CHAPTER SIX

Backtracking Algorithms

Introduction

- **Backtracking is a problem-solving algorithmic technique that uses a brute force approach to find the desired solution.**
  - **The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.**
- **The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions.**
  - **Thus, recursion is used in this approach.**
- **This approach is used to solve problems that have multiple solutions.**

When do we use backtracking algorithm?

- **Backtracking algorithm can be used for the following problems:**
  - **If the problem has multiple solutions or requires finding all possible solutions.**
  - **When the given problem can be broken down into smaller subproblems that are similar to the original problem.**
  - **If the problem has some constraints or rules that must be satisfied by the solution**

How Does a Backtracking Algorithm Work?

- The following is a general outline of how a backtracking algorithm works:

  1. Choose an initial solution.
  2. Explore all possible extensions of the current solution.
  3. If an extension leads to a solution, return that solution.
  4. If an extension does not lead to a solution, backtrack to the previous solution and try a different extension.
  5. Repeat steps 2-4 until all possible solutions have been explored.

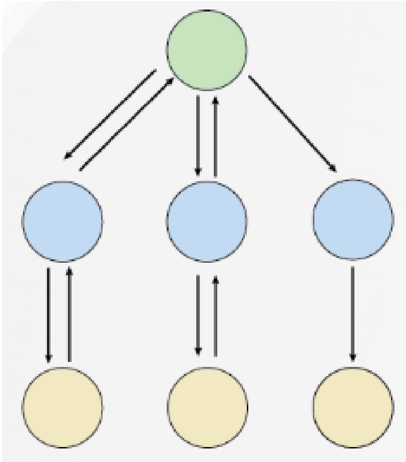
Backtracking Algorithm

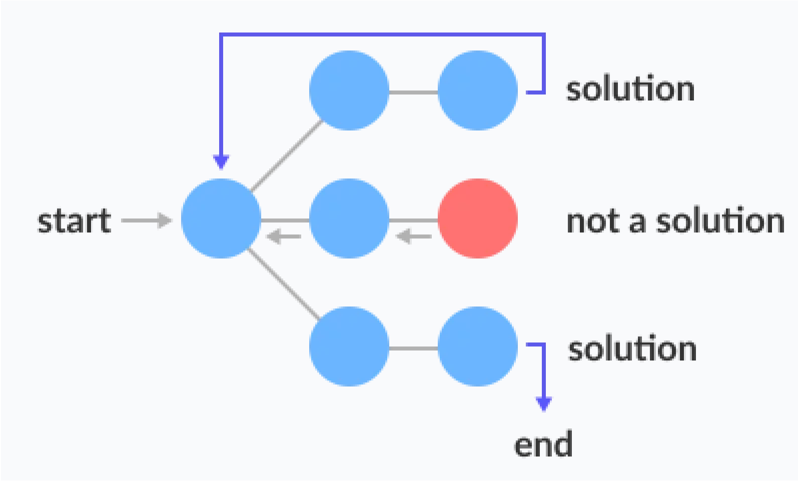
```
Backtrack(s)
    if s is not a solution
        return false
    if s is a new solution
        add to list of solutions
    backtrack(expand s)
```

- Backtracking algorithm **finds a solution** by building a solution **step by step**, increasing levels over time, using **recursive** calling.
- A search tree known as the **state-space tree** is used to find these solutions.

State-space tree

- **A space state tree is a tree representing all the possible states (solution or non-solution) of the problem from the root as an initial state to the leaf as a terminal state.**
- **Each branch in a state-space tree represents a variable, and each level represents a solution.**





Backtracking vs Recursion

- **Recursion** is a technique that calls the same function again and again until you reach the base case.
- **Backtracking** is an algorithm that finds all the possible solutions and selects the desired solution from the given set of solutions.
  - When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

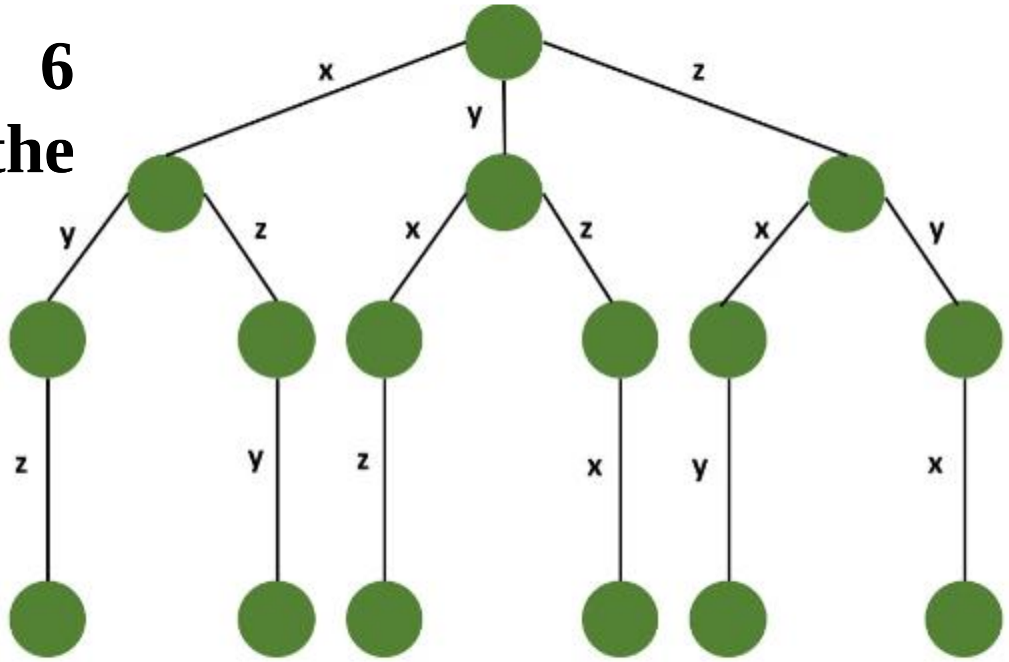
Example

- **You need to arrange the three letters x, y, and z so that z cannot be next to x.**
  - **According to the backtracking, you will first construct a state-space tree.**
  - **Look for all possible solutions and compare them to the given constraint.**
  - **You must only keep solutions that meet the constraint.**

Example

- The following are 3! = 6 possible solutions to the problems:

  (x,y,z), (x,z,y),
  (y,x,z), (y,z,x),
  (z,x,y) (z,y,x).



- However, valid solutions to this problem are those that satisfy the constraint that keeps only (x,y,z) and (z,y,x) in the final solution set.

Applications of Backtracking Algorithm

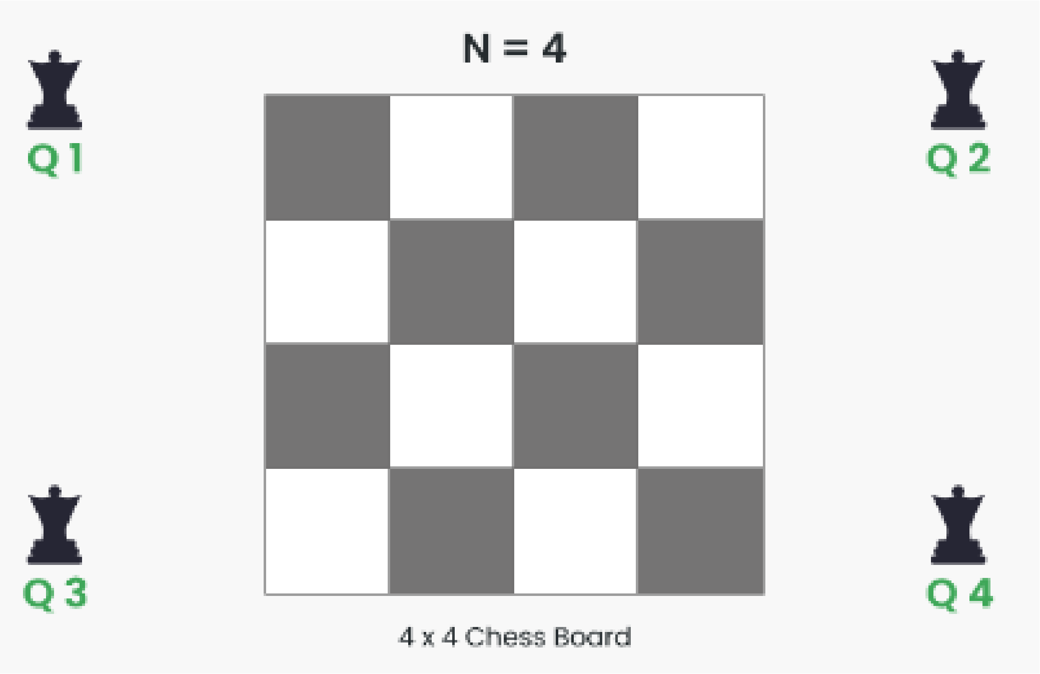
- The backtracking algorithm has the following applications (some of):
  - To Find All Hamiltonian Paths Present in a Graph
  - To Solve the N Queen Problem
  - Maze Solving Problems
  - The Knight's Tour Problem

N-Queens Problem

- **The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.**
- **A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way.**
- **The most popular approach for solving the N Queen puzzle is Backtracking.**
- **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.**

4 Queens Problem

- **The 4 Queens Problem consists in placing four queens on a 4 x 4 chessboard so that no two queens attack each other.**
  - **i.e., no two queens are allowed to be placed on the same row, the same column or the same diagonal.**

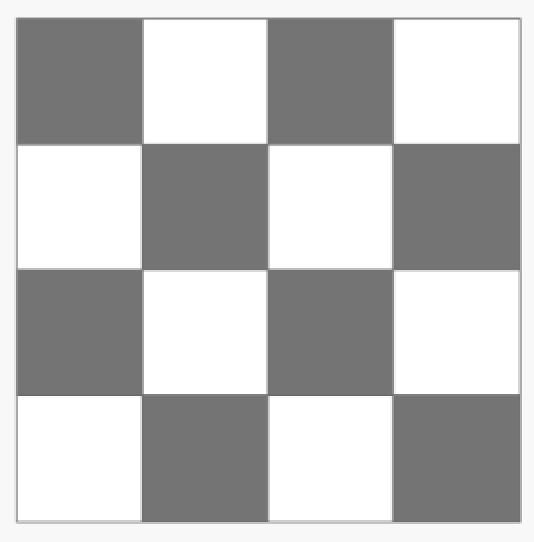


4 Queens Problem: Solution

- Place each queen one by one in different rows, starting from the topmost row.
- While placing a queen in a row, check for clashes with already placed queens.
- For any column, if there is no clash then mark this row and column as part of the solution by placing the queen.
- In case, if no safe cell found due to clashes, then backtrack (i.e, undo the placement of recent queen) and return false.

4 Queens Problem: Solution

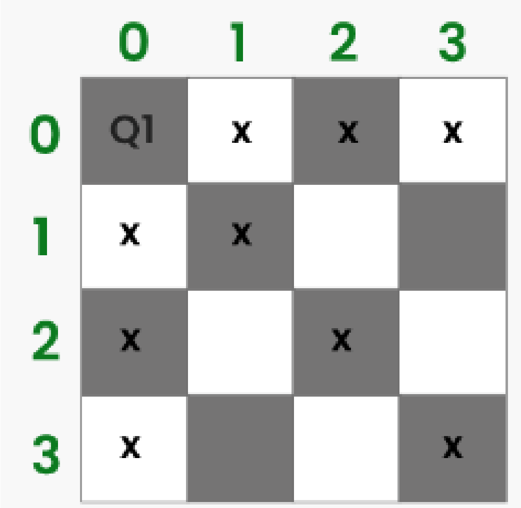
- **Step 0: Initialize a 4×4 board.**



4 x 4 Chess Board

Step 1:

- **Put our first Queen (Q1) in the (0,0) cell .**
- **'x' represents the cells which is not safe i.e. they are under attack by the Queen (Q1).**
- **After this; move to the next row [0 ⟶ 1].**

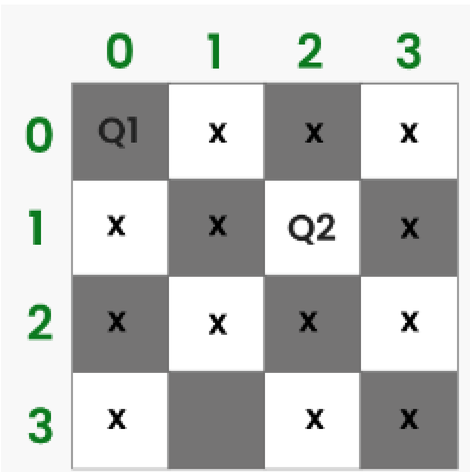


4 x 4 Chess Board

4 Queens Problem: Solution

Step 2:

- Put our next Queen (Q2) in the (1,2) cell .
- After this move to the next row [1 ⟶ 2].



4 x 4 Chess Board

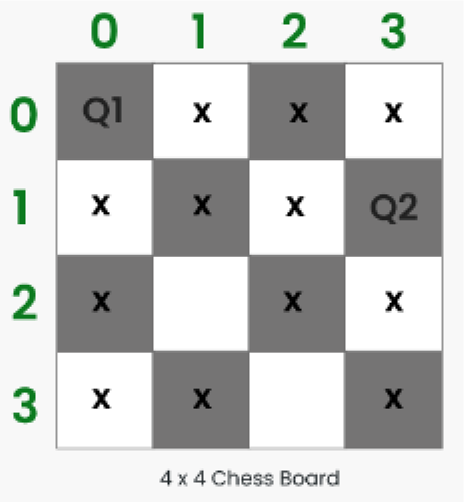
Step 3:

- At row 2 there is no cell which are safe to place Queen (Q3) .
- So, backtrack and remove queen Q2 queen from cell (1, 2) .

4 Queens Problem: Solution

Step 4:

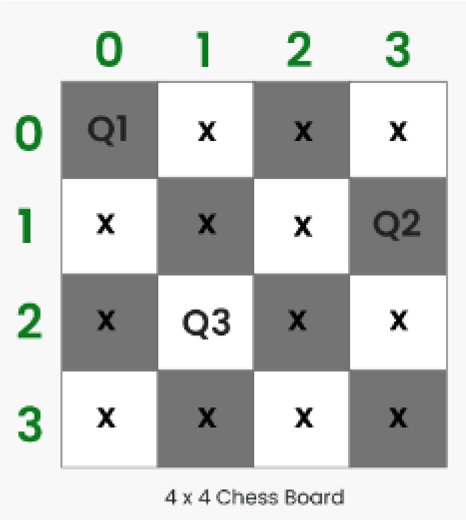
- There is still a safe cell in the row 1 i.e. cell (1, 3).
- Put Queen (Q2) at cell (1, 3).



4 x 4 Chess Board

Step 5:

- Put queen (Q3) at cell (2, 1).



4 x 4 Chess Board

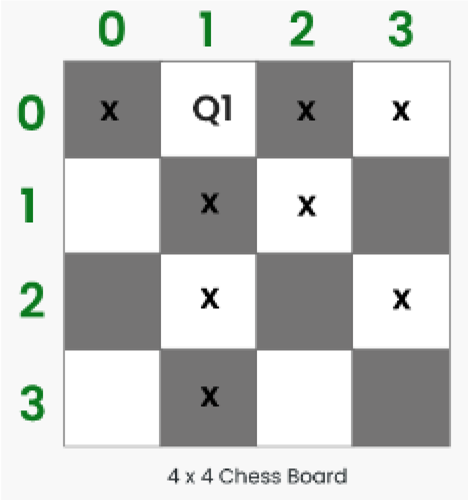
4 Queens Problem: Solution

Step 6:

- There is no any cell to place Queen (Q4) at row 3.
- Backtrack and remove Queen (Q3) from row 2.
- Again, there is no other safe cell in row 2, So backtrack again and remove queen (Q2) from row 1.
- Queen (Q1) will be removed from cell (0,0) and move to next safe cell i.e. (0,1).

Step 7:

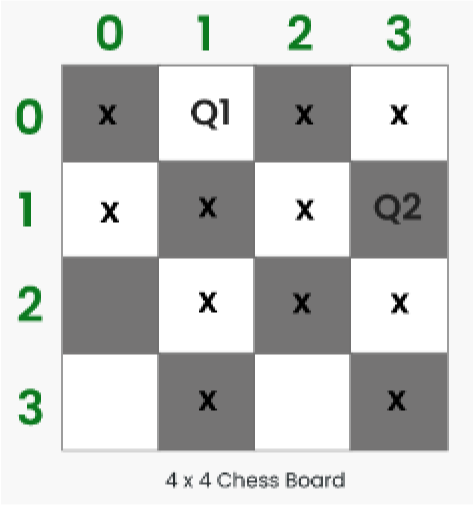
- Place Queen Q1 at cell (0,1), and move to next row.



4 Queens Problem: Solution

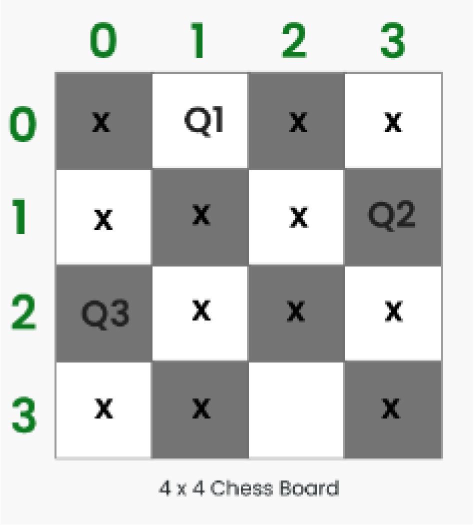
Step 8:

- Place Queen Q2 at cell (1,3), and move to next row.



Step 9:

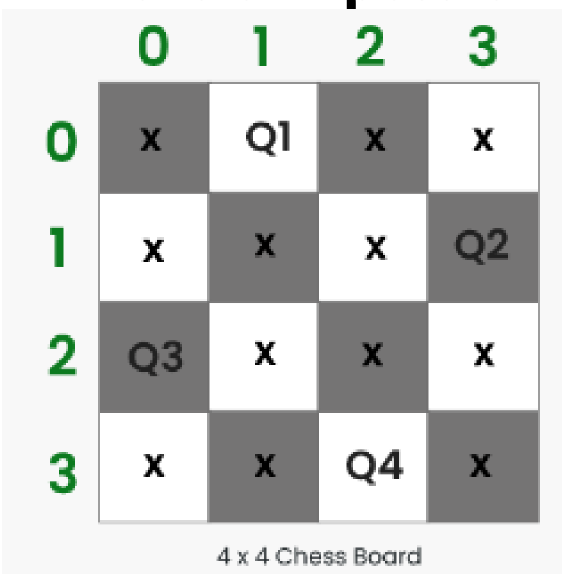
- Place Queen Q3 at cell (2,0), and move to next row.



4 Queens Problem: Solution

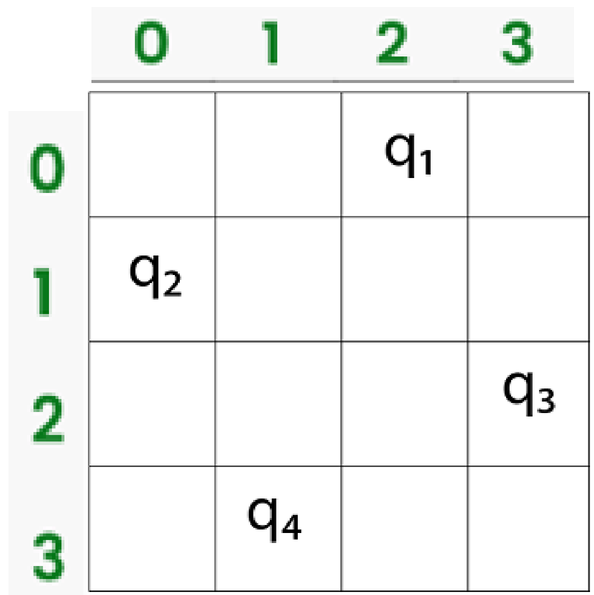
Step 10:

- Place Queen Q4 at cell (3 , 2), and move to next row.
- This is one possible configuration of solution (1,3,0,2).



$$4 \times 4 = \begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{matrix}$$

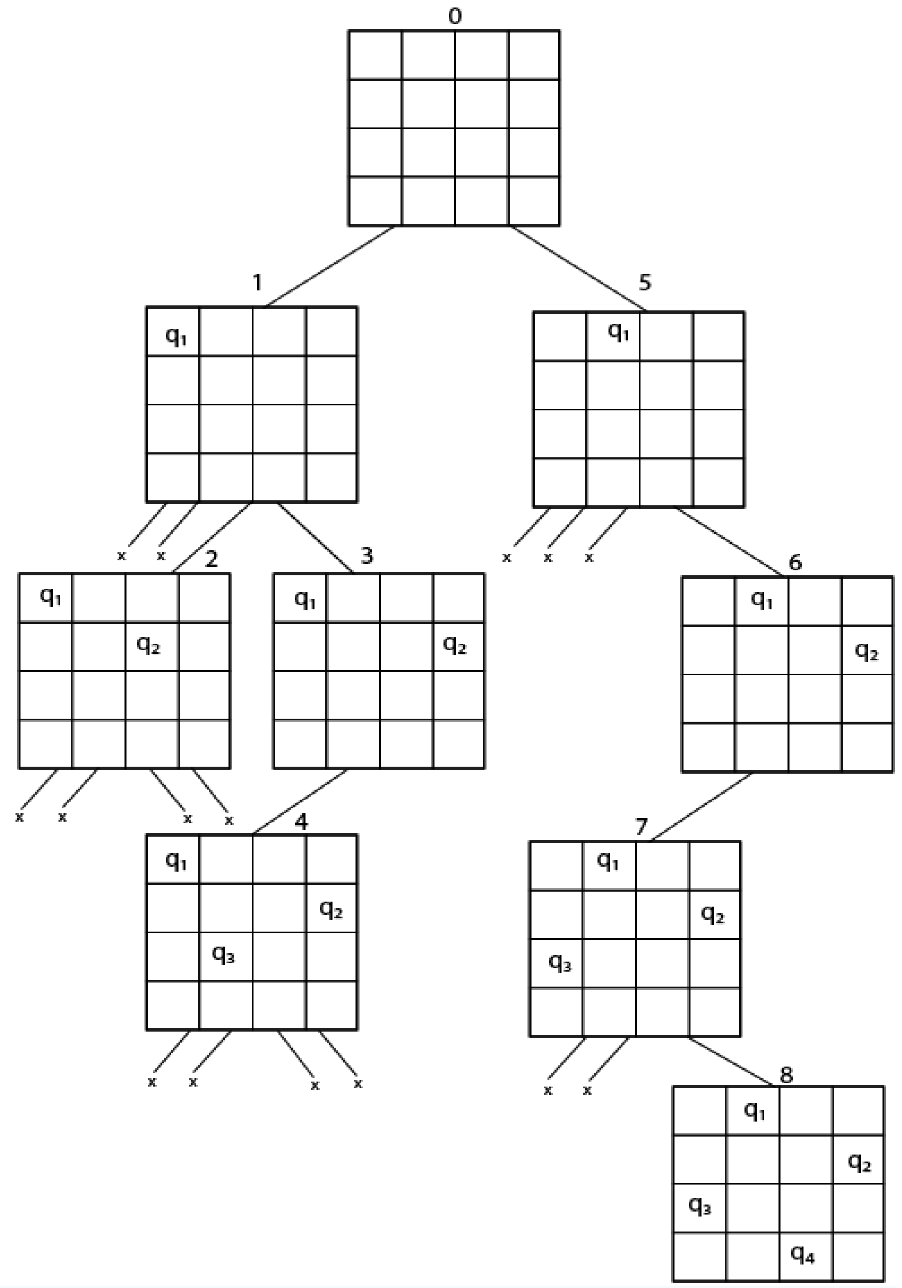
- The other solutions for 4 - queens' problems is (2, 0, 3, 1)



$$4 \times 4 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{matrix}$$

4 Queens Problem:
State - Space Tree

- **The implicit state - space tree for 4 - queen problem for a solution (1,3,0,2) is as follows:**



Time Complexity: O(N!)
Space Complexity: O(N^2)

Follow the steps mentioned below to implement the idea:

- **Make a recursive function that takes the state of the board and the current row number as its parameter.**
- **Start in the topmost row**
- **If all queens are placed return true**
- **For every row, do the following for each column in current row**
  - **If the queen can be placed safely in this column**
    - **Then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.**
    - **If placing the queen in [row, column] leads to a solution then return true.**
    - **If placing queen doesn’t lead to a solution then unmark this [row, column] then backtrack and try other columns.**
  - **If all columns have been tried and valid solution is not found return false to trigger backtracking.**

Hamiltonian Cycle Problem

- **A Hamiltonian Cycle or Circuit in a graph G is a cycle that visits every vertex of G exactly once and returns to the starting vertex, forming a closed loop.**

- **A graph is said to be a Hamiltonian graph only when it contains a Hamiltonian cycle, otherwise, it is called non-Hamiltonian graph.**

- **The Hamiltonian Cycle problem has practical applications in various fields, such as:**
  - **Logistics; delivery system,**
  - **network design, and**
  - **computer science**

Hamiltonian Cycle Problem

- **Hamiltonian Path in a graph G is a path that visits every vertex of G exactly once.**
- **Hamiltonian Path doesn’t have to return to the starting vertex; it is an open path.**
- **Hamiltonian Paths have applications in various fields, such as:**
  - **finding optimal routes in transportation networks,**
  - **circuit design, and**
  - **graph theory research**

How it works?

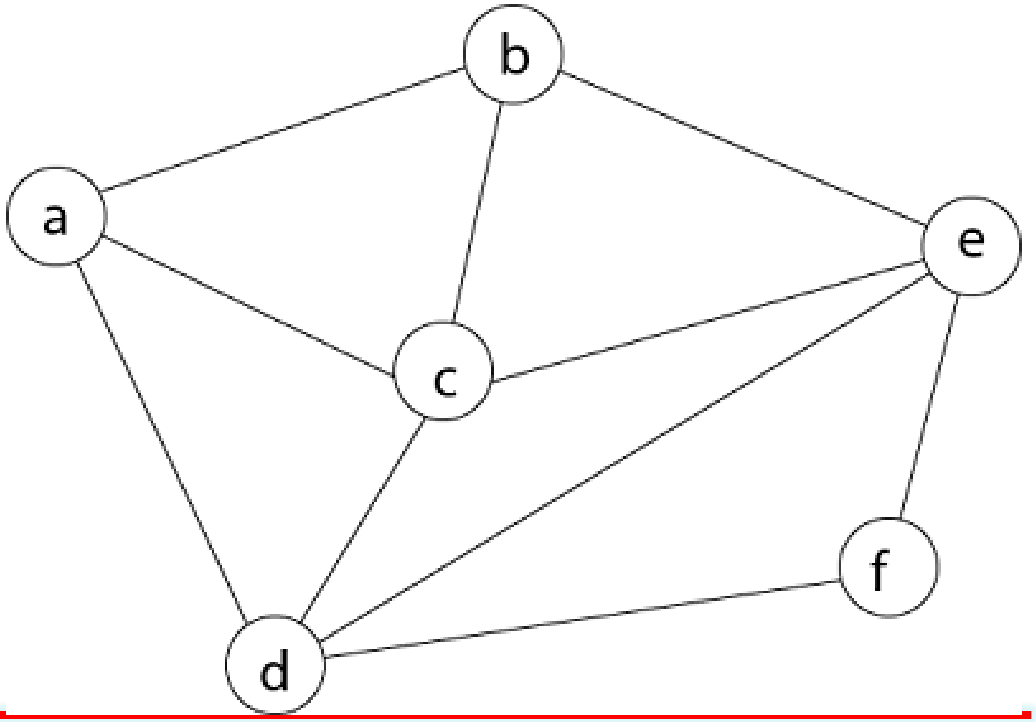
- **For a given graph G = (V, E) we have to find the Hamiltonian Cycle using Backtracking approach.**
  - **We start our search from any arbitrary vertex say ‘s’.**
  - **This vertex ‘s' becomes the root of our implicit tree.**
  - **The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed.**
  - **The next adjacent vertex is selected by alphabetical order.**

How it works?

- **If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex ‘s' 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.**

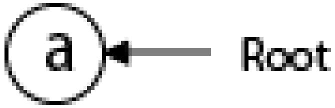
Hamiltonian Cycle Problem: Example

- **For the following given an undirected graph G = (V, E) , determine whether the graph contains a Hamiltonian cycle or not. If it contains, then prints the path. (consider vertex ‘a’ as a root)**

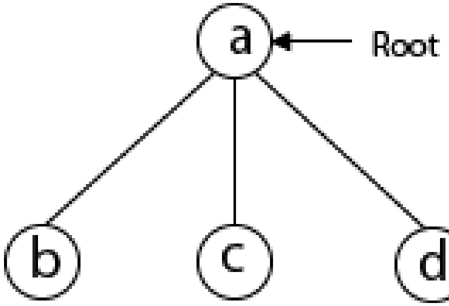


Hamiltonian Cycle Problem: Example

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

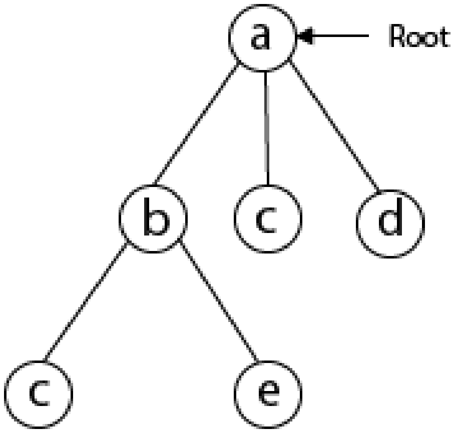


- Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).

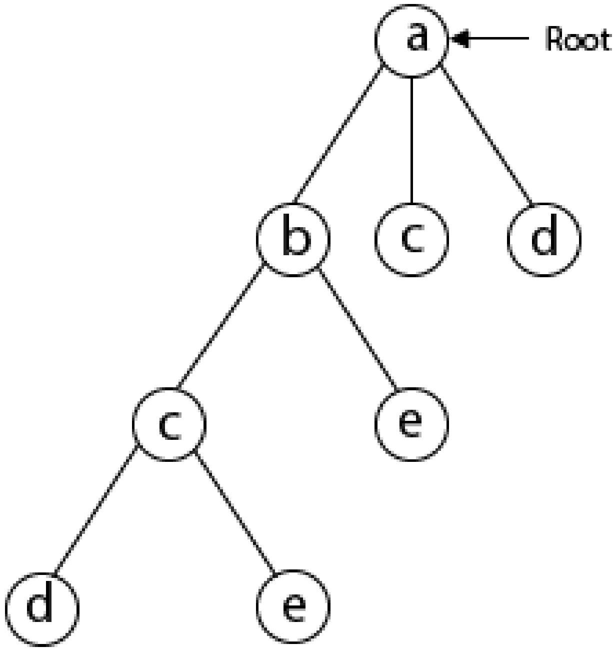


Hamiltonian Cycle Problem: Example

- **Next, we select 'c' adjacent to 'b'**

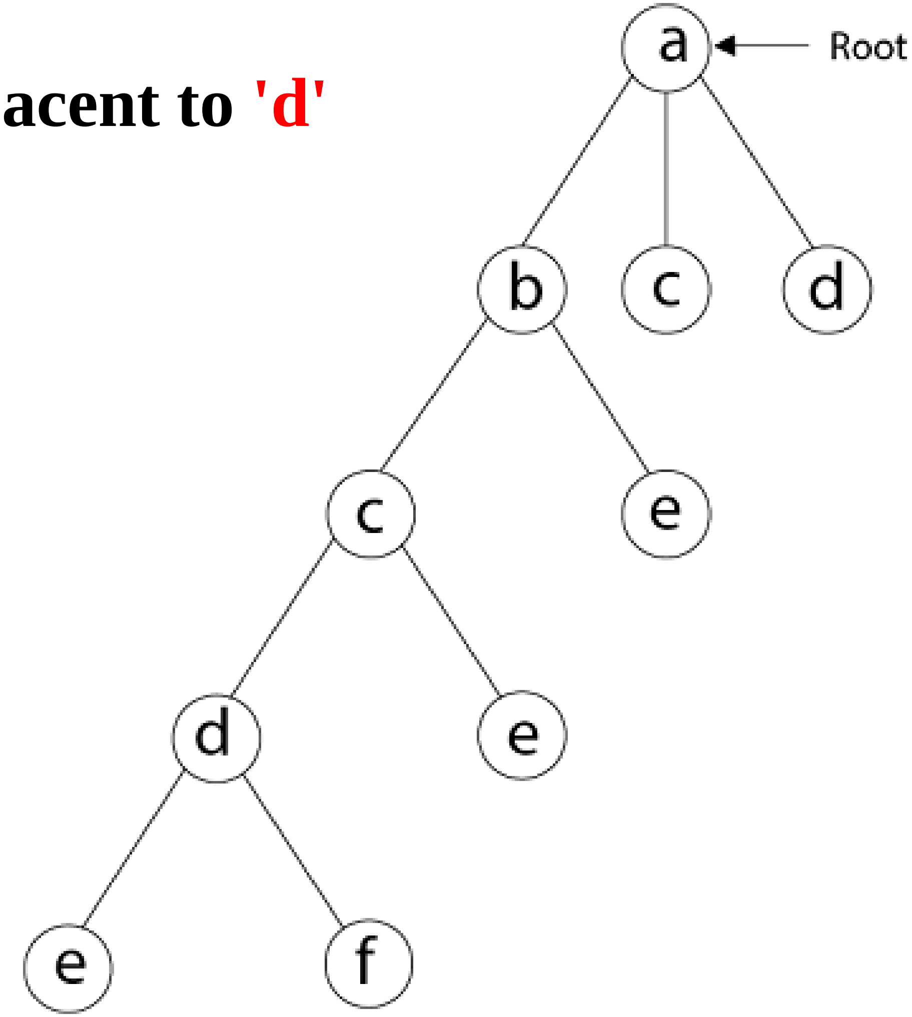


- **Next, we select 'd' adjacent to 'c'**



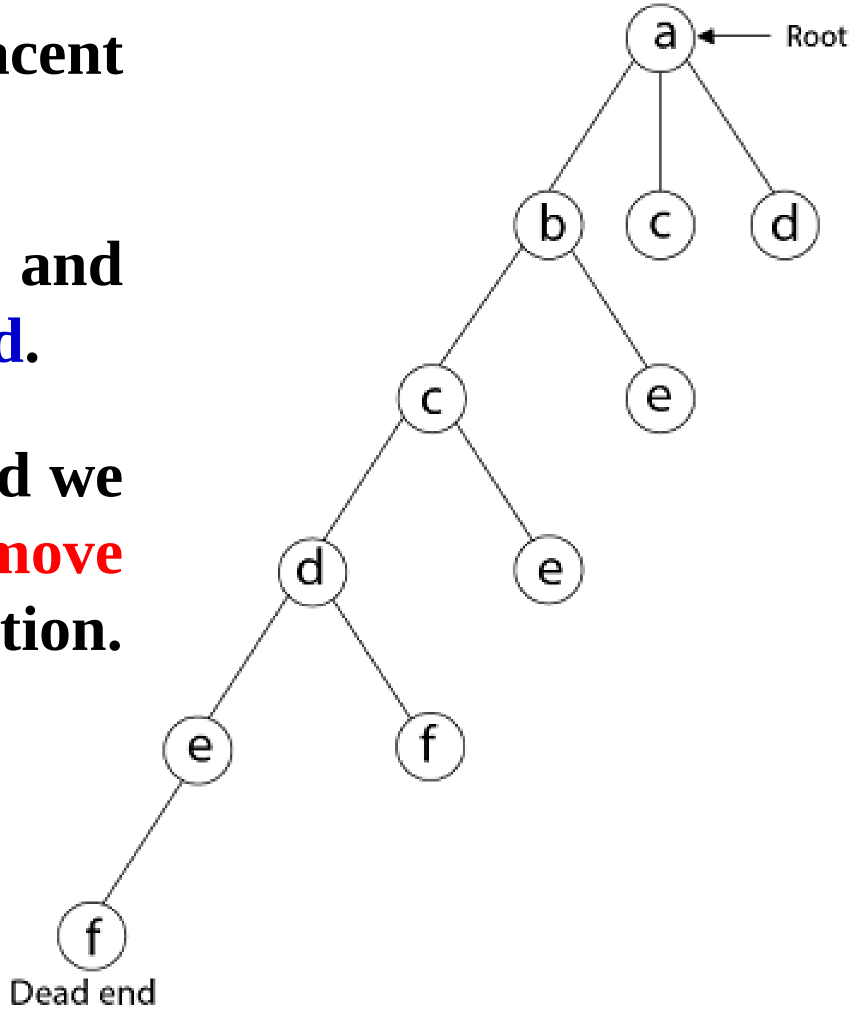
Hamiltonian Cycle Problem: Example

- **Next, we select 'e' adjacent to 'd'**



Hamiltonian Cycle Problem: Example

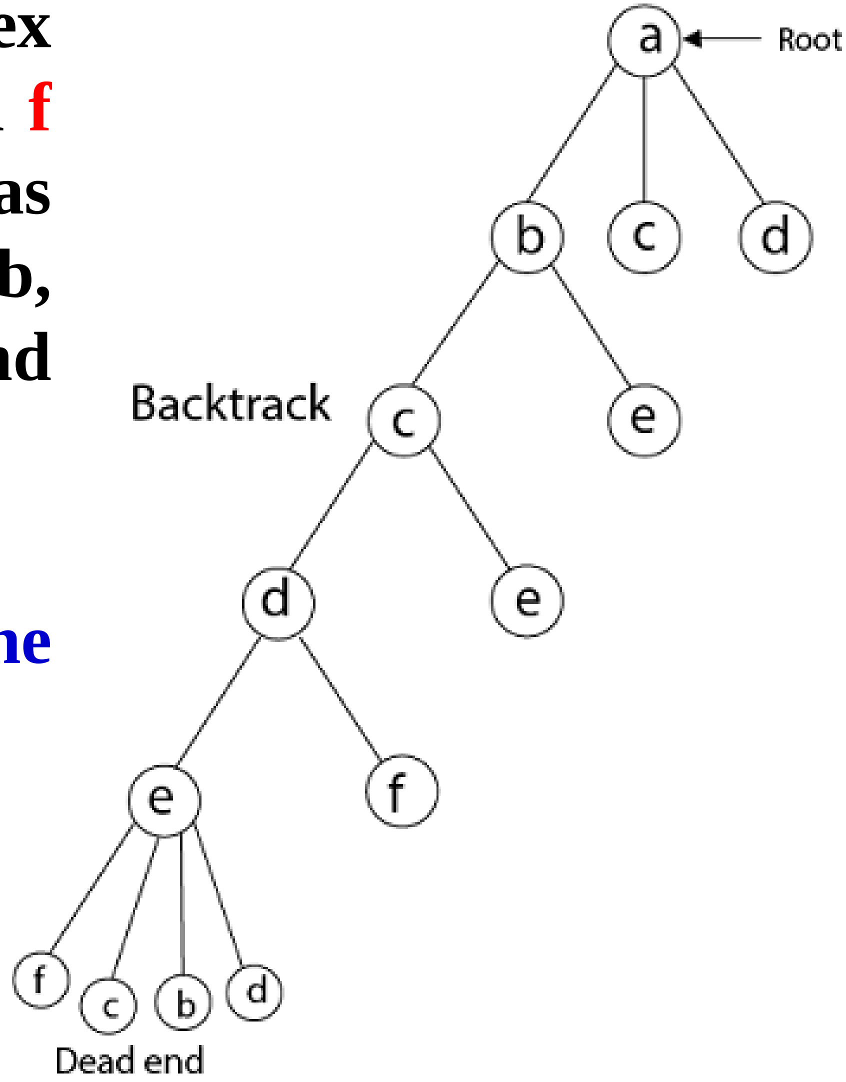
- Next, we select vertex 'f' adjacent to 'e'
- The vertex adjacent to 'f' is d and e, but they have already visited.
- Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.



Hamiltonian Cycle Problem: Example

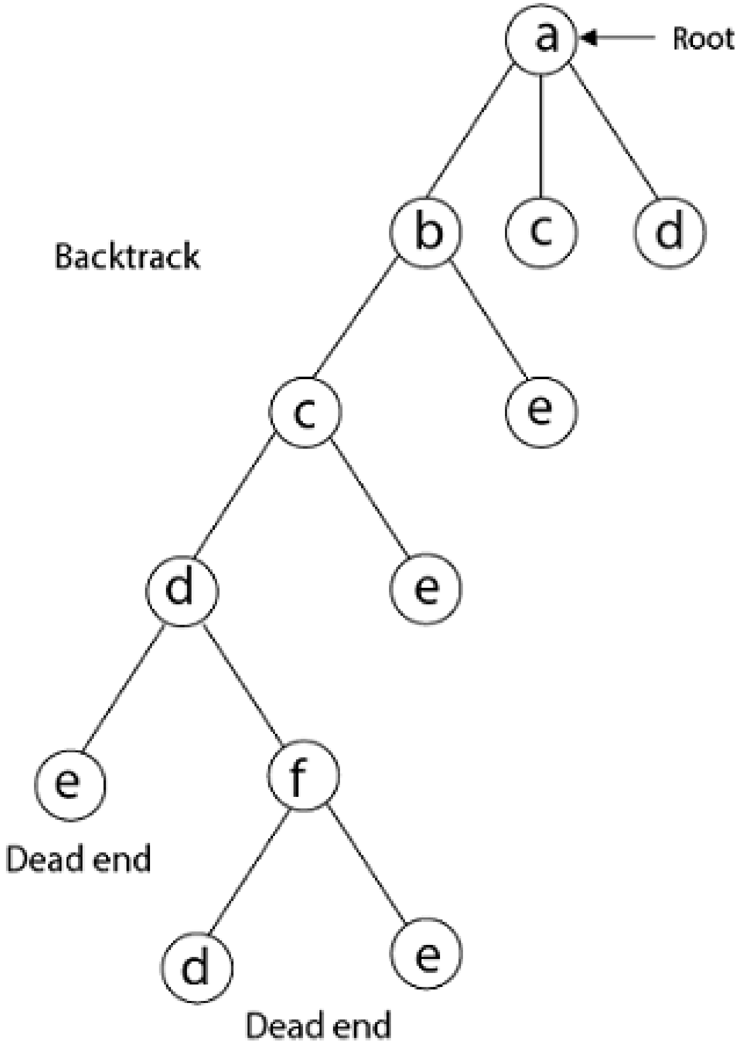
- From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited and we get the dead end.

- So, again we backtrack one step.



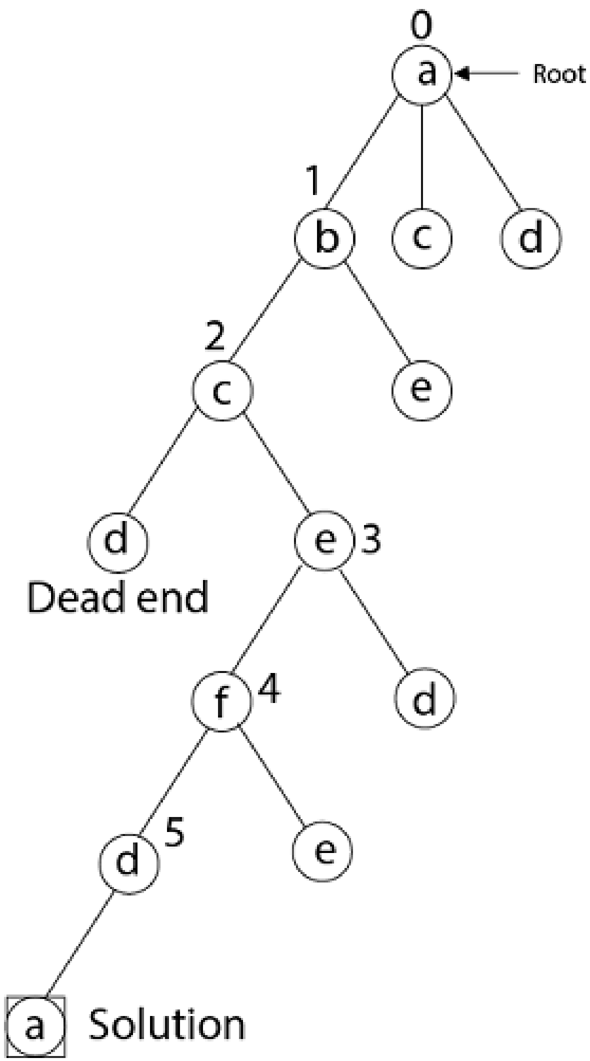
Hamiltonian Cycle Problem: Example

- 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.



Hamiltonian Cycle Problem: Example

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



Hamiltonian Cycle Problem

- **The following steps explain the working of backtracking approach:**
  - **First, create an empty path array and add a starting vertex 0 (root vertex) to it.**
  - **Next, start with vertex 1 (next to root - alphabetically) and then add other vertices one by one.**
  - **While adding vertices, check whether a given vertex is adjacent to the previously added vertex and hasn't been added already.**
  - **If any such vertex is found, add it to the path as part of the solution, otherwise, return false.**

THANK YOU!