5 CONSTRAINT SATISFACTION PROBLEMS

In which we see how treating states as more than just little black boxes leads to the invention of a range of powerful new search methods and a deeper understanding of problem structure and complexity.

Chapters 3 and 4 explored the idea that problems can be solved by searching in a space of **states**. These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states. From the point of view of the search algorithm, however, each state is a **black box** with no discernible internal structure. It is represented by an arbitrary data structure that can be accessed only by the *problem-specific* routines—the successor function, heuristic function, and goal test.

BLACK BOX

This chapter examines **constraint satisfaction problems**, whose states and goal test conform to a standard, structured, and very simple **representation** (Section 5.1). Search algorithms can be defined that take advantage of the structure of states and use *general-purpose* rather than *problem-specific* heuristics to enable the solution of large problems (Sections 5.2–5.3). Perhaps most importantly, the standard representation of the goal test reveals the structure of the problem itself (Section 5.4). This leads to methods for problem decomposition and to an understanding of the intimate connection between the structure of a problem and the difficulty of solving it.

REPRESENTATION

5.1 CONSTRAINT SATISFACTION PROBLEMS

CONSTRAINT SATISFACTION PROBLEM

Formally speaking, a **constraint satisfaction problem** (or CSP) is defined by a set of **variables**, $X_1, X_2, \ldots, X_n$, and a set of **constraints**, $C_1, C_2, \ldots, C_m$. Each variable X_i has a nonempty **domain** D_i of possible **values**. Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \ldots\}$. An assignment that does not violate any constraints is called a **consistent** or legal assignment. A complete assignment is one in which every variable is mentioned, and a **solution** to a CSP is a complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes an **objective function**.

VARIABLES
CONSTRAINTS
DOMAIN
VALUES
ASSIGNMENT
CONSISTENT
OBJECTIVE FUNCTION

So what does all this mean? Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories, as in Figure 5.1(a), and that we are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions: WA, NT, Q, NSW, V, SA, and T. The domain of each variable is the set $\{red, green, blue\}$. The constraints require neighboring regions to have distinct colors; for example, the allowable combinations for WA and NT are the pairs

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\} .$$

(The constraint can also be represented more succinctly as the inequality $WA \neq NT$, provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions, such as

$$\{ WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red \}.$$

CONSTRAINT GRAPH

It is helpful to visualize a CSP as a **constraint graph**, as shown in Figure 5.1(b). The nodes of the graph correspond to variables of the problem and the arcs correspond to constraints.

Treating a problem as a CSP confers several important benefits. Because the representation of states in a CSP conforms to a *standard* pattern—that is, a set of variables with assigned values—the successor function and goal test can written in a generic way that applies to all CSPs. Furthermore, we can develop effective, generic heuristics that require no additional, domain-specific expertise. Finally, the structure of the constraint graph can be used to simplify the solution process, in some cases giving an exponential reduction in complexity. The CSP representation is the first, and simplest, in a series of representation schemes that will be developed throughout the book.

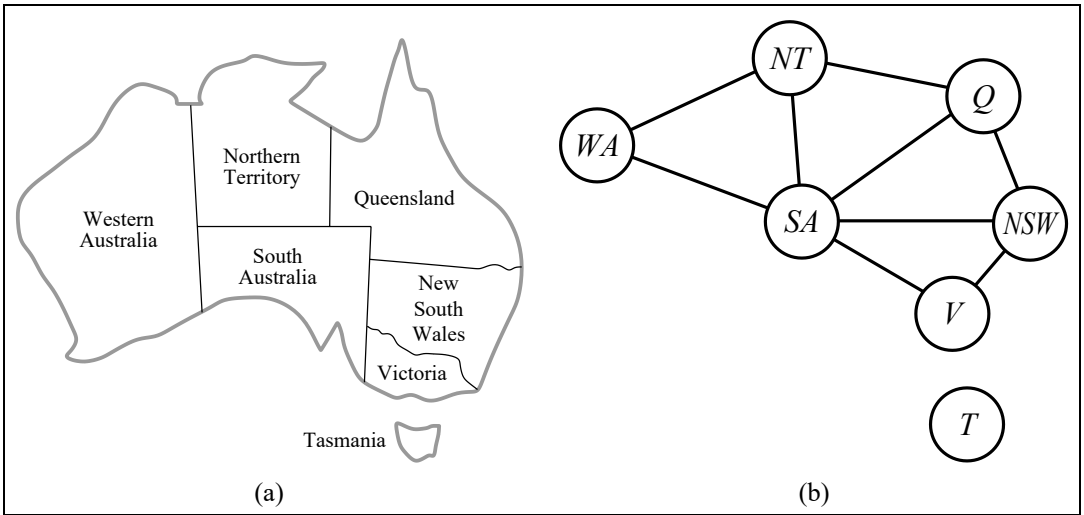


Figure 5.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

It is fairly easy to see that a CSP can be given an **incremental formulation** as a standard search problem as follows:

- ♢ **Initial state**: the empty assignment {}, in which all variables are unassigned.
- ♢ **Successor function**: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
- ♢ **Goal test**: the current assignment is complete.
- ♢ **Path cost**: a constant cost (e.g., 1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables. Furthermore, the search tree extends only to depth n. For these reasons, depth-first search algorithms are popular for CSPs. (See Section 5.2.) It is also the case that *the path by which a solution is reached is irrelevant.* Hence, we can also use a **complete-state formulation**, in which every state is a complete assignment that might or might not satisfy the constraints. Local search methods work well for this formulation. (See Section 5.3.)

FINITE DOMAINS

The simplest kind of CSP involves variables that are **discrete** and have **finite domains**. Map-coloring problems are of this kind. The 8-queens problem described in Chapter 3 can also be viewed as a finite-domain CSP, where the variables $Q_1, \ldots, Q_8$ are the positions of each queen in columns $1, \ldots, 8$ and each variable has the domain $\{1, 2, 3, 4, 5, 6, 7, 8\}$. If the maximum domain size of any variable in a CSP is d, then the number of possible complete assignments is $O(d^n)$—that is, exponential in the number of variables. Finite-domain CSPs include **Boolean CSPs**, whose variables can be either $true$ or $false$. Boolean CSPs include as special cases some NP-complete problems, such as 3SAT. (See Chapter 7.) In the worst case, therefore, we cannot expect to solve finite-domain CSPs in less than exponential time. In most practical applications, however, general-purpose CSP algorithms can solve problems *orders of magnitude* larger than those solvable via the general-purpose search algorithms that we saw in Chapter 3.

BOOLEAN CSPS

INFINITE DOMAINS

Discrete variables can also have **infinite domains**—for example, the set of integers or the set of strings. For example, when scheduling construction jobs onto a calendar, each job's start date is a variable and the possible values are integer numbers of days from the current date. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values. Instead, a **constraint language** must be used. For example, if Job_1, which takes five days, must precede Job_3, then we would need a constraint language of algebraic inequalities such as $StartJob_1 + 5 \leq StartJob_3$. It is also no longer possible to solve such constraints by enumerating all possible assignments, because there are infinitely many of them. Special solution algorithms (which we will not discuss here) exist for **linear constraints** on integer variables—that is, constraints, such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables. In some cases, we can reduce integer constraint problems to finite-domain problems simply by bounding the values of all the variables. For example, in a scheduling problem, we can set an upper bound equal to the total length of all the jobs to be scheduled.

CONSTRAINT LANGUAGE

LINEAR CONSTRAINTS

NONLINEAR CONSTRAINTS

CONTINUOUS DOMAINS

Constraint satisfaction problems with **continuous domains** are very common in the real world and are widely studied in the field of operations research. For example, the scheduling

of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear inequalities forming a *convex* region. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on.

LINEAR PROGRAMMING

In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the **unary constraint**, which restricts the value of a single variable. For example, it could be the case that South Australians actively dislike the color $green$. Every unary constraint can be eliminated simply by preprocessing the domain of the corresponding variable to remove any value that violates the constraint. A **binary constraint** relates two variables. For example, $SA \neq NSW$ is a binary constraint. A binary CSP is one with only binary constraints; it can be represented as a constraint graph, as in Figure 5.1(b).

UNARY CONSTRAINT

BINARY CONSTRAINT

Higher-order constraints involve three or more variables. A familiar example is provided by **cryptarithmetic** puzzles. (See Figure 5.2(a).) It is usual to insist that each letter in a cryptarithmetic puzzle represent a different digit. For the case in Figure 5.2(a)), this would be represented as the six-variable constraint $Alldiff\,(F, T, U, W, R, O)$. Alternatively, it can be represented by a collection of binary constraints such as $F \neq T$. The addition constraints on the four columns of the puzzle also involve several variables and can be written as

CRYPTARITHMETIC

$$
\begin{aligned}
O + O &= R + 10 \cdot X_1 \\
X_1 + W + W &= U + 10 \cdot X_2 \\
X_2 + T + T &= O + 10 \cdot X_3 \\
X_3 &= F
\end{aligned}
$$

where X_1, X_2, and X_3 are **auxiliary variables** representing the digit (0 or 1) carried over into the next column. Higher-order constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 5.2(b). The sharp-eyed reader will have noticed that the $Alldiff$ constraint can be broken down into binary constraints—$F \neq T$, $F \neq U$, and so on. In fact, as Exercise 5.11 asks you to prove, every higher-order, finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced. Because of this, we will deal only with binary constraints in this chapter.

AUXILIARY VARIABLES

CONSTRAINT HYPERGRAPH

The constraints we have described so far have all been **absolute** constraints, violation of which rules out a potential solution. Many real-world CSPs include **preference** constraints indicating which solutions are preferred. For example, in a university timetabling problem, Prof. X might prefer teaching in the morning whereas Prof. Y prefers teaching in the afternoon. A timetable that has Prof. X teaching at 2 p.m. would still be a solution (unless Prof. X happens to be the department chair), but would not be an optimal one. Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. X costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved using opti-

PREFERENCE

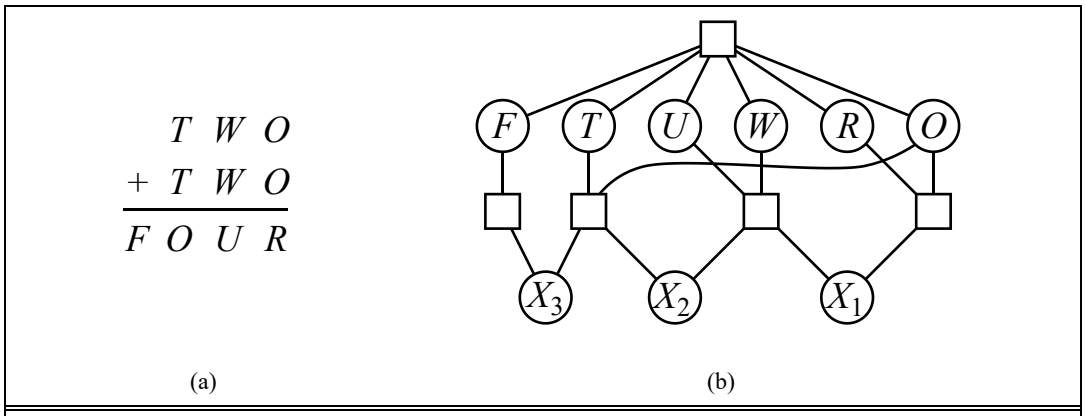


Figure 5.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint as well as the column addition constraints. Each constraint is a square box connected to the variables it constrains.

mization search methods, either path-based or local. We do not discuss such CSPs further in this chapter, but we provide some pointers in the bibliographical notes section.

5.2 Backtracking Search for CSPs

The preceding section gave a formulation of CSPs as search problems. Using this formulation, any of the search algorithms from Chapters 3 and 4 can solve CSPs. Suppose we apply breadth-first search to the generic CSP problem formulation given in the preceding section. We quickly notice something terrible: the branching factor at the top level is nd, because any of d values can be assigned to any of n variables. At the next level, the branching factor is $(n-1)d$, and so on for n levels. We generate a tree with $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!

COMMUTATIVITY

Our seemingly reasonable but naïve problem formulation has ignored a crucial property common to all CSPs: **commutativity**. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. This is the case for CSPs because, when assigning values to variables, we reach the same partial assignment, regardless of order. Therefore, *all CSP search algorithms generate successors by considering possible assignments for only a* single *variable at each node in the search tree.* For example, at the root node of a search tree for coloring the map of Australia, we might have a choice between $SA = red$, $SA = green$, and $SA = blue$, but we would never choose between $SA = red$ and $WA = blue$. With this restriction, the number of leaves is d^n, as we would hope.

BACKTRACKING SEARCH

The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in Figure 5.3. Notice that it uses, in effect, the one-at-a-time method of

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment according to CONSTRAINTS[csp] then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```

Figure 5.3 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES can be used to implement the general-purpose heuristics discussed in the text.

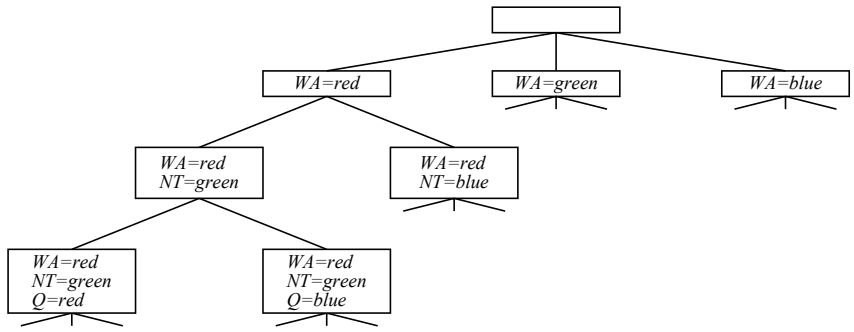


Figure 5.4 Part of the search tree generated by simple backtracking for the map-coloring problem in Figure 5.1.

incremental successor generation described on page 76. Also, it extends the current assignment to generate a successor, rather than copying it. Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, successor function, or goal test. Part of the search tree for the Australia problem is shown in Figure 5.4, where we have assigned variables in the order $WA, NT, Q, \ldots$.

Plain backtracking is an uninformed algorithm in the terminology of Chapter 3, so we do not expect it to be very effective for large problems. The results for some sample problems are shown in the first column of Figure 5.5 and confirm our expectations.

In Chapter 4 we remedied the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem. It turns out that we can solve CSPs efficiently without such domain-specific knowl-

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV	Min-Conflicts
USA	(> 1,000K)	(> 1,000K)	2K	60	64
n-Queens	(> 40,000K)	13,500K	(> 40,000K)	817K	4K
Zebra	3,859K	1K	35K	0.5K	2K
Random 1	415K	3K	26K	2K	
Random 2	942K	27K	77K	15K	

Figure 5.5 Comparison of various CSP algorithms on various problems. The algorithms from left to right, are simple backtracking, backtracking with the MRV heuristic, forward checking, forward checking with MRV, and minimum conflicts local search. Listed in each cell is the median number of consistency checks (over five runs) required to solve the problem; note that all entries except the two in the upper right are in thousands (K). Numbers in parentheses mean that no answer was found in the allotted number of checks. The first problem is finding a 4-coloring for the 50 states of the United States of America. The remaining problems are taken from Bacchus and van Run (1995), Table 1. The second problem counts the total number of checks required to solve all n-Queens problems for n from 2 to 50. The third is the "Zebra Puzzle," as described in Exercise 5.13. The last two are artificial random problems. (Min-conflicts was not run on these.) The results suggest that forward checking with the MRV heuristic is better on all these problems than the other backtracking algorithms, but not always better than min-conflicts local search.

edge. Instead, we find general-purpose methods that address the following questions:

1. Which variable should be assigned next, and in what order should its values be tried?
2. What are the implications of the current variable assignments for the other unassigned variables?
3. When a path fails—that is, a state is reached in which a variable has no legal values—can the search avoid repeating this failure in subsequent paths?

The subsections that follow answer each of these questions in turn.

Variable and value ordering

The backtracking algorithm contains the line

$$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(\text{VARIABLES}[csp], assignment, csp).$$

By default, SELECT-UNASSIGNED-VARIABLE simply selects the next unassigned variable in the order given by the list VARIABLES$[csp]$. This static variable ordering seldom results in the most efficient search. For example, after the assignments for $WA = red$ and $NT = green$, there is only one possible value for SA, so it makes sense to assign $SA = blue$ next rather than assigning Q. In fact, after SA is assigned, the choices for Q, NSW, and V are all forced. This intuitive idea—choosing the variable with the fewest "legal" values—is called the **minimum remaining values** (MRV) heuristic. It also has been called the "most constrained variable" or "fail-first" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If there is a variable X with zero legal values remaining, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables which always will fail when X is finally selected.

MINIMUM REMAINING VALUES

The second column of Figure 5.5, labeled BT+MRV, shows the performance of this heuristic. The performance is 3 to 3,000 times better than simple backtracking, depending on the problem. Note that our performance measure ignores the extra cost of computing the heuristic values; the next subsection describes a method that makes this cost manageable.

DEGREE HEURISTIC

The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors. In this case, the **degree heuristic** comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. In Figure 5.1, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T, which has 0. In fact, once SA is chosen, applying the degree heuristic solves the problem without any false steps—you can choose any consistent color at each choice point and still arrive at a solution with no backtracking. The minimum remaining values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.

LEAST-CONSTRAINING-VALUE

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. For this, the **least-constraining-value** heuristic can be effective in some cases. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. For example, suppose that in Figure 5.1 we have generated the partial assignment with $WA = red$ and $NT = green$, and that our next choice is for Q. Blue would be a bad choice, because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue. In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments. Of course, if we are trying to find all the solutions to a problem, not just the first one, then the ordering does not matter because we have to consider every value anyway. The same holds if there are no solutions to the problem.

Propagating information through constraints

So far our search algorithm considers the constraints on a variable only at the time that the variable is chosen by SELECT-UNASSIGNED-VARIABLE. But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Forward checking

FORWARD CHECKING

One way to make better use of constraints during search is called **forward checking**. Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y's domain any value that is inconsistent with the value chosen for X. Figure 5.6 shows the progress of a map-coloring search with forward checking. There are two important points to notice about this example. First, notice that after assigning $WA = red$ and $Q = green$, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q. The MRV heuristic, which is an obvious partner for forward checking, would automatically select SA and NT next. (Indeed, we can view forward checking as an efficient way to incrementally compute the information that the

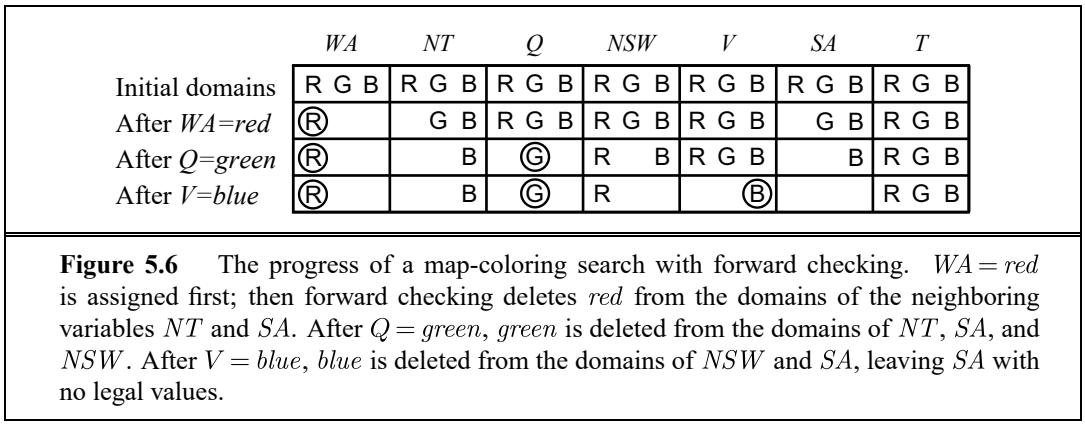


Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After $Q = green$, $green$ is deleted from the domains of NT, SA, and NSW. After $V = blue$, $blue$ is deleted from the domains of NSW and SA, leaving SA with no legal values.

MRV heuristic needs to do its job.) A second point to notice is that, after $V = blue$, the domain of SA is empty. Hence, forward checking has detected that the partial assignment $\{WA = red, Q = green, V = blue\}$ is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.

Constraint propagation

Although forward checking detects many inconsistencies, it does not detect all of them. For example, consider the third row of Figure 5.6. It shows that when WA is red and Q is $green$, both NT and SA are forced to be blue. But they are adjacent and so cannot have the same value. Forward checking does not detect this as an inconsistency, because it does not look far enough ahead. **Constraint propagation** is the general term for propagating the implications of a constraint on one variable onto other variables; in this case we need to propagate from WA and Q onto NT and SA, (as was done by forward checking) and then onto the constraint between NT and SA to detect the inconsistency. And we want to do this fast: it is no good reducing the amount of search if we spend more time propagating constraints than we would have spent doing a simple search.

CONSTRAINT PROPAGATION

ARC CONSISTENCY

The idea of **arc consistency** provides a fast method of constraint propagation that is substantially stronger than forward checking. Here, "arc" refers to a *directed* arc in the constraint graph, such as the arc from SA to NSW. Given the current domains of SA and NSW, the arc is consistent if, for *every* value x of SA, there is *some* value y of NSW that is consistent with x. In the third row of Figure 5.6, the current domains of SA and NSW are $\{blue\}$ and $\{red, blue\}$ respectively. For $SA = blue$, there is a consistent assignment for NSW, namely, $NSW = red$; therefore, the arc from SA to NSW is consistent. On the other hand, the reverse arc from NSW to SA is not consistent: for the assignment $NSW = blue$, there is no consistent assignment for SA. The arc can be made consistent by deleting the value $blue$ from the domain of NSW.

We can also apply arc consistency to the arc from SA to NT at the same stage in the search process. The third row of the table in Figure 5.6 shows that both variables have the domain $\{blue\}$. The result is that $blue$ must be deleted from the domain of SA, leaving the domain empty. Thus, applying arc consistency has resulted in early detection of an inconsis-

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
      for each Xk in NEIGHBORS[Xi] do
        add (Xk, Xi) to queue

function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff we remove a value
  removed ← false
  for each x in DOMAIN[Xi] do
    if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint between Xi and Xj
      then delete x from DOMAIN[Xi]; removed ← true
  return removed
```

Figure 5.7 The arc consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be made arc-consistent (and thus the CSP cannot be solved). The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

tency that is not detected by pure forward checking.

Arc consistency checking can be applied either as a preprocessing step before the beginning of the search process, or as a propagation step (like forward checking) after every assignment during search. (The latter algorithm is sometimes called MAC, for *Maintaining Arc Consistency*.) In either case, the process must be applied *repeatedly* until no more inconsistencies remain. This is because, whenever a value is deleted from some variable's domain to remove an arc inconsistency, a new arc inconsistency could arise in arcs pointing to that variable. The full algorithm for arc consistency, AC-3, uses a queue to keep track of the arcs that need to be checked for inconsistency. (See Figure 5.7.) Each arc (X_i, X_j) in turn is removed from the agenda and checked; if any values need to be deleted from the domain of X_i, then every arc (X_k, X_i) pointing to X_i must be reinserted on the queue for checking. The complexity of arc consistency checking can be analyzed as follows: a binary CSP has at most $O(n^2)$ arcs; each arc (X_k, X_i) can be inserted on the agenda only d times, because X_i has at most d values to delete; checking consistency of an arc can be done in $O(d^2)$ time; so the total worst-case time is $O(n^2 d^3)$. Although this is substantially more expensive than forward checking, the extra cost is usually worthwhile.[1]

Because CSPs include 3SAT as a special case, we do not expect to find a polynomial-time algorithm that can decide whether a given CSP is consistent. Hence, we deduce that arc consistency does not reveal every possible inconsistency. For example, in Figure 5.1, the partial assignment $\{WA = red, NSW = red\}$ is inconsistent, but AC-3 will not find the incon-

[1] The AC-4 algorithm, due to Mohr and Henderson (1986), runs in $O(n^2 d^2)$. See Exercise 5.10.

K-CONSISTENCY

sistency. Stronger forms of propagation can be defined using the notion called k**-consistency**. A CSP is k-consistent if, for any set of $k-1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable. For example, 1-consistency means that each individual variable by itself is consistent; this is also called

NODE CONSISTENCY

node consistency. 2-consistency is the same as arc consistency. 3-consistency means that any pair of adjacent variables can always be extended to a third neighboring variable; this is

PATH CONSISTENCY

also called **path consistency**.

STRONGLY K-CONSISTENT

A graph is **strongly** k**-consistent** if it is k-consistent and is also $(k-1)$-consistent, $(k-2)$-consistent, ... all the way down to 1-consistent. Now suppose we have a CSP problem with n nodes and make it strongly n-consistent (i.e., strongly k-consistent for $k=n$). We can then solve the problem with no backtracking. First, we choose a consistent value for X_1. We are then guaranteed to be able to choose a value for X_2 because the graph is 2-consistent, for X_3 because it is 3-consistent, and so on. For each variable X_i, we need only search through the d values in the domain to find a value consistent with $X_1, \ldots, X_{i-1}$. We are guaranteed to find a solution in time $O(nd)$. Of course, there is no free lunch: any algorithm for establishing n-consistency must take time exponential in n in the worst case.

There is a broad middle ground between n-consistency and arc consistency: running stronger consistency checks will take more time, but will have a greater effect in reducing the branching factor and detecting inconsistent partial assignments. It is possible to calculate the smallest value k such that running k-consistency ensures that the problem can be solved without backtracking (see Section 5.4), but this is often impractical. In practice, determining the appropriate level of consistency checking is mostly an empirical science.

Handling special constraints

Certain types of constraints occur frequently in real problems and can be handled using special-purpose algorithms that are more efficient than the general-purpose methods described so far. For example, the *Alldiff* constraint says that all the variables involved must have distinct values (as in the cryptarithmetic problem). One simple form of inconsistency detection for *Alldiff* constraints works as follows: if there are m variables involved in the constraint, and if they have n possible distinct values altogether, and $m > n$, then the constraint cannot be satisfied.

This leads to the following simple algorithm: First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

We can use this method to detect the inconsistency in the partial assignment $\{WA = red, NSW = red\}$ for Figure 5.1. Notice that the variables SA, NT, and Q are effectively connected by an *Alldiff* constraint because each pair must be a different color. After applying AC-3 with the partial assignment, the domain of each variable is reduced to $\{green, blue\}$. That is, we have three variables and only two colors, so the *Alldiff* constraint is violated. Thus, a simple consistency procedure for a higher-order constraint is sometimes more effec-

tive than applying arc consistency to an equivalent set of binary constraints.

RESOURCE CONSTRAINT

Perhaps the most important higher-order constraint is the **resource constraint**, sometimes called the $atmost$ constraint. For example, let $PA_1, \ldots, PA_4$ denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as $atmost(10, PA_1, PA_2, PA_3, PA_4)$. An inconsistency can be detected simply by checking the sum of the minimum values of the current domains; for example, if each variable has the domain $\{3, 4, 5, 6\}$, the $atmost$ constraint cannot be satisfied. We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains. Thus, if each variable in our example has the domain $\{2, 3, 4, 5, 6\}$, the values 5 and 6 can be deleted from each domain.

For large resource-limited problems with integer values—such as logistical problems involving moving thousands of people in hundreds of vehicles—it is usually not possible to represent the domain of each variable as a large set of integers and gradually reduce that set by consistency checking methods. Instead, domains are represented by upper and lower bounds and are managed by bounds propagation. For example, let's suppose there are two flights, 271 and 272, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then

$$Flight271 \in [0, 165] \quad \text{and} \quad Flight272 \in [0, 385] \ .$$

Now suppose we have the additional constraint that the two flights together must carry 420 people: $Flight271 + Flight272 \in [420, 420]$. Propagating bounds constraints, we reduce the domains to

$$Flight271 \in [35, 165] \quad \text{and} \quad Flight272 \in [255, 385] \ .$$

We say that a CSP is bounds-consistent if for every variable X, and for both the lower bound and upper bound values of X, there exists some value of Y that satisfies the constraint between X and Y, for every variable Y. This kind of **bounds propagation** is widely used in practical constraint problems.

BOUNDS PROPAGATION

Intelligent backtracking: looking backward

The BACKTRACKING-SEARCH algorithm in Figure 5.3 has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking**, because the *most recent* decision point is revisited. In this subsection, we will see that there are much better ways.

CHRONOLOGICAL BACKTRACKING

Consider what happens when we apply simple backtracking in Figure 5.1 with a fixed variable ordering Q, NSW, V, T, SA, WA, NT. Suppose we have generated the partial assignment $\{Q = red, NSW = green, V = blue, T = red\}$. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot resolve the problem with South Australia.

A more intelligent approach to backtracking is to go all the way back to one of the set of variables that *caused the failure*. This set is called the **conflict set**; here, the conflict set for SA is $\{Q, NSW, V\}$. In general, the conflict set for variable X is the set of previously assigned variables that are connected to X by constraints. The **backjumping** method

CONFLICT SET

BACKJUMPING

backtracks to the *most recent* variable in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V. This is easily implemented by modifying BACKTRACKING-SEARCH so that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, it should return the most recent element of the conflict set along with the failure indicator.

The sharp-eyed reader will have noticed that forward checking can supply the conflict set with no extra work: whenever forward checking based on an assignment to X deletes a value from Y's domain, it should add X to Y's conflict set. Also, every time the last value is deleted from Y's domain, the variables in the conflict set of Y are added to the conflict set of X. Then, when we get to Y, we know immediately where to backtrack if needed.

The eagle-eyed reader will have noticed something odd: backjumping occurs when every value in a domain is in conflict with the current assignment; but forward checking detects this event and prevents the search from ever reaching such a node! In fact, it can be shown that *every branch pruned by backjumping is also pruned by forward checking.* Hence, simple backjumping is redundant in a forward-checking search or, indeed, in a search that uses stronger consistency checking, such as MAC.

Despite the observations of the preceding paragraph, the idea behind backjumping remains a good one: to backtrack based on the reasons for failure. Backjumping notices failure when a variable's domain becomes empty, but in many cases a branch is doomed long before this occurs. Consider again the partial assignment $\{WA = red, NSW = red\}$ (which, from our earlier discussion, is inconsistent). Suppose we try $T = red$ next and then assign NT, Q, V, SA. We know that no assignment can work for these last four variables, so eventually we run out of values to try at NT. Now, the question is, where to backtrack? Backjumping cannot work, because NT *does* have values consistent with the preceding assigned variables—NT doesn't have a complete conflict set of preceding variables that caused it to fail. We know, however, that the four variables NT, Q, V, and SA, *taken together*, failed because of a set of preceding variables, which must be those variables which directly conflict with the four. This leads to a deeper notion of the conflict set for a variable such as NT: it is that set of preceding variables that caused NT, *together with any subsequent variables*, to have no consistent solution. In this case, the set is WA and NSW, so the algorithm should backtrack to NSW and skip over Tasmania. A backjumping algorithm that uses conflict sets defined in this way is called **conflict-directed backjumping**.

CONFLICT-DIRECTED BACKJUMPING

We must now explain how these new conflict sets are computed. The method is in fact very simple. The "terminal" failure of a branch of the search always occurs because a variable's domain becomes empty; that variable has a standard conflict set. In our example, SA fails, and its conflict set is (say) $\{WA, NT, Q\}$. We backjump to Q, and Q *absorbs* the conflict set from SA (minus Q itself, of course) into its own direct conflict set, which is $\{NT, NSW\}$; the new conflict set is $\{WA, NT, NSW\}$. That is, there is no solution from Q onwards, given the preceding assignment to $\{WA, NT, NSW\}$. Therefore, we backtrack to NT, the most recent of these. NT absorbs $\{WA, NT, NSW\} - \{NT\}$ into its own direct conflict set $\{WA\}$, giving $\{WA, NSW\}$ (as stated in the previous paragraph). Now the algorithm backjumps to NSW, as we would hope. To summarize: let X_j be the current variable, and let $conf(X_j)$ be its conflict set. If every possible value for X_j fails, backjump

to the most recent variable X_i in $conf(X_j)$, and set

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\} .$$

CONSTRAINT LEARNING

Conflict-directed backjumping takes us back to the right point in the search tree, but doesn't prevent us from making the same mistakes in another branch of the tree. **Constraint learning** actually modifies the CSP by adding a new constraint that is induced from these conflicts.

5.3 Local Search for Constraint Satisfaction Problems

Local-search algorithms (see Section 4.3) turn out to be very effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the successor function usually works by changing the value of one variable at a time. For example, in the 8-queens problem, the initial state might be a random configuration of 8 queens in 8 columns, and the successor function picks one queen and considers moving it elsewhere in its column. Another possibility would be start with the 8 queens, one per column in a permutation of the 8 rows, and to generate a successor by having two queens swap rows.[2] We have actually already seen an example of local search for CSP solving: the application of hill climbing to the n-queens problem (page 112). The application of WALKSAT (page 223) to solve satisfiability problems, which are a special case of CSPs, is another.

MIN-CONFLICTS

In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts** heuristic. The algorithm is shown in Figure 5.8 and its application to an 8-queens problem is diagrammed in Figure 5.9 and quantified in Figure 5.5.

Min-conflicts is surprisingly effective for many CSPs, particularly when given a reasonable initial state. Its performance is shown in the last column of Figure 5.5. Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the runtime of min-conflicts is roughly *independent of problem size*. It solves even the *million*-queens problem in an average of 50 steps (after the initial assignment). This remarkable observation was the stimulus leading to a great deal of research in the 1990s on local search and the distinction between easy and hard problems, which we take up in Chapter 7. Roughly speaking, n-queens is easy for local search because solutions are densely distributed throughout the state space. Min-conflicts also works well for hard problems. For example, it has been used to schedule observations for the Hubble Space Telescope, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.

Another advantage of local search is that it can be used in an online setting when the problem changes. This is particularly important in scheduling problems. A week's airline schedule may involve thousands of flights and tens of thousands of personnel assignments, but bad weather at one airport can render the schedule infeasible. We would like to repair the schedule with a minimum number of changes. This can be easily done with a local search algorithm starting from the current schedule. A backtracking search with the new set of

[2] Local search can easily be extended to CSPs with objective functions. In that case, all the techniques for hill climbing and simulated annealing can be applied to optimize the objective function.

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
      if current is a solution for csp then return current
      var ← a randomly chosen, conflicted variable from VARIABLES[csp]
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var = value in current
  return failure
```

Figure 5.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

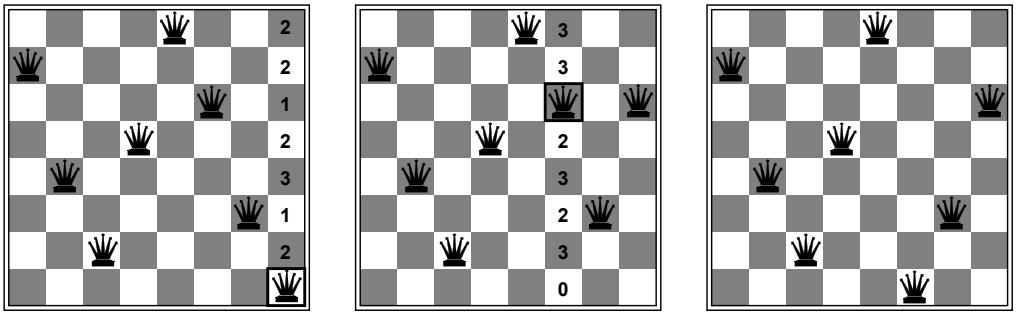


Figure 5.9 A two-step solution for an 8-queens problem using min-conflicts. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflict square, breaking ties randomly.

constraints usually requires much more time and might find a solution with many changes from the current schedule.

5.4 THE STRUCTURE OF PROBLEMS

In this section, we examine ways in which the *structure* of the problem, as represented by the constraint graph, can be used to find solutions quickly. Most of the approaches here are very general and are applicable to other problems besides CSPs, for example probabilistic reasoning. After all, the only way we can possibly hope to deal with the real world is to decompose it into many subproblems. Looking again at Figure 5.1(b) with a view to identifying problem

INDEPENDENT SUBPROBLEMS

CONNECTED COMPONENTS

structure, one fact stands out: Tasmania is not connected to the mainland.[3] Intuitively, it is obvious that coloring Tasmania and coloring the mainland are **independent subproblems**—any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map. Independence can be ascertained simply by looking for **connected components** of the constraint graph. Each component corresponds to a subproblem CSP_i. If assignment S_i is a solution of CSP_i, then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$. Why is this important? Consider the following: suppose each CSP_i has c variables from the total of n variables, where c is a constant. Then there are n/c subproblems, each of which takes at most d^c work to solve. Hence, the total work is $O(d^c n/c)$, which is *linear* in n; without the decomposition, the total work is $O(d^n)$, which is exponential in n. Let's make this more concrete: dividing a Boolean CSP with $n = 80$ into four subproblems with $c = 20$ reduces the worst-case solution time from the lifetime of the universe down to less than a second.

Completely independent subproblems are delicious, then, but rare. In most cases, the subproblems of a CSP are connected. The simplest case is when the constraint graph forms a **tree**: any two variables are connected by at most one path. Figure 5.10(a) shows a schematic example.[4] We will show that *any tree-structured CSP can be solved in time linear in the number of variables.* The algorithm has the following steps:

1. Choose any variable as the root of the tree, and order the variables from the root to the leaves in such a way that every node's parent in the tree precedes it in the ordering. (See Figure 5.10(b).) Label the variables $X_1, \ldots, X_n$ in order. Now, every variable except the root has exactly one parent variable.
2. For j from n down to 2, apply arc consistency to the arc (X_i, X_j), where X_i is the parent of X_j, removing values from DOMAIN$[X_i]$ as necessary.
3. For j from 1 to n, assign any value for X_j consistent with the value assigned for X_i, where X_i is the parent of X_j.

There are two key points to note. First, after step 2 the CSP is directionally arc-consistent, so the assignment of values in step 3 requires no backtracking. (See the discussion of k-consistency on page 147.) Second, by applying the arc-consistency checks in reverse order in step 2, the algorithm ensures that any deleted values cannot endanger the consistency of arcs that have been processed already. The complete algorithm runs in time $O(nd^2)$.

Now that we have an efficient algorithm for trees, we can consider whether more general constraint graphs can be *reduced* to trees somehow. There are two primary ways to do this, one based on removing nodes and one based on collapsing nodes together.

The first approach involves assigning values to some variables so that the remaining variables form a tree. Consider the constraint graph for Australia, shown again in Figure 5.11(a). If we could delete South Australia, the graph would become a tree, as in (b). Fortunately, we can do this (in the graph, not the continent) by fixing a value for SA and deleting from the domains of the other variables any values that are inconsistent with the value chosen for SA.

[3] A careful cartographer or patriotic Tasmanian might object that Tasmania should not be colored the same as its nearest mainland neighbor, to avoid the impression that it *might* be part of that state.

[4] Sadly, very few regions of the world, with the possible exception of Sulawesi, have tree-structured maps.

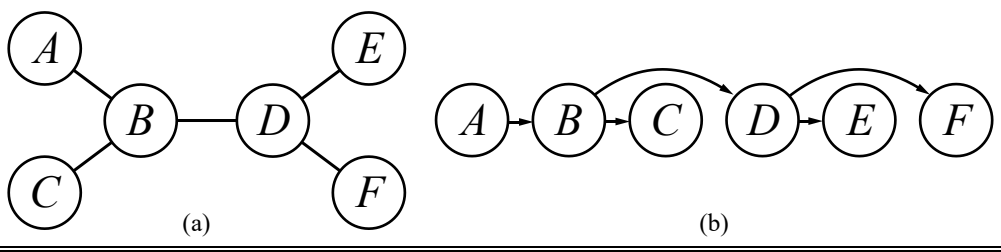


Figure 5.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root.

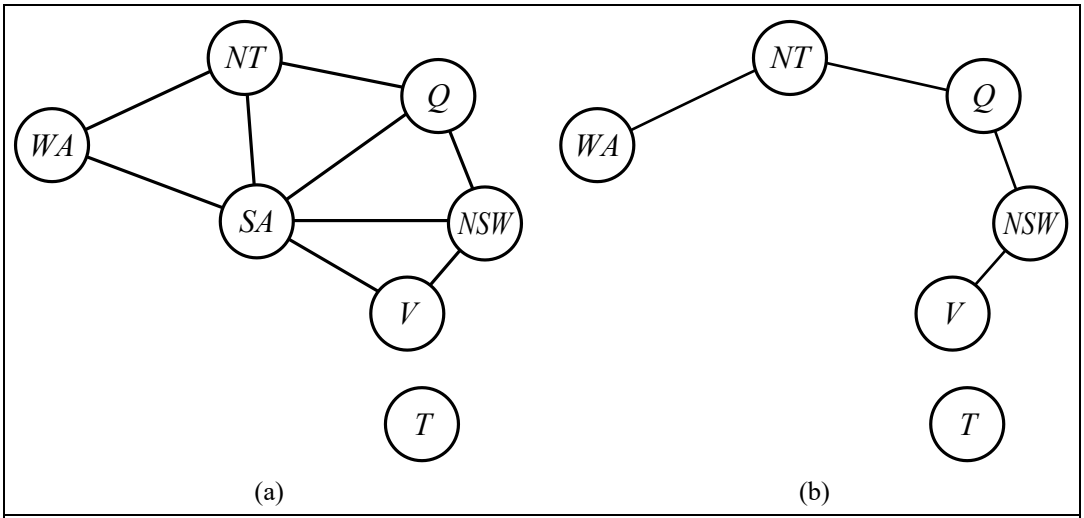


Figure 5.11 (a) The original constraint graph from Figure 5.1. (b) The constraint graph after the removal of SA.

Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA. (This works for binary CSPs; the situation is more complicated with higher-order constraints.) Therefore, we can solve the remaining tree with the algorithm given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring) the value chosen for SA could be the wrong one, so we would need to try each of them. The general algorithm is as follows:

CYCLE CUTSET

1. Choose a subset S from VARIABLES[csp] such that the constraint graph becomes a tree after removal of S. S is called a **cycle cutset**.
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and
   (b) If the remaining CSP has a solution, return it together with the assignment for S.

If the cycle cutset has size c, then the total runtime is $O(d^c \cdot (n-c)d^2)$. If the graph is "nearly a tree" then c will be small and the savings over straight backtracking will be huge.

In the worst case, however, c can be as large as $(n-2)$. Finding the *smallest* cycle cutset is NP-hard, but several efficient approximation algorithms are known for this task. The overall algorithmic approach is called **cutset conditioning**; we will see it again in Chapter 14, where it is used for reasoning about probabilities.

CUTSET CONDITIONING

TREE DECOMPOSITION

The second approach is based on constructing a **tree decomposition** of the constraint graph into a set of connected subproblems. Each subproblem is solved independently, and the resulting solutions are then combined. Like most divide-and-conquer algorithms, this works well if no subproblem is too large. Figure 5.12 shows a tree decomposition of the map-coloring problem into five subproblems. A tree decomposition must satisfy the following three requirements:

- Every variable in the original problem appears in at least one of the subproblems.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
- If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

The first two conditions ensure that all the variables and constraints are represented in the decomposition. The third condition seems rather technical, but simply reflects the constraint that any given variable must have the same value in every subproblem in which it appears; the links joining subproblems in the tree enforce this constraint. For example, SA appears in all four of the connected subproblems in Figure 5.12. You can verify from Figure 5.11 that this decomposition makes sense.

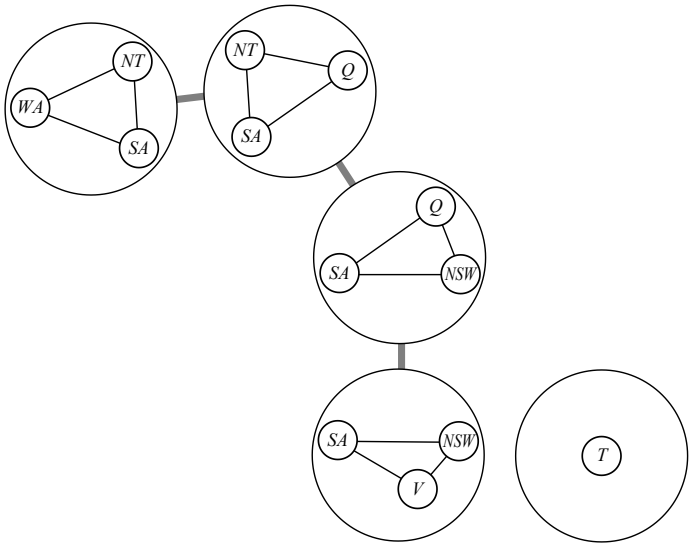


Figure 5.12 A tree decomposition of the constraint graph in Figure 5.11(a).

We solve each subproblem independently; if any one has no solution, we know the entire problem has no solution. If we can solve all the subproblems, then we attempt to construct

a global solution as follows. First, we view each subproblem as a "mega-variable" whose domain is the set of all solutions for the subproblem. For example, the leftmost subproblems in Figure 5.12 is a map-coloring problem with three variables and hence has six solutions—one is $\{WA = red, SA = blue, NT = green\}$. Then, we solve the constraints connecting the subproblems using the efficient algorithm for trees given earlier. The constraints between subproblems simply insist that the subproblem solutions agree on their shared variables. For example, given the solution $\{WA = red, SA = blue, NT = green\}$ for the first subproblem, the only consistent solution for the next subproblem is $\{SA = blue, NT = green, Q = red\}$.

TREE WIDTH

A given constraint graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible. The **tree width** of a tree decomposition of a graph is one less than the size of the largest subproblem; the tree width of the graph itself is defined to be the minimum tree width among all its tree decompositions. If a graph has tree width w, and we are given the corresponding tree decomposition, then the problem can be solved in $O(nd^{w+1})$ time. Hence, *CSPs with constraint graphs of bounded tree width are solvable in polynomial time.* Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice.

5.5 SUMMARY

- **Constraint satisfaction problems** (or CSPs) consist of variables with constraints on them. Many important real-world problems can be described as CSPs. The structure of a CSP can be represented by its **constraint graph**.
- **Backtracking search**, a form of depth-first search, is commonly used for solving CSPs.
- The **minimum remaining values** and **degree** heuristics are domain-independent methods for deciding which variable to choose next in a backtracking search. The **least-constraining-value** heuristic helps in ordering the variable values.
- By propagating the consequences of the partial assignments that it constructs, the backtracking algorithm can reduce greatly the branching factor of the problem. **Forward checking** is the simplest method for doing this. **Arc consistency** enforcement is a more powerful technique, but can be more expensive to run.
- Backtracking occurs when no legal assignment can be found for a variable. **Conflict-directed backjumping** backtracks directly to the source of the problem.
- Local search using the **min-conflicts** heuristic has been applied to constraint satisfaction problems with great success.
- The complexity of solving a CSP is strongly related to the structure of its constraint graph. Tree-structured problems can be solved in linear time. **Cutset conditioning** can reduce a general CSP to a tree-structured one and is very efficient if a small cutset can be found. **Tree decomposition** techniques transform the CSP into a tree of subproblems and are efficient if the **tree width** of the constraint graph is small.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

DIOPHANTINE EQUATIONS

The earliest work related to constraint satisfaction dealt largely with numerical constraints. Equational constraints with integer domains were studied by the Indian mathematician Brahmagupta in the seventh century; they are often called **Diophantine equations**, after the Greek mathematician Diophantus (c. 200–284), who actually considered the domain of positive rationals. Systematic methods for solving linear equations by variable elimination were studied by Gauss (1829); the solution of linear inequality constraints goes back to Fourier (1827).

GRAPH COLORING

Finite-domain constraint satisfaction problems also have a long history. For example, **graph coloring** (of which map coloring is a special case) is an old problem in mathematics. According to Biggs *et al.* (1986), the four-color conjecture (that every planar graph can be colored with four or fewer colors) was first made by Francis Guthrie, a student of de Morgan, in 1852. It resisted solution—despite several published claims to the contrary—until a proof was devised, with the aid of a computer, by Appel and Haken (1977).

Specific classes of constraint satisfaction problems occur throughout the history of computer science. One of the most influential early examples was the SKETCHPAD system (Sutherland, 1963), which solved geometric constraints in diagrams and was the forerunner of modern drawing programs and CAD tools. The identification of CSPs as a *general* class is due to Ugo Montanari (1974). The reduction of higher-order CSPs to purely binary CSPs with auxiliary variables (see Exercise 5.11) is due originally to the 19th-century logician Charles Sanders Peirce. It was introduced into the CSP literature by Dechter (1990b) and was elaborated by Bacchus and van Beek (1998). CSPs with preferences among solutions are studied widely in the optimization literature; see Bistarelli *et al.* (1997) for a generalization of the CSP framework to allow for preferences. The bucket-elimination algorithm (Dechter, 1999) can also be applied to optimization problems.

Backtracking search for constraint satisfaction is due to Bitner and Reingold (1975), although they trace the basic algorithm back to the 19th century. Bitner and Reingold also introduced the MRV heuristic, which they called the *most-constrained-variable* heuristic. Brelaz (1979) used the degree heuristic as a tie-breaker after applying the MRV heuristic. The resulting algorithm, despite its simplicity, is still the best method for k-coloring arbitrary graphs. Haralick and Elliot (1980) proposed the *least-constraining-value heuristic*.

Constraint propagation methods were popularized by Waltz's (1975) success on polyhedral line-labeling problems for computer vision. Waltz showed that, in many problems, propagation completely eliminates the need for backtracking. Montanari (1974) introduced the notion of constraint networks and propagation by path consistency. Alan Mackworth (1977) proposed the AC-3 algorithm for enforcing arc consistency as well as the general idea of combining backtracking with some degree of consistency enforcement. AC-4, a more efficient arc consistency algorithm, was developed by Mohr and Henderson (1986). Soon after Mackworth's paper appeared, researchers began experimenting with the tradeoff between the cost of consistency enforcement and the benefits in terms of search reduction. Haralick and Elliot (1980) favored the minimal forward checking algorithm described by McGregor (1979), whereas Gaschnig (1979) suggested full arc consistency checking after each variable

assignment—an algorithm later called MAC by Sabin and Freuder (1994). The latter paper provides somewhat convincing evidence that, on harder CSPs, full arc consistency checking pays off. Freuder (1978, 1982) investigated the notion of k-consistency and its relationship to the complexity of solving CSPs. Apt (1999) describes a generic algorithmic framework within which consistency propagation algorithms can be analyzed.

Special methods for handling higher-order constraints have been developed primarily within the context of **constraint logic programming**. Marriott and Stuckey (1998) provide excellent coverage of research in this area. The *Alldiff* constraint was studied by Regin (1994). Bounds constraints were incorporated into constraint logic programming by Van Hentenryck *et al.* (1998).

The basic backjumping method is due to John Gaschnig (1977, 1979). Kondrak and van Beek (1997) showed that this algorithm is essentially subsumed by forward checking. Conflict-directed backjumping was devised by Prosser (1993). The most general and powerful form of intelligent backtracking was actually developed very early on by Stallman and Sussman (1977). Their technique of **dependency-directed backtracking** led to the development of **truth maintenance systems** (Doyle, 1979), which we will discuss in Section 10.8. The connection between the two areas is analyzed by de Kleer (1989).

DEPENDENCY-DIRECTED BACKTRACKING

The work of Stallman and Sussman also introduced the idea of **constraint recording**, in which partial results obtained by search can be saved and reused later in the search. The idea was introduced formally into backtracking search by Dechter (1990a). **Backmarking** (Gaschnig, 1979) is a particularly simple method in which consistent and inconsistent pairwise assignments are saved and used to avoid rechecking constraints. Backmarking can be combined with conflict-directed backjumping; Kondrak and van Beek (1997) present a hybrid algorithm that provably subsumes either method taken separately. The method of **dynamic backtracking** (Ginsberg, 1993) retains successful partial assignments from later subsets of variables when backtracking over an earlier choice that does not invalidate the later success.

CONSTRAINT RECORDING

BACKMARKING

DYNAMIC BACKTRACKING

Local search in constraint satisfaction problems was popularized by the work of Kirkpatrick *et al.* (1983) on **simulated annealing** (see Chapter 4), which is widely used for scheduling problems. The *min-conflicts* heuristic was first proposed by Gu (1989) and was developed independently by Minton *et al.* (1992). Sosic and Gu (1994) showed how it could be applied to solve the 3,000,000 queens problem in less than a minute. The astounding success of local search using min-conflicts on the n-queens problem led to a reappraisal of the nature and prevalence of "easy" and "hard" problems. Peter Cheeseman *et al.* (1991) explored the difficulty of randomly generated CSPs and discovered that almost all such problems either are trivially easy or have no solutions. Only if the parameters of the problem generator are set in a certain narrow range, within which roughly half of the problems are solvable, do we find "hard" problem instances. We discuss this phenomenon further in Chapter 7.

Work relating the structure and complexity of CSPs originates with Freuder (1985), who showed that search on arc-consistent trees works without any backtracking. A similar result, with extensions to acyclic hypergraphs, was developed in the database community (Beeri *et al.*, 1983). Since those papers were published, there has been a great deal of progress in developing more general results relating the complexity of solving a CSP to the structure of

its constraint graph. The notion of tree width was introduced by the graph theorists Robertson and Seymour (1986). Dechter and Pearl (1987, 1989), building on the work of Freuder, applied the same notion (which they called **induced width**) to constraint satisfaction problems and developed the tree decomposition approach sketched in Section 5.4. Drawing on this work and on results from database theory, Gottlob *et al.* (1999a, 1999b) developed a notion, **hypertree width**, that is based on the characterization of the CSP as a hypergraph. In addition to showing that any CSP with hypertree width w can be solved in time $O(n^{w+1} \log n)$, they also showed that hypertree width subsumes all previously defined measures of "width" in the sense that there are cases where the hypertree width is bounded and the other measures are unbounded.

There are several good surveys of CSP techniques, including those by Kumar (1992), Dechter and Frost (1999), and Bartak (2001); and the encyclopedia articles by Dechter (1992) and Mackworth (1992). Pearson and Jeavons (1997) survey tractable classes of CSPs, covering both structural decomposition methods and methods that rely on properties of the domains or constraints themselves. Kondrak and van Beek (1997) give an analytical survey of backtracking search algorithms, and Bacchus and van Run (1995) give a more empirical survey. The texts by Tsang (1993) and by Marriott and Stuckey (1998) go into much more depth than has been possible in this chapter. Several interesting applications are described in the collection edited by Freuder and Mackworth (1994). Papers on constraint satisfaction appear regularly in *Artificial Intelligence* and in the specialist journal, *Constraints*. The primary conference venue is the International Conference on Principles and Practice of Constraint Programming, often called *CP*.

EXERCISES

5.1 Define in your own words the terms constraint satisfaction problem, constraint, backtracking search, arc consistency, backjumping and min-conflicts.

5.2 How many solutions are there for the map-coloring problem in Figure 5.1?

5.3 Explain why it is a good heuristic to choose the variable that is *most* constrained, but the value that is *least* constraining in a CSP search.

5.4 Consider the problem of constructing (not solving) crossword puzzles:[5] fitting words into a rectangular grid. The grid, which is given as part of the problem, specifies which squares are blank and which are shaded. Assume that a list of words (i.e., a dictionary) is provided and that the task is to fill in the blank squares using any subset of the list. Formulate this problem precisely in two ways:

a. As a general search problem. Choose an appropriate search algorithm, and specify a heuristic function, if you think one is needed. Is it better to fill in blanks one letter at a time or one word at a time?

[5] Ginsberg *et al.* (1990) discuss several methods for constructing crossword puzzles. Littman *et al.* (1999) tackle the harder problem of solving them.

b. As a constraint satisfaction problem. Should the variables be words or letters?

Which formulation do you think will be better? Why?

5.5 Give precise formulations for each of the following as constraint satisfaction problems:

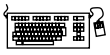
FLOOR-PLANNING

a. Rectilinear **floor-planning**: find nonoverlapping places in a large rectangle for a number of smaller rectangles.

CLASS SCHEDULING

b. **Class scheduling**: There is a fixed number of professors and classrooms, a list of classes to be offered, and a list of possible time slots for classes. Each professor has a set of classes that he or she can teach.

5.6 Solve the cryptarithmetic problem in Figure 5.2 by hand, using backtracking, forward checking, and the MRV and least-constraining-value heuristics.



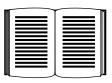
5.7 Figure 5.5 tests out various algorithms on the n-queens problem. Try these same algorithms on map-coloring problems generated randomly as follows: scatter n points on the unit square; selecting a point X at random, connect X by a straight line to the nearest point Y such that X is not already connected to Y and the line crosses no other line; repeat the previous step until no more connections are possible. Construct the performance table for the largest n you can manage, using both $d = 3$ and $d = 4$ colors. Comment on your results.

5.8 Use the AC-3 algorithm to show that arc consistency is able to detect the inconsistency of the partial assignment $\{WA = red, V = blue\}$ for the problem shown in Figure 5.1.

5.9 What is the worst-case complexity of running AC-3 on a tree-structured CSP?

5.10 AC-3 puts back on the queue *every* arc (X_k, X_i) whenever *any* value is deleted from the domain of X_i, even if each value of X_k is consistent with several remaining values of X_i. Suppose that, for every arc (X_k, X_i), we keep track of the number of remaining values of X_i that are consistent with each value of X_k. Explain how to update these numbers efficiently and hence show that arc consistency can be enforced in total time $O(n^2d^2)$.

5.11 Show how a single ternary constraint such as "$A + B = C$" can be turned into three binary constraints by using an auxiliary variable. You may assume finite domains. (*Hint:* consider a new variable that takes on values which are pairs of other values, and consider constraints such as "X is the first element of the pair Y.") Next, show how constraints with more than three variables can be treated similarly. Finally, show how unary constraints can be eliminated by altering the domains of variables. This completes the demonstration that any CSP can be transformed into a CSP with only binary constraints.



5.12 Suppose that a graph is known to have a cycle cutset of no more than k nodes. Describe a simple algorithm for finding a minimal cycle cutset whose runtime is not much more than $O(n^k)$ for a CSP with n variables. Search the literature for methods for finding approximately minimal cycle cutsets in time that is polynomial in the size of the cutset. Does the existence of such algorithms make the cycle cutset method practical?

5.13 Consider the following logic puzzle: In five houses, each with a different color, live 5 persons of different nationalities, each of whom prefer a different brand of cigarette, a

different drink, and a different pet. Given the following facts, the question to answer is "Where does the zebra live, and in which house do they drink water?"

The Englishman lives in the red house.
The Spaniard owns the dog.
The Norwegian lives in the first house on the left.
Kools are smoked in the yellow house.
The man who smokes Chesterfields lives in the house next to the man with the fox.
The Norwegian lives next to the blue house.
The Winston smoker owns snails.
The Lucky Strike smoker drinks orange juice.
The Ukrainian drinks tea.
The Japanese smokes Parliaments.
Kools are smoked in the house next to the house where the horse is kept.
Coffee is drunk in the green house.
The Green house is immediately to the right (your right) of the ivory house.
Milk is drunk in the middle house.

Discuss different representations of this problem as a CSP. Why would one prefer one representation over another?