AISSMS
COLLEGE OF ENGINEERING
ज्ञानम् सकलजनहिताय

Approved by AICTE, New Delhi, Recognized by Government of Maharashtra
Affiliated to Savitribai Phule Pune University and recognized 2(f) and 12(B) by UGC
(Id.No. PU/PN/Engg./093 (1992)
Accredited by NAAC with "A+" Grade | NBA - 7 UG Programmes

Department of Computer Engineering

"Artificial Intelligence Lab Manual"

Submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF ENGINEERING

In

COMPUTER ENGINEERING

Submitted By

Name of the Student: Piyusha Rajendra Supe

Roll No: 23CO315

Batch: C (TE-B)

Under the Guidance of

Prof. Ashish U. Khandait

ALL INDIA SHRI SHIVAJI MEMORIAL SOCIETY'S COLLEGE OF ENGINEERING PUNE-411001

Academic Year: 2024-25(Term-II)

Savitribai Phule Pune University

A.I.S.S.M.S. COLLEGE OF ENGINEERING PUNE 411001

This is to certify that Mr./ **Miss. Piyusha Rajendra Supe** of class **TE-B (Computer)** Roll No. **23CO315** has completed all the practical work as listed above, satisfactorily in the subject of **Laboratory Practice – II (Artificial Intelligence)** in the Department of **Computer** Engineering as prescribed by the Savitribai Phule Pune University. During the academic year **2024-2025**.

Date: **Prof. In-charge** **Head of the Department**

Practical 01

Title of the Assignment: Searching Algorithms (Uninformed Search)

Problem Statement: Implement depth first search algorithm and Breadth First Search algorithm, use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

Objective of the Assignment: Students must be able to use searching strategies and identify their use cases

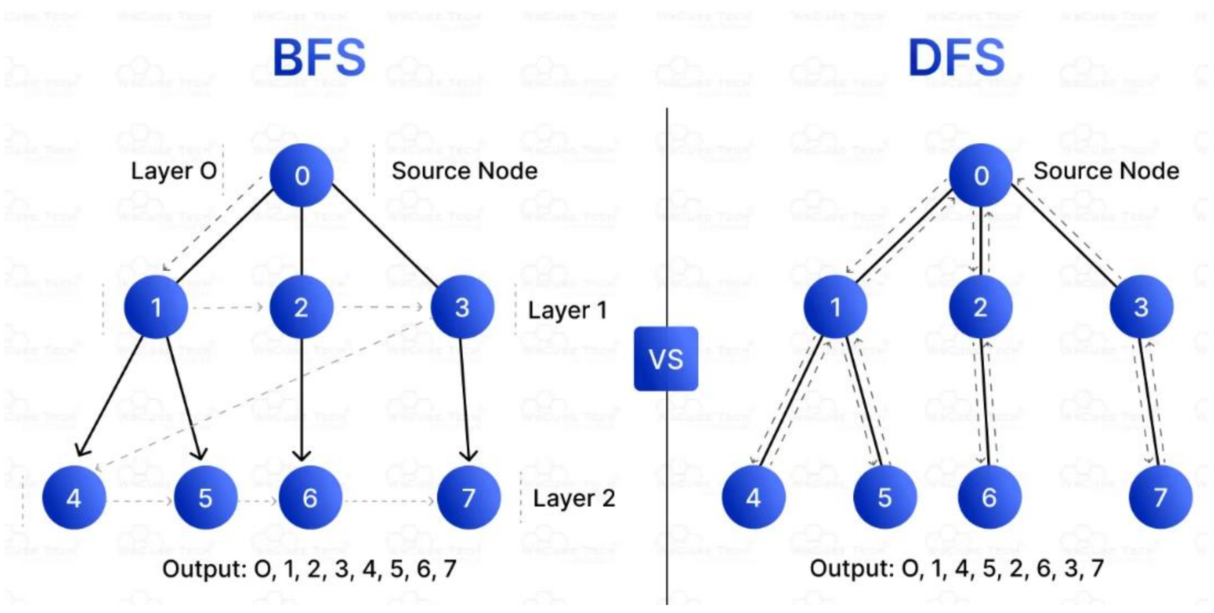
Prerequisite: Data Structures

Theory:

Searching algorithms are fundamental techniques used to explore graphs or tree data structures to locate a specific node or to traverse all the nodes. There are two primary types of uninformed (or blind) search strategies:

1. **Depth First Search (DFS)**:
    - DFS explores as far as possible along each branch before backtracking.
    - It uses a **stack** data structure (either explicitly or through recursion).
    - Ideal for scenarios where the solution is deep in the tree or graph.
2. **Breadth First Search (BFS)**:
    - BFS explores all the neighbours at the current depth before moving to nodes at the next level.
    - It uses a **queue** data structure.
    - Best used when the shortest path or closest node is required.

Both algorithms are used in various domains such as AI, pathfinding in games, puzzle solving, and network traversal.


BFS
Layer 0
Source Node
Layer 1
Layer 2
Output: 0, 1, 2, 3, 4, 5, 6, 7
VS
DFS
Source Node
Output: 0, 1, 4, 5, 2, 6, 3, 7

Algorithm Stepwise

1. Depth First Search (DFS) - Recursive

```
DFS(graph, vertex, visited):
    Mark vertex as visited
    Print or process the vertex
    For each adjacent vertex v of vertex:
        If v is not visited:
            DFS(graph, v, visited)
```

Steps:

1. Initialize a set or list to keep track of visited nodes.
2. Start DFS from any node (e.g., starting vertex).
3. Recursively visit all unvisited neighbours.
4. Continue the recursion until all nodes are visited.

2. Breadth First Search (BFS) - Iterative

```
BFS(graph, start):
    Initialize a queue Q
    Initialize a visited set
    Add start node to Q and mark as visited

    While Q is not empty:
        vertex = Q.dequeue()
        Print or process the vertex
        For each adjacent vertex v of vertex:
            If v is not visited:
                Mark v as visited
                Enqueue v to Q
```

Steps:

1. Initialize a queue and visited list.
2. Enqueue the starting node and mark it as visited.
3. Dequeue a node and enqueue all its unvisited neighbours.
4. Repeat until the queue is empty.

Conclusion / Analysis

- **DFS** and **BFS** are both effective graph traversal algorithms, but they serve different purposes:
    - DFS is **memory efficient** (especially with recursion) and useful for **pathfinding** in deep graphs.

    - BFS guarantees the **shortest path** in unweighted graphs and is suitable for **level-order traversal**.
- **Time Complexity**:
    - Both DFS and BFS have a time complexity of **O (V + E)**, where `V` is the number of vertices and `E` is the number of edges.
- **Space Complexity**:
    - DFS: O(V) in the worst case due to recursive stack.
    - BFS: O(V) due to the queue and visited list.

Use Cases:

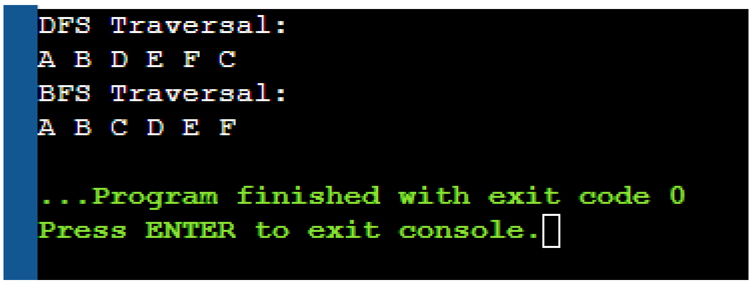
- **DFS**: Puzzle solving (e.g., maze), topological sorting, connected components.
- **BFS**: Shortest path in unweighted graphs, finding all neighbours within N steps, web crawlers.

In conclusion, understanding and implementing both DFS and BFS enables students to solve a wide variety of problems in data structures, AI, and networking.

Program:

```
from collections import deque

# Create a graph using adjacency list

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F'],

    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E']

}

# DFS using recursion

def dfs(graph, node, visited=None):

    if visited is None:

        visited = set()

    if node not in visited:

        print(node, end=' ')

        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)

# BFS using queue
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)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

# Driver Code
print("DFS Traversal:")
dfs(graph, 'A')

print("\nBFS Traversal:")
bfs(graph, 'A')
```

Output:



```
DFS Traversal:
A B D E F C
BFS Traversal:
A B C D E F

...Program finished with exit code 0
Press ENTER to exit console.
```

Practical 02

Title of the Assignment: Informed Search

Problem Statement: Implement A Star Algorithm for any game search problem

Objective of the Assignment: Students must be able to use searching strategies and identify their use cases

Prerequisite: Knowledge of heuristic functions and Manhattan distance concept

Theory:

Informed Search Algorithms use problem-specific knowledge (heuristics) to find the goal state more efficiently than uninformed methods. One of the most popular informed search techniques is the **A* (A-star) algorithm**.

A* Search Algorithm combines the advantages of **Uniform Cost Search** and **Greedy Best-First Search** by considering both the cost to reach a node and the estimated cost to reach the goal from that node.

It uses the **evaluation function**:

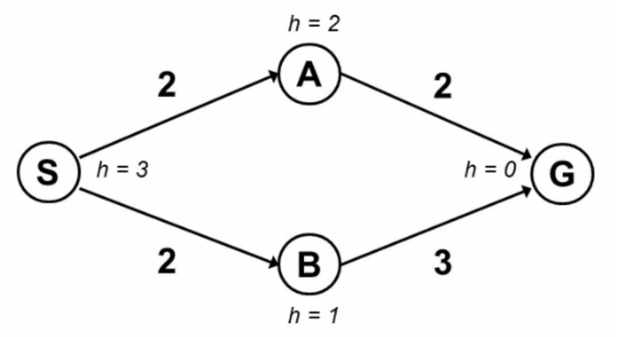
$$f(n) = g(n) + h(n)$$

- **g(n)**: the actual cost from the start node to node n
- **h(n)**: the heuristic estimated cost from node n to the goal (e.g., Manhattan Distance)
- **f(n)**: total estimated cost of the cheapest solution through node n

Heuristic Function: The performance of A* heavily depends on the heuristic function. A common heuristic for grid-based pathfinding is the **Manhattan Distance**, calculated as:

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

where (x1, y1) is the current position and (x2, y2) is the goal.



Algorithm Stepwise

A* Algorithm

```
1. Initialize an open list (priority queue) and add the start node.
2. Initialize a closed list (visited set).
3. While the open list is not empty:
   a. Remove the node with the lowest f(n) from the open list.
   b. If the node is the goal, return the path.
   c. Add the node to the closed list.
   d. For each neighbour of the current node:
      i. If the neighbour is in the closed list, skip it.
      ii. Calculate g(n), h(n), and f(n) for the neighbour.
      iii. If the neighbour is not in the open list, add it.
      iv. If the neighbour is in the open list with a higher f(n), update its values.
4. If the open list is empty and goal not found, return failure.
```

Conclusion / Analysis

- **A*** is both **complete** and **optimal** if the heuristic used is **admissible** (never overestimates the true cost).
- It performs better than uninformed search methods because it uses additional information (heuristics) to guide the search.
- The **Manhattan Distance** is an effective heuristic for grid-based pathfinding problems like 8-puzzle or maze navigation.
- **Time Complexity**: Depends on the branching factor and quality of the heuristic. In the worst case, it's **exponential**.
- **Space Complexity**: Also high, since it keeps all generated nodes in memory (open and closed lists).

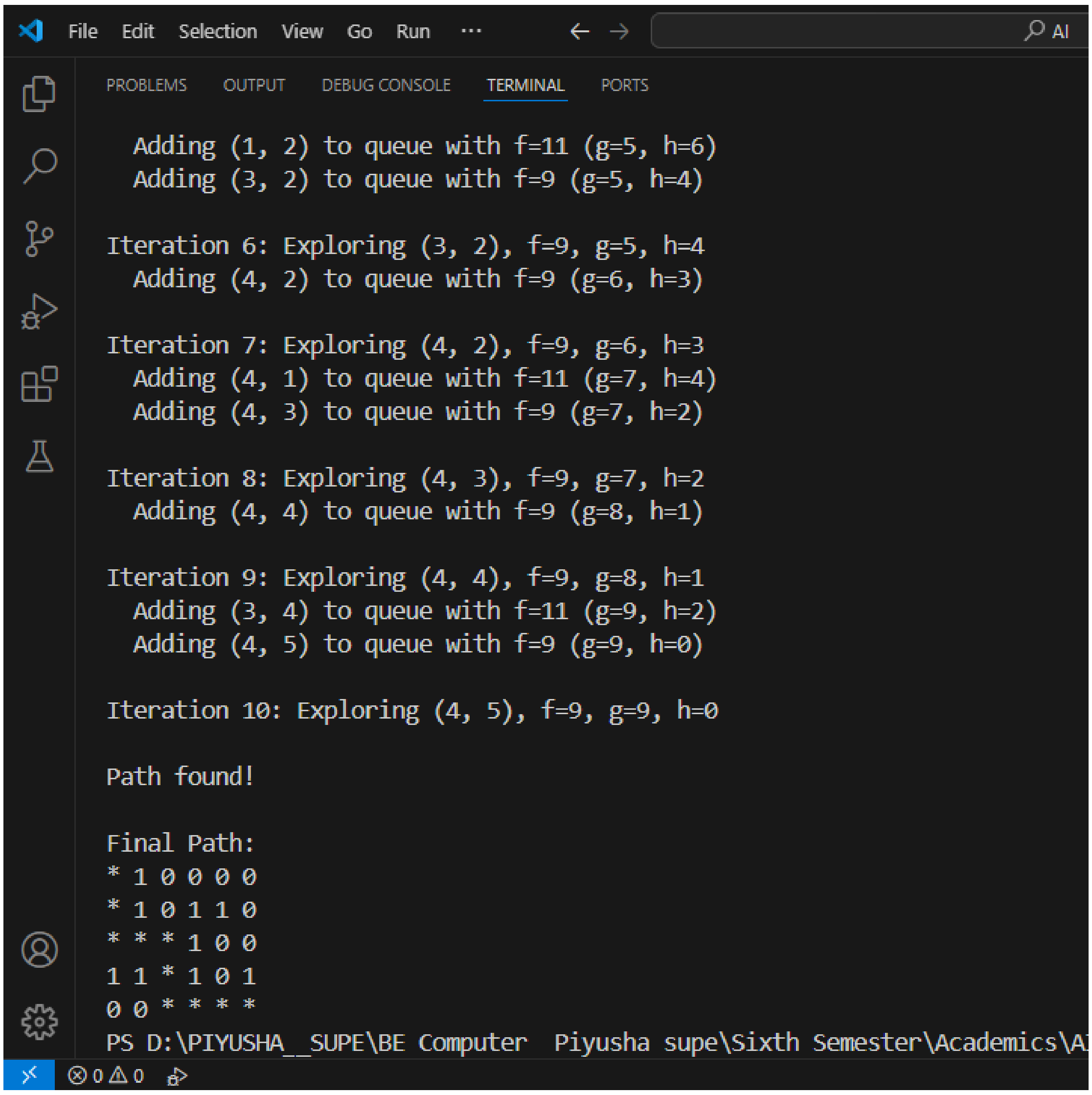
Use Cases:

- Pathfinding in games and robotics
- Puzzle solving (8-puzzle, 15-puzzle)
- Navigation systems (like GPS)
- AI decision-making

In conclusion, A* search is a powerful informed search algorithm widely used in AI and game development. Its effectiveness depends on the quality of the heuristic function, making it an excellent tool when we have domain-specific knowledge.

Program:

```
import heapq


class Node:
    def __init__(self, position, parent=None, g=0, h=0):
        self.position = position  # (row, col)
        self.parent = parent  # Parent node
        self.g = g  # Cost from start
        self.h = h  # Heuristic cost
        self.f = g + h  # Total cost

    def __lt__(self, other):
        return self.f < other.f  # Priority queue based on f-score

def heuristic(a, b):
    """Calculate Manhattan distance heuristic."""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar_search(maze, start, goal):
    """Perform A* search on the maze."""
    open_list = []  # Priority queue
    closed_set = set()  # Visited nodes

    # Start node
    start_node = Node(start, None, 0, heuristic(start, goal))
    heapq.heappush(open_list, start_node)

    # Movement directions (Up, Down, Left, Right)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    iteration = 0  # Step counter

    while open_list:
        iteration += 1
        current_node = heapq.heappop(open_list)  # Node with lowest f-score

        print(f"\nIteration {iteration}: Exploring {current_node.position}, f={current_node.f},
g={current_node.g}, h={current_node.h}")

        if current_node.position == goal:
            print("\nPath found!")
            return reconstruct_path(current_node)

        closed_set.add(current_node.position)

        for direction in directions:
            new_pos = (current_node.position[0] + direction[0], current_node.position[1] +
direction[1])

            if not (0 <= new_pos[0] < len(maze) and 0 <= new_pos[1] < len(maze[0])):
                continue  # Skip out-of-bounds

            if maze[new_pos[0]][new_pos[1]] == 1 or new_pos in closed_set:
                continue  # Skip walls and visited nodes

            g_new = current_node.g + 1
            h_new = heuristic(new_pos, goal)
            new_node = Node(new_pos, current_node, g_new, h_new)
            heapq.heappush(open_list, new_node)
                print(f"  Adding {new_pos} to queue with f={new_node.f} (g={new_node.g}, h={new_node.h})")

    print("\nNo path found!")
    return None


def reconstruct_path(node):
    """Reconstruct the path from goal to start."""
    path = []
    while node:
        path.append(node.position)
        node = node.parent
    return path[::-1]  # Reverse the path


def print_maze_with_path(maze, path):
    """Print the maze with the found path."""
    maze_copy = [row[:] for row in maze]  # Create a copy to modify
    for r, c in path:
        maze_copy[r][c] = "*"

    for row in maze_copy:
        print(" ".join(str(cell) for cell in row))


# Define the maze (0 = open path, 1 = wall)
maze = [
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0, 0],
    [1, 1, 0, 1, 0, 1],
    [0, 0, 0, 0, 0, 0],
]
# Start and Goal positions
start = (0, 0)
goal = (4, 5)
# Run A* Search
path = astar_search(maze, start, goal)
# Print final path
if path:
    print("\nFinal Path:")
    print_maze_with_path(maze, path)
```



```
  Adding (1, 2) to queue with f=11 (g=5, h=6)
  Adding (3, 2) to queue with f=9 (g=5, h=4)

Iteration 6: Exploring (3, 2), f=9, g=5, h=4
  Adding (4, 2) to queue with f=9 (g=6, h=3)

Iteration 7: Exploring (4, 2), f=9, g=6, h=3
  Adding (4, 1) to queue with f=11 (g=7, h=4)
  Adding (4, 3) to queue with f=9 (g=7, h=2)

Iteration 8: Exploring (4, 3), f=9, g=7, h=2
  Adding (4, 4) to queue with f=9 (g=8, h=1)

Iteration 9: Exploring (4, 4), f=9, g=8, h=1
  Adding (3, 4) to queue with f=11 (g=9, h=2)
  Adding (4, 5) to queue with f=9 (g=9, h=0)

Iteration 10: Exploring (4, 5), f=9, g=9, h=0

Path found!

Final Path:
* 1 0 0 0 0
* 1 0 1 1 0
* * * 1 0 0
1 1 * 1 0 1
0 0 * * * *
PS D:\PIYUSHA__SUPE\BE Computer  Piyusha supe\Sixth Semester\Academics\AI
```

Practical 03

Title of the Assignment: Greedy algorithms

Problem Statement: Implement Greedy search algorithm for any of the following application:
I. Selection Sort
II. Minimum Spanning Tree
III. Single-Source Shortest Path Problem
IV. Job Scheduling Problem
V. Prim's Minimal Spanning Tree Algorithm
VI. Kruskal's Minimal Spanning Tree Algorithm
VII. Dijkstra's Minimal Spanning Tree Algorithm

Objective of the Assignment: Implement greedy algorithms

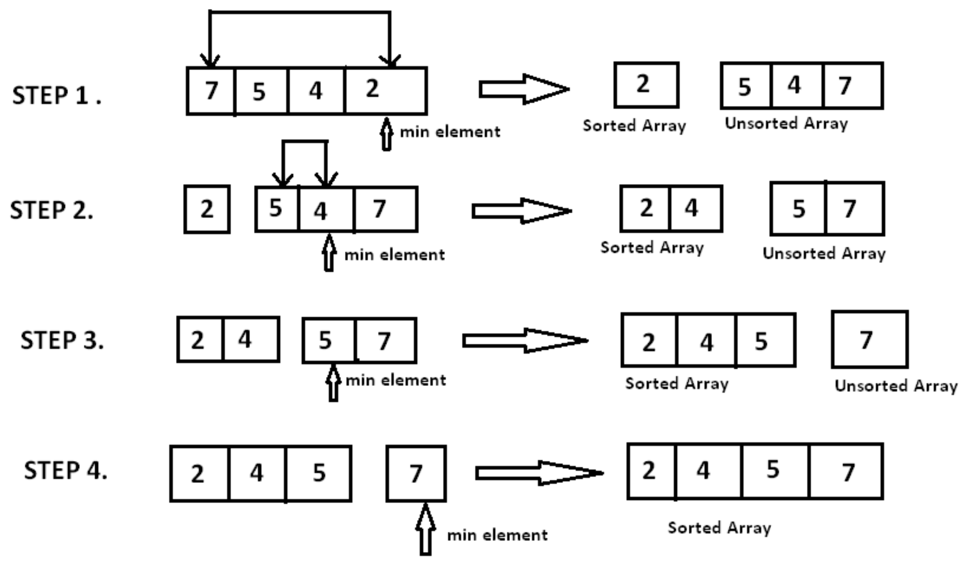
Prerequisite: Working of greedy strategies

Theory:

Greedy Algorithms are a class of algorithms that make the **locally optimal choice at each step**, hoping that these choices will lead to a **globally optimal solution**.

In the context of sorting, **Selection Sort** is a classic example of a greedy algorithm. The idea is to **repeatedly select the smallest (or largest) element** from the unsorted portion and place it at the correct position in the sorted portion.

Selection Sort works on the principle of **minimizing cost at each step** — it greedily selects the smallest element from the remaining array and places it at the front.



Algorithm Stepwise

Selection Sort Algorithm

1. Start with the first element of the array (index i = 0).
2. Assume it is the minimum element.
3. Loop through the rest of the array to find the actual minimum element.
4. Once found, swap it with the current index element (i).
5. Move to the next index (i = i + 1) and repeat until the array is sorted.

Pseudocode:

```
SelectionSort(array):
    for i = 0 to length(array) - 1:
        min_index = i
        for j = i + 1 to length(array) - 1:
            if array[j] < array[min_index]:
                min_index = j
        swap(array[i], array[min_index])
```

Conclusion / Analysis

- **Selection Sort** is a simple yet powerful example of a greedy algorithm.
- At each step, it selects the smallest remaining element — a greedy choice — and places it in its correct position.
- It performs well on small datasets but is inefficient on large datasets due to its **O(n^2)** time complexity.
- **Advantages**:
  - Easy to understand and implement.
  - Does not require extra memory (in-place sorting).
- **Disadvantages**:
  - Inefficient for large data.
  - The number of comparisons remains high even if the array is already sorted.

Use Cases:

- Useful in teaching sorting concepts.

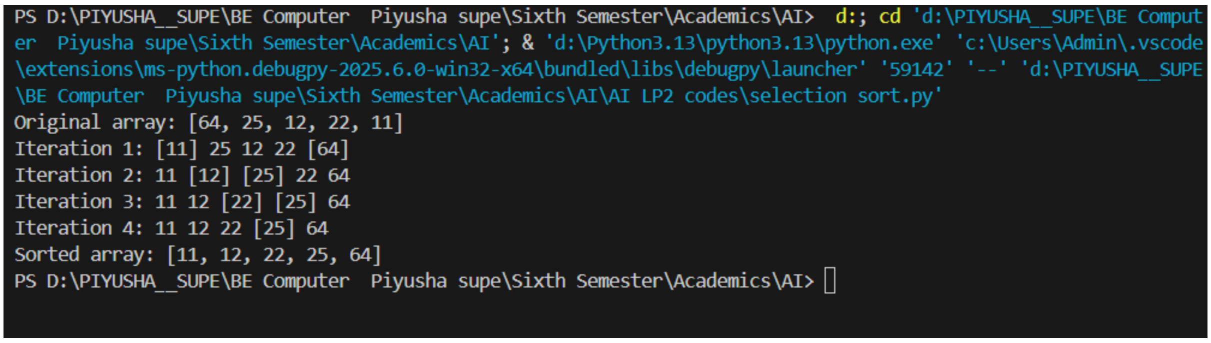
- Works well when memory write operations are expensive (since it performs minimal swaps).

In conclusion, Selection Sort is a great example of applying the greedy approach — making the best choice at each step with the hope that it leads to an optimal overall solution.

Program:

```
def print_array(arr, min_idx, i):
    """Prints the array with brackets around the selected elements."""
    result = []
    for idx, num in enumerate(arr):
        if idx == min_idx or idx == i:
            result.append(f"[{num}]")  # Highlight selected elements
        else:
            result.append(str(num))
    print(" ".join(result))


def selection_sort(arr):
    """Performs selection sort and displays each iteration."""
    n = len(arr)
    for i in range(n - 1):
        min_idx = i
        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]

        # Print the array after swapping
        print(f"Iteration {i + 1}: ", end="")
        print_array(arr, min_idx, i)
```

```
# Example usage

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

print("Original array:", arr)

selection_sort(arr)

print("Sorted array:", arr)
```



```
PS D:\PIYUSHA__SUPE\BE Computer  Piyusha supe\Sixth Semester\Academics\AI>  d:; cd 'd:\PIYUSHA__SUPE\BE Comput
er  Piyusha supe\Sixth Semester\Academics\AI'; & 'd:\Python3.13\python3.13\python.exe' 'c:\Users\Admin\.vscode
\extensions\ms-python.debugpy-2025.6.0-win32-x64\bundled\libs\debugpy\launcher' '59142' '--' 'd:\PIYUSHA__SUPE
\BE Computer  Piyusha supe\Sixth Semester\Academics\AI\AI LP2 codes\selection sort.py'
Original array: [64, 25, 12, 22, 11]
Iteration 1: [11] 25 12 22 [64]
Iteration 2: 11 [12] [25] 22 64
Iteration 3: 11 12 [22] [25] 64
Iteration 4: 11 12 22 [25] 64
Sorted array: [11, 12, 22, 25, 64]
PS D:\PIYUSHA__SUPE\BE Computer  Piyusha supe\Sixth Semester\Academics\AI>
```

Practical 04

Title of the Assignment: Constraint Satisfaction problem

Problem Statement: Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph colouring problem

Objective of the Assignment: Using Constraint satisfaction problem

Prerequisite: Concept of CSP, backtracking

Theory

A **Constraint Satisfaction Problem (CSP)** is a mathematical problem defined by a set of **variables**, **domains** for each variable, and **constraints** that specify allowable combinations of values.

The **N-Queens Problem** is a classic example of a CSP:

- Place N queens on an N × N chessboard such that **no two queens attack each other**.
- Constraints:
  - No two queens should be in the same row.
  - No two queens should be in the same column.
  - No two queens should be on the same diagonal.

To solve CSPs like the N-Queens problem, we can use **Backtracking** and **Branch and Bound**:

1. **Backtracking** is a depth-first search algorithm that incrementally builds candidates and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly lead to a valid solution.
2. **Branch and Bound** improves on backtracking by discarding subtrees using cost estimates (bounds), reducing the number of explored nodes.

Algorithm Stepwise

Backtracking Algorithm for N-Queens

1. Place a queen in the first row, first column.

2. Move to the next row:

   a. Try placing a queen in each column (left to right).

   b. For each position, check if it’s safe (i.e., not under attack).

   c. If safe, place the queen and move to the next row.

d. If not safe or no valid column, backtrack to previous row and try the next column.

3. Repeat until all N queens are placed.

4. If a complete configuration is found, print the solution.

5. If no configuration leads to a solution, report failure.

Branch and Bound Improvement

- Maintain auxiliary arrays:
    - cols[i] — whether a queen is placed in column i.
    - diag1[i] — for / diagonals.
    - diag2[i] — for \ diagonals.
- Before placing a queen, check if the position is already blocked via these arrays.
- Update these bounds while placing and removing queens to reduce unnecessary checks.

Conclusion / Analysis

- **Backtracking** provides a basic but effective solution for CSPs like N-Queens. It explores all possibilities and backtracks when a conflict occurs.
- **Branch and Bound** reduces the number of unnecessary recursive calls by pruning invalid branches earlier based on constraints.
- For **N-Queens**, both methods will eventually lead to a solution, but branch and bound is more efficient, especially for larger values of N.

Time Complexity:

- Worst-case time for backtracking is **O(N!)** due to permutation possibilities.
- Branch and Bound can reduce this by cutting branches early.

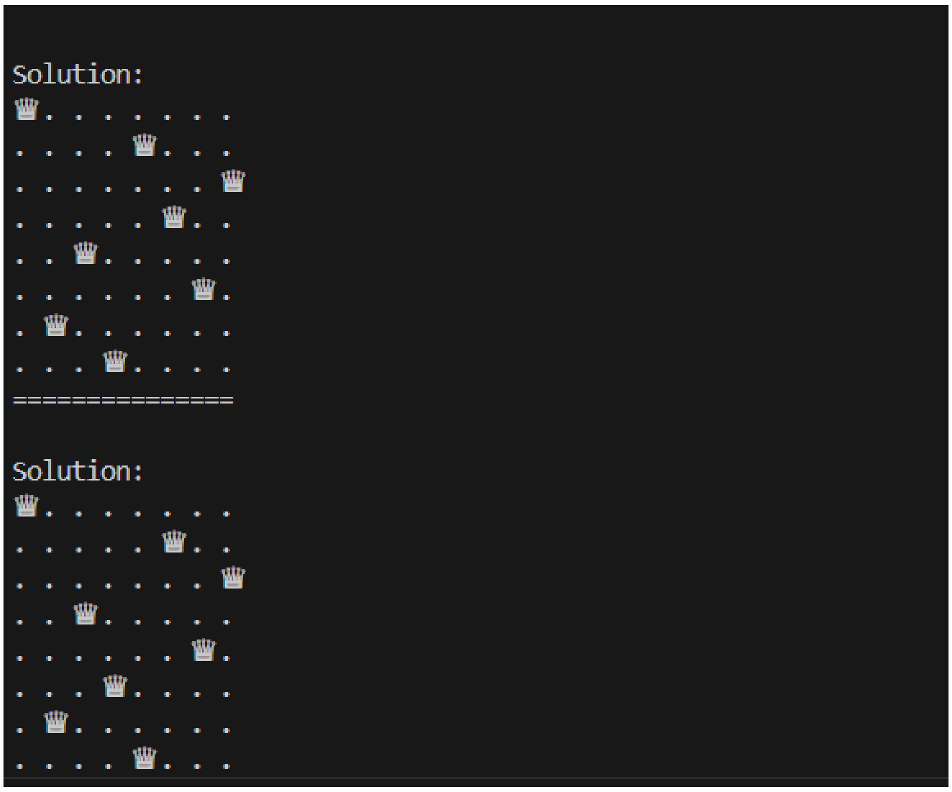
Use Cases:

- CSPs such as scheduling problems, puzzle solvers (Sudoku), and resource allocation.
- Graph colouring, map colouring, and register allocation in compilers.

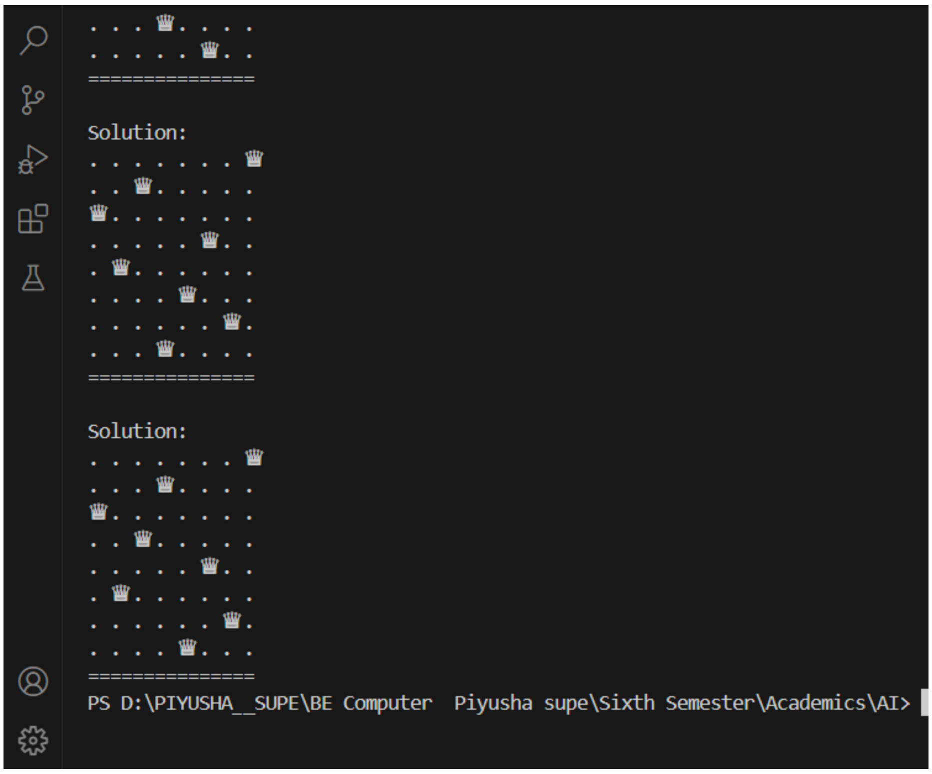
In conclusion, the combination of **backtracking** and **branch and bound** forms a powerful toolset for solving constraint satisfaction problems efficiently, as demonstrated through the N-Queens problem

Program:

```
def is_safe(board, row, col, n):
    """Check if a queen can be placed safely at board[row][col]"""
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):
            return False
    return True

def solve_n_queens(n, row=0, board=[]):
    """Recursive function to solve N-Queens using backtracking"""
    if row == n:
        print_solution(board, n)
        return True  # Return True to find only one solution
    for col in range(n):
        if is_safe(board, row, col, n):
            solve_n_queens(n, row + 1, board + [col])

def print_solution(board, n):
    """Print the board with emojis for better visualization"""
    print("\nSolution:")
    for row in range(n):
        line = ["♛" if board[row] == col else "." for col in range(n)]
        print(" ".join(line))
    print("=" * (2 * n - 1))  # Separator for clarity

# Run for 8-Queens (change N for different sizes)
N = 8
solve_n_queens(N)
```



```
Solution:
♛ . . . . . . .
. . . . ♛ . . .
. . . . . . . ♛
. . . . . ♛ . .
. . ♛ . . . . .
. . . . . . ♛ .
. ♛ . . . . . .
. . . ♛ . . . .
================

Solution:
♛ . . . . . . .
. . . . . ♛ . .
. . . . . . . ♛
. . ♛ . . . . .
. . . . . . ♛ .
. . . ♛ . . . .
. ♛ . . . . . .
. . . . ♛ . . .
```



```
. . . ♛ . . . .
. . . . . ♛ . .
================

Solution:
. . . . . . . ♛
. . ♛ . . . . .
♛ . . . . . . .
. . . . . ♛ . .
. ♛ . . . . . .
. . . . ♛ . . .
. . . . . . ♛ .
. . . ♛ . . . .
================

Solution:
. . . . . . . ♛
. . . ♛ . . . .
♛ . . . . . . .
. . ♛ . . . . .
. . . . . ♛ . .
. ♛ . . . . . .
. . . . . . ♛ .
. . . . ♛ . . .
================
PS D:\PIYUSHA__SUPE\BE Computer  Piyusha supe\Sixth Semester\Academics\AI>
```

Practical 05

Title of the Assignment: Application of AI

Problem Statement: Develop an elementary chatboat for any suitable customer interaction application.

Objective of the Assignment: Implement the AI concepts in an application

Prerequisite: Basic concepts of AI, Python

Theory

Artificial Intelligence (AI) aims to simulate human intelligence in machines, enabling them to think, learn, and interact intelligently. One practical application of AI is in **chatbots**—software systems that simulate human-like conversation with users.

A **chatbot** for customer interaction uses **Natural Language Processing (NLP)** to understand user queries and respond accordingly. Basic chatbots operate using **pattern matching and rule-based responses**, while more advanced ones use **machine learning models**.

For an elementary chatbot:

- We use **Python** and simple **if-else conditions**, keyword matching, or regular expressions.
- The chatbot can be tailored for a specific domain (e.g., FAQs for an online store, hotel booking, technical support).

Algorithm Stepwise

Basic Rule-Based Chatbot Algorithm

1. Start the chatbot.
2. Display a welcome message and instructions.
3. Accept user input (query/message).
4. Preprocess input (convert to lowercase, remove unnecessary characters).
5. Check for keywords or patterns in the user input.
6. Respond with predefined answers based on recognized keywords.
7. If input is not recognized, reply with a default message.
8. Repeat until user types "exit" or "bye".

- This elementary chatbot demonstrates the practical application of **AI concepts** in real-world problems.
- It uses basic **NLP techniques** and rule-based logic to interact with users.
- While simple, such chatbots can greatly improve **customer service efficiency** by handling repetitive queries.
- **Limitations**:
    - Cannot understand complex or unstructured input.
    - Lacks learning ability (no machine learning).
    - Limited to predefined responses.

Future Extensions:

- Integrate with **Dialogflow** or **Rasa** for more advanced conversations.
- Add **machine learning** or **intent classification** using libraries like NLTK or spaCy.
- Deploy using a **Flask web interface** or integrate into a **messaging platform** like WhatsApp or Telegram.

In conclusion, building a basic chatbot helps students grasp the foundational concepts of AI in a hands-on way and sets the stage for developing more intelligent systems in the future.

Program:

```
from flask import Flask, request, jsonify, render_template
from datetime import datetime


app = Flask(__name__)


@app.route('/')
def index():
    return render_template('index.html')


@app.route('/chat', methods=['POST'])
def chat():
    user_message = request.json.get('message', '').lower()


    def generate_response(msg):
        if "whale" in msg:
            return (
                "Whales are magnificent marine mammals. The blue whale is the largest animal "
                "to have ever lived on Earth—reaching up to 100 feet in length!"
            )
        elif "cpp" in msg or "c++" in msg:
            return (
                "C++ is a powerful programming language widely used for system/software development, "
                "game engines, and performance-critical applications. It supports OOP and low-level memory manipulation."
            )
        elif "engineering" in msg:
            return (
                "Engineering is the application of science and math to solve real-world problems. "
                "There are many branches: mechanical, electrical, civil, software, and more!"
            )
        elif "python" in msg:
            return (
                "Python is a high-level, interpreted language known for its readability and vast ecosystem. "
                "Great for web development, data science, automation, and more."
            )
        elif "ai" in msg or "artificial intelligence" in msg:
            return (
                "AI stands for Artificial Intelligence. It refers to machines or software mimicking human intelligence—"
                "like learning, reasoning, problem-solving, and understanding language!"
            )
        elif "hello" in msg or "hi" in msg:
            return "Hey there! 👋 How can I assist you today?"
        elif "time" in msg:
            return f"The current time is {datetime.now().strftime('%H:%M:%S')}."
        elif "joke" in msg:
            return "Why do programmers prefer dark mode? Because light attracts bugs. 😄"
        else:
            return (
                "That's interesting! I'm still learning, but I'd love to help. "
                "Try asking me about whales, C++, Python, engineering, AI, or even a joke!"
            )

    bot_response = generate_response(user_message)
    return jsonify({'response': bot_response})

if __name__ == '__main__':
    app.run(debug=True)
```

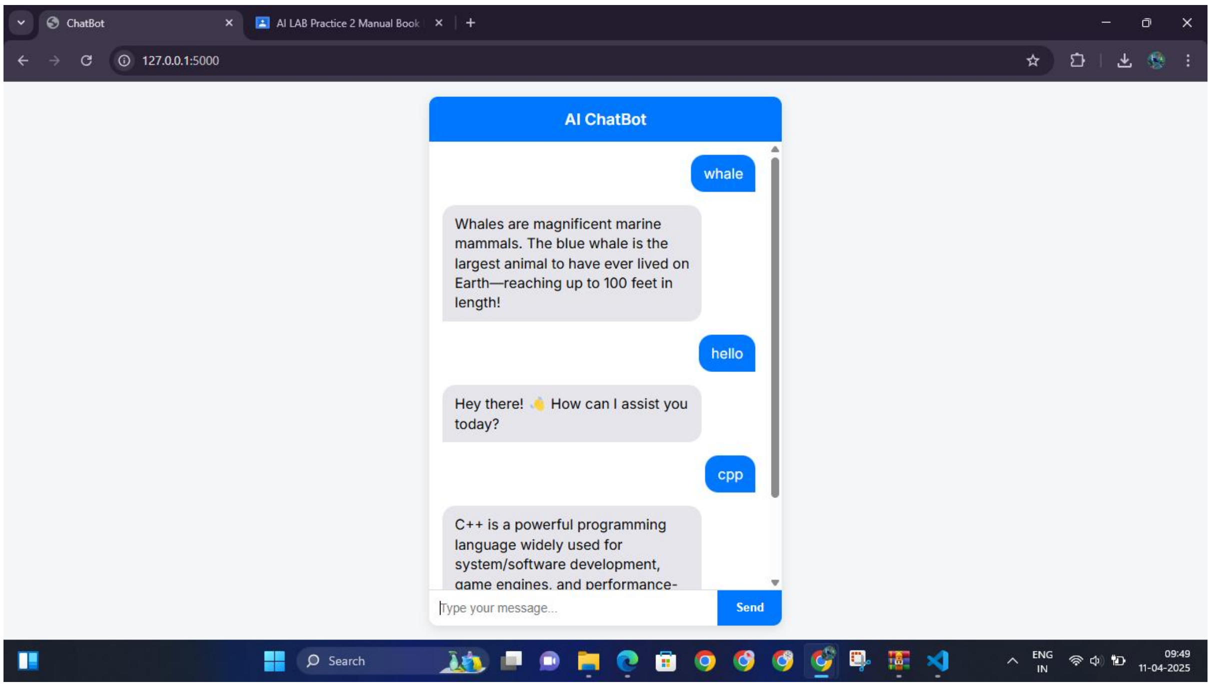
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>ChatBot</title>
    <link
href="https://fonts.googleapis.com/css2?family=Inter:wght@400;600&display=swap"
rel="stylesheet">
    <style>
        body {
            font-family: 'Inter', sans-serif;
            background: #f4f6f8;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            margin: 0;
        }

        .chat-container {
            width: 400px;
            height: 600px;
            background: white;
            border-radius: 10px;
            box-shadow: 0 4px 12px rgba(0, 0, 0, 0.1);
            display: flex;
            flex-direction: column;
            overflow: hidden;
        }

        .chat-header {
            background-color: #0077ff;
            padding: 15px;
            color: white;
            font-size: 18px;
            font-weight: 600;
            text-align: center;
        }

        .chat-box {
            flex: 1;
            padding: 15px;
            overflow-y: auto;
            display: flex;
            flex-direction: column;
        }
```

```
.message {
   margin-bottom: 15px;
   max-width: 75%;
   padding: 10px 14px;
   border-radius: 15px;
   line-height: 1.4;
}

.user-message {
   align-self: flex-end;
   background-color: #0077ff;
   color: white;
   border-bottom-right-radius: 0;
}

.bot-message {
   align-self: flex-start;
   background-color: #e5e5ea;
   color: black;
   border-bottom-left-radius: 0;
}

.chat-input {
   display: flex;
   border-top: 1px solid #ddd;
}

input[type="text"] {
   flex: 1;
   padding: 12px;
```

```
            border: none;
            outline: none;
            font-size: 14px;
        }

        button {
            background-color: #0077ff;
            border: none;
            color: white;
            padding: 0 20px;
            cursor: pointer;
            font-weight: 600;
        }

        button:hover {
            background-color: #005fd1;
        }
    </style>
</head>
<body>
    <div class="chat-container">
        <div class="chat-header">AI ChatBot</div>
        <div class="chat-box" id="chat-box"></div>
        <div class="chat-input">
            <input type="text" id="user-input" placeholder="Type your message..."
autocomplete="off">
            <button onclick="sendMessage()">Send</button>
        </div>
    </div>

    <script>
```

```
const chatBox = document.getElementById('chat-box');
const userInput = document.getElementById('user-input');

function appendMessage(message, className) {
    const msg = document.createElement('div');
    msg.className = `message ${className}`;
    msg.textContent = message;
    chatBox.appendChild(msg);
    chatBox.scrollTop = chatBox.scrollHeight;
}

async function sendMessage() {
    const message = userInput.value.trim();
    if (!message) return;

    appendMessage(message, 'user-message');
    userInput.value = '';

    const response = await fetch('/chat', {
        method: 'POST',
        headers: {'Content-Type': 'application/json'},
        body: JSON.stringify({ message })
    });

    const data = await response.json();
    appendMessage(data.response, 'bot-message');
}

// Optional: Send message with Enter key
userInput.addEventListener('keypress', function (e) {
```

```
            if (e.key === 'Enter') sendMessage();
        });
    </script>
</body>
</html>
```



AI ChatBot

whale

Whales are magnificent marine mammals. The blue whale is the largest animal to have ever lived on Earth—reaching up to 100 feet in length!

hello

Hey there! 👋 How can I assist you today?

cpp

C++ is a powerful programming language widely used for system/software development, game engines, and performance-

Practical 06

Title of the Assignment: Application of AI

Problem Statement: Implement any one of the following Expert System
I. Information management
II. Hospitals and medical facilities
III. Help desks management
IV. Employee performance evaluation
V. Stock market trading
VI. Airline scheduling and cargo schedules

Objective of the Assignment: Implement a real scenario

Prerequisite: Python

Theory:

An **Expert System** is a computer-based application designed to simulate the decision-making ability of a human expert. It uses a **knowledge base** (rules/facts) and an **inference engine** (logic) to derive conclusions based on user input.

In the **airline industry**, expert systems can:

- Automate **flight and cargo scheduling**
- Optimize **resource allocation**
- Minimize **human error**
- Provide **decision support** to airline staff

This system is a **rule-based expert system**, where flight schedules are determined based on:

- **Flight type** (Passenger or Cargo)
- **Destination**
- **Cargo weight**
- **Preferred time slot**

The system checks these inputs against predefined rules and assigns an appropriate flight schedule or suggests alternatives.

Algorithm (Stepwise):

1. **Start the system**
2. Ask the user to:

   - Choose flight type
   - Enter destination
   - Input cargo weight
   - Select preferred time slot (Morning/Afternoon/Evening)
3. Validate inputs
4. Match the inputs against rule-based conditions in the knowledge base
   - Check cargo weight limits
   - Match available time slots
5. If matched:
   - Assign a flight schedule
   - Display all details clearly
6. If mismatched:
   - Reject the scheduling with explanation
   - Provide advice/suggestions
7. End the system or allow user to try again

Advantages:

- 24x7 automation with no manual intervention required.
- Consistent and predictable outputs.
- Fast decision-making and scheduling.
- Easily extendable to include additional rules or constraints.

Limitations:

- Cannot handle highly dynamic or rare logistical constraints.
- No learning or optimization capability as it is not based on machine learning.
- Does not take into account external systems like weather, maintenance, or human resource availability.

Future Enhancements:

1. Add a graphical or web-based user interface using frameworks like Tkinter, Flask, or Streamlit.
2. Integrate machine learning models for predictive scheduling and demand forecasting.

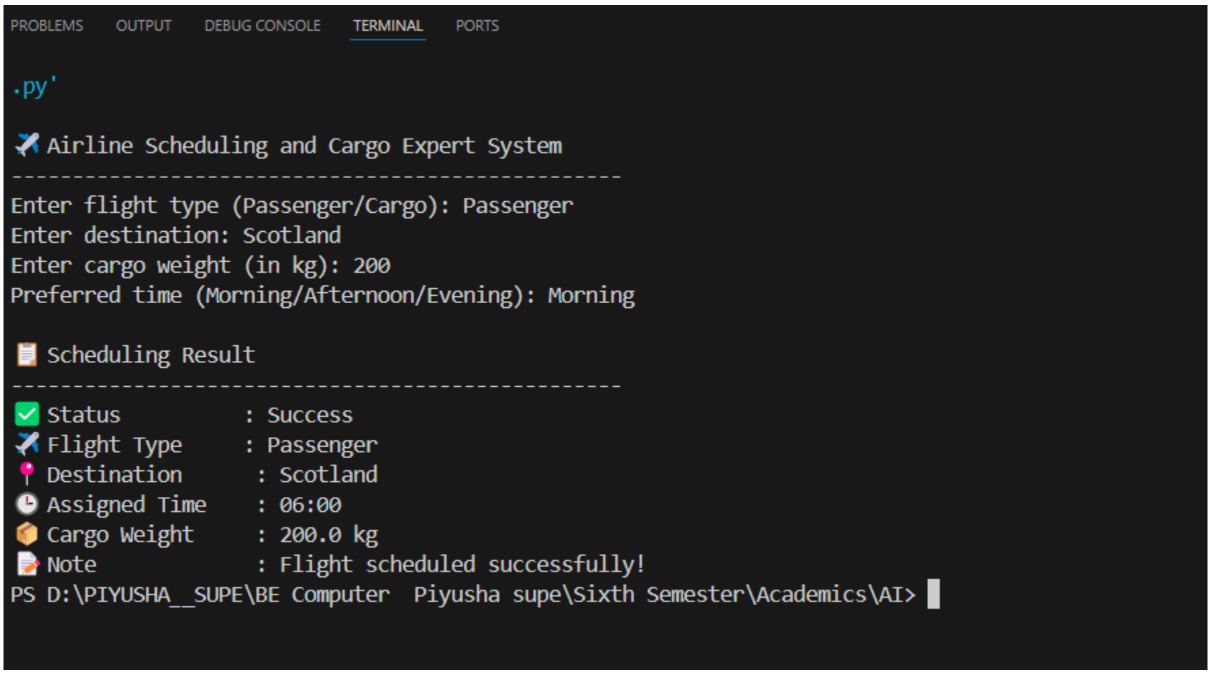
3. Include real-time integration with APIs for weather updates, traffic control, or cargo tracking.
4. Support dynamic rule loading from configuration files or connected databases.
5. Add regulatory compliance checks for international operations.

Conclusion / Analysis:

- This expert system replicates the decision-making process of an airline scheduler using a **simple rule-based structure**.
- It is designed to be **fast, reliable, and scalable**, which is useful for small airports, cargo operators, or airline training systems.
- The project introduces learners to **knowledge representation**, **logical rules**, and **AI applications in logistics and aviation**.

Program:

```
# Airline Scheduling and Cargo Expert System


def get_flight_schedule(flight_type, destination, cargo_weight, preferred_time):
    schedules = {
        "Morning": ["06:00", "07:30", "09:00"],
        "Afternoon": ["12:00", "14:00", "15:30"],
        "Evening": ["18:00", "19:30", "21:00"]
    }

    cargo_limits = {
        "Passenger": 200,   # in kg
        "Cargo": 1000       # in kg
    }

    # Rule 1: Validate Cargo Weight
    if cargo_weight > cargo_limits[flight_type]:
        return {
            "status": "Rejected",
            "reason": f"Cargo exceeds limit for {flight_type} flight ({cargo_limits[flight_type]}
kg)",
            "suggestion": "Switch to Cargo flight or reduce cargo weight."
        }

    # Rule 2: Match Preferred Time
    time_slots = schedules.get(preferred_time, [])

    if not time_slots:
        return {
            "status": "Failed",
            "reason": "Invalid preferred time slot.",
            "suggestion": "Choose Morning, Afternoon, or Evening."
        }

    # Rule 3: Assign First Available Slot
    assigned_slot = time_slots[0]

    return {
        "status": "Success",
        "flight_type": flight_type,
        "assigned_time": assigned_slot,
        "destination": destination,
        "cargo_weight": cargo_weight,
        "note": "Flight scheduled successfully!"
    }

# ---- MAIN PROGRAM ----
def main():
    print("\n✈  Airline Scheduling and Cargo Expert System")
    print("-----------------------------------------------")
    flight_type = input("Enter flight type (Passenger/Cargo): ").strip().title()
    destination = input("Enter destination: ").strip().title()
    cargo_weight = float(input("Enter cargo weight (in kg): "))
    preferred_time = input("Preferred time (Morning/Afternoon/Evening): ").strip().title()

    result = get_flight_schedule(flight_type, destination, cargo_weight, preferred_time)

    print("\n📋 Scheduling Result")
    print("-------------------------------------------------")
    if result["status"] == "Success":
        print(f"✅ Status          : {result['status']}")
        print(f"✈️  Flight Type     : {result['flight_type']}")
        print(f"📍 Destination     : {result['destination']}")
        print(f"🕒 Assigned Time   : {result['assigned_time']}")
        print(f"📦 Cargo Weight    : {result['cargo_weight']} kg")
        print(f"📝 Note            : {result['note']}")
    else:
        print(f"❌ Status          : {result['status']}")
        print(f"⚠️  Reason          : {result['reason']}")
        print(f"💡 Suggestion      : {result['suggestion']}")

if __name__ == "__main__":
    main()
```



```
.py'

✈️ Airline Scheduling and Cargo Expert System
--------------------------------------------------
Enter flight type (Passenger/Cargo): Passenger
Enter destination: Scotland
Enter cargo weight (in kg): 200
Preferred time (Morning/Afternoon/Evening): Morning

📋 Scheduling Result
--------------------------------------------------
✅ Status           : Success
✈️ Flight Type      : Passenger
📍 Destination       : Scotland
🕒 Assigned Time     : 06:00
📦 Cargo Weight      : 200.0 kg
📝 Note              : Flight scheduled successfully!
PS D:\PIYUSHA__SUPE\BE Computer  Piyusha supe\Sixth Semester\Academics\AI>
```