Final Project Report

COMPSCI 2XC3

Arshnoor Kaur, kaura69

Garv Rastogi, rastogig

Zain Siddiqui, siddiz6

Dr. Swati Mishra

April 04, 2024

TABLE OF CONTENTS

LIST OF FIGURES

Section 1

Part 1.3

Experiment 1: Comparing the Dijkstra and Bellman Ford Algorithm on the basis of the number of runs

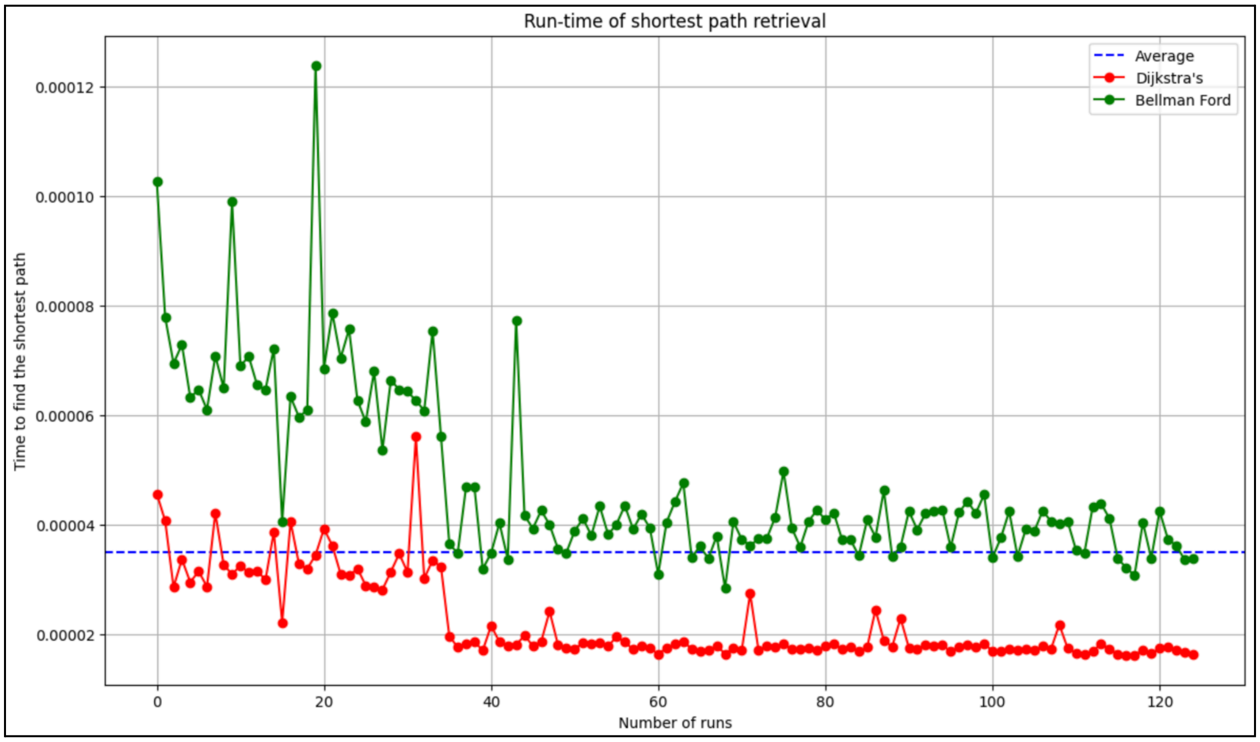

Fig: 1.3.1 Comparing the performance against the number of runs

The design of the experiment
The number of the runs taken: 125 runs
The size of the graph: 10
The number of relaxations allowed (k): 2

In this experiment we plot the time taken to find the shortest path from the source node to each and every node of the graph, using Dijkstra's algorithm and the Bellman Ford Algorithm. On every run we plot a new random graph with the same source node for all the runs i.e node 0 and the same number of relaxations allowed i.e (k=2).

Observations
We observe that

1. With increasing number of runs, the run time of the Bellman Ford and Dijkstra's algorithm starts decreasing.

2. The run-time of the Bellman-Ford algorithm is more than the time taken by Dijkstra's algorithm, indicating that Dijkstra's algorithm is faster than Bellman-Ford algorithm.

The faster performance of the Dijkstra's algorithm could be attributed to the use of the priority queue data structure that is being used to extract the element of greater priority to be chosen for exploration. The greater time of performance of the Bellman Ford algorithm can also be because the Bellman-Ford algorithm is checking for negative cycles in the graph. Which is an additional for loop (extra traversal) in the execution.

Experiment 2: Comparing the Dijkstra and Bellman Ford Algorithm on the basis of the number of maximum allowed relaxations (k)

Design of the experiment:
The size of the graph: 50
The number of runs per relaxation: 10
The number of relaxations allowed (k): 1 - len(Graph)

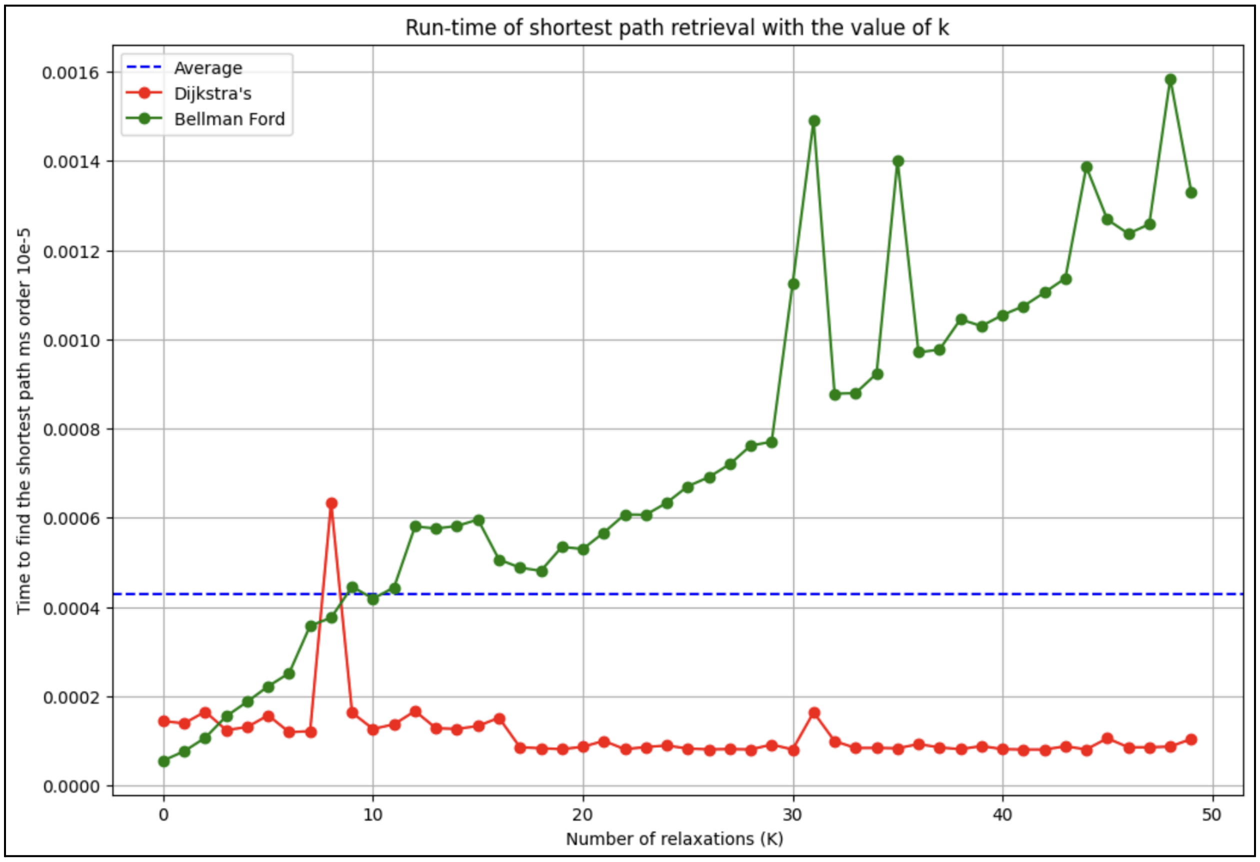

Fig: 1.3.2 Comparing the performance against the number of relaxations (k)

In this experiment we plot the graph between the time taken to retrieve the shortest path from the start node to each and every node of the graph depending on the maximum number of allowed relaxations (K) and for every value of k, we find the time of retrieval and append the average of the times of retrieval and then plot the graphs.

For all the runs and for all the values of k, the graph remains the same. We are increasing the number of allowed relaxations on the same graph.

Observations

After plotting the graph between the number of maximum allowed relations (k) and the time of retrieval of the shortest path using the two algorithms.

We observe that:

1. The time of retrieval of the shortest path (run-time) using the Bellman-Ford algorithm was much greater than the run-time of retrieval using the Dijkstra's algorithm.
2. As the number of allowed relaxations keeps on increasing, the time taken by the Dijkstra's algorithm keeps on increasing. There are some values for which the run-time decreases for Dijkstra's algorithm, but overall the run-time keeps on increasing.
3. For Dijkstra's algorithm, the run-time of retrieval of the shortest path remains almost constant, there is not any significant change in the speed of execution. There are some values where the time of execution decreased and there are some values of k where the time of execution increased relatively. However, overall the algorithm maintains it run-time and the number of relaxations allowed doesn't significantly affect the performance.

Experiment 3: Comparing the Dijkstra and Bellman Ford Algorithm on the basis of the size of the graph (number of nodes)

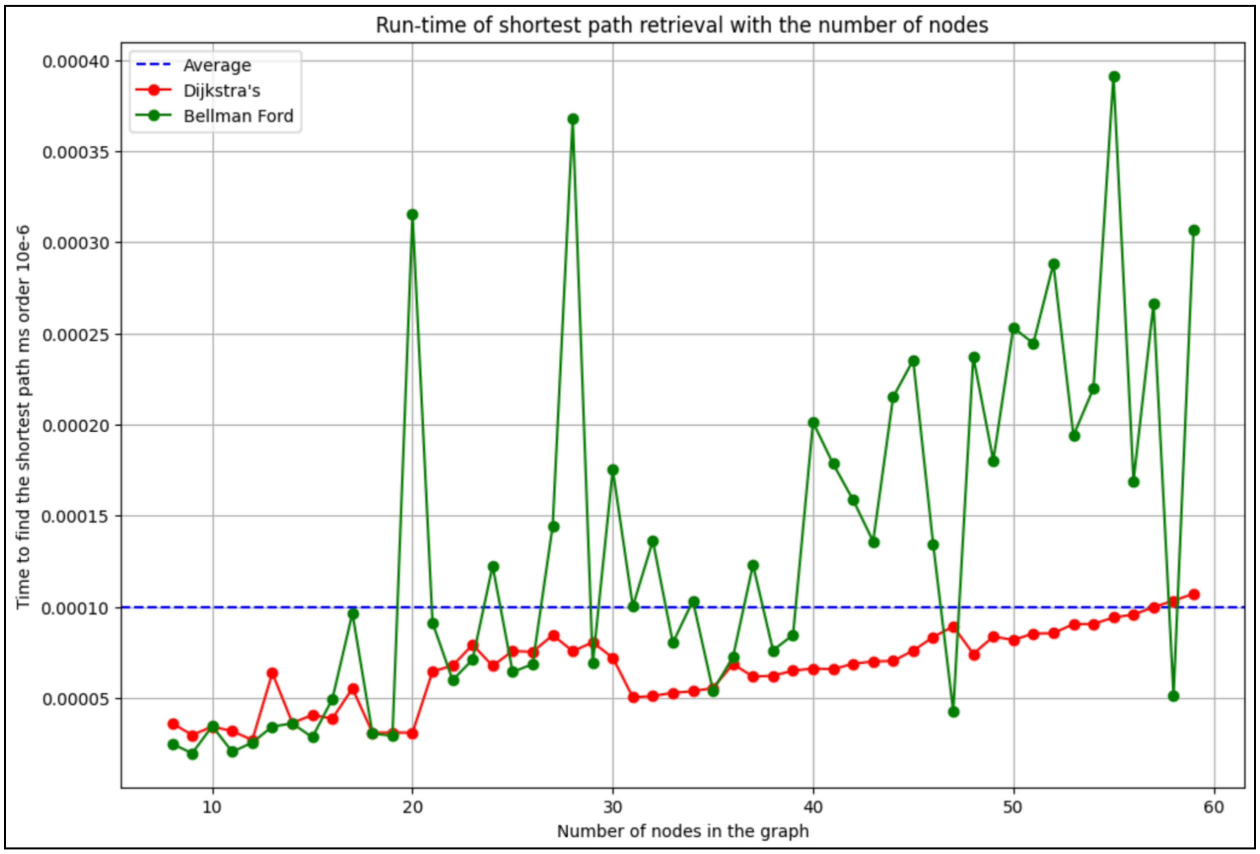

Fig: 1.3.3 Comparing the performance against the size of the graph

Design of the experiment:
The number of the runs taken: 50 runs
The range of the size of the graph: 10 - 60 nodes
The number of relaxations allowed (k): Random integer between 1 and n//4 where n is the number of nodes.

In this experiment we are comparing the performance between the Bellman Ford and Dijkstras algorithm on the basis of the time taken to find the shortest path from the start node to all the nodes in the graph, we are performing this experiment 50 times and for each run we are increasing the number of nodes in the graph by 1.

We observe that as we keep increasing the number of nodes in the graph, the time taken to find the shortest path from the start node to the other nodes keeps on increasing both Dijkstra's and Bellman Ford algorithm. However, we observe that the time taken for the Bellman Ford algorithm increases much more rapidly and steeply as compared to Dijkstra's algorithm, which also shows an increase in the time taken, however we see that the time increase is relatively as compared to Bellman-Ford algorithm.

This can be attributed to the fact that the time complexity of Dijkstra's algorithm is Θ() or Θ() and the time complexity of the Bellman-Ford algorithm is Θ(VE) or $\Theta(V^3)$ where V is the number of nodes/vertices in the graph and E is the number of edges connecting the nodes. Since the time complexity of Dijkstra's algorithm is lesser $\Theta(n^2)$ as compared to the Bellman-ford algorithm $\Theta(n^3)$.

Therefore, we can conclude that the Dijkstra's algorithm is faster than than the Bellman-Ford algorithm in retrieving the shortest path from the start-node to each and every node in the graph when we have bigger graphs. Thus, having a comparatively better performance.

Experiment 4: Comparing the Dijkstra and Bellman Ford Algorithm on the basis of the density of the graph

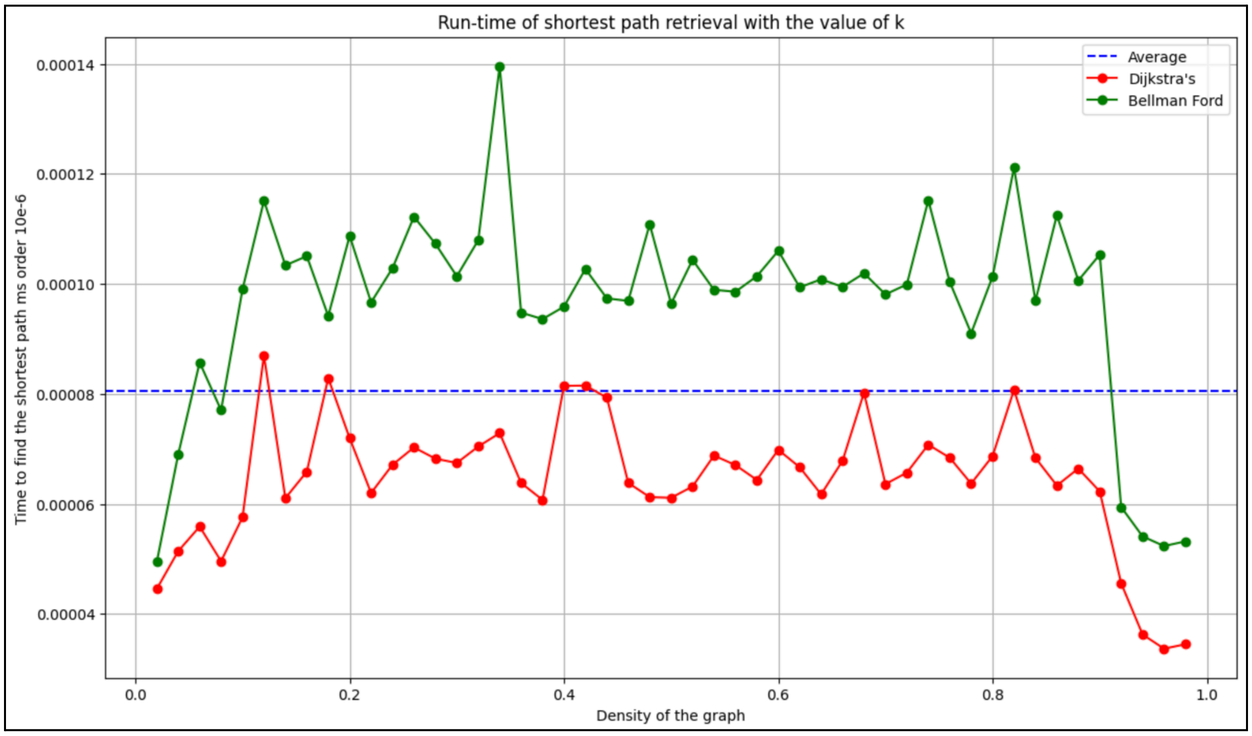

Fig: 1.3.4 Comparing the performance against the density of the graph(no. of edges per node)

Design of the experiment:
The number of the runs taken: 50 runs
The size of the graph: 50
The number of relaxations allowed (k): 3

In this experiment we are comparing the performance of the Dijkstra and the bellman ford algorithm on the basis of the time taken to find the shortest path from the start node to each node of the graph with a limitation of the number of relaxations allowed per edge = k, in the experiment with every run we are increasing the density of the graph.

Observations:
As the density increases from 0 to 1 the number of edges for the same number of nodes keep on increasing (0 meaning that there are no edges connecting the nodes and 1 means that every node is connected to every other node in the graph through an edge).
For the density = 0, the time taken to find the shortest path was approximately the same for both the Dijkstra’s and the Bellman Ford algorithm, as the density increases, the time taken to find the shortest path for the Bellman ford algorithm increases much more rapidly as compared to the Dijkstra’s algorithm.

This can be attributed to the time complexities of both the algorithms, the time complexity of Dijkstra’s algorithm is O(E + Vlog V) and when the graph is dense the time complexity of Dijkstra becomes $\Theta(V^2)$ and the time complexity of the Bellman Ford algorithm is $\Theta(VE)$ and when the graph is dense the time complexity of the Bellman ford algorithm becomes $\Theta(V^3)$.

Since, the time complexity of the Bellman Ford algorithm is higher than the time complexity of Dijkstra’s therefore we see that the performance of the Bellman-Ford algorithm becomes much more slower and inefficient for dense graphs as relatively comparing to Dijkstra’s algorithm.

Time Complexity Analysis

The time complexity of Dijkstra's algorithm and Bellman-Ford algorithm can be analyzed as follows:

1. Dijkstra's Algorithm:
   - In this implementation we have implemented Dijkstra’s algorithm using a priority queue, which is implemented as a set. Dijkstra's algorithm typically has a time complexity of O((V + E) log V), where V is the number of vertices and E is the number of edges.
   - This complexity arises from the fact that each vertex is extracted from the priority queue once, and for each extracted vertex, its adjacent vertices are relaxed, which takes O(log V) time. Relaxing an edge can take up to O(log V) time as well.
   - With a dense graph (where E is close to V^2), Dijkstra's algorithm has a very poor performance as compared to its theoretical time complexity. This can also be seen from the algorithm as we have a for loop inside of a while loop, resulting in a quadratic time complexity.

2. Bellman-Ford Algorithm:
   - The Bellman-Ford algorithm has a time complexity of O(V * E), where V is the number of vertices and E is the number of edges.
   - In the algorithm we iterate over all edges in the graph for |V| - 1 iterations, where we relaxing each edge “k” no. of times.
   - The algorithm also checks for negative weight cycles, which takes another O(E) time.

- In the worst case, Bellman-Ford has to iterate over all vertices and edges for |V| - 1 iterations to find the shortest paths, resulting in O(V * E) time complexity.

Conclusion:
- Dijkstra's algorithm typically has a time complexity of O((V + E) log V) and in the case of dense graphs the time complexity becomes $O(V^2)$.
- Bellman-Ford algorithm typically has a time complexity of O(V * E) and incase of dense graphs the time complexity becomes $O(V^3)$.

Accuracy Analysis

Original Algorithm VS Algorithm with Controlled Relaxations

Relation between the shortest path and the number of relaxations: We observe that in the original **dijkstra's algorithm and Bellman Ford algorithm** there was no restriction on the number of allowed relaxations for each node, for each node the number of allowed relaxations where | V | - 1 where V is the total number of nodes in the graph.

For small values of k (k<15) in a graph with (>= 100 nodes), we observed that the shortest distance between the source node and a large number of nodes in the graph was infinity. This is because not all the nodes are directly connected to the source node or there are nodes that have (> k) intermediate nodes between start and the destination node.
This shows that for relatively small values of k (number of allowed relaxations), the dijkstra's algorithm is not accurate as the shortest path and the weights returned are not accurate.

As we keep increasing the values of k (number of allowed relaxations), we observe that the distance between the source node and all the other nodes of the graph keeps on decreasing and the path keeps on becoming shorter. This indicates that the accuracy of the graph starts increasing as we increase the value of k.

After a certain point, we get a value of k say k_1 at which we get the shortest path between the source node and every node of the graph and the shortest distance and this is the same as the output of the original dijkstra's algorithm, this indicates that when the number of relaxations (k) = k_1 then at this point the controlled dijkstra's algorithm becomes the most efficient and starts behaving like the original dijkstra's algorithm.

If we increase the value of k further we see that there is no change in the output received as the shortest path has already been achieved and no other shortest path can be found. This is where the output stays the same and the algorithm is the most efficient.

Relation between the total weight of the graph and the number of relaxations:

In the experiment we plot a graph between the total weight of the graph and the number of allowed relaxations for both the **Dijkstra** and **Bellman Ford** algorithms, we perform 100 runs on a graph with 100 nodes and in each node it increases the number of allowed relaxations by 1 and performs Dijkstra's algorithm to find the shortest path and another similar experiment where we perform Bellman Ford's algorithm instead of Dijkstra's algorithm.

We observe that when the value of k was small the total weight was very large indicating that the path found is not actually the shortest path and there are some nodes which were unreachable from the start state, therefore in the distance dictionary the distance between the start nodes and these nodes was infinity (***NOTE: Since we couldn't plot infinity values on the graph we have scaled down the infinity values so that they could be plotted on the graph***).
As we keep increasing the value of k, the algorithm starts becoming more and more efficient and the path starts becoming close to the shortest path and the distances start decreasing, hence the weight of the total path starts decreasing.
However, we see that after a certain value of k, where the algorithm becomes the most efficient and the path found is the shortest path with all the nodes being at the minimum distance from the start node. The total weight of the path doesn't can't decrease any further.
This can be seen in graph plotted below that after a value of k (approx 13 or 15). There is no further change in the total weight of the all the paths found.

Dijkstra's algorithm

Similar to the above graph, we get a similar graph for the Bellman Ford algorithm, when we plot a graph between the total weight to reach from the start node to every other node and the number of relaxations.

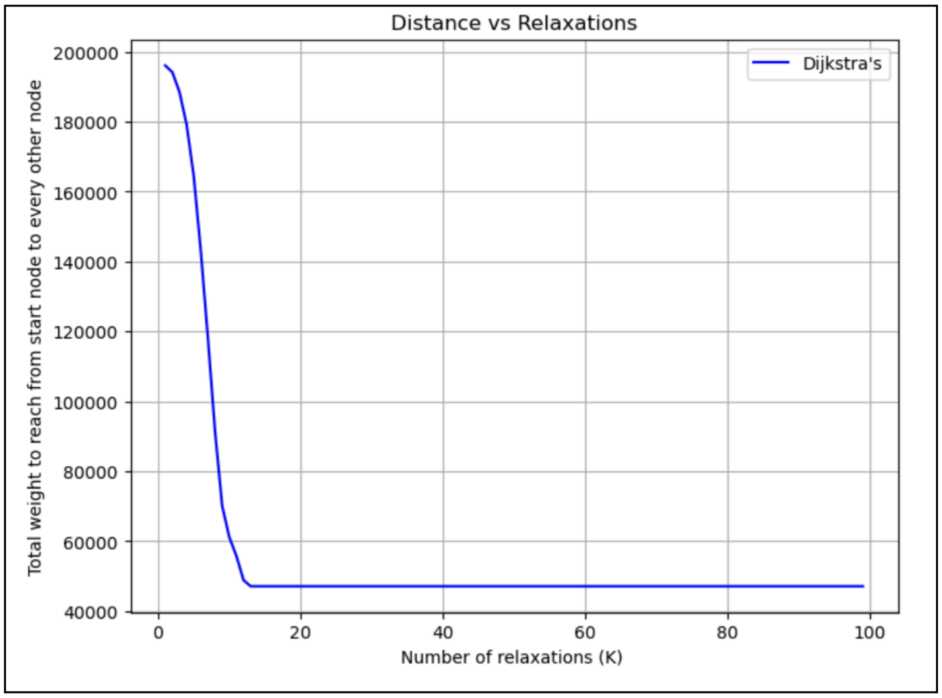

Fig: 1.4 Accuracy analysis: Total weight of shortest path VS Relaxations (Dijkstra)

Bellman Ford algorithm

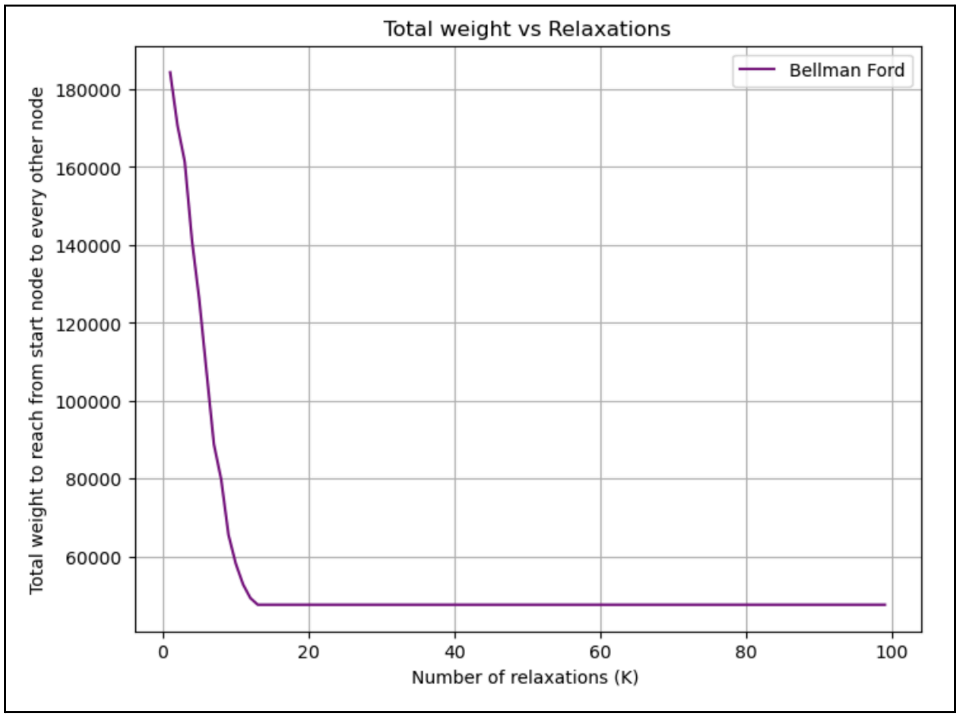

Fig: 1.5 Accuracy analysis: Total weight of shortest path VS Relaxations (Bellman Ford)

Space Complexity Analysis

We present the following analysis of the space complexity of both the Dijkstra and the Bellman Ford algorithm respectively:

1. Dijkstra's Algorithm:
   - In our algorithm, we have created a priority queue (min heap) that will keep a track of the nodes to be analyzed, the priority queue retrieves the value depending on the priority (node with the smallest weight first)
   - Every node has an array associated with it that keeps a track of the intermediate nodes that come in the path from the start node to that current node.
   - The space complexity of Dijkstra's algorithm is expressed as O(V + E), where V is the number of vertices and E is the number of edges in the graph.
   - The priority queue i.e pq typically requires O(V) space, and the distance dictionary requires O(V) space as well where V is the number of nodes in the graph.

2. Bellman-Ford Algorithm:
   - Bellman-Ford algorithm typically uses a dictionary to store the shortest distances from the source vertex to all other vertices where the keys of the dictionary are the nodes and the value of every node is the smallest distance between the source node and the current node.

- Additionally, it Bellman Ford uses another dictionary to store the path from the source node to every other node.
- The space complexity of Bellman-Ford algorithm is O(V), where V is the number of vertices in the graph.

Conclusion:
- Dijkstra's algorithm has a space complexity of O(V + E), where V is the number of vertices and E is the number of edges.
- Bellman-Ford algorithm has a space complexity of O(V), where V is the number of vertices.

Section 2

Part 2

All-Pair Shortest Path Algorithm for positive edge weights:

The algorithm **all_pair_shortest_paths_positive** takes two arguments:

- **graph**, which represents a random graph, generated by the function generate_random_graph(num_nodes) taking the number of nodes as input.
  The graph is in the form of a dictionary where keys are nodes and values are dictionaries of neighboring nodes and their corresponding edge weights,
- **k**, which represents the maximum number of relaxations allowed in the paths.

Design of Algorithm:

Inside this function, two dictionaries **all_shortest_paths** and **all_second_to_last** are initialized to store the shortest paths and their second-to-last vertices respectively.

The code defines a helper function **dijkstra_2** which implements Dijkstra's algorithm to find the shortest paths from a given start node to all other nodes in the graph. It returns a nested dictionary containing the shortest distances from the start node to each node in the graph.

The function then iterates through each node in the graph containing 'n' nodes.

For each source node, it calls the **dijkstra** function from part 1.1 of the lab to find the shortest distances and the previous vertices leading to those distances. These distances are stored in the all_shortest_paths dictionary with the source node as the key.

Additionally, it extracts the second-to-last vertices encountered on the shortest paths and stores them in the all_second_to_last dictionary with the source node as the key.

Once the iteration is complete, the function returns the all_shortest_paths and all_second_to_last dictionaries, containing the shortest paths and their second-to-last vertices for each source node in the graph.

Complexity of algorithm:

For each node in the graph, we are running Dijkstra's algorithm.

In the case of dense graphs, **Dijkstra's** algorithm complexity is approximately $\boldsymbol{\Theta(V^2)}$, as stated.
Since the time complexity of running Dijkstra's algorithm on a single node is $\Theta(V^2)$, we are doing this procedure for all the nodes in the graph (having V nodes and E edges). Therefore, this gives us the resultant complexity of the "all_pair_shortest_paths_positive(graph, k)" to be V x $\Theta(V^2)$, which is $\boldsymbol{\Theta(V^3)}$.

Therefore, the time complexity of the "all_pair_shortest_paths_positive(graph, k)" becomes Θ(V^3).

Similarly, obtaining the second-to-last vertices from the paths obtained involves iterating through the previous vertices for each destination node, **which involves a time complexity of Θ(V).**

Therefore, from the above results we can conclude that the overall time complexity of the**"all_pair_shortest_paths_positive(graph, k)"** algorithm for **positive edge weights** in a dense graph is **Θ(V^3).**

All-Pair Shortest Path Algorithm for negative edge weights:

The algorithm **all_pair_shortest_paths_negative** takes two arguments:

- **graph**, which represents a random graph, generated by the function generate_random_graph_2(num_nodes) taking the number of nodes as input.
  The graph is in the form of a dictionary where keys are nodes and values are dictionaries of neighboring nodes and their corresponding edge weights,
- **k**, which represents the maximum number of relaxations allowed in the paths.

This function computes the shortest paths between all pairs of nodes in a graph where edge weights may be negative. It employs the Bellman-Ford algorithm to handle negative weights and identify negative cycles.

Design of Algorithm:

We create empty dictionaries to store shortest paths and second-to-last vertices.

The code defines a helper function **bellman_ford2** which implements Bellman Ford's algorithm and computes the shortest paths from a given source node to all other nodes in the graph. This algorithm runs for a specified number of iterations (k), updating distances and previous vertices along the way.

After the iterations, the algorithm checks for **negative cycles** by iterating over all edges again. If it finds a shorter path, it indicates the presence of a negative cycle and raises a **ValueError**.

Additionally, it extracts the second-to-last vertices encountered on the shortest paths and stores them in the all_second_to_last dictionary with the source node as the key.

Once the iteration is complete, the function returns the all_shortest_paths and all_second_to_last dictionaries, containing the shortest paths and their second-to-last vertices for each source node in the graph.

Complexity of algorithm:

For each node in the graph, the algorithm calls Bellman-Ford's algorithm.

In a dense graph, Bellman-Ford's algorithm complexity is approximately Θ(VE), which simplifies to **Θ(V^3)** since **E can be at most V^2**.

The outer loop of the all_pair_shortest_paths_negative function iterates through each node in the graph (V times).
For each node, we run the **bellman_ford2** algorithm, an implementation of the Bellman-Ford algorithm, which has a complexity of **Θ(V^3)** in a dense graph.
Hence, calling Bellman-Ford's algorithm for every single node to calculate the shortest path gives a total complexity of **Θ(V^4)**.

After obtaining the shortest paths for each node, the algorithm extracts the second-to-last vertices from the paths obtained. This involves iterating through the previous vertices for each destination node, which takes **Θ(V)** time.

Thus, the algorithm's overall complexity remains **Θ(V^4)** for computing all-pair shortest path and previous nodes in a dense graph with negative edge weights.

Conclusion:
We have analyzed and compared the all-pair shortest path algorithms for both positive and negative edge weights in dense graphs.

For the case of **positive edge weights**, the algorithm **'all_pair_shortest_paths_positive'** utilizes Dijkstra's algorithm to efficiently compute the shortest paths from a given source node to all other nodes. With a time complexity of **Θ(V^3)**, where V represents the number of nodes in the graph, it offers an effective solution for finding shortest paths in scenarios where edges have positive weights. The algorithm provides both the shortest paths and their corresponding second-to-last vertices, facilitating various applications such as network routing and optimization.

On the other hand, for graphs with **negative edge weights**, the algorithm **'all_pair_shortest_paths_negative'** employs the Bellman-Ford algorithm to handle such scenarios. Despite having a higher time complexity of **Θ(V^4)** due to the nature of Bellman-Ford's algorithm, it efficiently detects negative cycles and computes the shortest paths between all pairs of nodes. This capability is vital in scenarios where negative weights represent costs or distances, ensuring accurate route planning and optimization even in the presence of negative edge weights.

Both algorithms demonstrate their effectiveness in solving the all-pair shortest path problem, catering to different scenarios based on the nature of edge weights in the graph.

Section 3

Part 3.2

What issues with Dijkstra’s algorithm is A* trying to address?

Both Dijkstra’s algorithm A* algorithm can be used to find the shortest path in a positive weighted graph, while Dijkstra’s algorithm is effective in finding the shortest path in weighted graphs, it is a very efficient algorithm as compared to A* and lacks the ability to efficiently guide the search towards the destination. In Dijkstra’s algorithm we find the shortest path from the source node to all other nodes, and then from that get the shortest path between the source and the destination. This often leads to the exploration of unnecessary paths, especially in scenarios where the goal is in a particular direction but Dijkstra's algorithm still explores in all directions uniformly. On the other hand, A* aims to address this issue by introducing a heuristic function that returns a dictionary of heuristic costs, where each node is the key and the value is the heuristic cost of that node. The heuristic cost provides information about the direction to the destination, allowing for a more directed search and potentially reducing the search space and making the program execution faster. Thus, reducing the time complexity.

Empirical Testing Methodology

To empirically compare Dijkstra’s algorithm with A*, various experiments can be conducted. The various experiments can be conducted on the **basis of the size of the graph**, this would help us understand that which algorithm is better used for bigger graphs.

Testing **on the basis of the density of the graph**, this would allow us to see, as the number of possible paths between the start and the destination nodes keep on increasing, which algorithm is more efficient in choosing the path thus being faster and having a higher efficiency.

The algorithms can then be run on these graphs, measuring their runtime and memory usage. Additionally, experiments can involve different pairs of source and destination nodes with varying distances between them to check the efficiency of both algorithms.

Comparison with Arbitrary Heuristic Function

We know that Dijkstra’s algorithm doesn’t decide the shortest path on the basis of some external dictionary/external cost criteria like heuristic cost in the A* star algorithm, Dijkstra’s algorithm follows a standard procedure for determining the shortest paths.

If an arbitrary heuristic function, such as randomly generated weights, is used, the behavior of Dijkstra’s algorithm and A* would differ significantly. Dijkstra’s algorithm would continue to explore paths uniformly, without any preference of the path towards the goal. In contrast, A* would utilize the heuristic function to prioritize exploration in the direction of the goal. As a result, A* would likely outperform Dijkstra’s algorithm by exploring fewer nodes and finding the optimal path more efficiently.

Applications of A* over Dijkstra's

A* being faster than Dijkstra's algorithm is particularly suitable for applications where finding the shortest path is crucial. However, for A* we need additional information about the nodes of the graph to make the intelligent choice. Therefore, we having some of the following applications where A* outperform and is much more efficient than Dijkstra's algorithm. Some of the applications include:

- Robotics Path Finding: A* algorithm finds extensive use in robotics for path planning of autonomous robots and drones. It helps robots navigate complex environments, avoid obstacles, and reach target locations efficiently.
- Maze Solving: A* algorithm is used in solving mazes or puzzle games where finding the shortest path from a start point to an end point is required. Because of it's ability to intelligently search the solution space makes it suitable for solving maze-like structures.
- Terrain Analysis and GIS, GPS: Geographic Information Systems (GIS) utilize A* algorithm for analyzing terrains, planning routes, and performing spatial analysis. It helps in determining optimal paths over terrains while considering different heuristics like elevation, slope, and other geographical features.
- Puzzle Solving: A* algorithm is applied in various puzzle-solving scenarios, such as the famous Eight Puzzle or Fifteen Puzzle. It efficiently searches through the puzzle state space to find the optimal sequence of moves to reach the goal state

Section 4

Part 4

The experiment compares the Dijkstra algorithm with the A* algorithm under 3 different scenarios. The first scenario is when the 2 stations are running on the same line, the second is when they are running on parallel lines, and the third when they are on completely different lines. This is to test the effectiveness of A*'s heuristic functionality.

From our experiment, it is evident that A* is far more effective at selecting paths than Dijkstra's algorithm when comparing time. By utilizing the heuristic function, the A* algorithm gets a stronger estimate of how much further it needs to 'move' to reach its destination node, something that Dijkstra algorithm cannot do. Through this method, A* can delegate its resources towards only those paths that have a higher chance of reaching the destination whereas Dijkstra is forced to investigate every path available to it in order to arrive at an answer. In this sense, the stronger the heuristic function, the stronger the A* algorithm. It is because of this 'greedy' nature of A* that it can choose/find the optimal path much faster than Dijkstra.

A* really shines when the destination node is closer to the source node. Less edges to investigate means less time taken to arrive at the destination node. Even with multiple nodes, A* appears to do well so long as the edges between nodes are distinct (like it is in the example data). A* has the advantage when the pathway is distinct since Dijkstra seeks to iterate through a larger number of nodes to find the best path, whereas A* utilizes the heuristic function and uses a sort of 'short-cut'. It could become more complicated, however, if multiple stations are at similar lines. This is where the heuristic function would not have as large of an impact on the efficiency of the A* algorithm. Having too large of a graph to traverse could mean that the heuristic does not cover the 'bases' needed for the A* algorithm to be optimal in its search pattern. Similar to Dijkstra, A* would essentially be traversing through a large number of nodes to find the shortest path. Dijkstra, on the other hand, would perform as uniformly as it usually does, resulting in both having similar results.

Station observations:
1) A* efficiently finds the best path when the station are closer to each other on the same line. This is, again, due to the heuristic function helping provide A* already paths to the second station whereas Dijkstra must find it through more exhaustive methods.
2) Stations that are on parallel lines could make A* significantly slower to the point where it performs worse, or at the very least similar, to Dijkstra's algorithm.
3) Stations on the same line could potentially slow A* but more or less keeps it efficient.

By looking at the "line" information, we can count how many times we need to change lines to get to the destination. This tells us how intricate the route is in terms of changing trains. It also shows us how well A* handles finding the best path through the subway system

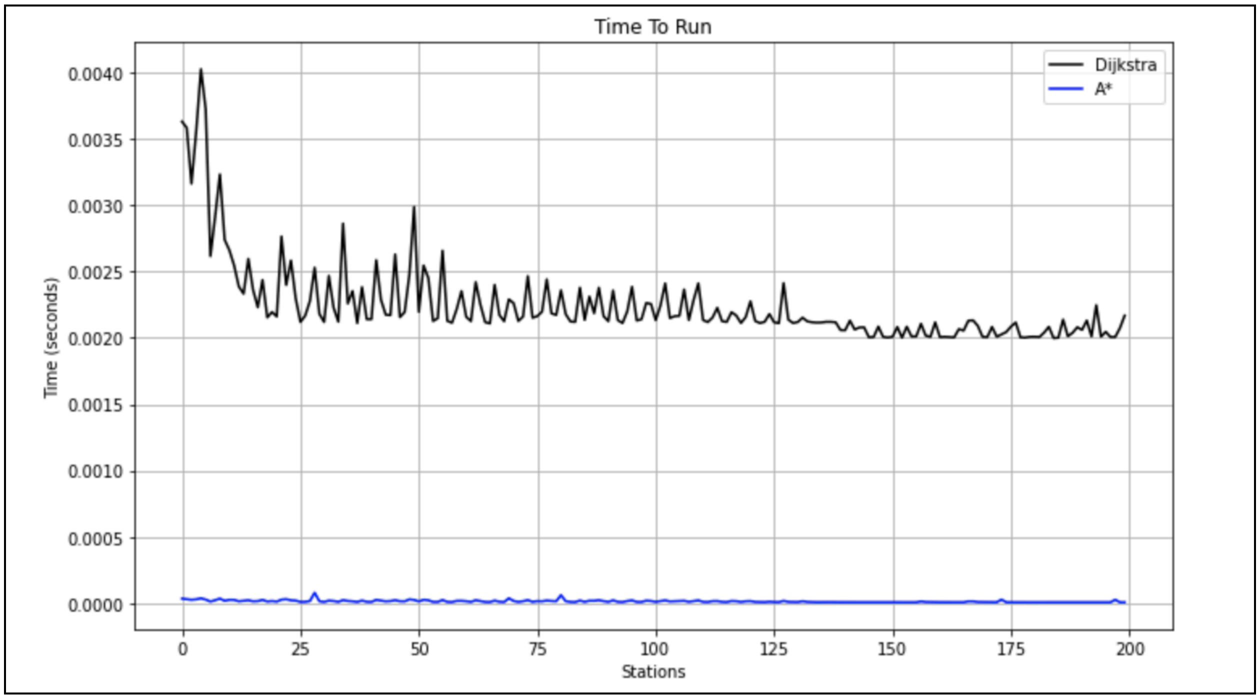

Figure 4.1: Comparison when stations are on the same line

The graph plotted above showed a uniform straight line for the A* algorithm which made it look like A* is taking no time (0 secs) to find the shortest path.
A closer look of the time taken by the A* algorithm has been provided below.

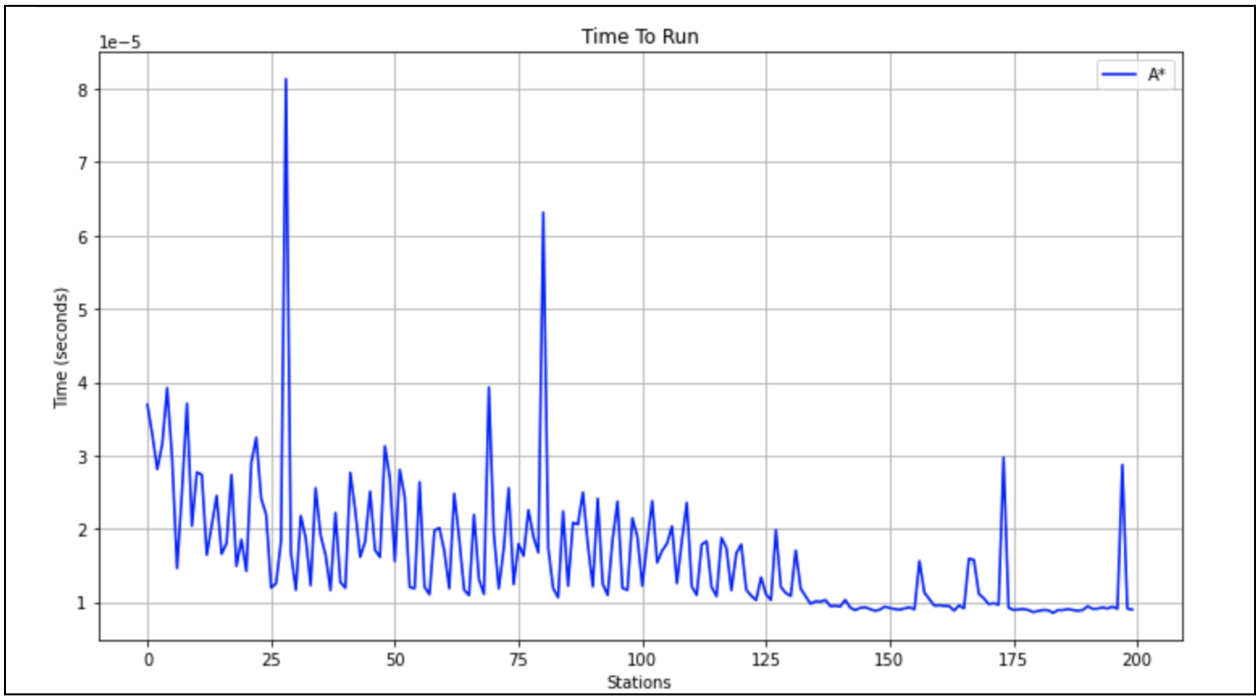

Figure 4.2: A closer look at the A* algorithm from figure 4.1

From the above graph it becomes quite evident that the time taken by the A* algorithm is very less as compared to the Dijkstra's algorithm, making it more efficient and faster for finding the shortest paths.

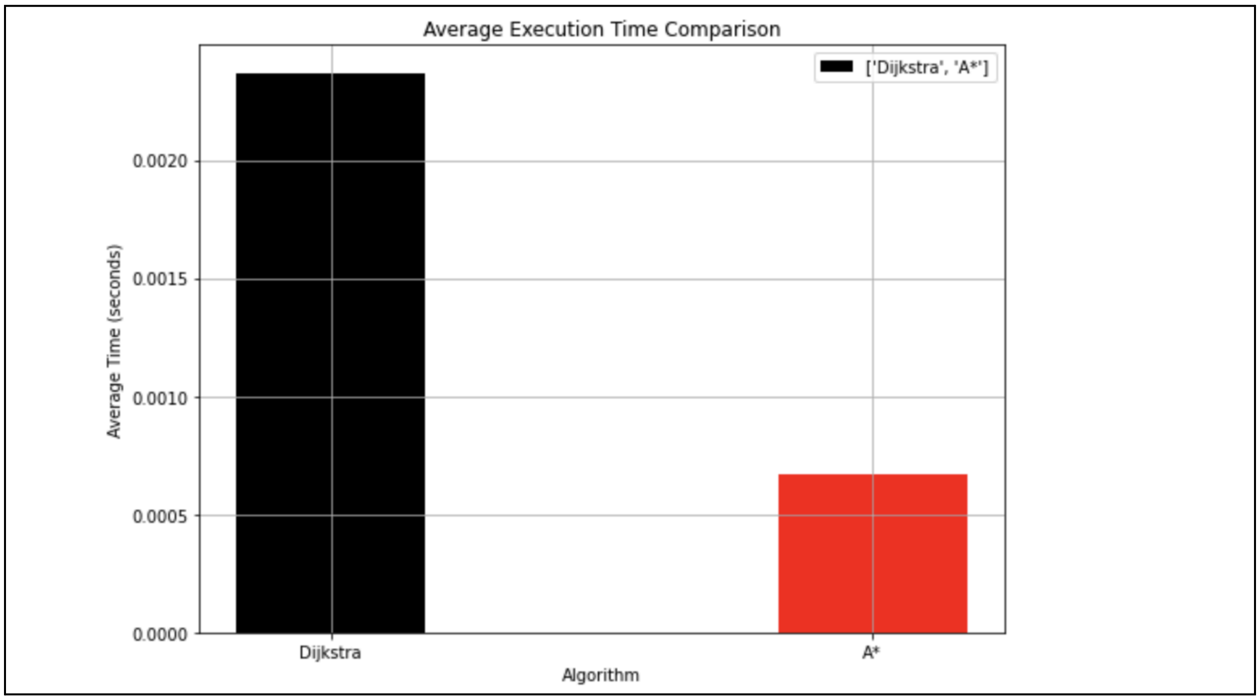

Figure 4.3: Comparison of average time when stations are on the same line

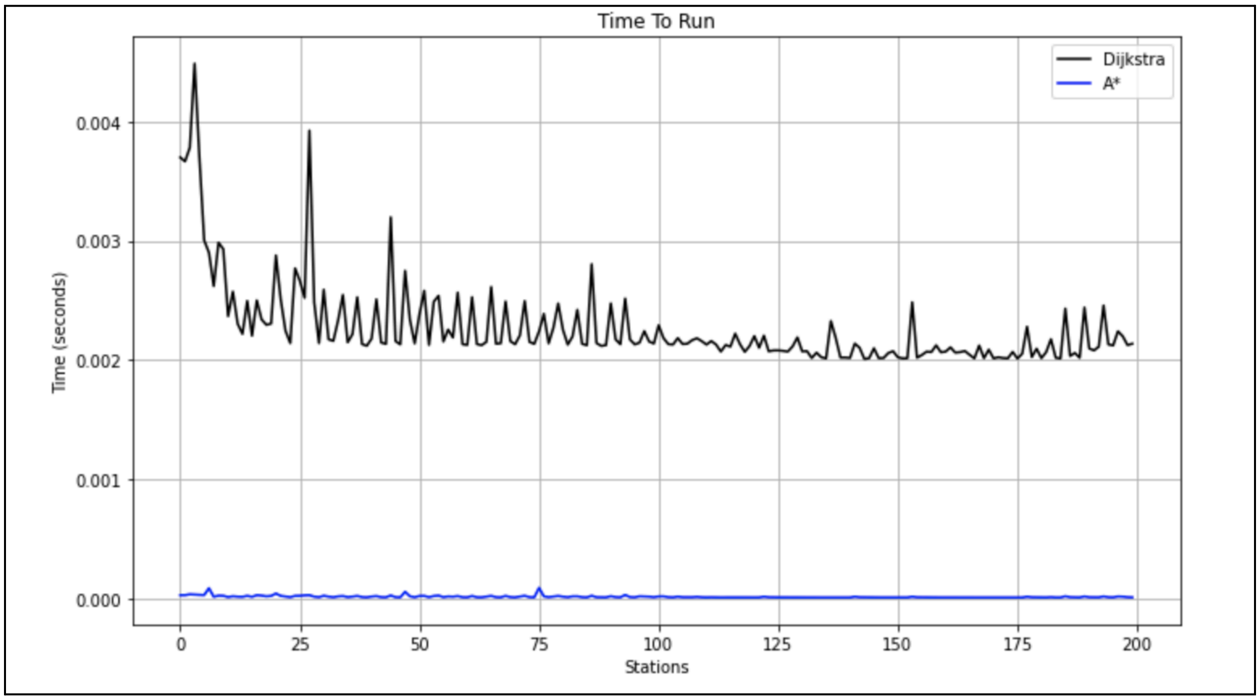

Figure 4.4: Comparison when stations are on parallel lines

The graph plotted above showed a uniform straight line for the A* algorithm which made it look like A* is taking no time (0 secs) to find the shortest path.
A closer look of the time taken by the A* algorithm has been provided below.

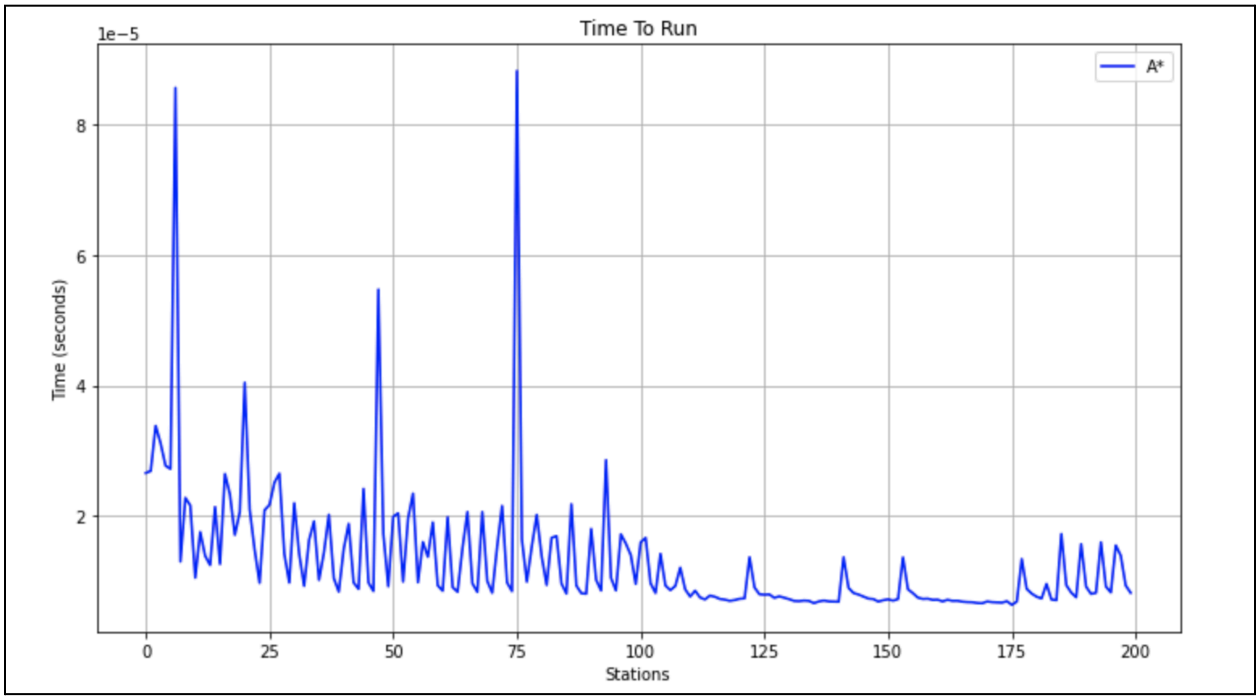

Figure 4.5: A closer look at the A* algorithm from figure 4.4

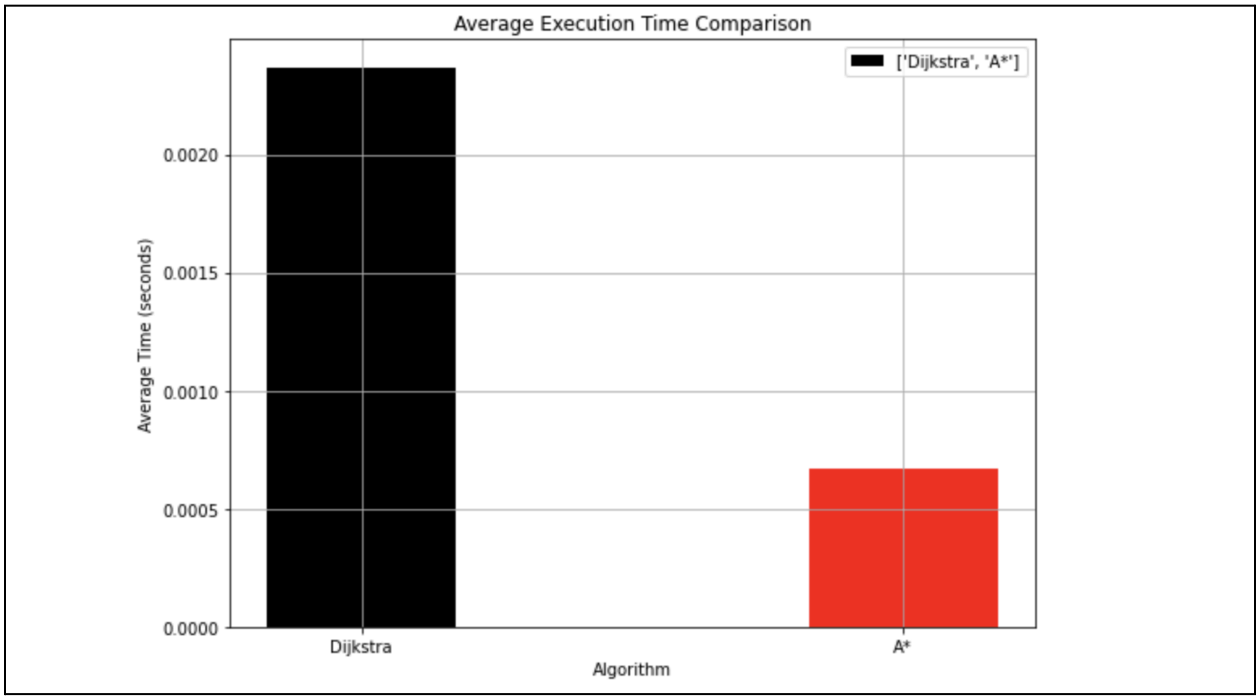

Figure 4.6: Comparison of average time when stations are on parallel lines

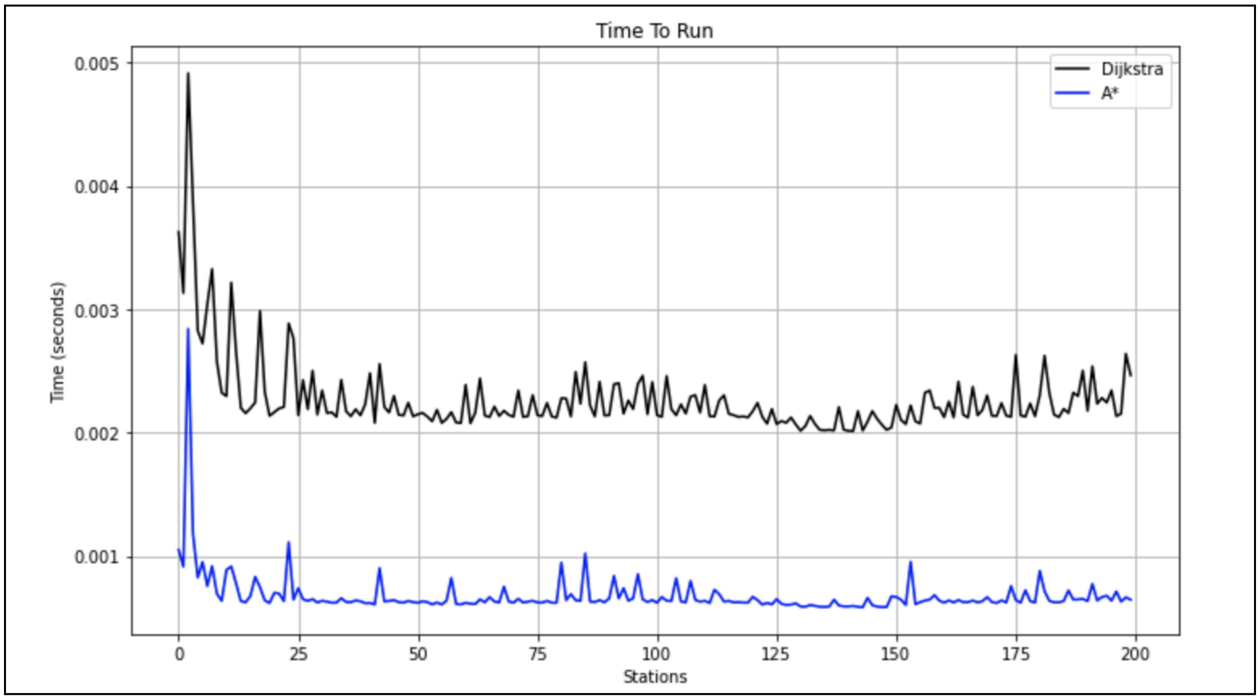

Figure 4.7: Comparison when stations are on completely different lines

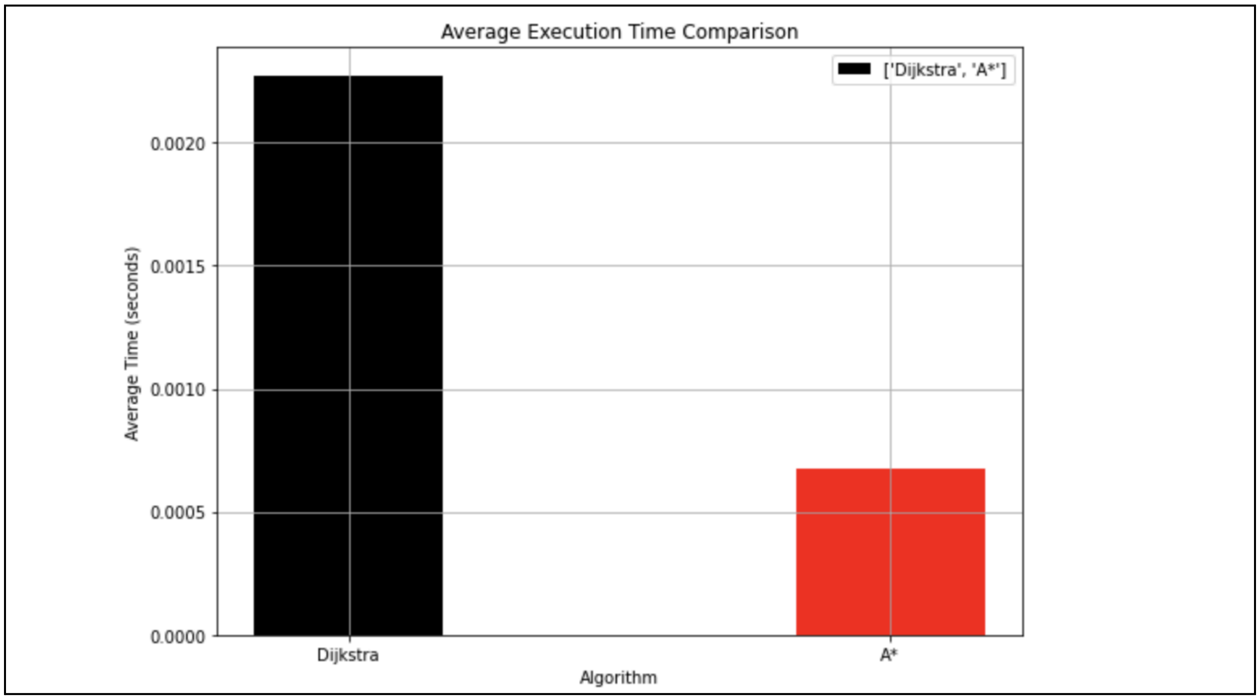

Figure 4.8: Comparison of average time when stations are on completely different lines

Section 5

We have implemented the **second option** by creating a separate .py file for Part 5. We have pasted the code from Parts 1-4 and organized it as per the UML.

A* algorithm implementation as an "Adapter"

The UML class diagram (Figure 2) depicts a framework for shortest path algorithms. The existing structure includes abstract classes (<Graph> and <SPAlgorithm>) and concrete classes (WeightedGraph, HeuristicGraph, etc.). We to integrate the A* algorithm into this UML without modifying the existing classes directly.

Adapter Pattern Overview

The adapter pattern allows us to make incompatible interfaces work together. It acts as a bridge between two different interfaces, enabling them to collaborate seamlessly. In our case, we want to adapt the A* algorithm to fit the existing <SPAlgorithm> interface.

Implementation Steps

- **Creating the AStar Adapter**

  We introduce a new class called AStar that extends <SPAlgorithm>. This class serves as the adapter for the A* algorithm. Here's how we implemented it:

  ```
  class AStar(SPAlgorithm):
     class PriorityQueue:
        # ... (PriorityQueue implementation)

     def calc_sp(graph: Graph, source: int, dest: int) -> float:
           heuristic = graph.get_heuristic()
        # A* algorithm implementation
  ```

  The key modification incorporates the **get_heuristic()** method from the HeuristicGraph class.

- **Priority Queue:**
  We have added a priority queue class that is essential for A* since it manages the priority of nodes during traversal.

- **Calculation of Total Cost:**
  - We calculate the total cost as the sum of the actual cost (accumulated cost from the source) and the heuristic value.
  - The heuristic value is obtained by calling graph.get_heuristic(neighbor) for each neighboring node.

- **Priority Queue Insertion**:
  - The priority for inserting nodes into the priority queue is now total_cost[neighbor] + heuristic[neighbor].
  - This ensures that nodes with lower total cost (including the heuristic estimate) are explored first.

- **Path Retrieval and Backtracking:**

  The backtracking process remains the same, constructing the shortest path using the predecessor_dictionary.

Robustness and Flexibility:

By adapting the A* algorithm to the existing framework, we maintain compatibility with other shortest-path algorithms.
The adapter pattern allows us to incorporate A* seamlessly while preserving the original class hierarchy.

Conclusion

By implementing the AStar class as an adapter, we seamlessly integrate the A* algorithm into the existing shortest path framework. The adapter pattern allows us to extend the system's functionality without altering the original classes, promoting modularity and maintainability.

Design Principles and Patterns:

- ### Encapsulation:

  The classes effectively encapsulate their data and behavior. Methods like calc_short_path() and add_edge() along with attributes such as heuristic are encapsulated within their respective classes, ensuring proper data hiding and abstraction.

- **Inheritance:**

  The diagram illustrates a clear inheritance hierarchy. WeightedGraph and HeuristicGraph inherit from the abstract class <Graph>, while Dijkstra, Bellman_Ford, and A_Star inherit from <SPAlgorithm>. This inheritance structure promotes code reusability and allows for the extension of functionality.

- **Polymorphism:**

  Through inheritance, subclasses like Dijkstra and Bellman_Ford can exhibit polymorphic behavior by overriding or extending the functionality of methods defined in their superclass <SPAlgorithm>. This enables different shortest-path algorithms to be utilized interchangeably.

- **Abstraction:**

  Abstract classes such as <Graph> and <SPAlgorithm> provide a blueprint for subclasses to implement specific functionalities. They abstract away implementation details, allowing for a high-level view of the system's architecture.

Class Hierarchy:

- **ShortPathFinder:**
  This class acts as the central orchestrator responsible for finding the shortest paths. It likely coordinates interactions between other classes, facilitating the overall pathfinding process.

- **<Graph> (Abstract):**
  As an abstract class, <Graph> defines a common interface for subclasses to implement graph-related functionality. Its subclasses, WeightedGraph and HeuristicGraph, specialize in handling weighted graphs and graphs with heuristics, respectively.

- **<SPAlgorithm> (Abstract):**
  Similarly, <SPAlgorithm> abstracts away specific shortest path algorithms, providing a common interface for subclasses like Dijkstra, Bellman_Ford, and A_Star to implement their respective algorithms.

Attributes and Methods:

- Each class box contains attributes (variables) and methods (functions) relevant to its functionality. For instance:
  - ShortPathFinder includes methods like calc_short_path(source: int, dest: int) and set_graph(graph: Graph).

- <Graph> defines methods like get_adj_nodes(node: int) and add_edge(start: int, end: int, w: float).
- WeightedGraph likely introduces specific weighted graph-related attributes and methods.
- HeuristicGraph incorporates a heuristic attribute and a method to retrieve heuristics.

Inheritance Relationships:

Arrows depict inheritance relationships, indicating that subclasses inherit attributes and methods from their parent classes. This inheritance mechanism facilitates code reuse and promotes a hierarchical structure within the system.

Abstraction and Encapsulation:

Abstraction allows for a high-level representation of functionality without delving into implementation details, enhancing modularity and maintainability. Encapsulation ensures that each class encapsulates its data and behavior, promoting information hiding and minimizing dependencies between components.

Overall, the UML class diagram demonstrates a thoughtful application of object-oriented design principles and patterns, fostering a modular and extensible architecture suitable for developing efficient shortest-path algorithms.

Modified Node representation

To accommodate more flexible representations of graph nodes, we can make the following modifications to the UML diagram:

Node Representation:

We can introduce a new class specifically for nodes, called **Node**, which encapsulates information about a graph node. This class can have attributes like id, label, and additional data relevant to the node.

Modify Existing Classes:

- **<Graph>:**
  - We'll update methods like add_node(node: int) to accept a Node object.
  - Modify the get_adj_nodes(node: int) method to return adjacent nodes (also as Node objects).
  - WeightedGraph and HeuristicGraph:

    - We'll adapt these classes to work with Node objects.
    - If needed, we'll add attributes specific to weighted or heuristic graphs.

- **<SPAlgorithm>:**
    - We'll ensure algorithm classes (e.g., Dijkstra, Bellman_Ford) operate on Node objects.
    - Modify the calc_sp(graph: Graph; source: int; dest: int) method to accept Node objects as arguments.

Node Attributes:

The Node class can have various attributes such as:

id: A unique identifier for the node (e.g., an integer or string).

label: A human-readable label (e.g., “A,” “B,” “New York,” etc.).

Additional data specific to the application (e.g., coordinates for geographical nodes, population for city nodes).

Heuristics and Custom Data:

For heuristic graphs, the HeuristicGraph class can store additional data related to heuristics (e.g., estimated distances from each node to the goal).
We can extend the Node class to include attributes relevant to heuristics (e.g., heuristic_value).

Edge Representation:

We can enhance the diagram by adding an Edge class, this class can be used to store edge attributes like start_node, end_node, and weight.

Overall Benefits:

With the introduction of a Node class and adaptation of existing classes, we achieve greater flexibility in representing nodes.
The design becomes more extensible, allowing for custom data associated with nodes (e.g., geographical coordinates, population, etc.).
Algorithms can work with richer node information, improving their accuracy and applicability.

Other types of Graph implementation

The <Graph> class can be extended to handle various types of graphs beyond the basic representation. Here are some additional implementations:

Finite Graphs:

These graphs have a finite number of vertices and edges. Implementation involves handling a fixed number of nodes and edges using data structures like arrays or dictionaries.

Infinite Graphs:

Infinite graphs have an unbounded number of vertices and edges. Implementations may use lazy evaluation or generators to represent infinite sequences of nodes. Algorithms must handle infinite graphs, possibly with termination conditions.

Weighted Graphs:

Weighted graphs associate a weight or cost with each edge. Extend the <Graph> class to include edge weights and modify algorithms like Dijkstra's to consider these weights.

Directed Graphs (Digraphs):

Edges have a direction in directed graphs. Modify the <Graph> class to handle directed edges and add methods for querying incoming/outgoing edges.

Acyclic Graphs (DAGs):

Directed acyclic graphs have no cycles. Implementation involves enforcing acyclic behavior and implementing topological sorting algorithms.

Random Graphs:

Generating random graphs with random number of nodes and edge weights.