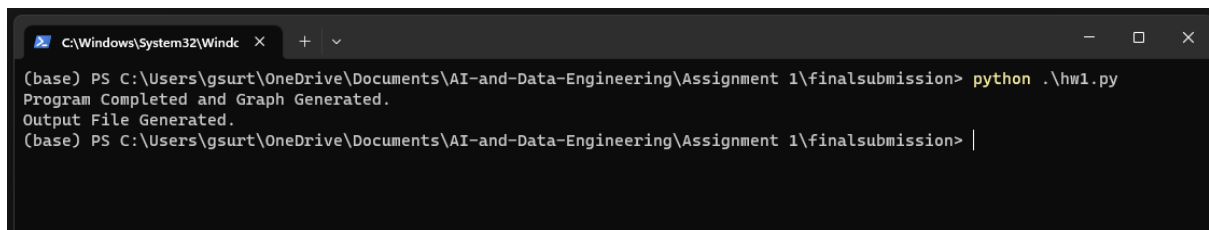


Dijkstra's Algorithm Documentation

How to run the project?

We just need to place the input.txt and coords.txt file and we are good to go.

Windows PowerShell – “python .\hw1.py”

A screenshot of a Windows PowerShell terminal window. The title bar shows the path 'C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe'. The terminal content shows a command prompt at '(base) PS C:\Users\gsurt\OneDrive\Documents\AI-and-Data-Engineering\Assignment 1\finalsubmission>' where the command 'python .\hw1.py' has been entered. The output of the command is displayed as 'Program Completed and Graph Generated.' followed by 'Output File Generated.' on the next line. The prompt is then shown again on the next line: '(base) PS C:\Users\gsurt\OneDrive\Documents\AI-and-Data-Engineering\Assignment 1\finalsubmission> |'.

This will generate the a graph plotting with the final graph plot and an output file with Dijkstra’s algorithm for the shortest path to the destination vertex which we get from the input file.

Introduction

This document gives us documentation for the code implemented in this assignment.

These are the variables used for this code:

1. Dictionary (**self.graph**): Used to represent the graph. Keys represent vertices, and values are lists of tuples representing the neighboring vertices and their corresponding edge weights.
2. Dictionary (**visited**): Used in Dijkstra's algorithm to keep track of visited vertices.
3. Dictionary (**dist**): Used in Dijkstra's algorithm to store the shortest distances from the start vertex to each vertex.
4. Dictionary (**parent**): Used in Dijkstra's algorithm to store the parent vertices in the shortest path tree.
5. List (**visited_vertices**): Used to keep track of visited vertices during the execution of Dijkstra's algorithm.
6. List (**path**): Used to store the vertices in the shortest path.
7. List (**distances_between**): Used to store distances between nodes on the path.

Code Implementation and Explanation:

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v, w):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append((v, w))

    def min_distance(self, dist, visited):
        min_dist = float('inf')
        min_vertex = None
        for vertex in self.graph:
            if not visited[vertex] and dist[vertex] < min_dist:
                min_dist = dist[vertex]
                min_vertex = vertex
        return min_vertex
```

Class: Graph

Constructor: __init__(self)

- Initializes a **Graph** object with an empty dictionary to store the graph representation.

Method: add_edge(self, u, v, w)

- Adds a weighted edge from node **u** to node **v** with weight **w**. If node **u** is not already in the graph, it is added.

Method: min_distance(self, dist, visited)

- Finds the vertex with the minimum distance value from the set of vertices that have not been visited.

```

def dijkstra(self, start):
    visited = {vertex: False for vertex in self.graph}
    dist = {vertex: float('inf') for vertex in self.graph}
    parent = {vertex: None for vertex in self.graph} # Store parent vertices

    dist[start] = 0

    for _ in range(len(self.graph)):
        u = self.min_distance(dist, visited)
        visited[u] = True
        visited_vertices.append(u)

        for v, w in self.graph[u]:
            if not visited[v] and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                parent[v] = u

        # Uncomment this to remove all the multiple plots
        # fig = plt.subplot()
        # setup initial plots()
        # plt.scatter([all_x], [all_y], color='lightgray', marker='o')
        # plt.plot([pointer_dict[u]['x'], pointer_dict[v]['x']], [pointer_dict[u]['y'], pointer_dict[v]['y']])
        # for index, vnum in enumerate(visited_vertices):
        #     plt.scatter(pointer_dict[visited_vertices[index]]['x'], pointer_dict[visited_vertices[index]]['y'], color='gray', marker='x')
        # plt.xlabel('X-axis')
        # plt.ylabel('Y-axis')
        # plt.xlim(-2, 22)
        # plt.ylim(-2, 22)
        # clear_output(wait=True)
        # display(fig)
        # plt.legend()
        # plt.show()

    return dist, parent

```

Method: dijkstra(self, start)

- Implements Dijkstra's algorithm to find the shortest paths from a specified starting vertex to all other vertices in the graph.
- Returns a tuple containing two dictionaries:
 - **dist**: Maps each vertex to its shortest distance from the starting vertex.
 - **parent**: Maps each vertex to its parent vertex in the shortest path tree.

Note: Some part of the code has been commented. This is related to the visualization and animation required for showing the plots. Extra comments are used for check-pointing and debugging the program.

```

def shortest_path_with_distances(self, start, end, parent):
    path = []
    current = end
    distances = {}

    while current is not None:
        path.append(current)
        current = parent[current]

    path.reverse()

    for i in range(len(path) - 1):
        u = path[i]
        v = path[i + 1]
        for neighbor, weight in self.graph[u]:
            if neighbor == v:
                distances[v] = weight
                break
    return path, distances

```

Method: `shortest_path_with_distances(self, start, end, parent)`

- Computes the shortest path from a specified starting vertex to an end vertex, along with the distances between nodes on the path.
- Returns a tuple containing two dictionaries:
 - **path**: Contains the vertices in the shortest path from **start** to **end**.
 - **distances**: Maps each vertex on the path to the distance from the previous vertex.

```

if __name__ == "__main__":
    visited_vertices = []
    graph = Graph()
    for each_path in connection_array:
        graph.add_edge(each_path[0], each_path[1], each_path[2])

    start_vertex = int(starting_vertex)
    end_vertex = int(target_vertex)

    distances, parents = graph.dijkstra(start_vertex)
    path, distances_between = graph.shortest_path_with_distances(start_vertex, end_vertex, parents)

    #print(f'Shortest distance from {start_vertex} to {end_vertex} is {distances[end_vertex]}')
    #print(*path)
    #print('Distances between nodes on the path:')
    intermediate_distances = 0
    dist_array = []
    for node, distance in distances_between.items():
        #print(f'Distance from {path[path.index(node) - 1]} to {node} is {distance}')
        intermediate_distances = distance + intermediate_distances
        #intermediate_distances = "{0:.4f}".format(intermediate_distances)
        dist_array.append(intermediate_distances)
        #print(distance, end=' ')

    #print(*path)
    #print(*dist_array)

```

This is the main function of the program:

- In the beginning we add all the edges to the graph using the add edge function defined earlier.
- We set the starting point and target point for our Dijkstra's algorithm to function. We get the parent which tells us the shortest and vertex before the current vertex, also known as parent.
- Path stores the all vertices visited from starting point to ending point
- Distance Between gives distance between 2 points in the path.
- Intermediate distance, goes to our output file is the distance taken to reach each vertex in the path.