ASSIGNMENT

1. Write a Python program to find the shortest path between two nodes in a graph using the A* algorithm. The graph should be represented using an adjacency matrix.

```
#1
import heapq

def heuristic(node, target):
    return abs(node[0] - target[0]) + abs(node[1] - target[1])
def astar(graph, start, target):
    rows, cols = len(graph), len(graph[0])
    visited = [[False for _ in range(cols)] for _ in range(rows)]
    # Priority queue to store nodes to visit, based on their f-score
    open_list = [(0 + heuristic(start, target), start)] # (f, node)
    came_from = {}


    while open_list:
        _, current_node = heapq.heappop(open_list)
        if current_node == target:
            path = []
            while current_node in came_from:
                path.append(current_node)
                current_node = came_from[current_node]
            path.append(start)
            path.reverse()
            return path

        visited[current_node[0]][current_node[1]] = True

        for neighbor in get_neighbors(current_node[0], current_node[1], rows, cols):
            if not visited[neighbor[0]][neighbor[1]] and graph[neighbor[0]][neighbor[1]] != -1:
                new_g = heuristic(current_node, neighbor) + 1
                heapq.heappush(open_list, (new_g + heuristic(neighbor, target), neighbor))
                came_from[neighbor] = current_node

    return None

def get_neighbors(row, col, rows, cols):
    neighbors = []
    if row > 0:
        neighbors.append((row - 1, col))
    if row < rows - 1:
        neighbors.append((row + 1, col))
```

```
    if col > 0:
        neighbors.append((row, col - 1))
    if col < cols - 1:
        neighbors.append((row, col + 1))
    return neighbors

# Example usage:
if __name__ == "__main__":
    # Replace the following adjacency matrix with your own graph representation.
    # -1 represents an obstacle, and any non-negative integer represents the cost of moving
between nodes.
    graph = [
        [0, 3, -1, 2],
        [2, -1, 5, 1],
        [4, 2, 1, 3],
        [-1, 1, 2, 0]
    ]
    start_node = (0, 0)
    target_node = (3, 3)


    shortest_path = astar(graph, start_node, target_node)


    if shortest_path:
        print("Shortest Path:", shortest_path)
        print("Total Cost:", sum(graph[node[0]][node[1]] for node in shortest_path))
    else:
        print("No path found!")
```

```
Shortest Path: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (3, 3)]
Total Cost: 12
```

2. Write a Python program to navigate a grid map with obstacles to find the shortest path from a start node to a goal node using A*. The map should be represented as a 2D array.

```
#2
import heapq


def heuristic(node, target):
    # Calculate the Manhattan distance as a heuristic
    return abs(node[0] - target[0]) + abs(node[1] - target[1])


def astar(grid, start, goal):
```

```
    rows, cols = len(grid), len(grid[0])
    visited = [[False for _ in range(cols)] for _ in range(rows)]
    open_list = [(0 + heuristic(start, goal), start)] # (f, node)
    came_from = {}


    while open_list:
        _, current_node = heapq.heappop(open_list)
        if current_node == goal:
            path = []
            while current_node in came_from:
                path.append(current_node)
                current_node = came_from[current_node]
            path.append(start)
            path.reverse()
            return path


        visited[current_node[0]][current_node[1]] = True


        for neighbor in get_neighbors(current_node[0], current_node[1], rows, cols):
            if not visited[neighbor[0]][neighbor[1]] and grid[neighbor[0]][neighbor[1]] != -1:
                new_g = grid[neighbor[0]][neighbor[1]] + heuristic(current_node, neighbor)
                heapq.heappush(open_list, (new_g + heuristic(neighbor, goal), neighbor))
                came_from[neighbor] = current_node


    return None


def get_neighbors(row, col, rows, cols):
    neighbors = []
    if row > 0:
        neighbors.append((row - 1, col))
    if row < rows - 1:
        neighbors.append((row + 1, col))
    if col > 0:
        neighbors.append((row, col - 1))
    if col < cols - 1:
        neighbors.append((row, col + 1))
    return neighbors


# Example usage:
if __name__ == "__main__":
```

```
    # Replace the following 2D array with your own grid map representation.
    # -1 represents an obstacle, and any non-negative integer represents the cost of moving
between cells.
    grid = [
        [0, 3, -1, 2],
        [2, -1, 5, 1],
        [4, 2, 1, 3],
        [-1, 1, 2, 0]
    ]
    start_node = (0, 0)
    goal_node = (3, 3)


    shortest_path = astar(grid, start_node, goal_node)


    if shortest_path:
        print("Shortest Path:", shortest_path)
        print("Total Cost:", sum(grid[node[0]][node[1]] for node in shortest_path))
    else:
        print("No path found!")
```

```
Shortest Path: [(0, 0), (1, 0), (2, 0), (2, 1), (3, 1), (3, 2), (3, 3)]
Total Cost: 11
```

3. Implement the A* algorithm in Python to solve the 8 puzzle problem. The state space should be represented using a list.

```
#3
import heapq

def heuristic(state, target):
    # Manhattan distance heuristic
    return sum(abs(s // 3 - t // 3) + abs(s % 3 - t % 3) for s, t in zip(state, target))

def get_neighbors(state):
    neighbors = []
    empty_tile_idx = state.index(0)

    for shift in [-3, 3, -1, 1]:
        neighbor_idx = empty_tile_idx + shift

        if 0 <= neighbor_idx < len(state) and abs(empty_tile_idx // 3 - neighbor_idx // 3) <= 1:
            neighbor = list(state)
            neighbor[empty_tile_idx], neighbor[neighbor_idx] = neighbor[neighbor_idx],
```

```
neighbor[empty_tile_idx]
            neighbors.append(tuple(neighbor))

    return neighbors

def a_star_8puzzle(initial_state, target_state):
    open_list = [(0, initial_state)]
    came_from = {}
    g_scores = {initial_state: 0}

    while open_list:
        _, current_state = heapq.heappop(open_list)

        if current_state == target_state:
            path = []
            while current_state is not None:
                path.append(current_state)
                current_state = came_from.get(current_state)
            return path[::-1]

        for neighbor_state in get_neighbors(current_state):
            tentative_g_score = g_scores[current_state] + 1
            if neighbor_state not in g_scores or tentative_g_score < g_scores[neighbor_state]:
                came_from[neighbor_state] = current_state
                g_scores[neighbor_state] = tentative_g_score
                f_score = tentative_g_score + heuristic(neighbor_state, target_state)
                heapq.heappush(open_list, (f_score, neighbor_state))

    return None

# Example usage:
initial_state = (1, 2, 3, 0, 4, 6, 7, 5, 8)
target_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)

solution = a_star_8puzzle(initial_state, target_state)

if solution:
    print("Shortest Path:")
    for step, state in enumerate(solution):
        print(f"Step {step}: {state[0:3]}\n        {state[3:6]}\n        {state[6:9]}")
else:
    print("No solution found.")
```

```
Shortest Path:
Step 0: (1, 2, 3)
        (0, 4, 6)
        (7, 5, 8)
Step 1: (1, 2, 3)
        (4, 0, 6)
        (7, 5, 8)
Step 2: (1, 2, 3)
        (4, 5, 6)
        (7, 0, 8)
Step 3: (1, 2, 3)
        (4, 5, 6)
        (7, 8, 0)
```

4. Write a Python program to find the shortest path in a Pacman grid map from Pacman's position to the closest food pellet using A*. Represent the grid map as a 2D array.

```
#4
import heapq


def heuristic(node, target):
    # Calculate the Manhattan distance as a heuristic
    return abs(node[0] - target[0]) + abs(node[1] - target[1])


def astar(grid, start, target):
    rows, cols = len(grid), len(grid[0])
    visited = [[False for _ in range(cols)] for _ in range(rows)]
    open_list = [(0 + heuristic(start, target), start)] # (f, node)
    came_from = {}


    while open_list:
        _, current_node = heapq.heappop(open_list)
        if current_node == target:
            path = []
            while current_node in came_from:
                path.append(current_node)
                current_node = came_from[current_node]
            path.append(start)
            path.reverse()
            return path
```

```
        visited[current_node[0]][current_node[1]] = True


        for neighbor in get_neighbors(current_node[0], current_node[1], rows, cols):
            if not visited[neighbor[0]][neighbor[1]] and grid[neighbor[0]][neighbor[1]] != 1:
                new_g = heuristic(current_node, neighbor) + 1
                heapq.heappush(open_list, (new_g + heuristic(neighbor, target), neighbor))
                came_from[neighbor] = current_node


    return None


def get_neighbors(row, col, rows, cols):
    neighbors = []
    if row > 0:
        neighbors.append((row - 1, col))
    if row < rows - 1:
        neighbors.append((row + 1, col))
    if col > 0:
        neighbors.append((row, col - 1))
    if col < cols - 1:
        neighbors.append((row, col + 1))
    return neighbors


# Example usage:
if __name__ == "__main__":
    # Replace the following 2D array with your own Pacman grid map representation.
    # 0 represents an empty cell, 1 represents a wall, and 2 represents a food pellet.
    grid = [
        [0, 0, 0, 1],
        [1, 0, 2, 1],
        [0, 0, 1, 1],
        [1, 0, 0, 0]
    ]
    pacman_position = (0, 0)


    food_pellet_positions = [(1, 2)]


    shortest_path = None
    closest_distance = float('inf')
```

```
    for food_pellet in food_pellet_positions:
        path = astar(grid, pacman_position, food_pellet)
        if path and len(path) - 1 < closest_distance:
            shortest_path = path
            closest_distance = len(path) - 1


    if shortest_path:
        print("Shortest Path:", shortest_path)
    else:
        print("No path found!")
```

```
Shortest Path: [(0, 0), (0, 1), (0, 2), (1, 2)]
```

5. Implement the A* algorithm in Python to find the optimal sequence of chess moves that leads to checkmate. Represent the chess board as a 2D array.

```
#5
import heapq


def heuristic(node, target):
    # Calculate the Manhattan distance as a heuristic
    return abs(node[0] - target[0]) + abs(node[1] - target[1])


def is_valid_move(pos):
    # Check if the position is within the chessboard boundaries
    return 0 <= pos[0] < 8 and 0 <= pos[1] < 8


def get_neighbors(pos):
    # Generate all valid knight moves from the given position
    knight_moves = [(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)]
    neighbors = [(pos[0] + dr, pos[1] + dc) for dr, dc in knight_moves]
    return [neighbor for neighbor in neighbors if is_valid_move(neighbor)]


def astar(start, target):
    open_list = [(0 + heuristic(start, target), start)]  # (f, node)
    came_from = {}
    g_score = {start: 0}
```

```
    while open_list:
        _, current_node = heapq.heappop(open_list)
        if current_node == target:
            path = []
            while current_node in came_from:
                path.append(current_node)
                current_node = came_from[current_node]
            path.append(start)
            path.reverse()
            return path


        for neighbor in get_neighbors(current_node):
            new_g = g_score[current_node] + 1
            if neighbor not in g_score or new_g < g_score[neighbor]:
                g_score[neighbor] = new_g
                heapq.heappush(open_list, (new_g + heuristic(neighbor, target), neighbor))
                came_from[neighbor] = current_node


    return None


# Example usage:
if __name__ == "__main__":
    start_position = (0, 0)
    target_position = (7, 7)


    shortest_path = astar(start_position, target_position)


    if shortest_path:
        print("Shortest Path:", shortest_path)
    else:
        print("No path found!")
```

```
Shortest Path: [(0, 0), (1, 2), (2, 4), (4, 5), (3, 7), (5, 6), (7, 7)]
```

6. Write a Python program to find the shortest driving route between two locations using A* search and OpenStreetMap data. Use graph representation with nodes as intersections and edges as road segments.

```
#6
import networkx as nx
from geopy.distance import geodesic


# Helper function to calculate the distance between two geographic coordinates
def get_distance(coord1, coord2):
    return geodesic(coord1, coord2).kilometers


# Create a graph to represent the road network
G = nx.Graph()


# Add intersections as nodes to the graph with geographic coordinates
G.add_node("Intersection1", pos=(latitude1, longitude1))
G.add_node("Intersection2", pos=(latitude2, longitude2))
# Add more intersections as needed...


# Add road segments (edges) between intersections with appropriate weights (distances)
G.add_edge("Intersection1", "Intersection2", weight=get_distance((latitude1, longitude1),
(latitude2, longitude2)))
# Add more road segments as needed...


# Implement A* search algorithm
def astar(graph, start, target):
    open_list = [(0 + get_distance(graph.nodes[start]['pos'], graph.nodes[target]['pos']), start)]  #
(f, node)
    came_from = {}
    g_score = {start: 0}


    while open_list:
        _, current_node = min(open_list)
        if current_node == target:
            path = []
            while current_node in came_from:
                path.append(current_node)
                current_node = came_from[current_node]
            path.append(start)
```

```
            path.reverse()
            return path


        open_list.remove((_, current_node))
        for neighbor, data in graph[current_node].items():
            new_g = g_score[current_node] + data['weight']
            if neighbor not in g_score or new_g < g_score[neighbor]:
                g_score[neighbor] = new_g
                heapq.heappush(open_list, (new_g + get_distance(graph.nodes[neighbor]['pos'],
graph.nodes[target]['pos']), neighbor))
                came_from[neighbor] = current_node


    return None


# Example usage:
if __name__ == "__main__":
    start_location = "Intersection1"
    target_location = "Intersection2"
    shortest_path = astar(G, start_location, target_location)


    if shortest_path:
        print("Shortest Path:", shortest_path)
        total_distance = sum(G.edges[u, v]['weight'] for u, v in zip(shortest_path,
shortest_path[1:]))
        print("Total Distance:", total_distance, "km")
    else:
        print("No path found!")
```

7. Develop an A* based Pathfinding AI in Python for a tower defense game on a grid map. The map is represented as a 2D array with obstacles.

```
# 7
import heapq

def heuristic(node, goal):
    # Calculate the Manhattan distance as the heuristic for A* algorithm
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

def get_neighbors(grid, current):
    neighbors = []
    rows, cols = len(grid), len(grid[0])
```

```
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    for dx, dy in directions:
        x, y = current[0] + dx, current[1] + dy
        if 0 <= x < rows and 0 <= y < cols and grid[x][y] == 0:
            neighbors.append((x, y))

    return neighbors

def a_star_pathfinding(grid, start, goal):
    open_list = [(0, start)]
    closed_set = set()

    g_scores = {point: float('inf') for point in np.ndindex(grid.shape)}
    g_scores[start] = 0

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

        if current == goal:
            path = []
            while current is not None:
                path.append(current)
                current = came_from.get(current)
            return path[::-1]

        closed_set.add(current)

        for neighbor in get_neighbors(grid, current):
            tentative_g_score = g_scores[current] + 1
            if tentative_g_score < g_scores[neighbor]:
                came_from[neighbor] = current
                g_scores[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic(neighbor, goal)
                heapq.heappush(open_list, (f_score, neighbor))

    return None

# Example usage:
grid = np.array([
    [0, 0, 1, 0, 0],
    [1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0]
```

```
])

start_point = (0, 0)
goal_point = (4, 4)

came_from = {}
shortest_path = a_star_pathfinding(grid, start_point, goal_point)

if shortest_path:
    print("Shortest Path:")
    for point in shortest_path:
        print(point)
else:
    print("No path found from the start to the goal in the grid map.")
```

```
Shortest Path:
(0, 0)
(0, 1)
(1, 1)
(2, 1)
(2, 2)
(3, 2)
(3, 3)
(3, 4)
(4, 4)
```

8. Implement a Python program using A* to find the optimal ordering of jobs on a single machine to minimize total weighted completion time.

```
# 8
import itertools

def total_weighted_completion_time(perm, processing_times, weights):
    total_time = 0
    completion_time = 0

    for job in perm:
        completion_time += processing_times[job]
        total_time += completion_time * weights[job]

    return total_time

def find_optimal_ordering(processing_times, weights):
    num_jobs = len(processing_times)
    min_total_time = float('inf')
```

```
    optimal_ordering = None

    for perm in itertools.permutations(range(num_jobs)):
        total_time = total_weighted_completion_time(perm, processing_times, weights)
        if total_time < min_total_time:
            min_total_time = total_time
            optimal_ordering = perm

    return optimal_ordering

# Example usage:
processing_times = [3, 5, 2]   # Processing times of jobs [Job 1, Job 2, Job 3]
weights = [2, 1, 3]            # Weights of jobs [Job 1, Job 2, Job 3]

optimal_ordering = find_optimal_ordering(processing_times, weights)

if optimal_ordering:
    print("Optimal Job Ordering:")
    for job in optimal_ordering:
        print(f"Job {job + 1}")
else:
    print("No jobs found.")
```

```
Optimal Job Ordering:
Job 3
Job 1
Job 2
```

9. Write a Python program to navigate a robot through a maze to the exit using A* search. Represent the maze as a 2D array.

```
# 9
import numpy as np
import heapq

def heuristic(node, goal):
    # Calculate the Manhattan distance as the heuristic for A* algorithm
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

def get_neighbors(maze, current):
    neighbors = []
    rows, cols = len(maze), len(maze[0])
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    for dx, dy in directions:
        x, y = current[0] + dx, current[1] + dy
```

```
        if 0 <= x < rows and 0 <= y < cols and maze[x][y] == 0:
            neighbors.append((x, y))

    return neighbors

def a_star_maze(maze, start, goal):
    open_list = [(0, start)]
    closed_set = set()

    g_scores = {point: float('inf') for point in np.ndindex(maze.shape)}
    g_scores[start] = 0

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

        if current == goal:
            path = []
            while current is not None:
                path.append(current)
                current = came_from.get(current)
            return path[::-1]

        closed_set.add(current)

        for neighbor in get_neighbors(maze, current):
            tentative_g_score = g_scores[current] + 1
            if tentative_g_score < g_scores[neighbor]:
                came_from[neighbor] = current
                g_scores[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic(neighbor, goal)
                heapq.heappush(open_list, (f_score, neighbor))

    return None

# Example usage:
maze = np.array([
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
])

start_point = (0, 0)
goal_point = (4, 4)
```

```
came_from = {}
shortest_path = a_star_maze(maze, start_point, goal_point)

if shortest_path:
    print("Shortest Path:")
    for point in shortest_path:
        print(point)
else:
    print("No path found from the start to the goal in the maze.")
```

```
Shortest Path:
(0, 0)
(1, 0)
(2, 0)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(3, 4)
(4, 4)
```

10. Develop a Python program using A* to find the shortest path between multiple points on a map. Allow specifying arbitrary start and end points. Represent the map as a graph.

```
# 10
import numpy as np
import heapq

def heuristic(node, goal):
    # Calculate the Manhattan distance as the heuristic for A* algorithm
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

def a_star(graph, points, start, goal):
    open_list = [(0, start)]
    closed_set = set()

    g_scores = {point: float('inf') for point in points}
    g_scores[start] = 0

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

        if current == goal:
            path = []
```

```
            while current is not None:
                path.append(current)
                current = came_from.get(current)
            return path[::-1]

        closed_set.add(current)

        for neighbor, weight in enumerate(graph[points.index(current)]):
            if weight == 0 or points[neighbor] in closed_set:
                continue

            tentative_g_score = g_scores[current] + weight
            if tentative_g_score < g_scores[points[neighbor]]:
                came_from[points[neighbor]] = current
                g_scores[points[neighbor]] = tentative_g_score
                f_score = tentative_g_score + heuristic(points[neighbor], goal)
                heapq.heappush(open_list, (f_score, points[neighbor]))

    return None

# Example usage:
points = [(0, 0), (1, 2), (2, 2), (3, 4), (4, 4)]
# For simplicity, let's assume a 5x5 grid with coordinates (0, 0) to (4, 4)

# The adjacency matrix representing the map's connections/costs between points
adjacency_matrix = np.array([
    [0, 5, 0, 0, 0],
    [5, 0, 4, 0, 0],
    [0, 4, 0, 3, 0],
    [0, 0, 3, 0, 2],
    [0, 0, 0, 2, 0]
])

start_point = (0, 0)
goal_point = (4, 4)

came_from = {}
shortest_path = a_star(adjacency_matrix, points, start_point, goal_point)

if shortest_path:
    print("Shortest Path:")
    for point in shortest_path:
        print(point)
else:
    print("No path found between the start and goal points.")
```

```
Shortest Path:
(0, 0)
(1, 2)
(2, 2)
(3, 4)
(4, 4)
```