Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution. Wikipedia

Many of the problems we solve using backtracking require that all the solutions satisfy a complex set of constraints. For any problem these constraints can be divided into two categories: explicit and implicit.

Explicit constraints: Explicit constraints are rules that restrict each Xi to take

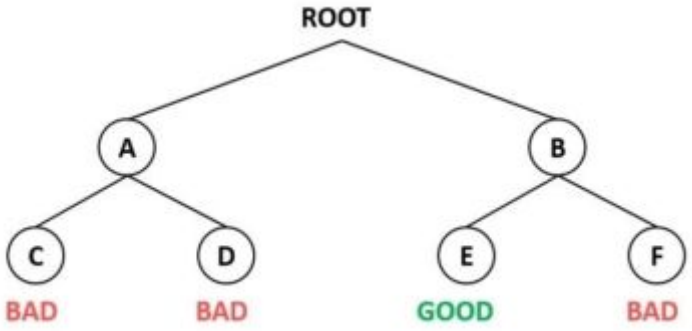
on values only from a given set

Implicit constraints: The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the Xi must relate to each other.

Backtracking algorithm is applied to some specific types of problems,

- Decision problem used to find a feasible solution of the problem.
- Optimisation problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

This approach is used to solve problems that have multiple solutions.



Starting at Root, your options are A and B. You choose A.

1. At A, your options are C and D. You choose C.

2. C is bad. Go back to A.
3. At A, you have already tried C, and it failed. Try D.
4. D is bad. Go back to A.
5. At A, you have no options left to try. Go back to Root.
6. At Root, you have already tried A. Try B.
7. At B, your options are E and F. Try E.
8. E is good. Congratulations!

The backtracking algorithm.

Here is the algorithm (in pseudocode) for doing backtracking from a given node n:

```
boolean solve(Node n) {
    if n is a leaf node {
        if the leaf is a goal node, return true
        else return false
    } else {
        for each child c of n {
            if solve(c) succeeds, return true
        }
        return false
    }
}
```

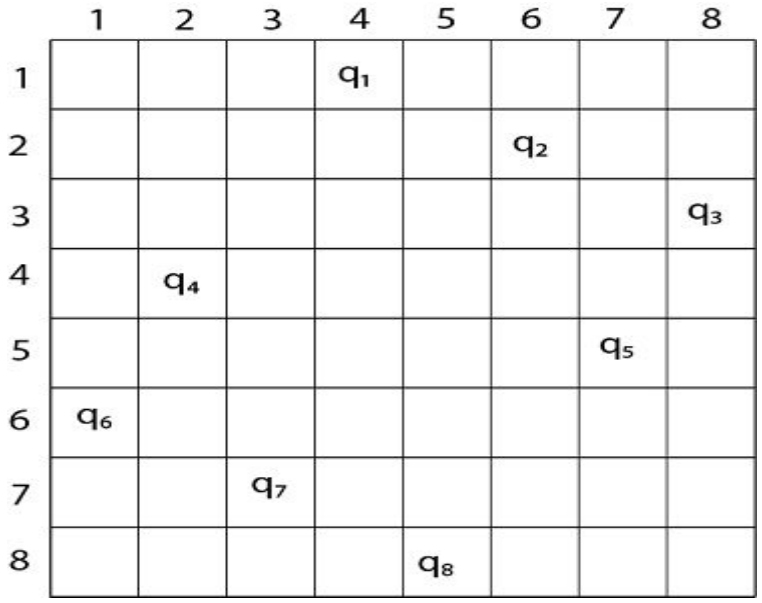
Notice that the algorithm is expressed as a *boolean* function. This is essential to understanding the algorithm. If solve(n) is true, that means node n is part of a solution--that is, node n is one of the nodes on a path from the root to some goal node. We say that n is *solvable*. If solve(n) is false, then there is *no* path that includes n to any goal node.

State Space Tree

A space state tree is a tree representing all the possible states (solution or nonsolution) of the problem from the root as an initial state to the leaf as a terminal state.

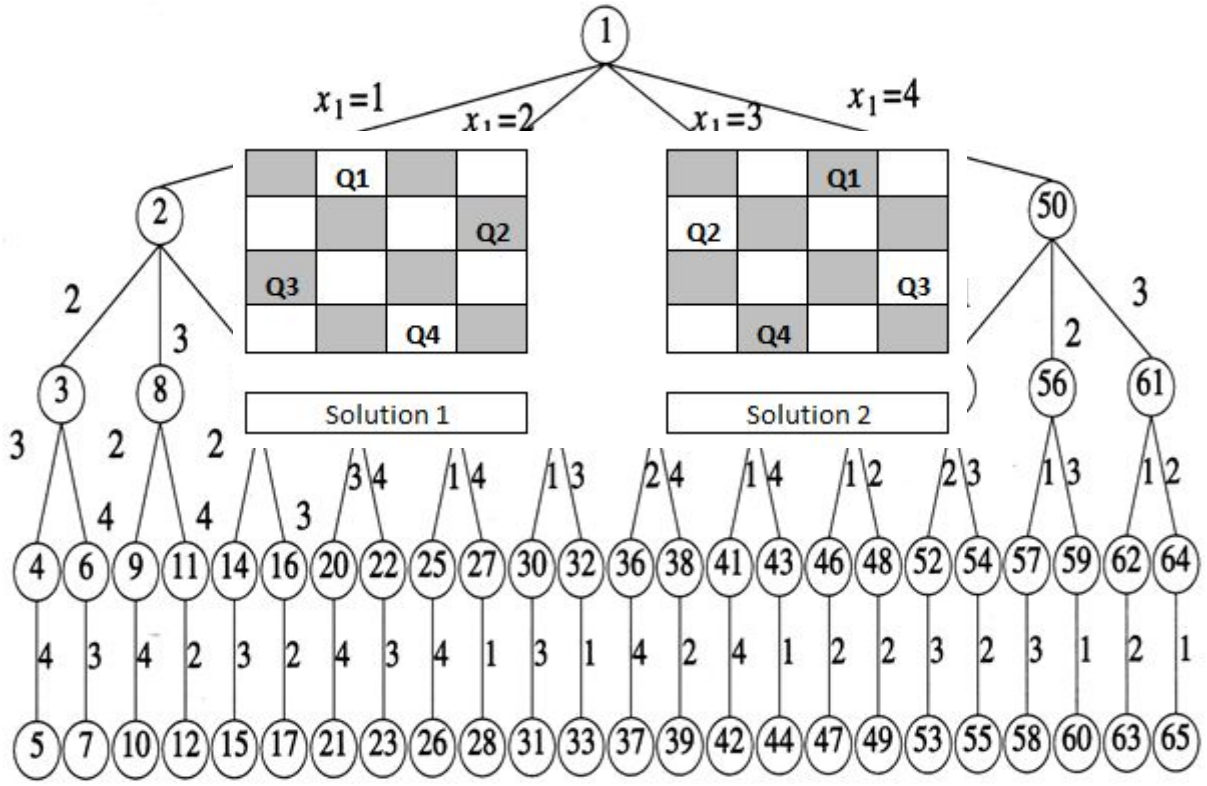
8 Queen Problem

A classic combinatorial problem is to place eight queens on an 8 x 8 chess board so that no two "attack" that is, so that no two of them are on the same row, column or diagonal. Let us number the rows and columns of the chess board1through 8 (Figure).The queen can also be numbered1through8.Since each queen must be on a different row, we can without loss of generality assume queen i is to be placed on row i. All solutions to the 8-queens problem can therefore be represented as 8-tuples(x\\,..,.xs),where Xi is the column on which queen i is placed. The explicit constraints using this formulation are Si = {1,2,3,4,5,6,7, 8}, 1≤ i ≤ 8. Therefore the solution space consists of 8^8 tuples. The implicit constraints for this problem are that no two x_i 's can be the same(i.e., all queens must be on different columns)and no two queens can be on the same diagonal. The first of these two constraints implies that all solutions are permutations of the 8-tuple(1,2,3, 4, 5, 6,7, 8). This realization reduces the size of the solution space from 8^8 tuples to 8! tuples. We see later how to formulate the second constraint in terms of the xi. Expressed as an 8-tuple, the solution in following Figure is (4, 6,8,2,7, 1,3,5).



In the following Figure shows a possible tree organization for the case n = 4. A tree such as this is called a permutation tree. The edges are labelled by possible values of Xi. Edges from level 1to level 2 nodes specify the values for x1. Thus, the leftmost subtree contains all solutions with x1= 1; its leftmost subtree contains all solutions with x1= 1 and x2= 2,and so on. Edges from level i to level i +1 are labelled with the values of xi. The solution space is defined by all paths from the

root node to a leaf node. There are 4!= 24 leaf nodes in the tree of the following Figure



Algorithm

```
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    /* Check this row on left side */
    for (i = 0; i < col; i++)
        if (board[row][i]==1)
            return false;
```

```
    /* Check upper diagonal on left side */
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j]==1)
            return false;

    /* Check lower diagonal on left side */
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j]==1)
            return false;

    return true;
}

/* A recursive utility function to solve N
   Queen problem */
bool solveNQueen(int board[N][N], int col)
{
    /* base case: If all queens are placed
      then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
       this queen in all rows one by one */
    for (int i = 0; i < N; i++) {
        /* Check if the queen can be placed on
          board[i][col] */
        if (isSafe(board, i, col)) {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;
```

```
            /* recur to place rest of the queens */
            if (solveNQueen(board, col + 1))
                return true;

            /* If placing queen in board[i][col]
               doesn't lead to a solution, then
               remove queen from board[i][col] */
            board[i][col] = 0; // BACKTRACK
        }
    }

    /* If the queen cannot be placed in any row in
        this colum col  then return false */
    return false;
}
```

Time Complexity

From the reference of above state space tree Time complexity of n queen problem is n!.

Or use this recurrence relation and find T.C T(n) = n*T(n-1) + N*n