**Assignment 1**

**Title: Implementation of BFS and DFS for 8-Puzzle Problem**

**Theory**

The **8-Puzzle Problem** is a classic example in Artificial Intelligence used to demonstrate **state space search**. It consists of a 3×3 grid with eight numbered tiles and one empty space (represented as 0). The goal is to reach a target arrangement of tiles by sliding them one at a time into the empty space.

Search algorithms such as **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** are commonly applied to solve this problem.

* **Breadth-First Search (BFS):**  
  An uninformed search algorithm that explores all nodes at the present depth before moving to nodes at the next level. It guarantees the shortest solution but requires more memory.
* **Depth-First Search (DFS):**  
  Explores as deep as possible along a branch before backtracking. It uses less memory but may not find the optimal solution.

The 8-Puzzle is an ideal problem for illustrating **state representation**, **transition functions**, and **search space traversal**.

**Definition**

The **8-Puzzle Problem** can be defined as:

* **State:** Arrangement of the 8 tiles and one empty space in a 3×3 grid.
* **Initial State:** The starting configuration provided by the user.
* **Goal State:** The target configuration (usually [1,2,3,4,5,6,7,8,0]).
* **Operators:** Move the empty tile *up, down, left,* or *right*, when possible.
* **Transition:** Each operator generates a new state from the current one.
* **Goal Test:** Check whether the current state matches the goal configuration.

**Explanation of Implementation**

The program implements BFS and DFS to solve the 8-Puzzle. It includes checking solvability, generating valid moves, and exploring the search space.

**Step 1: Solvability Check**

Before performing any search, the algorithm checks whether the puzzle configuration can be solved.  
This is determined by counting the number of **inversions** — pairs of tiles where a higher-numbered tile precedes a lower-numbered one.

*Pseudo code:*

FUNCTION isSolvable(state):

inversion ← 0

FOR i FROM 0 TO n-1:

FOR j FROM i+1 TO n-1:

IF state[i] > state[j] AND state[i] != 0 AND state[j] != 0:

inversion ← inversion + 1

RETURN (inversion MOD 2 == 0)

If the inversion count is even, the puzzle is solvable.

**Step 2: Move Generation**

The empty tile (represented by 0) can move in four possible directions if valid:  
**UP**, **DOWN**, **LEFT**, and **RIGHT**.

*Pseudo code:*

FUNCTION getMoves(state):

find index of empty tile

create empty list possibleMoves

IF move UP is valid → add new state to possibleMoves

IF move DOWN is valid → add new state to possibleMoves

IF move LEFT is valid → add new state to possibleMoves

IF move RIGHT is valid → add new state to possibleMoves

RETURN possibleMoves

Each generated state represents a new configuration after swapping the empty tile with a neighbor.

**Step 3: Breadth-First Search (BFS)**

BFS explores all configurations level by level until the goal state is found.

*Pseudo code:*

FUNCTION solveByBFS(start):

create empty queue Q

create empty list VISITED

enqueue start into Q

WHILE Q is not empty:

state ← dequeue from Q

add state to VISITED

IF state == GOAL:

RETURN VISITED

FOR each neighbor in getMoves(state):

IF neighbor not in VISITED:

enqueue neighbor into Q

BFS ensures that the solution found is the shortest possible.

**Step 4: Depth-First Search (DFS)**

DFS explores as deep as possible before backtracking.

*Pseudo code:*

FUNCTION solveByDFS(start):

create empty stack S

create empty list VISITED

push start into S

WHILE S not empty:

state ← pop from S

add state to VISITED

IF state == GOAL:

RETURN VISITED

FOR each neighbor in getMoves(state):

IF neighbor not in VISITED:

push neighbor into S

DFS may find a solution faster in smaller spaces but not always the shortest path.

**Step 5: Execution**

Two different start states are used for testing:

State 1: [8,1,2,0,4,3,7,6,5]

State 2: [1,8,2,0,4,3,7,6,5]

The program first checks solvability. If solvable, BFS or DFS is applied to find the goal state.

**Output**

**Sample Output:**

Solvable

Number of visited states: 232

Final state: 1 2 3 4 5 6 7 8 0

No solution

**Explanation of Output:**

* For **State 1**, the inversion count is even, making it solvable.  
  BFS successfully reaches the goal configuration.
* For **State 2**, the inversion count is odd, so it is unsolvable.

**Conclusion**

The 8-Puzzle problem was successfully solved using **Breadth-First Search** and **Depth-First Search** algorithms.  
BFS explored the search space systematically and guaranteed the shortest path to the solution, while DFS provided an alternative strategy for deeper exploration with less memory usage.

The project demonstrates key principles of **state space search**, **graph traversal**, and **problem-solving using AI algorithms**.  
It also shows the importance of solvability conditions and the role of search strategies in efficiently exploring large state spaces.