DESIGN AND ANALYSIS OF ALGORITHMS (BTECCE21501)

Prof. Jayendra Jadhav

Vishwakarma University, Pune

UNIT 4

Backtracking Strategy

Outline

- **Backtracking**
  - General Strategy
  - N-Queens Problem
  - Graph Coloring
  - Subset Sum Problem
  - Knapsack Problem
  - Hamiltonian Cycle

Backtracking

- Suppose you have to make a **Series of Decisions**, among various choices, where
  - You don't have enough information **to know what to choose**
  - Each decision leads to a **new set of choices**
  - **Some sequence of choices** (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that "works"

Backtracking

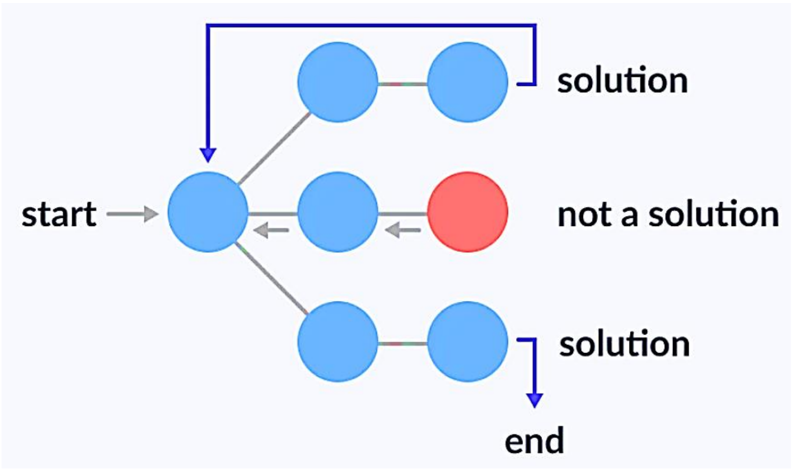
- **Backtracking is a problem-solving algorithmic technique that incrementally builds candidates for solutions and abandons candidates ("backtracks") as soon as it determines that the candidate cannot lead to a valid solution.**
- When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

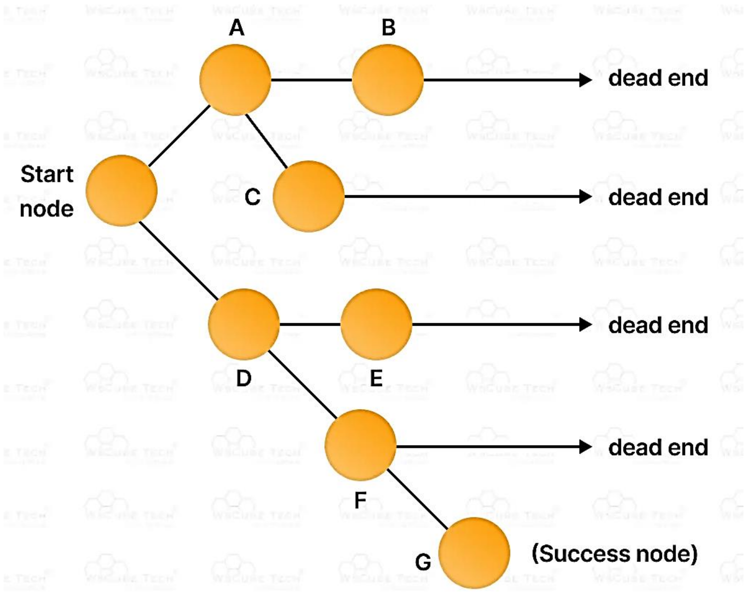
Backtracking

- A backtracking algorithm is a problem-solving algorithm that uses a **brute force approach** for finding the desired output.
- A backtracking algorithm is a way to solve problems by trying out different options one by one, and if an option doesn't work, it "backtracks" and tries the next option.
- It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like **Sudoku**.

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.





Backtracking Algorithms

- General Strategy
- N-Queens Problem
- Graph Coloring
- Subset Sum Problem
- Knapsack Problem
- Hamiltonian Cycle

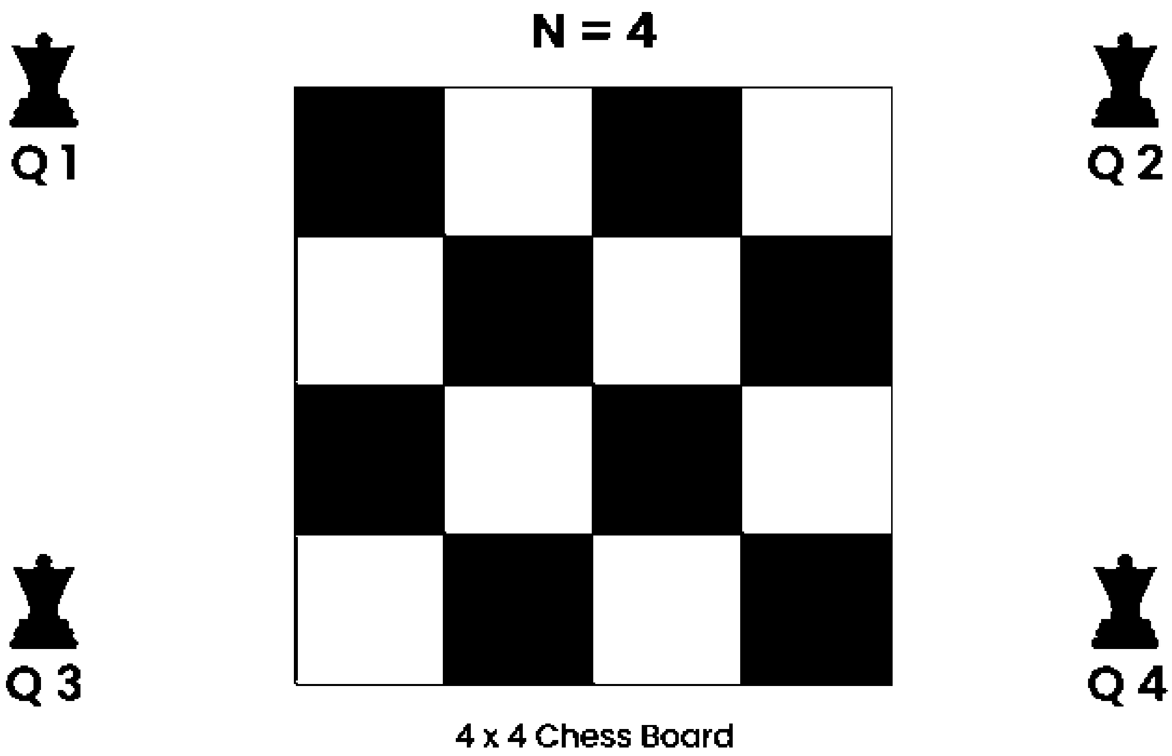
How Backtracking Works? General Strategy

1. <u>**Start at the Initial Position:**</u> The algorithm begins at the initial position or the root of the decision tree. This is the starting point from where different paths will be explored.
2. <u>**Make a Decision:**</u> At each step, the algorithm makes a decision that moves it forward. This could be moving in a certain direction in a maze or choosing a particular option in a decision tree.
3. <u>**Check for Validity:**</u> After making a decision, the algorithm checks if the current path is valid or if it meets the problem's constraints. If the path is invalid or leads to a dead end, the algorithm backtracks to the previous step.
4. <u>**Backtrack if Necessary:**</u> If a dead end is reached or if the path doesn't lead to a solution, the algorithm backtracks by undoing the last decision. It then tries a different option from the previous decision point.
5. <u>**Continue Exploring**</u>: The algorithm continues to explore different paths, making decisions, checking validity, and backtracking when necessary. This process repeats until a solution is found or all possible paths have been explored.
6. <u>**Find the Solution or Exhaust All Options:**</u> The algorithm stops when it finds a valid solution or when all possible paths have been explored and no solution exists.

N-Queens Problem

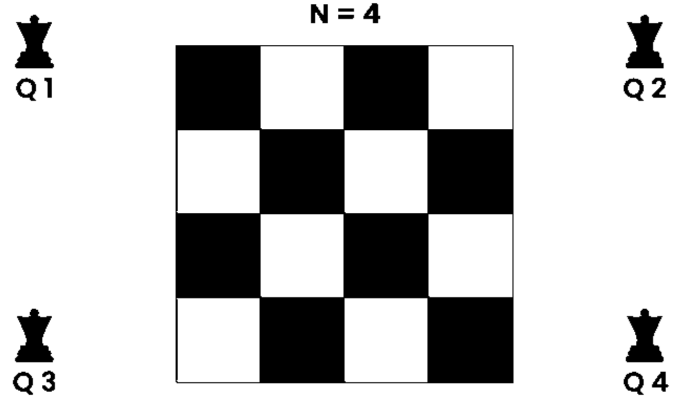
- The N-Queens problem is a classic example of backtracking.
- The task is to place **N Queens** on an **N×N Chessboard** such that **No Two Queens Attack Each Other**, meaning **No Two Queens can share the same Row, Column, or Diagonal**.
- The backtracking algorithm places queens one by one in different rows, checking for conflicts, and backtracking when a conflict is found until all queens are safely placed.

N-Queens Problem



4-Queens Problem

N = 4



4 x 4 Chess Board

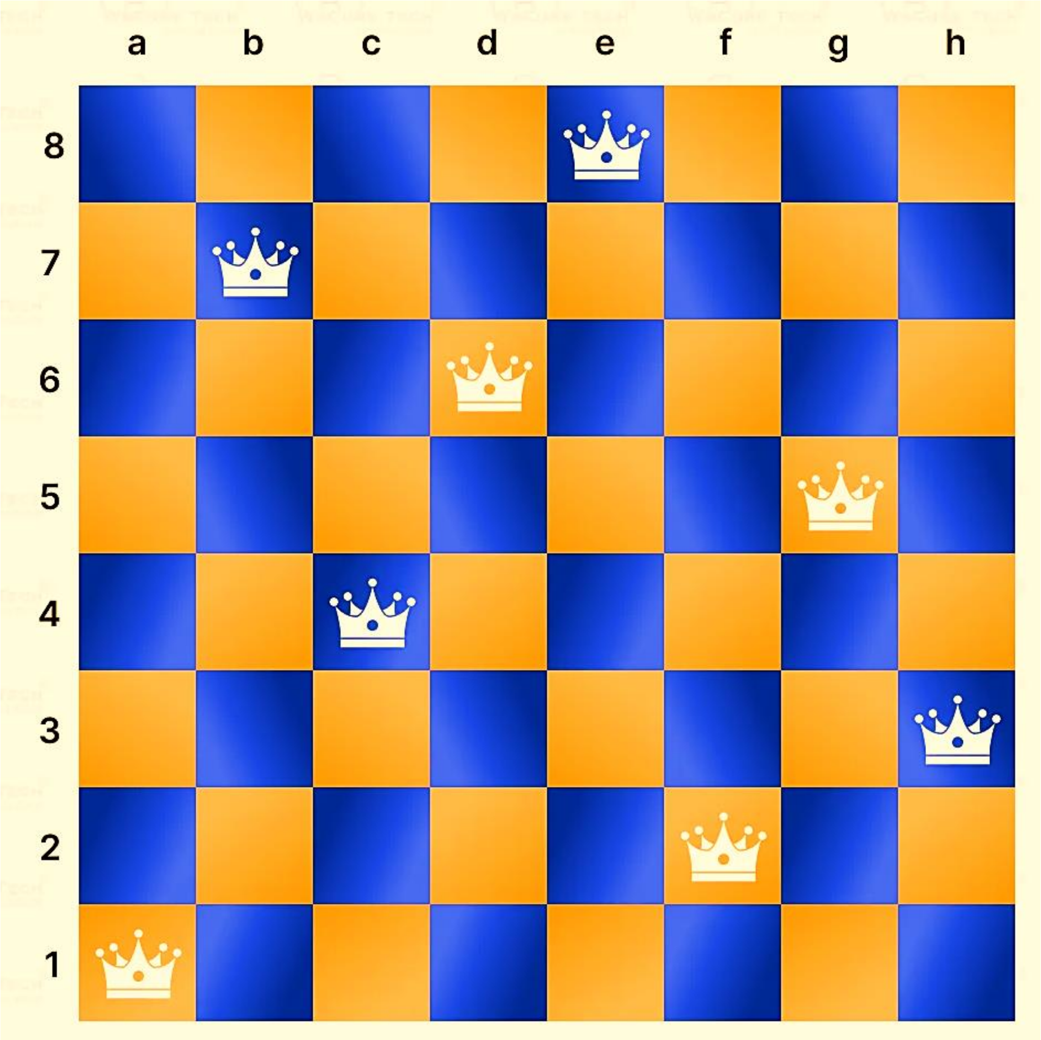
	Q1		
			Q2
Q3			
		Q4	

Solution 1

		Q1	
Q2			
			Q3
	Q4		

Solution 2

N-Queens Problem



N-Queens Problem

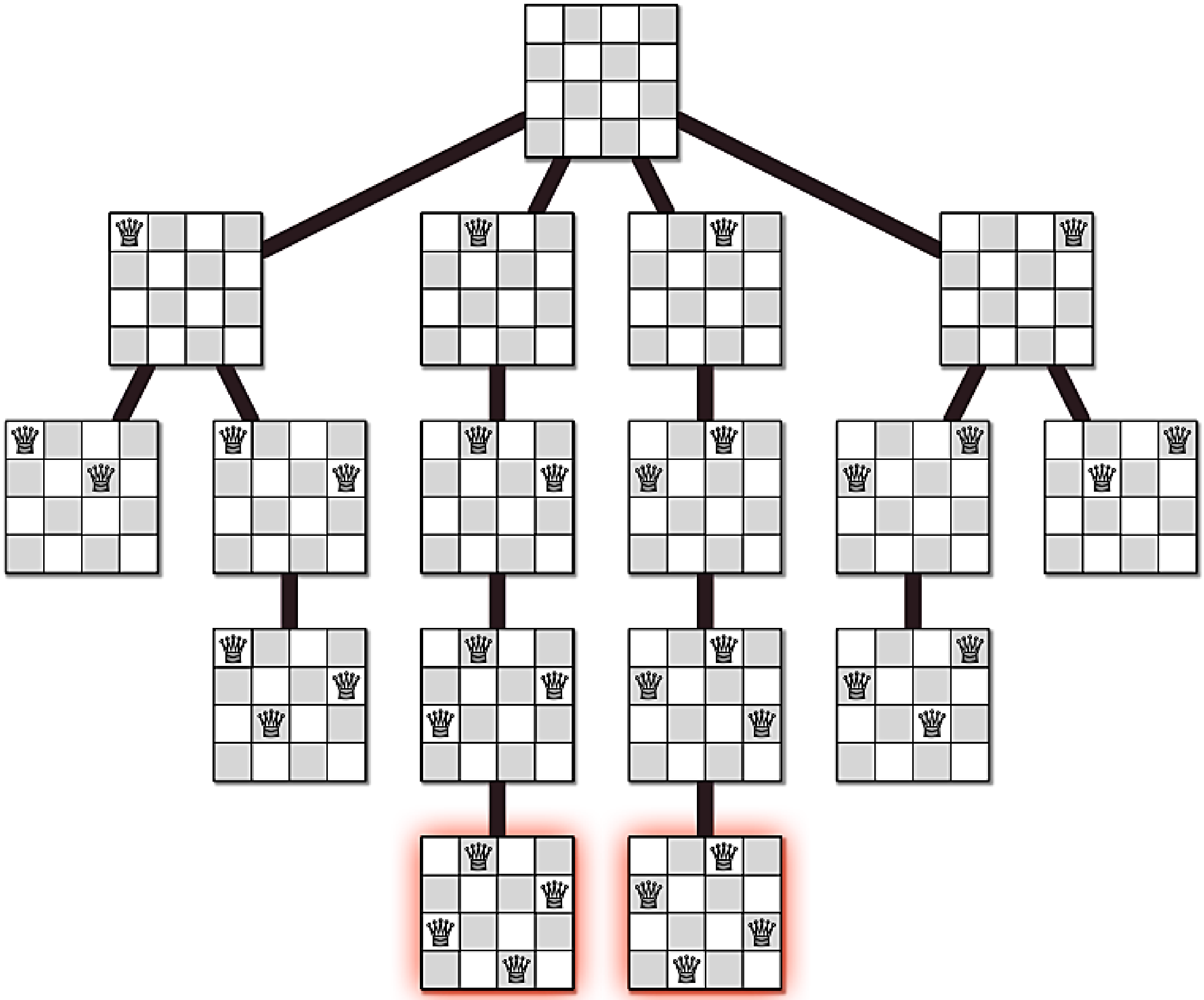
- **Approach using Backtracking:**
  - **Place queens row by row**: Starting with the first row, try to place a queen in each column. If a queen can be placed in a column without being attacked by previously placed queens, move to the next row. If not, backtrack and try the next column in the current row.
  - **Check for safety**: Before placing a queen in a column, ensure it isn't under attack from any queen in the previous rows. This involves checking the same column, the left diagonal, and the right diagonal.
  - **Backtrack**: If a valid placement isn't possible in the current row, remove the previously placed queen and try a different position for that queen.

N-Queens Problem

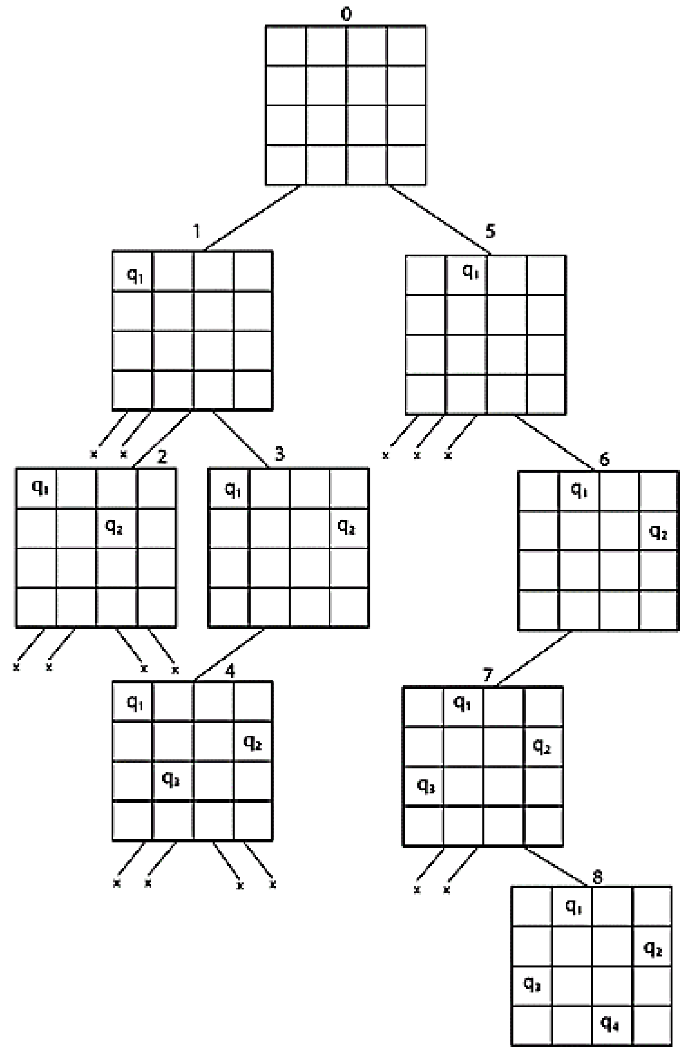
- **Algorithm Steps :**
  - Start by placing a queen in the first row.
  - For each row, try to place the queen in a valid column.
  - If a valid column is found, move to the next row and repeat.
  - If no valid position is found, backtrack to the previous row and move the queen to the next possible column.
  - If all queens are placed successfully, a solution is found.

```
N - Queens (k, n)
{
  For i ← 1 to n
     do if Place (k, i) then
  {
    x [k] ← i;
    if (k ==n) then
     write (x [1....n]);
    else
    N - Queens (k + 1, n);
  }
}
```

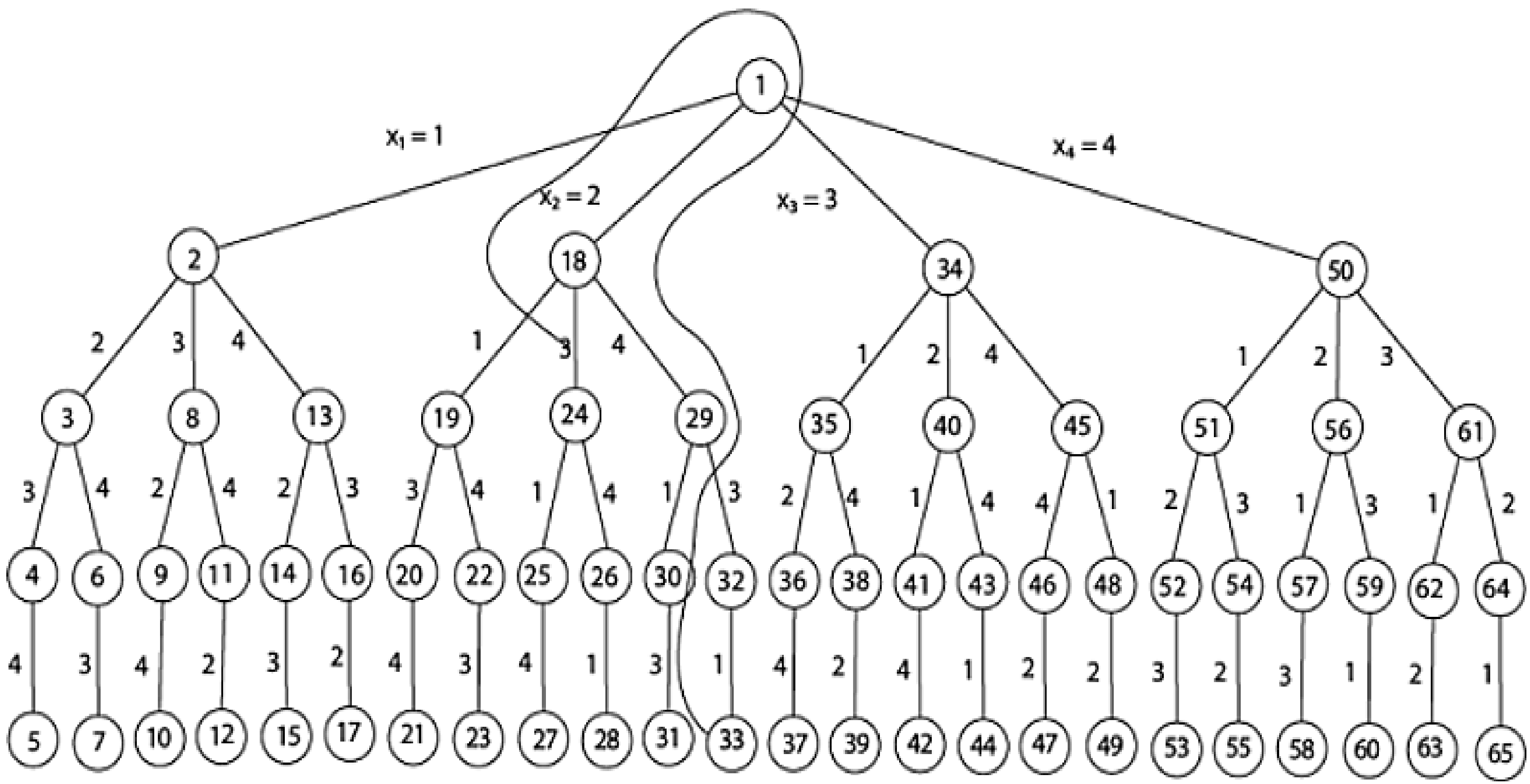
4-Queens Problem



4-Queens Problem

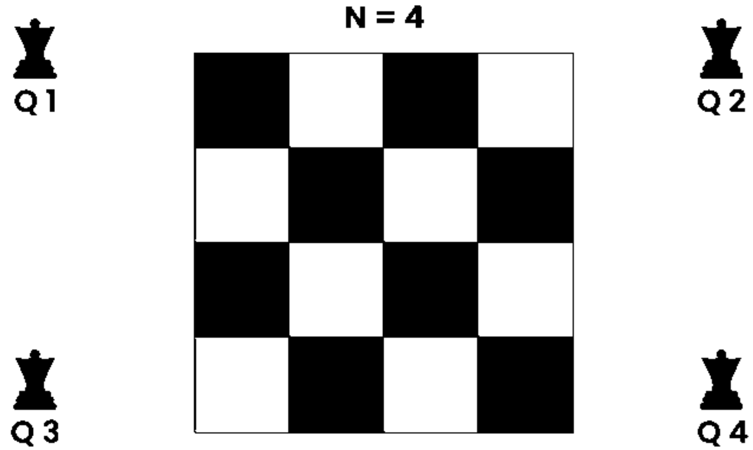


4-Queens Problem



4 - Queens solution space with nodes numbered in DFS

4-Queens Problem



4 x 4 Chess Board

	Q1		
			Q2
Q3			
		Q4	

Solution 1

		Q1	
Q2			
			Q3
	Q4		

Solution 2

Kanpsack Problem

- Given a set of items, each with a weight and profit, and a knapsack with a maximum capacity, we need to select items to maximize the profit without exceeding the knapsack's capacity.
- Each item can either be included or excluded, hence the 0/1 characteristic.

Kanpsack Problem Backtracking Approach

- **Start with the First Item**: Start with the first item and try to include it in the knapsack if it doesn't exceed the weight limit.
- **Recursive Exploration**:
  - If including the item doesn’t exceed the weight, recursively try to solve for the next item with the remaining capacity.
  - Also, consider the case of not including the item, and recursively solve for the next item with the same capacity.
- **Backtracking**:
  - If adding an item leads to exceeding the capacity, we backtrack by removing the item from the solution and explore the next possibility.
- **Track Maximum Profit**:
  - Keep track of the maximum profit achieved across all valid solutions.

Kanpsack Problem using Backtracking Approach

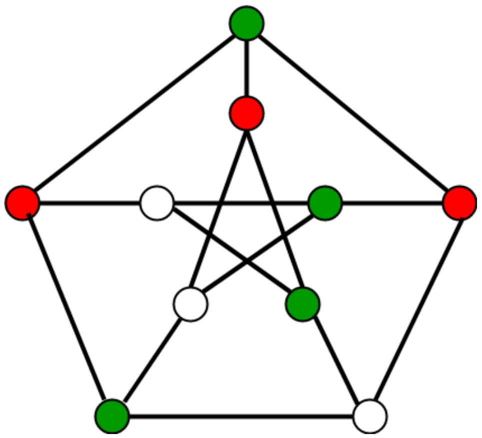
- Let's consider an example with the following items:

Items	Weight	Profit/Value
1	2	3
2	3	5
3	4	6
4	5	10

- Knapsack capacity W = 8

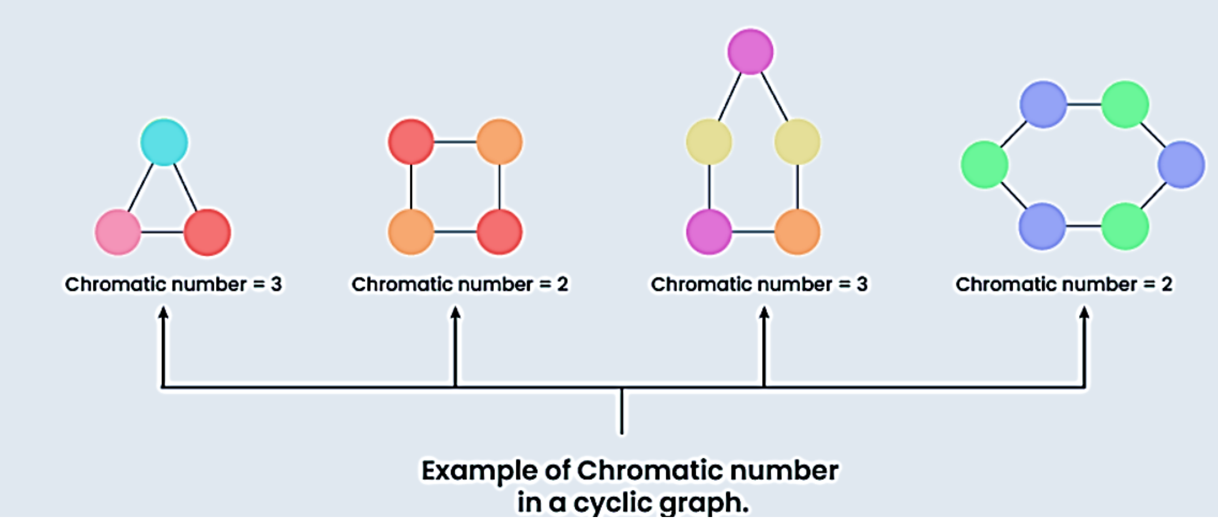
Graph Colouring Problem

- The **Graph Colouring Problem** is a classic problem in computer science and combinatorial optimization.
- The objective is to assign colours to the vertices of a graph so that **no two adjacent vertices share the same colour**.
- The challenge is to use the minimum number of colours possible, known as the **chromatic number** of the graph.



Graph Colouring Problem

- Chromatic Number
  - The minimum number of colours needed to colour a graph is called its chromatic number.
  - For example, the following can be coloured a minimum of 2 colours.



Graph Colouring Problem

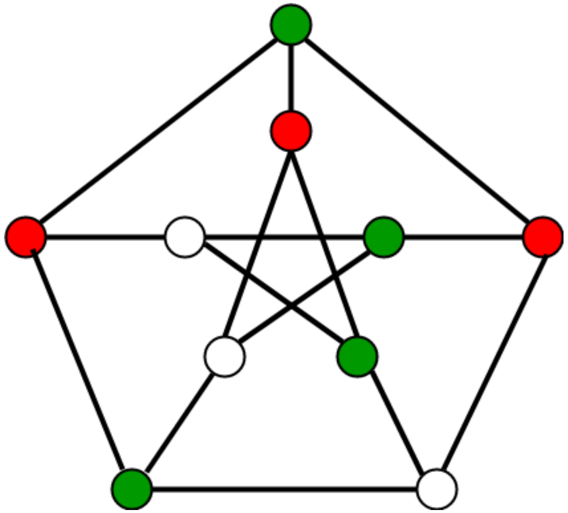
Problem Definition

Given:

- A graph $G = (V, E)$ where V is the set of vertices and E is the set of edges.
- The goal is to color each vertex in V such that no two adjacent vertices (vertices connected by an edge) have the same color.

Objective:

- Minimize the number of colors used.



Graph Colouring Problem

- Applications

  Graph colouring is widely used in areas such as:

  – **Scheduling**: Assigning times to exams or tasks where conflicts exist.

  – **Register Allocation**: Allocating limited CPU registers to variables in a program.

  – **Map Colouring**: Ensuring no two adjacent regions on a map share the same colour.

THANKYOU !!!