Chapter 5

General search strategies: Look-ahead

Once a problem of interest has been formulated as a constraint satisfaction problem, it can be attacked with a general purpose constraint algorithm. Most CSP algorithms are based on the two principles of search and deduction, and more sophisticated algorithms often combine both principles. The term *search* is used to characterize a large category of algorithms which solve problems by guessing the next operation to perform, possibly with the aid of a heuristic. A good guess results in a new state that is nearer to a goal. If the operation does not result in progress towards the goal (which may not be apparent until later in the search), then the operation can be retracted and another operation can be tried.

For CSPs, search is exemplified by the backtracking algorithm. The algorithm was first described more than a century ago and has since then been reintroduced several times [16]. Backtracking search for CSPs uses the operation of assigning a value to a variable, so that the current partial solution is extended. 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. 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 solutions, 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 *expanded* search space, which is determined both by the size of the *underlying* search space, and by the algorithm's control strategy. The size of the underlying search space depends on the way in which the constraints are represented (e.g. on the level of local consistency), and on the order of variable instantiation. Using these factors, researchers have developed improvement procedures of two types: those employed *before* performing the search that bound the size of the underlying search space, and those used dynamically *during* the search that decide which parts of the search space will not be visited. Commonly used preprocessing techniques are arc- and path-consistency algorithms and heuristic approaches for determining the variable ordering

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 improvement schemes to 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 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 be understood as a traversal algorithm looking for a solution path in a search graph. For details about search spaces and general search algorithms [123, 126].

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 transform states to states. Thus, an operator takes a partial solution $(< x_1, a_1 >, \ldots, < x_j, a_j >)$, $0 \leq j \leq n$, and augments 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.

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 which requires that the value assigned to z evenly divides the values for 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 denotes intermediary states, and black boxed denote dead-ends. The hollow boxes connected by broken lines represent illegal states that correspond to instantiation attempts that fail. These illegal states will sometimes be depicted in the search graph since they express problem-solving activity. (In fact, some search space definitions include these leaf nodes as legal states in the search space.) □

We next 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 can have different search spaces depending on the variable ordering. We can see that the search graph for ordering d_1 includes 20 legal states, whereas the search graph for ordering d_2 includes 48 legal states. (Remember that all the hollow nodes are illegal 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, whereas the search graph for ordering d_2 has 18 dead-end leaves.

When variable ordering is unrestricted, the choice of variable ordering is left to the search algorithm rather than fixed in advance, and the search space should include all possible orderings of variables. The unordered search space may at first seem undesirable since it is so large and may not seem able 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; it shrinks if we change the representation of a problem to an equivalent, but tighter set of constraints.

Example 5.1.2 Consider again the constraint network of Example 5.1.1. As given, the network is not arc-consistent. For instance, the value 5 of z does not evenly divide any of x's values. Applying arc-consistency to the network results in the deletion of the value 5 from the domains of z and l. Consequently, for the resulting arc-consistent 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, arc-consistency eliminates nine dead-end leaves (all corresponding to $< l, 5 >$)

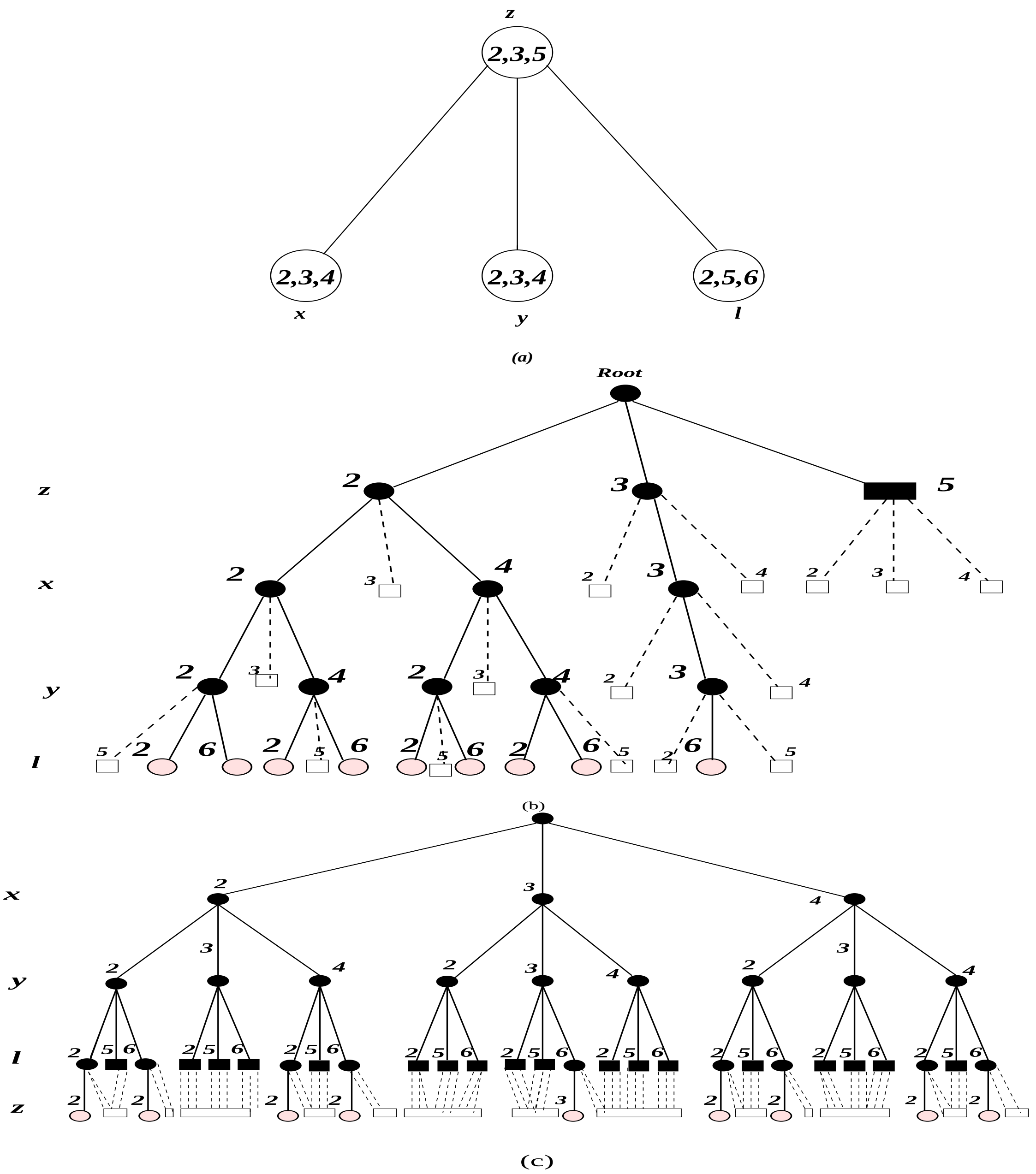


Figure 5.1: (a) A constraint graph, (b) its search space along ordering $d_1 = (z, x, y, l)$, and (c) its search space along ordering $d_2 = (x, y, l, z)$. Hollow nodes and bars in the search space graphs represent illegal states that may be considered, but will be rejected as states. Numbers next to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

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, a collection of binary constraints are introduced. We get a path-consistent network $\mathcal{R}'$ which includes the following set of constraints:

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

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 network tighter than $\mathcal{R}'$, where both represent the same set of solutions for a given constraint network. Then, for any ordering d, any path appearing in the search graph derived from $\mathcal{R}$ which leads to a solution 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 before searching for a solution. However, applying a local consistency algorithm is costly, and its cost may not always be offset by the smaller search space because tighter representation normally possesses many more explicit constraints than a looser one. As a result, it may require many more constraint checks (each testing if an 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; it may be $O(n^{r-1})$ when r bounds the constraints' arity. Generating the whole search space is often unnecessary, especially if we are seeking just one solution, since the expense of pre-processing (by arc- or path-consistency) is more likely to be recouped over the generation of all solutions.

Example 5.1.4 Consider again the network $\mathcal{R}$ in Example 5.1.1. 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 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

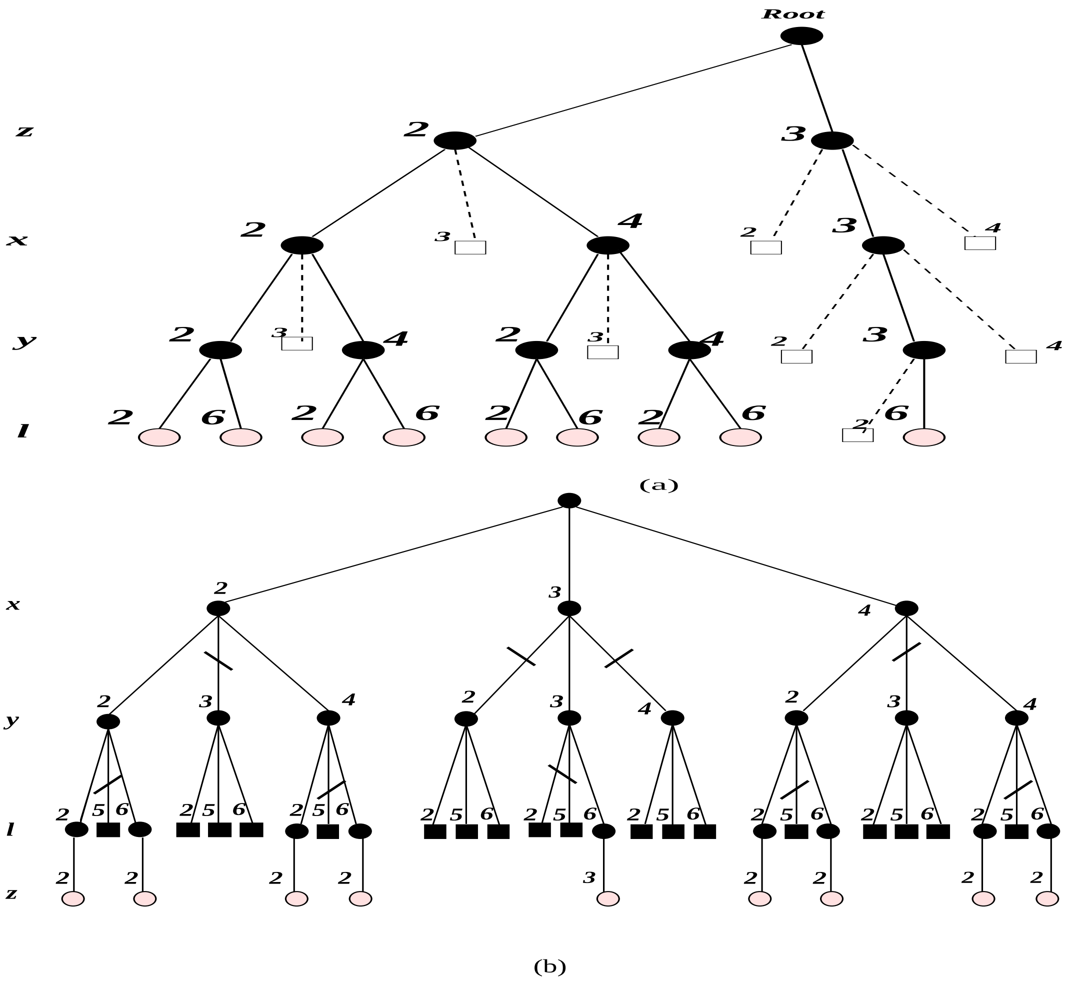


Figure 5.2: (a) Search space Example 5.1.1 with ordering d_1 after arc-consistency. (b) Search space for ordering d_2 with reduction effects from enforcing path-consistency marked with slashes.

checks (around 40) are required for $\mathcal{R}'$. When generating the search graph for $\mathcal{R}$ along ordering d_2, 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 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 tests before enforcing path-consistency is higher (around 80), while afterwards, it is reduced considerably (to about 50).

□

Finally, observe that following 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 algorithm that searches such a space is guaranteed to find a solution in linear time (provided that the cost of state generation is bounded). When this is the case, we call the search space *backtrack-free* which is redefined next.

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.*

We remind the following notation. We denote by $\vec{a_i}$ the subtuple of consecutive values $(a_1, ..., a_i)$ for a given ordering of the variables $x_1, ..., x_i$. We denote by $\vec{a}$ an arbitrary subtuple of values.

5.2 Backtracking

Backtracking search traverses the search space of partial instantiations in a depth-first manner. As mentioned, the algorithm maintains a partial solution that denotes a state in the search space. Backtracking has two phases: a forward phase in which the next variable is selected and the current partial solution is extended by assigning a consistent value, if one exists for the next variable; and a backward phase in which, when no consistent solution exists for the current variable, focus returns to the previous variable assigned. Figure 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. The D' sets are not needed if the values can be mapped to a contiguous set of integers that

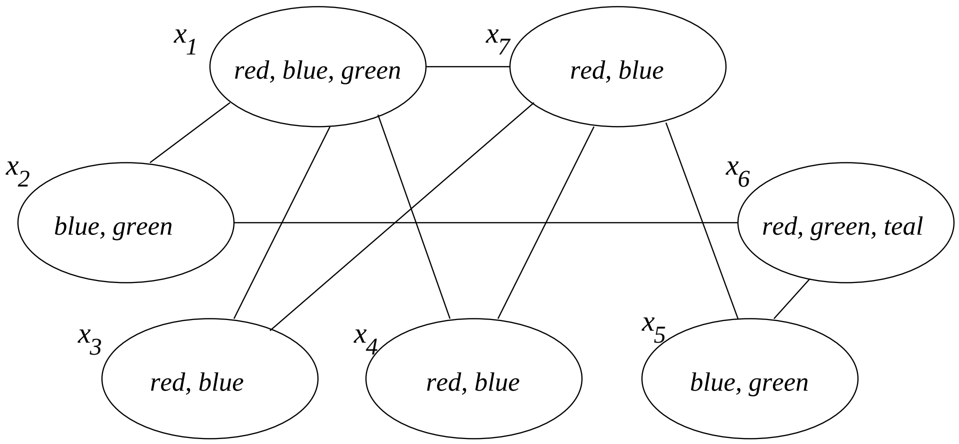


Figure 5.3: A coloring problem with variables $(x_1, x_2, \ldots, x_7)$. The domain of each variable is written inside the corresponding node. Each arc represents the constraint that the two variables it connects must be assigned different colors.

are always considered in ascending order; in this case a single integer can be used as a marker to divide values that have been considered from those that have not.

Example 5.2.1 Consider the coloring problem in Figure 5.3. Assume backtracking search for a solution using two possible orderings: $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$ and $d_2 = x_1, x_7, x_4, x_5, x_6, x_3, x_2$. The search spaces along orderings d_1 and d_2, as well as those portions explicated by backtracking from left to right, are depicted in Figure 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, $\vec{a}_{i-i}$. (Namely, this procedure describe the overhead associated with generating the next node in the search space.) SELECTVALUE, in turn, relies on the CONSISTENT subprocedure, which returns *true* only if current partial solution is consistent with the candidate assignment to the next variable. If SELECTVALUE succeeds in finding a value, BACKTRACKING proceeds to the next variable, x_{i+1}. If SELECTVALUE cannot find a consistent value for x_i, then a *dead-end* occurs, and BACKTRACKING looks for a new value for the previous variable, x_{i-1}. The algorithm terminates when all variables have assignments, or when it has proven that all values of x_1 do not lead to a solution, and thus that the problem is unsolvable. Our presentation of BACKTRACKING stops after a single solution has been found, but it could easily be modified to return all solutions, or a desired number.

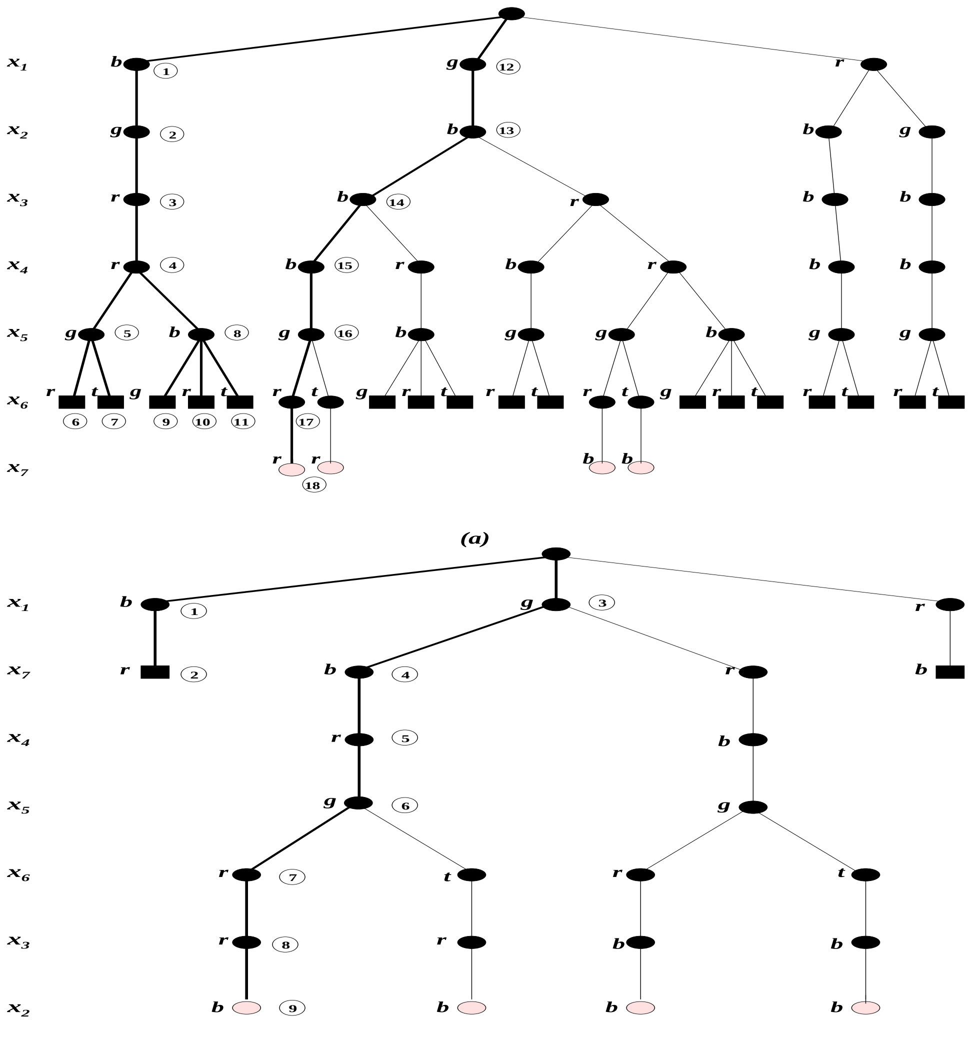
procedure BACKTRACKING
Input: A constraint network $P = (X, D, C)$.
Output: Either a solution, or notification that the network is inconsistent.

    $i \leftarrow 1$ (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.



(b)

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 empty ovals. The colors are considered in order (*blue, green, red, teal*), and are abbreviated by their first letters. Thick lines denote the portion of the search space explored by backtracking when stopping after the first solution. Circled numbers indicate the order in which nodes are expanded.

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.) Note that although, for sake of simplicity, our examples involve binary constraints, the description is completely general. CONSISTENT handles general binary and non-binary constraints; its implementation depends on how constraints are represented by the computer program. We next analyze its complexity.

5.2.1 Complexity of extending a partial solution

The complexity of CONSISTENT and of SELECTVALUE can be determined, based on the premise that the constraints are 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 variable x_i. Clearly, the ordering of contraint testing can matter in practice, 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, it can be practical to store the constraints as a table of boolean values, indexed by the two variables and their values. The tentative instantiation $< x_i, a >$ must then be checked with at most n earlier variables, effectively yielding $O(n)$ complexity for CONSISTENT and $O(nk)$ complexity for SELECTVALUE. If CONSISTENT performs computations other than table lookups, its complexity is of course dependent on the nature of these computations. In summary,

Proposition 5.2.2 *For general CSPs with constraints stored in tables, having n variables, e constraints, with constraint arity bounded by r, the number of tuples in a constraint bounded by t, and at most k values for a variable, 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*, namely, repeatedly rediscovering the same inconsistencies and the same partial successes during search. Efficient cures for such behavior in all cases are unlikely, since the problem is NP-hard.

As noted the performance of backtracking can be improved by preprocessing algorithms that reduce the size of the *underlying* search space, and by improving the algorithm's control strategy dynamically during search.

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

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

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

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

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

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

In the reminder of this chapter we focus on look-ahead algorithms, while in Chapter 6 we present look-back schemes. But before we proceede we conclude this section with one of the earliest proposals for improving the state generation cost, called *backmarking*, which does not quite fit as either a look-ahead or a look-back scheme.

Backmarking

Backmarking [64] is a caching-type algorithm that focuses on lowering the cost of node generation without trying to prune the search space itself.

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

Recall that a backtracking algorithm moves either forward or backward in the search space. Suppose that the current variable is x_i and that x_p is the earliest variable in the ordering which changed its value since the last visit to x_i. Clearly, any testing of constraints whose scope includes x_i and 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. Therefore, by maintaining the right information from earlier parts of the search, values in the domain of x_i can either be recognized immediately as inconsistent or else, the only constraints to be tested are those involving instantiations starting from x_p on.

Backmarking implements this idea by maintaining two new tables. First, for each variable x_i and for each of its values a_v, backmarking remembers the earliest prior variable x_p such that the current partial instantiation $\vec{a}_p$ conflicted with $x_i = a_v$. This information is maintained in a table with elements $M_{i,v}$. (Note that this assumes that constraints involving earlier variables are tested before those involving later variables.) If $x_i = a_v$ is consistent with all earlier partial instantiations $\vec{a}_j$, $j < i$, then $M_{i,v} = i$. The second table, with elements low_i, records the earliest variable that has changed value since the last time x_i was instantiated. This information is put to use at every step of node generation. If $M_{i,v}$ is less than low_i, then the algorithm knows that the variable pointed to by $M_{i,v}$ did not change and that $x_i = a_v$ will fail again when checked against $\vec{a}[M_{i,v}]$, so no further consistency checking is needed 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 6.10.

Example 5.2.3 Consider again the example in Figure 5.3, and assume backmarking uses ordering $d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$. Once the first dead-end is encountered 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. When trying to instantiate a new value for x_7, the algorithm notices that the M values are smaller than $low(7)$ for both values of x_7 (red, blue) and, consequently, both values can be determined to be inconsistent without performing any other consistency checks. □

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

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

Figure 5.6: The backmarking algorithm.

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

5.3 Look-ahead strategies

Look-ahead schemes are invoked whenever the algorithm is preparing to select the next variable or to assign a value to the next variable. The essence of these schemes is to discover from a restricted amount of consistency enforcing how the current decisions about variable and value selection will restrict future search. Once a certain amount of forward constraint propagation is complete, the algorithm can use the results to decide which *variable* to instantiate next, if the order is not predetermined, and to decide which *value* to assign to the next variable, when there is more than one candidate. Generally, it is advantageous to first instantiate variables that maximally constrain the rest of the search space. Normally, the most highly constrained variable (which is interpreted to be one having the least number of values in its domain) is selected. In addition, when searching for a single solution, an attempt is generally made to assign a value that maximizes the number of options available for future assignments.

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 provisionally removes those values. A more extensive look-ahead procedure may then note that x_3 and x_7 are connected and are now left with incompatible values; each variable has the domain $\{blue\}$ and the problem with $x_1 = red$ is therefore not arc-consistent. The implication is that assigning *red* to x_1 will inevitably lead to a dead-end, and thus this assignment 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 the two partial solutions $< x_1 = red >$ or $< x_1 = blue >$ will be determined as deadends, and the corresponding branches below these partial solutions will be pruned in Figure 5.5b. □

Look-ahead strategies incur an extra cost after each instantiation, but they can provide important benefits. First and formost, if *all* values of an uninstantiated variable are removed by the look-ahead procedure, then the current instantiation cannot be part of a solution and the algorithm must backtrack. Consequently, dead-ends occur earlier in the search, and much smaller portions of the search spaces are typically explored. In general, the stronger the level of constraint propagation being used for look-ahead, the smaller the

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

   D'_i ← D_i for 1 ≤ i ≤ n      (copy all domains)
   i ← 1                         (initialize variable counter)
   while 1 ≤ i ≤ n
      instantiate x_i ← SELECTVALUE-XXX
      if x_i is null             (no value was returned)
         i ← i − 1               (backtrack)
         reset each D'_k, k > i, to its value before x_i was last instantiated
      else
         i ← i + 1               (step forward)
   end while
   if i = 0
      return "inconsistent"
   else
      return instantiated values of {x_1, ..., 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 interleaves arc-consistency and search.

search space explored and the higher the computational overhead. Finally, by removing from each future variable's domain all values that are not consistent with the current partial instantiation, the need to test values of the current variable for consistency with previous variables, is eliminated.

5.3.1 Look-ahead algorithms for pruning search

Look-ahead strategies rarely yields better worst-case performance guarantees over naive backtracking. The key issue is to determine 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, ex-

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

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

pressed in the specific SELECTVALUE subprocedure employed. GENERALIZED-LOOKAHEAD initially sets up all the tentative domains (the D' sets) to be equal to the original domains (the D sets), and the SELECTVALUE subprocedure propagates the current instantiation to remove values from the D' sets. In general, the algorithm may also records inferred constraints on future variables. However, 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 no longer current partial instantiations. Usually, (see implementation discussion ahead) n copies of each D' set, one for each level in the search tree, are maintained, to permit the reset action to be performed efficiently.

We will define four levels of look-ahead that are based on arc-consistency.

Forward-Checking

Forward-checking does the most limited form of constraint propagation during search. It propagates the effect of a tentative value selection to each of the future variables, *separately*. If, as a result, the domain of one of these future variables becomes empty, the value is not selected, and the next candidate value is tried. Forward checking uses SELECTVALUE-FORWARD-CHECKING 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 and 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_u and are a subset of $\{x_1, \ldots, x_i\}$. Enforcing consistency on a subproblem is achieved by removing from x_u's domain any values that conflict with $\vec{a}_i$. Forward checking treats these $n - i$ subproblems independently of each other, removing values from the domains of future variables; namely from the D' sets, as necessary. If the domain of one of the future variable x_u becomes empty, then the partial instantiation $\vec{a}_i$ cannot be extended to that variable, another value for x_i is considered.

Complexity of SelectValue-forward-checking. If the complexity of a constraint check is $O(1)$, then SELECTVALUE-FORWARD-CHECKING's complexity can be shown to be $O(ek^2)$, where k is the cardinality of the largest domain and e is the number of constraints, as follows. For a given tentative value of x_i the procedure tests 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 modified 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 has only blue in its domain, and selecting that value causes the domain of x_7 to be empty, so $x_3 = blue$ is rejected and x_3 is a deadend. See Figure 5.9. □

Arc-consistency look-ahead.

More work is done at each instantiation by look-ahead algorithms that enforce full arc-consistency on the uninstantiated variables after each assignment of a value to the current variable. If a variable's domain becomes empty during the process of enforcing arc-consistency, then the current value is rejected. SELECTVALUE-ARC-CONSISTENCY in Figure 5.10 implements this approach. The **repeat** ... **until** loop in the subprocedure is essentially an arc-consistency algorithm AC-1. The more efficient arc-consistency procedures can also be used within a SELECTVALUE subprocedure.

Complexity of SelectValue-arc-consistency. The optimal time complexity for any arc-consistency procedure is $O(ek^2)$, where e is the number of constraints in the subproblem, k is the cardinality of the largest domain. Since arc-consistency may need to be

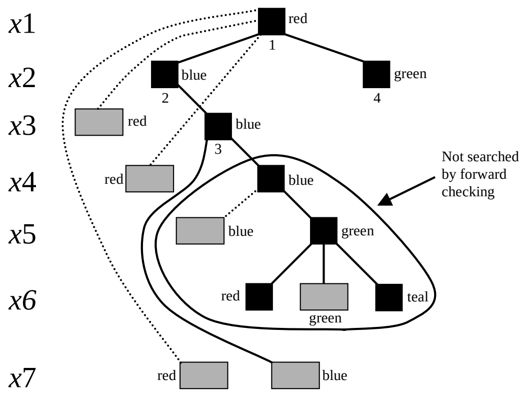


Figure 5.9: Part of the search space explored by forward-checking, on the example in Figure 5.3. Only the search space below $x_1 = red$ and $x_2 = blue$ is drawn. Dotted lines connect values with future values that are filtered out.

applied for each value of the current variable, SELECTVALUE-ARC-CONSISTENCY yields an optimal worst-case bound of $O(ek^3)$ checks. As presented, however, SelectValue-arc-consistency is far less efficient since it uses AC-1 algorithm. Its analysis is left as an exercise.

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**" 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 following them.

Example 5.3.3 Conside the problem in Figure 5.3 using the same ordering of variables and values as in 5.9, Applying 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 therefore rejected. Therefore, the whole tree in

```
subprocedure selectValue-arc-consistency

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

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

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 is applying directional arc-consistency to k subproblems, each require on the worst-case $O(ek^2)$ complexity and therefore the total complexity is bounded by $O(ek^3)$. This bound is identical tothat or arc-consistency look-ahead so clearly the bound does not reflect well the actual complexity. Notice, however, that this algorithm will not require a complex implementation of arc-consistency. In order to capture in the analysis the difference between these algorithms one may need to analyze the complexity of creating a full partial solution

5.3.2 Look-ahead for variable and value selection

Dynamic variable orderings (DVO)

Variable ordering has a tremendous effect on the size of the search space. Empirical and theoretical studies have shown that there are several effective fixed orderings that result in smaller search spaces. In particular, the *min-width ordering* and the *max-cardinality ordering*, both of which were first introduced in Chapter 4 and both of which use information from the constraint graph, are quite effective.

When determining variable ordering dynamically during search, one common heuristic, known as "fail-first", is to select as the next variable the one that is likely to most constrain the remainder of the search space. All other factors being equal, the variable with the smallest number of values in its (current) domain will have the fewest subtrees rooted at those values, and therefore the smallest search space below it. Given a current partial solution $\vec{a_i}$, we wish to determine the domain values for each future variable that are consistent with $\vec{a_i}$. The fewer such candidates, the stronger the selected variable's expected pruning power. We may estimate the domain sizes of future variables using the various levels of look-ahead propagation discussed above. Such methods are called *dynamic variable ordering* (DVO) strategies. 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 specify explicitly DVO with forward-checking. Forward checking, which does 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). Namely,

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

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 ith 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.

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 moved to be next in the ordering, and a dead-end will occur when this next variable becomes the current variable. This algorithm is described fully 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, assign it its only possible value, *red*, and subsequently forward-checking causes variable x_1 to also have a singleton domain. We choose x_1 and its only consistent value, *green*. At this point, after propagating this choice, we see that x_4 has one value, *red*; it is selected and assigned the value. Subsequently, x_2 can be selected and assigned its only consistent value, *blue*. Propagating this assignment does not further shrink any future domain. Next, x_5 can be selected and assigned *green*, and the solution is completed, without dead-ends, by assigning *red* or *teal* to x_6. □

Notice that $DVFC$ accomplishes also the 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.

Dynamic value orderings (LVO)

The *look-ahead value ordering* (LVO) algorithm [59] is alsoalso based on forward checking. Instead of accepting the first value for the current variable that is not shown to lead to a dead-end, LVO tentatively instantiates each value of the current variable and examines

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.

the effects of a forward checking style look-ahead on the domains of future variables. (Each instantiation and its effects are retracted before the next instantiation is made.) LVO uses a heuristic measure to transform this information into a ranking of the values. Experimental results indicate that the cost of performing the additional look-ahead is not justified on smaller and easier problems, but can be extremely useful on particularly hard problems.

5.3.3 Complexity and implementation issues

The cycle-cutset effect

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

Proposition 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 expressed graphically by deleting the corresponding variable from the constraint graph. Therefore, once a set of variables that forms a cycle-cutset is instantiated, the remaining problem can be perceived as a tree. A s noted in Chapter 4t a tree can be solved by directional arc-consistency, and therefore partial looking ahead performing directional arc-consistency at each node is guaranteed to solve the problem if the cycle-cutset variables initiate the search ordering. Since there are k^c possible instantiations of the cutset variables, and since each remaining tree is solved in $(n-c)k^2$ consistency checks, the complexity follows. □

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 values relative to the partial solution currently being assembled. When testing a new value of the current variable or after committing to a value for the current variable, the tables will tentatively be updated once constraint propagation has been applied. This strategy requires an additional $O(n \cdot k)$ space. Still, whenever a dead-end occurs, the algorithm has to recompute the tables associated with the state in the search to which the algorithm retracted. This may require n times the node generation cost that is dependent on the level of look-ahead used.

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

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

5.3.4 Extensions

The list of look-ahead techniques is not meant to be exhaustive. In particular, a search algorithm could enforce a higher degree of consistency than arc-consistency after each instantiation. Doing so will entail not only deleting values from domains, but adding new constraints. More recent work has shown that as larger and more difficult problems are experimented with, higher levels of look-ahead become more useful. It is likely that as experiments are conducted with larger and harder problems, look-ahead based on path-consistency will be cost-effective.

Look-ahead algorithms can also be extended to handle some generalized definition of consistency such as relational consistency that will be described in Chapter 8. Those definitions focus on constraints rather than on variables and are suited particularly for the non-binary case. We will mention the use of "generalized arc-consistency" next.

Definition 5.3.7 (general 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 $t[x] = a$. 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 user can try to extend SelectValue-arc-consistency and forward-checking to the generalized arc-consistency definition. (See exercises).

5.4 DPLL: Backtracking for Propositional cnfs

Algorithm backtracking and all its advances are clearly applicable to the special case of propositional satisfiability. The most well known, and still one of the best variation of backtracking for this domain is known as the Davis-Putnam algrithm (DP) [34] or more accurately, the DPLL algorithm [33]. The algorithm corresponds to backtracking with a simple form of look-ahead that applies unit-resolution at each node. This level of look-ahead correspond to applying a type of full arc-consistency at each node. This level of look-ahead is exploited both for dynamic variale and value seclection strategies.

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. $\quad Q$ = some unassigned variable;
6. $\quad$ return(**DPLL**($\varphi \wedge Q$) $\vee$
$\qquad\qquad$ **DPLL**($\varphi \wedge \neg Q$))

Figure 5.13: The DPLL Procedure

Figure 5.13 presents the *Davis-Putnam, Logemann and Loveland procedure (DPLL)* [33]. 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. At each step, DPLL performs unit propagation (line 1).which is similar to generalized *arc-consistency.* DPLL is often augmented by dynamic variable ordering heuristic for selecting the next unassigned variable (line 5 in Figure 5.13), normally prefering unit literals propositions.

A useful additional heuristic that has been recently adopted for these algorithms is the *2-literal clause heuristic*. This heuristic suggests instantiating next 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 demonstrted to significantly outperforms DPLL without this heuristic.

Example 5.4.1 An example of a search tree is shown in Figure 5.14. This tree is traversed when deciding the satisfiability of the propositional theory $\varphi = \{(\neg A \vee B), (\neg C \vee A), \neg B, C\}$. The tree nodes correspond to the variables while the tree branches to the two value assignments (0 and 1). Deadend nodes are crossed out. The theory is obviously inconsistent. Note that in this case, if we apply unit-resolution as in DPLL the problem will be solved without search with the first application of unit resolution. □

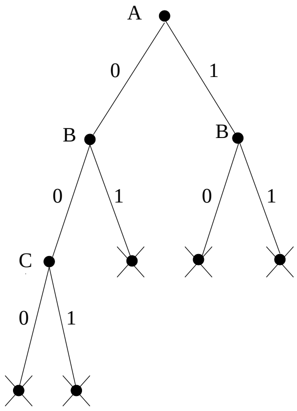


Figure 5.14: A backtracking search tree along the variables A, B, C for a cnf theory $\varphi = \{(\neg A \vee B), (\neg C \vee A), \neg B, C\}$.

5.5 Summary

The chapter introduces backtracking search for solving constraint satisfaction problems and focused on it enhancements using look-ahead algorithms. The primary variants of look-ahead, Forward-checking, full-look-ahead, partial look-ahead and arc-consistency look-ahead, were introduced. Their use for value pruning and for variable and value selection was discussed. The complexity overheads of these methods were analyzed. In addition the worst-case complexity bound as function of the problem's cycle-cutset was presented. The Chapter also presented backtracking search with look-ahead for propositional satisfiability, called DPLL.

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 [16]. One of the early and still one of the most influential ideas for improving backtracking's performance on constraint satisfaction problems was introduced by Waltz [168]. Waltz demonstrated that often, when constraint propagation in the form of arc-consistency is applied to a two-dimensional line-drawing interpretation, the problem can be solved without encountering any dead-ends. However, Golomb and Baumert [69] may have been the first to informally describe this idea. Consistency techniques for look-ahead are used in Lauriere's Alice system [99]. Explicit algorithms employing this idea have been given by Gaschnig [64], who described a backtracking algorithm that incorporates arc-consistency; McGregor [111], who described backtracking combined with forward checking; Haralick and Elliott

[72], who also added the various look-ahead methods that are described in this chapter (forward-checking, full look-ahead and partial look-ahead); and Nadel [119], who discussed backtracking combined with many variations of partial arc-consistency. Gaschnig [63] has compared Waltz-style look-ahead backtracking (that corresponds to arc-consistency look-ahead) with look-back improvements that he introduced, such as backjumping and backmarking. Haralick and Elliot [72] have done a relatively comprehensive study of look-ahead and look-back methods, in which they compared the performance of the various methods on n-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 until recently, when larger and more difficult problem classes were tested [139, 59, 60]. In these studies, forward checking lost its superiority on many problem instances to full looking ahead and other stronger looking-ahead variants. Empirical evaluation of backtracking with dynamic variable ordering on the n-queens problem was reported in [152]. More sophisticated DVO schemes have been proposed [75, 74, 150].

In the context of solving propositional satisfiability, Logemann, Davis and Loveland [33] introduced a backtracking algorithm that uses look-aheads for variable selection in the form of *unit resolution*, which is similar to arc-consistency. The *2-literal clause heuristic* was proposed by Crawford et. al. [28]. To date, this algorithm is perceived as one of the most successful procedures for that task. Analytical average-case analysis for some backtracking algorithms has been pursued for satisfiability [131] and for constraint satisfaction [72, 124, 120].

The balance between overhead and pruning is studied in [59, 139, 10, 61].

5.7 Exercises

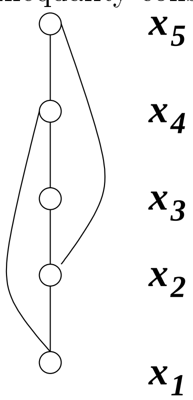
1. The two primary search algorithms for exploring any search space, are depth-first search (DFS) and breadth-first search (BFS). These algorithms have complementary properties. For instance, when BFS terminates, it finds an optimal solution. However, it may require exponential space to do so. DFS requires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

   Discuss why do you think DFS-based algorithms are the algorithms of choice for exploring constraint satisfaction search spaces.

2. Extend algorithm forward-checking and arc-consistency look-ahead (ACLH) 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 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 as best you can the overhead of each of the algorithm, backtracking, backmarking, forward-checking, partial look-ahead arc-consistency look-ahead. Assume that you keep a table of current domains at each level of the search tree. a) analyze each node generation separately, b) 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.

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 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 an graph having induced-width of 2.)

Bibliography

[1]

[2] Bar-Yehuda R A. Becker and D. Geiger. Random algorithms for the loop-cutset problem. In *Uncertainty in AI (UAI'99)*, pages 81–89, 1999.

[3] P. Ladkin ad A. Reinefeld. Effective solution of qualitative interval constraint problrms. *Artificial Intelligence*, 57:105–124, 1992.

[4] James Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.

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

[6] S. Arnborg and A. Proskourowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete and Applied Mathematics*, 23:11–24, 1989.

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

[8] F. Bacchus and P. van Run. Dynamic variable ordering in csps. In *Principles and Practice of Constraints Programming (CP'95)*, Cassis, France, 1995. Available as Lecture Notes on CS, vol 976, pp 258 – 277, 1995.

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

[10] A. B. Baker. Intelligent backtracking on constraint satisfaction problems: experimental and theoretical results. Technical report, Ph.D. thesis, Graduate school of the University of Oregon, Oregon, 1995.

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

[12] A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal junction trees. In *Uncertainty in AI (UAI'96)*, pages 81–89, 1996.

[13] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database ochemes. *Journal of the ACM*, 30(3):479–513, 1983.

[14] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming.* Academic Press, 1972.

[15] C. Bessiere and M Cordier. Arc-consistency and arc-consistency again. In *National Conference of Artificial Intelligence (AAAI-93)*, pages 108–113, 1993.

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

[17] M. Bjäreland and P. Jonsson. Exploiting bipartiteness to identify yet another tractable subclass of CSP. In J. Jaffar, editor, *Principles and Practice of Constraint Programming—CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 118–128. Springer-Verlag, 1999.

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

[19] A.A. Bulatov and P.G. Jeavons. Tractable constraints closed under a binary operation. Technical Report PRG-TR-12-00, Oxford University Computing Laboratory, 2000.

[20] A.A. Bulatov, A.A. Krokhin, and P.G. Jeavons. Constraint satisfaction problems and finite algebras. In *Proceedings 27th International Colloquium on Automata, Languages and Programming—ICALP'00*, volume 1853 of *Lecture Notes in Computer Science*, pages 272–282. Springer-Verlag, 2000.

[21] A.A. Bulatov, A.A. Krokhin, and P.G. Jeavons. The complexity of maximal constraint languages. In *Proceedings of STOC'01*, 2001.

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

[23] D.A. Cohen, M. Gyssens, and P.G. Jeavons. Derivation of constraints and database relations. In *Proceedings 2nd International Conference on Constraint Programming—CP'96 (Boston, August 1996)*, volume 1118 of *Lecture Notes in Computer Science*, pages 134–148. Springer-Verlag, 1996.

[24] P.M. Cohn. *Universal Algebra.* Harper & Row, 1965.

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

[26] M.C. Cooper, D.A. Cohen, and P.G. Jeavons. Characterising tractable constraints. *Artificial Intelligence*, 65:347–361, 1994.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

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

[29] V. Dalmau. A new tractable class of constraint satisfaction problems. In *6th International Symposium on Mathematics and Artificial Intelligence*, 2000.

[30] V. Dalmau and J. Pearson. Closure functions and width 1 problems. In J. Jaffar, editor, *Principles and Practice of Constraint Programming—CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 159–173. Springer-Verlag, 1999.

[31] G. B. Dantzig. *Linear programming and extensions.* Princeton university Press, Princeton,NJ, 1962.

[32] E. Davis. Constraint propagation with interva labels. *Artificial Intelligence*, 32:281–331, 1987.

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

[34] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association of Computing Machinery*, 7(3), 1960.

[35] T. L. Dean. Using temporal hierarchies to efficiently maintain large temporal databases. *Journal of the ACM*, 36:687–714, 1989.

[36] T. L. Dean and D. V. McDermott. Temporal database management. *Artificial Intelligence*, 32:1–55, 1987.

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

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

[39] R. Dechter. From local to global consistency. *Artificial Intelligence*, 55(1):87–107, 1992.

[40] R. Dechter. Mini-buckets: A general scheme of generating approximations in automated reasoning. In *IJCAI-97: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1297–1302, 1997.

[41] R. Dechter. Constraint satisfaction. In *The MIT Encyclopedia of Cognitive Sciences (MITECS)*, pages 195–197, 1999.

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

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

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

[45] Y. Deville and P. Van Hentenryck. An efficient arc consistency algorithm for a of CSP problems. pages 325–330, Sydney, Australia, 1991. Use updated reference VHDT92.

[46] Thomas Drakengren and Peter Jonsson. A complete classification of tractability in Allen's algebra relative to subsets of basic relations. *Artificial Intelligence*, 106(2):205–219, 1998.

[47] T. Drankengren and P. Jonsson. Towards a complete classification of tractability in allen's algebra. In *International Joint Conference on Artificial Intelligence (Ijcai-97)*.

[48] T. Drankengren and P. Jonsson. Maximal tractable subclasses of allen'sinterval algebra: preliminary report. In *National Conference on Artificial Intelligence (AAAI'96)*, pages 389–394, 1996.

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

[50] T. Feder and M.Y. Vardi. The computational structure of monotone monadic SNP and constraint satisfaction: a study through datalog and group theory. *SIAM Journal of Computing*, 28(1):57–104, 1998.

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

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

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

[54] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21:958–966, 1978.

[55] E.C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32:755–761, 1985.

[56] E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI-91*, pages 227–233, 1991.

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

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

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

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

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

[62] M. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, CA., 1979.

[63] J. Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence*, pages 268–277, Toronto, Ont., 1978.

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

[65] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for sat. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, pages 28–33, 1993.

[66] Nicola Leone Georg Gottlob and Francesco Scarello. A comparison of structural csp decomposition methods. *Ijcai-99*, 1999.

[67] A. Gerevini and L. Schubert. Efficient algorithms for qualitative reasoning about-time. *Artificial Intelligence*, 74.

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

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

[70] M. C. Golumbic and R. Shamir. Complexity and algorithms for reasoning about time: a graph-theoretic approach. *Journal of the ACM*, 40:1108–1133, 1993.

[71] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 394–399. Morgan Kaufmann, 1999.

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

[73] J. F. Allenand P. J. Hayes. A common-sense theory of time. In *International Joint Conference oon Artificial Intelligece (Ijcai-85)*, pages 528–531, 1985.

[74] P. Prosser I. P. Gent, E. MacIntyre and T. Walsh. The constrainedness of search. In *Thirteen National Conference on Artificial Intelligence*, 1996.

[75] P. Prosser B. M. Smith I. P. Gent, E. MacIntyre and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Principles and Practice of Constraint Programming (CP95)*, pages 179–173, 1995.

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

[77] P.G. Jeavons. On the algebraic structure of combinatorial problems. *Theoretical Computer Science*, 200:185–204, 1998.

[78] P.G. Jeavons and D.A. Cohen. An algebraic characterization of tractable constraints. In *Computing and Combinatorics. First International Conference COCOON'95 (Xi'an,China,August 1995)*, volume 959 of *Lecture Notes in Computer Science*, pages 633–642. Springer-Verlag, 1995.

[79] P.G. Jeavons, D.A. Cohen, and M. Gyssens. A unifying framework for tractable constraints. In *Proceedings 1st International Conference on Constraint Programming—CP'95 (Cassis, France, September 1995)*, volume 976 of *Lecture Notes in Computer Science*, pages 276–291. Springer-Verlag, 1995.

[80] P.G. Jeavons, D.A. Cohen, and M. Gyssens. Closure properties of constraints. *Journal of the ACM*, 44:527–548, 1997.

[81] P.G. Jeavons, D.A. Cohen, and J.K. Pearson. Constraints and universal algebra. *Annals of Mathematics and Artificial Intelligence*, 24:51–67, 1999.

[82] P.G. Jeavons and M.C. Cooper. Tractable constraints on ordered domains. *Artificial Intelligence*, 79(2):327–339, 1995.

[83] K. Kask. Approximation algorithms for graphical models. Technical report, Ph.D. thesis, Information and Computer Science, University of California, Irvine, California, 2001.

[84] H. Kautz and P. B. Ladkin. Integrating metric and qualitative temporal reasoning. In *National Conference of Artificial Intelligence (AAAI-91)*, pages 241–246, 1991.

[85] L. G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, pages 191–194, 1979.

[86] L. Kirousis. Fast parallel constraint satisfaction. *Artificial Intelligence*, 64:147–160, 1993.

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

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

[89] M. Koubarakis. Dense time and temporal constraints with inequalities. In *Proceedings of the Third international Conference on Principles of Knowledge-bases and Reasoning (KR-92)*, pages 24–35, 1992.

[90] M. Koubarakis. From local to global consistency in temporal constraint networks. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP'95)*, pages 53–69, 1995.

[91] M. Koubarakis. tractable disjunctions of linear constraints. In *Principles and Practice of Constraint Programming (CP'96)*, pages 297–307, 1996.

[92] Manolis Koubarakis. From local to global consistency in temporal constraint networks. *Theoretical Computer Science*, 173(1):89–112, 20 February 1997.

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

[94] P. B. Ladkin. Time representation: A taxonomy of interval relations. In *National Conference of Artificial Intelligence (AAAI-86)*, pages 360–366, 1988.

[95] P. B. Ladkin. Metric constraint satisfaction with intervals. In *Technical Report, TR-89-038, Interional Computer Science Institut, Berkeley, CA*, 1989.

[96] P. B. Ladkin and R. D. Maddux. On binary constraint networks. In *Technical Report, Kerstel Institute, Palo Alto, CA*, 1989.

[97] P.B. Ladkin and R.D. Maddux. On binary constraint problems. *Journal of the ACM*, 41:435–469, 1994.

[98] J. larrosa. Boosting search with variable-elimination. *In pproceedings of CP2000*, 2000.

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

[100] S.L. Lauritzen and D.J. Spiegelhalter. Local computation with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B*, 50(2):157–224, 1988.

[101] C.E. Leiserson and J. B. Saxe. A mixed-integer linear programming problem which is efficiently solvable. In *21st annual Allerton Conference on Communication, Control and Computing*, pages 204–213, 1983.

[102] H. Kautz M. Villain and P. van Beek. Constraint propagation algorithms for temporal reasoning: A revised report. In *Readings in Qualitative Reasoning about Physical Systems¡ Daniel Weld and J. de Kleer, Morgan Kaufman*, pages 373–381, 1989.

[103] H. Machida and I.G. Rosenberg. Classifying essentially minimal clones. In *Proceedings 14th International Symposium on Multiple-Valued Logic (Winnipeg, 1984)*, pages 4–7, 1984.

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

[105] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. John Wiley & Sons, 1987. Use updated reference Mackworth92.

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

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

[108] D. Maier. The theory of relational databases. In *Computer Science Press, Rockville, MD*, 1983.

[109] K. Marriot and P. Stuckey. Programming with constraints, an introduction. In *MIT Press*, 1998.

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

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

[112] I. Meiri. Combining qualitative and quantitative constraints in temporal reasoning. *Artificial Intelligence*, 87:343–385, 1996.

[113] I. Meiri. Combining qualitative and quantitative constraint satisfaction problems. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI'91)*, Anaheim, CA, July 1991.

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

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

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

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

[118] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, pages 40–45, 1993.

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

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

[121] B. Nebel and H. Bürckert. Reasoning about temporal relations: A maximal tractable subclass of allen's interval algebra. *Journal of the ACM*, 42(1):43–66, 1995.

[122] B. Nebel and H-J. Bürckert. Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra. *Journal of the ACM*, 42(1):43–66, January 1995.

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

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

[125] Y. Deville P. van Henetenryck and Teng C. A generic arc-consistency algorithm and its specialization. *Artificial Intelligence*, 57:291–321, 1992.

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

[127] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[128] G. Pinkas and R. Dechter. Improving connectionist energy minimization. *Journal of Artificial Intelligence Research (JAIR)*, pages 223–248, 1995.

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

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

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

[132] Jr. R. Bayardo and D. P. Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *Fourteenth International Joint Conference on Artificial Intelligence(I95)*, pages 558–562, 1995.

[133] A. Dechter R. Dechter and J. Pearl. Optimization in constraint networks. In *Influence Diagrams, Belief Nets and Decision Analysis*, pages 411–425. John Wiley & Sons, 1990.

[134] I. Meiri R. Dechter and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1990.

[135] I. Rish and R. Dechter. Resolution vs. search; two strategies for sat. *Journal of Automated Reasoning*, 24(1/2):225–275, 2000.

[136] I.G. Rosenberg. Minimal clones I: the five types. In *Lectures in Universal Algebra (Proc. Conf. Szeged 1983)*, volume 43 of *Colloq. Math. Soc. Janos Bolyai*, pages 405–427. North-Holland, 1986.

[137] D. G. Corneil S. A. Arnborg and A. Proskourowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Discrete Mathematics.*, 8:277–284, 1987.

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

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

[140] T.J. Schaefer. The complexity of satisfiability problems. In *Proceedings 10th ACM Symposium on Theory of Computing (STOC)*, pages 216–226, 1978.

[141] E. Schwalb. Temporal reasoning with constraints. Technical report, Ph.d. thesis, Information and Computer Science, Universiy of California, Irvine, 1998.

[142] E. Schwalb and R. Dechter. Processing disjunctions in temporal constraint networks. *Artificial Intelligence*, pages 29–61, 1997.

[143] E. Schwalb and L. Villa. Temporal constaints: A survey. *Constraints*, pages 1–20, 1998.

[144] B. Selman and H. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 46–51, 1993.

[145] B. Selman, H. Kautz, and B. Cohen. Noise strategies for local search. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 337–343, 1994.

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

[147] P.P. Shenoy. Binary join trees. pages 492–499, 1996.

[148] K. Shoiket and D. Geiger. A proctical algorithm for finding optimal triangulations. In *Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 185–190, 1997.

[149] R. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28:769–779, 1981.

[150] B. Smith and S. A. Grant. Trying harder to failfirst. In *European conference on Artificial Intelligence (ECAI-92)*, pages 249–253, 1998.

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

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

[153] A. Szendrei. *Clones in Universal Algebra*, volume 99 of *Seminaires de Mathematiques Superieures.* University of Montreal, 1986.

[154] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs. *SIAM Journal of Computation.*, 13(3):566–579, 1984.

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

[156] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. 1.* Computer Science Press, 1988.

[157] R. E. Valdes-Perez. Spatio-temporal reasoning and linear inequalities. In *Tech Report AIM-875, Artificial Intelligence Lab, MIT, Cambridge, MA*, 1986.

[158] R. E. Valdes-Perez. The satisfiability of temporal constraint networks. In *National Conference of Artificial Intelligence (AAAI-87)*, pages 256–260, 1987.

[159] P. van Beek. *Exact and Approximate Reasoning about Qualitative Temporal Relations.* PhD thesis, University of Waterloo, 1990. Available as: Department of Computing Science Technical Report TR-90-29, University of Alberta.

[160] P. van Beek and R. Cohen. Exact and approximate reasoning about temporal relations. *Computational Intelligence*, 6:132–144, 1990.

[161] P. van Beek and R. Dechter. On the minimality and decomposability of row-convex constraint networks. *Journal of the ACM*, 42:543–561, 1995.

[162] P. van Beek and R. Dechter. On the minimality and global consistency of row-convex constraint networks. *Journal of the ACM*, 42(3):543–561, May 1995.

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

[164] P. VanBeek. Approximation algorithms for tempral reasoning. In *International Joint Conference on on Artificial Intelligence (IJCAI89)*, pages 1291–1296, 1989.

[165] P. VanBeek. Reasoning about qualitative temporal information. *Artificial Intelligence*, 58:297–326, 1992.

[166] M. Vilain. A system of resoningabout time. In *National confrence of Artificial Intelligence (AAAI'82)*, pages 197–201, 1982.

[167] M. Villain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *National Conference on Artificial Intelligence (AAAI'86)*, pages 377–382, 1986.

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

[169] R. Dechter Z. Collin and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of the twelfth International Conference of Artificial Intelligence (IJCAI-91)*, pages 318–324, Sidney, Australia, 1991.

[170] R. Zeidel. A new method for solving constraint satisfaction problems. In *International Joint Conference oon Artificial Intelligece (Ijcai-81)*, pages 338–342, 1981.