Assignment No. 1

1.Problem Statement:

Implement DFS, BFS for 8-puzzle problem.

2.Objectives:

The main objective is to implement an algorithm using Breadth-First Search (BFS) and Depth-First-Search (DFS) to solve the 8-puzzle problem. These algorithms are well-suited for this task because it systematically explores all possible configurations starting from the initial state and finds the shortest solution path to the goal state.

The aim is to:

• Develop a Python program that uses BFS and DFS to solve the puzzle.

• Ensure the algorithm prints each step of the solution, showing how the tiles move from the initial state to the goal state.

3. Theory:

The 8-puzzle problem is a classic combinatorial problem where the goal is to rearrange a set of numbered tiles in a 3x3 grid by sliding them into an empty space. The puzzle consists of eight numbered tiles and one empty space, and the task is to move the tiles into a specified "goal" configuration, starting from an arbitrary initial configuration. The problem is an example of a state-space search problem, where each configuration of the puzzle represents a unique state, and the allowable moves define state transitions.

State-Space Representation:

• Each state is a 3x3 matrix where 8 positions are filled with numbers (1 to 8), and one position is empty (represented by 0).

• The possible transitions between states occur when a tile adjacent to the empty space moves into that space.

For example, if the empty space is at position (2,2), the tiles adjacent to it at (1,2), (2,1), and (2,3) can slide into that space.

The problem can be solved using search algorithms, and the two major algorithms we focus on are:

1. Breadth-First Search (BFS).

2. Depth-First Search (DFS).

Breadth-First Search (BFS):

BFS is an uninformed search algorithm that explores all nodes level by level, starting from the root node (initial state) and systematically exploring all possible moves from the current state. It uses a queue to store the states that need to be explored.

Characteristics of BFS:

• Complete: BFS will always find a solution if one exists because it explores all possible configurations.

• Optimal: BFS guarantees that the first solution it finds is the shortest path to the goal, as it explores states layer by layer, ensuring that solutions at the shallowest depth are found first.

• Space and Time Complexity: BFS consumes more memory compared to DFS because it stores all possible states at each level. The time complexity grows exponentially as the depth of the solution increases.

BFS Search Tree:

• The search tree starts at the initial configuration (root) and expands by exploring all possible moves from the current state.

• Each valid move (up, down, left, or right) generates a new child state.

• The algorithm explores the states level by level, moving to the next level only when all the states at the current level have been explored.

Depth-First Search (DFS):

DFS is another uninformed search algorithm, but instead of exploring states level by level like BFS, it explores as deep as possible along each branch before backtracking. DFS uses a stack to keep track of the states being explored. DFS can also be implemented recursively.

Characteristics of DFS:

• Incomplete: Unlike BFS, DFS may not find a solution if the search goes down a branch infinitely or fails to backtrack properly.

• Non-optimal: DFS does not guarantee finding the shortest solution. If the solution lies deep in another part of the tree, DFS could waste time exploring irrelevant paths first.

• Space and Time Complexity: DFS is more memory efficient than BFS since it only stores the current path being explored. However, its time complexity can also grow exponentially if the search goes very deep without finding the solution.

DFS Search Tree:

• DFS explores one branch of the search tree as deep as possible before backtracking.

• It follows one possible path (sequence of moves) until no more valid moves are available. Then it backtracks to the last state where unexplored moves exist and continues the search.

• The search may end up in dead ends before finding the solution.

4.Methodology

The 8-puzzle problem can be approached using both BFS and DFS by treating it as a search problem where the puzzle configurations represent states, and sliding the tiles creates transitions between states. Both algorithms explore possible solutions by navigating the state space, but they do so differently in terms of depth and breadth of the search.

Step-by-Step Procedure for BFS and DFS:

1. Initial State Representation:

o The 3x3 grid is represented as a list or an array, where the number 0 represents the empty space.

o The goal state is a predefined arrangement of tiles that must be achieved through a series of moves.

2. State Transitions:

o The empty space (0) can move up, down, left, or right by swapping with an adjacent tile.

o Each move creates a new state (a new configuration of the grid).

3. Breadth-First Search (BFS) Implementation:

o Step 1: Start by initializing a queue with the initial state (the starting configuration of the puzzle).

o Step 2: Initialize an empty list of explored states to avoid re-exploring the same configurations.

o Step 3: While the queue is not empty, take the first state from the queue and check if it matches the goal state.

o Step 4: If it matches the goal, stop and return the solution.

o Step 5: If it doesn’t match, generate all possible new states by moving the empty tile in all valid directions (up, down, left, right).

o Step 6: Add the new states to the queue if they haven't been explored or queued yet.

o Step 7: Repeat until the goal state is found.

o BFS continues level by level until the shortest solution is found.

4. Depth-First Search (DFS) Implementation:

o Step 1: Start by initializing a stack with the initial state (starting configuration of the puzzle).

o Step 2: Initialize an empty list of explored states to avoid revisiting states.

o Step 3: While the stack is not empty, pop the last state from the stack and check if it matches the goal state.

o Step 4: If it matches, stop and return the solution.

o Step 5: If it doesn’t match, generate all possible new states by moving the empty tile (up, down, left, right).

o Step 6: Push the new states onto the stack if they haven’t been explored yet.

o Step 7: DFS explores as deep as possible along each branch before backtracking, continuing until the goal state is found or all paths are exhausted.

5. State Representation and Moves:

o Both BFS and DFS require identifying the current position of the empty tile (0).

o The valid moves are determined based on the position of the empty tile. For example, if the tile is at the top-left corner, only "down" and "right" moves are valid.

o The puzzle’s new state is generated by swapping the empty tile with the adjacent tile in the chosen direction.

6. Goal State Check:

o At each step of the BFS or DFS, the current state is compared with the goal state to check for a match.

o If a match is found, the sequence of moves that led to the solution is printed or returned.

5. Advantages:

BFS:

• Optimal: Guarantees the shortest path.

• Complete: Always finds a solution if one exists.

DFS:

• Low memory usage: DFS uses less memory since it only needs to store the current path (stack), not all nodes at the current level.

• Good for deep solutions: DFS can be more efficient if the solution lies deep in the search space.

6. Limitations:

BFS:

• Memory-intensive: Requires storing all nodes at the current depth level, which can be large for deep search spaces.

• Slow for deep solutions: BFS explores all shallow nodes before moving deeper, so deep solutions take time to reach.

DFS:

• Non-optimal: DFS may not find the shortest path.

• Gets stuck in deep paths: DFS might explore deep branches unnecessarily, leading to inefficiency.

• Risk of infinite recursion: Without safeguards, DFS can enter an infinite loop in some configurations.

7. Applications:

Both DFS and BFS have diverse applications:

• BFS: Useful in situations where the shortest path is required, like finding the shortest route in a map, solving puzzles where optimality is important.

• DFS: Useful in scenarios where memory is constrained, and deep exploration is prioritized, such as solving mazes, detecting cycles in graphs, and exploring decision trees.

8. Conclusion:

In this assignment, we have implemented a BFS and DFS for the 8-puzzle problem demonstrates the fundamental trade-offs between search strategies. BFS guarantees an optimal solution but consumes more memory, while DFS is more memory-efficient but may not always find the best solution. Understanding these algorithms helps us in selecting the appropriate search technique based on the problem's constraints, such as time, space, and optimality requirements.