**Lab Assignment 1: Implementation of DFS and BFS for the 8-Puzzle Problem**

**1. Objective:**

The goal of this lab assignment is to implement **Depth-First Search (DFS)** and **Breadth-First Search (BFS)** algorithms to solve the 8-puzzle problem. The 8-puzzle is a sliding puzzle that consists of a 3x3 grid with 8 numbered tiles and one empty space. The objective is to rearrange the tiles to match a target configuration by sliding the tiles horizontally or vertically into the empty space.

**2. Problem Statement:**

Given an initial configuration of the 8-puzzle, implement DFS and BFS algorithms to find the sequence of moves required to reach the goal configuration. The puzzle's initial and goal states are represented as a 2D list, and both algorithms should explore the state space and return the path of moves leading to the solution.

**3. Theory:**

**3.1. The 8-Puzzle Problem:**

The 8-puzzle problem consists of a 3x3 grid with 8 numbered tiles and one blank space. The puzzle can be described by the following components:

* **State**: A particular arrangement of the tiles.
* **Action**: Moving one of the adjacent tiles into the blank space.
* **Goal State**: The target arrangement of tiles (typically the numbers 1 to 8 in order, with the blank in the last position).

An example of the initial and goal states is shown below:

* **Initial State:**

Copy code

1 2 3

4 5 6

7 8 \_

* **Goal State:**

Copy code

1 2 3

4 5 6

7 8 \_

The challenge is to find a sequence of actions (moves) that leads from the initial state to the goal state.

**3.2. Depth-First Search (DFS):**

* **Approach**: DFS explores as far as possible along each branch before backtracking. It uses a stack (LIFO) data structure, which allows it to explore deeper paths first.
* **Strategy**:
  + Start from the initial state.
  + Explore a child node by moving a tile into the empty space.
  + Continue exploring deeper until either the goal is found or all possibilities in that path are exhausted.
  + If a path is exhausted, backtrack to the previous node and try another path.
* **Pros**: DFS may require less memory, as it only needs to store the current path.
* **Cons**: It may get stuck in deep, irrelevant branches and might not find the shortest solution.

**3.3. Breadth-First Search (BFS):**

* **Approach**: BFS explores all nodes level by level. It uses a queue (FIFO) data structure, allowing it to explore the closest nodes first, ensuring the shortest path to the goal state.
* **Strategy**:
  + Start from the initial state.
  + Expand all possible moves from the current state before moving deeper into the next level.
  + Continue until the goal state is reached or all possibilities are exhausted.
* **Pros**: BFS guarantees finding the shortest solution if it exists.
* **Cons**: It may require more memory, as it must store all nodes at the current level before moving to the next.

**3.4. State Representation:**

Each state of the puzzle is represented as a 2D list (or array), where each element is either a tile number or a blank space (represented as 0 or "\_"). For example:

lua

Copy code

[[1, 2, 3],

[4, 5, 6],

[7, 8, 0]]

Here, 0 represents the blank space.

**3.5. State Transition:**

A move is defined as sliding a tile from one of the four possible directions (up, down, left, right) into the blank space. The state changes after each valid move.

**4. Algorithm Design:**

**4.1. DFS Algorithm for 8-Puzzle:**

1. Initialize the stack with the initial state.
2. While the stack is not empty:
   * Pop the top node (state) from the stack.
   * If it is the goal state, return the path.
   * Else, generate all valid children (new states) by sliding tiles into the blank space.
   * Push these children onto the stack.

**4.2. BFS Algorithm for 8-Puzzle:**

1. Initialize the queue with the initial state.
2. While the queue is not empty:
   * Dequeue the front node (state).
   * If it is the goal state, return the path.
   * Else, generate all valid children (new states) by sliding tiles into the blank space.
   * Enqueue these children.

**5. Pseudocode:**

**5.1. DFS Pseudocode:**

c

Copy code

DFS(initial\_state, goal\_state):

stack = [initial\_state]

visited = set()

while stack is not empty:

current\_state = stack.pop()

if current\_state == goal\_state:

return the solution path

if current\_state not in visited:

visited.add(current\_state)

for each child\_state generated from current\_state:

if child\_state is valid:

stack.push(child\_state)

**5.2. BFS Pseudocode:**

c

Copy code

BFS(initial\_state, goal\_state):

queue = [initial\_state]

visited = set()

while queue is not empty:

current\_state = queue.pop(0)

if current\_state == goal\_state:

return the solution path

if current\_state not in visited:

visited.add(current\_state)

for each child\_state generated from current\_state:

if child\_state is valid:

queue.append(child\_state)

**6. Expected Output:**

Both DFS and BFS will output a sequence of moves that transforms the initial state into the goal state. BFS will always return the shortest sequence, while DFS may return a solution that is not the shortest.

**7. Conclusion:**

In this lab, DFS and BFS are applied to solve the 8-puzzle problem. While both algorithms are capable of finding solutions, BFS is better suited for finding the optimal solution, whereas DFS might be faster in finding a solution but may not provide the shortest path.

**8. References:**

* Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.
* Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.