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 atomic, or indivisible—a black box with no internal structure.

This chapter describes a way to solve a wide variety of problems more efficiently. We use a factored representation for each state: a set of variables, each of which has a value. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a constraint satisfaction problem, or CSP.

CSP search algorithms take advantage of the structure of states and use general-purpose rather than problem-specific heuristics to enable the solution of complex problems. The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

### 6.1 Defining Constraint Satisfaction Problems

A constraint satisfaction problem consists of three components, $X$, $D$, and $C$:

- $X$ is a set of variables, $\{X_1, \ldots, X_n\}$.
- $D$ is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.
- $C$ is a set of constraints that specify allowable combinations of values.

Each domain $D_i$ consists of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable $X_i$. Each constraint $C_i$ consists of a pair $\langle \text{scope}, \text{rel} \rangle$, where $\text{scope}$ is a tuple of variables that participate in the constraint and $\text{rel}$ is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. For example, if $X_1$ and $X_2$ both have...
the domain \{A,B\}, then the constraint saying the two variables must have different values can be written as \((X_1, X_2), (A, B), (B, A)\) or as \((X_1, X_2), X_1 \neq X_2\).

To solve a CSP, we need to define a state space and the notion of a solution. Each state in a CSP 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 assigned, and a solution to a CSP is a consistent, complete assignment. A partial assignment is one that assigns values to only some of the variables.

### 6.1.1 Example problem: Map coloring

Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories (Figure 6.1(a)). 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

\[
X = \{WA, NT, Q, NSW, V, SA, T\}
\]

The domain of each variable is the set \(D_i = \{\text{red, green, blue}\}\). The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

\[
C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.
\]

Here we are using abbreviations; \(SA \neq WA\) is a shortcut for \(((SA, WA), SA \neq WA)\), where \(SA \neq WA\) can be fully enumerated in turn as

\[
\{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), (\text{blue, red}), (\text{blue, green})\}.
\]

There are many possible solutions to this problem, such as

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

It can be helpful to visualize a CSP as a constraint graph, as shown in Figure 6.1(b). The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint.

Why formulate a problem as a CSP? One reason is that the CSPs yield a natural representation for a wide variety of problems; if you already have a CSP-solving system, it is often easier to solve a problem using it than to design a custom solution using another search technique. In addition, CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space. For example, once we have chosen \(\{SA = \text{blue}\}\) in the Australia problem, we can conclude that none of the five neighboring variables can take on the value \(\text{blue}\). Without taking advantage of constraint propagation, a search procedure would have to consider \(3^5 = 243\) assignments for the five neighboring variables; with constraint propagation we never have to consider \(\text{blue}\) as a value, so we have only \(2^5 = 32\) assignments to look at, a reduction of 87%.

In regular state-space search we can only ask: is this specific state a goal? No? What about this one? With CSPs, once we find out that a partial assignment is not a solution, we can
Figure 6.1  (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). 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.

immediately discard further refinements of the partial assignment. Furthermore, we can see why the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter. As a result, many problems that are intractable for regular state-space search can be solved quickly when formulated as a CSP.

6.1.2 Example problem: Job-shop scheduling

Factories have the problem of scheduling a day’s worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques. Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes. Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once. Constraints can also specify that a task takes a certain amount of time to complete.

We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly. We can represent the tasks with 15 variables:

\[ X = \{ Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, \\
Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect \} . \]

The value of each variable is the time that the task starts. Next we represent precedence constraints between individual tasks. Whenever a task \( T_1 \) must occur before task \( T_2 \), and task \( T_1 \) takes duration \( d_1 \) to complete, we add an arithmetic constraint of the form

\[ T_1 + d_1 \leq T_2 . \]
In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

\[ \text{Axle}_F + 10 \leq \text{Wheel}_{RF}; \quad \text{Axle}_F + 10 \leq \text{Wheel}_{LF}; \]
\[ \text{Axle}_B + 10 \leq \text{Wheel}_{RB}; \quad \text{Axle}_B + 10 \leq \text{Wheel}_{LB}. \]

Next we say that, for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

\[ \text{Wheel}_{RF} + 1 \leq \text{Nuts}_{RF}; \quad \text{Nuts}_{RF} + 2 \leq \text{Cap}_{RF}; \]
\[ \text{Wheel}_{LF} + 1 \leq \text{Nuts}_{LF}; \quad \text{Nuts}_{LF} + 2 \leq \text{Cap}_{LF}; \]
\[ \text{Wheel}_{RB} + 1 \leq \text{Nuts}_{RB}; \quad \text{Nuts}_{RB} + 2 \leq \text{Cap}_{RB}; \]
\[ \text{Wheel}_{LB} + 1 \leq \text{Nuts}_{LB}; \quad \text{Nuts}_{LB} + 2 \leq \text{Cap}_{LB}. \]

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a **disjunctive constraint** to say that \( \text{Axle}_F \) and \( \text{Axle}_B \) must not overlap in time; either one comes first or the other does:

\(( \text{Axle}_F + 10 \leq \text{Axle}_B ) \quad \text{or} \quad ( \text{Axle}_B + 10 \leq \text{Axle}_F ).\)

This looks like a more complicated constraint, combining arithmetic and logic. But it still reduces to a set of pairs of values that \( \text{Axle}_F \) and \( \text{Axle}_B \) can take on.

We also need to assert that the inspection comes last and takes 3 minutes. For every variable except \( \text{Inspect} \) we add a constraint of the form \( X + d_X \leq \text{Inspect} \). Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables:

\[ D_i = \{1, 2, 3, \ldots, 27\}. \]

This particular problem is trivial to solve, but CSPs have been applied to job-shop scheduling problems like this with thousands of variables. In some cases, there are complicated constraints that are difficult to specify in the CSP formalism, and more advanced planning techniques are used, as discussed in Chapter 11.

### 6.1.3 Variations on the CSP formalism

The simplest kind of CSP involves variables that have **discrete, finite domains**. Map-coloring problems and scheduling with time limits are both 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 \( D_i = \{1, 2, 3, 4, 5, 6, 7, 8\} \).

A discrete domain can be **infinite**, such as the set of integers or strings. (If we didn’t put a deadline on the job-scheduling problem, there would be an infinite number of start times for each variable.) 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 that understands constraints such as \( T_1 + d_1 \leq T_2 \) directly, without enumerating the set of pairs of allowable values for \((T_1, T_2)\). Special solution algorithms (which we do 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.
Constraint satisfaction problems with **continuous domains** are 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 equalities or inequalities. 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.

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, in the map-coloring problem it could be the case that South Australians won’t tolerate the color green; we can express that with the unary constraint \((SA), \overline{SA} \neq \text{green}\).

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 6.1(b).

We can also describe higher-order constraints, such as asserting that the value of \(Y\) is between \(X\) and \(Z\), with the ternary constraint \(\text{Between}(X,Y,Z)\).

A constraint involving an arbitrary number of variables is called a **global constraint**. (The name is traditional but confusing because it need not involve all the variables in a problem). One of the most common global constraints is \(\text{Alldiff}\), which says that all of the variables involved in the constraint must have different values. In Sudoku problems (see Section 6.2.6), all variables in a row or column must satisfy an \(\text{Alldiff}\) constraint. Another example is provided by **cryptarithmetic** puzzles. (See Figure 6.2(a).) Each letter in a cryptarithmetic puzzle represents a different digit. For the case in Figure 6.2(a), this would be represented as the global constraint \(\text{Alldiff}(F,T,U,W,R,O)\). The addition constraints on the four columns of the puzzle can be written as the following \(n\)-ary constraints:

\[
\begin{align*}
O + O &= R + 10 \cdot C_{10} \\
C_{10} + W + W &= U + 10 \cdot C_{100} \\
C_{100} + T + T &= O + 10 \cdot C_{1000} \\
C_{1000} &= F,
\end{align*}
\]

where \(C_{10}, C_{100},\) and \(C_{1000}\) are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column. These constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 6.2(b). A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent \(n\)-ary constraints.

Alternatively, as Exercise 6.6 asks you to prove, every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced, so we could transform any CSP into one with only binary constraints; this makes the algorithms simpler. Another way to convert an \(n\)-ary CSP to a binary one is the **dual graph** transformation: create a new graph in which there will be one variable for each constraint in the original graph, and
one binary constraint for each pair of constraints in the original graph that share variables. For example, if the original graph has variables \( \{X, Y, Z\} \) and constraints \( \langle (X, Y, Z), C_1 \rangle \) and \( \langle (X, Y), C_2 \rangle \) then the dual graph would have variables \( \{C_1, C_2\} \) with the binary constraint \( \langle (X, Y), R_1 \rangle \), where \( (X, Y) \) are the shared variables and \( R_1 \) is a new relation that defines the constraint between the shared variables, as specified by the original \( C_1 \) and \( C_2 \).

There are however two reasons why we might prefer a global constraint such as \textit{Alldiff} rather than a set of binary constraints. First, it is easier and less error-prone to write the problem description using \textit{Alldiff}. Second, it is possible to design special-purpose inference algorithms for global constraints that are not available for a set of more primitive constraints. We describe these inference algorithms in Section 6.2.5.

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 \textit{preference constraints} indicating which solutions are preferred. For example, in a university class-scheduling problem there are absolute constraints that no professor can teach two classes at the same time. But we also may allow preference constraints: Prof. R might prefer teaching in the morning, whereas Prof. N prefers teaching in the afternoon. A schedule that has Prof. R teaching at 2 p.m. would still be an allowable solution (unless Prof. R 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. R costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved with optimization search methods, either path-based or local. We call such a problem a \textit{constraint optimization problem}, or COP. Linear programming problems do this kind of optimization.
In regular state-space search, an algorithm can do only one thing: search. In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of inference called constraint propagation: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on. Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

The key idea is local consistency. If we treat each variable as a node in a graph (see Figure 6.1(b)) and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

6.2.1 Node consistency

A single variable (corresponding to a node in the CSP network) is node-consistent if all the values in the variable’s domain satisfy the variable’s unary constraints. For example, in the variant of the Australia map-coloring problem (Figure 6.1) where South Australians dislike green, the variable $SA$ starts with domain \{\textit{red, green, blue}\}, and we can make it node consistent by eliminating \textit{green}, leaving $SA$ with the reduced domain \{\textit{red, blue}\}. We say that a network is node-consistent if every variable in the network is node-consistent.

It is always possible to eliminate all the unary constraints in a CSP by running node consistency. It is also possible to transform all $n$-ary constraints into binary ones (see Exercise 6.6). Because of this, it is common to define CSP solvers that work with only binary constraints; we make that assumption for the rest of this chapter, except where noted.

6.2.2 Arc consistency

A variable in a CSP is arc-consistent if every value in its domain satisfies the variable’s binary constraints. More formally, $X_i$ is arc-consistent with respect to another variable $X_j$ if for every value in the current domain $D_i$ there is some value in the domain $D_j$ that satisfies the binary constraint on the arc $(X_i, X_j)$. A network is arc-consistent if every variable is arc consistent with every other variable. For example, consider the constraint $Y = X^2$ where the domain of both $X$ and $Y$ is the set of digits. We can write this constraint explicitly as

$$\langle (X, Y), \{(0, 0), (1, 1), (2, 4), (3, 9)\}\rangle.$$  

To make $X$ arc-consistent with respect to $Y$, we reduce $X$’s domain to \{0, 1, 2, 3\}. If we also make $Y$ arc-consistent with respect to $X$, then $Y$’s domain becomes \{0, 1, 4, 9\} and the whole CSP is arc-consistent.

On the other hand, arc consistency can do nothing for the Australia map-coloring problem. Consider the following inequality constraint on $(SA, WA)$:

$$\{(\textit{red, green}), (\textit{red, blue}), (\textit{green, red}), (\textit{green, blue}), (\textit{blue, red}), (\textit{blue, green})\}.$$
Section 6.2. Constraint Propagation: Inference in CSPs

No matter what value you choose for \(SA\) (or for \(WA\)), there is a valid value for the other variable. So applying arc consistency has no effect on the domains of either variable.

The most popular algorithm for arc consistency is called AC-3 (see Figure 6.3). To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider. (Actually, the order of consideration is not important, so the data structure is really a set, but tradition calls it a queue.) Initially, the queue contains all the arcs in the CSP. AC-3 then pops off an arbitrary arc \((X_i, X_j)\) from the queue and makes \(X_i\) arc-consistent with respect to \(X_j\). If this leaves \(D_i\) unchanged, the algorithm just moves on to the next arc. But if this revises \(D_i\) (makes the domain smaller), then we add to the queue all arcs \((X_k, X_i)\) where \(X_k\) is a neighbor of \(X_i\). We need to do that because the change in \(D_i\) might enable further reductions in the domains of \(D_k\), even if we have previously considered \(X_k\). If \(D_i\) is revised down to nothing, then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure. Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue. At that point, we are left with a CSP that is equivalent to the original CSP—they both have the same solutions—but the arc-consistent CSP will in most cases be faster to search because its variables have smaller domains.

The complexity of AC-3 can be analyzed as follows. Assume a CSP with \(n\) variables, each with domain size at most \(d\), and with \(c\) binary constraints (arcs). Each arc \((X_k, X_i)\) can be inserted in the queue only \(d\) times because \(X_i\) has at most \(d\) values to delete. Checking

---

### Figure 6.3 The arc-consistency algorithm AC-3

<table>
<thead>
<tr>
<th>function</th>
<th>AC-3(csp) returns false if an inconsistency is found and true otherwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs:</td>
<td>csp, a binary CSP with components ((X, D, C))</td>
</tr>
<tr>
<td>local variables:</td>
<td>queue, a queue of arcs, initially all the arcs in csp</td>
</tr>
<tr>
<td>while</td>
<td>queue is not empty do</td>
</tr>
<tr>
<td>(Xi, Xj) ← REMOVE-FIRST(queue)</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td>REVISE(csp, Xi, Xj) then</td>
</tr>
<tr>
<td>if</td>
<td>size of (D_i) = 0 then return false</td>
</tr>
<tr>
<td>for each</td>
<td>(X_k) in (X_i.NEIGHBORS - {X_j}) do</td>
</tr>
<tr>
<td>add (Xk, Xi) to queue</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>true</td>
</tr>
</tbody>
</table>

| function | REVISE(csp, Xi, Xj) returns true iff we revise the domain of \(X_i\) |
| revised ← false |
| for each | \(x\) in \(D_i\) do |
| if | no value \(y\) in \(D_j\) allows \((x, y)\) to satisfy the constraint between \(X_i\) and \(X_j\) then |
| delete \(x\) from \(D_i\) |
| revised ← true |
| return | revised |

No matter what value you choose for \(SA\) (or for \(WA\)), there is a valid value for the other variable. So applying arc consistency has no effect on the domains of either variable.
consistency of an arc can be done in $O(d^2)$ time, so we get $O(cd^3)$ total worst-case time.\footnote{The AC-4 algorithm (Mohr and Henderson, 1986) runs in $O(cd^2)$ worst-case time but can be slower than AC-3 on average cases. See Exercise 6.13.}

It is possible to extend the notion of arc consistency to handle $n$-ary rather than just binary constraints; this is called generalized arc consistency or sometimes hyperarc consistency, depending on the author. A variable $X_i$ is \textbf{generalized arc consistent} with respect to an $n$-ary constraint if for every value $v$ in the domain of $X_i$ there exists a tuple of values that is a member of the constraint, has all its values taken from the domains of the corresponding variables, and has its $X_i$ component equal to $v$. For example, if all variables have the domain $\{0, 1, 2, 3\}$, then to make the variable $X$ consistent with the constraint $X < Y < Z$, we would have to eliminate 2 and 3 from the domain of $X$ because the constraint cannot be satisfied when $X$ is 2 or 3.

\subsection*{6.2.3 Path consistency}

Arc consistency can go a long way toward reducing the domains of variables, sometimes finding a solution (by reducing every domain to size 1) and sometimes finding that the CSP cannot be solved (by reducing some domain to size 0). But for other networks, arc consistency fails to make enough inferences. Consider the map-coloring problem on Australia, but with only two colors allowed, red and blue. Arc consistency can do nothing because every variable is already arc consistent: each can be red with blue at the other end of the arc (or vice versa). But clearly there is no solution to the problem: because Western Australia, Northern Territory and South Australia all touch each other, we need at least three colors for them alone.

Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints). To make progress on problems like map coloring, we need a stronger notion of consistency. \textbf{Path consistency} tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable $X_m$ if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to $X_m$ that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$. This is called path consistency because one can think of it as looking at a path from $X_i$ to $X_j$ with $X_m$ in the middle.

Let’s see how path consistency fares in coloring the Australia map with two colors. We will make the set $\{WA, SA\}$ path consistent with respect to $NT$. We start by enumerating the consistent assignments to the set. In this case, there are only two: $\{WA = \text{red}, SA = \text{blue}\}$ and $\{WA = \text{blue}, SA = \text{red}\}$. We can see that with both of these assignments $NT$ can be neither red nor blue (because it would conflict with either $WA$ or $SA$). Because there is no valid choice for $NT$, we eliminate both assignments, and we end up with no valid assignments for $\{WA, SA\}$. Therefore, we know that there can be no solution to this problem. The PC-2 algorithm (Mackworth, 1977) achieves path consistency in much the same way that AC-3 achieves arc consistency. Because it is so similar, we do not show it here.
6.2.4 *K*-consistency

Stronger forms of propagation can be defined with the notion of *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 *k*th variable. 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called node consistency. 2-consistency is the same as arc consistency. For binary constraint networks, 3-consistency is the same as path consistency.

A CSP 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 with *n* nodes and make it strongly *n*-consistent (i.e., strongly *k*-consistent for *k* = *n*). We can then solve the problem as follows: 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*ₖ, we need only search through the *d* values in the domain to find a value consistent with *X*₁,...,*X*ₖ₋₁. We are guaranteed to find a solution in time *O*(*n*²*d*). Of course, there is no free lunch: any algorithm for establishing *n*-consistency must take time exponential in *n* in the worst case. Worse, *n*-consistency also requires space that is exponential in *n*. The memory issue is even more severe than the time. In practice, determining the appropriate level of consistency checking is mostly an empirical science. It can be said practitioners commonly compute 2-consistency and less commonly 3-consistency.

6.2.5 Global constraints

Remember that a global constraint is one involving an arbitrary number of variables (but not necessarily all variables). Global constraints occur frequently in real problems and can be handled by 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 above and Sudoku puzzles below). One simple form of inconsistency detection for *Alldiff* constraints works as follows: if *m* variables are 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.

This method can detect the inconsistency in the assignment \{*WA* = *red*, *NSW* = *red*\} for Figure 6.1. Notice that the variables *SA*, *NT*, and *Q* are effectively connected by an *Alldiff* constraint because each pair must have two different colors. 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 effective than applying arc consistency to an equivalent set of binary constraints. There are more
complex inference algorithms for \textit{Alldiff} (see van Hoeve and Katriel, 2006) that propagate more constraints but are more computationally expensive to run.

Another important higher-order constraint is the \textbf{resource constraint}, sometimes called the \textit{atmost} constraint. For example, in a scheduling problem, let $P_1, \ldots, P_4$ 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 $Atmost(10, P_1, P_2, P_3, P_4)$. We can detect an inconsistency 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 \textit{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 \textbf{bounds propagation}. For example, in an airline-scheduling problem, let’s suppose there are two flights, $F_1$ and $F_2$, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then

$$D_1 = [0, 165] \quad \text{and} \quad D_2 = [0, 385].$$

Now suppose we have the additional constraint that the two flights together must carry 420 people: $F_1 + F_2 = 420$. Propagating bounds constraints, we reduce the domains to

$$D_1 = [35, 165] \quad \text{and} \quad D_2 = [255, 385].$$

We say that a CSP is \textbf{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.

\subsection*{6.2.6 Sudoku example}

The popular \textbf{Sudoku} puzzle has introduced millions of people to constraint satisfaction problems, although they may not recognize it. A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or $3 \times 3$ box (see Figure 6.4). A row, column, or box is called a \textbf{unit}.

The Sudoku puzzles that are printed in newspapers and puzzle books have the property that there is exactly one solution. Although some can be tricky to solve by hand, taking tens of minutes, even the hardest Sudoku problems yield to a CSP solver in less than 0.1 second.

A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names $A1$ through $A9$ for the top row (left to right), down to $I1$ through $I9$ for the bottom row. The empty squares have the domain $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and the pre-filled squares have a domain consisting of a single value. In addition, there are 27 different
Section 6.2. Constraint Propagation: Inference in CSPs

Let us see how far arc consistency can take us. Assume that the \textit{Alldiff} constraints have been expanded into binary constraints (such as $A_1 \neq A_2$) so that we can apply the AC-3 algorithm directly. Consider variable $E_6$ from Figure 6.4(a)—the empty square between the 2 and the 8 in the middle box. From the constraints in the box, we can remove not only 2 and 8 but also 1 and 7 from $E_6$'s domain. From the constraints in its column, we can eliminate 5, 6, 2, 4 (since we now know $E_6$ must be 4), 8, 9, and 3. That leaves $E_6$ with a domain of \{4\}; in other words, we know the answer for $E_6$.

Now consider variable $I_6$—the square in the bottom middle box surrounded by 1, 3, and 3. Applying arc consistency in its column, we eliminate 5, 6, 2, 4 (since we now know $E_6$ must be 4), 8, 9, and 3. We eliminate 1 by arc consistency with $I_5$, and we are left with only the value 7 in the domain of $I_6$. Now there are 8 known values in column 6, so arc consistency can infer that $A_6$ must be 1. Inference continues along these lines, and eventually, AC-3 can solve the entire puzzle—all the variables have their domains reduced to a single value, as shown in Figure 6.4(b).

Of course, Sudoku would soon lose its appeal if every puzzle could be solved by a
mechanical application of AC-3, and indeed AC-3 works only for the easiest Sudoku puzzles. Slightly harder ones can be solved by PC-2, but at a greater computational cost: there are 255,960 different path constraints to consider in a Sudoku puzzle. To solve the hardest puzzles and to make efficient progress, we will have to be more clever.

Indeed, the appeal of Sudoku puzzles for the human solver is the need to be resourceful in applying more complex inference strategies. Aficionados give them colorful names, such as “naked triples.” That strategy works as follows: in any unit (row, column or box), find three squares that each have a domain that contains the same three numbers or a subset of those numbers. For example, the three domains might be \{1, 8\}, \{3, 8\}, and \{1, 3, 8\}. From that we don’t know which square contains 1, 3, or 8, but we do know that the three numbers must be distributed among the three squares. Therefore we can remove 1, 3, and 8 from the domains of every other square in the unit.

It is interesting to note how far we can go without saying much that is specific to Sudoku. We do of course have to say that there are 81 variables, that their domains are the digits 1 to 9, and that there are 27 \textit{Alldiff} constraints. But beyond that, all the strategies—arc consistency, path consistency, etc.—apply generally to all CSPs, not just to Sudoku problems. Even naked triples is really a strategy for enforcing consistency of \textit{Alldiff} constraints and has nothing to do with Sudoku \textit{per se}. This is the power of the CSP formalism: for each new problem area, we only need to define the problem in terms of constraints; then the general constraint-solving mechanisms can take over.

### 6.3 Backtracking Search for CSPs

Sudoku problems are designed to be solved by inference over constraints. But many other CSPs cannot be solved by inference alone; there comes a time when we must search for a solution. In this section we look at backtracking search algorithms that work on partial assignments; in the next section we look at local search algorithms over complete assignments.

We could apply a standard depth-limited search (from Chapter 3). A state would be a partial assignment, and an action would be adding \textit{var} = \textit{value} to the assignment. But for a CSP with \(n\) variables of domain size \(d\), 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!

Our seemingly reasonable but naive formulation ignores crucial property common to all CSPs: \textit{commutativity}. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order. Therefore, we need only consider 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 make a choice between \(SA = \text{red}, \ SA = \text{green},\) and \(SA = \text{blue}\), but we would never choose between \(SA = \text{red}\) and \(WA = \text{blue}\). With this restriction, the number of leaves is \(d^n\), as we would hope.
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
return BACKTRACK({}, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
if assignment is complete then return assignment
var ← SELECT-UNASSIGNED-VARIABLE(csp)
for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
  if value is consistent with assignment then
    add \{ var = value \} to assignment
    inferences ← INFERENCE(csp, var, value)
    if inferences ≠ failure then
      add inferences to assignment
      result ← BACKTRACK(assignment, csp)
      if result ≠ failure then
        return result
    remove \{ var = value \} and inferences from assignment
  return failure

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

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 6.5. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value. Part of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order WA, NT, Q, ... Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

Notice that BACKTRACKING-SEARCH keeps only a single representation of a state and alters that representation rather than creating new ones, as described on page 87.

In Chapter 3 we improved 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 knowledge. Instead, we can add some sophistication to the unspecified functions in Figure 6.5, using them to address the following questions:

1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?
2. What inferences should be performed at each step in the search (INERENCE)?

3. When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

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

### 6.3.1 Variable and value ordering

The backtracking algorithm contains the line

```
var ← SELECT UNASSIGNED VARIABLE(csp).
```

The simplest strategy for `SELECT UNASSIGNED VARIABLE` is to choose the next unassigned variable in order, \{X₁, X₂, \ldots\}. This static variable ordering seldom results in the most efficient search. For example, after the assignments for \(WA = \text{red}\) and \(NT = \text{green}\) in Figure 6.6, there is only one possible value for \(SA\), so it makes sense to assign \(SA = \text{blue}\) 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 some variable \(X\) has no legal values left, the MRV heuristic will select \(X\) and failure will be detected immediately—avoiding pointless searches through other variables. The MRV heuristic usually performs better than a random or static ordering, sometimes by a factor of 1,000 or more, although the results vary widely depending on the problem.

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 6.1, \(SA\) is the variable with highest degree, 5; the other variables have degree 2 or 3, except for \(T\), which has degree 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.

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 6.1 we have generated the partial assignment with \( WA = \text{red} \) and \( NT = \text{green} \) 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.

Why should variable selection be fail-first, but value selection be fail-last? It turns out that, for a wide variety of problems, a variable ordering that chooses a variable with the minimum number of remaining values helps minimize the number of nodes in the search tree by pruning larger parts of the tree earlier. For value ordering, the trick is that we only need one solution; therefore it makes sense to look for the most likely values first. If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.

### 6.3.2 Interleaving search and inference

So far we have seen how AC-3 and other algorithms can infer reductions in the domain of variables before we begin the search. But inference can be even more powerful in the course of a search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables.

One of the simplest forms of inference is called forward checking. Whenever a variable \( X \) is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable \( Y \) that is connected to \( X \) by a constraint, delete from \( Y \)'s domain any value that is inconsistent with the value chosen for \( X \). Because forward checking only does arc consistency inferences, there is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

Figure 6.7 shows the progress of backtracking search on the Australia CSP with forward checking. There are two important points to notice about this example. First, notice that after \( WA = \text{red} \) and \( Q = \text{green} \) are assigned, 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 \( Q \). A second point to notice is that after \( V = \text{blue} \), the domain of \( SA \) is empty. Hence, forward checking has detected that the partial assignment \( \{ WA = \text{red}, Q = \text{green}, V = \text{blue} \} \) is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.

For many problems the search will be more effective if we combine the MRV heuristic with forward checking. Consider Figure 6.7 after assigning \( \{ WA = \text{red} \} \). Intuitively, it seems that that assignment constrains its neighbors, \( NT \) and \( SA \), so we should handle those
variables next, and then all the other variables will fall into place. That’s exactly what happens with MRV: NT and SA have two values, so one of them is chosen first, then the other, then Q, NSW, and V in order. Finally T still has three values, and any one of them works. We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.

Although forward checking detects many inconsistencies, it does not detect all of them. The problem is that it makes the current variable arc-consistent, but doesn’t look ahead and make all the other variables arc-consistent. For example, consider the third row of Figure 6.7. It shows that when WA is red and Q is green, both NT and SA are forced to be blue. Forward checking does not look far enough ahead to notice that this is an inconsistency: NT and SA are adjacent and so cannot have the same value.

The algorithm called MAC (for Maintaining Arc Consistency (MAC)) detects this inconsistency. After a variable $X_i$ is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs $(X_j, X_i)$ for all $X_j$ that are unassigned variables that are neighbors of $X_i$. From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately. We can see that MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC’s queue; but unlike MAC, forward checking does not recursively propagate constraints when changes are made to the domains of variables.

6.3.3 Intelligent backtracking: Looking backward

The BACKTRACKING-SEARCH algorithm in Figure 6.5 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 consider better possibilities.

Consider what happens when we apply simple backtracking in Figure 6.1 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