Experiment 3

Aim: To implement the 8-Puzzle problem using heuristic search techniques:

- **Part A:** Best First Search
- **Part B:** A* Search Algorithm

Theory: The 8-Puzzle problem is a classic search problem in artificial intelligence that involves a 3x3 grid with 8 numbered tiles and one empty space. The objective is to move the tiles using the empty space to reach a goal configuration from a given initial configuration.

To solve this problem efficiently, heuristic search techniques are used:

1. **Best First Search (BFS):**
    - This algorithm selects the most promising node based on a heuristic function.
    - It uses a priority queue where nodes are sorted based on their estimated cost to reach the goal.
    - However, BFS does not guarantee the shortest path, as it only considers the heuristic function without the actual cost incurred.
2. *A Search Algorithm:**
    - A* combines both the actual cost from the start node to the current node (g(n)) and the estimated cost to the goal (h(n)).
    - It ensures an optimal solution by balancing exploration and exploitation.
    - The total cost function is given by: **f(n) = g(n) + h(n)**.

A heuristic function like the Manhattan distance is commonly used to estimate the cost in both search strategies.

Code:

Part A: Implementation Using Best First Search

```
import heapq

from typing import List, Tuple
```

```
class EightPuzzle:
    def __init__(self, initial_state: List[List[int]]):
        self.initial_state = initial_state
        self.goal_state = [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 0]  # 0 represents the empty tile
        ]

    def get_blank_position(self, state: List[List[int]]) -> Tuple[int, int]:
        for r in range(3):
            for c in range(3):
                if state[r][c] == 0:
                    return r, c
        raise ValueError("No blank tile found")

    def get_possible_moves(self, state: List[List[int]]) -> List[List[List[int]]]:
        moves = []
        directions = [
            (0, 1),   # right
            (0, -1),  # left
            (1, 0),   # down
            (-1, 0)   # up
        ]
```

```
    blank_r, blank_c = self.get_blank_position(state)
    for dr, dc in directions:
        new_r, new_c = blank_r + dr, blank_c + dc
        if 0 <= new_r < 3 and 0 <= new_c < 3:
            # Create a deep copy of the state
            new_state = [row[:] for row in state]
            new_state[blank_r][blank_c], new_state[new_r][new_c] =\
            new_state[new_r][new_c], new_state[blank_r][blank_c]
            moves.append(new_state)
    return moves

def calculate_heuristic(self, state: List[List[int]]) -> int:
    distance = 0
    for r in range(3):
        for c in range(3):
            if state[r][c] != 0:
                goal_r = (state[r][c] - 1) // 3
                goal_c = (state[r][c] - 1) % 3
                distance += abs(r - goal_r) + abs(c - goal_c)
    return distance

def best_first_search(self) -> List[List[List[int]]]:
    pq = [(self.calculate_heuristic(self.initial_state),
           self.initial_state,
```

```
            [self.initial_state])]
        visited = set(tuple(map(tuple, self.initial_state)))
        while pq:
            _, current_state, path = heapq.heappop(pq)
            if current_state == self.goal_state:
                return path
            for move in self.get_possible_moves(current_state):
                # Convert move to hashable type for visited check
                move_tuple = tuple(map(tuple, move))
                if move_tuple not in visited:
                    visited.add(move_tuple)
                    heuristic = self.calculate_heuristic(move)
                    heapq.heappush(pq, (heuristic, move, path + [move]))
        return []  # No solution found
    def print_solution(self, solution: List[List[List[int]]]):
        if not solution:
            print("No solution found.")
            return
        print("Solution Path:")
        for i, state in enumerate(solution):
            print(f"Step {i}:")
            for row in state:
                print(row)
            print()
```

```python
# Example usage

def main():

    # Example initial state

    initial_state = [

        [1, 2, 3],

        [4, 0, 6],

        [7, 5, 8]

    ]

    puzzle = EightPuzzle(initial_state)

    solution = puzzle.best_first_search()

    puzzle.print_solution(solution)

if __name__ == "__main__":

    main()
```

Output:

```
...  Solution Path:
     Step 0:
     [1, 2, 3]
     [4, 0, 6]
     [7, 5, 8]

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

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

Part B: Implementation Using A Search Algorithm*

```
# using a star

import heapq

from typing import List, Tupl

class AStarPuzzleSolver:

    def __init__(self, initial_board: List[List[int]]):

        self.initial_board = initial_board

        self.target_board = [

            [1, 2, 3],

            [4, 5, 6],

            [7, 8, 0]  # 0 represents the empty tile

        ]


    def find_empty_tile_position(self, board_state: List[List[int]]) -> Tuple[int, int]:

        for row_idx in range(3):

            for col_idx in range(3):

                if board_state[row_idx][col_idx] == 0:

                    return row_idx, col_idx

        raise ValueError("No empty tile found in the board")


    def generate_possible_configurations(self, board_state: List[List[int]]) -> List[List[List[int]]]:

        possible_configurations = []

        movement_directions = [

            (0, 1),   # right
```

```python
            (0, -1),  # left
            (1, 0),   # down
            (-1, 0)   # up
        ]
        empty_row, empty_col = self.find_empty_tile_position(board_state)
        for delta_row, delta_col in movement_directions:
            new_row, new_col = empty_row + delta_row, empty_col + delta_col

            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = [row[:] for row in board_state]
                new_board[empty_row][empty_col], new_board[new_row][new_col] = \
                new_board[new_row][new_col], new_board[empty_row][empty_col]
                possible_configurations.append(new_board)
        return possible_configurations

    def calculate_manhattan_distance(self, board_state: List[List[int]]) -> int:
        total_distance = 0
        for row_idx in range(3):
            for col_idx in range(3):
                if board_state[row_idx][col_idx] != 0:
                    target_row = (board_state[row_idx][col_idx] - 1) // 3
                    target_col = (board_state[row_idx][col_idx] - 1) % 3
                    total_distance += abs(row_idx - target_row) + abs(col_idx - target_col)
        return total_distance
```

```python
def solve_puzzle_a_star(self) -> List[List[List[int]]]:
    search_queue = [(
        self.calculate_manhattan_distance(self.initial_board),  # f_score
        0,  # g_score (initial cost)
        self.initial_board,
        [self.initial_board]
    )]
    explored_configurations = set(tuple(map(tuple, self.initial_board)))

    while search_queue:
        # Extract board with lowest f_score
        _, current_path_cost, current_board, solution_path = heapq.heappop(search_queue)
        if current_board == self.target_board:
            return solution_path
        for next_board in self.generate_possible_configurations(current_board):
            # Convert board to hashable type
            board_signature = tuple(map(tuple, next_board))
            next_path_cost = current_path_cost + 1
            heuristic_cost = self.calculate_manhattan_distance(next_board)
            total_f_score = next_path_cost + heuristic_cost
            if board_signature not in explored_configurations:
                explored_configurations.add(board_signature)
                heapq.heappush(search_queue, (
```

```
                total_f_score,  # f_score = g_score + h_score
                next_path_cost,  # g_score (cost to reach this state)
                next_board,
                solution_path + [next_board]
            ))
        return []  # No solution found

    def display_solution_steps(self, solution_steps: List[List[List[int]]]):
        if not solution_steps:
            print("No solution found.")
            return
        print("Solution Path:")
        for step_number, board_configuration in enumerate(solution_steps):
            print(f"Step {step_number}:")
            for row in board_configuration:
                print(row)
            print()


# Example usage
def main():
    # Example initial board configuration
    initial_board = [
        [1, 2, 0],
        [4, 6, 3],
        [7, 5, 8]
```

```
    ]
    puzzle = AStarPuzzleSolver(initial_board)
    solution = puzzle.solve_puzzle_a_star()
    puzzle.display_solution_steps(solution)
if __name__ == "__main__":
    main()
```

Output:

```
...   Solution Path:
      Step 0:
      [1, 2, 0]
      [4, 6, 3]
      [7, 5, 8]

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

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

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

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

Conclusion: In this experiment, we implemented the 8-Puzzle problem using heuristic search techniques. The Best First Search algorithm selects states based on heuristic values but may not guarantee the optimal solution. The A* Search algorithm considers both the heuristic and path cost, ensuring the shortest solution path. This highlights the importance of using an effective heuristic function in search-based problem-solving.