Name: Yash Ravindra Burad
Roll no.: 2022301004
Experiment: 3
Batch: Comps A - Batch A.
Aim: Problem solving using informed search(a* algo)
Theory:
Introduction to A* Algorithm:
The A* algorithm is a widely used pathfinding algorithm in artificial intelligence and computer science. It is designed to find the shortest path or optimal solution from a start state to a goal state in a graph or search space. A* combines the advantages of both uniform-cost search and greedy search by considering both the cost to reach a state and an estimate of the cost to reach the goal. This estimate, known as a heuristic, guides the search process and helps A* make informed decisions.

Components of A* Algorithm
1. Cost Function (g): A* uses a cost function to keep track of the actual cost from the start state to the current state. It starts with a cost of zero at the start state and increments as it explores neighboring states.
2. Heuristic Function (h): The heuristic function estimates the cost from the current state to the goal state. A good heuristic should be admissible, meaning it never overestimates the true cost. The choice of heuristic influences the algorithm's efficiency and optimality.
3. Total Cost (f): A* combines the cost function and the heuristic function to compute the total cost of a state. The total cost is defined as f = g + h. A* prioritizes states with lower total costs for exploration.

A Algorithm Steps*
The A* algorithm follows these steps:
Initialize the open list with the start state and a total cost of zero.
While the open list is not empty:
a. Pop the state with the lowest total cost from the open list.
b. Check if the current state is the goal state. If yes, the solution is found.
c. Generate neighboring states and calculate their total costs.
d. Add neighboring states to the open list.
If the open list becomes empty and the goal state is not reached, no solution exists.
Different Heuristic Methods

A* can use different heuristic methods to estimate the cost to the goal state:

1. Manhattan Distance: This heuristic is commonly used for grid-based problems like the 8-puzzle. It calculates the sum of the horizontal and vertical distances between each tile's current position and its goal position.

2. Euclidean Distance: This heuristic computes the straight-line (Euclidean) distance between the current state and the goal state. It can be used when movement is not constrained to grid cells.

3. Misplaced Tiles: In the 8-puzzle, this heuristic counts the number of tiles that are not in their correct positions in the current state compared to the goal state.

4. Admissible Heuristics: An admissible heuristic never overestimates the true cost. If h(n) is an admissible heuristic, then A* is guaranteed to find an optimal solution.

Applications of A* Algorithm:

1. Robotics Path Planning: A* is widely used in robotics to plan paths for robots navigating in real-world environments. It helps robots find the shortest path to their destinations while avoiding obstacles.

2. Video Game AI: Game developers use A* for character pathfinding in video games. Non-player characters (NPCs) use A* to navigate game maps efficiently.

3. Network Routing: A* can be applied to find optimal routes in computer networks, transportation systems, and communication networks. It minimizes data transfer delays and congestion.

4. Natural Language Processing: A* is used in various natural language processing tasks, including machine translation, text summarization, and speech recognition, to find the most likely sequence of words or phrases.

5. Puzzle Solving: A* is applied to solve puzzles like the 8-puzzle, 15-puzzle, and Sudoku by finding the optimal sequence of moves to reach the goal state.

6. Artificial Intelligence: A* is a fundamental algorithm in AI and is used in search algorithms for game-playing agents, constraint satisfaction problems, and automated planning.

the A* algorithm is a versatile and widely used search algorithm that combines cost and heuristic information to find optimal solutions in various domains. Its efficiency and effectiveness make it a crucial tool in computer science, artificial intelligence, and problem-solving applications. Different heuristic methods can be tailored to specific problem domains to improve the algorithm's performance and optimality.

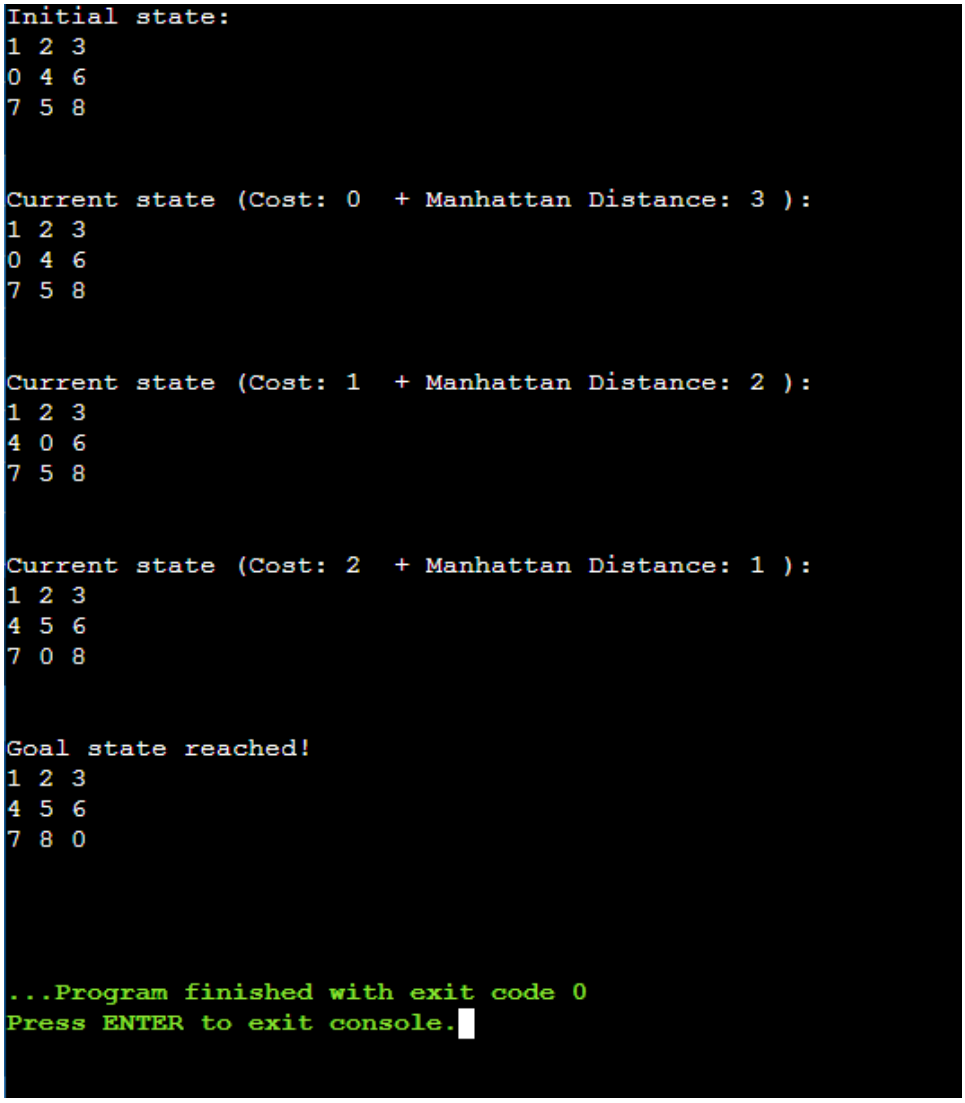
Program:(A* algorithm with the Manhattan heuristic method)

```
import heapq

initial_state = [[1, 2, 3], [0, 4, 6], [ 7,5, 8]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

```
n = 3

moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]

def calculate_manhattan_distance(state):
    distance = 0
    for i in range(n):
        for j in range(n):
            if state[i][j] != 0:
                target_value = state[i][j]
                target_row, target_col = divmod(target_value - 1, n)
                distance += abs(i - target_row) + abs(j - target_col)
    return distance

def is_valid_move(x, y):
    return 0 <= x < n and 0 <= y < n

def get_neighbors(state):
    neighbors = []
    for dx, dy in moves:
        new_state = [row[:] for row in state]
        for i in range(n):
            for j in range(n):
                if state[i][j] == 0:
                    new_x, new_y = i + dx, j + dy
                    if is_valid_move(new_x, new_y):
                        new_state[i][j], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[i][j]
                        neighbors.append((new_state, (i, j)))
                    break
    return neighbors

def print_state(state):
    for row in state:
        print(" ".join(map(str, row)))
    print("\n")

def a_star(initial_state, goal_state):
    open_list = [(0 + calculate_manhattan_distance(initial_state), 0, initial_state, None)]
    closed_set = set()
    while open_list:
        _, cost, current_state, parent = heapq.heappop(open_list)
        if current_state == goal_state:
```

```
            print("Goal state reached!")
            print_state(current_state)
            return
        if tuple(map(tuple, current_state)) in closed_set:
            continue
        closed_set.add(tuple(map(tuple, current_state)))
        print("Current state (Cost:", cost, " + Manhattan Distance:",
calculate_manhattan_distance(current_state), "):")
        print_state(current_state)
        for neighbor_state, move in get_neighbors(current_state):
            g_score = cost + 1
            h_score = calculate_manhattan_distance(neighbor_state)
            f_score = g_score + h_score
            heapq.heappush(open_list, (f_score, g_score, neighbor_state, current_state))

    print("No solution found!")

if __name__ == "__main__":
    print("Initial state:")
    print_state(initial_state)
    a_star(initial_state, goal_state)
```

Output:



Observation:
Above program will give exponential time complexity in the worst case,
In the worst case, when the A* algorithm explores a large portion of the search space before finding the goal state, the time complexity can be exponential. However, the use of heuristics, such as the Manhattan distance heuristic, often significantly reduces the effective branching factor and, in practice, leads to more efficient search.

Conclusion:
In this practical, I learnt about A* algorithm using manhattan method, and solved 8 puzzle problem by specifying initial state and final state. This path finding algorithm is very useful in artificial intelligence and computer science.