**✅ Practical 5: Implementing DFS and BFS Algorithms (Graph/Tree Traversal)**

**🔹 Concept Explanation (Viva – In-depth)**

**DFS (Depth First Search)** and **BFS (Breadth First Search)** are two fundamental algorithms used for graph and tree traversal.

**🔍 1. Difference Between DFS and BFS (Detailed)**

| **Aspect** | **DFS (Depth First Search)** | **BFS (Breadth First Search)** |
| --- | --- | --- |
| Approach | Goes deep before backtracking | Explores neighbors level by level |
| Data Structure | Stack (usually recursion) | Queue |
| Completeness | May not find the shortest path (in unweighted graph) | Always finds the shortest path (in unweighted graph) |
| Memory | Uses less memory in sparse graphs | High memory in wide/deep graphs |
| Time Efficiency | May be faster in trees or deep paths | Better in finding shortest path |
| Cycle Handling | Needs visited set to prevent infinite loops | Also needs visited set |
| Tree Example | Goes from root to one branch completely | Visits all children of root before going deeper |
| Use Cases | Topological sorting, solving mazes | Shortest path, social networks |
| Recursive | Naturally implemented recursively | Typically iterative |
| Search Direction | Depth-wise | Breadth-wise |

🔍 **Can BFS be Recursive?**  
No, BFS is not typically implemented recursively because it relies on a queue (FIFO) to explore nodes level by level. Recursion uses a call stack (LIFO), which suits DFS but not BFS. While you *can* simulate BFS with recursion, it's not efficient or natural.

**🧠 2. Time and Space Complexities**

Let:

* **b** = branching factor (average number of children per node)
* **d** = depth of the solution (goal node)
* **n** = total number of nodes

**DFS:**

* Time Complexity: O(b^d) (in worst-case when goal is at depth d)
* Space Complexity: O(d) (if recursive)

**BFS:**

* Time Complexity: O(b^d) (same worst-case scenario)
* Space Complexity: O(b^d) (because all nodes at current level are stored in queue)

👉 **DFS is more memory-efficient**, but BFS guarantees shortest path in unweighted graphs.

**⚙️ 3. Algorithm Working**

**📘 Depth First Search (DFS) – Recursive:**

1. Start from a source node
2. Mark it as visited and print it
3. Recursively visit all unvisited neighbors
4. Backtrack when no unvisited neighbors remain

**📗 Breadth First Search (BFS) – Iterative:**

1. Start from a source node
2. Enqueue the source and mark it as visited
3. While queue is not empty:
   * Dequeue a node
   * Visit and print it
   * Enqueue all unvisited neighbors

**💻 Python Code (DFS & BFS – Ready to Run)**

from collections import deque

def add\_edge(graph, u, v):

graph[u].append(v)

graph[v].append(u)

def dfs\_recursive(graph, node, visited):

if node not in visited:

print(node, end=" ")

visited.add(node)

for neighbor in graph[node]:

dfs\_recursive(graph, neighbor, visited)

def bfs(graph, start):

visited = set()

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

print(node, end=" ")

visited.add(node)

queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

# Sample graph

graph = {

'A': [], 'B': [], 'C': [], 'D': [], 'E': [], 'F': [], 'G': []

}

add\_edge(graph, 'A', 'B')

add\_edge(graph, 'A', 'C')

add\_edge(graph, 'B', 'D')

add\_edge(graph, 'B', 'E')

add\_edge(graph, 'C', 'F')

add\_edge(graph, 'C', 'G')

print("DFS Traversal:")

dfs\_recursive(graph, 'A', set())

print("\nBFS Traversal:")

bfs(graph, 'A')

**❓ Important Viva Questions**

1. **How does DFS differ from BFS in goal search efficiency?**
   * DFS may reach the goal faster in some structures, but BFS is optimal for shortest path in unweighted graphs.
2. **Why is BFS memory-intensive?**
   * Because it stores all nodes at the current level before going deeper (O(b^d)).
3. **Can BFS be implemented recursively?**
   * No, it's not natural or efficient because recursion follows a LIFO pattern, not suitable for BFS’s level-order behavior.
4. **When would DFS be preferred over BFS?**
   * In space-limited situations, or when we expect solutions to be deep in the tree.

**🎯 Objective:**

To implement and understand DFS and BFS traversal techniques on graphs/trees using Python. To analyze their working mechanism, compare their characteristics, and determine their best-fit use cases in problem solving.

**🔚 Conclusion:**

DFS and BFS are key strategies for traversing or searching graph data structures. Understanding their trade-offs in memory, completeness, and optimality is essential. BFS uses a queue and is best for shortest path discovery, whereas DFS is better for deep searching with less memory usage. Practical implementation builds critical thinking in AI-based search problems.

**✅ Practical 6: Implement A\* Algorithm for Maze Solver**

**🔹 Concept Explanation (Viva – In-depth)**

**A\* (A-star) Search Algorithm** is a popular informed search algorithm used in pathfinding and graph traversal. It finds the shortest path from a start node to a goal node using a combination of cost functions.

A\* uses:

* **g(n)**: Cost from start node to current node n
* **h(n)**: Heuristic — estimated cost from n to goal
* **f(n) = g(n) + h(n)**: Total estimated cost of path through n

**🔍 1. Difference Between A\*, BFS, DFS (Deeper Concept)**

| **Feature** | **A\* Algorithm** | **BFS** | **DFS** |
| --- | --- | --- | --- |
| Uses Heuristic | ✅ Yes (uses h(n)) | ❌ No | ❌ No |
| Optimality | ✅ Yes (if h(n) is admissible) | ✅ Yes | ❌ Not always |
| Completeness | ✅ | ✅ | ❌ |
| Memory Usage | High | High | Low |
| Time Complexity | O(b^d) but optimized | O(b^d) | O(b^d) |
| Suitable For | Shortest path with cost | Unweighted shortest path | Deep search/tree traversal |

**🧠 2. Time and Space Complexities**

Let:

* **b** = branching factor
* **d** = depth of the solution
* **h(n)** = heuristic function

If the heuristic is admissible and consistent:

* **Time Complexity**: O(b^d) (in worst case)
* **Space Complexity**: O(b^d) (due to priority queue and open/closed lists)

A\* can be faster than BFS if h(n) significantly reduces search space.

**⚙️ 3. Algorithm Working (Maze Solver)**

1. Initialize open and closed lists
2. Add start node to open list
3. Loop until open list is empty:
   * Pick node with lowest f(n)
   * Move it to closed list
   * If it's goal, return path
   * Else, for each neighbor:
     + If in closed list, skip
     + Calculate g, h, f
     + If not in open or better path, update and add to open

**💻 Python Code (Maze Solver using A\*)**

from queue import PriorityQueue

def heuristic(a, b):

return abs(a[0] - b[0]) + abs(a[1] - b[1]) # Manhattan distance

def astar(maze, start, goal):

rows, cols = len(maze), len(maze[0])

open\_set = PriorityQueue()

open\_set.put((0, start))

came\_from = {}

g\_score = {start: 0}

f\_score = {start: heuristic(start, goal)}

while not open\_set.empty():

\_, current = open\_set.get()

if current == goal:

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

path.reverse()

return path

for dx, dy in [(0,1),(1,0),(0,-1),(-1,0)]:

neighbor = (current[0] + dx, current[1] + dy)

if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols:

if maze[neighbor[0]][neighbor[1]] == 1:

continue

tentative\_g = g\_score[current] + 1

if tentative\_g < g\_score.get(neighbor, float('inf')):

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g

f\_score[neighbor] = tentative\_g + heuristic(neighbor, goal)

open\_set.put((f\_score[neighbor], neighbor))

return None

# 0 = path, 1 = wall

maze = [

[0, 1, 0, 0, 0],

[0, 1, 0, 1, 0],

[0, 0, 0, 1, 0],

[0, 1, 1, 1, 0],

[0, 0, 0, 0, 0],

]

start = (0, 0)

goal = (4, 4)

path = astar(maze, start, goal)

print("A\* Path from start to goal:")

print(path)

**❓ Important Viva Questions**

1. *Why is A more efficient than BFS?*\*
   * Because A\* uses heuristics to guide the search, reducing the explored space.
2. **What is an admissible heuristic?**
   * A heuristic that never overestimates the actual cost to reach the goal.
3. *Can A be used for weighted graphs?*\*
   * Yes, it's designed for graphs with costs, unlike BFS.

**🎯 Objective:**

To apply the A\* algorithm for solving game search problems such as maze navigation. Understand how heuristic improves pathfinding efficiency and implement the method in Python.

**🔚 Conclusion:**

A\* algorithm provides an optimal and complete solution in game or maze environments by balancing actual and estimated costs. It's a cornerstone of AI pathfinding and is widely used in robotics, games, and navigation systems.

**Practical 7: Implement Greedy Search Algorithm for Selection Sort**

**🔹 Concept Explanation (Viva – In-depth)**

**Greedy Algorithm:** A greedy algorithm is a straightforward approach that makes the locally optimal choice at each stage with the hope of finding a global optimum. It is used for optimization problems.

**Selection Sort:** Selection Sort is a simple comparison-based sorting algorithm. The idea behind Selection Sort is to repeatedly select the smallest (or largest, depending on sorting order) element from the unsorted portion of the list and place it in the correct position. It works by iterating over the unsorted portion, selecting the minimum element, and swapping it with the element at the beginning of the unsorted part.

**Greedy Search for Selection Sort:**

* At each step, find the smallest element in the remaining unsorted list and swap it with the element at the current position.
* This approach can be considered as a greedy algorithm because it makes the locally optimal choice (the smallest element) in each step, hoping it will lead to the globally sorted list.

**🔍 1. Difference Between Greedy Search and Other Algorithms**

| **Feature** | **Greedy Search (for Selection Sort)** | **Other Sorting Algorithms (e.g., QuickSort, MergeSort)** |
| --- | --- | --- |
| Approach | Makes the locally optimal choice at each step | Divide and conquer, recursive or iterative approach |
| Time Complexity | O(n^2) | O(n log n) (QuickSort, MergeSort) |
| Space Complexity | O(1) | O(n) (for MergeSort), O(log n) (for QuickSort) |
| Stability | Unstable (if equal elements are swapped) | Stable (for MergeSort) |
| Use Case | Simple sorting with small datasets | Efficient sorting with large datasets |
| Comparison Efficiency | O(n^2) comparisons | O(n log n) comparisons (in average-case for QuickSort) |

**🧠 2. Time and Space Complexities**

**Selection Sort:**

* Time Complexity: O(n^2) – Involves nested loops, one for traversing the unsorted portion and the other for finding the smallest element.
* Space Complexity: O(1) – Only a constant amount of extra space is required.

**Greedy Search in Selection Sort:**

* Time Complexity: O(n^2) – For each element, we perform a linear scan to find the smallest remaining element.
* Space Complexity: O(1) – The algorithm sorts the array in-place without requiring additional space.

**⚙️ 3. Algorithm Working**

1. **Initial Array**: Given an unsorted array, selection sort will find the smallest element from the unsorted portion and swap it with the first unsorted element.
2. **Iterative Process**:
   * Find the smallest element in the unsorted portion of the list.
   * Swap this smallest element with the first unsorted element.
   * Repeat the above steps for all elements in the array until the list is sorted.
3. **Greedy Choice**: The greedy choice is selecting the smallest element in each iteration.

**💻 Python Code (Greedy Selection Sort – Ready to Run)**

python

CopyEdit

def selection\_sort(arr):

"""Greedy approach to Selection Sort"""

n = len(arr)

for i in range(n):

# Assume the current index as the minimum

min\_idx = i

# Find the minimum element in the remaining unsorted array

for j in range(i+1, n):

if arr[j] < arr[min\_idx]:

min\_idx = j

# Swap the found minimum element with the first element

arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

return arr

# Example Usage

arr = [64, 25, 12, 22, 11]

print("Original Array:", arr)

sorted\_arr = selection\_sort(arr)

print("Sorted Array:", sorted\_arr)

**❓ Important Viva Questions**

1. **How does the Greedy algorithm apply in Selection Sort?**
   * The algorithm makes a locally optimal choice by selecting the smallest element in each iteration and swapping it with the first unsorted element. It hopes that these local optimal choices will result in a globally sorted array.
2. **Why is Selection Sort considered inefficient for large datasets?**
   * Because of its O(n^2) time complexity, which makes it inefficient when dealing with large datasets compared to more advanced sorting algorithms like MergeSort or QuickSort.
3. **What is the advantage of using Selection Sort despite its inefficiency?**
   * Selection Sort has the advantage of O(1) space complexity, meaning it sorts the array in-place without needing extra space. This makes it useful in situations with limited memory.

**🎯 Objective:**

The objective of this practical is to implement the Greedy Search strategy in the context of the Selection Sort algorithm. This will help understand the concept of greedy algorithms, their application in sorting, and the trade-offs between time and space complexity in comparison to other sorting techniques.

**🔚 Conclusion:**

Greedy algorithms are simple but effective for solving certain problems. In the case of Selection Sort, the algorithm makes a locally optimal decision (selecting the smallest element) at each step. However, its O(n^2) time complexity makes it less efficient compared to other sorting algorithms for large datasets. Nonetheless, it serves as a great introduction to greedy techniques and is still useful for small datasets or scenarios where memory efficiency is a concern.

**Practical 8: Implement Greedy Search Algorithm for Minimum Spanning Tree (MST)**

**🔹 Concept Explanation (Viva – In-depth)**

A **Minimum Spanning Tree (MST)** is a subset of the edges of a connected, undirected graph that connects all the vertices together, without cycles, and with the minimum possible total edge weight. There are two well-known algorithms used to find an MST in a graph: **Kruskal’s Algorithm** and **Prim’s Algorithm**. Both of these algorithms use the **Greedy** approach.

* **Kruskal’s Algorithm**: This algorithm sorts all the edges of the graph by their weights in ascending order. It then adds edges one by one to the MST, making sure that no cycles are formed. The algorithm stops when the MST is complete.
* **Prim’s Algorithm**: This algorithm starts with a single node and iteratively adds the minimum weight edge that connects a vertex in the tree to a vertex outside the tree. It continues this process until all vertices are included in the MST.

Here, we will implement **Prim’s Algorithm** for finding the Minimum Spanning Tree of a graph, applying the greedy strategy.

**🔍 1. Difference Between Greedy MST Algorithms**

| **Feature** | **Kruskal’s Algorithm** | **Prim’s Algorithm** |
| --- | --- | --- |
| Approach | Edge-based (process all edges) | Vertex-based (expands tree from a vertex) |
| Data Structure | Uses Disjoint Set (Union-Find) for cycle detection | Uses Min-Heap/Priority Queue |
| Time Complexity | O(E log E), where E is the number of edges | O(E log V), where V is the number of vertices |
| Space Complexity | O(V) (for Union-Find sets) | O(V) (for adjacency list and heap) |
| Suitable for Sparse Graphs | Yes, good for sparse graphs | Works better for dense graphs |
| Edge Processing Order | Processes edges in sorted order | Processes vertices and selects edges |

**🧠 2. Time and Space Complexities**

**Prim’s Algorithm** (Greedy Approach for MST):

* **Time Complexity**:
  + If we use a **min-heap** to find the minimum weight edge efficiently, the time complexity is O(E log V), where **E** is the number of edges and **V** is the number of vertices.
  + In the worst case, when every vertex is connected to every other vertex, the complexity can become O(V^2), which happens with an adjacency matrix.
* **Space Complexity**:
  + O(V + E) for storing the graph (adjacency list), the priority queue, and auxiliary data structures like visited and parent.

**⚙️ 3. Algorithm Working (Prim’s Algorithm)**

1. **Initialization**: Start with any vertex (say vertex u) and mark it as visited.
2. **Priority Queue**: Use a priority queue (min-heap) to store all the edges with their weights. The priority queue always extracts the edge with the smallest weight.
3. **Edge Selection**: At each step, pick the smallest weight edge that connects a visited vertex to an unvisited vertex.
4. **Tree Expansion**: Add the selected edge to the MST and mark the new vertex as visited.
5. **Repeat**: Continue the process until all vertices are included in the MST.

**💻 Python Code (Prim’s Algorithm for Minimum Spanning Tree)**

python

CopyEdit

import heapq

def prim\_mst(graph, start):

"""Prim's Algorithm for Minimum Spanning Tree (MST)"""

mst = [] # List to store the MST edges

visited = set([start]) # Set to track visited nodes

edges = []

# Add all edges from the start vertex to the heap

for to, weight in graph[start]:

heapq.heappush(edges, (weight, start, to))

while edges:

weight, frm, to = heapq.heappop(edges) # Get the edge with the minimum weight

if to not in visited:

visited.add(to) # Mark the destination vertex as visited

mst.append((frm, to, weight)) # Add the edge to the MST

# Add all the edges from the current vertex to the priority queue

for next\_to, next\_weight in graph[to]:

if next\_to not in visited:

heapq.heappush(edges, (next\_weight, to, next\_to))

return mst

# Example Usage

graph = {

'A': [('B', 1), ('C', 3), ('D', 4)],

'B': [('A', 1), ('C', 2), ('D', 5)],

'C': [('A', 3), ('B', 2), ('D', 6)],

'D': [('A', 4), ('B', 5), ('C', 6)],

}

start\_node = 'A'

mst = prim\_mst(graph, start\_node)

# Output the MST

print("Edges in the Minimum Spanning Tree (MST):")

for edge in mst:

print(f"{edge[0]} - {edge[1]} with weight {edge[2]}")

**❓ Important Viva Questions**

1. **How does Prim’s algorithm differ from Kruskal’s algorithm in finding MST?**
   * Prim’s algorithm grows the MST by adding edges one by one, starting from an arbitrary vertex and expanding outwards. Kruskal’s algorithm, on the other hand, processes edges in sorted order and uses union-find to avoid cycles.
2. **Why is Prim’s algorithm more efficient for dense graphs compared to Kruskal’s?**
   * Prim’s algorithm uses a priority queue to efficiently get the next smallest edge, making it better suited for dense graphs. Kruskal’s algorithm, however, requires sorting all edges and can be less efficient for graphs with many edges.
3. **How does the priority queue (min-heap) improve the performance of Prim’s algorithm?**
   * The priority queue allows us to efficiently retrieve the smallest edge in O(log V) time, which reduces the overall complexity of the algorithm, especially for dense graphs.

**🎯 Objective:**

The goal of this practical is to implement Prim’s algorithm using a greedy approach to find the minimum spanning tree of a graph. This will help in understanding the greedy paradigm, its application in MST problems, and the trade-offs involved in using different algorithms for graph-based problems.

**🔚 Conclusion:**

Prim’s algorithm is an efficient way to solve the Minimum Spanning Tree problem, particularly for dense graphs. By employing a greedy strategy, it continuously selects the smallest edge that expands the tree, ensuring an optimal solution. Understanding the differences between Prim’s and Kruskal’s algorithms can help determine the best approach for solving MST problems in different types of graphs.

**Practical 9: Implementing Branch and Bound and Backtracking for Constraint Satisfaction Problem (CSP) – N-Queens Problem**

**🔹 Concept Explanation (Viva – In-depth)**

A **Constraint Satisfaction Problem (CSP)** involves finding a solution that satisfies a set of constraints. One of the most famous CSPs is the **N-Queens problem**, where the goal is to place **N queens** on an **N×N chessboard** such that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal.

In this practical, we will solve the 4-Queens problem using two methods: **Backtracking** and **Branch and Bound**.

**🔍 1. Difference Between Backtracking and Branch and Bound**

| **Feature** | **Backtracking** | **Branch and Bound** |
| --- | --- | --- |
| Approach | Exhaustively searches all possible solutions by trying one option at a time | Prunes large parts of the search space using bounds |
| Use of Constraints | Checks constraints during each step and backtracks if violated | Uses bounds to eliminate non-promising solutions early |
| Time Complexity | Can be exponential without pruning (O(N!)) | More efficient (O(N^2)) in many cases |
| Space Complexity | Requires space for recursion stack | Typically lower as it avoids exploring large branches |
| Optimality | May not be optimal without further refinement | Can guarantee optimal solutions (depending on the bound used) |

**🧠 2. Time and Space Complexities**

* **Backtracking**:
  + **Time Complexity**: In the worst case, backtracking will explore all possible configurations. For the N-Queens problem, this is O(N!) due to the factorial growth of the recursive calls.
  + **Space Complexity**: O(N) for the recursion stack, where N is the number of queens (since each recursive call adds a new queen to the board).
* **Branch and Bound**:
  + **Time Complexity**: With proper pruning, branch and bound can reduce the search space, but it can still be exponential in the worst case (O(N!)). The exact complexity depends on the efficiency of the bound function.
  + **Space Complexity**: O(N) for the recursion stack and additional space for storing bounds and decisions.

**⚙️ 3. Algorithm Working**

**📘 Backtracking Approach:**

1. Place a queen on the board in the first available position.
2. Check if placing the queen violates any constraints (same row, column, or diagonal).
3. If no violation, proceed to place the next queen.
4. If a violation occurs, backtrack (remove the last queen placed) and try the next possible position.
5. Repeat the process until all queens are placed or all options are exhausted.

**📗 Branch and Bound Approach:**

1. Start with an empty board.
2. Place a queen on the first column and recursively place subsequent queens on each column.
3. If a queen violates a constraint, prune the current branch (discard that configuration).
4. Use a bound function to determine if a solution can still be found from the current configuration.
5. Repeat until a valid solution is found or all branches are pruned.

**💻 Python Code**

**Backtracking for N-Queens Problem**:

python

CopyEdit

def is\_safe(board, row, col):

"""Check if it's safe to place a queen at board[row][col]"""

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, len(board), 1), range(col, len(board), 1)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens(board, col):

"""Solve N-Queens problem using backtracking"""

if col >= len(board):

return True

for row in range(len(board)):

if is\_safe(board, row, col):

board[row][col] = 1

if solve\_n\_queens(board, col + 1):

return True

board[row][col] = 0 # Backtrack

return False

def print\_board(board):

"""Print the chessboard solution"""

for row in board:

print(" ".join(["Q" if x else "." for x in row]))

# Example usage for 4-Queens

N = 4

board = [[0 for \_ in range(N)] for \_ in range(N)]

if solve\_n\_queens(board, 0):

print("Solution found:")

print\_board(board)

else:

print("No solution exists.")

**Branch and Bound for N-Queens Problem**:

python

CopyEdit

def is\_safe(board, row, col):

"""Check if it's safe to place a queen at board[row][col]"""

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, len(board), 1), range(col, len(board), 1)):

if board[i][j] == 1:

return False

return True

def bound(board, col):

"""Estimate the bound for the current state"""

for row in range(len(board)):

if is\_safe(board, row, col):

return True

return False

def solve\_n\_queens\_branch\_bound(board, col):

"""Solve N-Queens problem using Branch and Bound"""

if col >= len(board):

return True

for row in range(len(board)):

if is\_safe(board, row, col):

board[row][col] = 1

if bound(board, col + 1) and solve\_n\_queens\_branch\_bound(board, col + 1):

return True

board[row][col] = 0 # Backtrack

return False

# Example usage for 4-Queens

N = 4

board = [[0 for \_ in range(N)] for \_ in range(N)]

if solve\_n\_queens\_branch\_bound(board, 0):

print("Solution found (Branch and Bound):")

print\_board(board)

else:

print("No solution exists.")

**❓ Important Viva Questions**

1. **What is the difference between backtracking and branch and bound in solving CSPs like N-Queens?**
   * Backtracking explores all possibilities and backtracks if a constraint is violated. Branch and Bound prunes the search tree using bounds to eliminate unpromising paths early, potentially reducing the search space.
2. **How does the bound function work in the Branch and Bound algorithm?**
   * The bound function checks whether it is possible to complete the solution from the current state. If the bound function returns False, that branch is pruned, as it is not possible to reach a solution from that point.
3. **Why does the backtracking approach sometimes take longer to find a solution compared to branch and bound?**
   * Backtracking explores all possibilities exhaustively without pruning unpromising paths, which may lead to redundant searches. Branch and Bound, on the other hand, can eliminate large parts of the search space early, speeding up the solution process.

**🎯 Objective:**

The goal of this practical is to implement both backtracking and branch and bound techniques to solve the N-Queens problem. By understanding and implementing these algorithms, students will gain insight into constraint satisfaction problems and how different techniques can be applied for optimization.

**🔚 Conclusion:**

Both backtracking and branch and bound are powerful techniques for solving constraint satisfaction problems, such as the N-Queens problem. While backtracking can be simple and intuitive, branch and bound often provides more efficiency by pruning unpromising branches early in the search process, making it better suited for large and complex CSPs.