LAB 8

Task 1 Implement greedy best first search in Python.

```
import heapq


class Node:
    def __init__(self, name, heuristic):
        self.name = name
        self.heuristic = heuristic


    def __lt__(self, other):
        return self.heuristic < other.heuristic


def greedy_best_first_search_hierarchical(graph, start, goal,
heuristic, region_map):
    priority_queue = []
    heapq.heappush(priority_queue, Node(start, heuristic[start]))


    visited = set()
    path = {start: None}


    while priority_queue:
        current_node = heapq.heappop(priority_queue).name
        print(f"Visiting: {current_node}")  # step-by-step output
```

```
        if current_node == goal:
            return reconstruct_path(path, start, goal)

        visited.add(current_node)
        current_region = region_map[current_node]

        # Explore neighbors in the same region first
        for neighbor in graph[current_node]:
            if neighbor not in visited and region_map[neighbor] ==
current_region:
                heapq.heappush(priority_queue, Node(neighbor,
heuristic[neighbor]))
                if neighbor not in path:
                    path[neighbor] = current_node

        # Explore neighbors in different regions next
        for neighbor in graph[current_node]:
            if neighbor not in visited and region_map[neighbor] !=
current_region:
                heapq.heappush(priority_queue, Node(neighbor,
heuristic[neighbor]))
                if neighbor not in path:
                    path[neighbor] = current_node
```

```
    return None

def reconstruct_path(path, start, goal):
    current = goal
    result_path = []
    while current is not None:
        result_path.append(current)
        current = path[current]
    result_path.reverse()
    return result_path

# Graph definition
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['H'],
    'E': ['I', 'J'],
    'F': ['K', 'M', 'E'],
    'G': ['L', 'M'],
    'H': [],
    'I': [],
```

```
    'J': [],
    'K': [],
    'L': [],
    'M': []
}

heuristic = {
    'A': 8, 'B': 6, 'C': 7,
    'D': 5, 'E': 4, 'F': 5, 'G': 4,
    'H': 3, 'I': 2, 'J': 1, 'K': 3,
    'L': 2, 'M': 1
}

region_map = {
    'A': 1, 'B': 1, 'C': 1,
    'D': 2, 'E': 2,
    'F': 3, 'G': 3,
    'H': 2, 'I': 2, 'J': 2,
    'K': 3, 'L': 3, 'M': 3
}

# Run the algorithm
start_node = 'A'
```

goal_node = 'M'

result_path = greedy_best_first_search_hierarchical(graph, start_node, goal_node, heuristic, region_map)

print("\nFinal Path from {} to {}: {}".format(start_node, goal_node, result_path))

OUTPUT:

Visiting: D

Visiting: H

Visiting: C

Visiting: G

Visiting: M

Final Path from A to M: ['A', 'C', 'G', 'M']

Task 2 Implement the A* search in python.

Source Code:

Python program for A* Search Algorithm

import math

import heapq

Define the Cell class

```python
class Cell:
   def __init__(self):
     # Parent cell's row index
       self.parent_i = 0
   # Parent cell's column index
       self.parent_j = 0
 # Total cost of the cell (g + h)
       self.f = float('inf')
   # Cost from start to this cell
       self.g = float('inf')
   # Heuristic cost from this cell to destination
       self.h = 0


# Define the size of the grid
ROW = 9
COL = 10

# Check if a cell is valid (within the grid)
```

```
def is_valid(row, col):
    return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

# Check if a cell is unblocked



def is_unblocked(grid, row, col):
    return grid[row][col] == 1

# Check if a cell is the destination



def is_destination(row, col, dest):
    return row == dest[0] and col == dest[1]

# Calculate the heuristic value of a cell (Euclidean distance to destination)



def calculate_h_value(row, col, dest):
    return ((row - dest[0]) ** 2 + (col - dest[1]) ** 2) ** 0.5

# Trace the path from source to destination
```

```
def trace_path(cell_details, dest):
    print("The Path is ")
    path = []
    row = dest[0]
    col = dest[1]

    # Trace the path from destination to source using parent cells
    while not (cell_details[row][col].parent_i == row and 
cell_details[row][col].parent_j == col):
        path.append((row, col))
        temp_row = cell_details[row][col].parent_i
        temp_col = cell_details[row][col].parent_j
        row = temp_row
        col = temp_col

    # Add the source cell to the path
    path.append((row, col))
    # Reverse the path to get the path from source to destination
    path.reverse()

    # Print the path
```

```python
    for i in path:
        print("->", i, end=" ")
    print()

# Implement the A* search algorithm


def a_star_search(grid, src, dest):
    # Check if the source and destination are valid
    if not is_valid(src[0], src[1]) or not is_valid(dest[0], dest[1]):
        print("Source or destination is invalid")
        return

    # Check if the source and destination are unblocked
    if not is_unblocked(grid, src[0], src[1]) or not is_unblocked(grid,
dest[0], dest[1]):
        print("Source or the destination is blocked")
        return

    # Check if we are already at the destination
    if is_destination(src[0], src[1], dest):
        print("We are already at the destination")
        return
```

```
# Initialize the closed list (visited cells)
closed_list = [[False for _ in range(COL)] for _ in range(ROW)]
# Initialize the details of each cell
cell_details = [[Cell() for _ in range(COL)] for _ in range(ROW)]

# Initialize the start cell details
i = src[0]
j = src[1]
cell_details[i][j].f = 0
cell_details[i][j].g = 0
cell_details[i][j].h = 0
cell_details[i][j].parent_i = i
cell_details[i][j].parent_j = j

# Initialize the open list (cells to be visited) with the start cell
open_list = []
heapq.heappush(open_list, (0.0, i, j))

# Initialize the flag for whether destination is found
found_dest = False

# Main loop of A* search algorithm
```

```
    while len(open_list) > 0:
        # Pop the cell with the smallest f value from the open list
        p = heapq.heappop(open_list)

        # Mark the cell as visited
        i = p[1]
        j = p[2]
        closed_list[i][j] = True

        # For each direction, check the successors
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0),
                      (1, 1), (1, -1), (-1, 1), (-1, -1)]
        for dir in directions:
            new_i = i + dir[0]
            new_j = j + dir[1]

            # If the successor is valid, unblocked, and not visited
            if is_valid(new_i, new_j) and is_unblocked(grid, new_i,
new_j) and not closed_list[new_i][new_j]:
                # If the successor is the destination
                if is_destination(new_i, new_j, dest):
                    # Set the parent of the destination cell
                    cell_details[new_i][new_j].parent_i = i
```

```
                    cell_details[new_i][new_j].parent_j = j
                    print("The destination cell is found")
                    # Trace and print the path from source to destination
                    trace_path(cell_details, dest)
                    found_dest = True
                    return
                else:
                    # Calculate the new f, g, and h values
                    g_new = cell_details[i][j].g + 1.0
                    h_new = calculate_h_value(new_i, new_j, dest)
                    f_new = g_new + h_new

                    # If the cell is not in the open list or the new f value is smaller
                    if cell_details[new_i][new_j].f == float('inf') or cell_details[new_i][new_j].f > f_new:
                        # Add the cell to the open list
                        heapq.heappush(open_list, (f_new, new_i, new_j))
                        # Update the cell details
                        cell_details[new_i][new_j].f = f_new
                        cell_details[new_i][new_j].g = g_new
                        cell_details[new_i][new_j].h = h_new
                        cell_details[new_i][new_j].parent_i = i
```

```
                cell_details[new_i][new_j].parent_j = j

    # If the destination is not found after visiting all cells
    if not found_dest:
        print("Failed to find the destination cell")

# Driver Code



def main():
    # Define the grid (1 for unblocked, 0 for blocked)
    grid = [
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 1],
        [1, 1, 1, 0, 1, 1, 0, 1, 0, 1],
        [0, 0, 1, 0, 1, 0, 0, 0, 0, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
        [1, 0, 1, 1, 1, 1, 0, 1, 0, 0],
        [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 0, 0, 1, 0, 0, 1]
    ]
```

```
    # Define the source and destination
    src = [8, 0]
    dest = [0, 0]

    # Run the A* search algorithm
    a_star_search(grid, src, dest)


if __name__ == "__main__":
    main()
```

Task 3 Implement the 8 Puzzle Problem using A* search in python.

```
import heapq

# 8-puzzle A* implementation

class PuzzleState:
    def __init__(self, board, parent=None, move="", g=0, h=0):
        self.board = board      # board as a tuple of numbers
        self.parent = parent    # parent state
        self.move = move        # move taken to reach this state
        self.g = g              # cost so far
        self.h = h              # heuristic (Manhattan distance)
        self.f = g + h          # total cost
```

```python
    def __lt__(self, other):
        return self.f < other.f


# Manhattan distance heuristic
def manhattan(board, goal):
    distance = 0
    for i in range(1, 9):  # ignore 0 (blank tile)
        xi, yi = divmod(board.index(i), 3)
        xg, yg = divmod(goal.index(i), 3)
        distance += abs(xi - xg) + abs(yi - yg)
    return distance


# Generate neighbors by sliding blank tile (0)
def get_neighbors(state, goal):
    neighbors = []
    idx = state.board.index(0)   # blank position
    x, y = divmod(idx, 3)

    moves = {
        "Up": (x - 1, y),
```

```
        "Down": (x + 1, y),
        "Left": (x, y - 1),
        "Right": (x, y + 1),
    }

    for move, (nx, ny) in moves.items():
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_idx = nx * 3 + ny
            new_board = list(state.board)
            # swap blank with target tile
            new_board[idx], new_board[new_idx] =
new_board[new_idx], new_board[idx]
            new_board = tuple(new_board)
            h = manhattan(new_board, goal)
            neighbors.append(PuzzleState(new_board, state, move,
state.g + 1, h))

    return neighbors


# Reconstruct path from goal to start
def reconstruct_path(state):
    path = []
```

```
    while state.parent:
        path.append(state.move)
        state = state.parent
    return path[::-1]  # reverse path


# A* algorithm
def a_star(start, goal):
    open_list = []
    start_state = PuzzleState(start, None, "", 0, manhattan(start,
goal))
    heapq.heappush(open_list, start_state)

    closed_set = set()

    while open_list:
        current = heapq.heappop(open_list)

        if current.board == goal:
            return reconstruct_path(current)

        closed_set.add(current.board)
```

```
        for neighbor in get_neighbors(current, goal):
            if neighbor.board not in closed_set:
                heapq.heappush(open_list, neighbor)

    return None


# Driver code
if __name__ == "__main__":
    start = (1, 2, 3,
             4, 0, 5,
             6, 7, 8)  # Example start state

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

    solution = a_star(start, goal)

    if solution:
        print("Solution found in", len(solution), "moves:")
        print(solution)
    else:
```

```
print("No solution exists")
```