Final Project

Yanna Lazarova, Lana Zgonjanin, Oriana Rueckert

Dr. Swati Mishra

April 9, 2025

Table of Contents and Figures

Executive Summary

This report focuses on the implementation and analysis of various shortest path algorithms, specifically single-source shortest path and all-pairs shortest path algorithms.

Part two implements' variations of Dijkstra's and Bellman Ford with an added constraint on the number of relaxations per node/edge. The time/space complexity and accuracy of Dijkstra's and Bellman Ford is analyzed on a variety of graphs with different sizes, densities and values of k (number of relaxations). Results demonstrate that Dijkstra's generally performs better due to its $O((V+E) \log V)$ time complexity as opposed to Bellman Ford's $O(VE)$ time complexity. Bellman Ford performs better for sparse graphs and small values of k due to its early termination. Furthermore, Dijkstra's and Bellman Ford both have a space complexity of $O(V+E)$ regardless of input. Finally, when k < V—1, Bellman Ford's accuracy diminishes as it returns suboptimal paths instead of the shortest paths.

Part three depicts an all-pairs shortest path algorithm which works for both negative and positive edge weights. This algorithm emulates the Floyd-Warshall algorithm and has a time complexity of $O(N^3)$ in all cases.

Part four implements the A* algorithm, which uses a heuristic function to determine the best path between two given nodes. The heuristic function makes A* more efficient than Dijkstra's because it bypasses irrelevant paths and prioritizes paths closest to the goal based on the heuristic. Adequately comparing the performance of A* and Dijkstra's requires modifying the heuristic function, $h(n)$. If $h(n) = 0$, the heuristic function provides no guidance and A* is identical to Dijkstra's. If $h(n) << cost$, the heuristic function underestimates the cost to the goal. In this case, the heuristic function provides very little guidance and A* behaves very similarly to Dijkstra's. If $h(n) >> cost$, the heuristic function overestimates the cost to the goal. In this case, the heuristic function provides skewed guidance and A* may not find the shortest path, although it runs faster. Consequently, it is evident that the heuristic function must be carefully chosen. A poorly chosen heuristic function can mislead the A* algorithm, causing it to perform worse than Dijkstra's. However, with a good heuristic function (like in a GPS), A* can find the shortest path between two destinations much faster than Dijkstra's and is the better choice.

Part five compares the performance of Dijkstra's and A* on a real-world data set—the London subway system—through a fair experiment. To ensure a fair experiment, Dijkstra's and A* find the shortest path between every pair of stations. A* always outperforms Dijkstra's due to its good heuristic function. A* and Dijkstra's are comparable when the two stations are on the same line because the shortest path is direct with no transfers, and both algorithms can find it quickly. As the number of transfers between stations increases, the performance of both algorithms slows as more paths must be explored. Experimental results conclude that the shortest bath between two stations requiring several transfers uses four lines on average.

Part six focuses on organizing the code written in previous sections using object-oriented programming principles to make it more flexible and reusable. Design patterns were utilized to seamlessly integrate many modules with each other.

Part 1: Team charter.

1. Our team will communicate through text, call and in-person meetings. To ensure smooth group operations, we will expect responses to communications within the same day during the week.
2. If team members repeatedly fail to adhere to our communication agreement, we will set up an in-person team meeting to discuss the negative effects caused by lack of communication and to catch-up on missed work together.
3. We will use a shared Word document to collaborate on the final report. To ensure collaboration within our team for the coding aspect, we will use GitHub for version control and VS Code.
4. We will resolve team disputes by frequently seeing each other in-person or meeting online, as resolving conflicts can be difficult over text. We will be patient with each other and recognize that everyone is doing their best and wants our team to succeed. We will resolve major disagreements via a vote, since out team consists of three people.
5. There are no further materials we wish to include in our team contract.

Part 2: Single source shortest path algorithms.

The shortest_path_experiment(*sizes*, *densities*, *k_values*) allows us to analyze the performance of Dijkstra's and Bellman Ford for graphs of different sizes, densities and values of *k*. Using this information, we can draw conclusions regarding the accuracy and time/space complexity of Dijkstra's and Bellman Ford. The experiment takes in three equal length lists of user-selected sizes, densities and values of *k*. The idea is to vary a set of attributes incrementally while keeping the remaining attributes relatively constant and observe how performances changes under each set of conditions. For the set of conditions given by index *i* of each input list, a random graph is generated for the number of trials given. This experiment design is very flexible and can help deduce many conclusions. A few of the conclusions we can make are shown below:

1. Graphs with a small size, increasing densities and a large value of *k*.
   shortest_path_experiment([20, 20, 20, 20, 20], [0.2, 0.3, 0.5, 0.75, 1], [15, 15, 15, 15, 15])

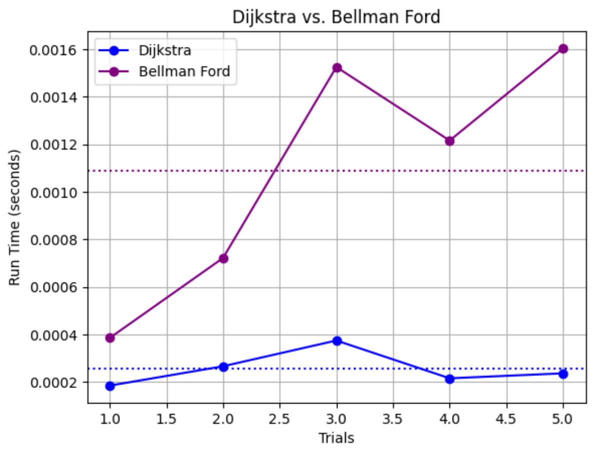

Figure 1: Dijkstra vs. Bellman Ford with graphs of increasing densities

We can observe that as graph density increases, the performance of Bellman Ford slows while the performance of Dijkstra’s remains similar. Overall, we can also observe that Dijkstra’s is faster than Bellman Ford. This is because Bellman Ford has a time complexity of *O(VE)* while Dijkstra’s has a time complexity of *O((V+E) log V)*. For dense graphs *O((V+E) log V)* ≈ *O(E log V)*, which is still much faster than *O(VE)*. Furthermore, Bellman Ford must process all edges *k* times, while Dijkstra’s processes each edge once. This makes Bellman Ford much slower when there are many edges.

2. Graphs with increasing sizes, sparse density and a small value of *k*.
shortest_path_experiment([10, 30, 50, 80, 120], [0.1, 0.1, 0.1, 0.1, 0.1], [3, 3, 3, 3, 3])

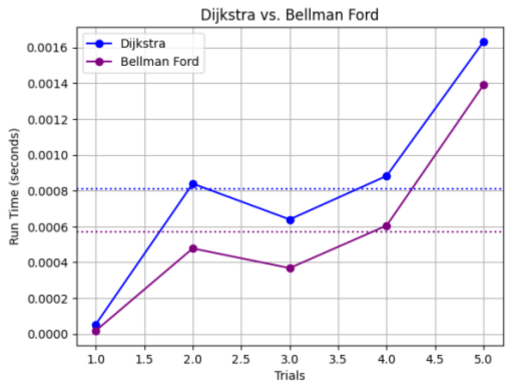

Figure 2: Dijkstra vs. Bellman Ford with graphs of increasing sizes

We can observe that for sparse graphs with a small value of *k*, Bellman Ford performs better no matter the size. For small *k*, Bellman Ford processes all edges *k* times so *O(VE)* ≈ *O(kE)*. For sparse graphs, the number of edges is approximately the number of vertices so *O(kE)* ≈ *O(kV)*. Meanwhile, Dijkstra’s time complexity remains similar as *O((V+E) log V)* ≈ *O(V log V)*. The for-loop in Bellman Ford ends much earlier than normal when *k* is very small compared to the size of the graph, significantly improving run time. In comparison, Dijkstra’s runtime remains relatively unchanged because it still processes graphs as normal aside from stopping edge relaxations early.

3. Graphs with moderate sizes, moderate density and increasing *k*.
shortest_path_experiment([40, 40, 40, 40, 40], [0.3, 0.3, 0.3, 0.3, 0.3], [5, 10, 20, 30, 38])

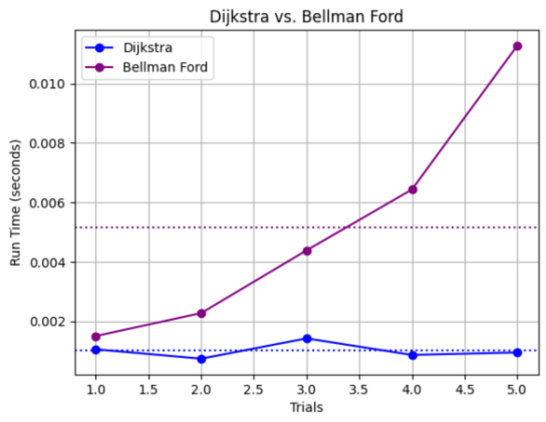

Figure 3: Dijkstra vs. Bellman Ford with graphs of increasing *k*

This experiment combines our observations from the first two experiments. We can observe that for average graphs, Dijkstra's performance is faster than Bellman Ford. We can also observe that as the value of *k* increases, Bellman Ford performs slower as it does more work. Meanwhile, increasing *k* values do not significantly affect Dijkstra's performance.

The above experiments depict how graphs of varying sizes, densities and *k* values impact the time complexities of Dijkstra's and Bellman Ford. We can further analyze how space complexity and accuracy are impacted by these constraints. Normally, Dijkstra's and Bellman Ford both have a space complexity of *O(V+E)*. Large, dense graphs require more storage while small spare graphs require less. However, these factors do not change the *O(V+E)* space complexity as the algorithms structure remains the same. Similarly, *k* does not alter the structure of the algorithms or their space complexity. On the other hand, the value of *k* does impact the accuracy of Dijkstra's and Bellman Ford. Since Dijkstra's is a greedy algorithm which selects the minimum weighted edge from the priority queue in each iteration, as long as $k \geq 1$, Dijkstra's will output the correct distance dictionary. However, Bellman Ford must normally relax all edges V—1 times. If $k <$ V—1, the distance dictionary returned by Bellman Ford may not be accurate. For example, when the shortest path to a node has more than *k* edges, the path to that node found by Bellman Ford will be suboptimal. Overall, we can see how the inputs passed into Dijkstra's and Bellman Ford may greatly influence their performance regarding time/space complexity and accuracy.

Part 3: All-pair shortest path algorithm.

1. What would you conclude the complexity of your algorithm to be for dense graphs? Explain your conclusion in your report.

The complexity of my all-pairs shortest path algorithm is $O(n^3)$ in all cases because of the 3 nested for-loops. My function follows the Floyd-Warshall algorithm which will always find the all-pairs shortest path for any value edge weight if there are no negative cycles. I decided to use the Floyd-Warshall algorithm instead of Dijkstra's repeated n times because Dijkstra's would not work for negative edge-weights, and the runtime of Dijkstra's compared to Floyd-Warshall is only marginally better. I decided not to repeat Bellman-Ford n times because in a dense graph the runtime would be $O(n^4)$, which is much worse than Floyd-Warshall.

Part 4: A* algorithm.

1. What issues with Dijkstra's algorithm is A* trying to address?

When trying to find the shortest path between a source node and a destination node, Dijkstra's algorithm explores all nodes based on smallest edge weights. It does not consider whether these nodes are close to the destination node, thus it may end up exploring nodes that are not close to the destination node. A* uses a heuristic function which assigns a value to each node, where this value is an estimate of the distance of the node from the destination. This heuristic function helps the algorithm prioritize exploring nodes that are closer to the destination node, thus reducing the number of total number of nodes visited. Therefore, A* tries to reduce the number of nodes that Dijkstra's visits to increase the efficiency of finding the shortest path between two nodes.

2. How would you empirically test Dijkstra's vs A*? Describe the experiment in detail.

To empirically test Dijkstra's vs A*, we would create an experiment which allows us to compare the efficiency of both algorithms in finding the shortest path between a source and destination node. Since the performance of A* depends on the heuristic function, we will test three different implementations of A* by varying the heuristic function. A*1 will use $h(n) = 0$, meaning it should have the same performance as Dijkstra's. A*2 will use a weak heuristic, so $h(n) << \text{cost}$, meaning it will likely be slower and is guaranteed to find the shortest path. A*3 will be an overestimation, so $h(n) >> \text{cost}$, meaning it will likely be faster, but it isn't guaranteed to find the shortest path. To compare these four algorithms, we test each of them on the same graph using the same source and destination nodes. We time how long each takes and document the shortest path computed, allowing us to compare their efficiencies. We test them on multiple different graphs, varying the number of edges, source, and destination. This experiment provides results which reflect the performance of Dijkstra's and A* across different graph structures, edge distributions, and heuristic effectiveness, ensuring a fair comparison of efficiency and correctness.

Before the experiment, we define N to be the number of nodes, which will stay constant throughout the experiment, and T to be the number of trials of the experiment. We initialize four empty arrays to store the run times of each algorithm: dijkstra_times, A*1_times, A*2_times, A*3_times. We initialize another four empty arrays which will be used to store the shortest path found by each algorithm each time: dijkstra_sp, A*1_sp, A*2_sp, A*3_sp. This will let us ensure that the correct shortest path was found, which is especially useful for A*3 since it is not guaranteed to find the shortest path.

The experiment then begins by randomly generating the number of edges E such that $N-1 \leq E \leq N(N-1)/2$ and there are no self-loops. Then, a random undirected graph with N nodes and E edges is generated, with a random integer edge weight assigned to each edge (none of which are negative). A random source node and random destination node are picked from the set of nodes in the graph (we know there exists a path between them since the graph is connected). Start the timer, call Dijkstra's on the graph, source, and destination, and then end the timer. Calculate the execution time and add it to the dijkstra_times array. Add the found shortest path to the dijkstra_sp array. Repeat this for A*1, A*2, A*3. The first trial of the experiment is now complete. Repeat the experiment T times.

3. If you generated an arbitrary heuristic function (like randomly generating weights), how would Dijkstra's algorithm compare to A*?

If we generated an arbitrary heuristic function, Dijkstra's may now be faster than A*. This is because the heuristic function would no longer provide a meaningful guide towards the destination node. Instead, the arbitrary function could mislead the algorithm, causing it to explore nodes that will not lead to the destination, and thus resulting in slower performance. For example, if the arbitrary function overestimates the actual cost, Dijkstra's will now be faster than A*. But, if by some chance the heuristic function is small and biased towards the destination, then A* could still be faster than Dijkstra's.

4. In what applications would you use A* instead of Dijkstra's?

A* would be used instead of Dijkstra's when the shortest path between two nodes is needed quickly and there is a good heuristic function available, meaning $h(n) \leq \text{cost}$ and $h(n)$ is consistent. For example, A* is used in navigation systems, like GPS. GPS is used to quickly find the shortest path between two locations, which is exactly what A* does. A good heuristic function would be an estimate of the cost from the start location to the destination, ensuring that it is not an overestimate.

Part 5: Comparing shortest path algorithms.

1. When does A* outperform Dijkstra? When are they comparable? Explain your observation why you might be seeing these results.

A* outperforms Dijkstra's each time, as can be seen in Figures 4-7 and Table 1. This is because it uses a good heuristic function to estimate the distance from a source to a destination, allowing it to prioritize stations that are likely to lead to the shortest path. Dijkstra's explores all possible paths uniformly, causing it to visit many unnecessary nodes that A* does not visit, thus making it slower than A*. Dijkstra's performance is most comparable to A*'s performance when the stations are on the same line, which is seen in Figure 5 and Table 1. This is because the shortest path is direct with no transfers and both algorithms discover the destination quickly. A* still performs slightly better because of the heuristic, but since the shortest path follows the physical layout of the stations, Dijkstra's reaches the destination almost as quickly. This is especially true since it includes an early termination once the destination station has been reached. Thus, Dijkstra's explores a similar number of stations as A*, making it comparable.

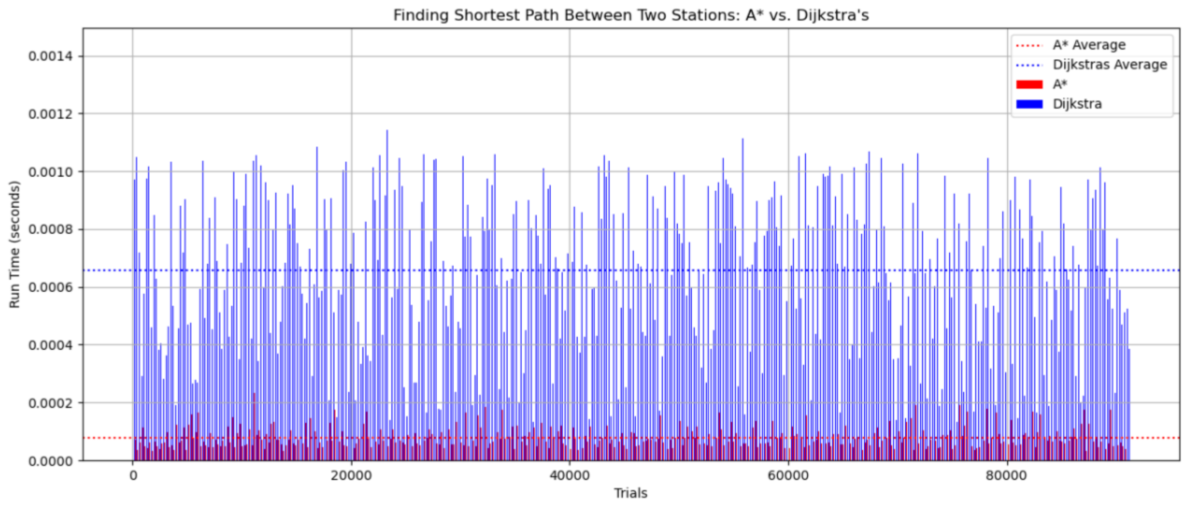

Figure 4: Finding the shortest path between two stations: A* vs. Dijkstra's

2. 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 general, we observe that the performance of both A* and Dijkstra's decreases as the number of transfers needed increases (execution time is lowest when stations are on the same line and highest when stations require more than two transfers). This can be seen in Figures 5-7 and Table 1. Although single-source Dijkstra's typically performs the same regardless of whether stations are on the same line or not, this source to destination Dijkstra's terminates when it reaches the destination, thus making it more efficient. A* is always more efficient because it uses a well-designed heuristic function. Both algorithms decrease in efficiency as the number of transfers increases because the paths involve more stations, more branching at transfer stations, and a larger search space. This causes the algorithms to explore more possibilities before finding the shortest path, resulting in longer execution times.

As explained in the question above, when stations are on the same line, both A* and Dijkstra's perform well because the path is direct and there is minimal exploration needed by both algorithms. Although Dijkstra's has no guide, it often will reach the destination quickly since it is closer to the source (on the same line) and thus terminates, making it faster than if the destination wasn't on the same line. A* is still faster because the heuristic, Euclidean distance, aligns closely with the actual path, allowing it to prioritize the correct direction and avoid unnecessary exploration entirely.

When stations are on adjacent lines, both algorithms become slightly slower, and Dijkstra's becomes less comparable to A*. The decrease in performance is due to the added complexity of requiring a transfer, which increases the number of possible paths to evaluate. This especially impacts Dijkstra's performance, as it explores all stations uniformly without guidance, while A* continues to benefit from its heuristic to prioritize promising directions. As a result, A* begins to outperform Dijkstra more noticeably. When stations require several transfers, the performance of both algorithms further decreases for the same reasons (due to even greater path complexity and larger search space).

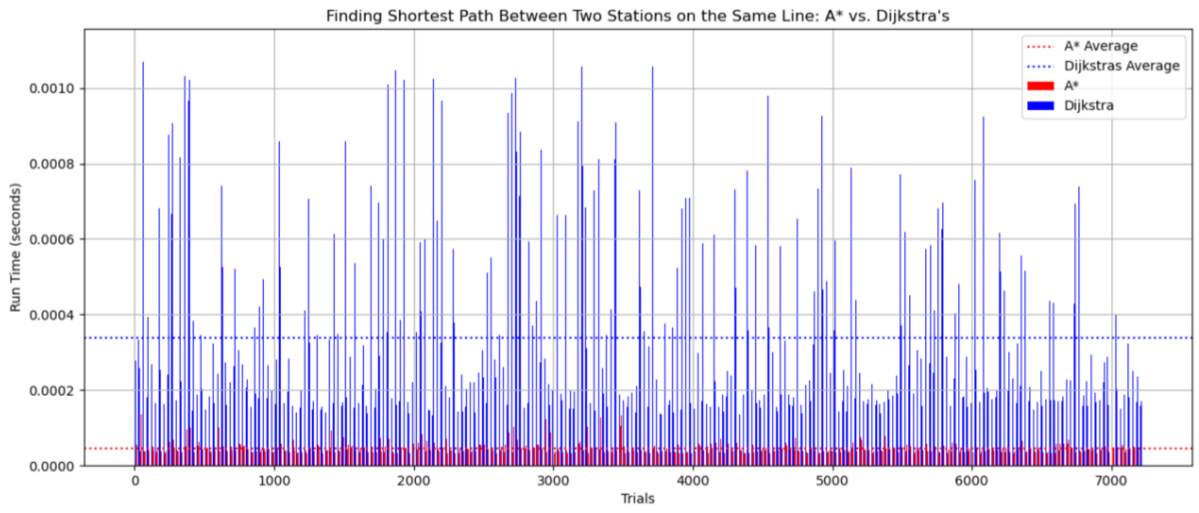

Figure 5: Finding the shortest path between two stations on the same line: A* vs. Dijkstra's

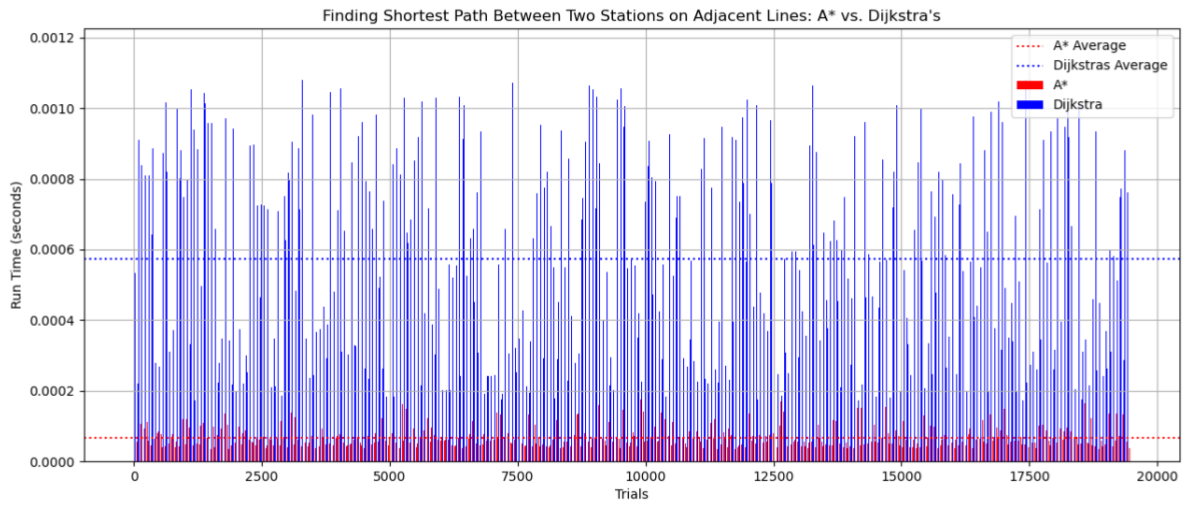

Figure 6: Finding the shortest path between two stations on adjacent lines: A* vs. Dijkstra's

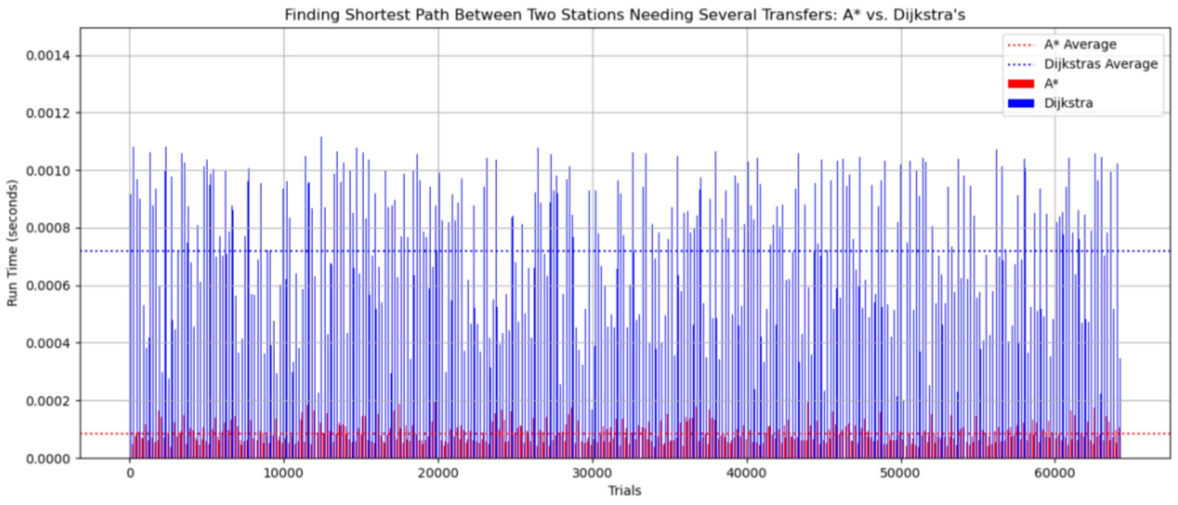

Figure 7: Finding the shortest path between two stations needing several transfers: A* vs. Dijkstra's

	All possible pairs of two stations	Two stations on the same line	Two stations on adjacent lines	Two stations needing several transfers
A*	7.79847310051876e-05	4.500678695192484e-05	6.666080480611826e-05	8.533751658915252e-05
Dijkstra's	0.0006559765944095783	0.0003389969348958078	0.0005744531834640426	0.0007187332255073945

Table 1: Average time taken to compute the shortest path between two stations by A* and Dijkstra's

3. Using the "line" information provided in the dataset, compute how many lines the shortest path uses in your results/discussion.

To compute how many lines the shortest path uses, we want to compute the shortest path between two random stations that require several transfers and count how many lines are used. We use stations requiring several transfers because otherwise, the shortest path would use either one line (for stations on the same line) or two lines (for stations on adjacent lines). We implemented this process in a function called *num_lines_experiment_part5*. This function randomly selects two different stations and runs A* on them to find the shortest path. We use A* because it is faster than Dijkstra's in finding the shortest path. Then we use the function *line_count_AStar* to count the number of lines the shortest path crosses. If it crosses more than two lines, we return the path and number of lines crossed (since we have found the shortest path between two random nodes that require several transfers). This was done five times. The results can be seen in Table 2. From these results, we conclude that the shortest path between two stations requiring several transfers use four lines on average.

Source	Destination	Shortest Path	Lines Crossed
59	195	[59, 258, 257, 12, 56, 54, 55, 245, 272, 198, 273, 229, 236, 99, 74, 287, 96, 195]	4
263	171	[263, 3, 295, 225, 155, 284, 292, 42, 120, 238, 61, 171]	4
147	111	[147, 283, 218, 193, 82, 163, 11, 28, 192, 277, 89, 40, 47, 22, 111]	6
91	79	[91, 16, 173, 98, 211, 275, 154, 153, 247, 289, 43, 79]	3
24	1	[24, 156, 13, 279, 285, 248, 273, 229, 236, 99, 74, 17, 110, 265, 1]	5

Table 2: Shortest path between two random stations (requiring several transfers)

Part 6: UML diagram.

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

Inheritance, an object-oriented design principle is being used in two different instances. Firstly, Graph is the parent class of WeightedGraph which is the parent class of HeuristicGraph. This means that WeightedGraph and HeurisitcGraph inherit all the attributes and methods of Graph. Additionally, WeightedGraph overloads the w() function as the method has the same name with different parameters. Secondly, Dijkstra, Bellman_Ford and A_Star all inherit from the abstract class SPAlgorithm, so that calc_sp() gets overwritten in all child classes.

ShortPathFinder is composed of Graph and SPAlgorithm shown by the solid diamond arrow in the UML. This allows us to instantiate any graph along with any shortest path algorithm and calculate the shortest path using these objects. This follows the strategy behavioural design pattern because we can choose to use many different shortest path algorithms dynamically using the SPAlgorithm interface.

2. How would you alter the UML diagram to accommodate various needs such as nodes being represented as 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.

You can alter this UML diagram to accommodate nodes being represented as Strings or other information by creating a node class. In this class you can define an identifier for each node as well as another attribute of whatever type you would like. The identifier can be used for uniquely identifying nodes while the other attribute holds whichever information you want some node to hold. When each node is an object instead of a primitive data type, it is much easier to alter or accommodate other needs. All other methods in the UML that have a node as a parameter would also have to be changed from type integer to Node.

Appendix

All experiments, test cases, charts and objects are instantiated/shown in the main file using the ShortPathFinder object. Each part of the assignment is clearly outlined with comments and in the title of each method. SPAlgorithm defines the functionality for Dijkstras, BellmanFord, AStarAdapted. AStar has a different return type than what is defined in SPAlgorithm. Therefore, I used the adapter design pattern through AStarAdapted to integrate AStar into the structure of the code. The Graph class outlines the functions are implemented in weighted_graph and then heuristic_graph. Short_path_finder puts all the pieces together and is how we perform a shortest path algorithm on some graph in main.