CMPE252: Artificial Intelligence and Data Engineering

Homework 2: Implementing and comparing the performance of Dijkstra, A *, and Weighted A * algorithm.

Dijkstra's Algorithm:

- It is used to find a path with minimum cost between two points.
- It is a greedy algorithm
- Time complexity depends on the number of nodes(n) and edges(e), which is o(n*e)
- This algorithm was created and published by Dr. Edsger W. Dijkstra in 1959

  Limitations of Dijkstra's Algorithm:
  - There is no sense of direction towards the destination due to which it may take longer to compute results, this limitation is overcome by the A* search algorithm.
  - The algorithm cannot accommodate negative weights.

  Applications
  - Maps and Navigation systems, Transportation and Logistics
  - Computer Networking (to find the shortest path for data packets)
  - Game Development (to find the shortest path between entities)

A * Algorithm:

- This is an enhanced version of Dijkstra's algorithm.
- It is goal-directed and uses a heuristic to guide the search
- A* uses a heuristic function that estimates the cost from the current node to the target node. This heuristic helps A* prioritize paths that are more likely to lead to the goal.
- Time complexity depends on nodes(n) and edges(e) O(e + n*log(n)).
- There is another variation in A* called Weighted A*. You can use weights to adjust the time complexity of the algorithm.

  Limitations of Dijkstra's Algorithm:
  - You can calculate the shortest distance between the specified start and the end node only.
  - The efficiency is majorly dependent on the heuristics of the algorithm.
  - The algorithm cannot accommodate negative weights.

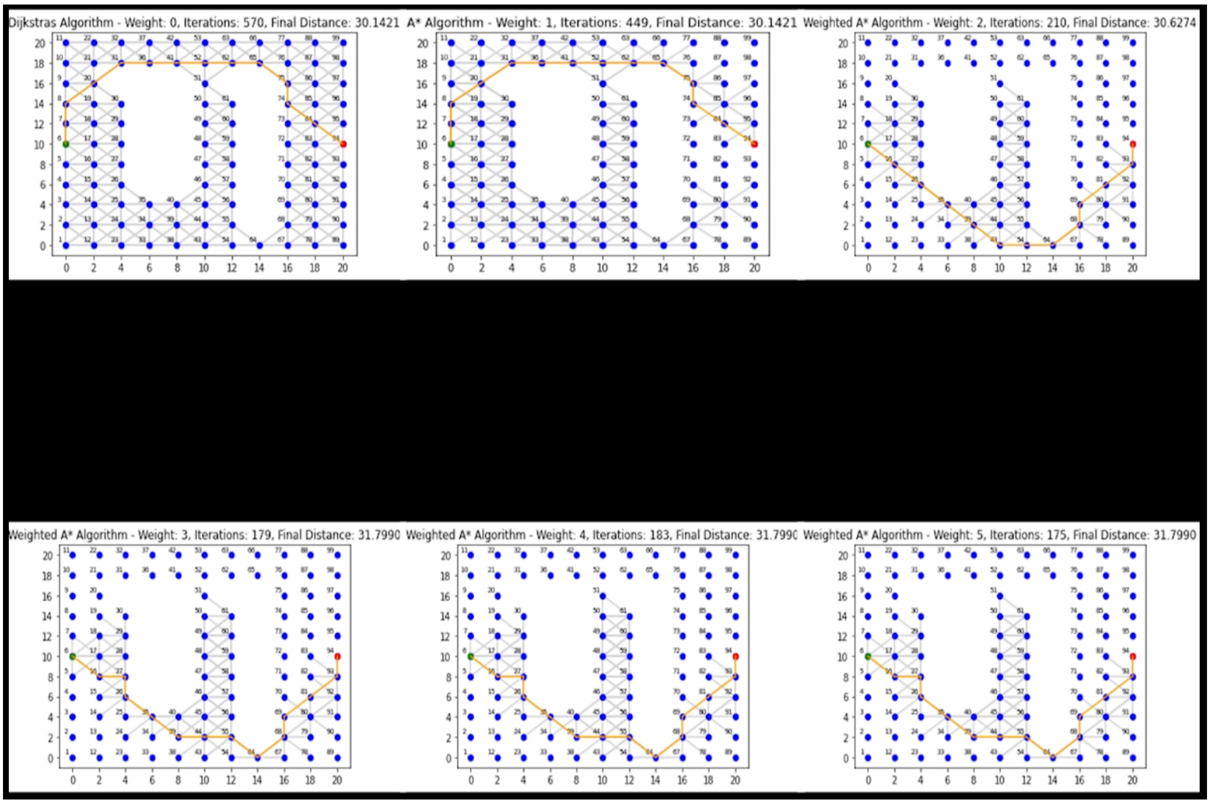
Final Table:

	Dijkstra's (w = 0)	A * (w = 1)	Weighted A* (w = 2)	Weighted A* (w = 2)	Weighted A* (w = 2)	Weighted A* (w = 2)
Final Cost	30.1421	30.1421	30.6274	31.7990	31.7990	31.7990
Iterations	570	449	210	179	183	175

Final Output:

```
6 7 8 20 31 36 41 52 62 65 75 74 84 94
0 2.0000 4.0000 6.8284 9.6569 11.6569 13.6569 15.6569 17.6569 19.6569 22.4853 24.4853 27.3137 30.1421
6 7 8 20 31 36 41 52 62 65 75 74 84 94
0 2.0000 4.0000 6.8284 9.6569 11.6569 13.6569 15.6569 17.6569 19.6569 22.4853 24.4853 27.3137 30.1421
6 16 26 35 39 43 54 64 68 69 81 93 94
0 2.8284 5.6569 8.4853 11.3137 14.1421 16.1421 18.1421 20.9706 22.9706 25.7990 28.6274 30.6274
6 16 27 26 35 39 44 55 64 68 69 81 93 94
0 2.8284 4.8284 6.8284 9.6569 12.4853 14.4853 16.4853 19.3137 22.1421 24.1421 26.9706 29.7990 31.7990
6 16 27 26 35 39 44 55 64 68 69 81 93 94
0 2.8284 4.8284 6.8284 9.6569 12.4853 14.4853 16.4853 19.3137 22.1421 24.1421 26.9706 29.7990 31.7990
6 16 27 26 35 39 44 55 64 68 69 81 93 94
0 2.8284 4.8284 6.8284 9.6569 12.4853 14.4853 16.4853 19.3137 22.1421 24.1421 26.9706 29.7990 31.7990
```

Final Path:



Implementation

1. **How to run the code.**
    a. In the zip file uploaded on Canvas is a Python script with the name hw2.py. Run this Python script to execute the code. Ensure the input and coords files are in the same directory as the script.
    b. After the script is executed it should generate an output.txt file with the desired output.

2. **How to make a video.**
    a. I captured and saved all the plot figures to a PNG file and then stitched them to make a video using ffmpeg.
    b. I used the below-mentioned command to stitch the video.
    .\ffmpeg.exe -framerate 10 -i hw1_5_%d.png -c:v libx264 -r 30 -vf "scale=1280:720" HomeWork5.mp4

3. **Code:**

```
import matplotlib.pyplot as plt
import math

# Reading coordinates and input file
f_coords = open("coords.txt", "r")
f_input = open("input.txt", "r")

# Contents of coords file
coords = f_coords.readlines()
coords = [i.replace('\n',"") for i in coords]
coords_x = [int(float(i.split(" ")[0])) for i in coords]
coords_y = [int(float(i.split(" ")[1])) for i in coords]

# contents of input file
total_nodes = int(f_input.readline())
start = int(f_input.readline())
end = int(f_input.readline())
dir_wei = f_input.readlines()
dir_wei = [i.replace('\n',"") for i in dir_wei]

# creating the graph using dictionary. Stores node as key, and stores neighbor and
weight as values
graph = {i: [] for i in range(1, total_nodes + 1)}
for i in range(len(dir_wei)):
    node, neigh_node, weight = map(float, dir_wei[i].split())
    graph[node].append((neigh_node, weight))

def plotGraph1(coords_x, coords_y):  # Takes input the x and y coords of the points
    # ploting the graph
    plt.xticks(range(0,22,2))
    plt.yticks(range(0,22,2))
    i=1
    for x,y in zip(coords_x,coords_y):
        label = "{}".format(i)
        plt.annotate(label, (x,y), textcoords="offset points", xytext=(-7,3.5), ha='center',
size=7.5, zorder=3)
        i+=1
        plt.scatter(x, y, c ="blue", zorder=2)

    start_x = coords_x[int(start)-1] # -1 as the the start number is not index
    start_y = coords_y[int(start)-1]
    end_x = coords_x[int(end)-1]
    end_y = coords_y[int(end)-1]
    plt.scatter(start_x,start_y, c ="green", zorder=4)
    plt.scatter(end_x,end_y, c ="red", zorder=4)
```

```
def aStar(graph, start, end, w=0):
    x=0
    # Initializing the distances as infinity
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    # Visited nodes
    relaxed_nodes = [(0, 0,start)]
    current_node = start

    while len(relaxed_nodes) and current_node!=end:
        # pop element with lowest f
        current_distance, current_f, current_node = relaxed_nodes.pop(0)
        # get neighbor of the current node
        i = 0
        while i<len(graph[current_node]) and current_node!=end:
            neighbor, weight = graph[current_node][i]
            neighbor = int(neighbor)
            neighbor_g = distances[current_node] + weight
            c1 = [coords_x[end-1], coords_y[end-1]]
            c2 = [coords_x[neighbor-1], coords_y[neighbor-1]]
            neighbor_h = math.dist(c1,c2)
            neighbor_f = neighbor_g + w*neighbor_h

            # if distance is less than the distance of the neighbor we have in the dict,
update
            if neighbor_g < distances[neighbor]:
                distances[neighbor] = neighbor_g
                relaxed_nodes.append((neighbor_g, neighbor_f, neighbor))

            short_x1  = coords_x[int(current_node)-1]
            short_x2  = coords_x[int(neighbor)-1]
            short_y1  = coords_y[int(current_node)-1]
            short_y2  = coords_y[int(neighbor)-1]
            plt.plot([short_x1, short_x2], [short_y1, short_y2], c="lightgrey", zorder=1)

            if(w==0):
                plt.title("Dijkstras Algorithm - Weight: {}, Iterations: {}".format(w,x))
            elif(w==1):
                plt.title("A-Star Algorithm - Weight: {}, Iterations: {}".format(w,x))
            else:
                plt.title("Weighted A-Star Algorithm - Weight: {}, Iterations: {}".format(w,x))
            x+=1
            i+=1
            relaxed_nodes.sort(key=lambda x: x[1])

            # save graphs to image
            # plt.savefig('hw1_{}_{}.png'.format(w,x))
```

```
    path = []
    current_node = end
    # traverse and check the smallest path from end to start
    while current_node != start:
        path.insert(0, current_node)
        i = 0
        while i<len(graph[current_node]):
            neighbor, weight = graph[current_node][i]
            if distances[current_node] == distances[neighbor] + weight:
                current_node = neighbor
                break
            i+=1

    path.insert(0, start)
    return path, distances, x

def plotShortGraph(shortest_path, distances, x, w):
    for i in range(len(shortest_path)-1):
        p1 = int(shortest_path[i]) - 1
        p2 = int(shortest_path[i+1]) - 1
        short_x1  = coords_x[p1]
        short_x2  = coords_x[p2]
        short_y1  = coords_y[p1]
        short_y2  = coords_y[p2]
        plt.plot([short_x1, short_x2], [short_y1, short_y2], c="orange", zorder=5)
        if(w==0):
            plt.title("Dijkstras Algorithm - Weight: {}, Iterations: {}".format(w,x))
        elif(w==1):
            plt.title("A-Star Algorithm - Weight: {}, Iterations: {}".format(w,x))
        else:
            plt.title("Weighted A-Star Algorithm - Weight: {}, Iterations: {}".format(w,x))
        # plt.savefig('hw1_{}_{}.png'.format(w,x))
        # total number of iterations program took to execute the algorithm
        x+=1
    if(w==0):
        plt.title("Dijkstras Algorithm - Weight: {}, Iterations: {}, Final Distance:
{:.4f}".format(w,x,distances[end]))
    elif(w==1):
        plt.title("A* Algorithm - Weight: {}, Iterations: {}, Final Distance:
{:.4f}".format(w,x,distances[end]))
    else:
        plt.title("Weighted A* Algorithm - Weight: {}, Iterations: {}, Final Distance:
{:.4f}".format(w,x,distances[end]))
    plt.show()

###################################################################
################
```

```
def writeToFile(all_paths, all_distances):
    with open('output.txt', 'w') as f:
        for i in range(len(all_paths)):
            shortest_path = all_paths[i]
            distances = all_distances[i]
            shortestpath_str = ""
            distances_str = ""
            for i in range(len(shortest_path)):
                shortestpath_str = shortestpath_str + " {}".format(int(shortest_path[i]))
                if(distances[shortest_path[i]] == 0):
                    distances_str = distances_str + "0"
                else:
                    distances_str = distances_str + "
{}".format("{:.4f}".format(distances[shortest_path[i]]))

            shortestpath_str = shortestpath_str.strip()
            distances_str = distances_str.strip()
            f.write(shortestpath_str + "\n" + distances_str + "\n")

            print(shortestpath_str)
            print(distances_str)
            print()


all_paths = []
all_distances = []

for i in range(6):
    w=i
    plotGraph1(coords_x, coords_y)
    shortest_path, distances, x = aStar(graph, start, end, w)
    plotShortGraph(shortest_path, distances, x, w)
    all_paths.append(shortest_path)
    all_distances.append(distances)
    print(x)

writeToFile(all_paths, all_distances)
```