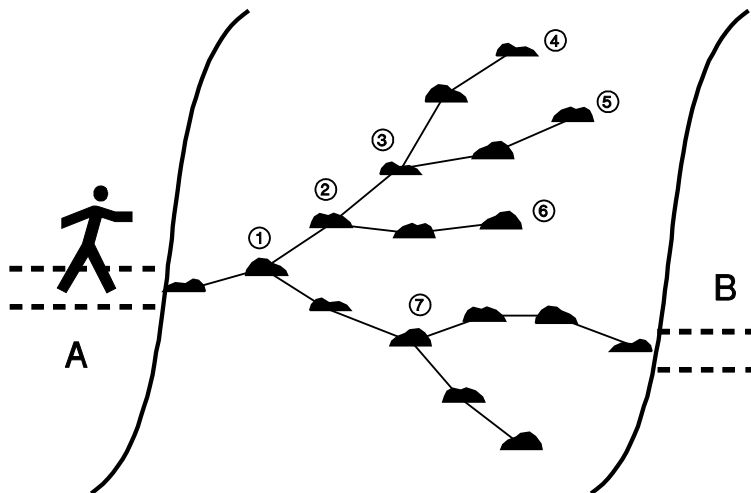


Backtracking

Backtracking is a general purpose programming strategy that is useful whenever a complete solution to a problem consists of a set of “partial” solutions that must be chosen from a large solution space. This handout describes the general approach for applying backtracking to a problem.

An Example

Before tackling a computer-based example, consider the problem of a hiker attempting to cross a stream by finding a path of stepping stones across the stream. The picture below illustrates an example.

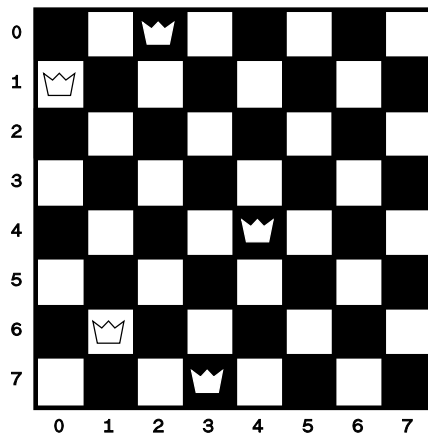


As the hiker starts across the stream, assume he decides to bear left, passing points 1, 2, and 3, and eventually reaching point 4. Recognizing that this path is a dead end, he turns around, *backtracks* to point 3, then heads on the path to point 5. Again being stymied, he returns to point 2, and takes the path toward point 6. Since this path, too, offers no complete solution he retreats to point 1, heads toward point 7, and finally finds a complete path to the other side.

In this example, the stones along a particular path represent the “partial” solutions to the problem of crossing the stream. After reaching a dead end, the hiker must “undo” part of the solution found so far in order to find another possible path. The complete solution consists of the list of stepping stones that make up the path across the stream.

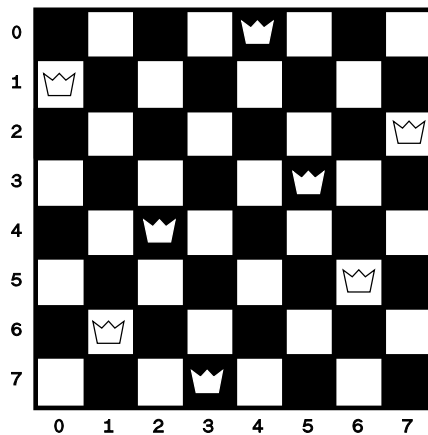
The Eight Queens Problem

Another classic backtracking problem is the Eight Queens puzzle. It is derived from the rules of chess - in particular, the moves of the queen piece. The object is to place eight queen pieces on the board such that no queen is able to attack the others (i.e., that no queen is placed in either the same row, same column, nor along any diagonal line as any other queen). This problem is typically solved manually by trial and error - queens are placed on the board, one at a time, each time checking to see if the “no attack” condition has been violated. Say, for example, that we start placing queens, one in each column of a chess board, starting from left to right, so that we arrive at the following pattern:



While the solution looks good so far (the placed queens cannot attack each other), we cannot place a queen anywhere in column 5 that doesn't violate the no-attack condition. This situation is similar to our dead-end path of stepping stones. While we could completely start over at this point, a more methodical approach would be to remove the last queen or two that we placed, (that is, to *backtrack* to an earlier point), then try some other placements in those columns to see if we can find a complete solution. Several "backtracks" are usually necessary to find a complete solution.

An aside - there are more than 10^{10} different ways to place 8 pieces on a 64 square board! While many of these possibilities represent obviously ridiculous solutions (for instance, placing all the queens in the same row or column), nonetheless the number of combinations is much too large to use a strictly trial-and-error approach. It may be comforting to know that there are also 92 solutions to the problem; one is shown below (and several more can be derived by rotating and mirroring this one):



We can represent this solution by an eight number list, with each number representing the row in which the queen for that column is placed. For example the solution shown above could be represented by the list (1, 6, 4, 7, 0, 3, 5, 2). This same scheme could also be used to represent a partial solution, like the (incorrect) one above: (1, 6, 0, 7, 4, ...).

A Computer Solution

A variety of computer-based problems (including the eight queens) can be solved with a backtracking-based algorithm. We will now present a general purpose algorithm that can be adapted for many of these problems. Several data structures and a function need to be defined in order to use this general algorithm framework:

- `space[n,m]` - a matrix that contains all of the possible partial solutions that comprise the solution space of the problem. The number of rows (the value `n`) represents the number of "partial solutions" that are required for a complete solution, and the number of columns represents the number of solution candidates that are available for each partial solution. This array is not necessarily rectangular, since the number of solution candidates for each partial solution may be different. The exact way in which the partial solutions are represented (i.e., the exact type of data that `space` contains) will vary depending upon the problem - using the backtracking framework below often requires considerable thought about how the solution information should be represented in order to best solve the problem.
- `int nspace[n]` - an array that contains the number of partial solution candidates that are available for each partial solution. Put another way, each element of `nspace` specifies the number of values in the corresponding row of `space`.
- `int sindex[n]` - an array which holds the number of each partial solution that makes up the current solution being checked. **IF** a complete solution is found, `sindex` will contain the numbers of all the partial solution candidates for the problem. (In the eight queens example, it is the list which comprises the solution).
- `bool is_soln(...)` - a function that checks whether a particular set of partial solutions comprises a possible solution, at least to this point. The routine returns `true` if the current partial solution is a legal solution, and `false` otherwise. In the case of the eight queens, this routine would check to see if any of the queens that have been placed on the board so far can be attacked by any of the other ones. This routine, like the `space` matrix, is unique to a particular problem. Writing this function is often the key to the entire problem.

As an example, if a problem is composed of 6 partial solutions, and each of the partial solutions had a different number of potential candidates, the data structures shown below might be used to represent the problem:

	space					nspace		
0	<i>Soln 0-0</i>	<i>Soln 0-1</i>	<i>Soln 0-2</i>	<i>Soln 0-3</i>		4		
1	<i>Soln 1-0</i>	<i>Soln 1-1</i>				2		
2	<i>Soln 2-0</i>	<i>Soln 2-1</i>	<i>Soln 2-2</i>	<i>Soln 2-3</i>	<i>Soln 2-4</i>		5	
3	<i>Soln 3-0</i>	<i>Soln 3-1</i>	<i>Soln 3-2</i>				3	
4	<i>Soln 4-0</i>	<i>Soln 4-1</i>	<i>Soln 4-2</i>	<i>Soln 4-3</i>				4
5	<i>Soln 5-0</i>	<i>Soln 5-1</i>	<i>Soln 5-2</i>				3	

Furthermore, if a complete solution is found that consists of the partial solutions that are shaded, then the values of `sindex` would be (2, 1, 3, 0, 1, 1).

Given these data structures, the following is the framework for a general purpose backtracking algorithm:

```
void main()
{
    // Define all data structures and other variables here

    int i = 0;        // Partial solution number
    int j = 0;        // Partial solution candidate to try

    while(i < n)
    {
        while (j < nspace[i] && (!is_soln(space, sindex, i, j)) j++;

        if(j < nspace[i])
        { // We found a possible partial solution, so advance to next one
            sindex[i] = j;
            i++;
            j = 0;
        }
        else
        { // We went through all candidates for this partial soln, so backtrack
            i--;
            if(i == 0)    // We tried them all - no solution
            {
                cout << "No solution" << endl;
                exit(0);
            }
            j = sindex[i] + 1;
        }
    } //END while

    // Report the solution

    cout << the solution is ("
    for (int k = 0; k < n; k++) cout << sindex[k];
    cout << )" << endl;

} //END main
```

The Eight Queens Implementation

The eight queens problem can be solved using the above framework. There are eight partial solutions (corresponding to the eight columns on the chess board), and there are eight candidates for each partial solution (i.e., the rows). Therefore, in theory `space` is an 8×8 array, and `nspace` contains the values $(8, 8, 8, 8, 8, 8, 8, 8)$. Actually, we don't really need a space array here, since each partial solution can be represented by keeping track of the row number of each queen placement.

The `is_soln` function needs to determine if any of the queens placed so far are in conflict with each other. We must check that no queens are in the same row, nor along the same diagonals, as any other queen. The latter conditions can easily be checked by noting that the sum of the coordinates for two queens are equal if they are on the same upper left to lower right diagonal, and the difference of the coordinates are equal if they are on the same lower left to upper right diagonal. If the queens are not in conflict with each other, then `is_soln` should return `true`. The writing of this function is left to the reader.