below the cut lines in Figure 6.8 are generated by the former but not by the latter. Leaf dead-ends are numbered (1) through (10) (only dead-end that appear in the search with learning are numbered). At each dead-end, search with learning can record a new constraint. At dead-end (1), no consistent value exists for x_1. The ancestor set of x_1 is $\{x_2, x_3, x_4, x_7\}$, so graph-based learning records the nogood $(x_2 = \mathit{green}, x_3 = \mathit{blue}, x_4 = \mathit{blue}, x_7 = \mathit{red})$. This nogood reappears later in the search, under the subtree rooted at $x_6 = \mathit{teal}$, and it can be used to prune the search at the dead-end numbered (9). The dead-ends labeled (2) and (4) occur because no consistent value is found for x_7, which has the ancestor set $\{x_3, x_4\}$. The nogoods $(x_3 = \mathit{blue}, x_4 = \mathit{red})$ and $(x_3 = \mathit{red}, x_4 = \mathit{blue})$ are therefore recorded by graph-based learning, in effect creating an "equality" constraint between x_3 and x_4. (The dead-ends at (3) and (5) involve the same nogoods; if graph-based backjumping is used instead of backtracking, these dead-ends will be avoided.) This learned constraint prunes the remaining search four times. The following additional nogoods are recorded by graph-based learning at the indicated dead-ends: (6) with deadend variable x_1: $(x_2 = \mathit{green}, x_3 = \mathit{red}, x_4 = \mathit{red}, x_7 = \mathit{blue})$; (7) with deadend variable x_5: $(x_4 = \mathit{blue}, x_6 = \mathit{green}, x_7 = \mathit{red})$; (8), deadend variable x_5: $(x_4 = \mathit{red}, x_6 = \mathit{green}, x_7 = \mathit{blue})$; (9), deadend variable x_7: $(x_2 = \mathit{green}, x_3 = \mathit{blue}, x_4 = \mathit{blue})$; (10), deadend variable x_7: $(x_2 = \mathit{green}, x_3 = \mathit{red}, x_4 = \mathit{red})$. 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 from Fig. 6.4 after a dead-end is encountered shown in Figure 6.4.

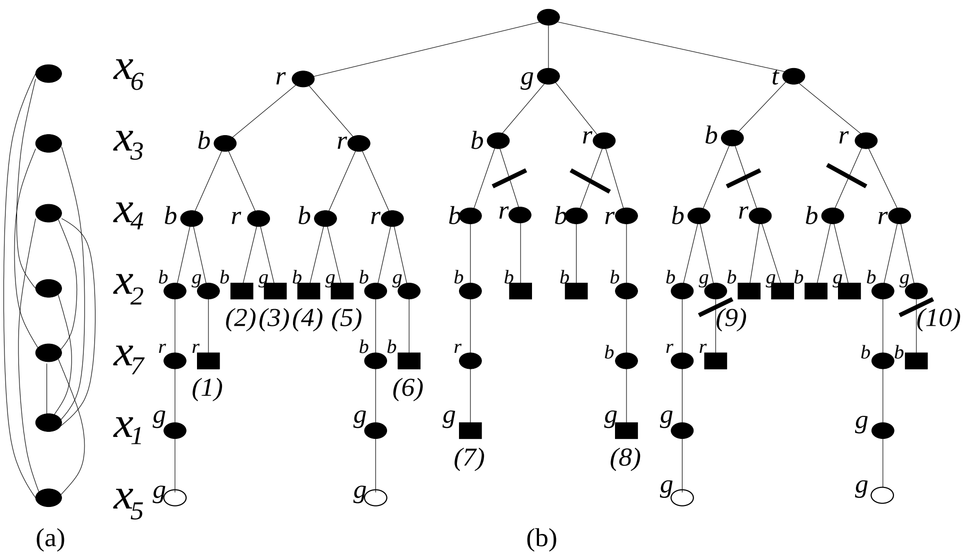


Figure 6.8: The search space explicated by backtracking on the CSP from Figure 8.16, using the variable ordering $(x_6, x_3, x_4, x_2, x_7, x_1, x_5)$ and the value ordering (*blue, red, green, teal*). Part (a) shows the constraint graph; part (b) illustrates the search space. The cut lines in (b) indicate branches not explored when graph-based learning is used.

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. 6.4)

Figure 6.9: 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 value 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.

6.6.2 Deep learning

Identifying and recording only minimal conflict sets constitutes *deep learning*. Discovering all minimal conflict sets means acquiring all the possible information out of a dead-end. For the problem and ordering of Example 6.6.1 at the first dead-end, deep learning will record the minimal conflict set $(x_2 = \text{green}, x_3 = blue, x_7 = red)$ (or perhaps $(x_2 = green, x_4 = blue, x_7 = red)$, or both), instead of the non-minimal conflict set including both x_3 and x_4 that is recorded by graph-based learning. Although deep learning is the most informative, its cost is prohibitive if we want all minimal conflict sets, and in the worst case, exponential in the size of the initial conflict set. If r is the cardinality of the graph-based conflict set, we can envision a worst case where all the subsets of size $r/2$ are minimal conflict sets of the dead-end variable. The number of such minimal conflict sets may be $\binom{r}{r/2} \cong 2^r$, which amounts to exponential time and space complexity at each dead-end. Discovering *all* minimal conflict sets can be implemented by enumeration: first, recognize all conflict sets of one element; then, all those of two elements; and so on.

In general, graph-based learning records the largest size constraints, and deep learning records the smallest ones. As noted for backjumping, the virtues of graph-based learning are mainly theoretical (see Section 6.6.5); using this algorithm in practice is not advocated since jumpback learning, described next, is always superior. Neither the use of deep learning is recommended, because its cost is usually prohibitive.

6.6.3 Jumpback learning

To avoid the explosion in time and space of full deep learning one may settle for identifying just one conflict set, minimal relative to prefix conflict sets. The obvious candidate is the *jumpback set* for leaf and internal dead-ends as it was explicated by conflict-directed backjumping. *Jumpback learning* 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 that of storing the conflict set. As with graph-based learning, the modification required to augment CONFLICT-DIRECTED-BACKJUMPING with a learning algorithm is minor: specifying that the conflict set is recorded as a nogood after each dead-end as described in Figure 6.10.

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. 6.6)

Figure 6.10: Conflict-directed bakjump-learning, modifying CBJ

Example 6.6.2 For the problem and ordering of Example 6.6.1 at the first dead-end, jumpback learning will record the nogood ($x_2 = green, x_3 = blue, x_7 = red$), since that tuple includes the variables in the jumpback set of x_1. □

6.6.4 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.

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

Definition 6.6.3 (i-relevant) *A nogood is i-relevant if it differs from the current partial assignment by at most i variable-value pairs.*

Definition 6.6.4 (i'th order relevance bounded learning) *An i'th order relevance bounded learning scheme maintains only those learned nogoods that are i-relevant.*

6.6.5 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 6.6.5 *Let d be an ordering of a constraint graph, and let $w^*(d)$ be its induced width. Any backtracking algorithm using ordering d with graph-based learning has a space complexity of $O((nk)^{w^*(d)+1})$ and a time complexity of $O((2nk)^{w^*(d)+1})$, where n is the number of variables and k bounds the domain sizes.*

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

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

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

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 [10] that for any graph $m_d^* \leq \log n \cdot w^*(d)$. Therefore, to reduce the time bound of graph-based backjumping by a factor of $\log n$, we need to invest $O(exp(w^*(d))$ in space, augmenting backjumping with learning.

6.7 Integration and comparison of algorithms

6.7.1 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 backjumping variants. Additionally, a combined algorithm can employ learning, and the dynamic variable and value ordering heuristics based 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 6.11 and 6.12.

The main procedure of FC-CBJ closely resembles CONFLICT-DIRECTED-BACKJUMPING (Figure 6.6). 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 step forward. These calls could be removed and the algorithm would revert without other modification to a static variable ordering. But given that look-ahead is being performed for the purposes of rejecting inconsistent values, there is little additional cost in performing SELECTVARIABLE, and in practice this heuristic has been found very effective in reducing the size of the search space.

Apparently, there is some 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 the simplest form of backjumping.

Proposition 6.7.1 *When using the same variable ordering, Gaschnig's backjumping always explores every node explored by forward-checking.* □.

6.7.2 Comparison of algorithms

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

Figure 6.13 shows the relationships between several algorithms discussed in this paper, based on worst case performance measured by the size of the search space (which is equivalent to the number of calls to a SELECTVALUE subprocedure).

Figure 6.14 summarizes results of experimental comparisons of several backtracking based constraint 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: "BT" refers to backtracking with f with fcc; "BT+AC" refers to backtracking with arc-consistency enforced after each instantiation; "CBJ" refers to conflict-directed backjumping; "CBJ+LVO" adds a value

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 ← 1                              (initialize variable counter)
call selectVariable                (determine first variable)
D'_i ← D_i for 1 ≤ i ≤ n           (copy all domains)
J_i ← ∅                            (initialize conflict set)
while 1 ≤ i ≤ n
   instantiate x_i ← selectValue-fc-cbj
   if x_i is null                  (no value was returned)
      iprev ← i
      i ← latest index in J_i      (backjump)
      J_i ← J_i ∪ J_iprev − {x_i}
      reset each D'_k, k > i, to its value before x_i was last instantiated
   else
      i ← i + 1                    (step forward)
      call selectVariable          (determine next variable)
      D'_i ← D_i
      J_i ← ∅
end while
if i = 0
   return "inconsistent"
else
   return instantiated values of {x_1, ..., x_n}
end procedure
```

Figure 6.11: The main procedure of the FC-CBJ algorithm.

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-domain $\leftarrow$ *false*
  for all k, $i < k \leq n$
    for all values b in D'_k
      if not CONSISTENT($\vec{a}_{i-1}, x_i = a, x_k = b$)
        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-domain $\leftarrow$ *true*
  endfor
  if *empty-domain*   (don't select a)
    reset each D'_k and $j_k, i < k \leq n$, to status before a was selected
  else
    return a
end while
return null   (no consistent value)
end subprocedure

Figure 6.12: The SelectValue subprocedure for FC-CBJ.

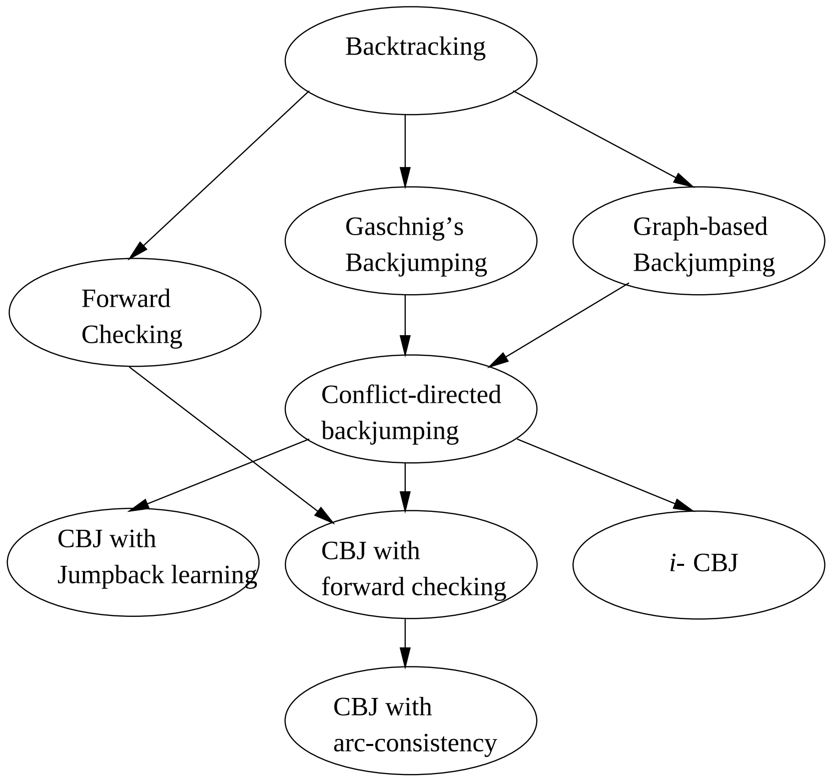


Figure 6.13: The relationships of selected backtracking based algorithms. CBJ is an abbreviation for conflict-directed backjumping. An arrow from A to B indicates that on the same problem and with the same variable and value orderings, the nodes in A's search tree will contain the nodes in B's.

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

Figure 6.14: 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.

ordering heuristic called LVO; "CBJ+LRN" is CBJ plus 4th-order jumpback learning; "CBJ+LRN+LVO" is CBJ with both LVO and learning. The columns labeled "Set 1" through "Set 3" report averages from 2000 randomly generated binary CSP instances. All instances had variables with three element value domains, and the number of constraints was selected to generate approximately 50% solvable problems. The number of variables and the number of valid tuples per constraint was: Set 1: 200 and 8; Set 2: 300 and 7; Set 3: 350 and 6. Algorithm BT was not run on sets 1 and 2. The rightmost two columns of the figure show results on two specific problems from the Second Dimacs Implementation Challenge [82]: "ssa7552-038" and "ssa7552-158." For more details about the experiments, see [60].

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

The empirical results shown in Figure 6.14 are only examples of typical experimental comparisons of algorithms. Unfortunately, from these and similar studies it is not necessarily possible to conclude how the algorithms will perform on all problems having different structural properties.

6.8 Summary

The chapter describe thye primary look-back methods, backjumping and learning for improving backtracking. Both graph-based approaches for backjumping and learning, as well as constraint-based approaches are described and analyzed. It is shown 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. It is also shown that the time and space complexity of learning during search can be bounded exponentially by the induced width. Some discussion of algorithms that combine look-ahead and look-back improvements are discussed and a comparison of the various algorithms using search-space size is given.

6.9 Chapter notes

The truth-maintenance systems area was the earliest to contribute to the look-back aspect of backtracking. Stallman and Sussman [151] 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 [111]. Their work was followed by Gaschnig [63] who introduced the backjumping algorithm and also coined the name. Researchers in the logic-programming community have tried to improve a backtracking algorithm used for interpreting logic programs. Their improvements, known under the umbrella name *intelligent backtracking*, focused on a limited amount of backjumping and constraint recording [17].

Later, following the introduction of graph-based methods [51, 42] Dechter [36] described the graph-based variant of backjumping, which was followed by Prosser's conflict-directed backjumping [130]. She also introduced learning nogoods into backtracking including graph-based learning [36]. Dechter and Pearl [42] developed the induced-width bound on learning algorithms. Frueder and Quinn [52] noted the dependence of backjumping's performance on the depth of the DFS tree of the constraint graph, and Bayardo and Mirankar [132] 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 $\log n$ at an exponential space cost in the induced-width [132]. *Dynamic backtracking* algorithm [67] employs a similar notion of keeping only learned nogoods that are most likely to be consulted in the near future search. The usefullness of the DFS ordering for distributed execution of backjumping was shown [170].

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 [36] evaluated combinations of graph-based backjumping, graph-based learning, and the cycle-cutset scheme. An evaluation of hybrid schemes was carried out by Prosser [130], who combined known look-ahead and look-back methods and ranked each combination based on average performance, primarily, Zebra problems. Dechter and Meiri [41] have evaluated the effect of pre-processing consistency algorithms on backtracking and backjumping.

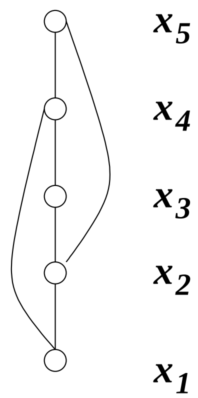
Before 1993, most of the empirical testing was done on relatively small problems (up to 25 variables), and the prevalent conclusion was that only low-overhead methods are cost effective. With improvements in hardware and recognition that empirical evaluation may be the best way to compare the various schemes has come a substantial increase in empirical testing. After Cheeseman, Kanefsky, and Taylor [21] 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 [57, 56, 58, 67, 27, 10, 8]. 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 [7], 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 [88].

The value of look-back improvements for solving propositional satisfiability was largely overlooked, where most algorithms focused on look-ahead improvements of DPLL [27]. However this was changed with the recent experimental work by Bayardo and Schrag [83].

Recently, constraint processing techniques were 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 usefull constraint techniques that those languages include is the use of various levels of arc-consistency in look-ahead search [163, 75]. It remains to be see whether higher level of look-ahead should be incorporated, as well, weather various styles of look-back methods can be include in a cost-effective manner.

6.10 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 $d_1 = (x_1, x_2, x_3, x_4, x_5)$, confirm, reject, and justify your answer:

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

3. 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 of each node in the DFS tree. (b) Prove that backjumping along d is bounded exponentially by m_d.

4. Describe algorithms backjump and graph-based learning for CNF propositional theories. You can write two separate algorithms or incorporate both ideas into one algorithm.

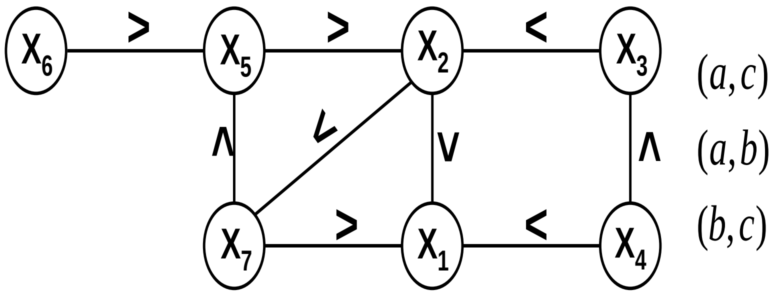


Figure 6.15: A constraint graph

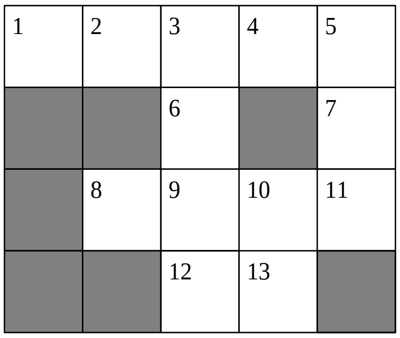


Figure 6.16: A crossword puzzle

5. Consider the crossword puzzle in Figure 6.16 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.

6. Consider the graph in Figure 6.15. Provide a DFS ordering of the graph and bound the complexity of solving any problem having that graph.

7. Define algorithm i-backjumping precisely by defining its i-CONSISTENT procedure.

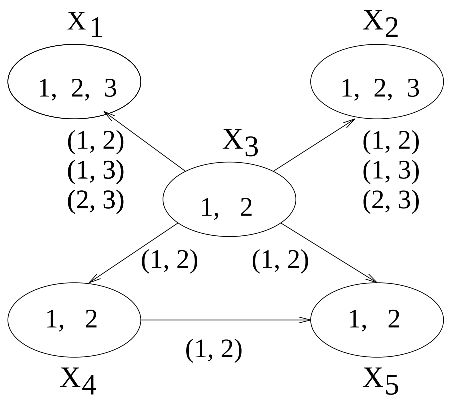


Figure 6.17: A small CSP. The constraints are: $x_3 < x_1, x_3 < x_2,\ x_3 < x_5,\ x_3 < x_4,\ x_4 < x_5$. The allowed pairs are shown on each arc.

8. Suppose we try to solve the problem in Figure 6.17 along the ordering $d = (x_1, x_2, x_3, x_4, x_5)$.
   (a) Show the trace and nogoods recorded by graph-based learning.
   (b) Discuss the performance of deep learning on that problem
   (c) Show how jumpback learning will behave upon a dead-end at x_5.
9. Prove the correctness of FC-CBJ
10. Prove that [89] When using the same variable ordering, Gaschnig's backjumping always explores every node explored by forward-checking.