Experiment # 4: 8 Puzzle problem using a Priority Queue

Aim/Objective:

Implement the A* search algorithm to solve the 8-puzzle problem.

Description:

Students will implement the A* search algorithm using heuristics such as the *Manhattan distance* to estimate the cost of reaching the goal state from each possible move. By considering both the cost incurred so far and the estimated cost to reach the goal, the A* search algorithm intelligently explores state space to find an optimal solution to the 8-puzzle problem.

Pre-Requisites:

- Basic understanding of search algorithms.
- Familiarity with Python programming language.
- Knowledge of the 8-puzzle problem and its rules.

Pre-Lab:

1. How does the A* algorithm utilize a priority queue to solve the 8 Puzzle problem?

A* uses a priority queue to manage nodes based on the total cost $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach the current state and $h(n)$ is the estimated cost to reach the goal. The node with the lowest $f(n)$ is expanded first, ensuring the most promising paths are explored.

2. What is the role of the heuristic function in the A* algorithm for the 15 Puzzle problem, and how does it affect the priority queue?

The heuristic function estimates the remaining cost to reach the goal. It affects the priority queue by prioritizing nodes with lower estimated total costs, guiding A* towards the goal more efficiently.

3. What is the A* search algorithm, and how does it work?

A* is a search algorithm that finds the shortest path by evaluating nodes based on $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost and $h(n)$ is the heuristic estimate. It expands the node with the lowest $f(n)$ and continues until the goal is reached.

4. What is a heuristic function, and why is it important in A* search?

A heuristic function estimates the cost to reach the goal from a given state. It helps A* prioritize more promising paths, making the search more efficient.

5. What is the Manhattan distance heuristic, and how is it calculated in the context of the 8-puzzle problem?

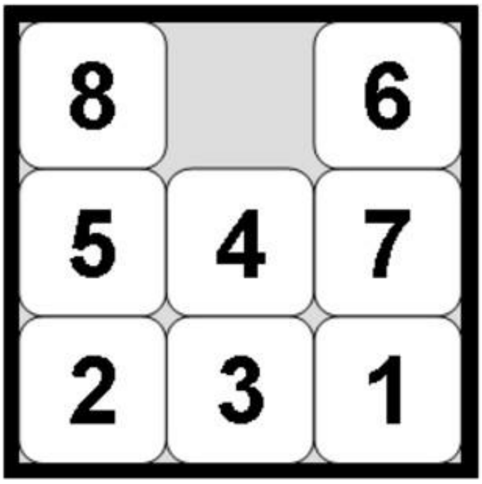
The Manhattan distance heuristic calculates the sum of the horizontal and vertical distances of each tile from its goal position. It is used in the 8 Puzzle problem to estimate how far the current state is from the goal state.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT] THANOS

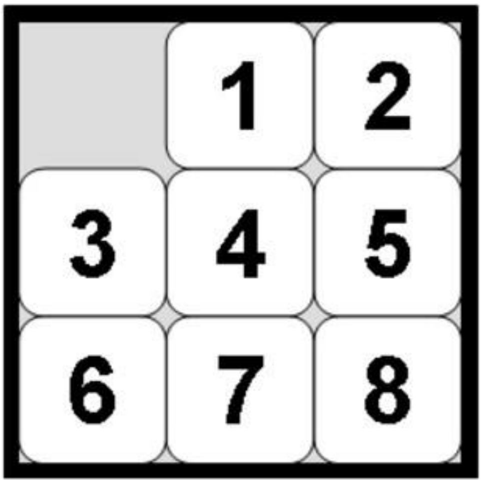
In-Lab:

Implement the A* search algorithm using a priority queue to solve a 8 puzzle problem.

Description: The 8-puzzle problem involves a 3X3 grid with 8 numbered tiles and one empty space. The goal is to rearrange the tiles from a given initial state to a desired goal state by sliding the tiles into the empty space. The A* search algorithm is used to find an optimal solution by considering both the cost incurred so far and the estimated cost to reach the goal state using a heuristic function.



Initial State



Final State

Procedure/Program:

```
import heapq
import copy

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

print("Initial State:")
for i in range(0, 9, 3):
   print(initial_state[i:i+3])
print()

inversions = 0
tiles = [tile for tile in initial_state if tile != 0]
for i in range(len(tiles)):
   for j in range(i + 1, len(tiles)):
      if tiles[i] > tiles[j]:
         inversions += 1
solvable = inversions % 2 == 0
```

```
if not solvable:
    print("Unsolvable puzzle")
else:
    frontier = []
    heapq.heappush(frontier, (0, initial_state))
    came_from = {}
    cost_so_far = {tuple(initial_state): 0}
    solution = None

    while frontier:
        _, current = heapq.heappop(frontier)

        if current == goal_state:
            path = []
            while tuple(current) in came_from:
                path.append(current)
                current = came_from[tuple(current)]
            path.append(initial_state)
            path.reverse()
            solution = path
            break

        zero_index = current.index(0)
        x, y = divmod(zero_index, 3)
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                neighbor = copy.deepcopy(current)
                new_index = nx * 3 + ny
                neighbor[zero_index], neighbor[new_index] = neighbor[new_index],
neighbor[zero_index]

                new_cost = cost_so_far[tuple(current)] + 1
                if tuple(neighbor) not in cost_so_far or new_cost <
cost_so_far[tuple(neighbor)]:
                    cost_so_far[tuple(neighbor)] = new_cost

                    distance = 0
                    for i in range(1, 9):
```

```
                x1, y1 = divmod(neighbor.index(i), 3)
                x2, y2 = divmod(goal_state.index(i), 3)
                distance += abs(x1 - x2) + abs(y1 - y2)

            priority = new_cost + distance
            heapq.heappush(frontier, (priority, neighbor))
            came_from[tuple(neighbor)] = current

if solution:
    print("Solution found with the following steps:")
    for idx, step in enumerate(solution):
        if idx == 0:
            print("Initial State:")
        elif idx == len(solution) - 1:
            print("Final State:")
        else:
            print(f"Step {idx}:")
        for i in range(0, 9, 3):
            print(step[i:i+3])
        print()
else:
    print("No solution exists")
```

OUTPUT

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

Solution found with the following steps:
Initial State:
[8, 0, 6]
[5, 4, 7]
[2, 3, 1]

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

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

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

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

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

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

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

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

Step 9:
[4, 5, 8]
[2, 7, 6]
[0, 3, 1]

Step 10:
[4, 5, 8]
[2, 7, 6]
[3, 0, 1]

Step 11:
[4, 5, 8]
[2, 7, 6]
[3, 1, 0]

Step 12:
[4, 5, 8]
[2, 7, 0]
[3, 1, 6]

Step 13:
[4, 5, 8]
[2, 0, 7]
[3, 1, 6]

Step 14:
[4, 5, 8]
[2, 1, 7]
[3, 0, 6]

Step 15:
[4, 5, 8]
[2, 1, 7]
[3, 6, 0]

Step 16:
[4, 5, 8]
[2, 1, 0]
[3, 6, 7]

Step 17:
[4, 5, 0]
[2, 1, 8]
[3, 6, 7]

Step 18:
[4, 0, 5]
[2, 1, 8]
[3, 6, 7]

Step 19:
[4, 1, 5]
[2, 0, 8]
[3, 6, 7]

Step 20:
[4, 1, 5]
[0, 2, 8]
[3, 6, 7]

Step 21:
[0, 1, 5]
[4, 2, 8]
[3, 6, 7]

Step 22:
[1, 0, 5]
[4, 2, 8]
[3, 6, 7]

Step 23:
[1, 2, 5]
[4, 0, 8]
[3, 6, 7]

Step 24:
[1, 2, 5]
[0, 4, 8]
[3, 6, 7]

Step 25:
[1, 2, 5]
[3, 4, 8]
[0, 6, 7]

Step 26:
[1, 2, 5]
[3, 4, 8]
[6, 0, 7]

Step 27:
[1, 2, 5]
[3, 4, 8]
[6, 7, 0]

Step 28:
[1, 2, 5]
[3, 4, 0]
[6, 7, 8]

Step 29:
[1, 2, 0]
[3, 4, 5]
[6, 7, 8]

Step 30:
[1, 0, 2]
[3, 4, 5]
[6, 7, 8]

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

- **Data and Results:**

DATA

The 8-puzzle problem is solved using A* search algorithm.

RESULT

The goal state was reached in an optimal number of moves.

- **Analysis and Inferences:**

ANALYSIS

Manhattan distance heuristic ensures efficient exploration of puzzle search space.

INFERENCES

A* search is effective for solving small sliding puzzle problems.

Sample VIVA-VOCE Questions (In-Lab):

1. What is the purpose of using the A* search algorithm in the 8-puzzle problem?

A* is used to find the optimal solution by exploring paths with the lowest total cost, combining actual and estimated costs to guide the search efficiently.

2. Explain the role of the heuristic function in the A* search algorithm.

The heuristic function estimates the cost to reach the goal, helping A* prioritize nodes to explore, improving search efficiency.

3. How is the Manhattan distance calculated in the context of the 8-puzzle problem?

Manhattan distance is the sum of the absolute differences in row and column positions of each tile from its goal position.

4. What are some advantages and limitations of the A* search algorithm?

Advantages: Guarantees optimal solution and improves search efficiency with a good heuristic.
Limitations: Memory-intensive and heuristic-dependent.

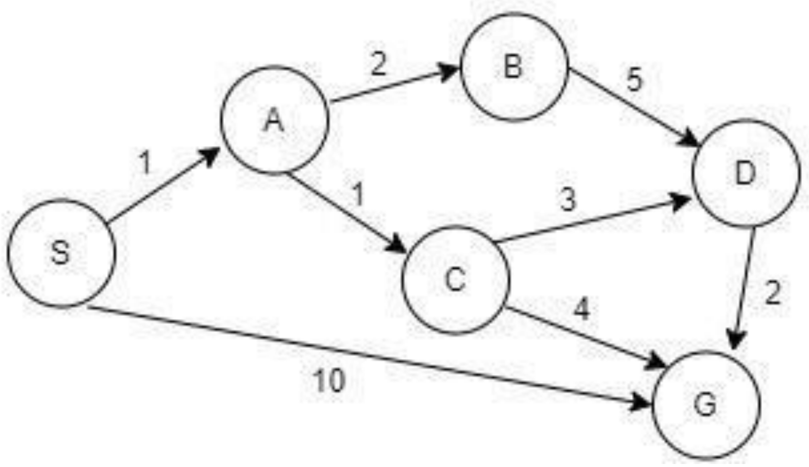
5. Can the A* search algorithm guarantee finding an optimal solution in all cases?

Yes, if the heuristic is admissible and consistent. Otherwise, it may not find the optimal solution.

Post-Lab:

Experiment: Implement the A* search algorithm for the following graph.

State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0



Procedure/ Program:

```
from queue import PriorityQueue

graph = {
   'S': [('A', 1), ('G', 10)],
   'A': [('B', 2), ('C', 1)],
   'B': [('D', 5)],
   'C': [('D', 3), ('G', 4)],
   'D': [('G', 2)],
   'G': []
}

heuristic = {
   'S': 5, 'A': 3, 'B': 4, 'C': 2, 'D': 6, 'G': 0
}

start = 'S'
goal = 'G'

open_list = PriorityQueue()
open_list.put((0, start))
```

```
came_from = {}
g_score = {node: float('inf') for node in graph}
g_score[start] = 0

while not open_list.empty():
    _, current = open_list.get()

    if current == goal:
        path = []
        total_cost = g_score[goal]
        while current:
            path.append(current)
            current = came_from.get(current)
        path.reverse()
        print("Shortest path:", path)
        print("Total cost:", total_cost)
        break

    for neighbor, cost in graph[current]:
        tentative_g_score = g_score[current] + cost
        if tentative_g_score < g_score[neighbor]:
            g_score[neighbor] = tentative_g_score
            f_score = tentative_g_score + heuristic[neighbor]
            open_list.put((f_score, neighbor))
            came_from[neighbor] = current
else:
    print("No path found")
```

OUTPUT

```
Shortest path: ['S', 'A', 'C', 'G']
Total cost: 6
```

- **Data and Results:**

Data

The graph has nodes, edges, and heuristic values for exploration.

Result

The shortest path and total cost were successfully determined.

- **Analysis and Inferences:**

Analysis

A* explores nodes efficiently using both path cost and heuristics.

Inferences

A* ensures the shortest path by balancing cost and heuristic.

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date