EXPERIMENT-3

Name: Adwait Purao
UID: 2021300101
Div: B
Batch: B2

Problem Definition: Implement an 8-puzzle problem using the Informed searching technique using A* algorithm. Analyze the algorithm with respect to Completeness, Optimality, time and space Complexity.

Theory:

- **A* algorithm:**
  - The A* algorithm is a search algorithm that uses a heuristic function to estimate the cost of reaching the goal state from a given state. The algorithm works by maintaining a priority queue of states to explore, sorted by their estimated costs. The algorithm starts with the initial state and continues until it reaches the goal state. At each step, it selects the state with the lowest estimated cost and expands its neighboring states.

  - The estimated cost of a state is calculated as the sum of the cost of reaching the state from the initial state (g(n)) and the heuristic estimate of the cost of reaching the goal state from the state (h(n)). The heuristic function should be admissible, meaning that it should never overestimate the actual cost of reaching the goal state.

  - The A* algorithm is guaranteed to find the optimal solution to the search problem, if it exists. This is because the heuristic function ensures that the algorithm will always explore the most promising states first.

- **Heuristic function:**
  - The heuristic function is a key component of the A* algorithm. It is a function that estimates the cost of reaching the goal state from a given state. The heuristic function should be admissible, meaning that it should never overestimate the actual cost of reaching the goal state.

  - There are many different heuristic functions that can be used for different problems. For the 8-puzzle problem, a common heuristic function is the Manhattan distance. The Manhattan distance is the sum of the distances between each tile and its desired position in the goal state.

Steps in the program:

1. Define the Manhattan distance heuristic function. This function calculates the sum of the distances between each tile and its desired position in the goal state. This heuristic function is used to estimate the cost of reaching the goal state from a given state.

2. *Define the A algorithm.** This algorithm works by maintaining a priority queue of states to explore, sorted by their estimated costs. The algorithm starts with the initial state and continues until it reaches the goal state. At each step, it selects the state with the lowest estimated cost and expands its neighboring states.

3. Helper functions:
    - get_possible_moves(): This function returns a list of all possible moves that can be made from the given state.
    - apply_move(): This function applies the given move to the given state and returns the resulting state.

4. *Call the A algorithm:**
    - astar(): This function implements the A* algorithm and returns a list of moves that lead from the initial state to the goal state, or None if no solution is found.

5. Call the function with user-provided initial and goal states:
    - solve_8_puzzle(): This function takes the initial and goal states as input and returns a list of moves that lead from the initial state to the goal state, or None if no solution is found.

Example:

Consider the following 8-puzzle problem:

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

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

The A* algorithm would start by exploring the initial state. It would then expand the neighboring states of the initial state, which are:

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

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

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

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

The A* algorithm would then select the state with the lowest estimated cost, which is the first state in the list. It would then expand the neighboring states of this state, and so on.

The A* algorithm would continue to explore states in this way until it reaches the goal state, or until it determines that there is no solution.

Code:

```
from heapq import heappop, heappush
import time

def solve_8_puzzle(initial_state, goal_state):
    # Define the Manhattan distance heuristic function
    def heuristic(state):
        distance = 0
        for i in range(3):
            for j in range(3):
                if state[i][j] != 0:
                    row = (state[i][j] - 1) // 3
                    col = (state[i][j] - 1) % 3
                    distance += abs(row - i) + abs(col - j)
        return distance

    # Define the A* algorithm
```

```
    def astar():
        open_set = []
        heappush(open_set, (heuristic(initial_state), 0, initial_state, []))
        closed_set = set()

        while open_set:
            _, cost, current_state, path = heappop(open_set)
            if current_state == goal_state:
                return path

            closed_set.add(tuple(map(tuple, current_state)))

            for move in get_possible_moves(current_state):
                new_state = apply_move(current_state, move)
                if tuple(map(tuple, new_state)) not in closed_set:
                    new_cost = cost + 1
                    new_path = path + [(move, new_state)]
                    heappush(open_set, (new_cost + heuristic(new_state),
new_cost, new_state, new_path))

    # Helper functions
    def get_possible_moves(state):
        moves = []
        empty_row, empty_col = 0, 0
        for i in range(3):
            for j in range(3):
                if state[i][j] == 0:
                    empty_row, empty_col = i, j
                    break

        # Possible moves: up, down, left, right
        if empty_row > 0:
            moves.append("UP")
        if empty_row < 2:
            moves.append("DOWN")
        if empty_col > 0:
            moves.append("LEFT")
        if empty_col < 2:
            moves.append("RIGHT")

        return moves

    def apply_move(state, move):
        # Clone the state to avoid modifying the original state
        new_state = [row[:] for row in state]
```

```
        empty_row, empty_col = 0, 0

        # Find the position of the empty tile
        for i in range(3):
            for j in range(3):
                if new_state[i][j] == 0:
                    empty_row, empty_col = i, j

        # Perform the move
        if move == "UP":
            new_state[empty_row][empty_col], new_state[empty_row - 1][empty_col]
= new_state[empty_row - 1][empty_col], new_state[empty_row][empty_col]
        elif move == "DOWN":
            new_state[empty_row][empty_col], new_state[empty_row + 1][empty_col]
= new_state[empty_row + 1][empty_col], new_state[empty_row][empty_col]
        elif move == "LEFT":
            new_state[empty_row][empty_col], new_state[empty_row][empty_col - 1]
= new_state[empty_row][empty_col - 1], new_state[empty_row][empty_col]
        elif move == "RIGHT":
            new_state[empty_row][empty_col], new_state[empty_row][empty_col + 1]
= new_state[empty_row][empty_col + 1], new_state[empty_row][empty_col]

        return new_state

    # Call the A* algorithm
    solution_steps = astar()
    return solution_steps

# Get user input for the initial state
initial_state = []
print("Enter the initial state of the 8-puzzle row-wise:")
for _ in range(3):
    row = list(map(int, input().split()))
    initial_state.append(row)

# Get user input for the goal state
goal_state = []
print("Enter the goal state of the 8-puzzle row-wise:")
for _ in range(3):
    row = list(map(int, input().split()))
    goal_state.append(row)

# Call the function with user-provided initial and goal states
start = time.time()
solution_steps = solve_8_puzzle(initial_state, goal_state)
```

```
end = time.time()
print("Solution Steps:")
steps=0
for move, state in solution_steps:
    steps+=1
    print("Move:", move)
    for row in state:
        print(row)
    print()
print("Number of steps taken: ",steps)
print("The time taken by the algo is: ",(end - start)," sec")
```

Output:

```
PS D:\Rohit College\5th sem\AIML\Exp 3> python -u "d:\Rohit College\5th sem\AIML\Exp 3\Exp_3.py"
Enter the initial state of the 8-puzzle row-wise:
1 2 3
8 7 6
5 4 0
Enter the goal state of the 8-puzzle row-wise:
1 2 3
4 5 6
7 8 0
Solution Steps:
Move: LEFT
[1, 2, 3]
[8, 7, 6]
[5, 0, 4]

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

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

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

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

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

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

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

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

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

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

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

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

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

Number of steps taken:  14
The time taken by the algo is:  0.003414630889892578  sec
PS D:\Rohit College\5th sem\AIML\Exp 3>
```

Analysis of the Code:

- **Completeness:**
  The above code is complete, meaning that it will find a solution to the 8-puzzle problem if one exists. This is because the A* algorithm is guaranteed to find the optimal solution to the search problem, if it exists.

- **Optimality:**
  The above code is optimal, meaning that it will find the shortest sequence of moves to solve the 8-puzzle problem. This is because the A* algorithm is guaranteed to find the optimal solution to the search problem, if it exists.

- **Time complexity:**
  The time complexity of the above code is O(b^d), where b is the branching factor of the search problem and d is the depth of the search tree. The branching factor of the 8-puzzle problem is 4 (up, down, left, right) and the depth of the search tree is bounded by the number of possible states of the 8-puzzle problem, which is 9!.

- **Space complexity:**
  The space complexity of the above code is O(b^d), where b is the branching factor of the search problem and d is the depth of the search tree. This is because the A* algorithm maintains a priority queue of states to explore, which can grow to be as large as the search tree.

  In practice, the time and space complexity of the above code can be reduced by using heuristics to guide the search. For example, the Manhattan distance heuristic used in the above code can help to reduce the size of the search tree and the number of states that need to be explored.

Conclusion:

The 8-puzzle code is a complete and optimal algorithm for solving the 8-puzzle problem, using the A* algorithm. The A* algorithm is guaranteed to find the shortest sequence of moves to solve the problem, if one exists. However, the A* algorithm can be computationally expensive for large problem spaces. Heuristics, such as the Manhattan distance heuristic, can be used to improve the efficiency of the algorithm.