LAB 2: UNINFORMED SEARCH ALGORITHMS

2.1 Aim

To understand problem formulation and implement uninformed search strategies.

2.2 Theory Overview

A search problem consists of:

- Initial State
- Actions
- Transition Model
- Goal Test
- Path Cost

Uninformed Search Algorithms:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Depth-Limited Search
- Iterative Deepening Search

Experiment 1: Breadth-First Search (BFS) for Maze Navigation

Problem Statement:

Consider a maze represented as a **graph**, where each node corresponds to a location in the maze and edges represent valid paths between locations.

An agent is placed at a **start node** and must reach a **goal node** by traversing the maze.

Task:

1. Formulate the maze navigation problem as a **search problem** by clearly defining:
    - State space
    - Initial state
    - Actions
    - Transition model
    - Goal test
    - Path cost
2. Implement the **Breadth-First Search (BFS)** algorithm to find the shortest path from the start node to the goal node.
3. Print the path returned by BFS.
4. Count and display the number of nodes expanded during the search.

Constraints:

- The maze is **unweighted**.
- The environment is **fully observable, deterministic, static, discrete, and single-agent**.
- BFS must be implemented **without using any built-in graph search libraries**.

```python
from collections import deque

def bfs(graph, start, goal):
     # Queue stores paths, not just nodes
    queue = deque()
    queue.append([start])

    # Visited set to avoid revisiting nodes
    visited = set()

    # Counter for expanded nodes
    nodes_expanded = 0

    while queue:
        # Remove the first path from the queue (FIFO)
        path = queue.popleft()
        current_node = path[-1]

        # Goal test
        if current_node == goal:
            return path, nodes_expanded

        # Expand node only if not visited
        if current_node not in visited:
            visited.add(current_node)
            nodes_expanded += 1

            # Generate successor states
            for neighbor in graph[current_node]:
                new_path = path + [neighbor]
                queue.append(new_path)

    # If goal is not reachable
    return None, nodes_expanded
```

```python
maze = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E', 'G'],
    'G': []
}

start_node = 'A'
goal_node = 'G'

path, expanded = bfs(maze, start_node, goal_node)

print("Path found:", path)
print("Nodes expanded:", expanded)
```

Experiment 2: Depth-First Search (DFS) for Graph Traversal

Problem Statement

Given a connected, finite graph represented using an **adjacency list**, design and implement the **Depth-First Search (DFS)** algorithm to find a path from a specified **start node** to a **goal node**.

Task Requirements

1. Formulate the graph traversal problem by identifying:
    - State representation
    - Initial state
    - Goal state
    - Actions
2. Implement a **recursive DFS algorithm** that:

    - Explores one branch of the search tree as deeply as possible before backtracking
    - Avoids revisiting nodes already present in the current path
3. Return and display:
    - A valid path from the start node to the goal node (if it exists)

Constraints

- The graph is unweighted.
- DFS must be implemented **without using any built-in graph traversal libraries**.
- The algorithm is **not required** to return the shortest path.

Expected Outcome

The algorithm should successfully find a path from the start node to the goal node by exploring depth-wise, though the path may not be optimal.

Experiment 3: Solving the 8-Puzzle Problem Using BFS

Problem Statement:

The **8-puzzle problem** consists of a 3×3 grid containing eight numbered tiles and one blank space. A tile adjacent to the blank space can be moved into the blank position.

Given an **initial configuration** and a **goal configuration**, implement **Breadth-First Search (BFS)** to find a sequence of moves that transforms the initial state into the goal state.

Task Requirements:

1. Represent the puzzle state using a suitable data structure.
2. Define:
    - Initial state
    - Goal state
    - Valid actions (up, down, left, right)
    - Transition model
3. Implement BFS to explore the state space.
4. Output:
    - The sequence of states from the initial state to the goal state
    - The total number of states expanded during the search

Constraints:

- Use BFS to guarantee the shortest solution.
- Avoid revisiting previously explored states.
- Assume the given initial state is solvable.

Expected Outcome:

The algorithm should produce the shortest sequence of moves required to reach the goal configuration.

Experiment 4: Performance Comparison of BFS and DFS

Problem Statement:

Breadth-First Search (BFS) and Depth-First Search (DFS) exhibit different performance characteristics in terms of **time complexity**, **space complexity**, and **solution optimality**.

Design an experiment to **compare BFS and DFS** when applied to the **same search problem**.

Task Requirements:

1. Apply both BFS and DFS to a common problem instance (graph traversal or maze navigation).
2. Instrument both algorithms to record:
    - Number of nodes expanded
    - Execution time
    - Maximum memory used (queue or recursion depth)

3. Tabulate the results obtained for BFS and DFS.
4. Analyze and explain the observed differences.

Constraints:

- Use the same start and goal states for both algorithms.
- Run each algorithm under identical conditions.
- Use Python's time module for execution time measurement.

Expected Outcome:

Students should observe that:

- BFS guarantees the shortest path but uses more memory
- DFS uses less memory but may find non-optimal paths

Post-Exercise Analysis Questions

1. Why does BFS guarantee an optimal solution while DFS does not?
2. In which scenarios is DFS preferred over BFS?
3. How does the branching factor affect BFS performance?
4. Why is BFS considered impractical for large state spaces?