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 \texttt{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 \texttt{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 \texttt{vec = 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 \(d^n\) 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 \(n! d^n\) leaves, even though there are only \(d^n\) possible complete assignments!

Our seemingly reasonable but naive formulation ignores crucial property common to all CSPs: 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 \texttt{SA = red, SA = green}, and \texttt{SA = blue}, but we would \textbf{never} choose between \texttt{SA = red} and \texttt{WA = 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({})

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 0 failure then
                add inferences to assignment
                result = BACKTRACK(assignment, csp)
                if result 0 failure then
                    return result
                remove {var = value} and inferences from assignment
                return failure
    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)?