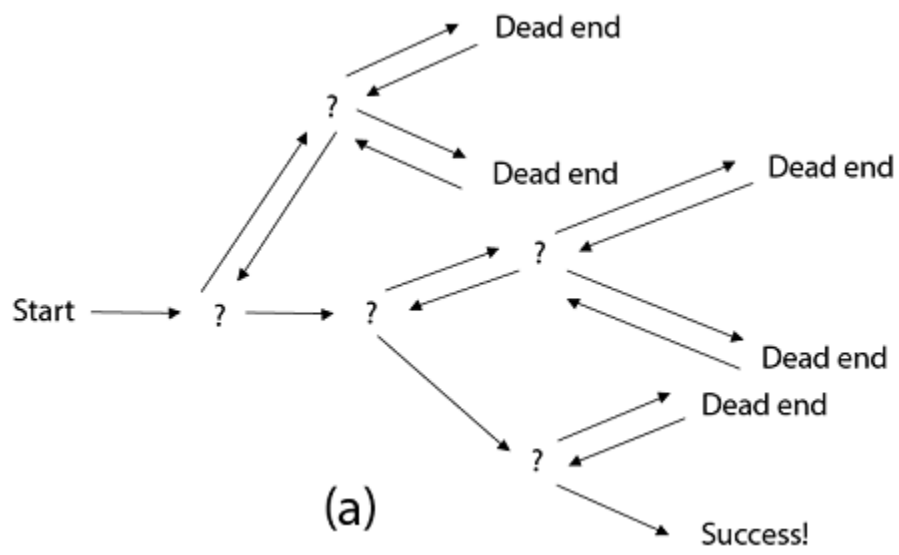
Introduction of Backtracking

The Backtracking is an algorithmic-method to solve a problem with an additional way. It uses a recursive approach to explain the problems. We can say that the backtracking is needed to find all possible combination to solve an optimization problem.

Backtracking is a systematic way of trying out different sequences of decisions until we find one that "works."

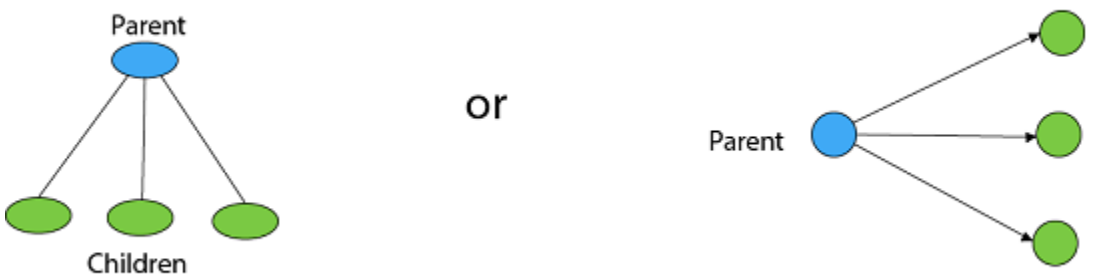
In the following Figure:

- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



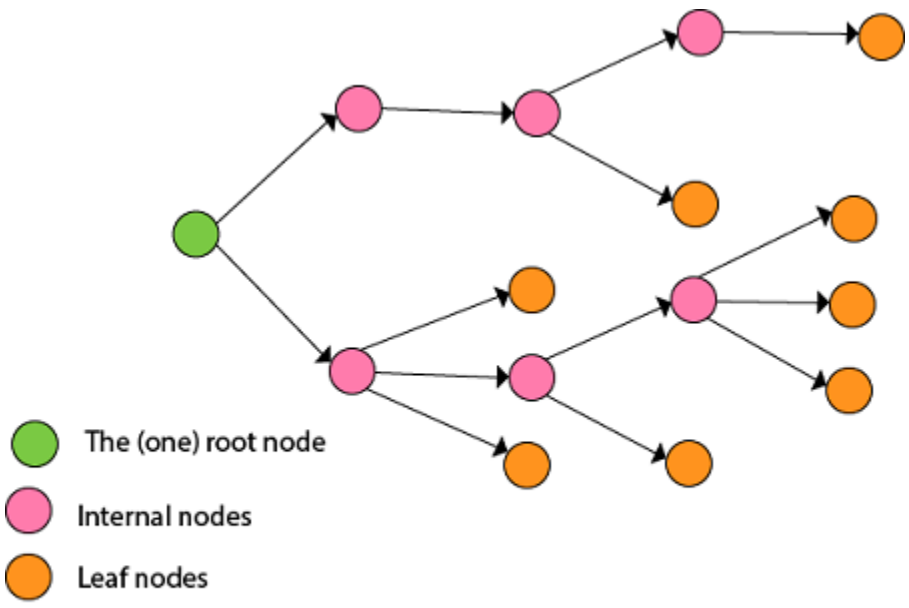
(a)

Generally, however, we draw our trees downward, with the root at the top.



(b)

A tree is composed of nodes.



(c)

Backtracking can understand of as searching a tree for a particular "goal" leaf node.

Backtracking is undoubtedly quite simple - we "explore" each node, as follows:

To "explore" node N:

```
1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
```

```
3. For each child C of N,
    Explore C
    If C was successful, return "success"
4. Return "failure"
```

Backtracking algorithm determines the solution by systematically searching the solution space for the given problem. Backtracking is a **depth-first search** with any bounding function. All solution using backtracking is needed to satisfy a complex set of constraints. The constraints may be explicit or implicit.

Explicit Constraint is ruled, which restrict each vector element to be chosen from the given set.

Implicit Constraint is ruled, which determine which each of the tuples in the solution space, actually satisfy the criterion function.

Recursive Maze Algorithm

Recursive Maze Algorithm is one of the best examples for backtracking algorithms. Recursive Maze Algorithm is one of the possible solutions for solving the maze.

Maze

The maze is an area surrounded by walls; in between, we have a path from starting point to ending position. We have to start from the starting point and travel towards from ending point.

Start

End

Principle of Maze

As explained above, in the maze we have to travel from starting point to ending point. The problem is to choose the path. If we find any dead-end before ending point, we have to backtrack and move the direction. The direction for traversing is North, East, West, and South. We have to continue "move and backtrack" until we reach the final point.

Consider that we are having a two-dimensional maze cell [WIDTH] [HEIGHT]. Here cell [x] [y] = 1 denotes wall and cell [x] [y] = 0 denotes free cell in the particular location x, y in the maze. The directions we can change in the array are North, East, West, and South. The first step is to make the boundary of the two - dimensional array as one so that we won't go out of the maze and usually reside inside the maze at any time.

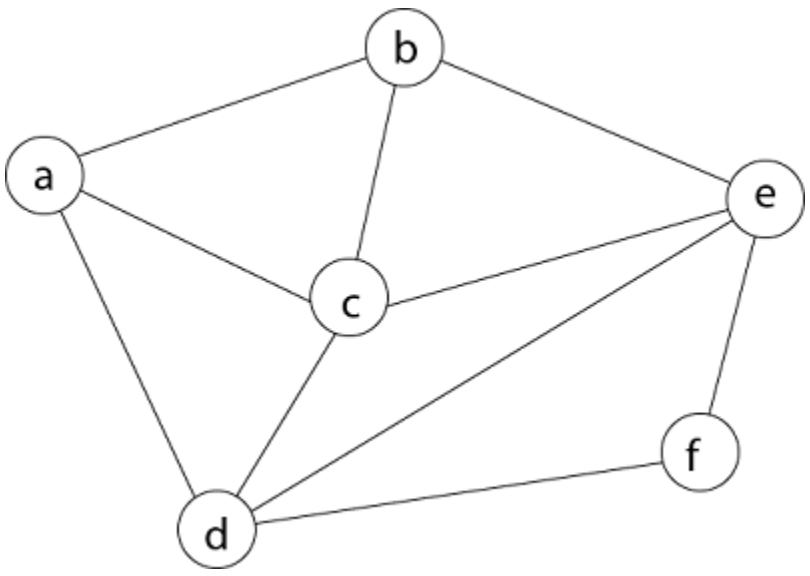
Example Maze						
1	1	1	1	1	1	1
1	0	0	0	1	1	1
1	1	1	0	1	1	1
1	1	1	0	0	0	1
1	1	1	1	1	0	1
1	1	1	1	1	1	1

Now start changing from the starting position (since the boundary is filled by 1) and find the next free cell then turn to the next free cell and so on. If we grasp a dead-end, we have to backtrack and make the cells in the path as 1 (wall). Continue the same process till the final point is reached.

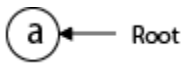
Hamiltonian Circuit Problems

Given a graph G = (V, E) we have to find the Hamiltonian Circuit using Backtracking approach. We start our search from any arbitrary vertex say 'a.' This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed. The next adjacent vertex is selected by alphabetical order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that **dead end** is reached. In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed. The search using backtracking is successful if a Hamiltonian Cycle is obtained.

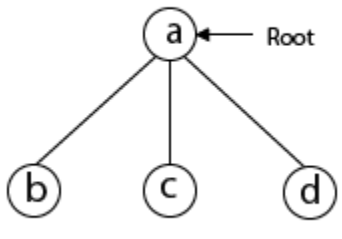
Example: Consider a graph G = (V, E) shown in fig. we have to find a Hamiltonian circuit using Backtracking method.



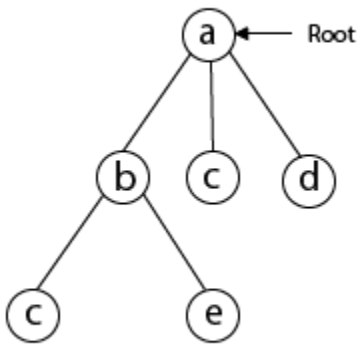
Solution: Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.



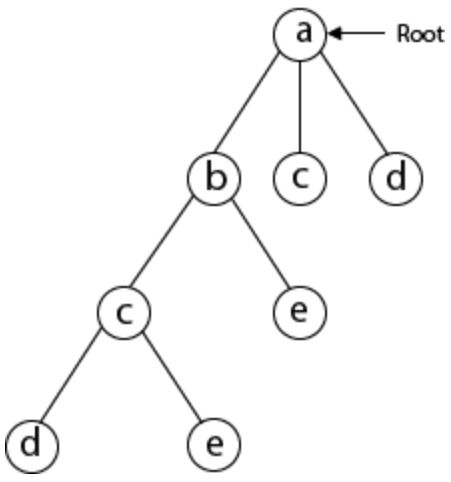
Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



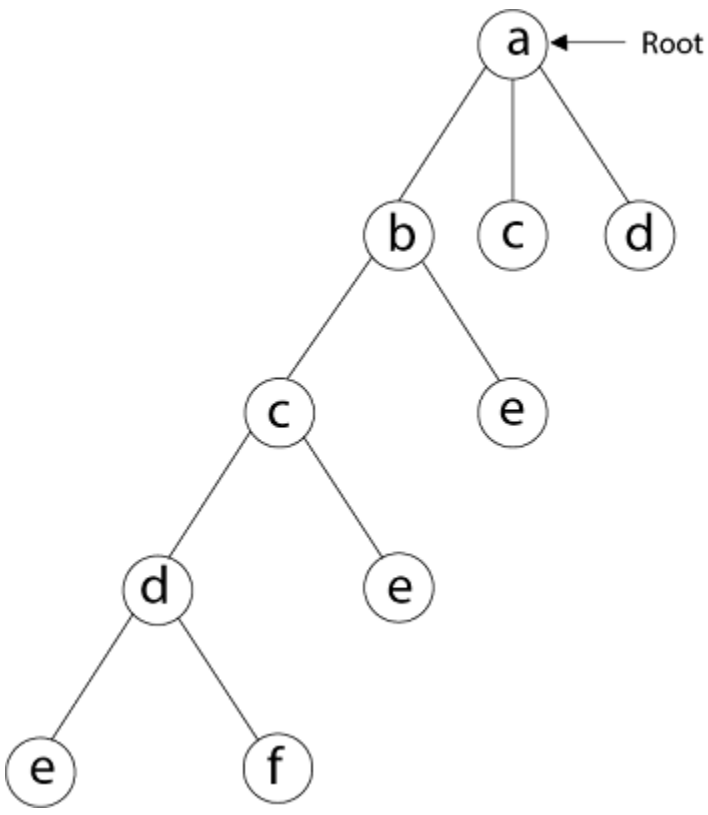
Next, we select 'c' adjacent to 'b.'



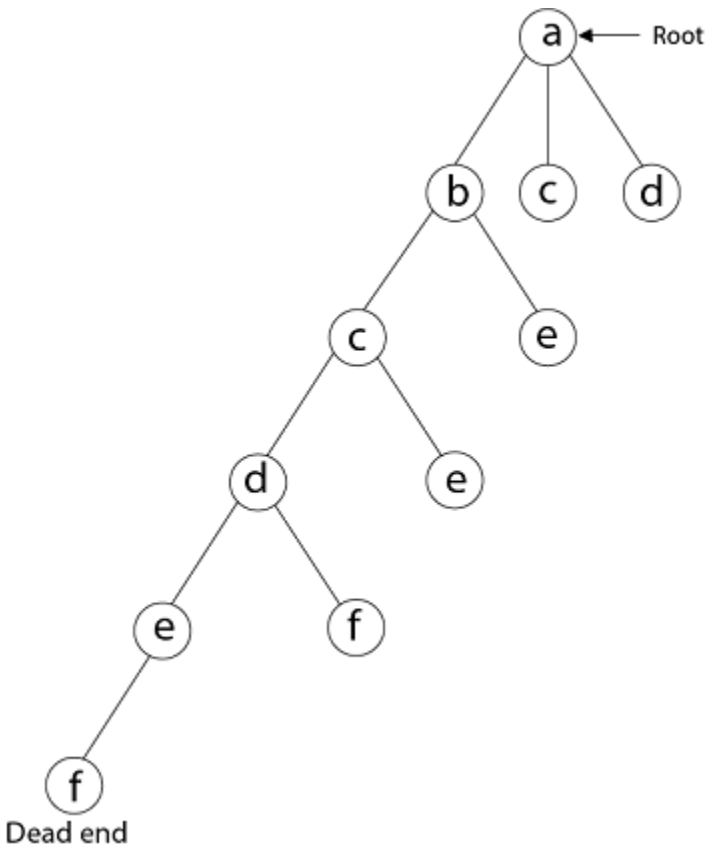
Next, we select 'd' adjacent to 'c.'



Next, we select 'e' adjacent to 'd.'



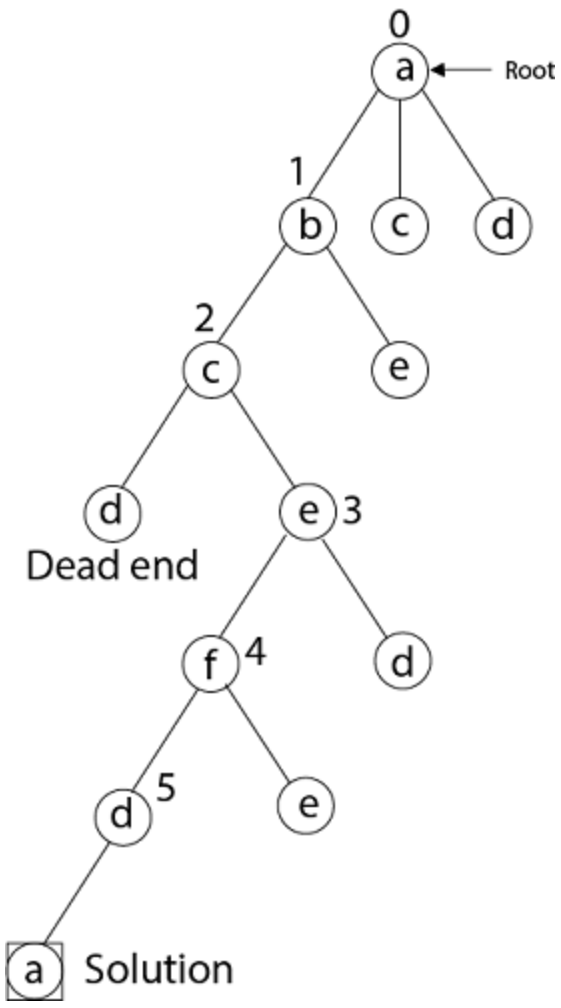
Next, we select vertex 'f' adjacent to 'e.' The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.



From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.

Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once. (a - b - c - e - f -d - a).

Again Backtrack



Here we have generated one Hamiltonian circuit, but another Hamiltonian circuit can also be obtained by considering another vertex.

Subset-Sum Problem

The Subset-Sum Problem is to find a subset's' of the given set S = (S_1 S_2 S_3...S_n) where the elements of the set S are n positive integers in such a manner that s'∈S and sum of the elements of subset's' is equal to some positive integer 'X.'

The Subset-Sum Problem can be solved by using the backtracking approach. In this implicit tree is a binary tree. The root of the tree is selected in such a way that represents that no decision is yet taken on any input. We assume that the elements of the given set are arranged in increasing order:

$$S_1 \leq S_2 \leq S_3 \ldots \leq S_n$$

The left child of the root node indicated that we have to include 'S_1' from the set 'S' and the right child of the root indicates that we have to execute 'S_1'. Each node stores the total of the partial solution elements. If at any stage the sum equals to 'X' then the search is successful and terminates.

The dead end in the tree appears only when either of the two inequalities exists:

- The sum of s' is too large i.e.

$$s' + S_i + 1 > X$$

- The sum of s' is too small i.e.

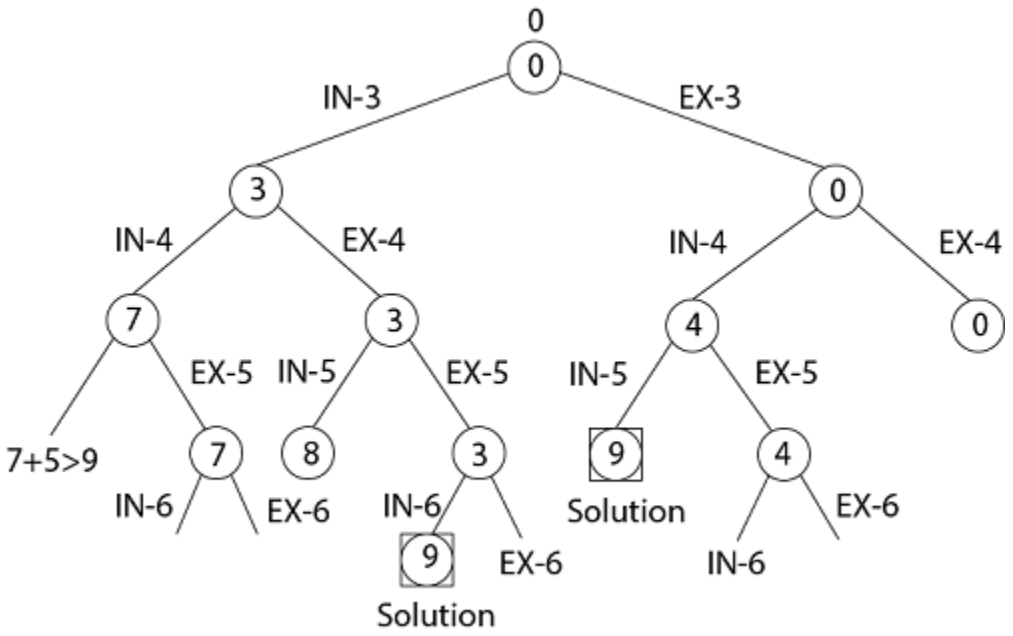
$$s' + \sum_{j=i+1}^{n} S_j < X$$

Example: Given a set S = (3, 4, 5, 6) and X =9. Obtain the subset sum using Backtracking approach.

Solution:

1. Initially S = (3, 4, 5, 6) and X =9.
2. S'= (Ø)

The implicit binary tree for the subset sum problem is shown as fig:



The number inside a node is the sum of the partial solution elements at a particular level.

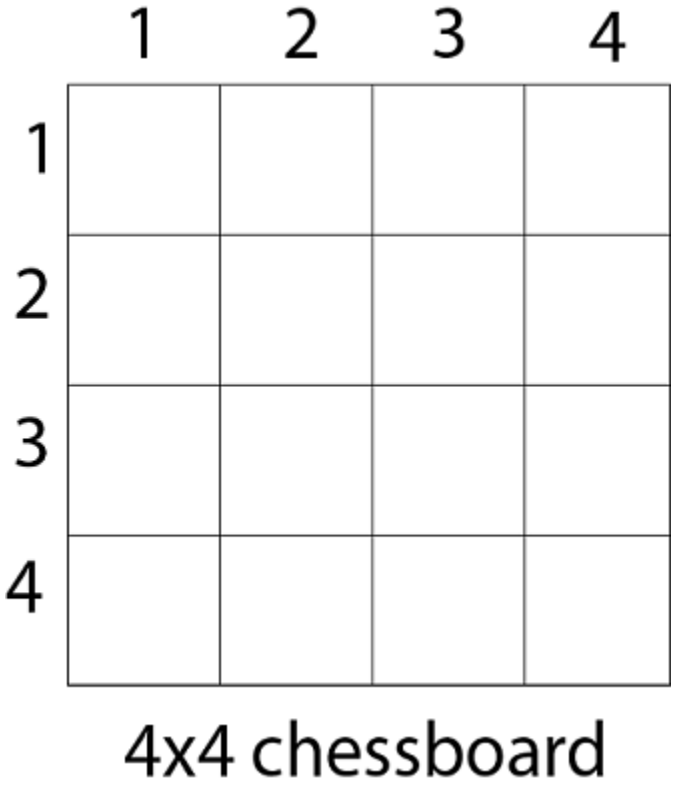
Thus, if our partial solution elements sum is equal to the positive integer 'X' then at that time search will terminate, or it continues if all the possible solution needs to be obtained.

N-Queens Problem

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

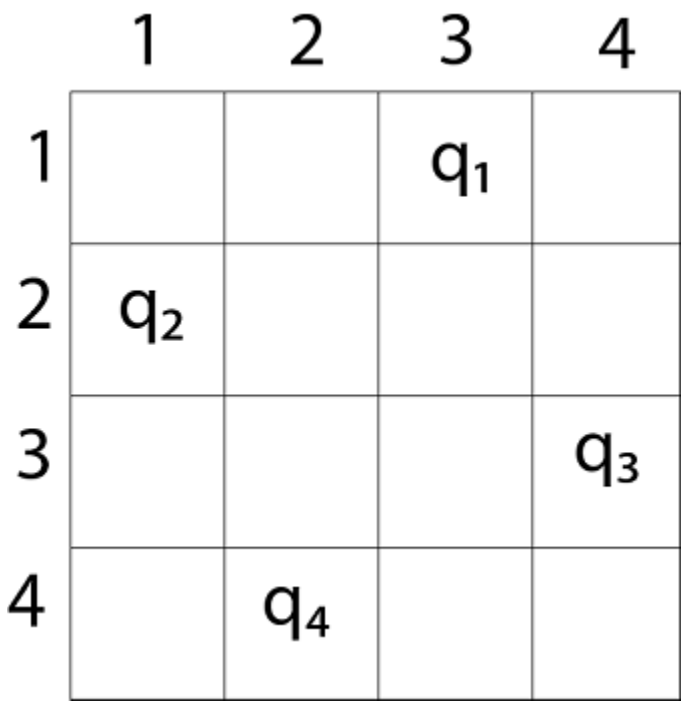
It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.



Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.



The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

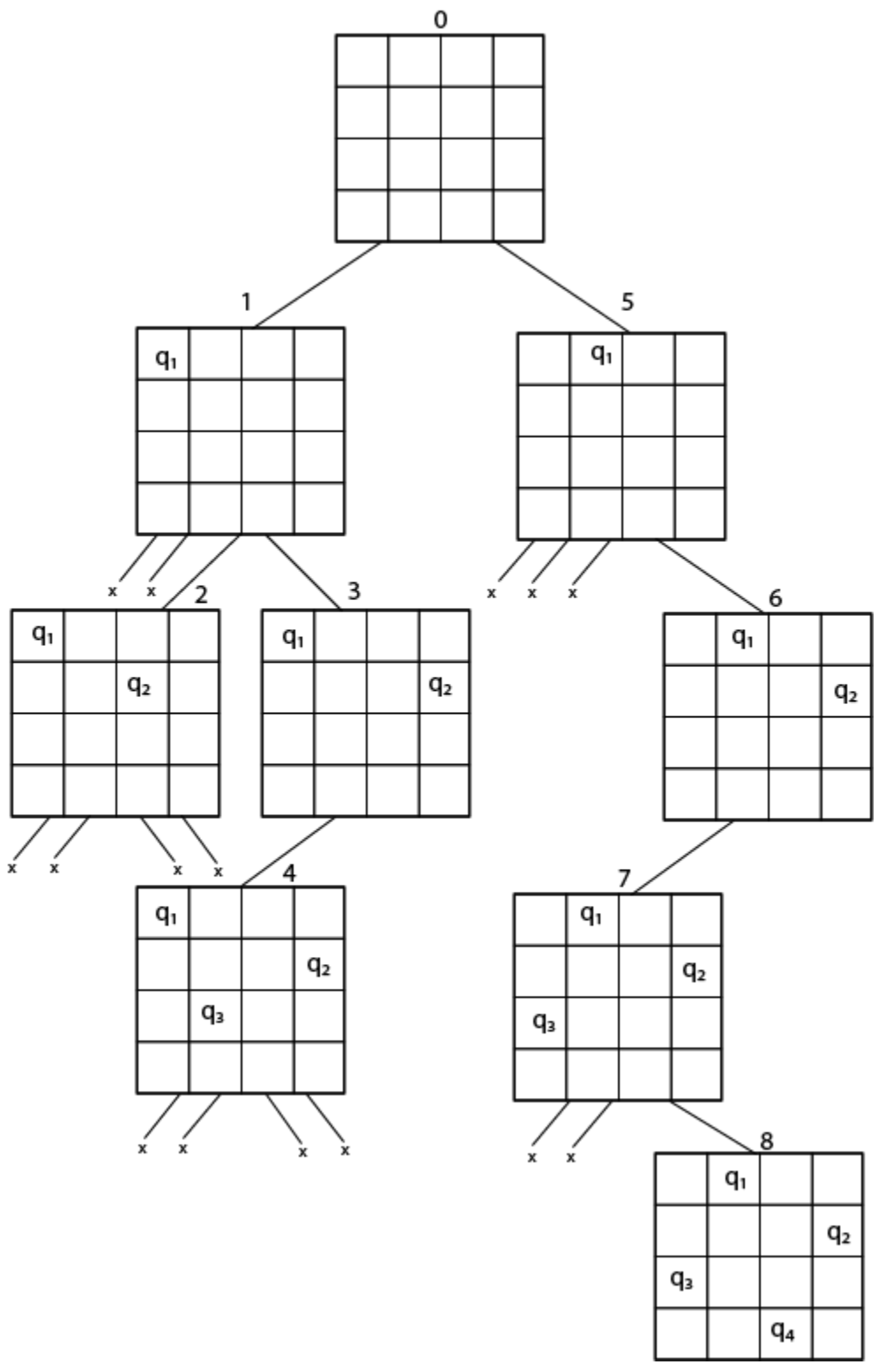
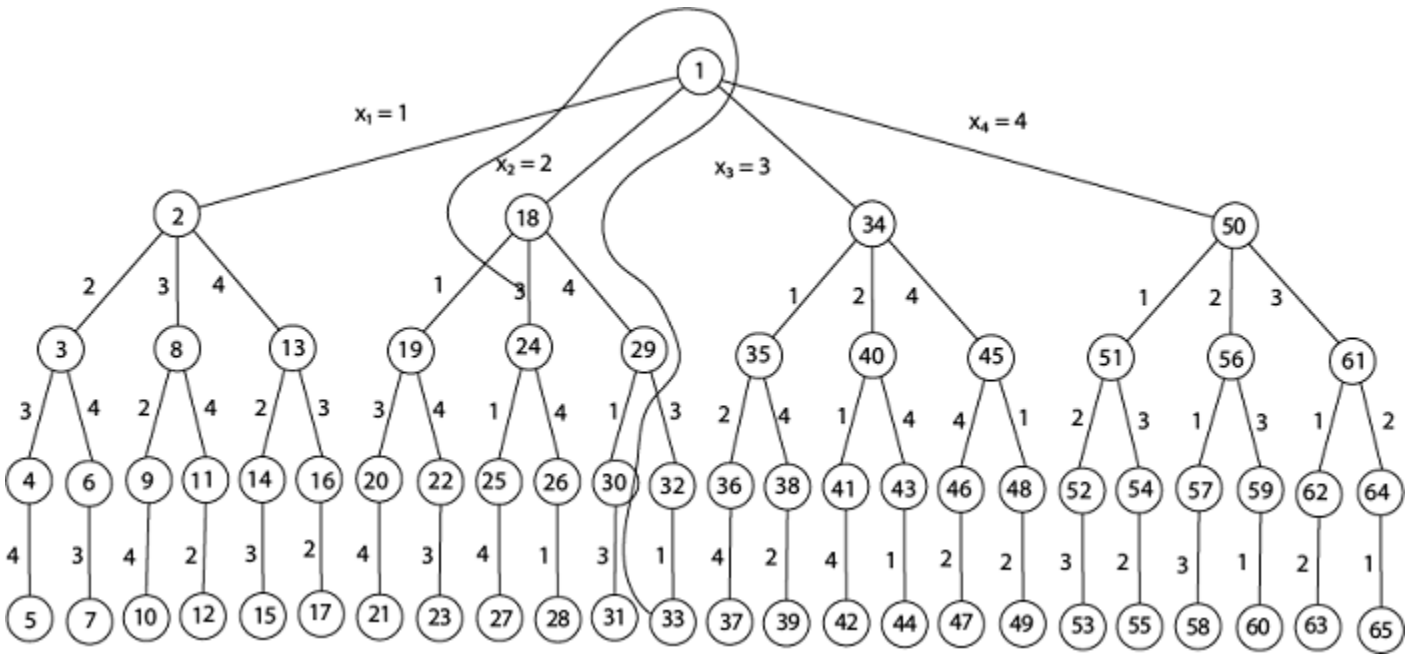


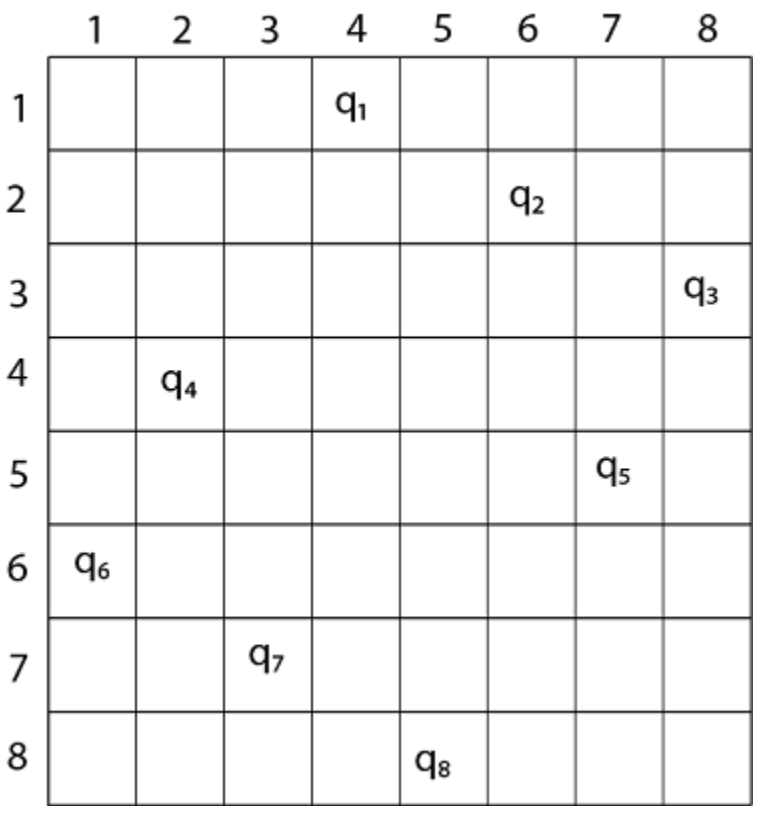
Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



4 - Queens solution space with nodes numbered in DFS

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in fig:



1. Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.
4. The first equation implies that j - l = i - k.
5. The second equation implies that j - l = k - i.
6. Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to then n- queens problem.

```
1. Place (k, i)
2.   {
3.     For j ← 1 to k - 1
4.      do if (x [j] = i)
5.       or (Abs x [j]) - i) = (Abs (j - k))
6.     then return false;
7.      return true;
```

```
8.  }
```

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

```
1.  N - Queens (k, n)
2.  {
3.     For i  ←  1 to n
4.          do if Place (k, i) then
5.     {
6.        x [k]  ←  i;
7.        if (k ==n) then
8.          write (x [1....n));
9.        else
10.       N - Queens (k + 1, n);
11.    }
12. }
```