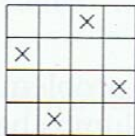


**Book: Richard Johnsonbaugh and Marcus Schaefer,
Algorithms, Pearson Education, 2004.**

BACKTRACKING

Backtracking (chapter 4.5)

The n -queens problem is to place n queens on an $n \times n$ board so that no two queens are in the same row, column, or diagonal (see Figure 4.5.1). In chess terminology, this is the problem of placing n queens on an $n \times n$ board so that no queen attacks another queen.



		×	
×			
			×
	×		

Figure 4.5.1 The 4-queens problem. Queens, shown as \times , are placed so that no two queens are in the same row, column, or diagonal.

The n -queens problem can be solved using depth-first search. The graph to which depth-first search is applied is a tree, called the **search tree**, with vertices that represent boards containing n or fewer queens. The root is the board with no queens. The tree is generated dynamically; that is, we place queens successively in the columns beginning in the left column and working from top to bottom. The next queen placed in a column is shown as a child. When it is impossible to place a queen in a column, we return to the parent and adjust the queen in the preceding column. Depth-first search applied to a dynamically generated tree is called **backtracking**. The name derives from the situation in which we cannot place the next queen and return to the parent.

Example 4.5.1. Figure 4.5.2 shows part of the search tree for the four-queens problem that is generated using backtracking.

We begin at the root with an empty board [board (0)]. We use the notation “board (i)” to indicate the board that is generated after i attempts to place

It is now impossible to place a queen in column 4 because every position conflicts with one of the queens already present. We therefore backtrack to board (9) and attempt to generate another child of (9) by moving the queen down in column 3 in board (11); however, both potential positions conflict. We therefore backtrack to board (1) and attempt to generate another child of (1) by moving the queen down in column 2 in board (9). Since the queen in column 2 is in the last row, this is impossible. We therefore backtrack to board (0), where we generate another child of (0) by moving the queen down in column 1 in board (1) [board (18)].

We are then able to generate successive children starting at board (18) until we find a solution [board (26)]. \square

We implement the backtracking solution of the n -queens problem recursively. The key function is *rn_queens*. When *rn_queens*(k, n) is called, queens have been properly placed in columns 1 through $k - 1$, and *rn_queens* tries to place a queen in column k . If it is successful and k equals n , it prints a solution. The algorithm can be terminated at this point if only one solution is desired. Our version, which outputs *all* solutions, continues. If *rn_queens*(k, n) succeeds in placing a queen in column k , and k does not equal n , it calls

$$\text{rn_queens}(k + 1, n),$$

which tries to place a queen in the next column. If *rn_queens*(k, n) does not succeed in placing a queen in column k , it backtracks by returning to its caller

$$\text{rn_queens}(k - 1, n),$$

which tries to adjust the queen in column $k - 1$. The backtracking solution is obtained by calling

$$\text{rn_queens}(1, n).$$

We track the positions of the queens by using an array *row*. The value of *row*[k] is the row in which the queen in column k is placed.

To test for a valid position for a queen in column k , we write a function *position_ok*(k, n), which returns true if the queen in column k does not conflict with the queens in columns 1 through $k - 1$ or false if it does conflict. *Position_ok*(k, n) is implemented in time constant. We maintain an array *row_used*, where *row_used*[r] is true if a queen occupies row r . To track the diagonals, we arbitrarily number the diagonals in the direction \searrow as shown in Figure 4.5.4(a). We call these the *ddiags* (downward diagonals). We maintain a second array *ddiag_used*, where *ddiag_used*[d] is true if a queen occupies *ddiag* d . Notice that the queen in column k , row r , is in *ddiag* $n - k + r$. Similarly, we arbitrarily number the diagonals in the direction \nearrow [see Figure 4.5.4(b)]. We call these the *udiags* (upward diagonals). We maintain a third array *udiag_used*, where *udiag_used*[d] is true if a queen occupies *udiag* d . Notice that the queen in column k , row r , is in *udiag* $k + r - 1$.

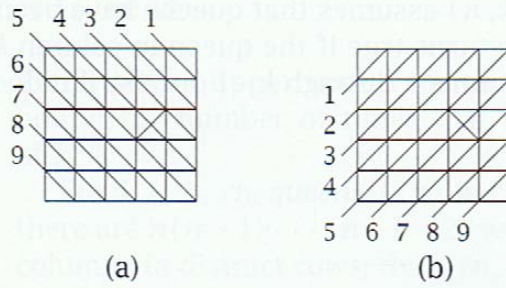


Figure 4.5.4 Board (a) shows the numbering of the diagonals in the direction \searrow in a 5×5 board. These diagonals are called the *ddiags*. The queen in column k , row r , is in *ddiag* $n - k + r$. For example, the queen in column 4, row 3, (shown as \times) is in *ddiag* $n - k + r = 5 - 4 + 3 = 4$. Board (b) shows the numbering of the diagonals in the direction \swarrow in a 5×5 board. These diagonals are called the *udiags*. The queen in column k , row r , is in *udiag* $k + r - 1$. For example, the queen in column 4, row 3, (shown as \times) is in *udiag* $k + r - 1 = 4 + 3 - 1 = 6$.

The final version of our n -queens algorithm is given as Algorithm 4.5.2.

Algorithm 4.5.2 Solving the n -Queens Problem Using Backtracking. The n -queens problem is to place n queens on an $n \times n$ board so that no two queens are in the same row, column, or diagonal. Using backtracking, this algorithm outputs all solutions to this problem. We place queens successively in the columns beginning in the left column and working from top to bottom. When it is impossible to place a queen in a column, we return to the previous column and move its queen down.

The value of $row[k]$ is the row where the queen in column k is placed.

The algorithm begins when rn_queens calls $rn_queens(1, n)$. When

$rn_queens(k, n)$

is called, queens have been properly placed in columns 1 through $k - 1$, and $rn_queens(k, n)$ tries to place a queen in column k . If it is successful and k equals n , it prints a solution. If it is successful and k does not equal n , it calls

$rn_queens(k + 1, n)$.

If it is not successful, it backtracks by returning to its caller

$rn_queens(k - 1, n)$.

The value of $row_used[r]$ is true if a queen occupies row r and false otherwise. The value of $ddiag_used[d]$ is true if a queen occupies *ddiag* diagonal d and false otherwise [see Figure 4.5.4(a)]. According to the numbering system used, the queen in column k , row r , is in *ddiag* diagonal $n - k + r$. The value of $udiag_used[d]$ is true if a queen occupies *udiag* diagonal d and false otherwise [see Figure 4.5.4(b)]. According to the numbering system used, the queen in column k , row r , is in *udiag* $k + r - 1$.

The function *position_ok(k, n)* assumes that queens have been placed in columns 1 through $k - 1$. It returns true if the queen in column k does not conflict with the queens in columns 1 through $k - 1$ or false if it does conflict.

Input Parameter: n
Output Parameters: None

```

n_queens(n) {
    for  $i = 1$  to  $n$ 
        row_used[i] = false
    for  $i = 1$  to  $2 * n - 1$ 
        ddiag_used[i] = udiag_used[i] = false
    rn_queens(1, n)
}

// When rn_queens(k, n) is called, queens have been
// properly placed in columns 1 through  $k - 1$ .
rn_queens(k, n) {
    for row[k] = 1 to  $n$ 
        if (position_ok(k, n)) {
            row_used[row[k]] = true
            ddiag_used[n - k + row[k]] = true
            udiag_used[k + row[k] - 1] = true
            if ( $k == n$ ) {
                // Output a solution. Stop if only one solution is desired.
                for  $i = 1$  to  $n$ 
                    print(row[i] + " ")
                println()
            }
            else
                rn_queens(k + 1, n)
            row_used[row[k]] = false
            ddiag_used[n - k + row[k]] = false
            udiag_used[k + row[k] - 1] = false
        }
    }

// position_ok(k, n) returns true if the queen in column  $k$ 
// does not conflict with the queens in columns 1
// through  $k - 1$  or false if it does conflict.
position_ok(k, n) {
    return !(row_used[row[k]]
        || ddiag_used[n - k + row[k]]
        || udiag_used[k + row[k] - 1])
}

```


Time for n -Queens

To obtain an upper bound for the worst-case time of Algorithm 4.5.2, we bound the number of times that $rn_queens(k, n)$ is called for each value of k .

For $k = 1$, $rn_queens(k, n)$ is called one time (by n_queens). For $k > 1$, there are $n(n-1) \cdots (n-k+2)$ ways to place $k-1$ queens in the first $k-1$ columns in distinct rows; thus, $rn_queens(k, n)$ is called at most

$$n(n-1) \cdots (n-k+2)$$

times. Ignoring recursive calls, $rn_queens(k, n)$ executes in time $\Theta(n)$ for $k < n$. Therefore, the worst-case time for $rn_queens(k, n)$ is at most

$$n, \text{ for } k = 1, \quad \text{and} \quad n[n(n-1) \cdots (n-k+2)], \text{ for } 1 < k < n. \quad (4.5.1)$$

When $k = n$, all rows except one are occupied; thus, $position_ok(k, n)$ is true for at most one value of k . The inner for loop in rn_queens executes in time $\Theta(n)$. Therefore, the time for rn_queens for $k = n$ is $O(n)$. There are $n(n-1) \cdots 2$ ways to place $n-1$ queens in the first $n-1$ columns in distinct rows; thus, the worst-case time of $rn_queens(n, n)$ is at most

$$n[n(n-1) \cdots 2]. \quad (4.5.2)$$

Combining equations (4.5.1) and (4.5.2), we find that the worst-case time of rn_queens is at most

$$\begin{aligned} & n[1 + n + n(n-1) + \cdots + n(n-1) \cdots 2] \\ &= n \cdot n! \left[\frac{1}{n!} + \frac{1}{(n-1)!} + \frac{1}{(n-2)!} + \cdots + \frac{1}{1!} \right]. \end{aligned}$$

A result from calculus tells us that

$$e = \sum_{i=0}^{\infty} \frac{1}{i!},$$

where e is the base of the natural logarithm. Therefore, the worst-case time of rn_queens is at most

$$n \cdot n! \left[\frac{1}{n!} + \frac{1}{(n-1)!} + \frac{1}{(n-2)!} + \cdots + \frac{1}{1!} \right] \leq n \cdot n! \sum_{i=1}^{\infty} \frac{1}{i!} = n \cdot n!(e - 1).$$

It follows that the worst-case time of rn_queens is $O(n \cdot n!)$. The for loops in n_queens run in time $\Theta(n)$; thus, the worst-case time for Algorithm 4.5.2 is $O(n \cdot n!)$.

Form of a Backtracking Algorithm

Suppose that we solve a problem using backtracking as in Algorithm 4.5.2 in which the solution is of the form

$$x[1], \dots, x[n].$$

Suppose also that the values of $x[i]$ are in the set S (e.g., in Algorithm 4.5.2, $S = \{1, \dots, n\}$). The general form of a backtracking algorithm becomes

```
backtrack(n) {  
    rbacktrack(1, n)  
}  
  
rbacktrack(k, n) {  
    for each  $x[k] \in S$   
        if ( $\text{bound}(k)$ )  
            if ( $k == n$ ) {  
                // Output a solution. Stop if only one solution is desired.  
                for  $i = 1$  to  $n$   
                    print( $x[i] + " "$ )  
                println()  
            }  
            else  
                rbacktrack( $k + 1, n$ )  
}
```

The function $\text{bound}(k)$ assumes that

$$x[1], \dots, x[k-1]$$

is a partial solution and that $x[k]$ has been assigned a value. It then returns true if

$$x[1], \dots, x[k]$$

is a partial solution and false otherwise. The key to writing a useful backtracking algorithm is to write an efficient bound function that eliminates many potential nodes from the search tree.

The Hamiltonian-Cycle Problem

Recall (see Section 2.5) that a Hamiltonian cycle in a graph G is a cycle in which every vertex, except the beginning and ending vertices which coincide, occurs exactly one time. For example,

$$(1, 3, 5, 2, 4, 1)$$

is a Hamiltonian cycle for the graph of Figure 4.5.5.

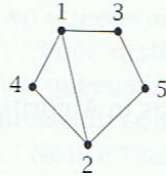


Figure 4.5.5 A graph with a Hamiltonian cycle: $(1, 3, 5, 2, 4, 1)$.

Determining whether a graph has a Hamiltonian cycle is known to be NP-complete (see Section 10.2), so there is likely no polynomial-time algorithm for this problem. However, we can write a backtracking algorithm to solve the Hamiltonian-cycle problem; the time is often acceptable for graphs of modest size.

Suppose that a graph with vertices $1, \dots, n$ has a Hamiltonian cycle

$$(x[1], x[2], \dots, x[n], x[1]).$$

We represent this cycle as the array

$$x[1], \dots, x[n].$$

Since a Hamiltonian cycle contains all of the vertices and can begin at any vertex, we assume that $x[1] = 1$. In our algorithm, if the graph has a Hamiltonian cycle, we return true and terminate the algorithm as soon as we find one Hamiltonian cycle. If the graph has no Hamiltonian cycle, we return false. We represent the graph using the adjacency matrix adj so that, given vertices i and j , we can quickly determine whether (i, j) is an edge.

Algorithm 4.5.4 Searching for a Hamiltonian Cycle. This algorithm inputs a graph with vertices $1, \dots, n$. The graph is represented as an adjacency matrix adj ; $adj[i][j]$ is true if (i, j) is an edge or false if it is not an edge. If the graph has a Hamiltonian cycle, the algorithm computes one such cycle

$$(x[1], \dots, x[n], x[1]).$$

If the graph has no Hamiltonian cycle, the algorithm returns false and the contents of the array x are not specified.

In the array $used$, $used[i]$ is true if i has been selected as one of the vertices in a potential Hamiltonian cycle or false if i has not been selected.

The function *path_ok*(*adj*, *k*, *x*) assumes that $(x[1], \dots, x[k-1])$ is a path from $x[1]$ to $x[k-1]$ and that the vertices $x[1], \dots, x[k-1]$ are distinct. It then checks whether $x[k]$ is different from each of $x[1], \dots, x[k-1]$ and whether $(x[k-1], x[k])$ is an edge. If $k = n$, *path_ok* also checks whether $(x[n], x[1])$ is an edge.

Input Parameter: *adj*
Output Parameter: *x*

```

hamilton(adj, x) {
    n = adj.last
    x[1] = 1
    used[1] = true
    for i = 2 to n
        used[i] = false
    rhamilton(adj, 2, x)
}

rhamilton(adj, k, x) {
    n = adj.last
    for x[k] = 2 to n
        if (path_ok(adj, k, x)) {
            used[x[k]] = true
            if (k == n || rhamilton(adj, k + 1, x))
                return true
            used[x[k]] = false
        }
    return false
}

path_ok(adj, k, x) {
    n = adj.last
    if (used[x[k]])
        return false
    if (k < n)
        return adj[x[k - 1]][x[k]]
    else
        return adj[x[n - 1]][x[n]] && adj[x[1]][x[n]]
}

```

The same technique we used to obtain an upper bound on the running time of the n -queens algorithm can be used to show that the worst-case time of Algorithm 4.5.4 is $O(n!)$ (see Exercise 23).

If G is the graph consisting of K_{n-1} (the complete graph on $n-1$ vertices), with vertices labeled $1, \dots, n-1$, and one additional, isolated vertex labeled n , the running time of Algorithm 4.5.4 is $\Omega((n-1)!)$ (see Exercise 24). Thus, the worst-case time of Algorithm 4.5.4 is $\Omega((n-1)!)$.