



Backtracking Algorithm

Brute force Approach

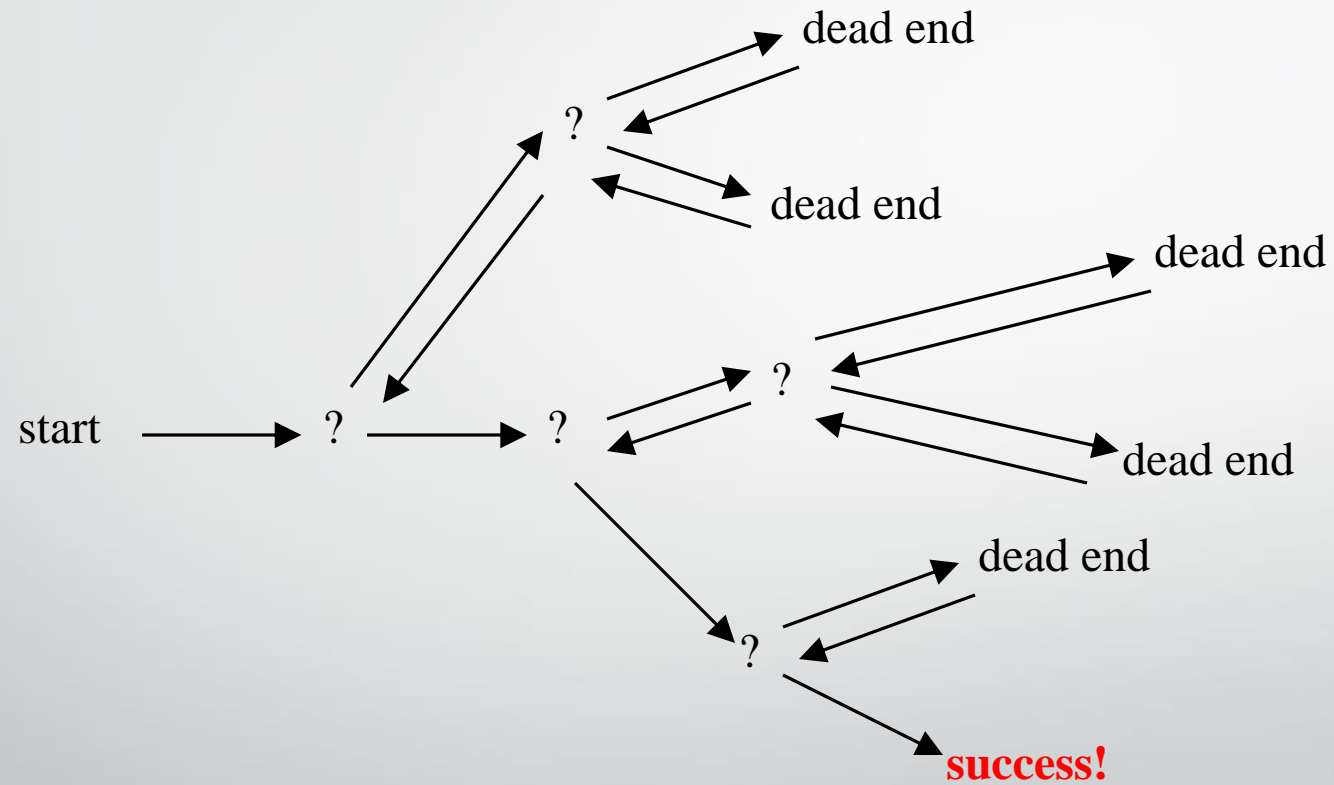
Introduction

- 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 way of trying out various sequences of decisions, until you find one that works.
- Based on depth-first recursive search.
- **Backtracking** is also called as modified **depth-first search** of a tree.
- It's a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.

Introduction ...

- **Backtracking** is used to solve problems, in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion.
- **Backtracking** is the procedure whereby, after determining that a node can lead to nothing but dead nodes, we go back (“backtrack”) to the node’s parent and proceed with the search on the next child.
- We call a node **nonpromising** if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise, we call it **promising**.
- In summary, backtracking consists of
 - Doing a depth-first search of a tree,
 - Checking whether each node is promising, and, if it is nonpromising, backtracking to the node’s parent.

Backtracking



Backtracking Approach

- 1. Tests whether solution has been found*
- 2. If solution found, return it*
- 3. Else for each choice that can be made*
 - a) Make that choice*
 - b) Recur*
 - c) If recursion returns a solution, return it*
- 4. If no choices remain, return failure*

Backtracking

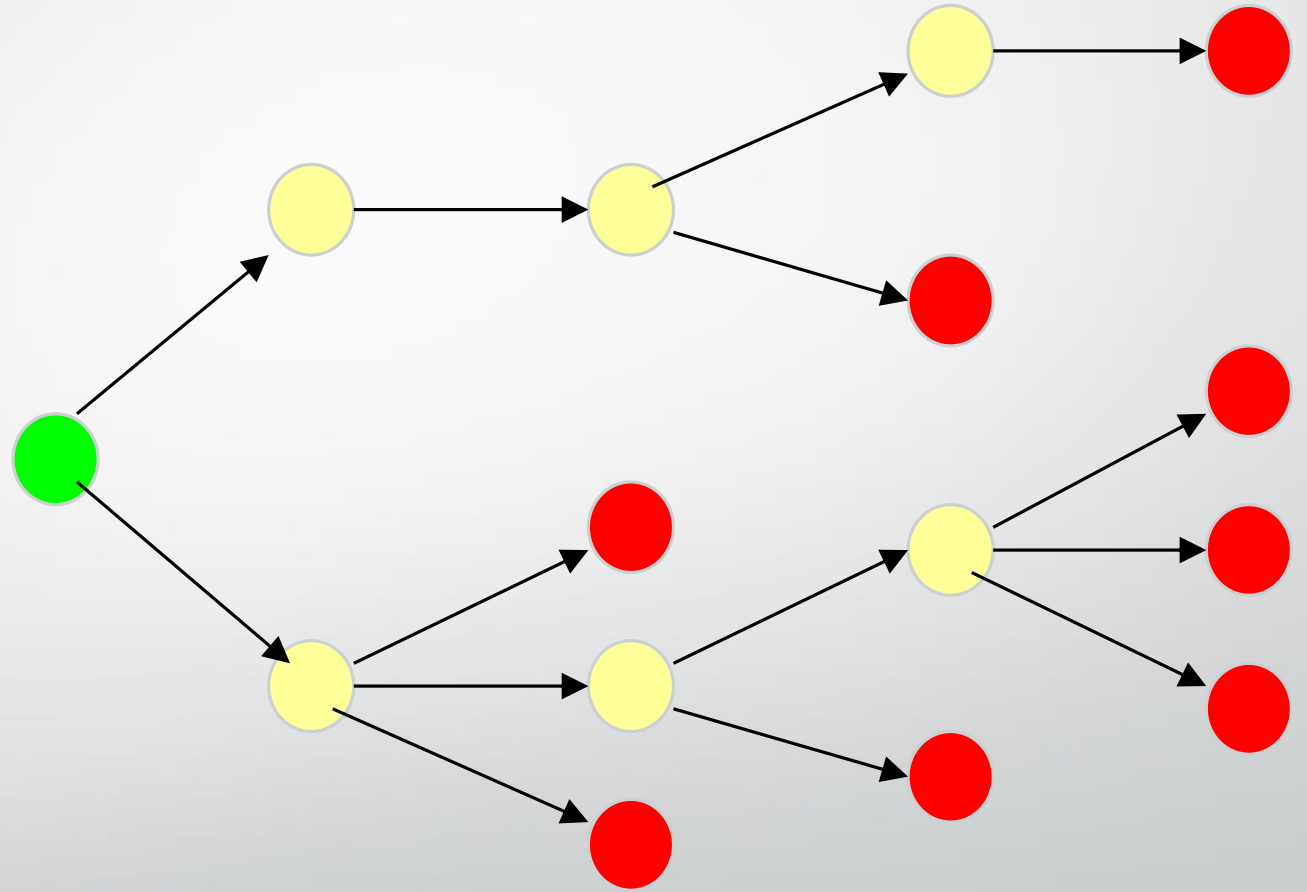
- Construct the **state space tree**:
 - Root represents an initial state
 - Nodes reflect specific choices made for a solution's components.
- Explore the state space tree using depth-first search
- “Prune” non-promising nodes
 - dfs stops exploring subtree rooted at nodes leading to no solutions and...
 - “backtracks” to its parent node

Terminology – STATE SPACE TREE

A tree is composed of nodes

There are three kinds of nodes:

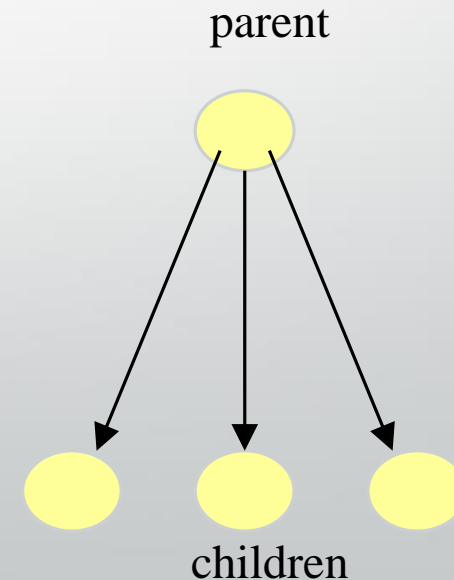
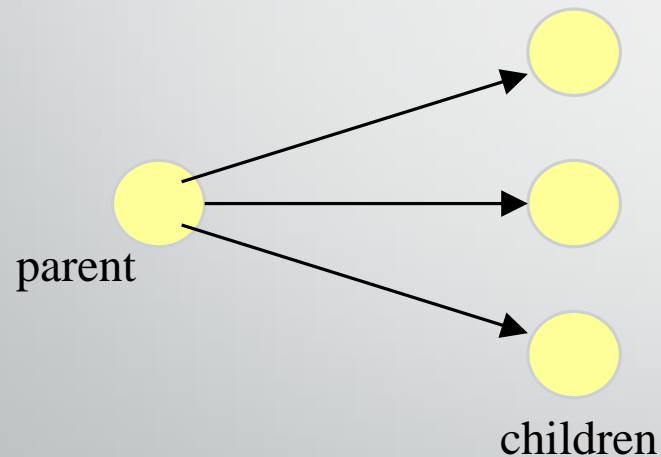
- The (one) root node
- Internal nodes
- Leaf nodes



Backtracking can be thought of as searching a tree for a particular “goal” leaf node.

Terminology I

- 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.
- Usually, we draw our trees *downward*, with the root at the top



The Backtracking algorithm

- Backtracking is really 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,
 - 3.1. Explore C
 - 3.1.1. If C was successful, return “success”
 4. Return “failure”

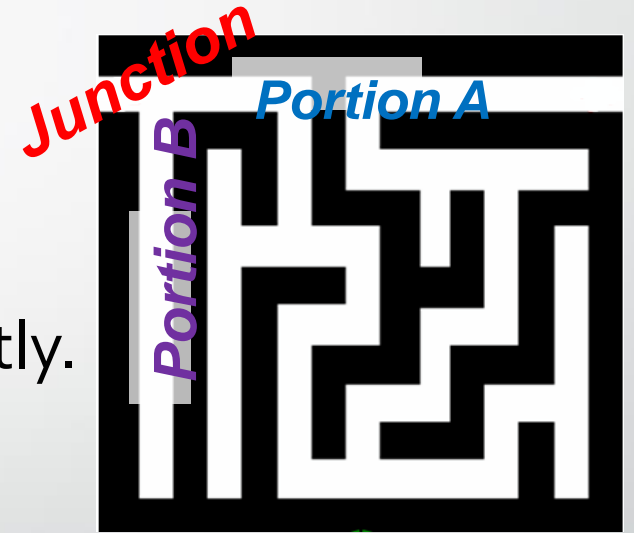
Backtracking examples

- The backtracking can be used in this cases:
 - Solving a maze
 - Coloring a map
 - Solving a puzzle
 - Sum of subsets
 - N queens problem etc.,

Solving a maze

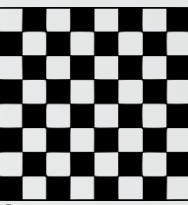


- Given a maze, find a path from start to finish.
- At each intersection, you have to decide between three or fewer choices:
 - Go straight
 - Go left
 - Go right
- You don't have enough information to choose correctly.
- Each choice leads to another set of choices.
- One or more sequences of choices may (or may not) lead to a solution
- The backtracking strategy says to try each choice, one after the other, if you ever get stuck, "backtrack" to the junction and try the next choice.



If you try all choices and never found a way out, then there IS no solution to the maze.

The N-Queens Problem

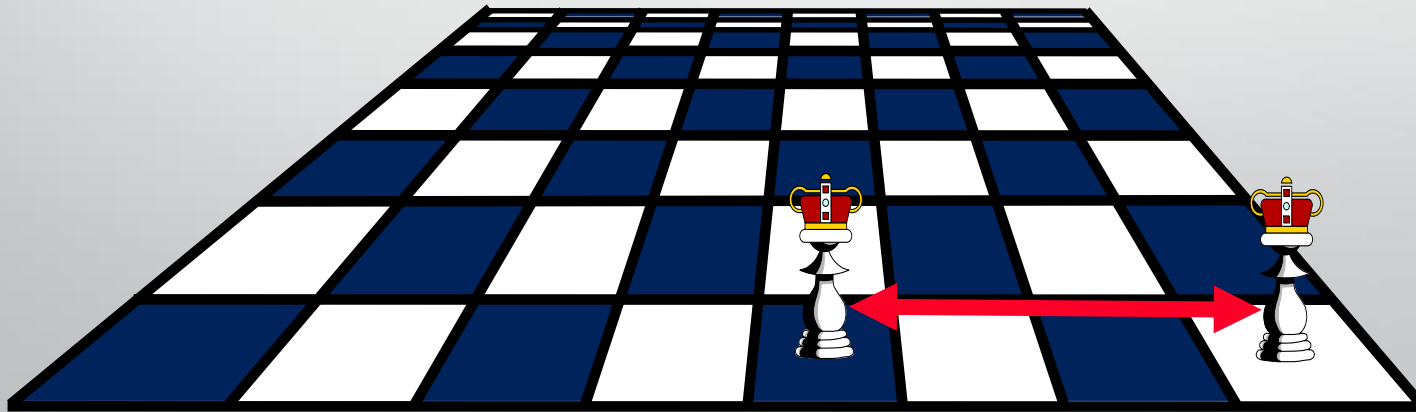


- Try to place N queens on an $N * N$ board such that none of the queens can attack another queen.
- Remember that queens can move horizontally, vertically, or diagonally any distance.
- A queen can attack another queen, if two of them are on the same row, column, or diagonal.
- The backtracking strategy works as follows:
 - 1) Place a queen on the first available square in row 1.
 - 2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
 - 3) Continue in this fashion until either:
 - a) you have solved the problem, or
 - b) you get stuck.

When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.

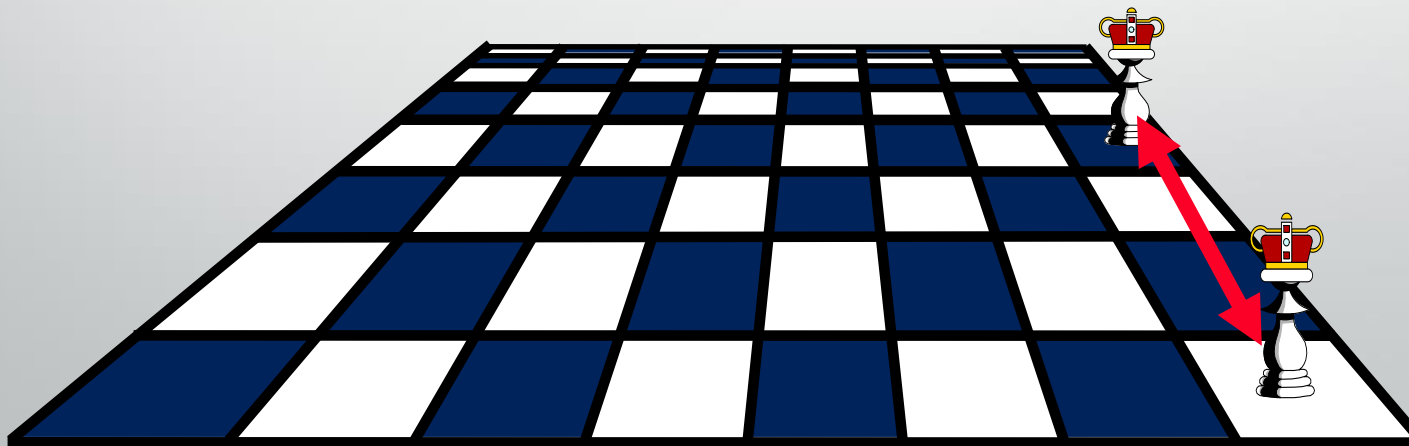
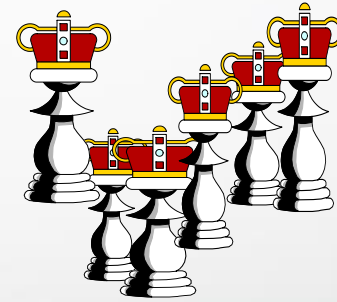
The N-Queens Problem

Two queens are not allowed in the same row...



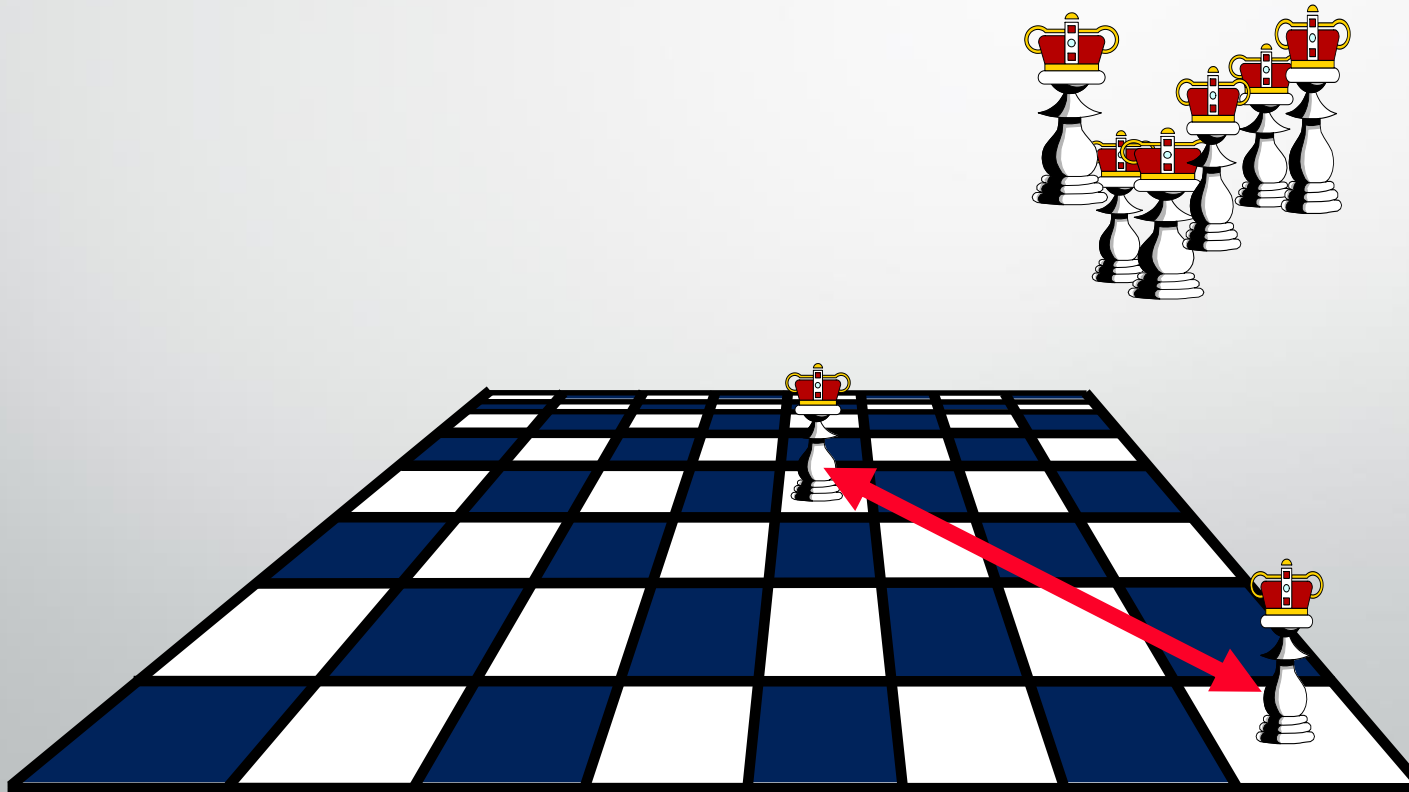
The N-Queens Problem

Two queens are not allowed in the same column...

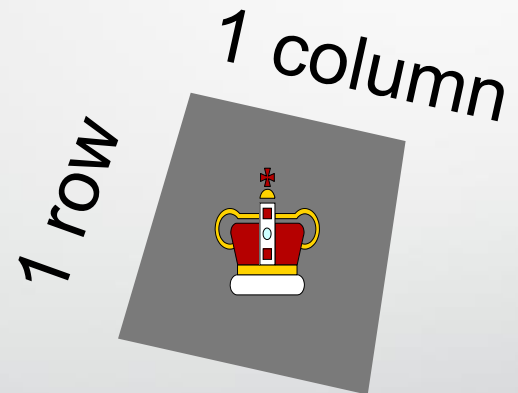


The N-Queens Problem

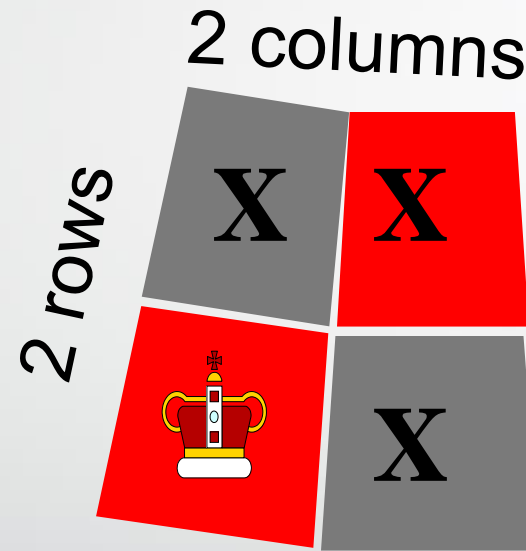
Two queens are not allowed along the same diagonal.



1 queen Problem

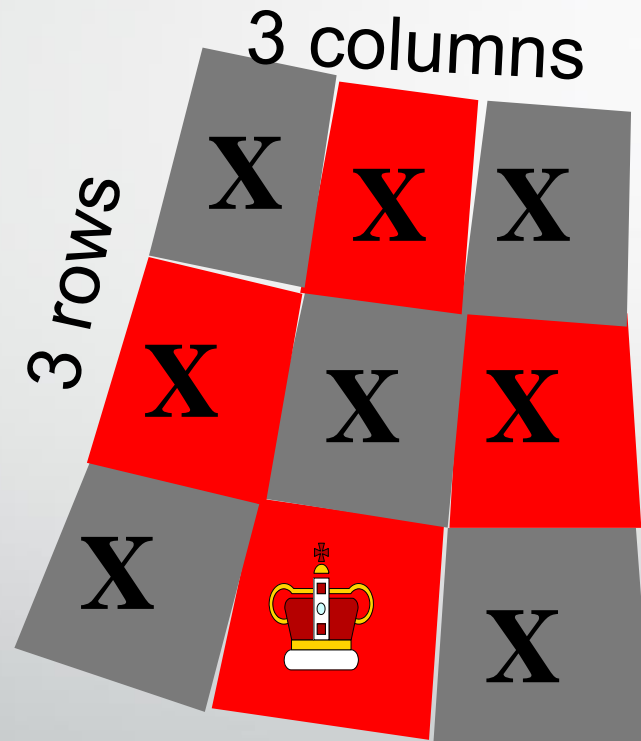


2 queen Problem



2 queen problem ::: No Solution exist

3 queen Problem



3 queen problem ::: No Solution exist

4 queen Problem

4 columns

4 rows

















| | | | |
|--|--|--|--|
| | | | |
| | | | |
| | | | |
| | | | |



4 queen Problem

4 columns

4 rows

| | | | |
|--|---|---|---|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**need
backtrack.....**

4 queen Problem

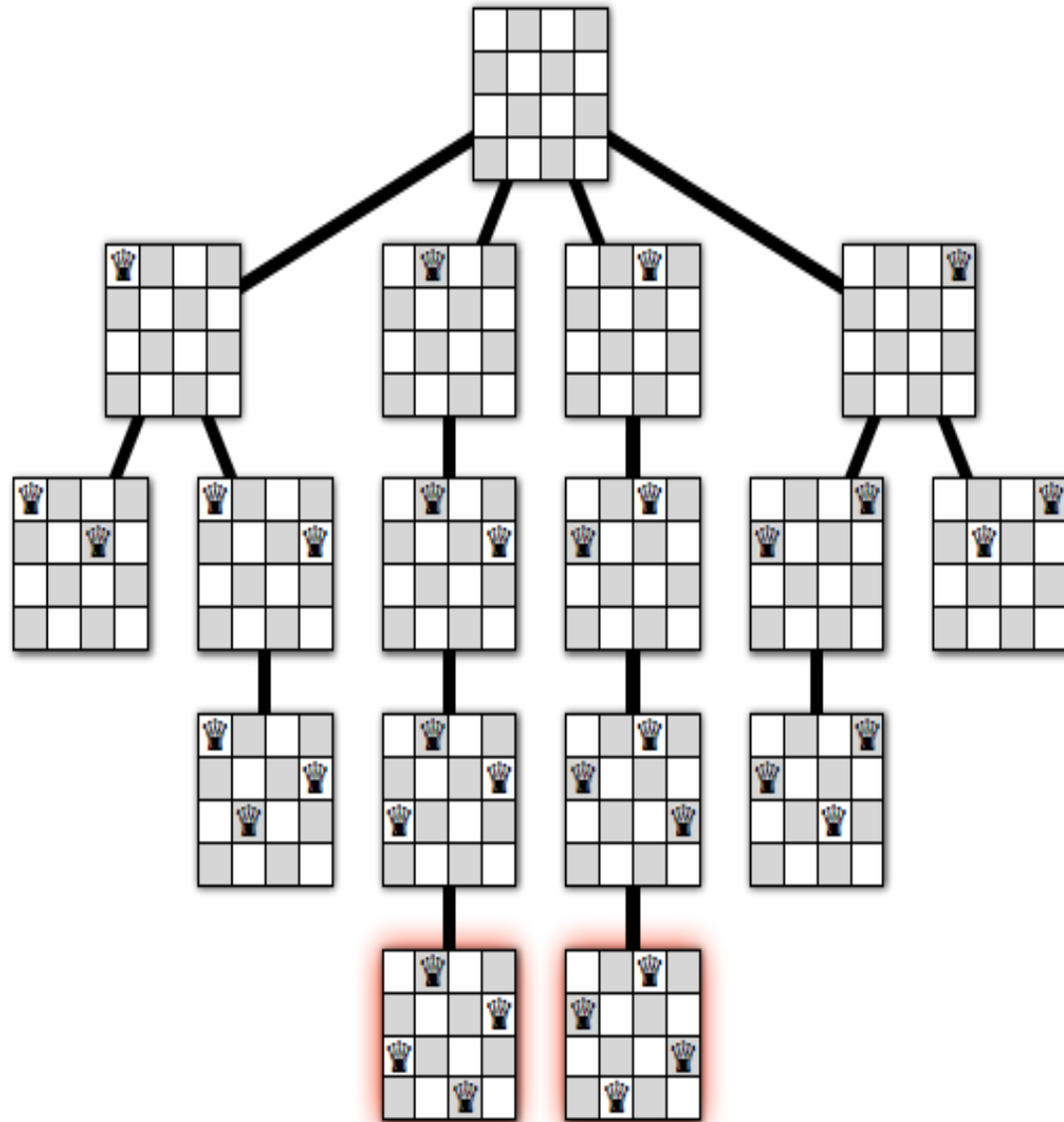
4 columns

4 rows

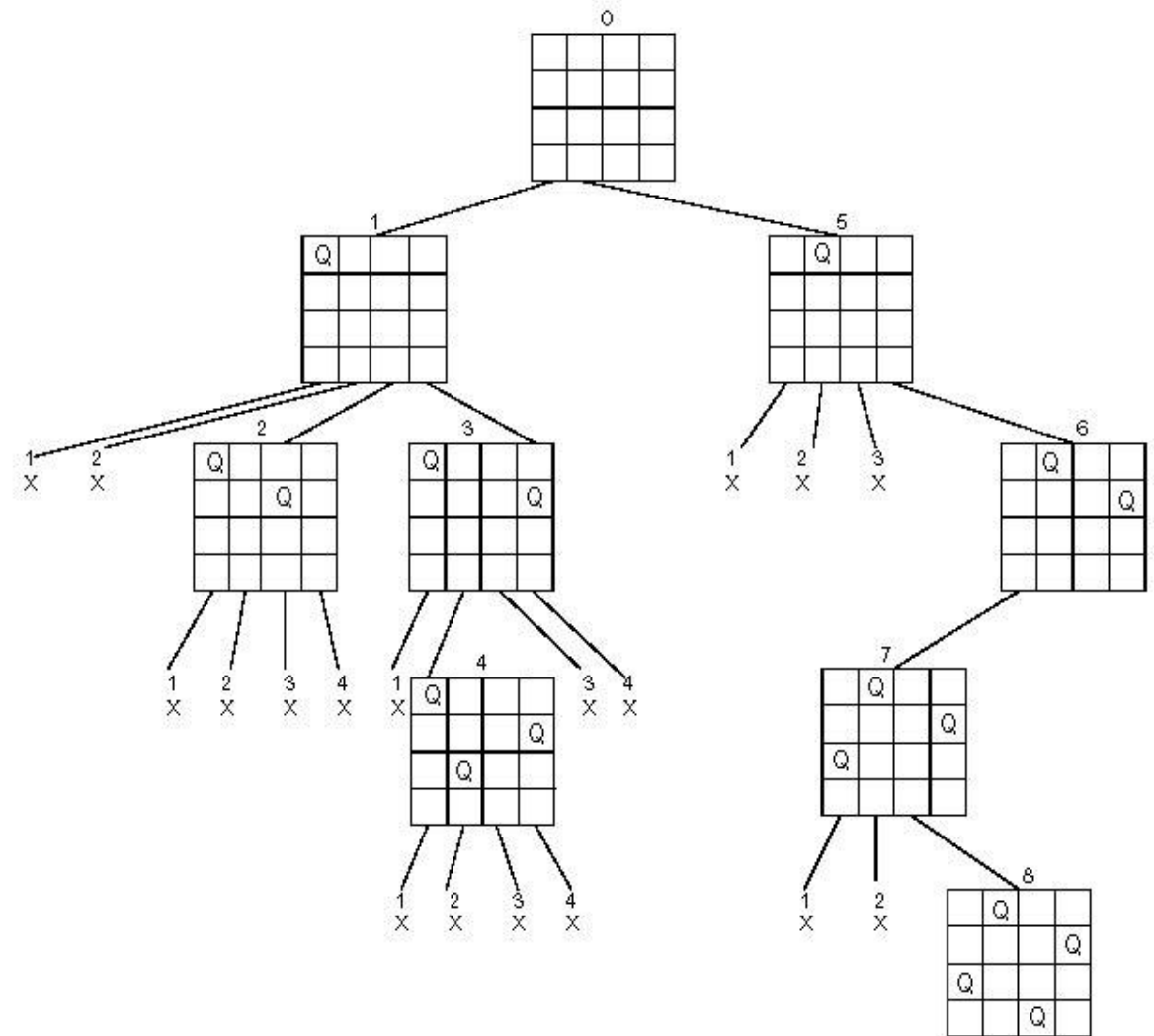
| | | | |
|---|---|---|---|
| × | × | ♔ | × |
| ♔ | × | × | × |
| × | × | × | ♔ |
| × | ♔ | × | × |

Solved.....

State Space Tree of the Four-queens Problem



The complete recursion tree

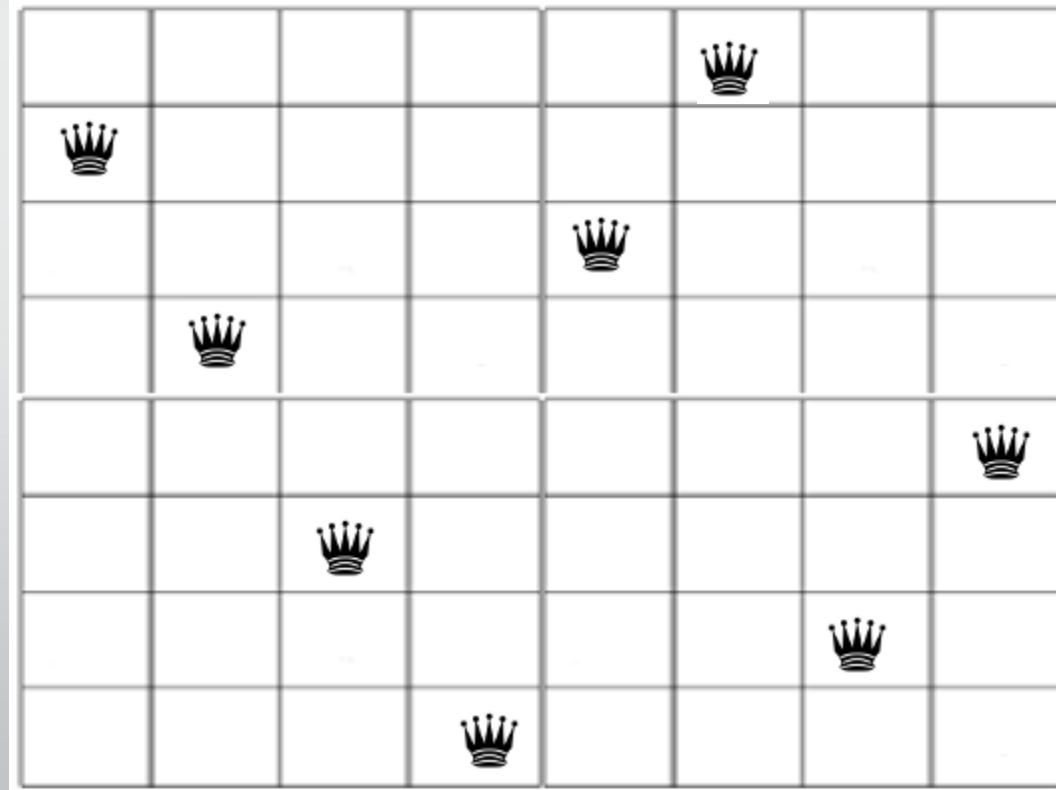


The DFS tree

8 queen Problem

8 columns

8 rows



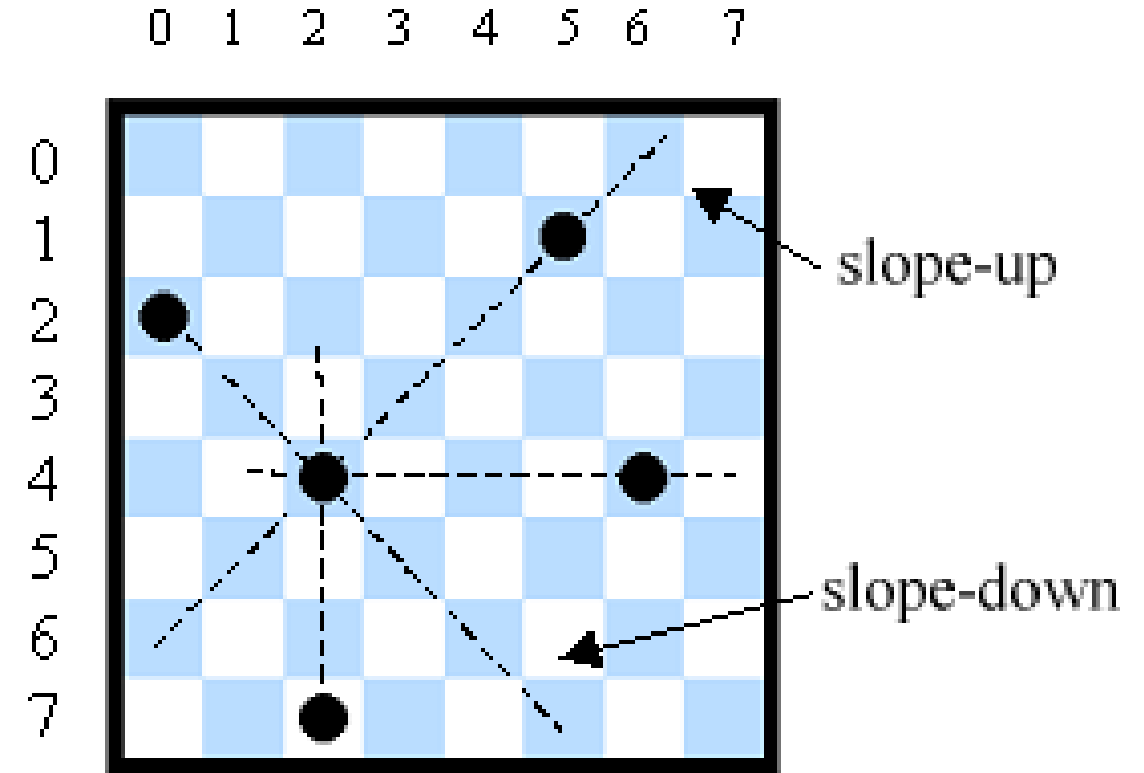
Algorithm for N Queen problem

- 1) Start in the leftmost column
- 2) If all queens are placed
return true
- 3) Try all rows in the current column. Do following for every tried row.
 - a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
 - b) If placing the queen in [row, column] leads to a solution then return true.
 - c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
- 3) If all rows have been tried and nothing worked, return false to trigger backtracking.

Algorithm for N Queen problem

Check if Queens placed at (x_1, y_1) and (x_2, y_2) are safe:

- $x_1 == x_2$ means same rows,
- $y_1 == y_2$ means same columns
- $|x_2 - x_1| == |y_2 - y_1|$ means they are placed in diagonals



Different Approach for solving N Queen problem

- Create a solution matrix of the same structure as chess board.
- Whenever place a queen in the chess board, mark that particular cell in solution matrix.
- At the end print the solution matrix, the marked cells will show the positions of the queens in the chess board.

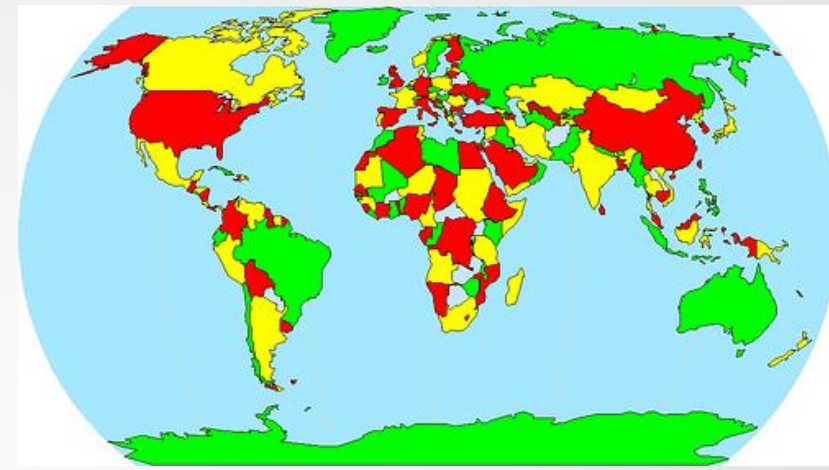
In above approach Solution matrix takes $O(N^2)$ space.

Can reduce it to $O(N)$?

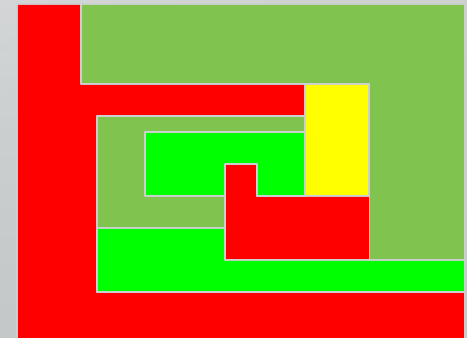
Yes, We will solve it by taking one dimensional array and consider `solution[1] = 2` as "Queen at 1st row is placed at 2nd column."

I.e. ***result[i]=j means queen at i-th row is placed at j-th column.***

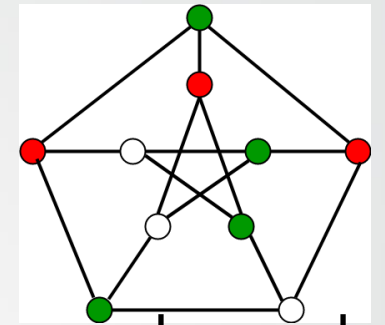
Coloring a map



- You wish to color a map with not more than m colors
 - For example $m=4$, {red, yellow, green, blue}
- Adjacent countries must be in different colors
- You don't have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking
- This type of problem is also called as chromatic Number problem.



Graph colouring problem



- Given an undirected graph and a number m , determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

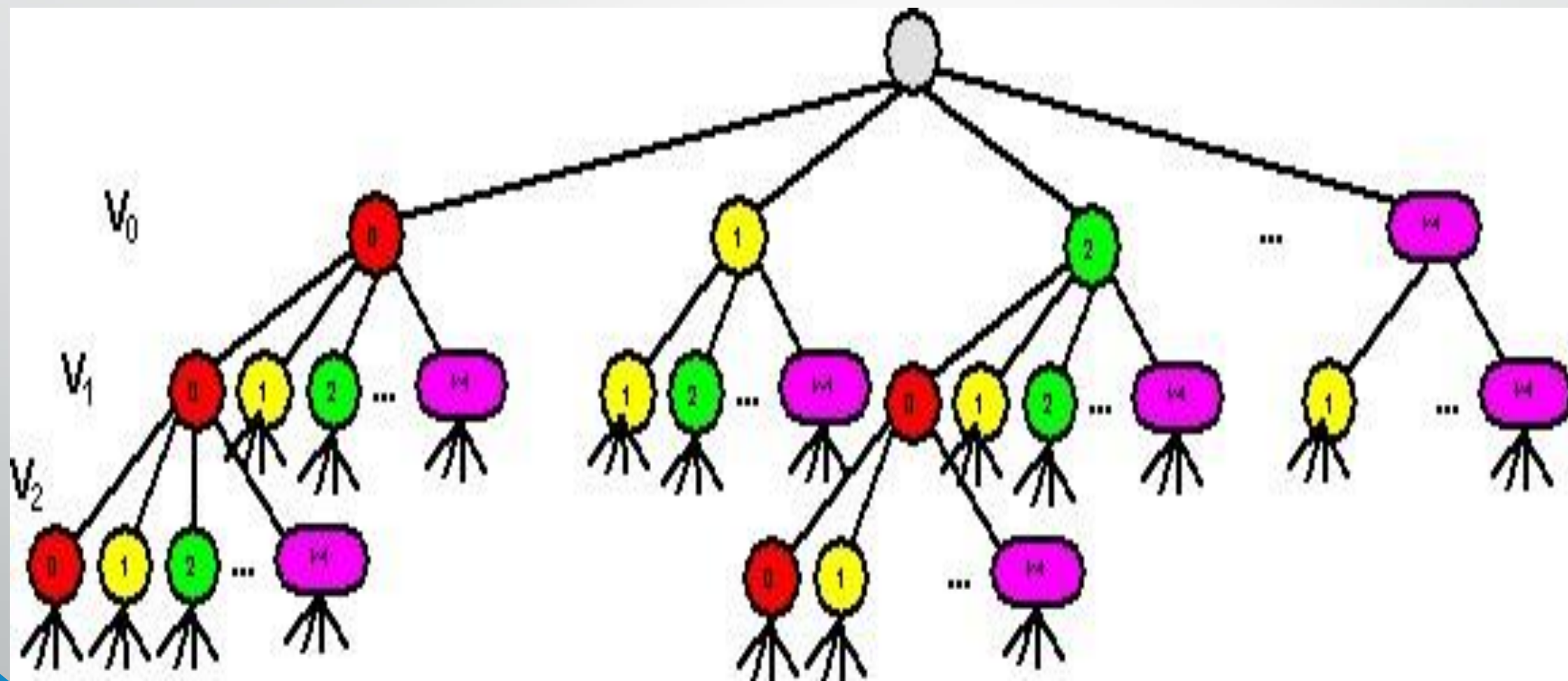
Input:

- 1) A 2D array `graph[V][V]` where V is the number of vertices in graph and `graph[V][V]` is adjacency matrix representation of the graph. A value `graph[i][j]` is 1 if there is a direct edge from i to j , otherwise `graph[i][j]` is 0.
- 2) An integer m which is maximum number of colors that can be used.

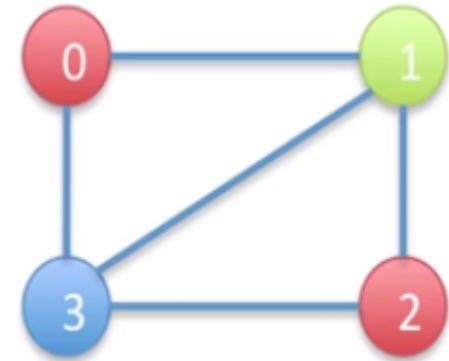
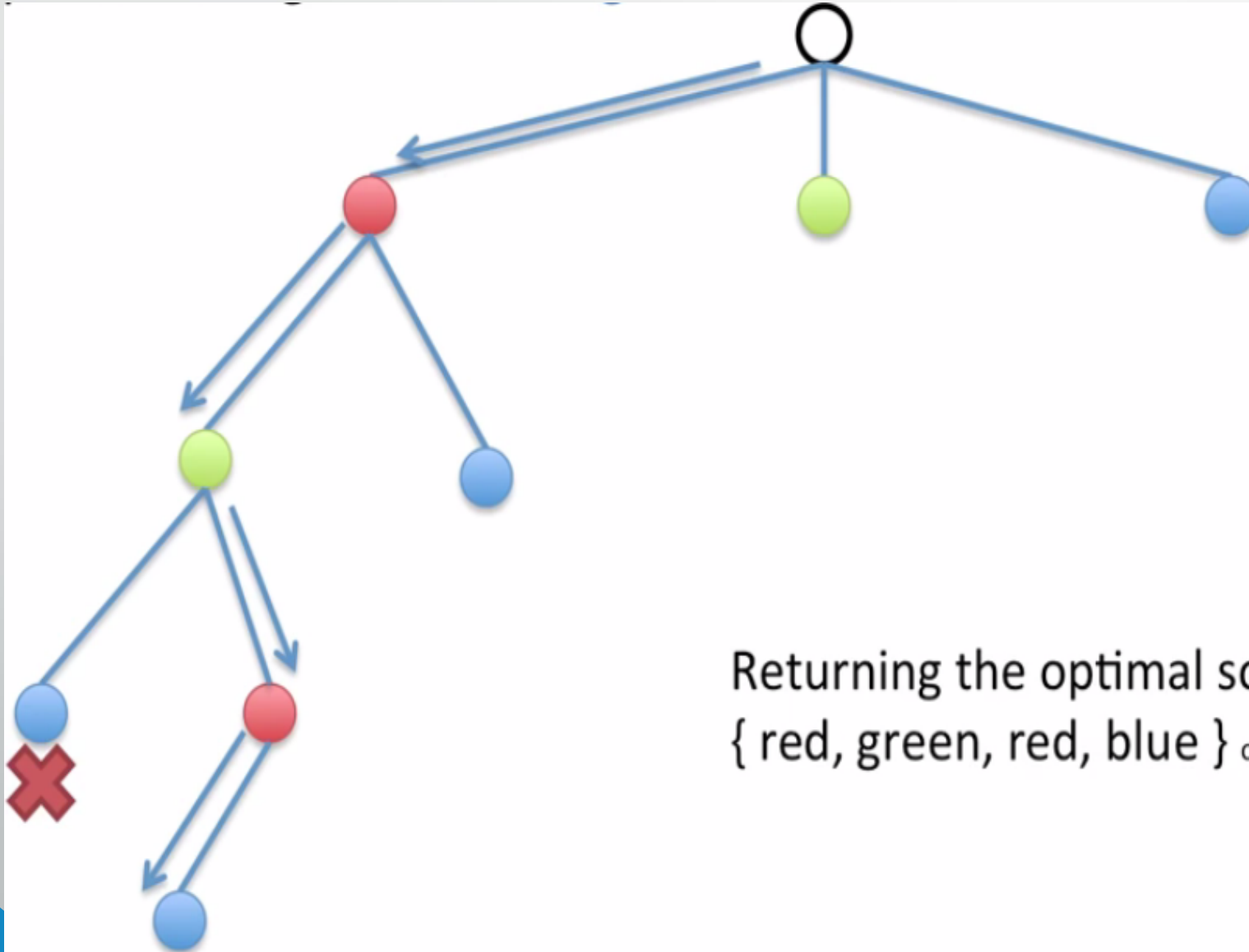
Output:

- An array `color[V]` that should have numbers from 1 to m . `color[i]` should represent the color assigned to the i^{th} vertex. The code should also return false if the graph cannot be colored with m colors.

State Space Tree of Graph colouring problem



State Space Tree of Graph colouring problem



Returning the optimal solution of:
 $\{ \text{red, green, red, blue} \}$ or $\{1,2,1,3\}$

Algorithm

- If all colors are assigned,
print vertex assigned colors
- Else
 - a. Trying all possible colors, assign a color to the vertex
 - b. If color assignment is possible, recursively assign colors to next vertices
 - c. If color assignment is not possible, de-assign color, return False

Sum of subsets Problem

- Given a set of positive integers, and a value *sum* S , find out if there exist a subset in array whose sum is equal to given *sum* S .
- It is assumed that the input set is unique (no duplicates are presented).
- **Simple (Naive) solution for the exponential time:**
- we will generate every possible set (the [power set](#)), and then check if the sum of any of these sets equals the sum S .
 - For example: arr = [1, 2, 3] sum = 5
 - All possible sets: [] [1] [2] [3] [1, 2] [1, 3] [2, 3] [1, 2, 3]
 - we can get a sum of 5 by adding the elements in the set [2, 3].

Algorithm

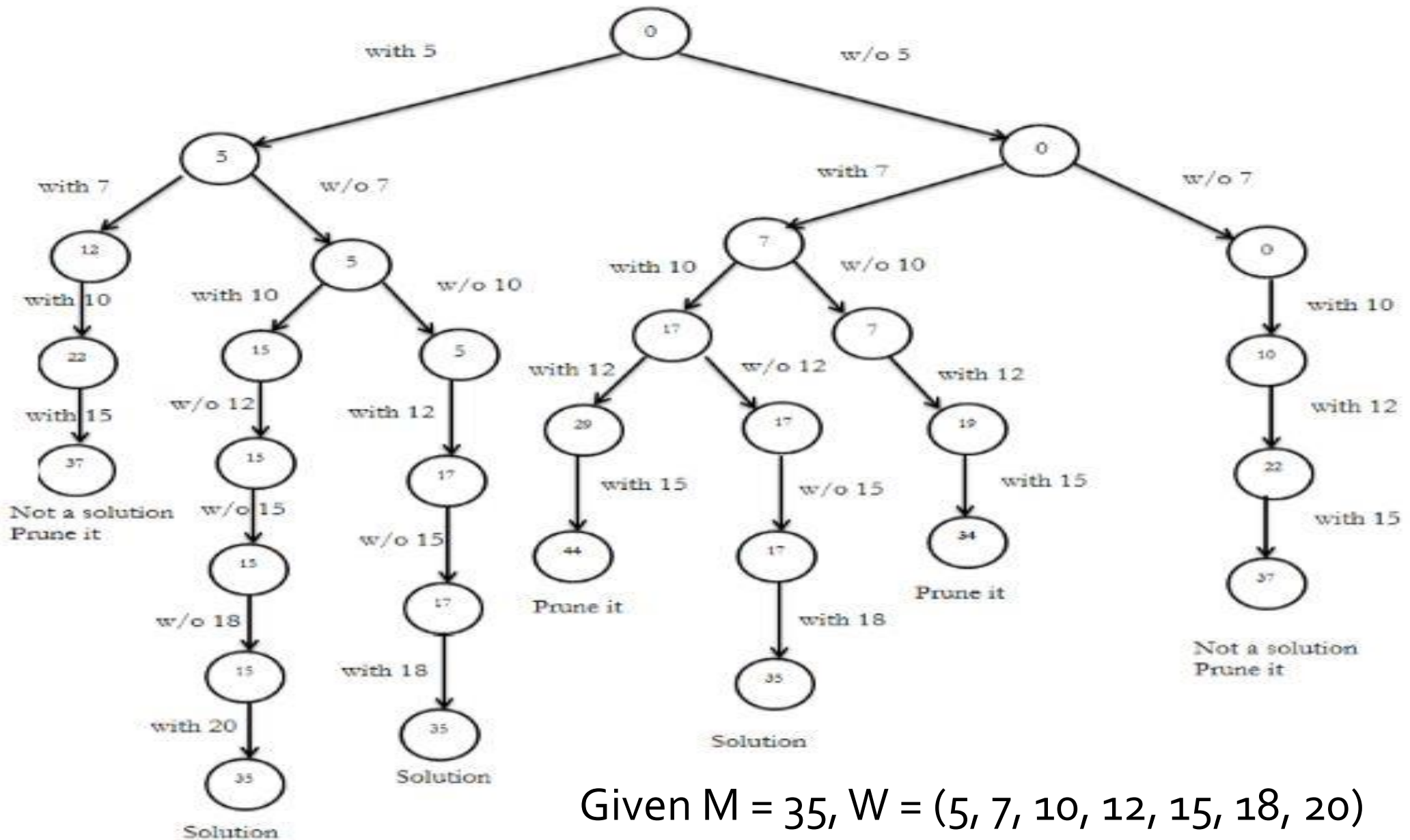
- Let, S is a set of elements and m is the expected sum of subsets. Then:
 1. Start with an empty set.
 2. Add to the subset, the next element from the list.
 3. If the subset is having sum m then stop with that subset as solution.
 4. If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.
 5. If the subset is feasible then repeat step 2.
 6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

Example

Given $M = 35$, $W = (5, 7, 10, 12, 15, 18, 20)$

| Initially subset = {} | Sum = 0 | Description |
|-----------------------|--------------------|---|
| 5 | 5 | Then add next element. |
| 5, 7 | 12, i.e. $12 < 35$ | Add next element. |
| 5, 7, 10 | 22, i.e. $22 < 35$ | Add next element. |
| 5, 7, 10, 12 | 34, i.e. $34 < 35$ | Add next element. |
| 5, 7, 10, 12, 15 | 49 | Sum exceeds $M = 35$. Hence backtrack. |
| 5, 7, 10, 12, 18 | 52 | Sum exceeds $M = 35$. Hence backtrack. |
| 5, 7, 10, 12, 20 | 54 | Sum exceeds $M = 35$. Hence backtrack. |
| 5, 12, 15 | 32 | Add next element. |
| 5, 12, 15, 18 | 50 | Not feasible. Therefore backtrack |
| 5, 12, 18 | 35 | Solution obtained as $M = 35$ |

State Space Tree



Given $M = 35$, $W = (5, 7, 10, 12, 15, 18, 20)$

Chapter Summary

- Backtracking is an algorithm design technique for solving problems in which the number of choices grows at least exponentially with their instant size.
- This approach makes it possible to solve many large instances of NP-hard problems in an acceptable amount of time.
- Backtracking constructs its state-space tree in the depth-first search fashion in the majority of its applications.