Problem Solving Strategies and Computational Approaches SCS1304

Handout 7 : Backtracking

Prof Prasad Wimalaratne PhD(Salford),SMIEEE

Algorithmic Design Techniques..

- Brute Force
- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- Dynamic Programming
  - Brute force through all possible solutions, storing solutions to subproblems to avoid repeat computation
- Backtracking
  - A clever form of exhaustive search
- Branch and Bound (next lesson)

https://courses.cs.washington.edu/courses/cse373/15su/lectures/lecture22.pptx

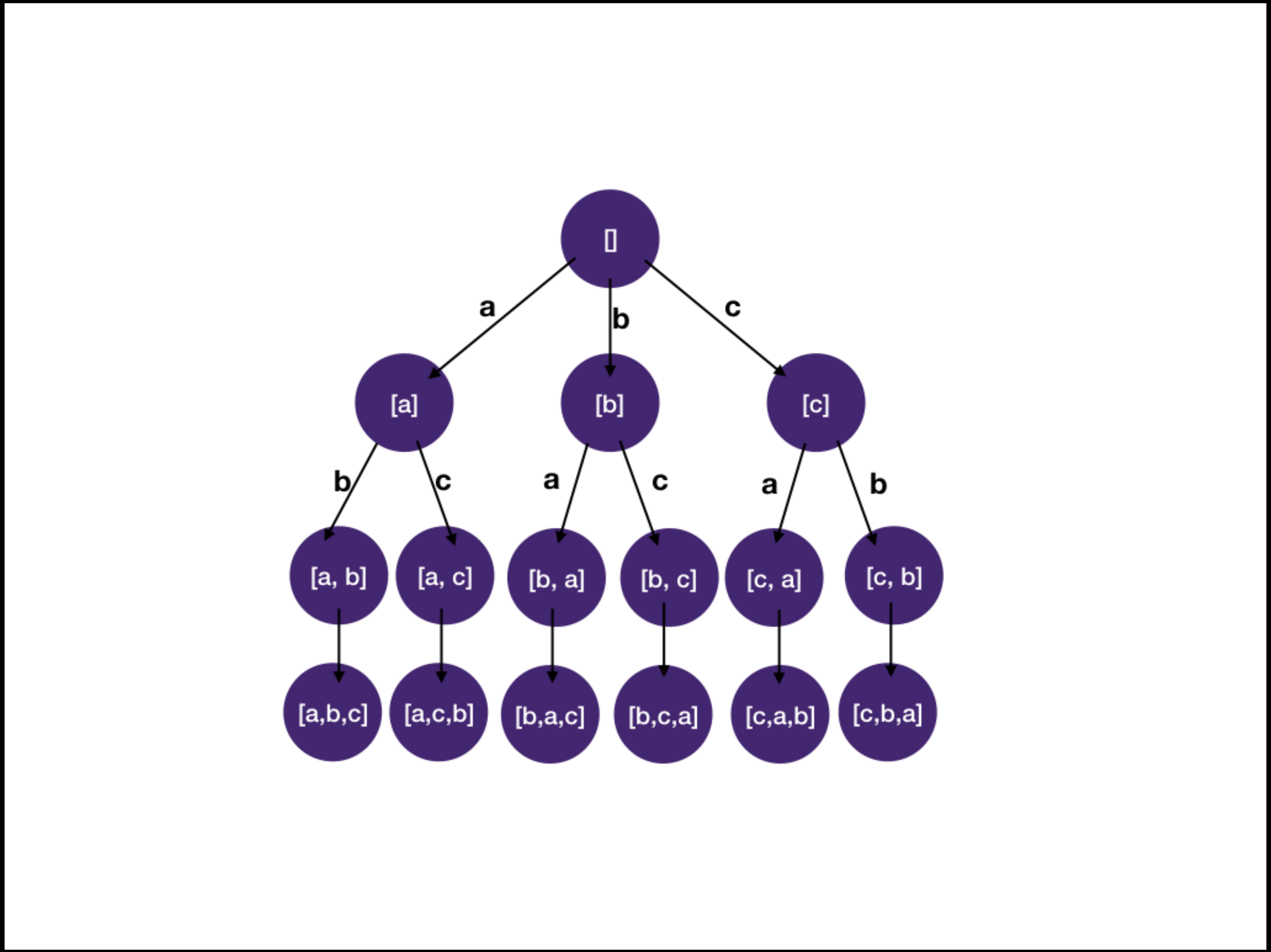
Backtracking

- Dynamic Programming is used in optimization problems. Backtracking is not for optimization problems
  - optimization problem is about minimum or maximum result (a single result). In Backtracking we use brute force approach, not for optimization problem. it is for when you have multiple results and you want all or some of them
- Generally Backtracking is used when multiple solutions exists and all solutions are required
- E.g A,B,C how many different ways can the three be sequenced. 3!
- Solution space can be represented as a state space tree.
- Backtracking will generally will have constraints
- Backtracking is a specific type of recursion , but not all recursive algorithms are backtracking algorithms.
  - While recursion refers to a function calling itself, backtracking involves using recursion to explore all possible options for a problem, undoing previous steps (backtracking) when a solution is not found, and trying different paths.

Backtracking

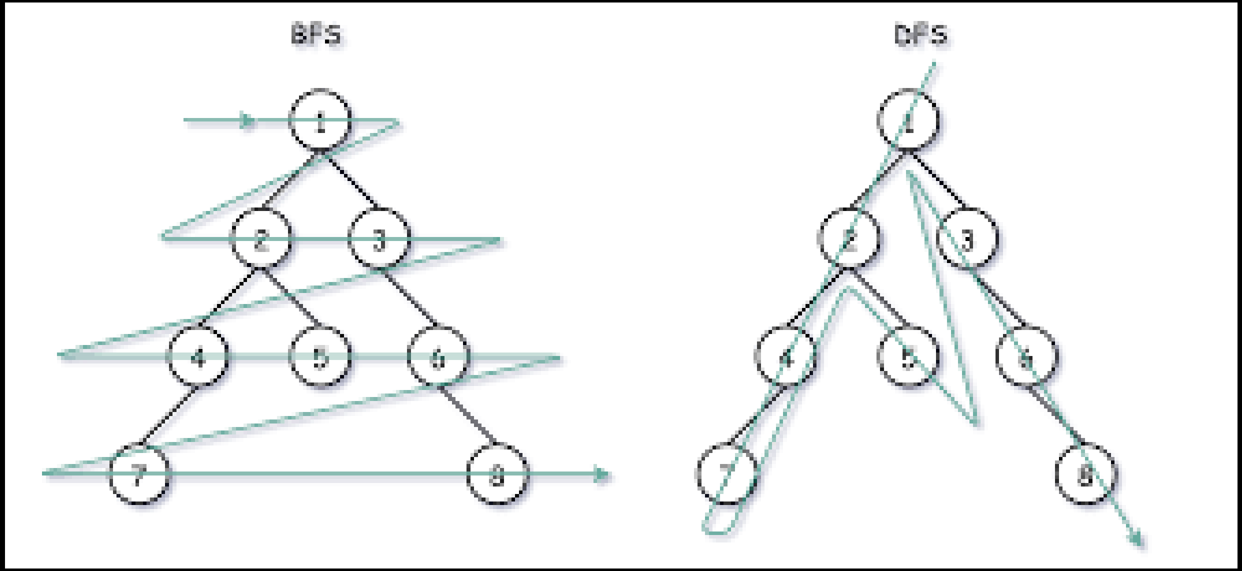
- Backtracking is a more specialized algorithmic paradigm that utilizes recursion to systematically search for solutions by building a candidate solution step-by-step.
  - When a partial solution is determined to be invalid or cannot lead to a complete solution, the algorithm "backtracks" by undoing the most recent choice and exploring alternative options.
- When to use a backtracking algorithm?
  - Can use backtracking when the problem involves exploring multiple possibilities, especially when you need to find all solutions or even the best solution under constraints.
  - It is particularly useful for problems where choices must be made in sequence, and each choice depends on previous ones.
- Backtracking can be used to find all possible solutions to a problem by continuing to explore all paths even after a solution is found, ensuring that no potential solutions are missed.

Sequencing three letters a,b,c



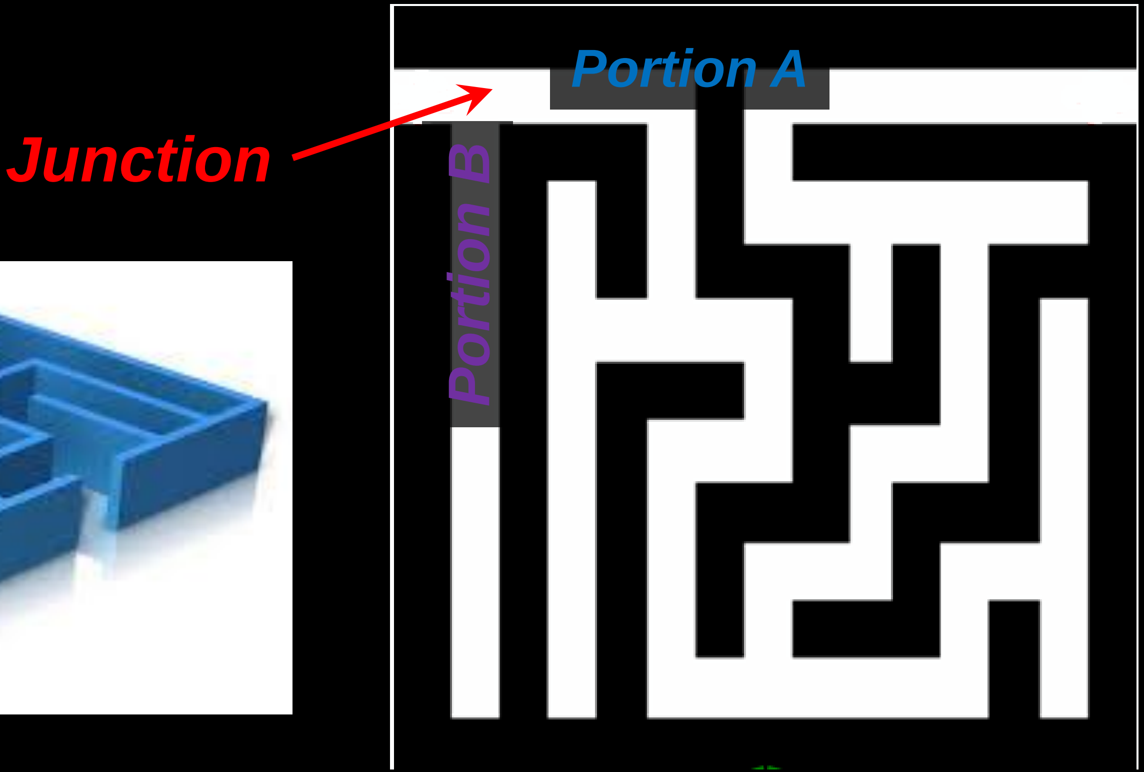
Backtracking

- Backtracking is typically implemented using a Depth-First Search (DFS) approach to generate the tree.
- It involves exploring possible solutions by diving deep into each branch of the solution space and backtracking when a solution path is not feasible.



Backtracking: Idea

- Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.
- A standard example of backtracking would be going through a maze.
    - At some point, you might have two options of which direction to go:



https://courses.cs.washington.edu/courses/cse373/15su/lectures/lecture22.pptx

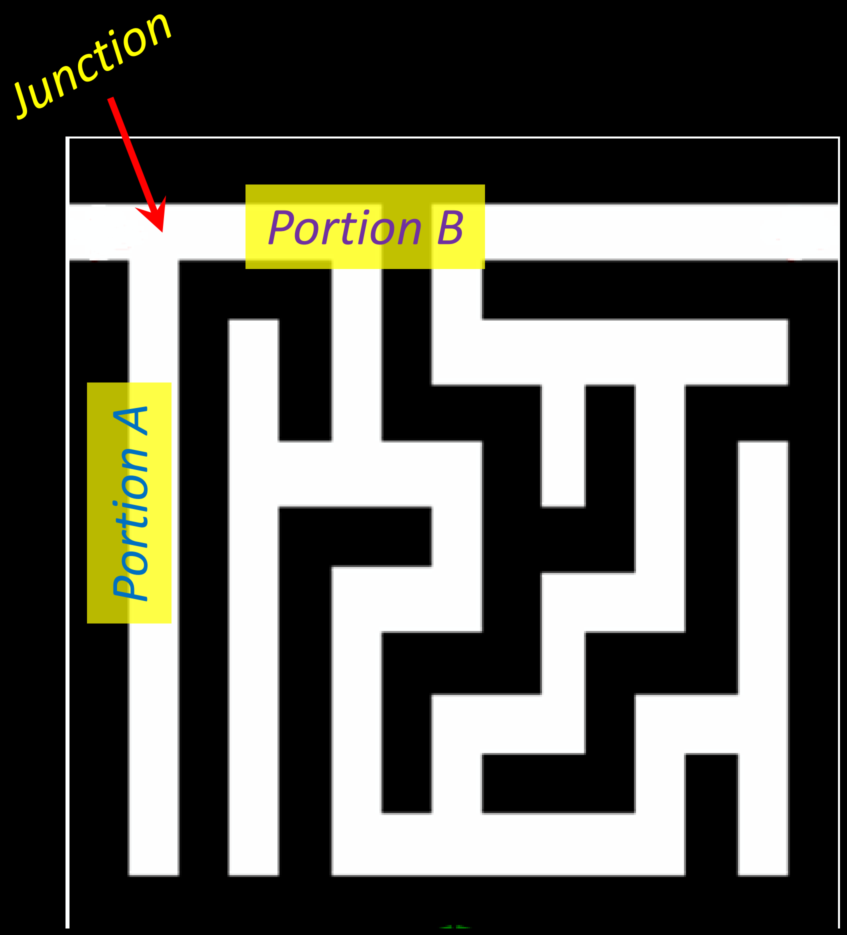
Backtracking : Maze Problem

One strategy would be to try going through Portion A of the maze.

If you get stuck before you find your way out, then you "*backtrack*" to the junction.

At this point in time you know that Portion A will *NOT* lead you out of the maze,

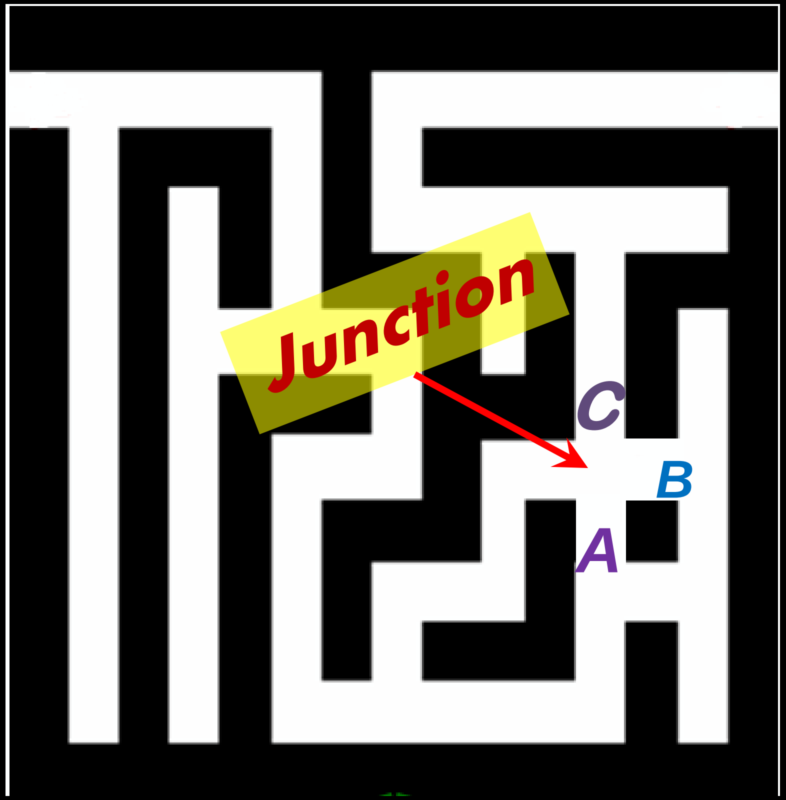
so you then start searching in Portion B



https://courses.cs.washington.edu/courses/cse373/15su/lectures/lecture22.pptx

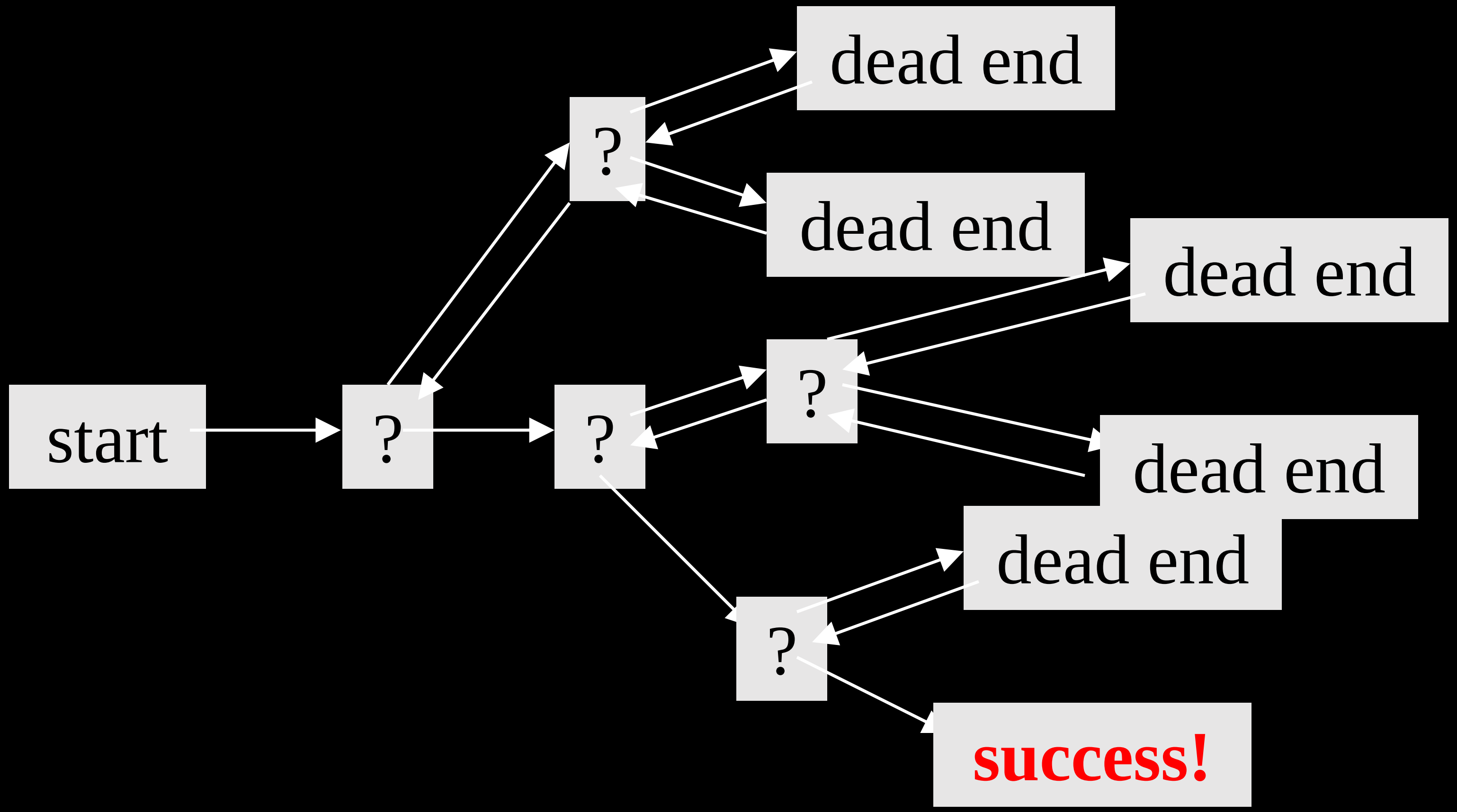
Backtracking

- Clearly, at a single junction you could have even more than 2 choices.
- 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.



https://courses.cs.washington.edu/courses/cse373/15su/lectures/lecture22.pptx

Backtracking (animation)



What is Backtracking?

- Backtracking is nothing but the modified process of the Brute force approach.
- It is a technique where multiple solutions to a problem are available, and it searches for the solution to the problem among all the available options.

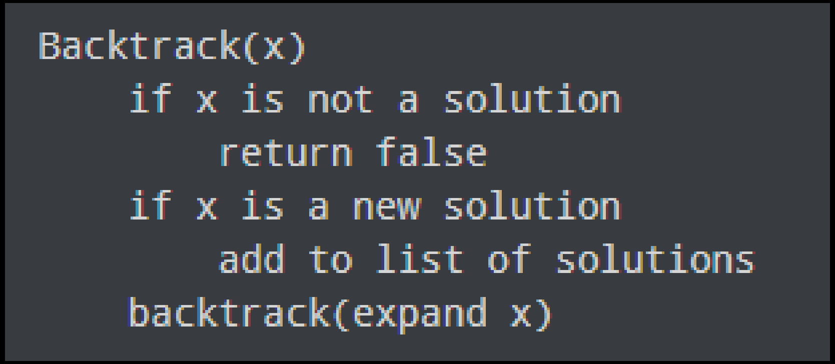
Backtracking

- Backtracking involves:
  1. Making a choice: At each step, a decision is made to extend the current partial solution.
  2. Recursive call: The function recursively calls itself with the updated partial solution.
  3. Checking constraints: After each choice, the algorithm checks if the current partial solution violates any constraints.
  4. Backtracking (undoing): If a choice leads to a dead end or violates constraints, the algorithm "backtracks" by undoing the last choice and trying a different one. This typically involves reversing the changes made by the previous recursive call.

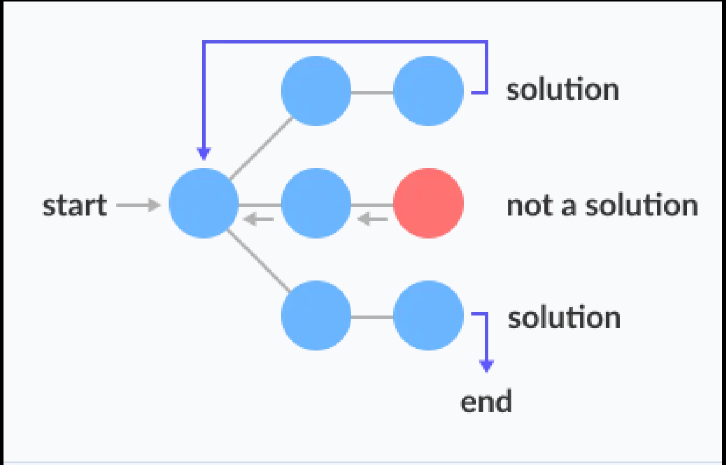
Therefore, while backtracking inherently relies on recursion to explore the solution space, the key distinguishing feature is the explicit "undoing" or "reverting" of choices when a path proves unfruitful, allowing the algorithm to explore other possibilities.

State Space Tree

- In backtracking, solutions are represented in the form of a tree and that tree is known as a 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.
- Since backtracking follows the DFS, the tree will be formed using DFS, which is known as a State Space tree.



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



State Space Tree

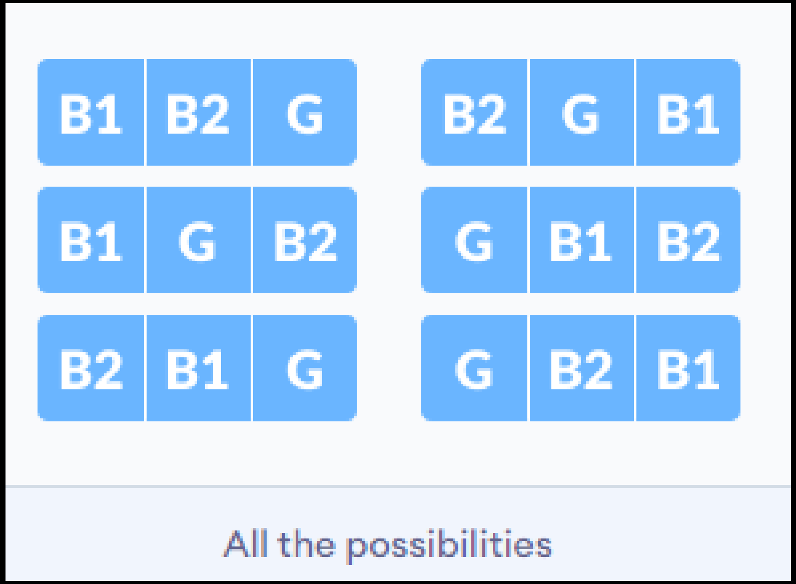
https://www.programiz.com/dsa/backtracking-algorithm

Example Backtracking Approach

- Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.
- Constraint: Girl should not be on the middle bench.
- Solution: There are a total of <u>3! = 6 possibilities</u>. We will try all the possibilities and get the possible solutions. <u>We recursively try all the possibilities.</u>

Example Backtracking Approach

- All the possibilities are:

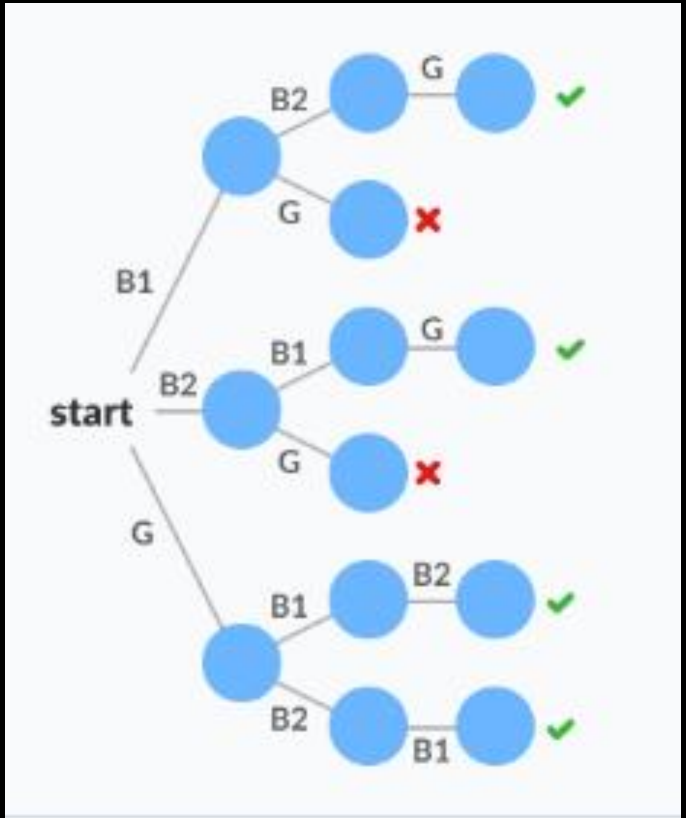


B1	B2	G	B2	G	B1
B1	G	B2	G	B1	B2
B2	B1	G	G	B2	B1

All the possibilities

- All the possibilities are:

The following **state space tree** shows the possible solutions



State tree with all the solutions

https://www.programiz.com/dsa/backtracking-algorithm

Backtracking

- Backtracking is a powerful approach in computer science used to solve problems by trying to build a solution incrementally, one part at a time, & removing those solutions that fail to satisfy the constraints of the problem at any point of time.
  - Imagine trying to solve a puzzle: if one piece does not fit, you remove it & try another piece instead.
- The following process shows how generally backtracking works:
  1. It starts with a possible move & tries to solve the problem.
  2. If the move leads to a solution, the algorithm stops (or can continue to search further solutions).
  3. However, if the move does NOT solve the problem, the algorithm undoes the move (backtracks) & makes a new try with different options.
- This process repeats until a solution is found or all options are exhausted.

https://www.naukri.com/code360/library/difference-between-backtracking-and-branch-and-bound

Backtracking

- Dealing with the maze:
  1. From your start point, you will iterate through each possible starting move.
  2. From there, you recursively move forward.
  3. If you ever get stuck, the recursion takes you back to where you were, and you try the next possible move.
- Make sure you do not try too many possibilities,
  - Mark which locations in the maze have been visited already so that no location in the maze gets visited twice.
  - If a place has already been visited, there is no point in trying to reach the end of the maze from there again.

Backtracking

Backtracking can be done recursively, without having to do all the bookkeeping at once.

- Instead, the stack of recursive calls does most of the bookkeeping
- (i.e., keeps track of which locations we've tried so far.)

- [Rat in a Maze | Backtracking using Stack](https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-using-stack/)
  - https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-using-stack/

https://i0.wp.com/thecleverprogrammer.com/wp-content/uploads/2020/12/rat-in-a-maze.jpg?resize=698%2C349&ssl=1

https://courses.cs.washington.edu/courses/cse373/15su/lectures/lecture22.pptx

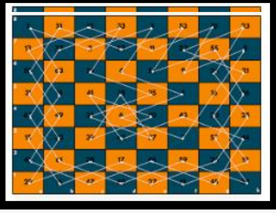
Backtracking Applications

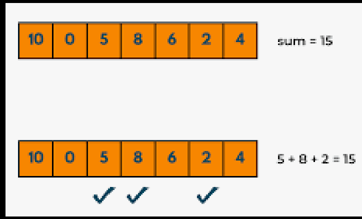
- Backtracking is widely used in several domains including artificial intelligence and operations research. Here are a few examples of applications:

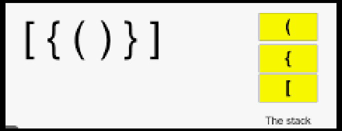
1. Logic games: Solving logic games, such as Sudoku, is a classic example of backtracking.
2. Pathfinding: In pathfinding problems, such as finding optimal paths in a graph, backtracking can be used to explore different path options.
3. Chess and other board games: In board games such as chess, backtracking can be used to evaluate possible sequences of actions and choose the best strategy.

Applications of Backtracking Algorithm

- Puzzle Solving: Used in solving puzzles like Sudoku, crosswords, and the N-Queens problem.
- Combinatorial Optimization: Generates all permutations, combinations, and subsets of a given set.
- Constraint Satisfaction Problems: Applied in problems like graph coloring, scheduling, and job assignment where constraints must be met.
- Pathfinding: Solves mazes and other pathfinding problems, such as the Rat in a Maze problem.
- Game Solving: Used in strategy games to explore possible moves, such as in the Knight's Tour problem.
- Decision-Making: Helps in making optimal decisions in situations where multiple choices are possible, like in the Subset Sum problem.
- String Processing: Generates valid strings that meet certain criteria, such as generating balanced parentheses.
- Optimization Problems: Finds solutions to complex optimization problems, such as maximizing profits or minimizing costs under certain constraints.







Back Tracking Algorithm

- Backtracking follows a recursive approach to solving a problem. Here are the different stages in the execution of this algorithm:
  - Step 1 – Choice: The algorithm begins by making a choice among all possible options. This choice is based on the current state of the problem.
  - Step 2 – Validation: After making a choice, the algorithm checks that the problem's constraints are still respected. If this is the case, it returns to step 1 to continue the in-depth path. If not, it backtracks to try another alternative. This is one of the differences with basic brute-force solving algorithms: backtracking can eliminate branches without having to explore them all the way.
  - Step 3 – Backtracking: If at any point the algorithm realizes that none of the choices in a branch will lead to a solution, or that it has explored all possible options without arriving at a valid final state, it goes back to explore other alternatives. This involves going back up the decision tree to explore other, as yet unexplored, branches.
  - Step 4 – Solution: The algorithm continues to traverse the tree branch and make choices, repeating steps 1 to 3 until it reaches a valid solution. A solution is generally defined as a final state of the problem that satisfies the constraints or objectives set.

https://datascientest.com/en/backtracking-what-is-it-how-do-i-use-it

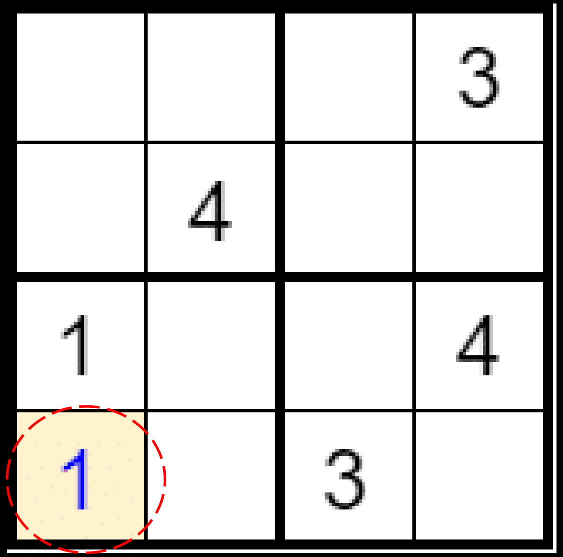
Example of backtracking application

- To illustrate the concept of backtracking, let us take a concrete example: solving a sudoku grid.
- In a sudoku game, the aim is to fill in a 9×9 grid in such a way that each row, column and 3×3 sub-grid contains all the numbers from 1 to 9 without repetition.
- Here we wll use a 4×4 grid to simplify the illustration.

https://datascientest.com/en/backtracking-what-is-it-how-do-i-use-it

Example of backtracking application: Sudoku

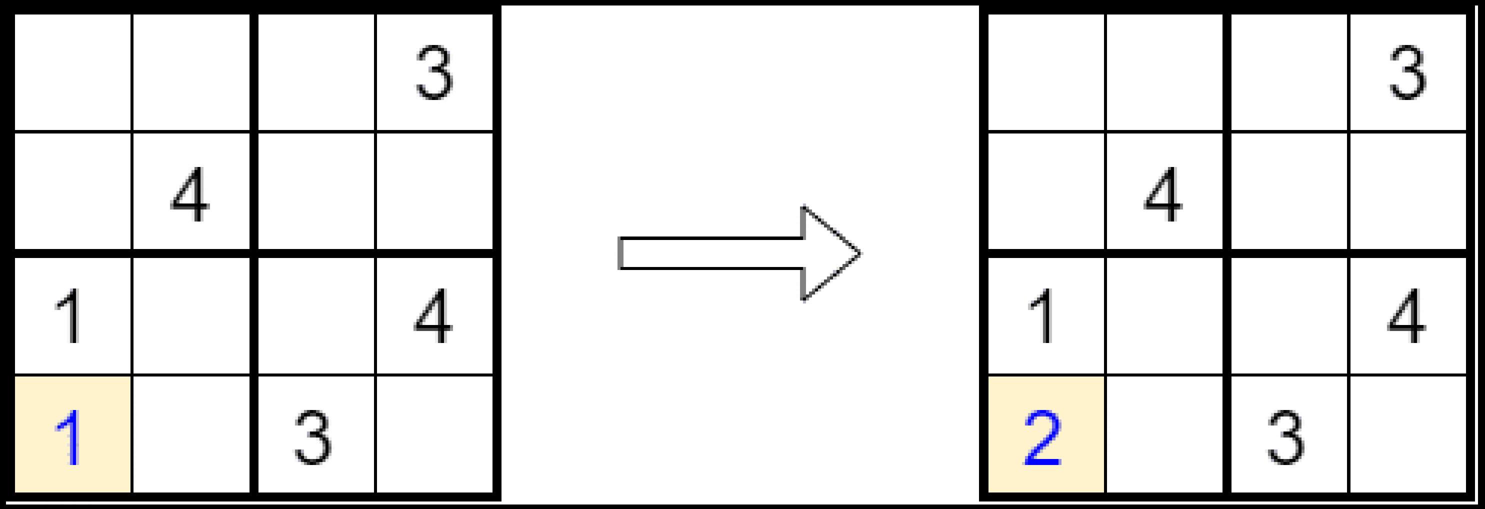
- Backtracking can be used to solve a Sudoku as follows:
- Step 1 – Choice: Start by choosing an empty square in the grid. Fill this square with a possible number (from 1 to 4).



https://datascientest.com/en/backtracking-what-is-it-how-do-i-use-it

Example of backtracking application: Sudoku

- Step 2 – Validation: Check whether the grid is still valid after making this choice. If so, return to step 1. If not, go back and try another number in the box.

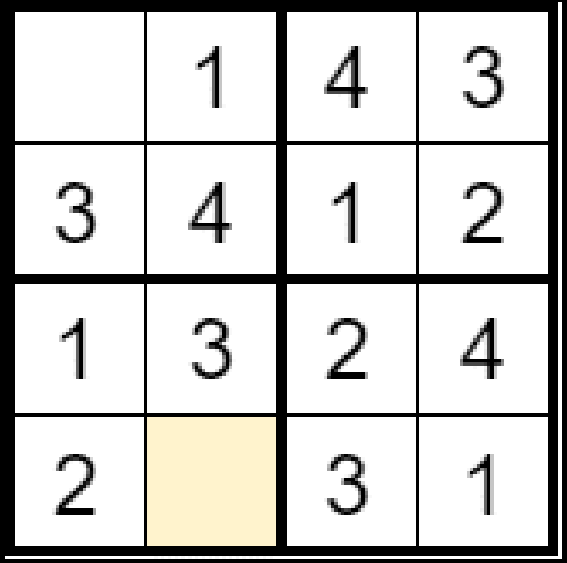


Here, the noted 1 doesn't comply with the rules, so we try a 2 instead.

https://datascientest.com/en/backtracking-what-is-it-how-do-i-use-it

Example of backtracking application: Sudoku

- Step 3 – Backtracking: If at any point you can not find a valid number for a square, go back and try another value in the previous square, and so on.



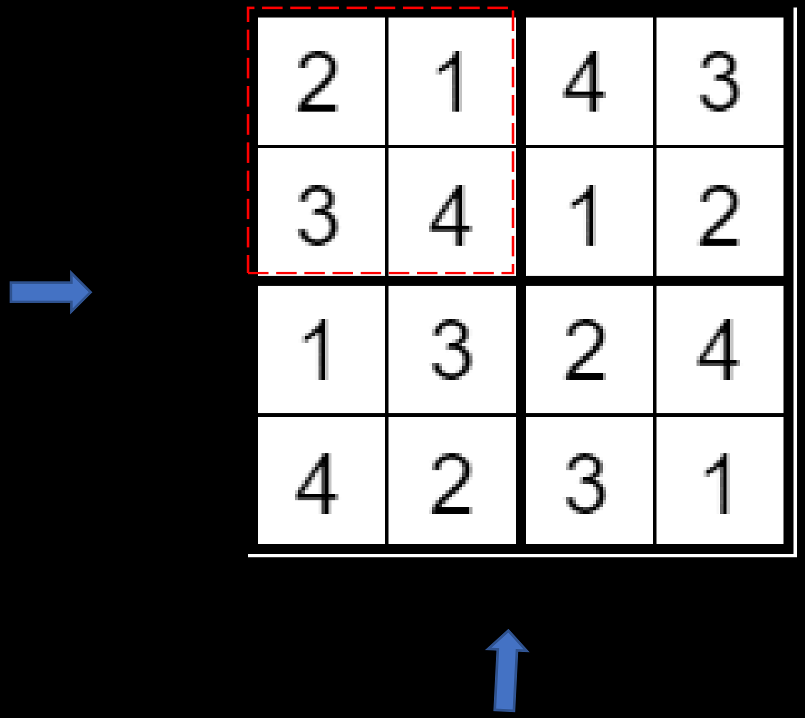
	1	4	3
3	4	1	2
1	3	2	4
2		3	1

In this case, you can not place any numbers within the rules, so you have to change the number just before it, and those before it if necessary.

https://datascientest.com/en/backtracking-what-is-it-how-do-i-use-it

Example of backtracking application: Sudoku

- Step 4 – Solution: Repeat steps 1 to 3 until all squares are filled in and meet the game’s constraints, which means you’ve found a valid sudoku solution.



2	1	4	3
3	4	1	2
1	3	2	4
4	2	3	1

https://datascientest.com/en/backtracking-what-is-it-how-do-i-use-it

Advantages of backtracking

- Backtracking offers several advantages as a problem-solving approach:
  1. Completeness: It ensures that all possible valid solutions are explored, making it useful for the search for optimal solutions.
  2. Efficiency: Backtracking can be very effective, especially in problems with strong constraints, enabling certain options to be quickly eliminated.
  3. Adaptability: It can be applied to a variety of problems, as it is based on a general tree structure.

https://datascientest.com/en/backtracking-what-is-it-how-do-i-use-it

Limitations of backtracking

Backtracking has certain limitations:

1. Combinatorial explosion: In some problems, the number of combinations to be explored can increase exponentially, as in the case of the travelling salesman problem, where if a path is calculated in 1 microsecond, it takes almost two millennia to calculate all the paths passing through 20 points.
2. Complexity: The implementation of backtracking can be complex, especially in problems where the definition of choices, constraints and stopping criteria is complicated.
3. No guarantee of convergence: Backtracking may not converge to a solution in some cases, for example when the solution does not exist, or get stuck in infinite loops if the problem is poorly defined.

https://datascientest.com/en/backtracking-what-is-it-how-do-i-use-it

Optimizing backtracking

- There are several techniques for optimizing the backtracking process, including :

1. Heuristics: The use of intelligent rules and heuristics to guide the search to the most promising branches of the decision tree.
2. Pruning: The identification and early elimination of certain branches of the tree that can not lead to a solution, thus reducing combinatorial explosion.
3. Memory: Caching of explored states to avoid recalculating the same configurations several times when they can be obtained in several different ways.

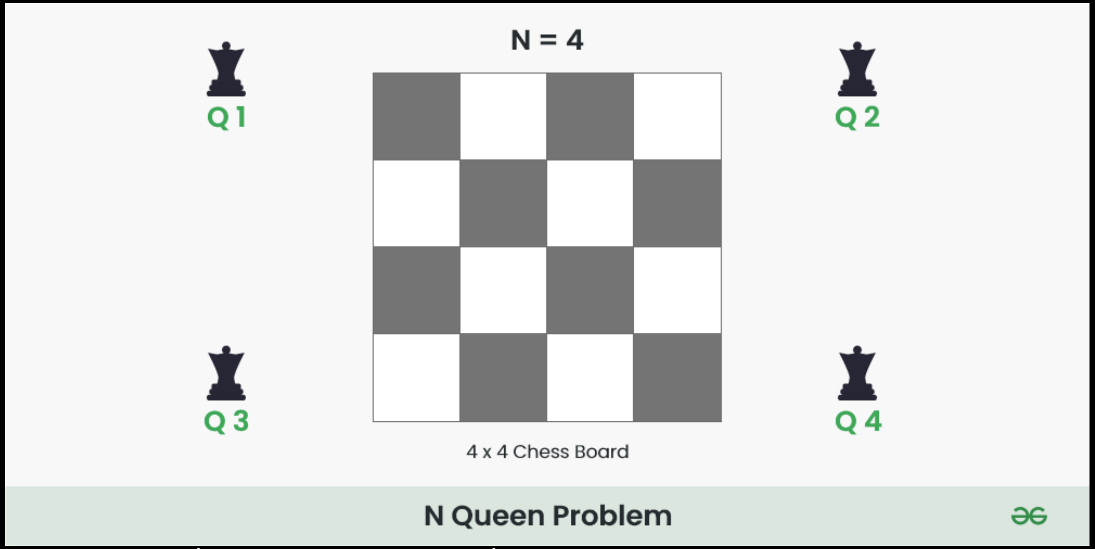
https://datascientest.com/en/backtracking-what-is-it-how-do-i-use-it

N queen problem using Backtracking

- The backtracking algorithm can be used to solve the N queen problem, in which the board is recursively filled with queens, one at a time, and backtracked if no valid solution is found.
- At each step, the algorithm checks to see if the newly placed queen conflicts with any previously placed queens, and if so, it backtracks
  - i.e If a queen is under attack at all the positions in a row, we backtrack and change the position of the queen placed prior to the current position.
  - We repeat this process of placing a queen and backtracking until all the N queens are placed successfully

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.
- That is, no two queens are allowed to be placed on the same row, the same column or the same diagonal.



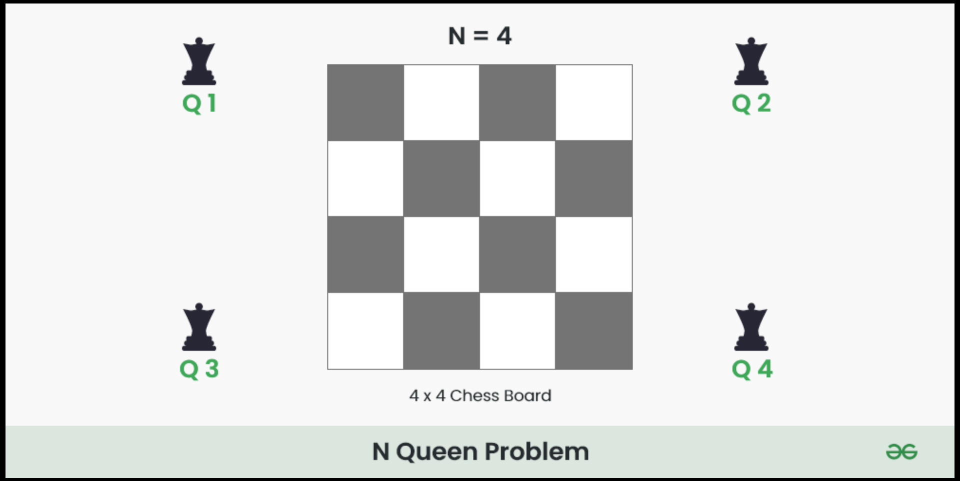
https://www.geeksforgeeks.org/4-queens-problem/

4 Queens Problem using Backtracking Algorithm:

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

N Queens Problem

- **Time Complexity:** O(N!)
  Auxiliary Space: O(N)



One Possible Solution

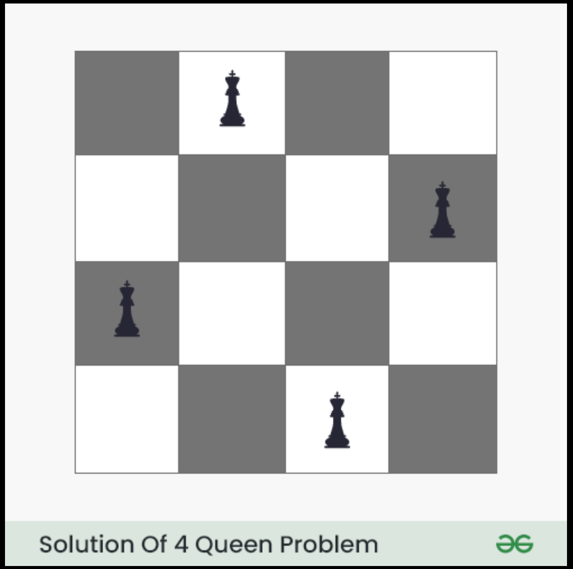


Illustration of 4 Queens Solution:

- **Step 0:** Initialize a 4×4 board.

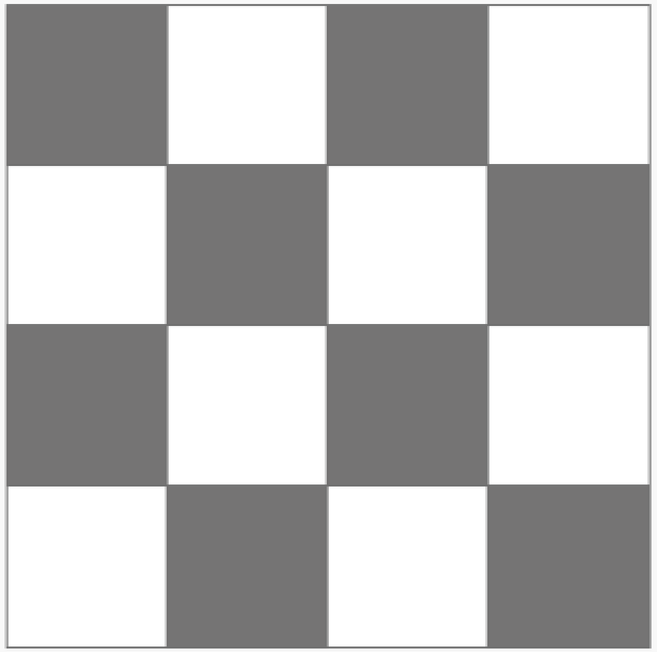
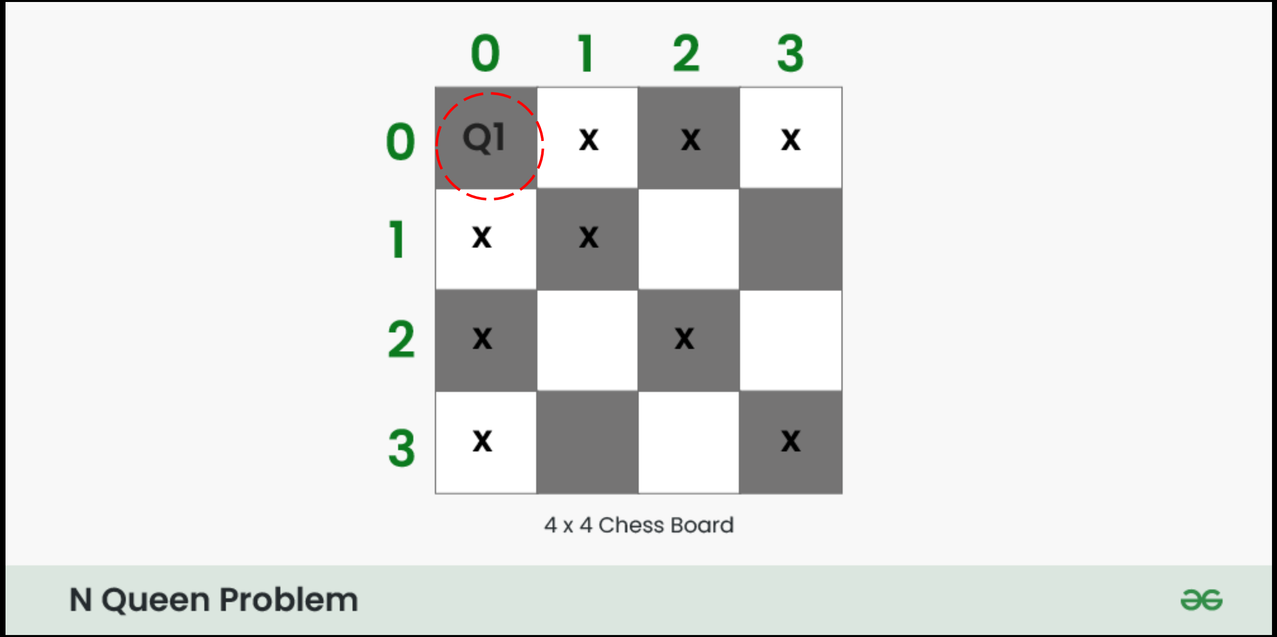

4 x 4 Chess Board

Illustration of 4 Queens Solution:

- 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].



https://www.geeksforgeeks.org/4-queens-problem/

Illustration of 4 Queens Solution:

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

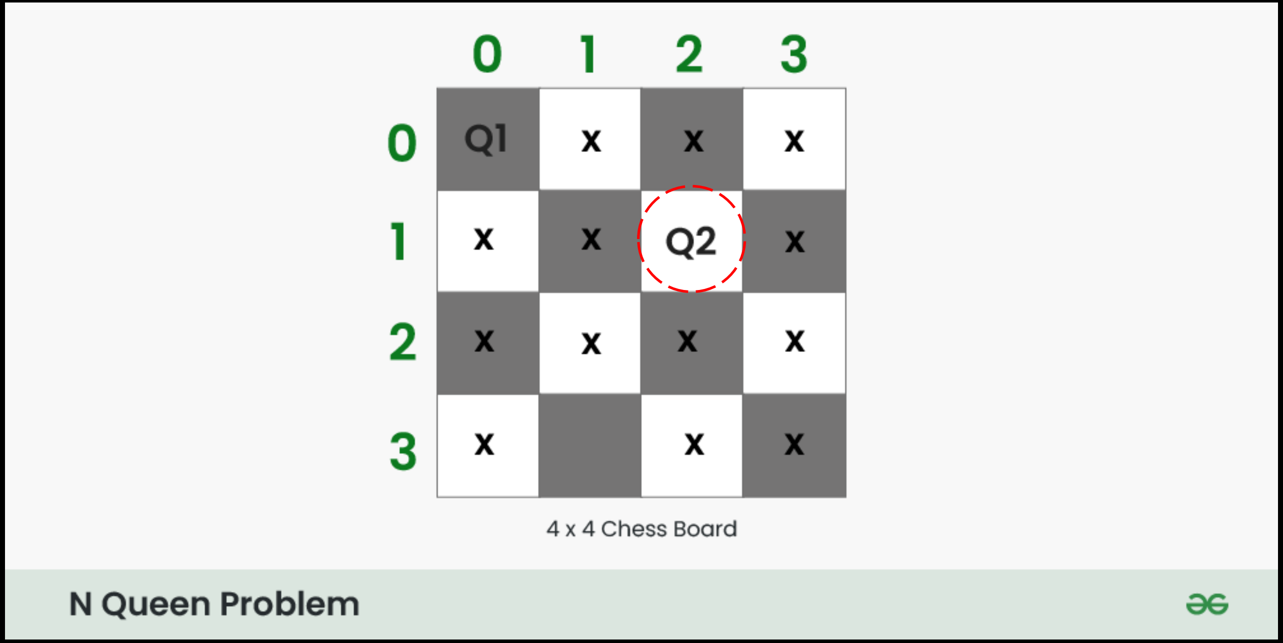


Illustration of 4 Queens Solution:

- Step 3:
- At row 2 there is no cell which are safe to place Queen (Q3) .
- So, backtrack and <u>remove queen Q2 queen from cell (1, 2)</u> .

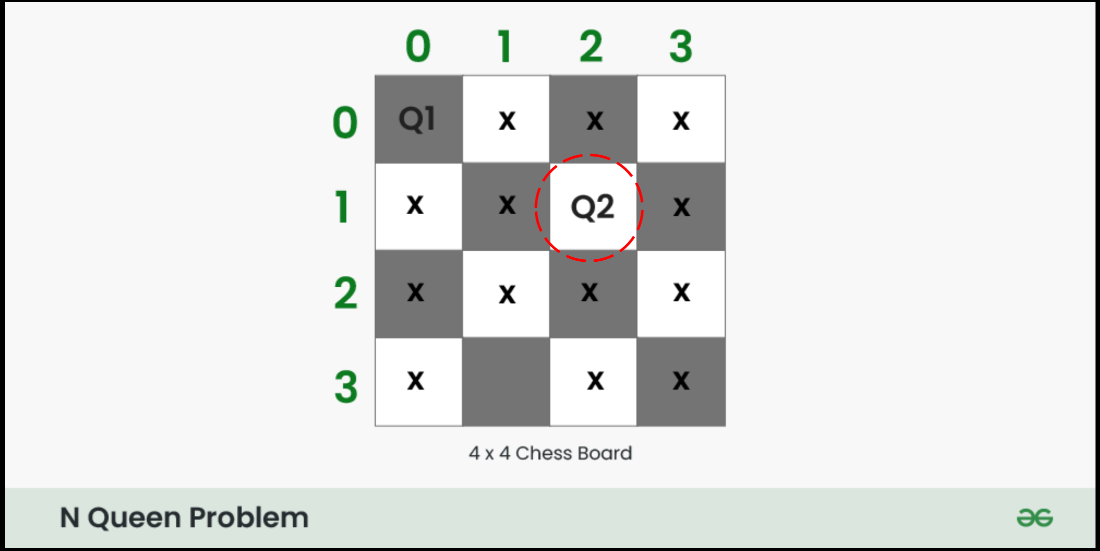


Illustration of 4 Queens Solution:

- **Step 4:**
  - There is still a safe cell in the row 1 i.e. cell (1, 3).
  - Put Queen (**Q2**) at cell (1, 3).

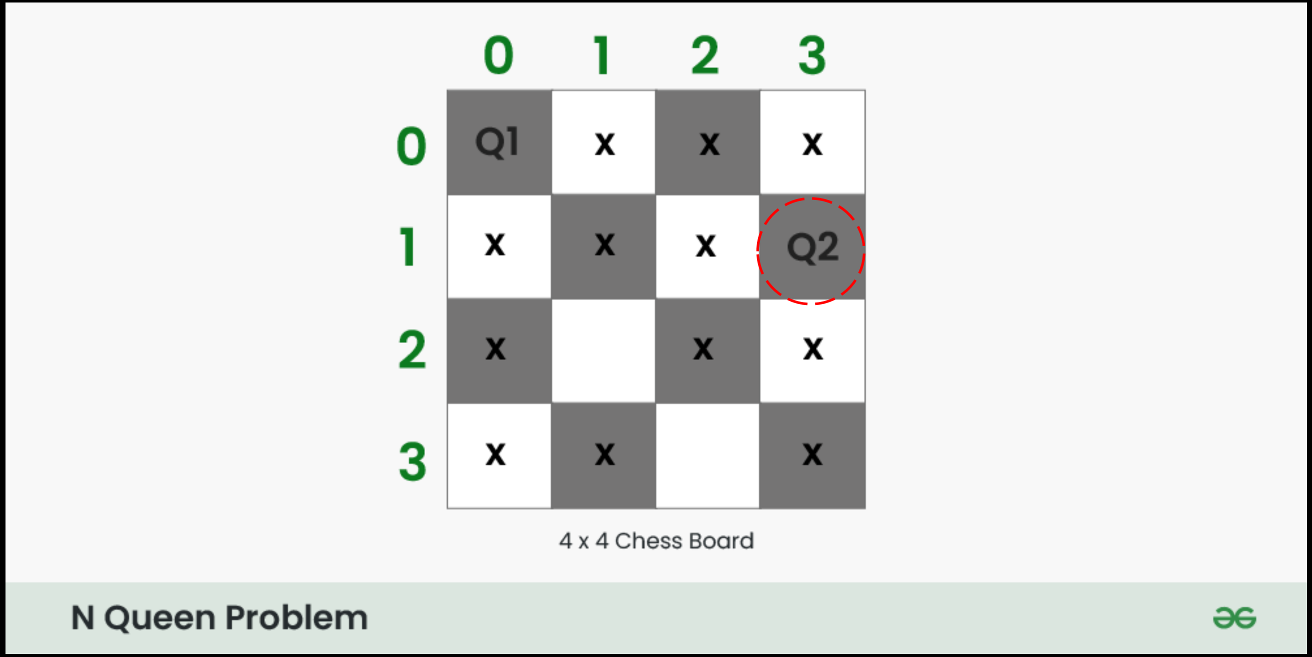
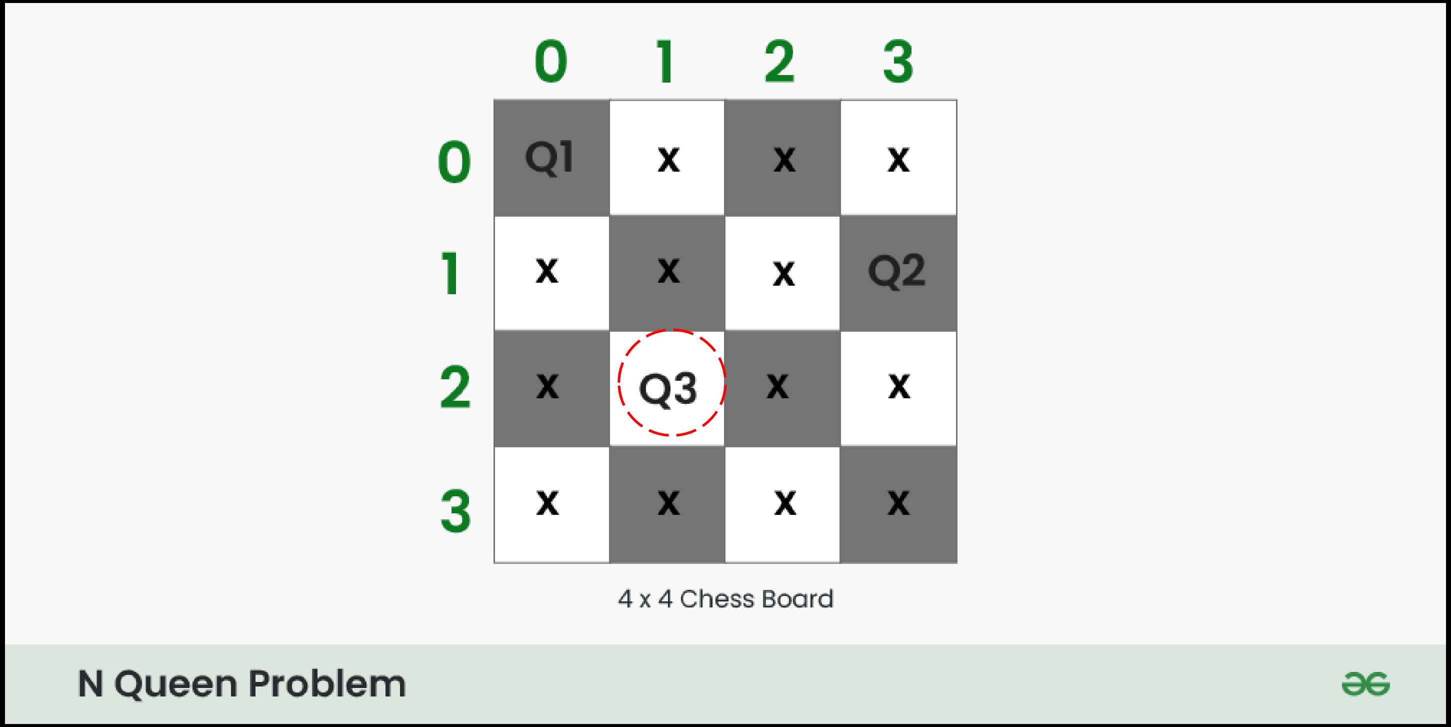


Illustration of 4 Queens Solution:

- Step 5:
  - Put queen (Q3) at cell (2, 1).



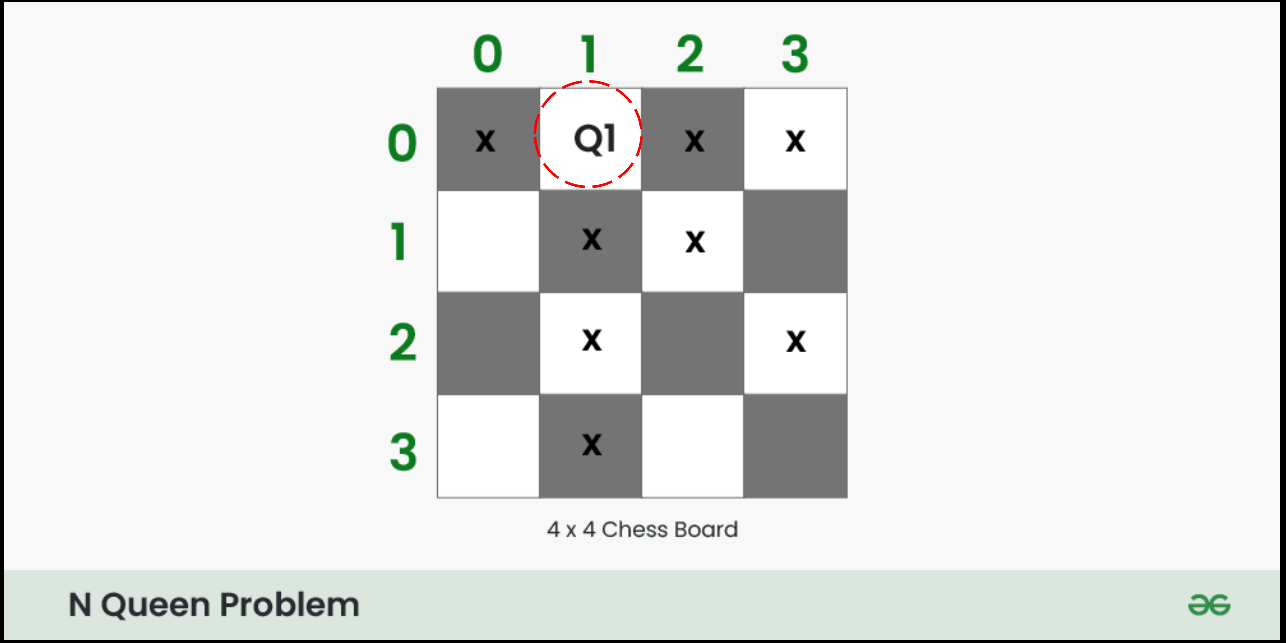
https://www.geeksforgeeks.org/4-queens-problem/

Illustration of 4 Queens Solution:

- **Step 6:**
  - There is NO 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 remove from cell **(0,0)** and move to next safe cell i.e. **(0 , 1)**.

Illustration of 4 Queens Solution:

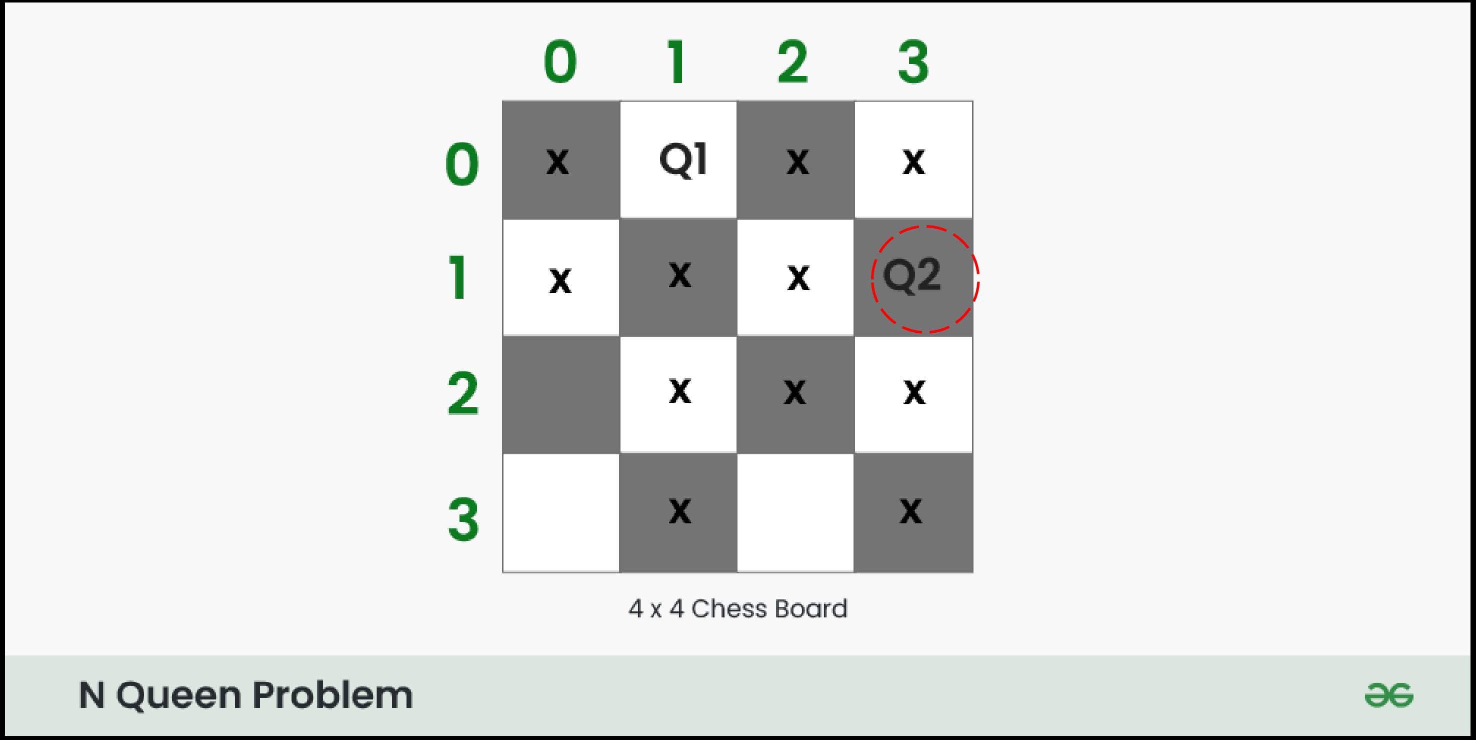
- **Step 7:**
  - Place Queen Q1 at cell (0 , 1), and move to next row.



https://www.geeksforgeeks.org/4-queens-problem/

Illustration of 4 Queens Solution:

- **Step 8:**
  - Place Queen **Q2** at cell **(1 , 3)**, and move to next row.



https://www.geeksforgeeks.org/4-queens-problem/

Illustration of 4 Queens Solution:

- **Step 9:**
  - Place Queen **Q3** at cell **(2 , 0)**, and move to next row.

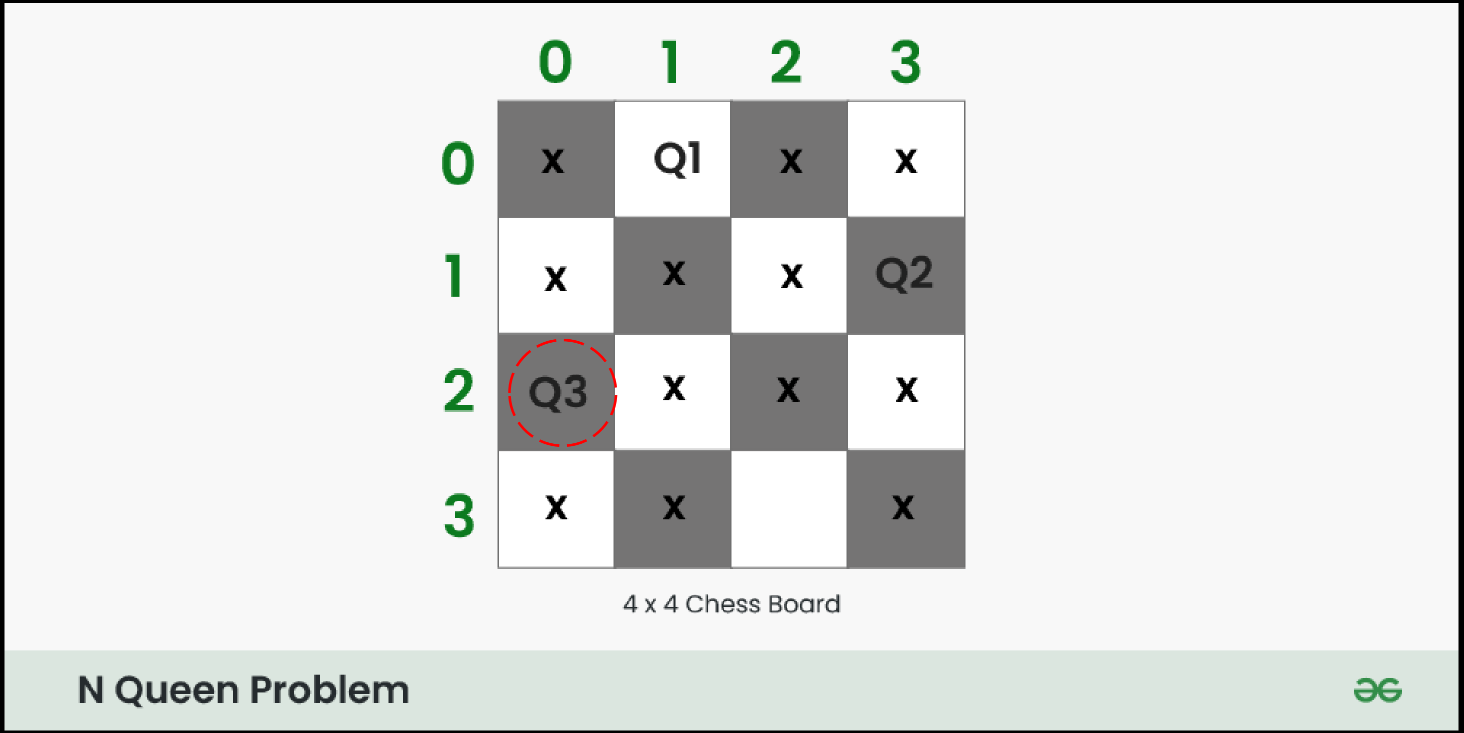
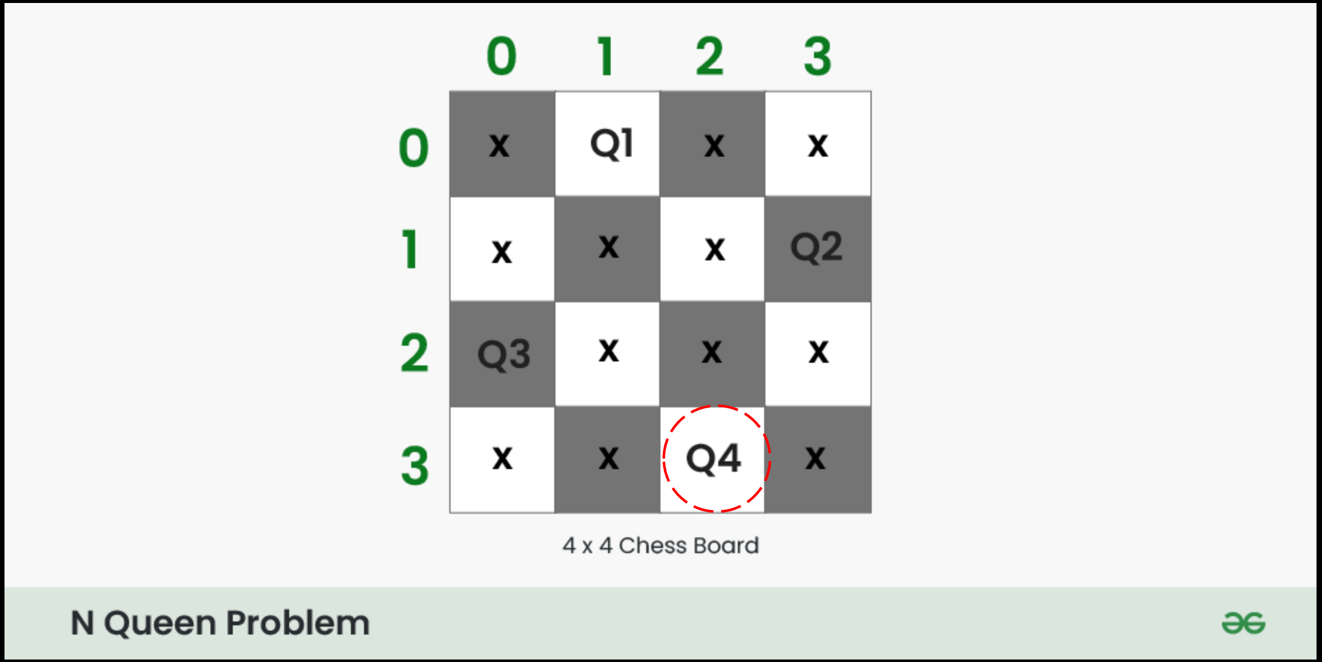


Illustration of 4 Queens Solution:

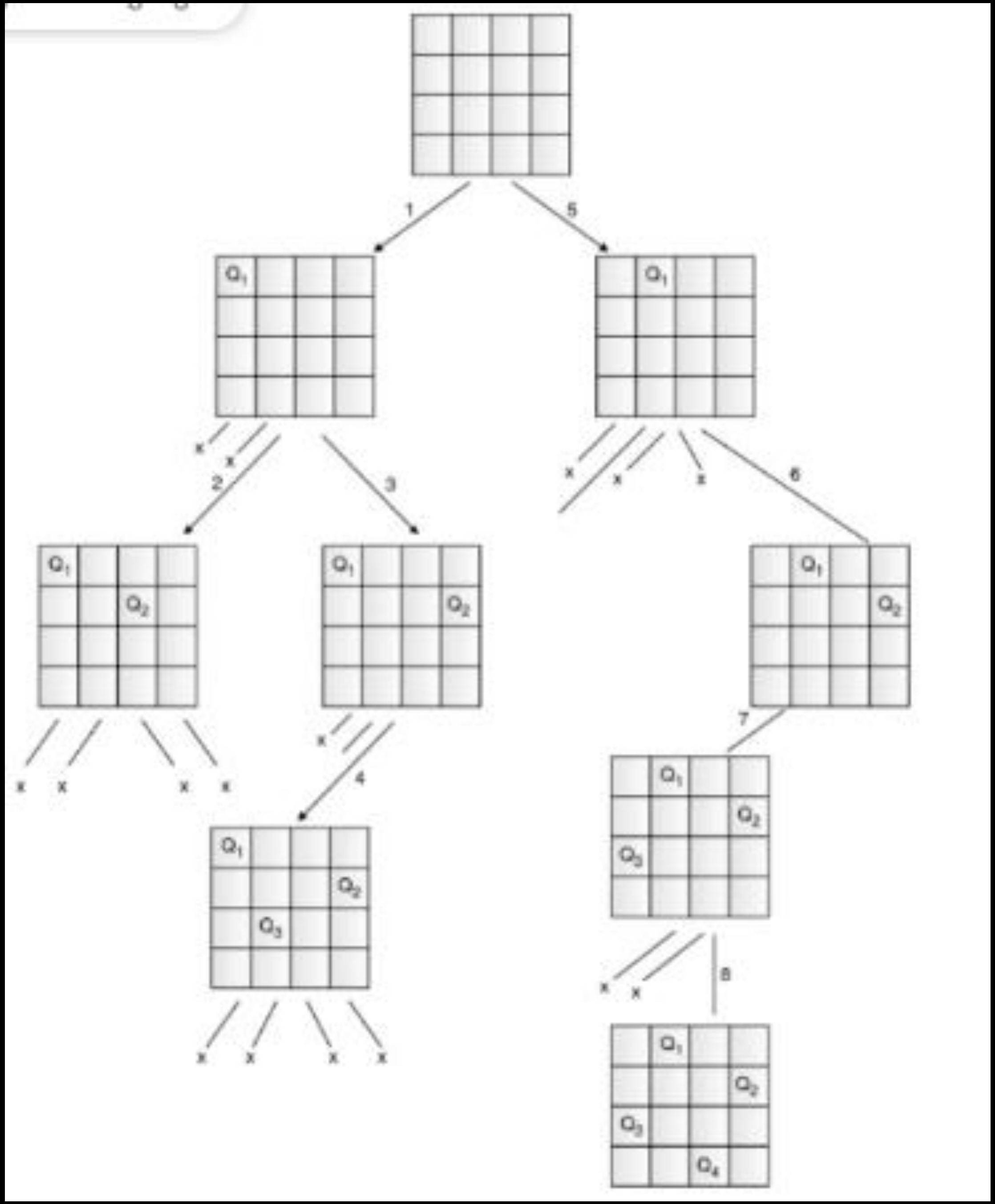
- **Step 10:**
  - Place Queen **Q4** at cell **(3 , 2)**, and move to next row.
  - This is one possible configuration of solution



https://www.geeksforgeeks.org/4-queens-problem/

4 Queens Solution

- => describes the backtracking sequence for the 4-queen problem.



4 Queens Problem Algorithm

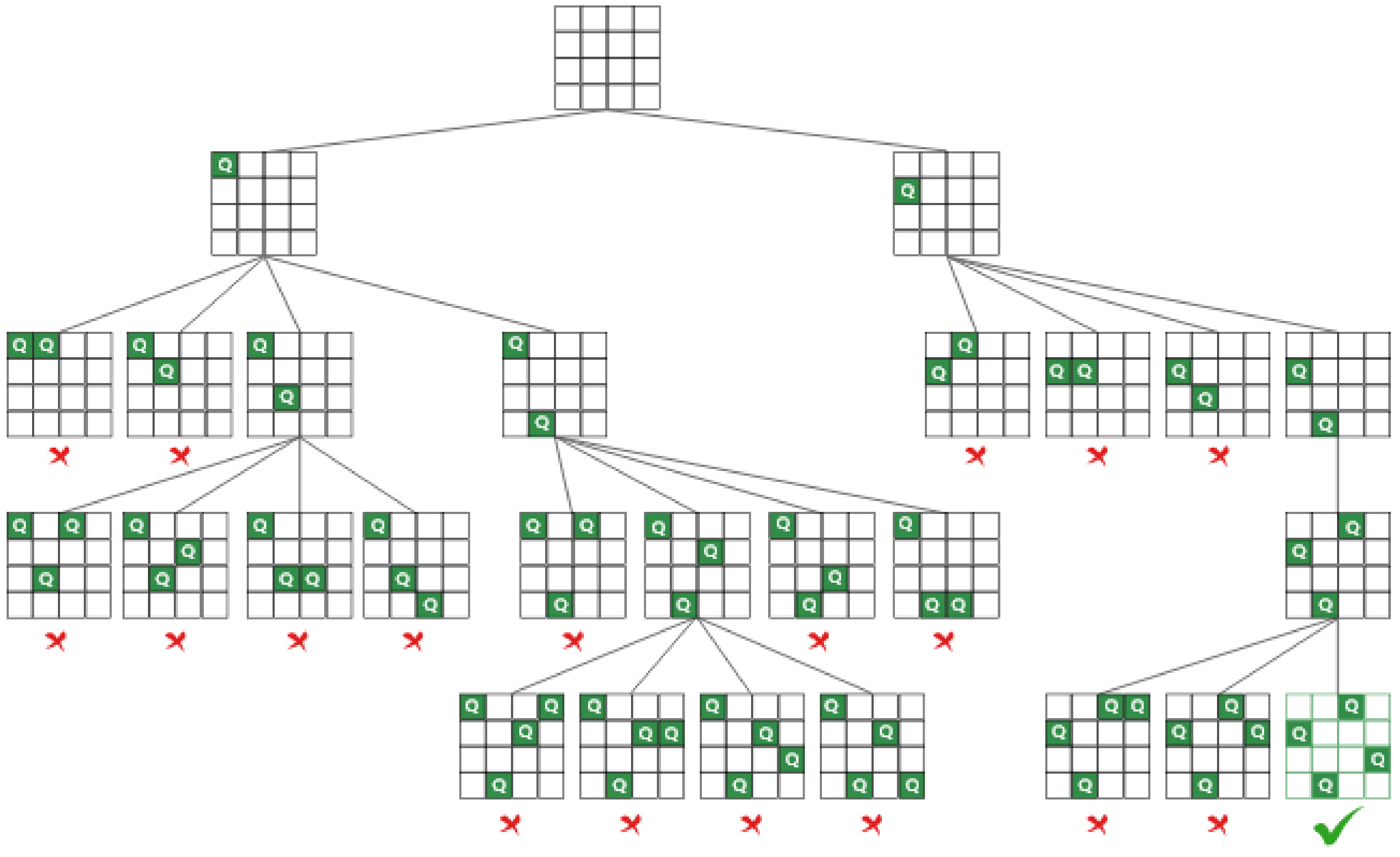
- Follow the steps mentioned below to implement the idea:
- Start in the leftmost column
- If all queens are placed return true
- Try all rows in the current column. Do the following for every row.
  1. If the queen can be placed safely in this row
     1. Then mark this **[row, column]** as part of the solution and recursively check if placing queen here leads to a solution.
     2. If placing the queen in **[row, column]** leads to a solution then return **true**.
     3. If placing queen doesn't lead to a solution then unmark this **[row, column]** then backtrack and try other rows.
  2. If all rows have been tried and valid solution is not found return **false** to trigger backtracking.

Code available at https://www.geeksforgeeks.org/4-queens-problem/

https://www.geeksforgeeks.org/4-queens-problem/

- Refer implementation of the above Backtracking solution:
    - https://www.geeksforgeeks.org/dsa/4-queens-problem/

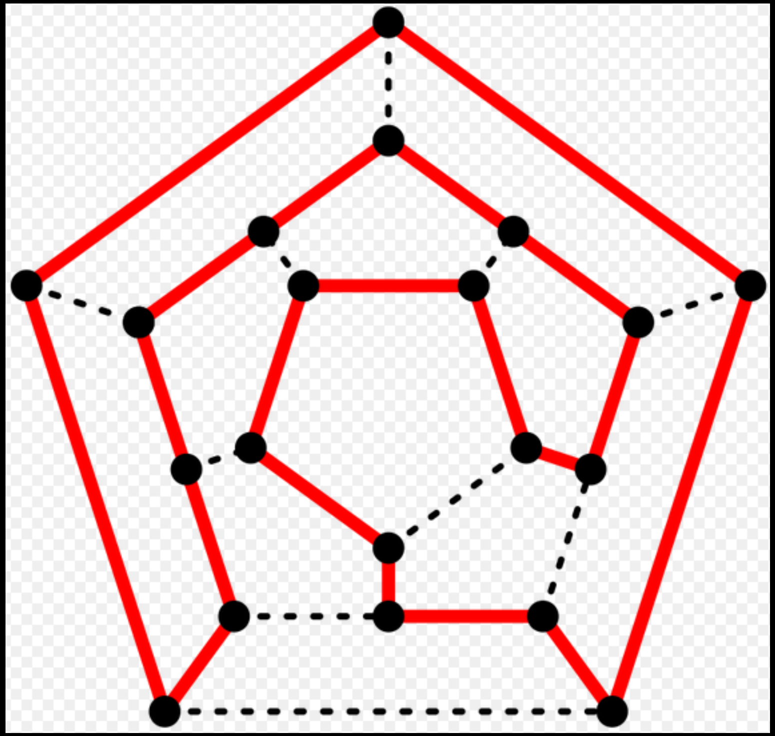
Recursive Tree of the Above Algorithm



Backtracking

Hamiltonian Path

- A **Hamiltonian path** is a path in a graph which contains each vertex of the graph exactly once. A Hamiltonian cycle is a Hamiltonian path, which is also a cycle.
- Knowing (a) whether such a path exists in a graph, as well as (b) finding the path is a fundamental problem of graph theory.
- Hamiltonian paths are applied in various sectors such as logistics for route optimisation and computer science for network and scheduling problems, illustrated by the Travelling Salesperson Problem (TSP)



Hamiltonian Cycle using Backtracking

- The **Hamiltonian cycle** is the cycle in the graph which visits all the vertices in graph exactly once and **terminates at the starting node**. It may NOT include all the edges
- The '*Hamiltonian cycle problem*' is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle.
  - The input to the problem is an undirected, connected graph.
  - For the graph shown in below, a path A – B – E – D – C – A forms a Hamiltonian cycle.
  - It visits all the vertices exactly once, but does NOT visit the edges <B, D>

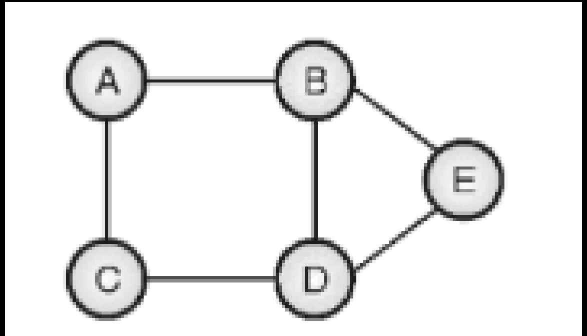


Figure (a)

Hamiltonian Cycle using Backtracking

- The Hamiltonian cycle problem is also both, (a) decision problem and an (b) optimization problem.

i. A decision problem is stated as, "Given a path, is it a Hamiltonian cycle of the graph?".

ii. The optimization problem is stated as, "Given graph G, find the Hamiltonian cycle for the graph."

- We can define the constraint for the Hamiltonian cycle problem as follows:
    1. In any path, vertex i and (i + 1) must be adjacent.
    2. 1st and (n – 1)th vertex must be adjacent (nth of cycle is the initial vertex itself).
    3. Vertex i must NOT appear in the first (i – 1) vertices of any path.

- With the adjacency matrix representation of the graph, the adjacency of two vertices can be verified in constant time

Complexity Analysis

- Looking at the state space graph, in worst case, total number of nodes in tree would be

$$T(n) = 1 + (n-1) + (n-1)^2 + (n-1)^3 + \ldots + (n-1)^{n-1}$$

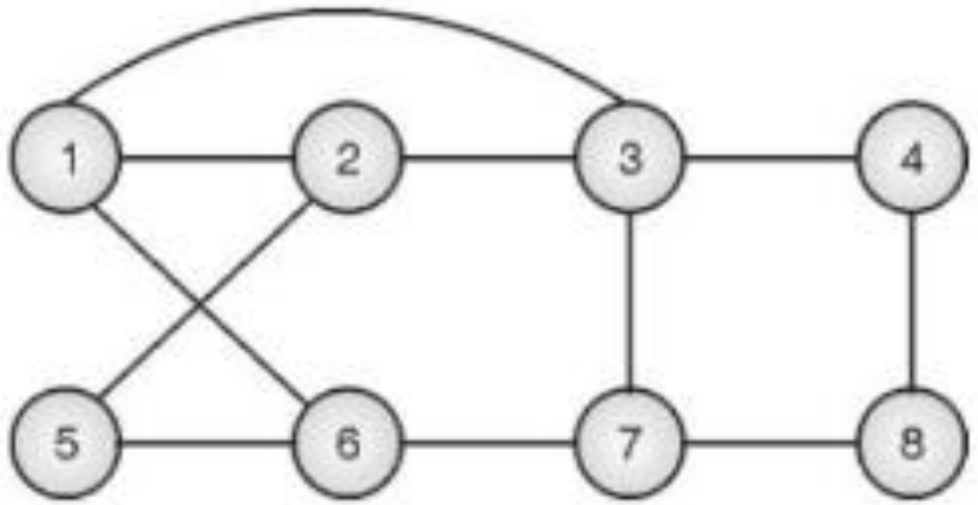
$$= \frac{(n-1)^n - 1}{n-2}$$

- $T(n) = O(n^n)$ Thus, the Hamiltonian cycle algorithm runs in exponential time.

Hamiltonian Cycle using Backtracking

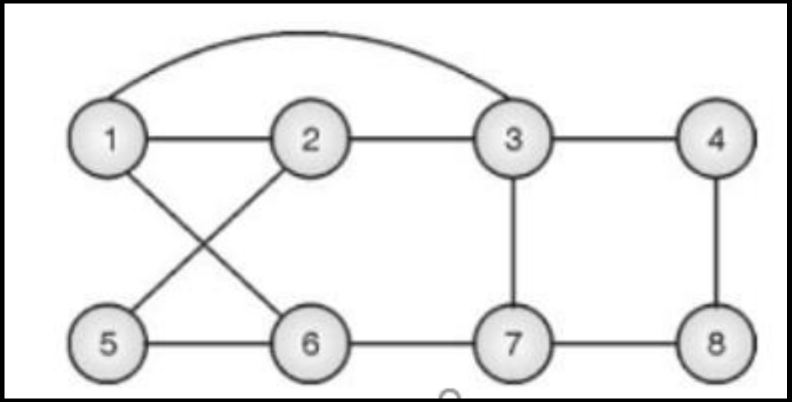
Example

Example: Explain how to find Hamiltonian Cycle by using Backtracking in a given graph



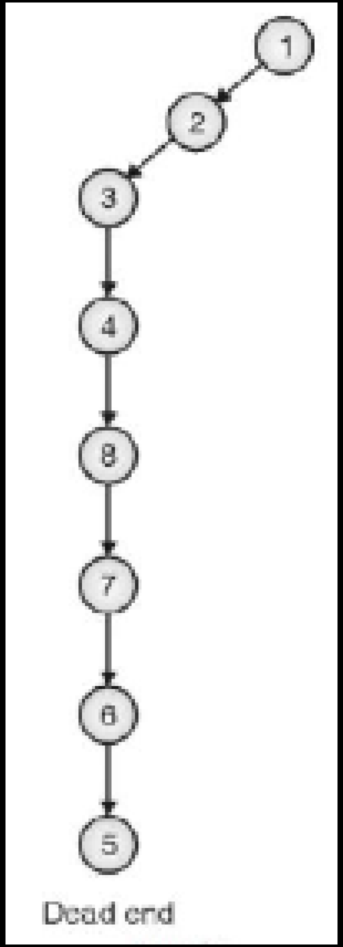
https://www.geeksforgeeks.org/hamiltonian-cycle/

Hamiltonian Cycle using Backtracking



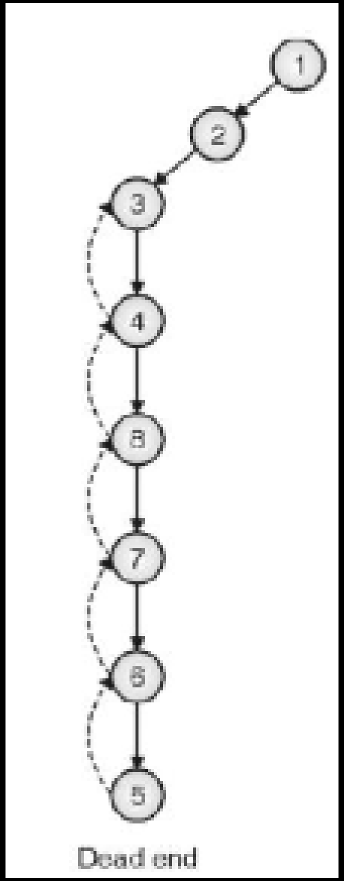
- Solution:
- The backtracking approach uses a state-space tree to check if there exists a Hamiltonian cycle in the graph. Figure shows the simulation of the Hamiltonian cycle algorithm.
    - For simplicity, we have not explored all possible paths, the concept is self-explanatory.

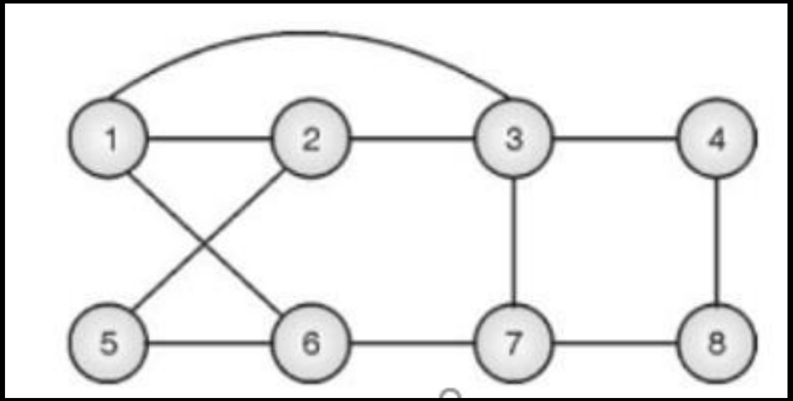
- Step 1: Tour is started from vertex 1. There is no path from 5 to 1. Therefore, it is the dead-end state.



Hamiltonian Cycle using Backtracking

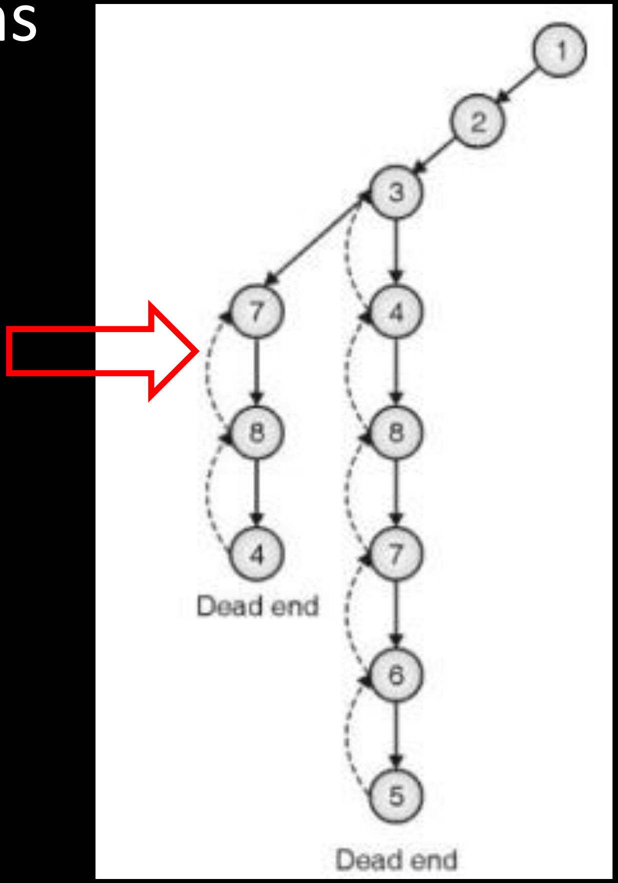
- Step 2: Backtrack to the node from where the new path can be explored, that is 3 here

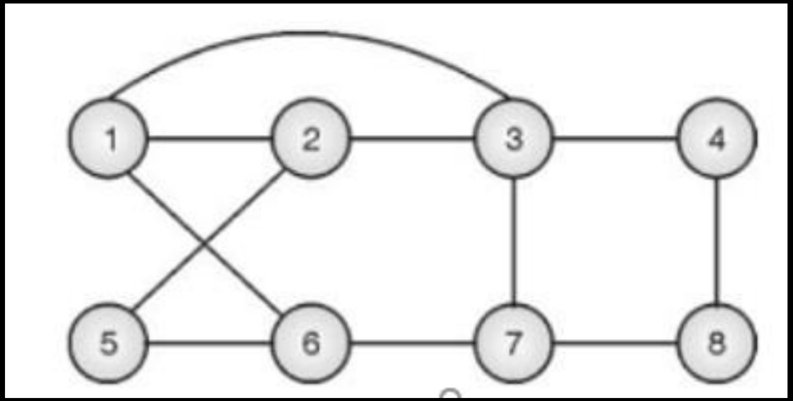




Hamiltonian Cycle using Backtracking

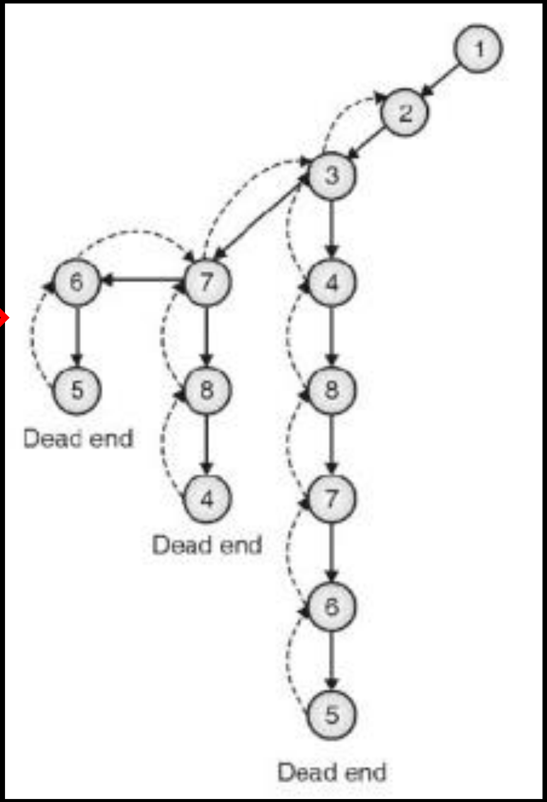
- Step 3: New path also leads to a dead end so backtrack and explore all possible paths

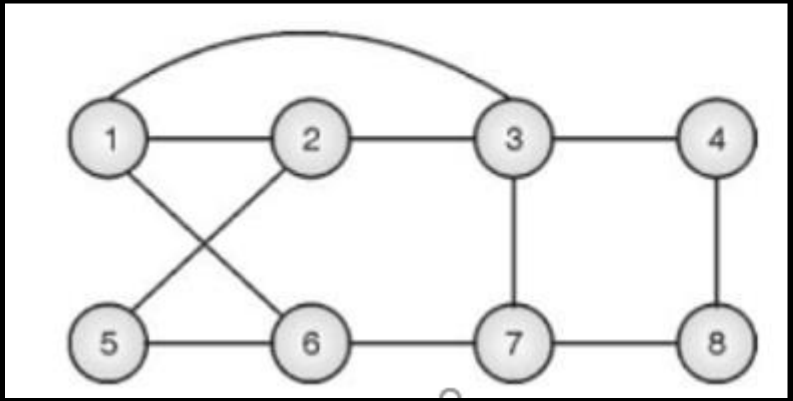




Hamiltonian Cycle using Backtracking

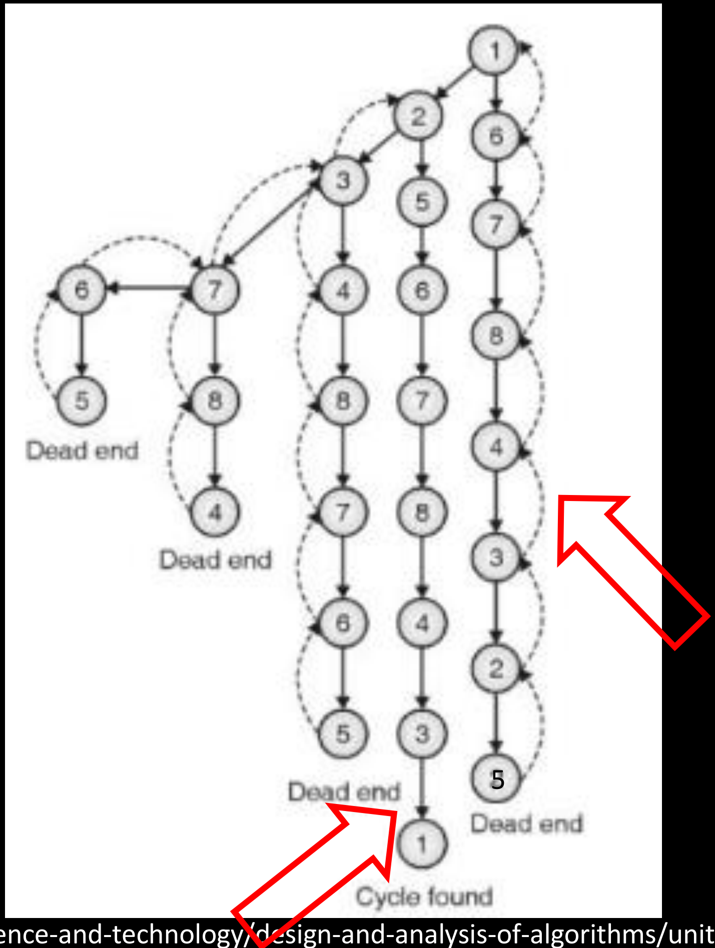
- Step 4: Next path is also leading to a dead-end so keep backtracking until we get some node that can generate a new path, i.e .vertex 2 here

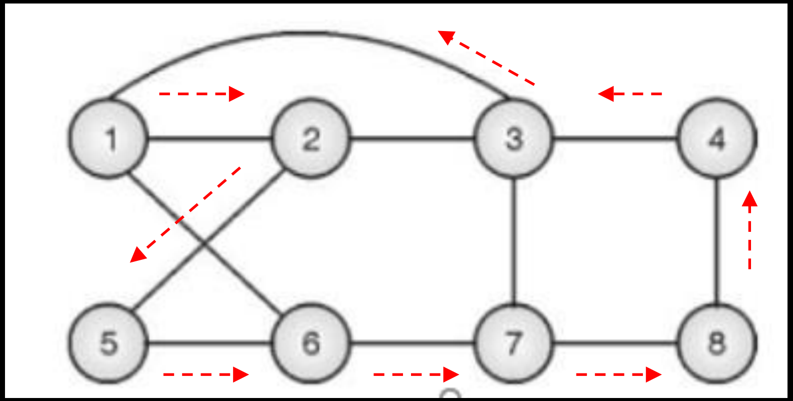




Hamiltonian Cycle using Backtracking

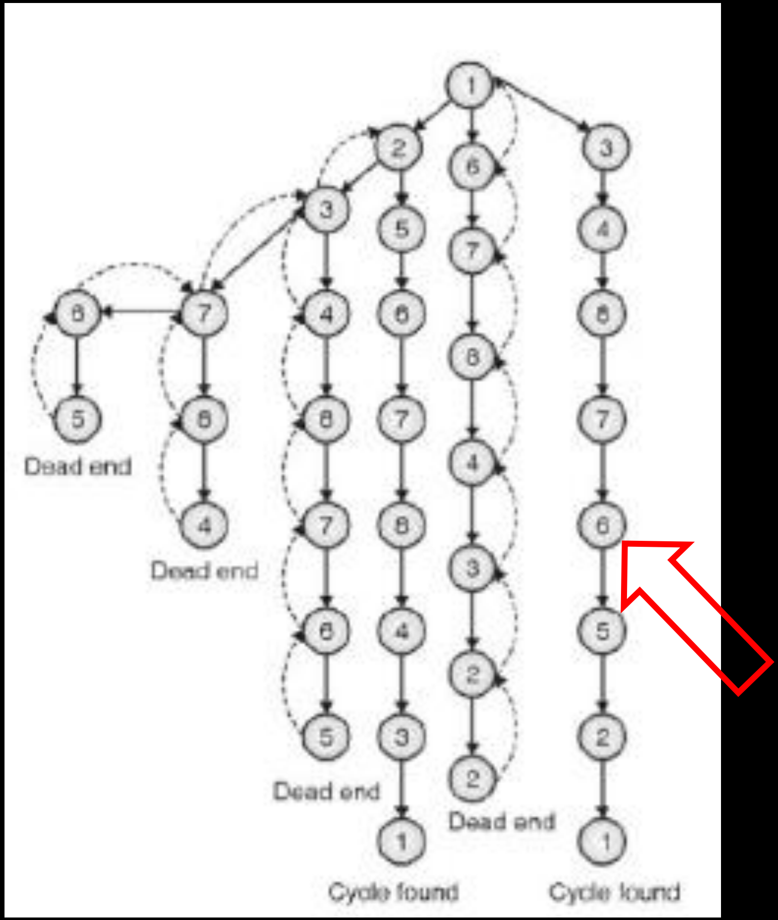
- Step 5: One path leads to Hamiltonian cycle, next leads to a dead end so backtrack and explore all possible paths at each vertex

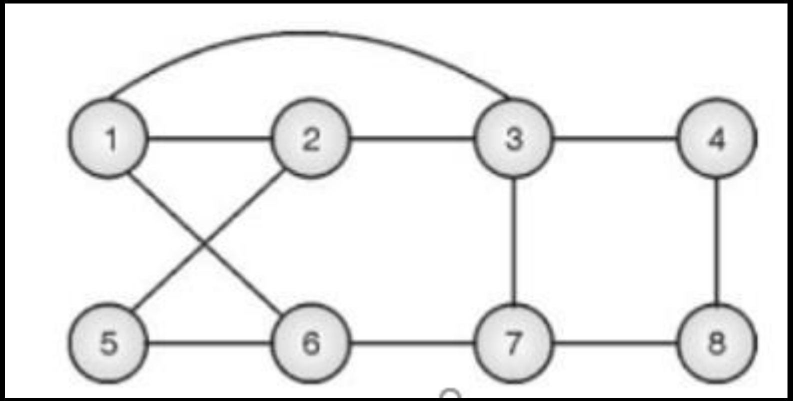




Hamiltonian Cycle using Backtracking

- Step 6: Total two Hamiltonian cycles are detected in the given graph



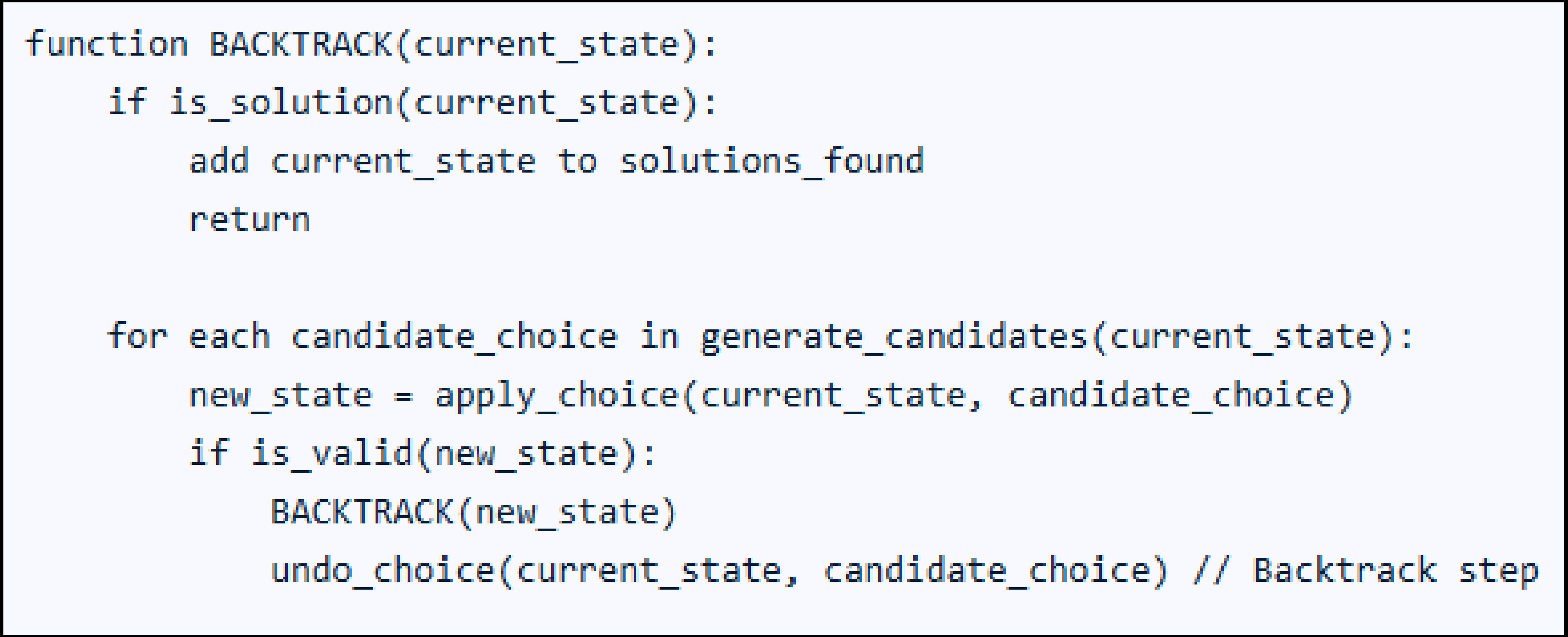


https://www.studocu.com/in/document/srm-institute-of-science-and-technology/design-and-analysis-of-algorithms/unit-4/94659699

Backtracking Optimisaiton

- Backtracking algorithms can be optimized by implementing pruning techniques, which involve eliminating paths that are unlikely to lead to a solution early on.
  - Memoization can also be used to store results of already computed subproblems to avoid redundant work.
- Pruning in backtracking refers to the process of cutting off or discarding paths that are unlikely to lead to a solution, thus reducing the number of possibilities the algorithm needs to explore.
- Effective pruning can significantly improve the efficiency of a backtracking algorithm.

Generic Pseudocode



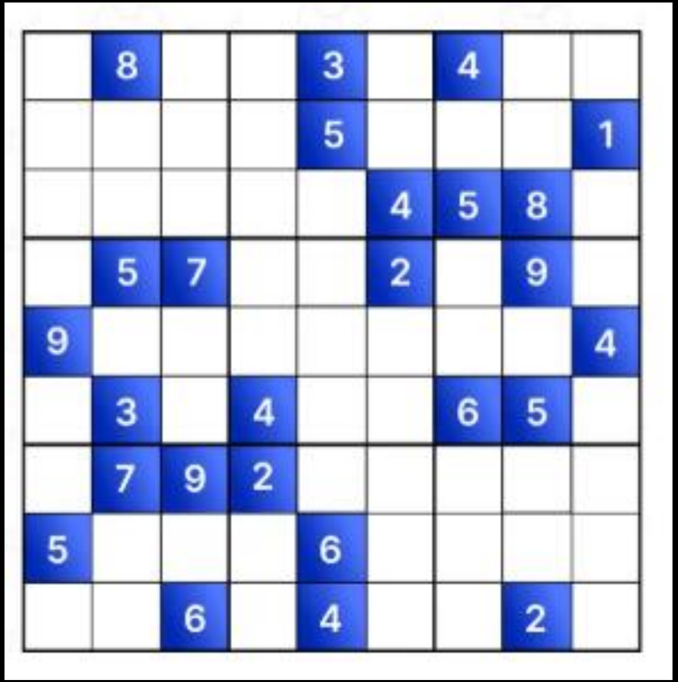
```
function BACKTRACK(current_state):
    if is_solution(current_state):
        add current_state to solutions_found
        return

    for each candidate_choice in generate_candidates(current_state):
        new_state = apply_choice(current_state, candidate_choice)
        if is_valid(new_state):
            BACKTRACK(new_state)
            undo_choice(current_state, candidate_choice) // Backtrack step
```

- Explanation of Components:
  - BACKTRACK(current_state): This is the recursive function that explores potential solutions. current_state represents the partial solution built so far.
  - is_solution(current_state): This function checks if the current_state represents a complete and valid solution to the problem. If it does, the solution is recorded.
  - generate_candidates(current_state): This function generates all possible choices or extensions that can be made from the current_state to build the solution further.
  - apply_choice(current_state, candidate_choice): This function creates a new_state by applying the candidate_choice to the current_state.
  - is_valid(new_state): This function checks if the new_state is still a potentially valid path towards a solution, respecting all problem constraints. If not, the current path is pruned, and no further exploration occurs down this branch.
  - undo_choice(current_state, candidate_choice): This crucial step is the "backtracking" part. After exploring a path (recursive call returns), this function reverts the changes made by apply_choice, restoring current_state to its state before the candidate_choice was applied. This allows the algorithm to explore other candidate choices from the same current_state.

Examples: Sudoku Solver

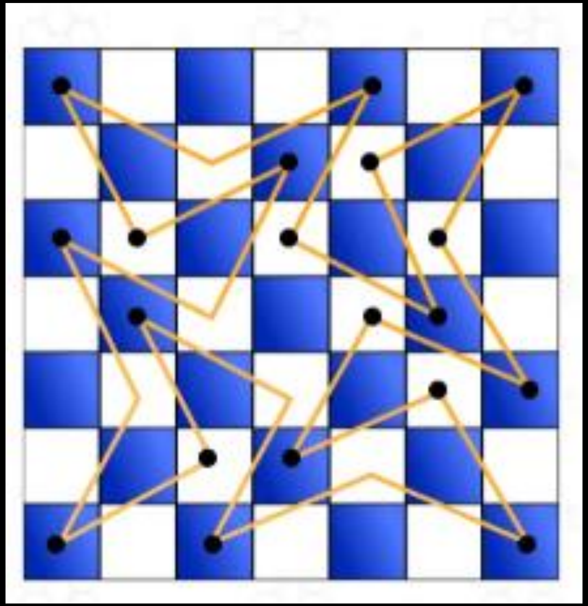
- Sudoku is a puzzle where the goal is to fill a 9×9 grid with numbers so that each row, column, and 3×3 subgrid contains all the digits from 1 to 9.
  - The backtracking algorithm tries to place numbers in empty cells, checking if the placement is valid.
  - If an invalid placement is detected, it backtracks and tries a different number, continuing until the puzzle is solved.



	8			3		4		
				5				1
					4	5	8	
	5	7			2		9	
9								4
	3		4			6	5	
	7	9	2					
5				6				
		6		4			2	

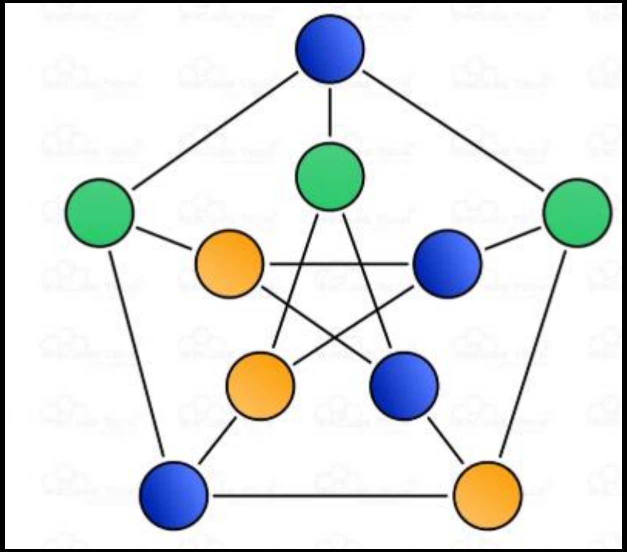
Examples: Knight’s Tour Problem

- The Knight's Tour problem involves moving a knight on a chessboard such that it visits every square exactly once. The backtracking algorithm tries different moves from the current position, marking squares as visited.
- If the knight cannot continue without revisiting a square, the algorithm backtracks and tries a different sequence of moves.



Graph Coloring Problem

- In the Graph Coloring problem, the goal is to color the vertices of a graph using the MINIMUM number of colors so that NO two adjacent vertices share the SAME color.
- The backtracking algorithm assigns colors to vertices one by one, ensuring that each color assignment is valid. If a conflict arises, it backtracks and tries a different color.
- Graph coloring has diverse applications, including timetabling and scheduling to avoid conflicts, frequency assignment in wireless networks to prevent interference, map coloring, and in computer science for tasks like register allocation and chip design.
-



Further Reference

- https://leetcode.com/problem-list/backtracking/