Documentation

Project1: Robot Path Planning
Team Members: Kaiwen Guo, Andy Lee
CS4613

Instructions:

To run **findpath.py**, type the following in your terminal:

python findpath.py <input_filename.txt> <output_filename.txt>

or try

python3 findpath.py <input_filename.txt> <output_filename.txt>

To run **vis.py**, type the following in your terminal:

python vis.py <input_filename.txt>

or try

python3 vis.py <input_filename.txt>

Source Code:

```python
import sys
import numpy as np
from heapq import heappush, heappop
import math

k = 2  # Penalty constant for angle change
# Calculates Euclidean distance heuristic from current
# position to the goal (estimates the cost)
def heuristic(curr_row, curr_column, end_row, end_column):
    return math.hypot(end_row - curr_row, end_column - curr_column)
```

```
# Identifies valid cells that the robot can move to
# and returns a list of valid cells (neighbors)
def find_neighbors(curr_row, curr_column, matrix):
    neighbors = []
    rows, cols = matrix.shape # Gets num of rows and cols
    directions = [  # (d_row, d_col, action)
        (0, 1, 0),    # Right
        (1, 1, 1),    # Up-Right
        (1, 0, 2),    # Up
        (1, -1, 3),   # Up-Left
        (0, -1, 4),   # Left
        (-1, -1, 5),  # Down-Left
        (-1, 0, 6),   # Down
        (-1, 1, 7)    # Down-Right
    ]

    # Iterates through all directions and checks for validity.
    # Calculates valid moves and stores them in a list (neighbors)
    for d_row, d_col, action in directions:
        new_row, new_column = curr_row + d_row, curr_column + d_col
        if 0 <= new_row < rows and 0 <= new_column < cols:
            if matrix[new_row, new_column] != '1':
                if abs(d_row) == 1 and abs(d_col) == 1: # Diagonal case
                    # Cells adjacent in horizontal and vertical directions
                    adj_cell1 = (curr_row + d_row, curr_column)
                    adj_cell2 = (curr_row, curr_column + d_col)
                    # Check if at least one adjacent cell is walkable
                    walkable_adj_cell1 = (0 <= adj_cell1[0] < rows and 0 <=
adj_cell1[1] < cols and
                                          matrix[adj_cell1] != '1')
                    walkable_adj_cell2 = (0 <= adj_cell2[0] < rows and 0 <=
adj_cell2[1] < cols and
                                          matrix[adj_cell2] != '1')
                    if walkable_adj_cell1 or walkable_adj_cell2:
                        neighbors.append((new_row, new_column, d_row,
d_col, action))
                else: # Non-diagonal case
                    neighbors.append((new_row, new_column, d_row, d_col,
action))
        #print(neighbors)
    return neighbors

# Finds shortest path to the goal
```

```
def a_star_search(matrix, start_row, start_column, end_row, end_column):
    frontier = []
    # θ(s) is None at the start
    heappush(frontier, (heuristic(start_row, start_column, end_row,
end_column),
                        start_row, start_column, None, 0, None,
heuristic(start_row, start_column, end_row, end_column)))  # priority, row,
col, theta_s, cost, action, f(n)
    came_from = {}  # To track the path: (current node) -> (previous node,
action, f(n))
    cost_so_far = {(start_row, start_column): (0, None)}  # cost and
theta_s
    nodes_generated = 1  # Start node is generated

    while frontier:
        priority, curr_row, curr_column, theta_s, current_cost, action, f_n
= heappop(frontier)
        if (curr_row, curr_column) == (end_row, end_column): # Check if the
goal is reached
            return reconstruct_path(came_from, start_row, start_column,
end_row, end_column, matrix), current_cost, nodes_generated
        # Explore neighbors
        for neighbor_row, neighbor_column, d_row, d_col, action in
find_neighbors(curr_row, curr_column, matrix):
            new_node = (neighbor_row, neighbor_column)
            if new_node in cost_so_far and cost_so_far[new_node][0] <=
current_cost:
                continue  # Skip if we've already found a better path

            # Compute θ(s') from movement vector
            theta_s_prime = math.degrees(math.atan2(d_row, d_col)) % 360

            # Compute c_d(s, a, s')
            if abs(d_row) + abs(d_col) == 1:  # Orthogonal move
                c_d = 1
            else:  # Diagonal move
                c_d = math.sqrt(2)

            # Compute c_a(s, a, s')
            if theta_s is None:  # Initial state
                c_a = 0
            else:
                delta_theta = abs(theta_s_prime - theta_s)
```

```
                if delta_theta > 180:
                    delta_theta = 360 - delta_theta
                c_a = k * (delta_theta / 180)

            c = c_d + c_a
            new_cost = current_cost + c

            if ((neighbor_row, neighbor_column) not in cost_so_far or
                    new_cost < cost_so_far[(neighbor_row,
neighbor_column)][0]):
                cost_so_far[(neighbor_row, neighbor_column)] = (new_cost,
theta_s_prime)
                total_estimated_cost = new_cost + heuristic(neighbor_row,
neighbor_column, end_row, end_column)
                heappush(frontier, (total_estimated_cost, neighbor_row,
neighbor_column, theta_s_prime, new_cost, action, total_estimated_cost))
                came_from[(neighbor_row, neighbor_column)] = ((curr_row,
curr_column), action, total_estimated_cost)
                nodes_generated += 1

    # If we exit the loop without finding the end, log that no path was
found
    return None, None, nodes_generated
```

```python
def reconstruct_path(came_from, start_row, start_column, end_row,
end_column, matrix):
    path = []
    actions = []
    f_values = []
    current = (end_row, end_column)
    while current != (start_row, start_column):
        prev_info = came_from.get(current)
        if prev_info is None:
            # No path found
            return None
        prev_node, action, f_n = prev_info
        path.append(current)
        actions.append(action)
        f_values.append(f_n)
        current = prev_node
    path.append((start_row, start_column))
    f_values.append(heuristic(start_row, start_column, end_row,
end_column))  # f(n) of start node
    path.reverse()
    actions.reverse()
    f_values.reverse()
    # Update the matrix with the path (excluding start and goal)
    for r, c in path[1:-1]:
        if matrix[r, c] == '0':
            matrix[r, c] = '4'
    # Ensure we don't overwrite the start and goal positions
    matrix[start_row, start_column] = '2'
    matrix[end_row, end_column] = '5'
    return (path, actions, f_values)
```

```python
def main():
    if len(sys.argv) < 3:
        print("Usage: python script.py <grid_file> <output_file>")
        sys.exit(1)
    file_path = sys.argv[1]
    output_file = sys.argv[2]
    matrix = []

    try:
        with open(file_path, 'r') as file:
            # Read the start and end positions
            first_line = file.readline().strip()
            start_column, start_row, end_column, end_row = map(int,
first_line.split())
            print(f"Start Position: ({start_row}, {start_column})")
            print(f"End Position: ({end_row}, {end_column})")

            # Read the grid lines
            for line in file:
                line = line.strip()
                if not line:
                    continue  # Skip empty lines
                # Split the line by spaces
                tokens = line.split()
                matrix.append(tokens)
    except Exception as e:
        print(f"Error reading file: {e}")
        sys.exit(1)

    # Reverses the matrix to match the coordinate system
    matrix = matrix[::-1]

    # Convert matrix to numpy array
    try:
        matrix_np = np.array(matrix)
    except Exception as e:
        print(f"Error converting grid to numpy array: {e}")
        sys.exit(1)

    rows, cols = matrix_np.shape
    # Print the dimensions of the grid
    print(f"Grid dimensions: {rows} rows x {cols} columns")
```

```
    # Check that start and end positions are within bounds
    if not (0 <= start_row < rows and 0 <= start_column < cols):
        print(f"Start position ({start_row}, {start_column}) is out of
bounds. Grid size is {rows}x{cols}.")
        sys.exit(1)
    if not (0 <= end_row < rows and 0 <= end_column < cols):
        print(f"End position ({end_row}, {end_column}) is out of bounds.
Grid size is {rows}x{cols}.")
        sys.exit(1)
    if matrix_np[start_row, start_column] == '1':
        print(f"Start position is not walkable. Value at start position:
{matrix_np[start_row, start_column]}")
        sys.exit(1)
    if matrix_np[end_row, end_column] == '1':
        print(f"End position is not walkable. Value at end position:
{matrix_np[end_row, end_column]}")
        sys.exit(1)

    # Perform A* search
    result, total_cost, nodes_generated = a_star_search(matrix_np,
start_row, start_column, end_row, end_column)
    if result:
        path, actions, f_values = result
        depth = len(path) - 1  # Root node is at level 0
        with open(output_file, 'w') as f_out:
            # Line 1: Depth level d
            f_out.write(f"{depth}\n")
            # Line 2: Total number of nodes N generated
            f_out.write(f"{nodes_generated}\n")
            # Line 3: Sequence of moves (actions)
            f_out.write(' '.join(map(str, actions)) + '\n')
            # Line 4: f(n) values along the solution path
            f_out.write(' '.join(f"{fv:.2f}" for fv in f_values) + '\n')
            # Lines 5 onward: Updated grid
            for row in matrix_np[::-1]:  # Reverse again to match your
output format
                f_out.write(' '.join(row) + '\n')
        print(f"Output written to {output_file}")
    else:
        print("No path found.")

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

Output Files:

Input1.txt result, k=2:
31
454
0 0 7 7 7 7 7 7 7 7 7 7 0 7 7 7 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
32.57 32.62 32.68 33.33 33.49 33.67 33.87 34.09 34.33 34.60 34.91 35.25 35.64 36.14 37.09 37.60 38.18 38.70 38.73 38.75 38.79 38.83 38.87 38.93 39.00 39.09 39.22 39.38 39.63 40.13 40.13 40.13
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 0 0 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 1 1 0 0 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 1 1 1 0 0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 2 4 4 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 4 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 4 0 0 1 1 1 0 0 1 1 0 1 1 1 1 1 0 0 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 0 4 0 0 1 1 0 0 1 1 0 1 1 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 4 0 1 1 1 0 1 0 0 1 1 1 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 4 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 4 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 1 1 0 1 1 0 0 0 0 4 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 1 1 0 1 1 0 0 0 1 1 4 0 0 0 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 1 1 0 4 4 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 0 5 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 1 1 0 0 0 0 0 1 1 0 0 1 1 4 1 1 1 1 0 0 1 1 1 0 0 0 0 1 1 4 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0 1 0 4 0 0 0 0 0 0 0 0 1 1 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Input1.txt, k=4 result:
31
466
0 0 7 7 7 7 7 7 7 7 7 7 0 7 7 7 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
32.57 32.62 32.68 33.83 33.99 34.17 34.37 34.59 34.83 35.10 35.41 35.75 36.14 37.14 38.59 39.10 39.68 40.70 40.73 40.75 40.79 40.83 40.87 40.93 41.00 41.09 41.22 41.38 41.63 42.63 42.63 42.63
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 0 0 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 1 1 0 0 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 1 1 1 0 0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 2 4 4 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 4 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 4 0 0 1 1 1 0 0 1 1 0 1 1 1 1 1 0 0 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 0 4 0 0 1 1 0 0 1 1 0 1 1 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 4 0 1 1 1 0 1 0 0 1 1 1 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 4 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 4 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 1 1 0 1 1 0 0 0 0 4 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 1 1 0 1 1 0 0 0 1 1 4 0 0 0 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 1 1 0 4 4 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 0 5 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 1 1 0 0 0 0 0 1 1 0 0 1 1 4 1 1 1 1 0 0 1 1 1 0 0 0 0 1 1 4 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0 1 0 4 0 0 0 0 0 0 0 0 1 1 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0