magic squares of an arbitrary order $n \geq 3$, which are especially efficient for odd $n$'s (e.g., [Pic02]). Of course, these algorithms are not based on exhaustive search: the number of candidate solutions the exhaustive search algorithm would have to consider becomes prohibitively large even for a computer for $n$ as small as 5. Indeed, $(5^2)! \simeq 1.5 \cdot 10^{25}$, and hence it would take a computer making 10 trillion operations per second about 49,000 years to finish the job.

## Backtracking

There are two major difficulties in applying exhaustive search. The first one lies in the mechanics of generating all possible solution candidates. For some problems, such candidates compose a well-structured set. For example, candidate arrangements of the first nine positive integers in the cells of the $3 \times 3$ table (see the *Magic Square* example above) can be obtained as *permutations* of these numbers, for which several algorithms are known. There are many problems, however, where solution candidates do not form a set with such a regular structure. The second, and more fundamental, difficulty lies in the number of solution candidates that need to be generated and processed. Typically, the size of this set grows at least exponentially with the problem size. Therefore, exhaustive search is practical only for very small instances of such problems.

*Backtracking* is an important improvement over the brute-force approach of exhaustive search. It provides a convenient method for generating candidate solutions while making it possible to avoid generating unnecessary candidates. The main idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows: If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for *any* remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with the next option for that component.

Typically, backtracking involves undoing a number of wrong choices—the smaller this number, the faster the algorithm finds a solution. Although in the worst-case scenario a backtracking algorithm may end up generating all the same candidate solutions as an exhaustive search, this rarely happens.

It is convenient to interpret a backtracking algorithm as a process of constructing a tree that mirrors decisions being made. Computer scientists use the term *tree* to describe hierarchical structures such as family trees and organizational charts. A tree is usually shown with its *root* (the only node without a parent) on the top and its *leaves* (nodes without children) on or closer to the bottom of the diagram. This is nothing but a convenient typographical convention, however. For a backtracking algorithm, such a tree is called a *state-space tree*. The root of a state-space tree corresponds to the start of a solution construction process; we consider the root to be on the zero level of the tree. The root's children—on the first level of the tree—correspond to possible choices of the first component

of a solution (e.g., the cell to contain 1 in the magic square construction). Their children—the nodes on the second level—correspond to possible choices of the second component of a solution, and so on. Leaves can be of two kinds. The first kind—called *nonpromising nodes* or *dead ends*—correspond to partially constructed candidates that cannot lead to a solution. After establishing that a particular node is nonpromising, a backtracking algorithm terminates the node (the tree is said to be *pruned*), undoes the decision regarding the last component of the candidate solution by backtracking to the parent of the nonpromising node, and considers another choice for that component. The second kind of a leaf provides a solution to the problem. If a single solution suffices, the algorithm stops; if other solutions need to be searched for, the algorithm continues searching for them by backtracking to the leaf's parent.

The following example is a perennial favorite for showing an application of backtracking to a particular problem.

*The n-Queens Problem* Place $n$ queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same column, row, or diagonal.

For $n = 1$, the problem has a trivial solution, and it is easy to see that there is no solution for $n = 2$ and $n = 3$. So let us consider the 4-queens problem and solve it by backtracking. Since each of the four queens has to be placed in its own column, all we need to do is to assign a row for each queen on the board shown in Figure 1.2.

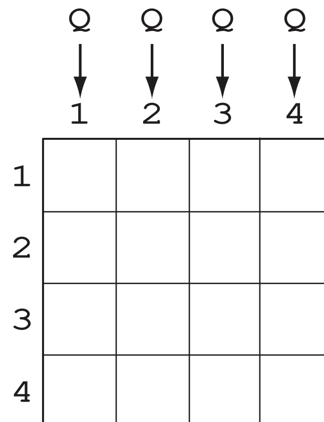


FIGURE 1.2 Board for the 4-queens problem.

We start with the empty board and then place queen 1 in the first possible position, which is in row 1 of column 1. Then we place queen 2, after trying unsuccessfully rows 1 and 2 of the second column, in the first acceptable position for it, which is square (3, 2), the square in row 3 and column 2. This proves to be a dead end because there is no acceptable position in the third column for

queen 3. Therefore, the algorithm backtracks and puts queen 2 in the next possible position (4, 2). Then queen 3 is placed at (2, 3), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (2, 1). Queen 2 then goes to (4, 2), queen 3 to (1, 3), and queen 4 to (3, 4), which is a solution to the problem. The state-space tree of this search is given in Figure 1.3.

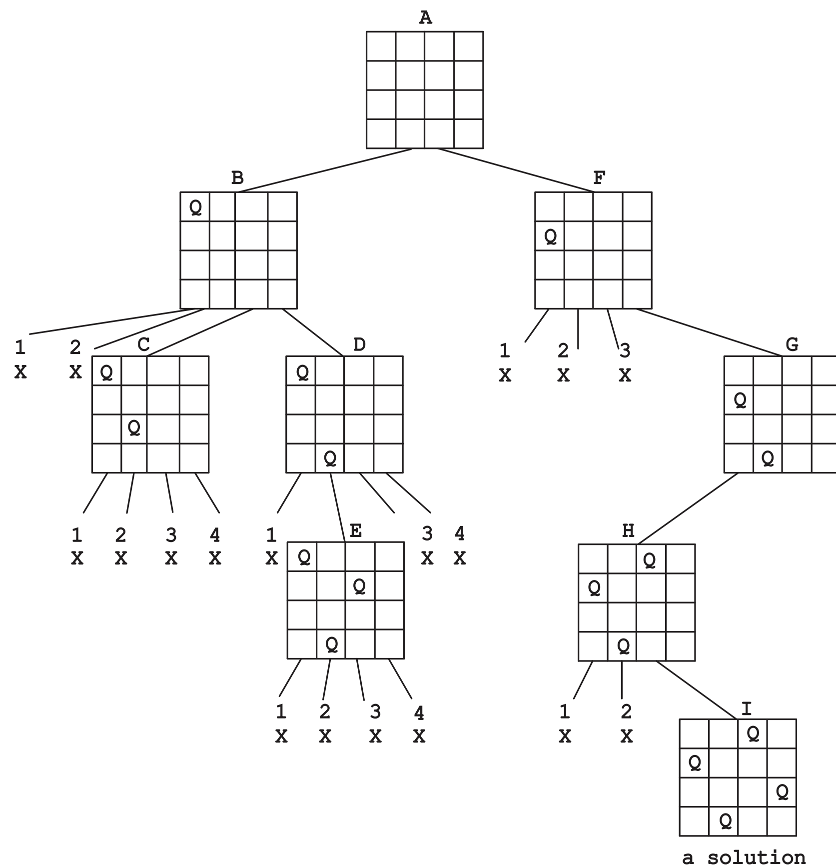


FIGURE 1.3 State-space tree of finding a solution to the 4-queens problem by backtracking. X denotes an unsuccessful attempt to place a queen in the indicated row. The letters above the nodes show the order in which the nodes are generated.

If other solutions need to be found (there is just one other solution to the 4-queens problem), the algorithm can simply resume its operations at the leaf at which it stopped. Alternatively, one can use the board's symmetry for this purpose.

How much faster is this solution by backtracking compared to exhaustive search? If we are to consider all possible placements of four queens on four different squares of the $4 \times 4$ board, the number of such placements is

$$\frac{16!}{4!(16-4)!} = \frac{16 \cdot 15 \cdot 14 \cdot 13}{4 \cdot 3 \cdot 2} = 1820.$$

(The general formula for the number of ways to choose $k$ different objects, the order of which is not of interest, from a given set of $n$ different objects, called by mathematicians *combinations* of $n$ objects taken $k$ at a time and denoted by either $\binom{n}{k}$ or $C(n,k)$, is $\frac{n!}{k!(n-k)!}$.) If we consider only the placements with the queens in different columns, the total number of solution candidates decreases to $4^4 = 256$. And if we add to the latter the constraint that the queens must also be in different rows, the number of choices drops to $4! = 24$. While the last number is quite manageable, it would not be the case for larger instances of the problem. For example, for a regular $8 \times 8$ chessboard, the number of such solution candidates is $8! = 40{,}320$.

The reader might be interested to know that the total number of different solutions to the 8-queens problem is 92, twelve of which are qualitatively distinct, with the remaining 80 obtainable from the basic twelve by rotations and reflections. As to the general $n$-queens problem, it has a solution for every $n \geq 4$ but no convenient formula for the number of solutions for an arbitrary $n$ has been discovered. It is known that this number grows very fast with the value of $n$. For example, the number of solutions for $n = 10$ is 724, of which 92 are distinct, while for $n = 12$ the respective numbers are 14,200 and 1787.

Many puzzles in this book can be solved by backtracking. For each of them, however, there is a more efficient algorithm the reader is expected to strive for. In particular, *The n-Queens Problem Revisited* (#140) in the main section of the book asks the reader to design a much faster algorithm for the $n$-queens problem.

## Decrease-and-Conquer

The *decrease-and-conquer* strategy is based on finding a relationship between a solution to a given problem and a solution to its smaller instance. Once found, such a relationship leads naturally to a *recursive algorithm*, which reduces the problem to a sequence of its diminishing instances until it becomes small enough to be solved directly.[1] Here is an example.

*Celebrity Problem* A celebrity among a group of $n$ people is a person who knows nobody but is known by everybody else. The task is to identify a celebrity by only asking questions to people of the following form: "Do you know this person?"

Assuming for simplicity that a celebrity is known to exist among a given group of $n$ people, the problem can be solved by the following decrease-by-one algorithm. If $n = 1$, that one person is vacuously a celebrity by the definition. If $n > 1$, select two people from the group, say, A and B, and ask A whether he or she knows B. If A knows B, remove A from the remaining people who can be a celebrity; if A does

[1] The notion of a *recursion* is one of the most important in computer science. The reader unfamiliar with it will find a wealth of information by, say, following the references and links in the Wikipedia article on "recursion (computer science)."