Experiment 1

1 Introduction

The **8-Puzzle Problem** is a classic example of a state space search problem in Artificial Intelligence (AI). It consists of a 3×3 grid with eight numbered tiles and one empty space (the blank, typically represented by 0). The objective is to rearrange a given starting configuration of the tiles to match a specified target configuration by sliding the tiles into the blank space. This experiment focuses on implementing and analyzing **uninformed search strategies**, specifically **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**, to find the optimal sequence of moves that solves the puzzle.

---

2 IMPLEMENT BASIC SEARCH STRATEGIES 8-PUZZLE PROBLEM

2.1 Objective

The primary objective is to implement and apply the Breadth-First Search (BFS) algorithm to solve the 8-Puzzle problem, demonstrating how to find the shortest (optimal) path solution. Secondary objectives include:

- Defining the problem formally as a **State Space Search** task.
- Comparing the core mechanics, completeness, and optimality of BFS versus Depth-First Search (DFS) for this class of problem.

2.2 Theory: 8-Puzzle as State Space Search

The 8-Puzzle is formally defined by its components:

1. **States:** The position of the 8 tiles and the blank in the 3×3 grid. Each state is a unique configuration.
2. **Initial State:** A given starting tile configuration.
3. **Operators (Actions):** The movement of the blank tile: Up, Down, Left, or Right. An action is valid only if the blank tile is not at the edge of the grid in that direction.
4. **Goal State:** The target configuration, typically in sorted order:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}$$

5. **Path Cost:** Each valid move has a unit cost of 1. The path cost is the total number of moves from the initial state to the goal state.

2.2.1 Breadth-First Search (BFS)

BFS is an uninformed search strategy that explores the search space level by level, expanding all nodes at depth d before moving to depth $d + 1$.

Characteristics:

- **Data Structure:** Utilizes a **Queue** (First-In, First-Out) to manage nodes awaiting expansion.
- **Completeness:** BFS is **complete**—it guarantees finding a solution if one exists, provided the search space is finite.
- **Optimality:** BFS is **optimal** for unit-cost actions (like the 8-Puzzle), as the first solution found is guaranteed to be the shallowest, representing the path with the fewest moves.
- **Complexity:** The time and space complexity are exponential, $O(b^d)$, where b is the branching factor (average number of moves, max $b = 4$) and d is the depth of the solution.

2.2.2 Depth-First Search (DFS)

DFS explores the search space by traversing as far as possible along a single path before backtracking.

Characteristics:

- **Data Structure:** Utilizes a **Stack** (Last-In, First-Out) or recursion to manage the search.
- **Completeness:** Standard DFS is **not complete** in infinite or very large cyclic state spaces, as it may follow an unproductive path indefinitely.
- **Optimality:** DFS is **not optimal**; it might find a deep solution (many moves) when a much shallower path exists.
- **Space Complexity:** DFS is highly space-efficient, requiring only linear space $O(b \times d)$, making it favorable when memory is highly constrained.

2.3 Program

The Python program below implements the 8-Puzzle solver using the **Breadth-First Search (BFS)** algorithm. We use a queue for the frontier and a set for visited states to ensure an optimal solution is found without cycles. The board state is represented as a flat tuple for hashing and comparison.

Example States Used for Testing:

Initial State	Goal State
$\begin{pmatrix}1 & 2 & 3\\4 & 0 & 6\\7 & 5 & 8\end{pmatrix}$	$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 0\end{pmatrix}$

```python
from collections import deque

# Node class to store the state and path history
class Node:
    def __init__(self, state, parent=None, action=None):
        # state is the 3x3 board represented as a 1D tuple
        self.state = state
        self.parent = parent # The parent node for path reconstruction
        self.action = action # The move (Up/Down/Left/Right) taken

    def __hash__(self):
        # Enables the node state to be used in a set/dictionary
        return hash(self.state)

    def __eq__(self, other):
        return self.state == other.state

# The standard goal state (1, 2, 3, 4, 5, 6, 7, 8, 0)
GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)

# Helper function to find the blank tile (0) position
def find_blank(state):
    idx = state.index(0)
    return idx // 3, idx % 3 # (row, col)

# Function to generate successor states (valid moves)
def get_successors(state):
    successors = []
    # Convert 1D tuple state back to 2D list board for easy swapping
    board = [list(state[i*3:i*3+3]) for i in range(3)]
    r_blank, c_blank = find_blank(state)

    # Possible moves: (dr, dc) for Up, Down, Left, Right
    moves = {
        'Up': (-1, 0), 'Down': (1, 0),
        'Left': (0, -1), 'Right': (0, 1)
    }

    for action, (dr, dc) in moves.items():
        r_new, c_new = r_blank + dr, c_blank + dc

        # Check if the move is within the grid bounds (0-2)
        if 0 <= r_new < 3 and 0 <= c_new < 3:
            # Create a new board by swapping the blank and the tile
            new_board = [row[:] for row in board]
            new_board[r_blank][c_blank] = new_board[r_new][c_new]
            new_board[r_new][c_new] = 0

            # Convert back to a tuple for the new state
            new_state = tuple(tile for row in new_board for tile in row)
            successors.append((new_state, action))

    return successors
```

Listing 1: Python Implementation of the 8-Puzzle BFS Solver (Part 1)

```python
def bfs_solve(initial_state):
    """Performs Breadth-First Search to solve the 8-Puzzle."""
    if initial_state == GOAL_STATE:
        return [] # Already solved

    # Setup BFS Queue (Frontier) and Visited Set (Explored)
    queue = deque([Node(initial_state)])
    visited = {initial_state}

    while queue:
        current_node = queue.popleft()

        # Check all possible successor states (valid moves)
        for successor_state, action in get_successors(current_node.state):
            if successor_state not in visited:

                # Create the new node
                new_node = Node(successor_state, current_node, action)

                # Check for goal state
                if successor_state == GOAL_STATE:
                    # Reconstruct and return the path
                    path = []
                    node = new_node
                    while node.parent:
                        path.append(node.action)
                        node = node.parent
                    return path[::-1] # Path from start to goal

                # Add new state to the queue and visited set
                visited.add(successor_state)
                queue.append(new_node)

    return None # No solution found after exploring the entire space

def print_solution(path):
    if path:
        print(f"Solution Found in {len(path)} moves.")
        print("Path: -> " + " -> ".join(path))
    else:
        print("No solution found for this configuration.")

if __name__ == "__main__":
    # Example solvable initial state: 1 2 3 / 4 0 6 / 7 5 8
    initial_board = (1, 2, 3, 4, 0, 6, 7, 5, 8)

    print("--- 8-Puzzle BFS Solver ---")

    # Run the BFS solver
    solution_path = bfs_solve(initial_board)

    print_solution(solution_path)
```

Listing 2: Python Implementation of the 8-Puzzle BFS Solver (Part 2)

2.4 Output

The program is executed using the Breadth-First Search algorithm on the provided initial state. The BFS algorithm successfully finds the optimal (shortest) sequence of moves required to solve the puzzle, which involves only two steps.

Sample Execution Output:

```
--- 8-Puzzle BFS Solver ---
Solution Found in 2 moves.
Path: -> Down -> Right
```

Trace of the Optimal Path Found by BFS:

1. **Initial State:** $\begin{pmatrix} 1 & 2 & 3 \\ 4 & \mathbf{0} & 6 \\ 7 & 5 & 8 \end{pmatrix}$
2. **Move 1: Down** (Move tile 5 up into the blank space) $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & \mathbf{0} & 8 \end{pmatrix}$
3. **Move 2: Right** (Move tile 8 left into the blank space) $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & \mathbf{0} \end{pmatrix}$
4. **Goal State Reached.**

2.5 Conclusion

This experiment successfully implemented the Breadth-First Search (BFS) algorithm to solve the 8-Puzzle problem. The implementation correctly modeled the puzzle as a state-space search problem, utilizing a queue for the frontier and a visited set for cycle detection. The results demonstrated the key property of BFS: finding the **optimal path** (shortest number of moves) to the goal state. This makes BFS a suitable, albeit sometimes resource-intensive, strategy for low-depth search problems like the 8-Puzzle.