2
X
C
3

FINAL PROJECT

LAB 2

(Harsh Upadhyay, Oliver Li, Sota Nakamura)

TABLE OF CONTENT

PART	PG
PART 1	4
PART 2	9
PART 3	10
PART 4	12
PART 5	18
APPENDIX	20

TABLE OF FIGURES

FIGURE	PG
FIGURE 1	5
FIGURE 2	6
FIGURE 3	7
FIGURE 4	13
FIGURE 5	15
FIGURE 6	16
FIGURE 7	17

PART 1

Graph Class:
The Graph class represents a graph data structure. It includes the following attributes and methods.
The graph is implemented using an adjacency list representation. Each vertex is represented by a dictionary key, and the value associated with each key is a list of edges connected to that vertex.
Each edge is represented by an instance of the **WeightEdge** class, which stores the start node, end node, and weight of the edge.

UNDERSTANDING Dijkstra's Algorithm :

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph.

Dijkstra's algorithm takes a source vertex s and an optional parameter k which limits the number of relaxations allowed per node. It returns two dictionaries:
1. dist : Contains the shortest distances from the source to all other vertices.
2. path : Contains the shortest paths from the source to all other vertices.

The algorithm uses a priority queue (implemented as a **min-heap**, taken from the lecture notes) to efficiently select the next vertex with the smallest distance.
It iteratively updates distances and paths of its neighboring vertex if a lesser cost path through the current vertex is possible, this iteration continues until all vertices have been visited or the relaxation limit `k` is reached, efficiently updating the (node, distance) pairs in the priority queue.

UNDERSTANDING Bellman-Ford Algorithm :

Bellman-Ford algorithm similar to Dijkstra's is an algorithm for finding the shortest paths between nodes in a weighted graph.

The bellmanFord takes a source vertex s and an optional parameter k which limits the number of relaxations allowed per node. It returns two dictionaries:
1. dist : Contains the shortest distances from the source to all other vertices.
2. path : Contains the shortest paths from the source to all other vertices.

The algorithm iterates through each node, along with their neighbours, comparing the cost of the adjacent node with the cost obtained by traversing through the current node, and finally relaxing the node in each iteration.

The function checks for negative graphs by iterating once again through the graph vertices and if there still exists any smaller cost between two nodes then this indicates the presence of negative cycles (as we should already have the best cost).

(NOTE: k defaults to the number of vertices and is stored for each node when no input is provided for it, and hence does not affect our implementations and iterations are carried out until all nodes have been visited)

EXPERIMENTS :

The code conducts experiments based on three different factors to compare the performance of Dijkstra's algorithm and Bellman-Ford algorithm:

1. Graph Size Variation :

AIM : It measures the average runtimes of both algorithms for graphs of increasing size (from 30 to 300 vertices).

EXPERIMENT OUTPUT :

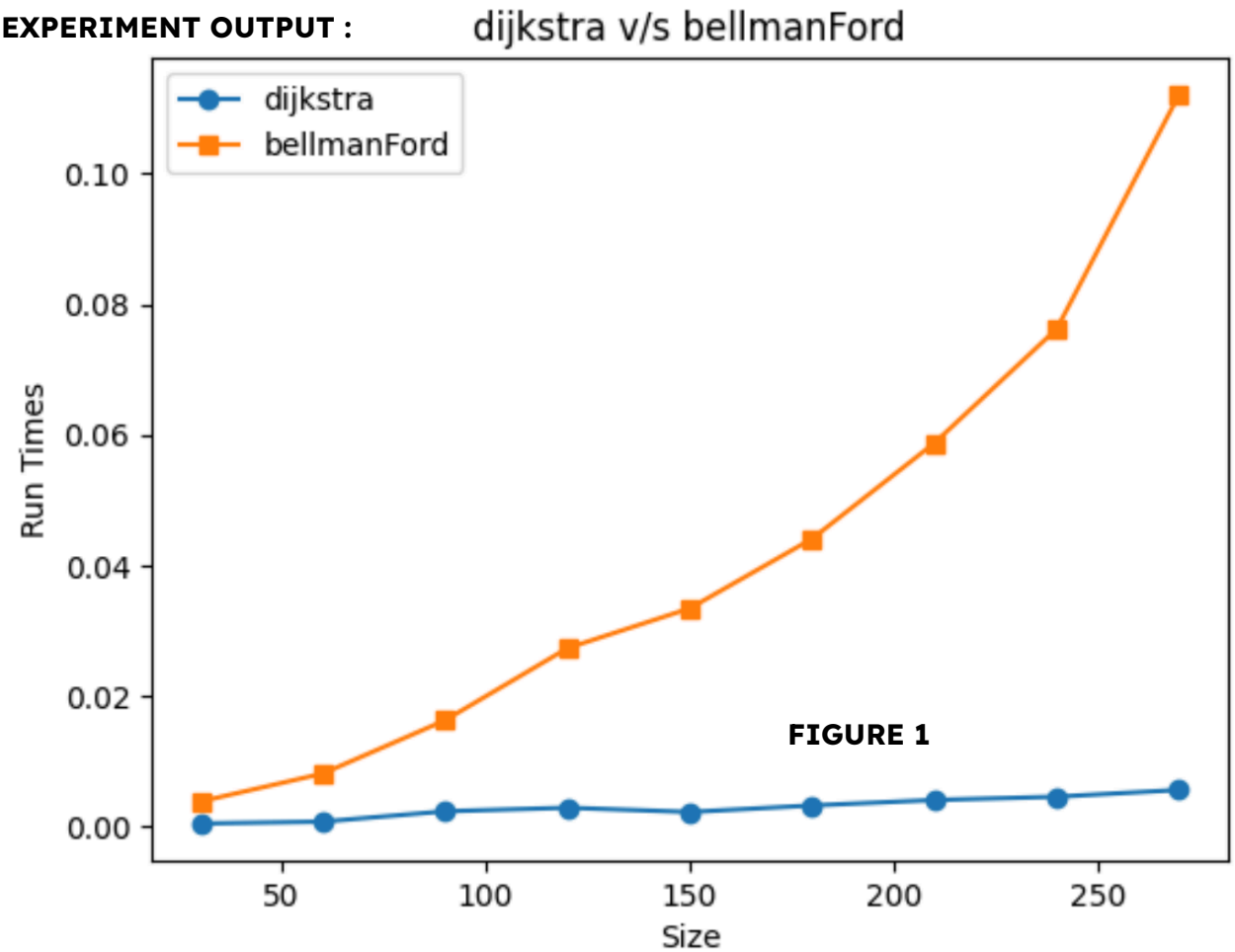


FIGURE 1

OBSERVATIONS :
We can see that there is a much more drastic increase in the run times of bellman ford compared to that of Dijkstra's as the size of the graph increases. This emphasizes that Bellman-Ford has a greater dependence on the size of the graph.

This behaviour can be attributed to the fundamental differences in the algorithms' approaches. Dijkstra's algorithm, being a **greedy** algorithm, maintains a priority queue and selectively estimates the smallest cost function to each node.

On the other hand, the Bellman-Ford algorithm iterates over all edges multiple times (has a triple nested loop), minimizing the cost to each node.

Hence, we can conclude that the results produced by our experiment are exactly what would be expected of the two implementations.

2. K Value Variation :

AIM : It measures the average runtimes of both algorithms for varying relaxation limits (k values).

EXPERIMENT OUTPUT :

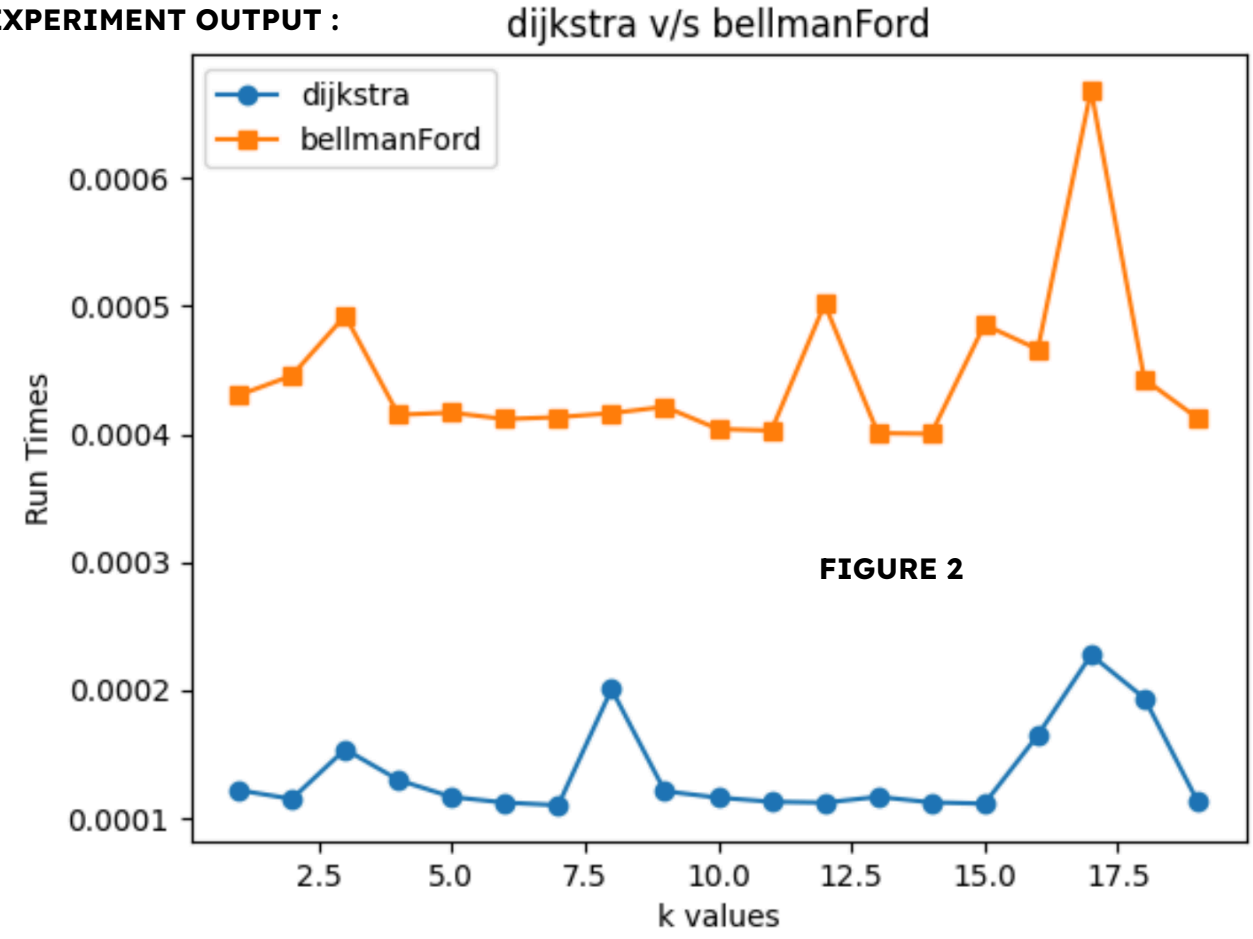


FIGURE 2

OBSERVATION :
It can be observed that in general bellman ford takes more time to compute than that of Dijkstra's.
Observing the behavior of the two algorithms for different values of K, it's apparent that there are no significant differences in runtime between the two algorithms. Both algorithms exhibit very similar behavior, with a slight increase in runtime towards the end for higher values of K.

These observations indicate that while K may influence the runtime of Dijkstra's algorithm to some extent, its impact is not as significant as the fundamental differences between the algorithms themselves.

3. Graph Density Variation :

AIM : It measures the average runtimes of both algorithms for graphs of increasing density (from 50 vertices with increasing numbers of edges).

EXPERIMENT OUTPUT

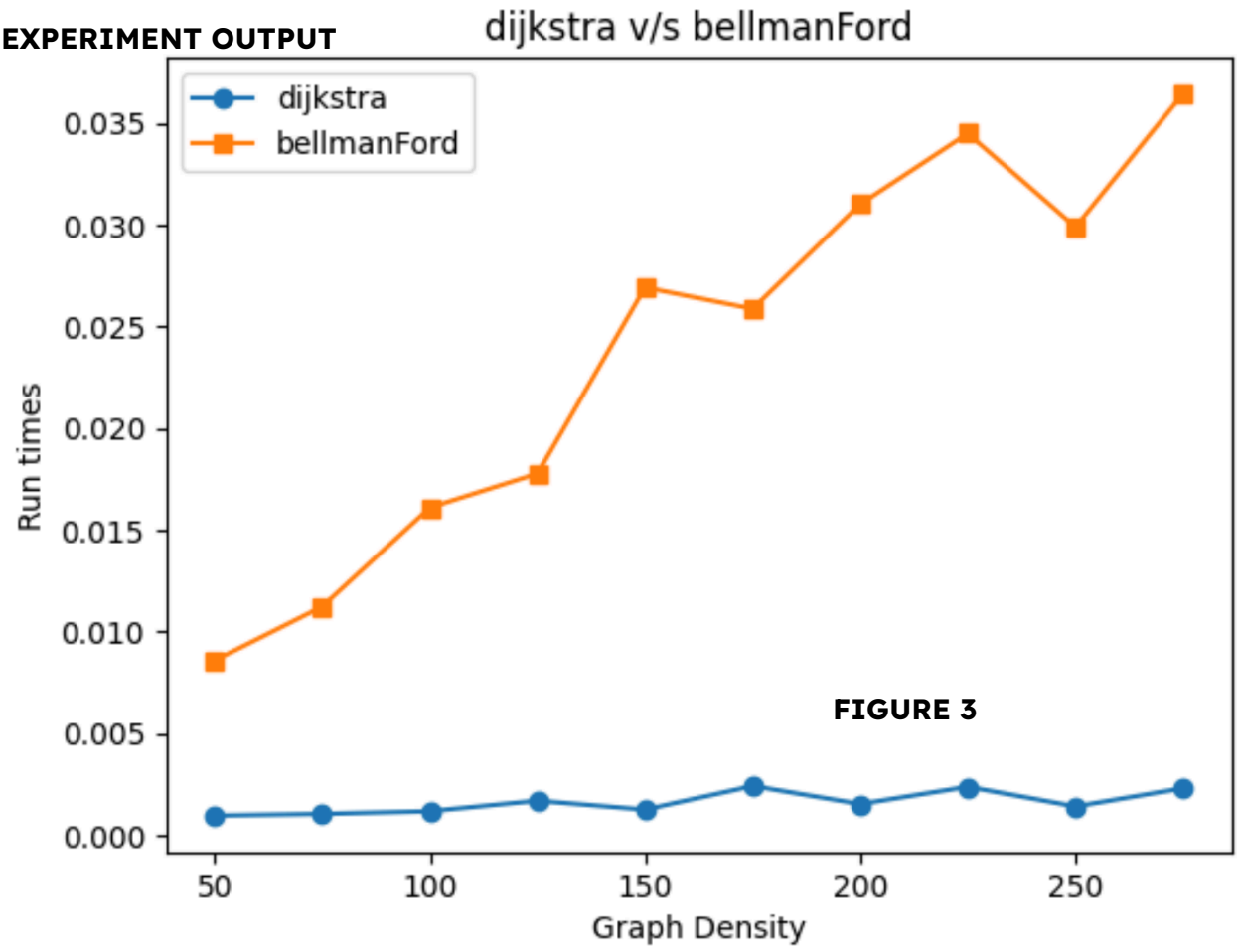


FIGURE 3

OBSERVATIONS :

It can be observed that an increase in the graph's density leads to a greater increase in the run times for Bellman Ford algorithm, while there is also a tiny increase in the run times of Dijkstra's, but the effect is much more pronounced for Bellman-Ford.

Dijkstra's algorithm, being a greedy algorithm, tends to perform relatively consistently for more sparse graphs, while as observed earlier Bellman-Ford algorithm's runtime is directly influenced by the number of nodes and edges in the graph, hence as the density increases Bellman-Ford's runtimes increases too.

FINAL OBSERVATIONS

Upon completing the experiments, it's evident that Dijkstra's algorithm consistently exhibits better execution timings compared to Bellman-Ford. However, it's essential to note that superior execution timings do not necessarily imply better performance overall, as there are trade-offs to consider.

Dijkstra's algorithm has the following performance issues which can be resolved by Bellman-Ford :
1. Sparse Graph : Dijkstra's algorithm's efficiency diminishes as the graph becomes denser. In denser graphs, Dijkstra's algorithm may struggle to produce accurate results efficiently.
2. Negative weights : Dijkstra's algorithm fails to produce correct outputs in graphs containing negative weights. This limitation renders Dijkstra's algorithm ineffective in scenarios where negative weights are present.

Thus, there are scenarios where Bellman-Ford's algorithm is preferred over Dijkstra's, despite its poorer runtime performance.

PART 2

Code Brief :
The algorithm begins by checking whether the graph contains negative weights. If negative weights are present, the algorithm utilizes the Bellman-Ford algorithm. Conversely, if there are no negative weights, Dijkstra's algorithm is employed. This decision is based on prior observations (done in part 1) indicating that Bellman-Ford performs better for graphs with negative weights.

Subsequently, with either Dijkstra's or Bellman-Ford, the algorithm iterates through all vertices to compute the *shortest path* from any vertex v to any other vertex u, while also storing the *previous to last* node from u to v.

Finally, the computed cost and prev (second to last node in the path) for each pair of vertices are returned as a dictionary containing the shortest distances and a dictionary containing the predecessor nodes, respectively.

Time Complexity ANALYSIS:

In the case of a dense graph, where the number of edges approaches the maximum possible (V * (V - 1) / 2), Dijkstra's algorithm has a time complexity of O(V^2), and Bellman-Ford has a time complexity of O(V^3).
(Where V represents the number of nodes in the graph)

To compute the shortest paths between every pair of vertices in the graph, the algorithm runs either Dijkstra's or Bellman-Ford on every vertex as the input source. This process involves iterating through each vertex and running the chosen algorithm to compute the shortest paths to all other vertices, we then finally iterate through the output distances and store them into our Dictionary.

As a result, the overall time complexity of the algorithm becomes O(V^4) if Dijkstra's algorithm is used or O(V^5) if Bellman-Ford is used. This complexity arises from the need to perform the chosen algorithm on each of the V vertices in the graph, and an additional V vertices for storing the output distances.

PART 3

ALGORITHM ANALYSIS :

Dijkstra's algorithm and A* algorithm are both used to find the shortest path in a graph. However, they exhibit differences in performance depending on the characteristics of the graph and the problem at hand.

Dijkstra's algorithm, as implemented in part1.py, systematically explores all possible paths from the source node to every other node in the graph. It maintains a priority queue of nodes to visit, continuously selecting the node with the shortest known distance from the source. While Dijkstra's algorithm guarantees finding the shortest path in most cases it can be computationally expensive, additionally in dense graphs or graphs with negative edge weights Dijkstra's algorithm is at the risk of producing the wrong output.

In contrast, the A* algorithm combines the advantages of Dijkstra's algorithm with heuristics to guide the search more efficiently towards the goal node. It evaluates nodes based on a combination of the actual cost from the source and a heuristic estimate of the cost to reach the destination. By prioritizing nodes that are likely to lead to the goal, A* can often find the shortest path more quickly than Dijkstra's algorithm, especially in graphs where the heuristic provides accurate guidance towards the goal. *(Great dependence a good heuristic function)*

Issues with Dijkstra's algorithm addressed by A* :
One issue with Dijkstra's algorithm is that it considers all possible paths, including those that are far from the destination, resulting in wasted time. In contrast, A* addresses this issue by prioritizing the exploration of paths more likely to lead to the destination. It achieves this by incorporating a heuristic function, which guides the search towards the goal node. By leveraging the heuristic, A* focuses its efforts on paths that have a higher probability of reaching the destination efficiently.

Empirical testing of Dijkstra's vs A* :
Dijkstra's algorithm computes paths from the source node to all other nodes in the graph, while A* focuses on finding a path from the source to a specific destination node. So, in order to test their performance, we conduct runtime comparisons by applying Dijkstra's algorithm to graph G and A* algorithm to compute paths from the

source node to all other nodes in G, treating each node in G as a potential destination. In order to properly test the two algorithms It's important to vary the size and structure of G across different tests, considering factors such as **density, sparsity, and overall size.**

Comparison with an arbitrary heuristic function :
When the heuristic function used in A* is arbitrary, the accuracy of A*'s results may be compromised. In such cases, Dijkstra's algorithm is likely to outperform A*.

Applications where A* is preferred over Dijkstra's* :
A* is generally preferred over Dijkstra's algorithm in applications where there exists additional domain knowledge in the form of a heuristic function. For instance, in a car navigation system, A* can determine optimal routes by incorporating factors like real-time traffic conditions, which are encoded in the heuristic function.

PART 4

GRAPH REPRESENTATION :

The question describes that each station is a node in a graph, and the edge between stations should exist if two stations are connected (which can be found from the file *london_connections.csv*). Additionally, we use the latitude and longitude for each station to find the distance travelled between the two stations, which will be used to represent the weights of different edges (which can be found from *london_stations.csv)*

CODE ANALYSIS :

We use the euclidean function to compute the distance between two stations (the physical driving distance).

The heuristic function calculates the distances from a given source station to all other stations based on Euclidean distance. Heuristic distances provide informed estimates of the cost to reach each station, guiding the A* algorithm towards the destination more efficiently.

The first algorithm, shortestPathDIJKSTRA, employs Dijkstra's algorithm to calculate the shortest paths from each station to all other stations. The second algorithm, shortestPathASTAR, utilizes the A* algorithm to achieve the same objective

EXPERIMENTS

1.

We do the following experiment to the answer this question :
When does A* outperform Dijkstra? When are they comparable? Explain your observation why you might be seeing these results.

AIM : To plot the runtimes for shortestPath function defined for both DIJKSTRA and A* and plot the iteration, this will be done for 10 Iterations.

EXPERIMENT OUTPUT :

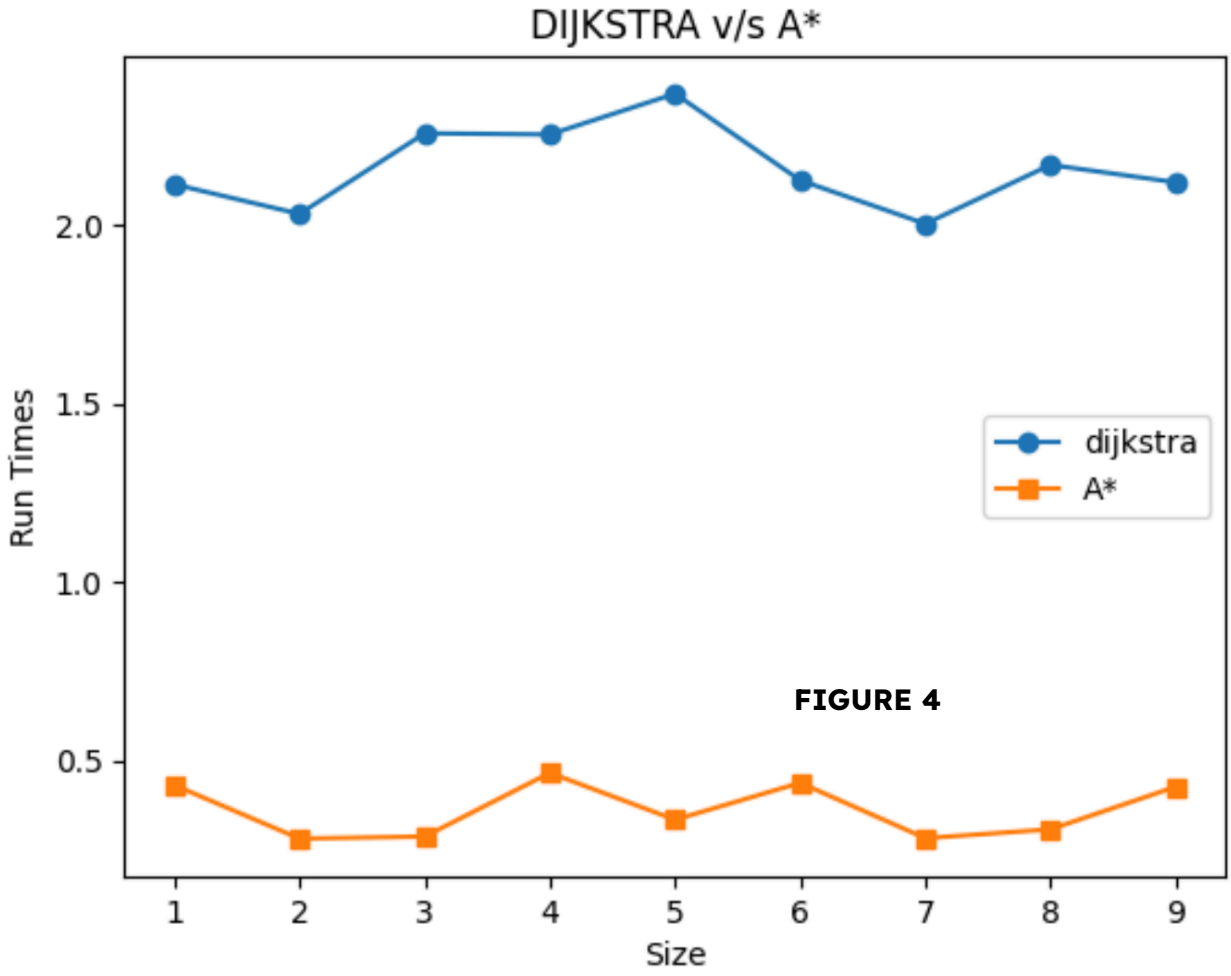


FIGURE 4

OBSERVATIONS :

(The output graph shows a fairly consistent graph in terms of runtimes for both the algorithms, this is because we are implementing the respective functions without making any changes to the graph or function.)

We observe that A* algorithm achieves significantly lower runtimes compared to Dijkstra's algorithm.
This behaviour can be attributed to the fundamental differences in the algorithms' approaches.
A* utilizes this heuristic information to guide its search towards the goal node more efficiently, prioritizing exploration in promising directions. As a result, A* often explores fewer nodes and completes the search more quickly, especially in scenarios where the heuristic provides accurate guidance towards the goal.
On the other hand, Dijkstra's algorithm explores all possible paths from the source node to all other nodes, which causes it to be more expensive than A*.

Cases when A* outperforms Dijkstra's :
when the path to the destination node is long, Dijkstra's tends to take more time as it needs to check for every possible neighbours, whereas A* on the other takes a lot shorter time as the heuristic function provides accurate guidance towards the destination node.

Cases when A* is comparable to Dijkstra's :
When there exist multiple shortest path to the destination node the runtimes of A* would be comparable to Dijkstra's.

2.
We do the following experiment to the answer this question :
What do you observe about stations which are 1) on the same lines, 2) on the adjacent lines, and 3) on the line which require several transfers?

In this particular experiment instead of comparing all the shortestPaths for all three cases we find station pairs and test their runtimes one by one, to gain broader insights.

AIM : To plot the runtimes for the given three cases.

QUESTION UNDERSTANDING :

1) Two stations are said to be on the same line if it is possible to reach station2 from station1 by staying on the same line.
2) Two stations are said to be on adjacent lines if we can reach station2 from station1 by changing only one line.
3) Two stations are said to have several transfer lines if we can reach station2 from station1 by changing only more than one line.

EXPERIMENT OUTPUTS :

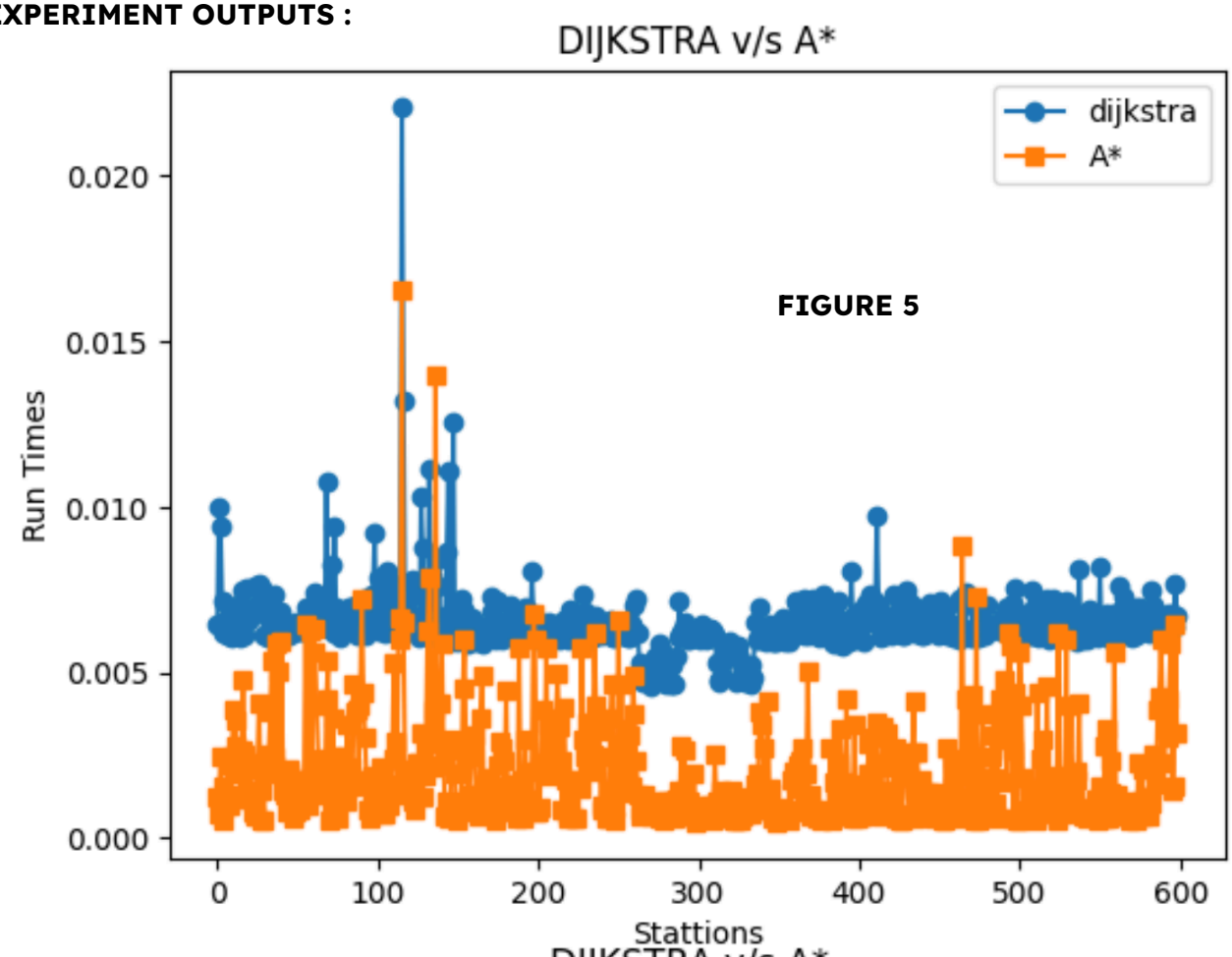


FIGURE 5

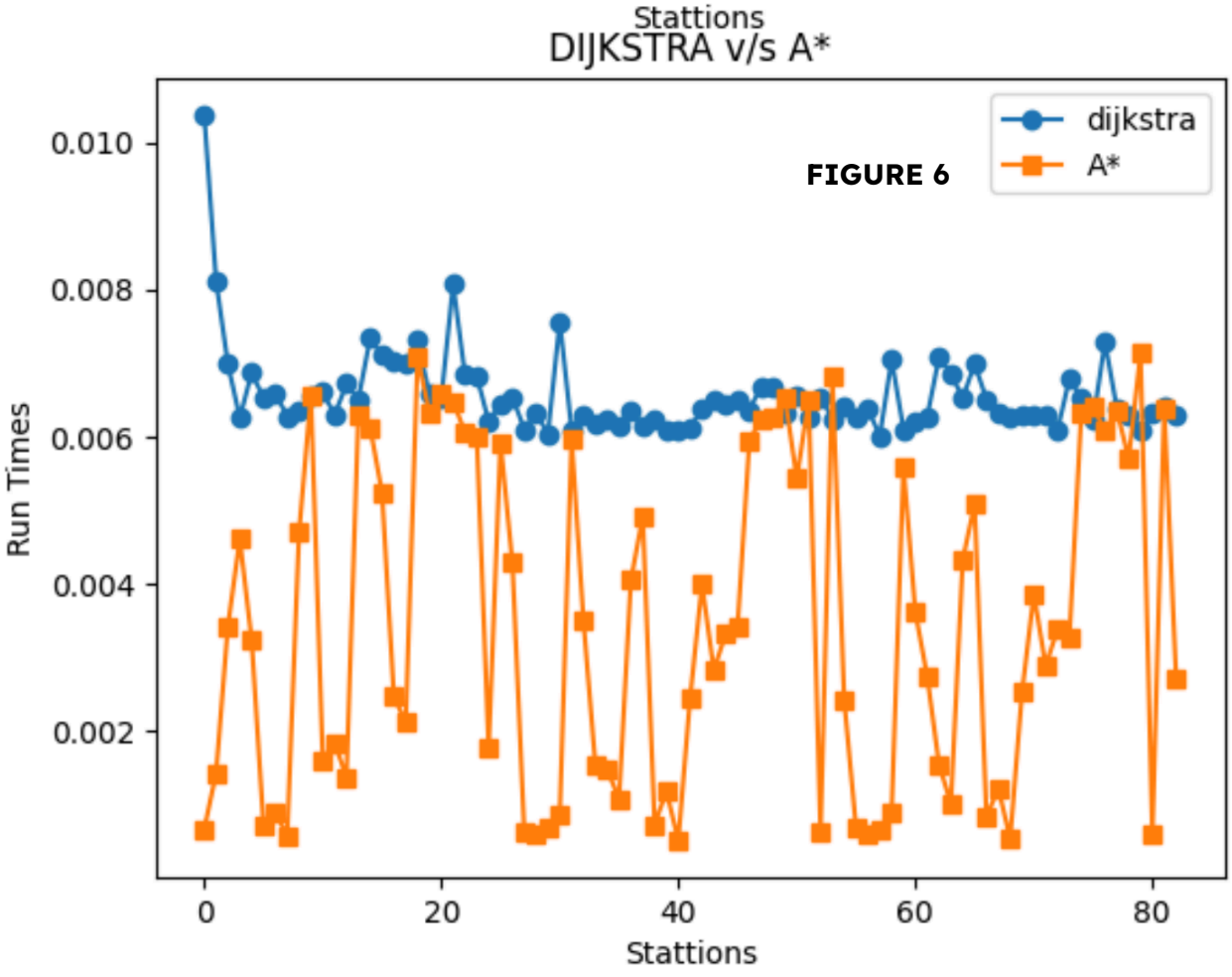


FIGURE 6

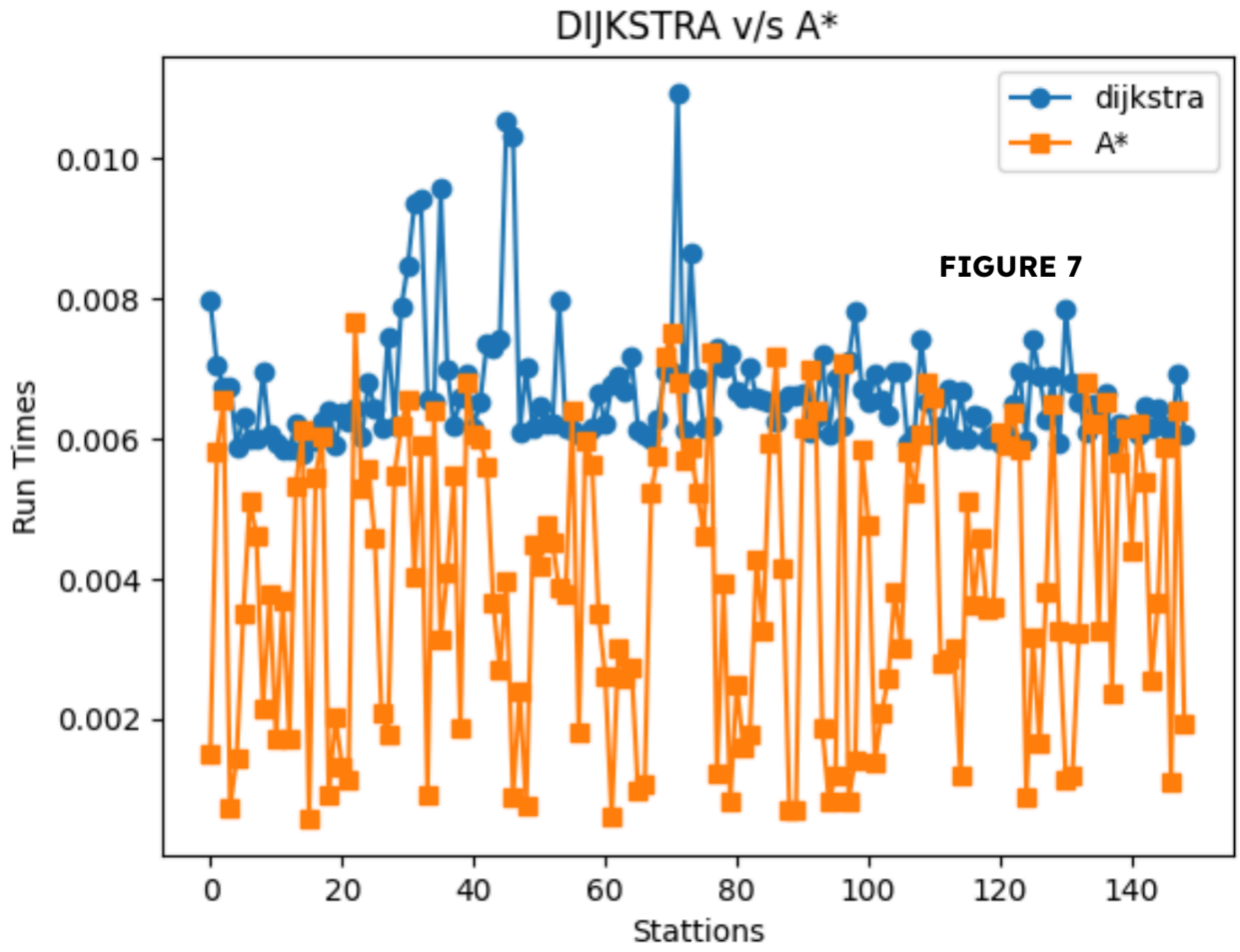


FIGURE 7

OBSERVATIONS :

In general it can be observed that A* always works better than Dijkstra's.

But upon closer examination of the graphs it can be noticed that as the number of lines increases between the source node and destination node increases, the run times produced by A* comes close to that of Dijkstra's.
The main reason why we see such pattern is due to how A* approaches to find the shortest path by looking at the best possible option instead of checking all the nodes. In our specific examples due to the nature of inputs the weights for all the edges is very close (difference in a few decimals), now as the number of transfer line increases the size of the path from source to destination shall also increase, and since the weight for all these nodes are very similar our A* has to take a look at all of them since most of the nodes can be used to form the shortest path (cost).

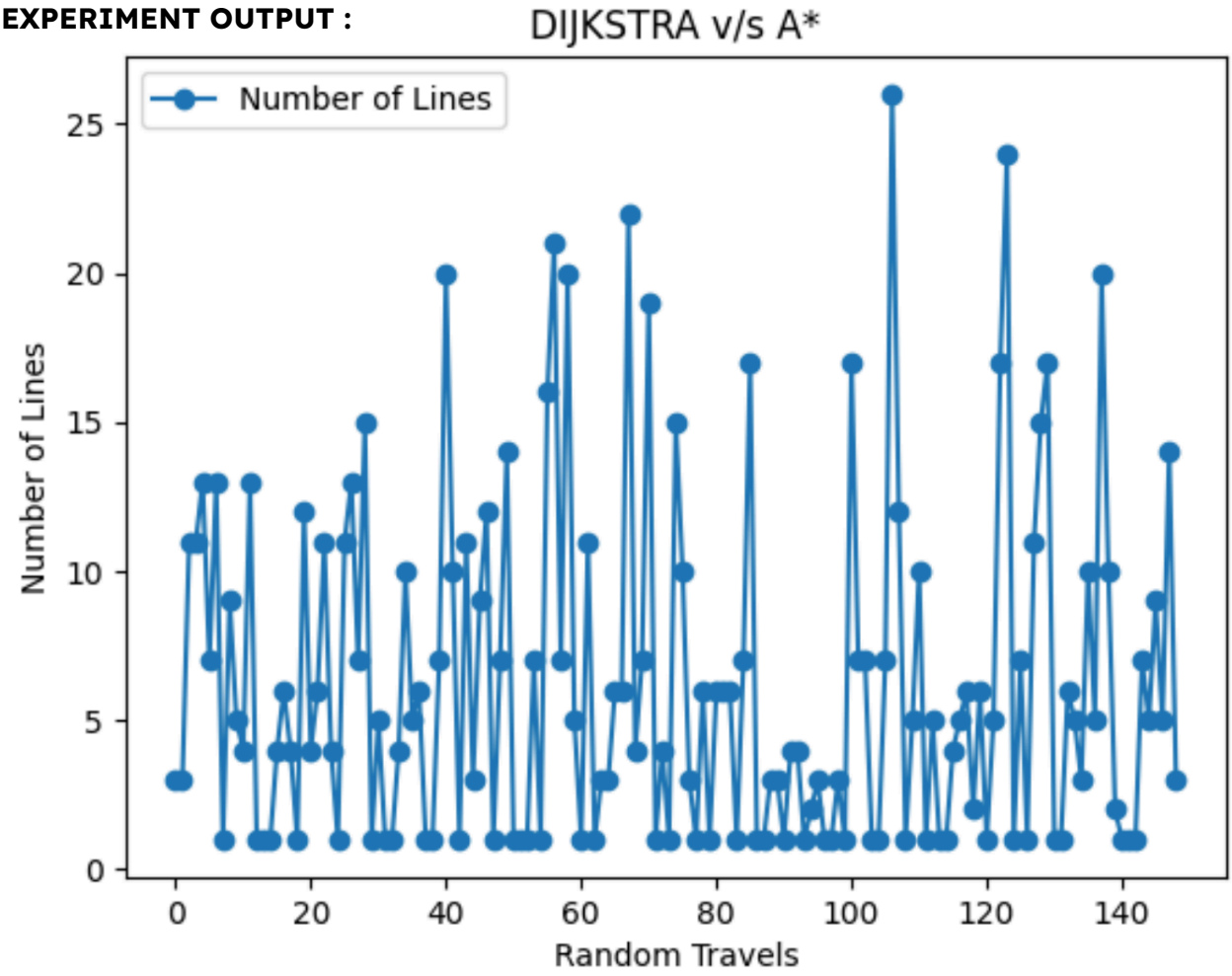
3.

We do the following experiment to the answer this question :

Using the “line” information provided in the dataset, compute how many lines the shortest path uses in your results/discussion?

AIM : To plot the number of lines for some random multi transfer lines generated above against the number of iterations.

EXPERIMENT OUTPUT :



OBSERVATIONS :

We observe that the number of lines used in the shortest path revealed from random high transfer lines produces a highly variable distribution, with fluctuations ranging from 0 to 25 across different iterations.

PART 5

1. **Discuss what design principles and patterns are being used in the diagram**

The diagram employs the following design principles and patterns:

Structural Patterns:

Bridge : The diagram may utilize the bridge pattern to separate the abstraction from its implementation, allowing for easier modification and extension of both components.
Adapter : This pattern might be applied to adapt different interfaces or classes, enabling them to work together seamlessly within the system.
Composite : The composite pattern could be used to treat individual objects and compositions of objects uniformly, simplifying the handling of complex hierarchical structures.

Behavioral Patterns:

Factory Method Pattern : This pattern may be employed to define an interface for creating objects, but let subclasses alter the type of objects that will be created.
Inheritance: Inheritance allows classes to inherit attributes and methods from other classes, facilitating code reuse and promoting a hierarchical structure within the system.
Abstraction: Abstraction helps to simplify complex systems by focusing on essential details while hiding unnecessary complexities, thus enhancing readability and maintainability.

2. **The UML is limited in the sense that graph nodes are represented by the integers. How would you alter the UML diagram to accommodate various needs such as nodes being represented Strings or carrying more information than their names.? Explain how you would change the design in Figure 2 to be robust to these potential changes.**

To address the limitation of the UML diagram, where graph nodes are represented solely by integers, we can introduce modifications to accommodate various needs, such as representing nodes with strings or carrying additional information beyond their names. One approach would be to incorporate a hashmap at a global scale, mapping

integer IDs to diverse identification data, such as building names, cities, or complex objects. This hashmap facilitates efficient retrieval of node representations based on their integer IDs, ensuring constant-time access complexity (O(1)).

In terms of UML design, we can introduce an additional class inheriting from the Graph class, which encapsulates a dictionary storing pairs of integer IDs and their corresponding representations.

3. **Discuss what other types of graphs we could have implement "Graph". What other implementations exist?**

We can implement a graph using an adjacency list, where integer IDs serve as keys and sets of edges serve as values. Each edge can be represented as an object containing the IDs of the connected nodes and the weight of the edge. This approach offers flexibility and efficiency, particularly in scenarios with sparse graphs.

Alternatively, we could utilize an adjacency matrix, which is represented as an n by n matrix, where n is the number of vertices in the graph. Each cell in the matrix represents the weight of the edge between two vertices. While the adjacency matrix allows for fast edge operations through simple indexing, it consumes more memory compared to the adjacency list, especially for large graphs. However, it is particularly suitable for dense graphs where most vertices are connected.

APPENDIX

STEPS TO NAVIGATE OUR CODE :

PART 1 :
Upon running part1.py you will be presented with three different graphs based on our three experiments conducted.

The code is structured in the following order :

- ☐ Class min-Heap : class implementation of a min-Heap which is later used in the Dijkstra algorithm
- ☐ Class WeightedEdge and Graph class : Used to implement a graph, this graph contains implementations for both Dijkstra and BellmanFord algorithms.
- ☐ Draw_2_graphs : matPlotlib function we use to plot two different lists against iterations. *(All lot of variation of this function is present here)*
- ☐ randEdge : Function to generate random edges which are imputed to create random graphs.
- ☐ Experiment 1
- ☐ Experiment 2
- ☐ Experiment 3

PART 2 :

The code is structured in the following order :

- ☐ Class min-Heap : class implementation of a min-Heap which is later used in the Dijkstra algorithm
- ☐ Class WeightedEdge and Graph class : Used to implement a graph, this graph contains implementations for both Dijkstra and BellmanFord algorithms.
- ☐ checkNegative_weight : function which checks if there exists any negative nodes in the imputed graph .
- ☐ shortestPathDIJKSTRA : Function to find the shortest path between all nodes using Dijkstra's algorithm.
- ☐ shortestPathBF: Function to find the shortest path between all nodes using Bellman Ford's algorithm.
- ☐ AllPair_shortestPath : Finds all shortest paths, if the imputed graph has negative nodes it uses shortestPathBF, otherwise shortestPathDIJKSTRA .

PART 3 :

- ☐ IndexMinPQ : class implementation of a min-PQ which is later used in the A* algorithm
- ☐ A_Star : to find the shortest path between the two inputted nodes using the A* algorithm .

PART 4 :

Upon running part4.py you will be presented with five different graphs based on our five experiments conducted.

The code is structured in the following order :

- ☐ Class min-Heap : class implementation of a min-Heap which is later used in the Dijkstra algorithm
- ☐ Class WeightedEdge and Graph class : Used to implement a graph, this graph contains implementations for both Dijkstra and BellmanFord algorithms.
- ☐ A_Star : to find the shortest path between the two inputted nodes using the A* algorithm .
- ☐ euclidean: Function to find the euclidean distance between the two inputted points.
- ☐ Using File I/O to read from the two csv files
- ☐ shortestPathDIJKSTRA : Function to find the shortest path between all nodes using Dijkstra's algorithm.
- ☐ shortestPathASTAR : Function to find the shortest path between all nodes using A* algorithm.
- ☐ Draw_2_graphs : uses matPlotlib to create a visual graph in which we use to plot two different lists against iterations. *(All lot of variation of this function is present here)*
- ☐ Experiment 1
- ☐ Experiment 2
- ☐ Experiment 3
- ☐ Experiment 4
- ☐ Experiment 5

PART 5 :
Contains all code from previous parts in the manner specified by the UML graph provided.

The code is structured in the following order :

- [ ] Class ShortPathFinder
- [ ] Class Graph: inherits ShortPathFinder, and is used to implement our generic graphs.
- [ ] Class WeightedGraph: inherits Graph, and is used to implement weighted Graphs.
- [ ] Class HeuristicGraph: inherits WeightedGraph, and is used to implement Heuristic Graphs.
- [ ] Class SPAlgorithm: inherits ShortPathFinder, the class in which three shortest path functions are implemented.