Final Project group 36

Ruizhe Li - 400435186 Chris Wang - 400496303
Efe Yencilek - 400494964

April 4, 2025

1 Setup

1.1 GitHub Link

- https://github.com/FarukEfe/pathfinder-system-london-2XC3

1.2 Setup Guide

Before running the project, please ensure you've set it up properly by running `pip install -r requirements.txt` or `pip3 install -r requirements.txt` based on your pip version. You can refer to the following commands to run their associated scripts.

- `main.py` includes a playground to test custom cases in algorithms and ensure they work. Run `main.py` by clicking the IDE run button (intuitive) or execute `python -m main` on your terminal from the project root directory.
- `part_5/part_5_experiments.py` includes the experiment code and scripts that evaluate and compare the performances of the A* and Dijkstra's. To run the file, simply execute `python -m part_5.part_5_experiment` on your terminal from the project root directory.
- `part_2_3/part_2_experiment.py` includes the experiment script associated with the section 2.3 of the project chapter 2. Run by simply executing `python -m part_2_3.part_2_experiment` from the project root directory on the terminal.

2 Team Charter (Part 1)

2.1 Communication Protocol

Communication will be handled primarily over teams and email. We expect each member to reply within 1 business day. Closer to the deadline (2 days before deadline) we expect them to answer within 3 hours. This approach will encourage each team member to own up to their responsibilities and stay in touch in case of any questions or problems occurred.

2.2 Team Penalty Policy

A reasonable penalty for repeated failure to adhere to the communication agreement could be a make-up with a treat of tim-bits to the next team meeting. This ensures that the students turn negative behavior into something positive and this will be followed by a briefing by the individual who has failed to effectively communicate. This way they'll be able to reflect and come up with actions steps that better enable them to be available for communication, and make other team members better understand their situation.

We will ensure to assign a challenging, but reasonable amount of tasks to each member. We'll also not take on one another's tasks to ensure that the workload is not dumped on just a single person, and every member takes accountability for the project.

2.3 Technology Stack

Here's a list of the software tools we'll use along with their purpose:

- GitHub: Version Control System
- VS Code: Integrated Development Environment (IDE)
- Overleaf: *latex* editor
- yEd: Editor for the UML Diagram

2.4 Team Disputes

To resolve team disputes, we would first address the issue directly and respectfully within the team by having an open discussion to understand each other's perspectives and find common ground. If the disagreement persists, we would collaboratively brainstorm solutions and compromise to reach a fair outcome. If we're unable to resolve it internally, we would seek advice from the TA to mediate or provide guidance. Only if the issue remains unresolved would we escalate it to the instructor. Throughout the process, we would maintain professionalism and focus on the project's success.

2.5 Other Considerations

We're thinking of putting heavy emphasis on making concise and descriptive commit messages. They should describe clearly what bug the commit is fixing or what feature is being added. No team member ideally pushes their code if there's bugs in it. However in the rare occasion that it may be necessary, they have to clearly state what the problem is and where it stems from.

3 Dijkstra's and Bellman-Ford (Part 2)

3.1 Dijkstra's algorithm code explanation

1. **Initialization:**
    - Create `dist_table` and `prev_table` to store the shortest distance from the source node to all other nodes, and the previous node for each node. Set the distance of the source node to 0 and its previous node to itself.
2. **Iteration Process:**
    - Perform up to k iterations, using a priority queue (min-heap) to extract nodes with the smallest distance and relax the edges of their neighboring nodes.
3. **Edge Relaxation:**

- For each neighboring node, compute the alternative distance thr -ough the current node. If it's smaller than the current shortest distance, update `dist_table` and `prev_table`, and push the updated node back into the priority queue.

3.2 Bellman-Ford algorithm code explanation

1. **Initialization:**

   - Create `distances` and `prev` dictionaries to store the shortest distance from the source node to all other nodes, and the previous node for each node. Set the distance of the source node to 0 and all others to infinity. Initialize `prev` for all nodes to -1.

2. **Relaxation Process:**

   - Perform edge relaxation up to k times. In each iteration, copy the current `distances` to `new_distances` to avoid interference during updates. For each edge (u, v) in the graph, check if the path through u provides a shorter distance to v. If so, update `new_distances[v]` and set `prev[v]` to u.

3. **Final Update:**

   - After relaxing all edges up to k times, update `distances` with the values in `new_distances`. Return the final `distances` and `prev` dictionaries.

3.3 Performance analysis

3.3.1 Time Cost: Density Fixed with proper k

We control the density to 50% and 100% of the graph by using an appropriate value of k, where k is selected from the 50% range at center between 1 and the total number of nodes. Then, using the number of nodes as the x-axis and the run time as the y-axis, we generate a line chart of the time cost.

From Figures 1 and 2, it can be observed that, while keeping the density constant, the time cost and growth rate of Dijkstra's algorithm are much smaller than those of the Bellman-Ford algorithm. Moreover, by comparing

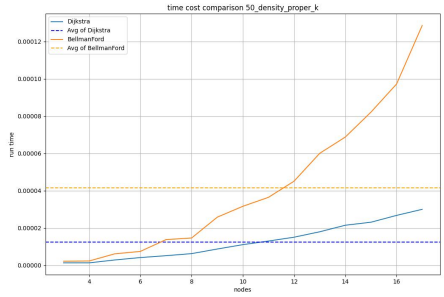

Figure 1: 50% density

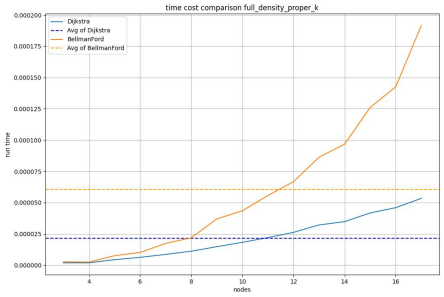

Figure 2: 100% density

the two graphs, we can conclude that, whether using Dijkstra or Bellman-Ford, as the density increases, the time cost also increases.

3.3.2 Time Cost: Edges Fixed with proper k

We control the number of edges to be 20, and use an appropriate value of k, where k is selected from the 50% range at center between 1 and the total number of nodes. Then, using the number of nodes as the x-axis and the run time as the y-axis, we generate a line chart of the time cost.

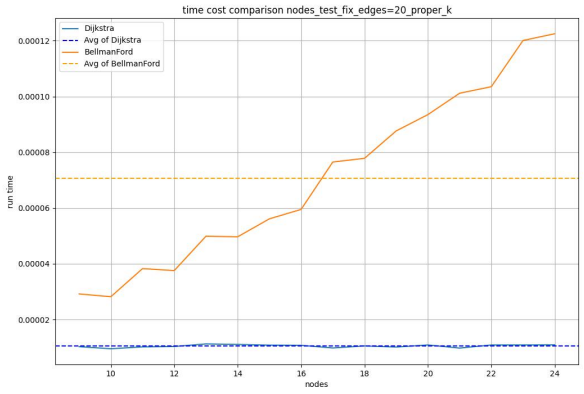

Figure 3: Fixed edges=20

From Figure 3, it can be observed that when the number of edges is fixed at 20, as the number of nodes increases, the time cost of the Bellman-Ford algorithm grows linearly, while the time cost of Dijkstra's algorithm remains almost unchanged.

3.3.3 Time Cost: Nodes Fixed with proper k

We control the number of nodes to be 10, and use an appropriate value of k, where k is selected from the 50% range between 1 and the total number of nodes. Then, using the number of edges as the x-axis and the run time as the y-axis, we generate a line chart of the time cost.

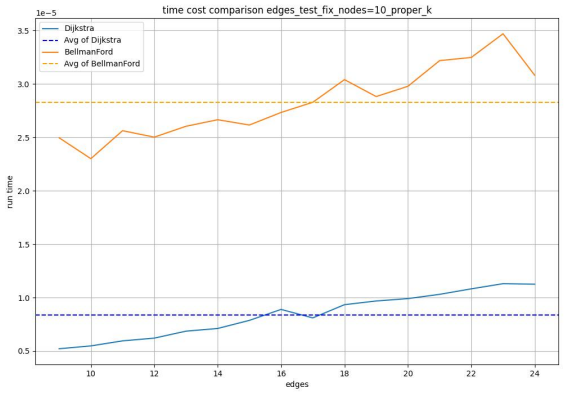

Figure 4: Fixed nodes=10

From the observation of Figure 4, when the number of nodes is fixed at 10, as the number of edges increases (i.e., the graph's space remains unchanged and the density increases), the time cost of both Dijkstra and Bellman-Ford algorithms rises gradually. Additionally, it can be directly observed that the time cost of Bellman-Ford is higher than that of Dijkstra by a noticeable margin.

3.3.4 Time Cost: k Test Node & Edge Fixed

We control the number of nodes and edges in the graph, and then test the relationship between k (ranging from 1 to the number of nodes) and the time cost. The k value is used as the x-axis, and the run time is used as the y-axis to generate a line chart of the time cost.

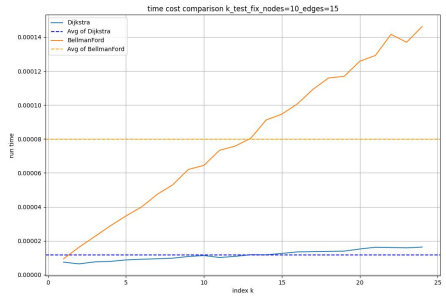

Figure 5: fixed nodes=10 edges=15

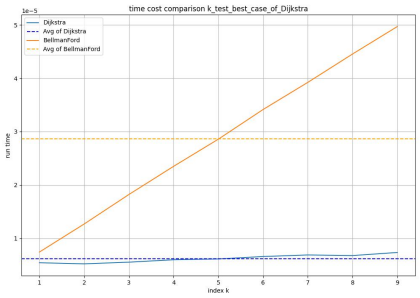

Figure 6: best case of Dijkstra

Based on Figures 5 and 6, it can be observed that, whether in the best case or average case, as the value of k increases, the time cost of the Bellman-Ford algorithm increases with k. In contrast, the time cost of Dijkstra increases only slightly.

3.3.5 Space Cost: Density Test & k Test

First, we create a line chart of space cost based on the changes in space cost when the edge density in the graph increases while the number of nodes remains unchanged. Next, we create another line chart of space cost, keeping the number of nodes and edges the same, and observe the relationship between k and time cost.

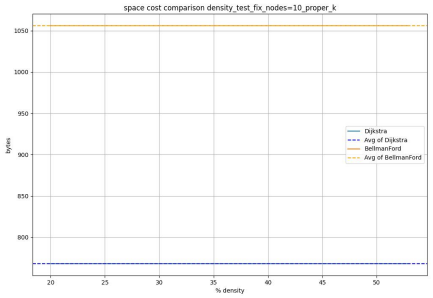

Figure 7: fixed nodes=10 proper k

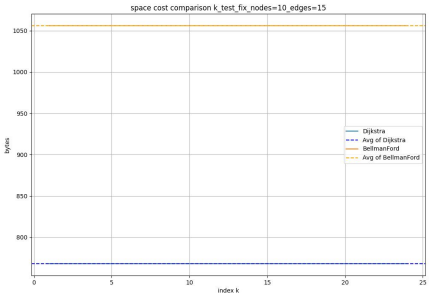

Figure 8: fixed nodes=10 edges=15

Based on Figures 7 and 8, we find that the relationship between space cost and density, as well as between space cost and k value, is almost nonex-

istent. However, we can easily observe that the space cost of Bellman-Ford is significantly higher than that of Dijkstra.

3.3.6 Space Cost: Density Fixed with proper k

We control the density to 50% of the graph by using an appropriate value of k, where k is selected from the 50% range at center between 1 and the total number of nodes. Then, using the number of nodes as the x-axis and the space cost as the y-axis, we generate a line chart of the time cost.

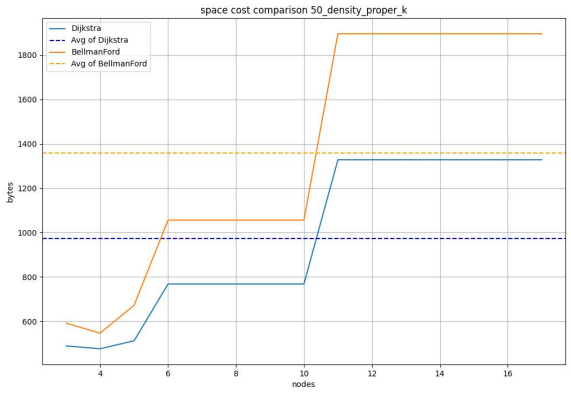

Figure 9: 50% density

Based on Figure 9, we observe that when the graph's density remains constant, the space cost grows quadratically as the number of nodes increases. Additionally, the space cost of Bellman-Ford is still greater than that of Dijkstra's.

3.3.7 Summary

The observations can be summarized as follows: As the number of nodes increases, with the graph density held constant, the space cost grows quadratically, and the space cost of the Bellman-Ford algorithm is consistently higher than that of Dijkstra. There is little to no correlation between space cost and graph density or k value, but Bellman-Ford's space cost remains significantly higher than Dijkstra's. As k increases, the time cost of Bellman-Ford increases noticeably, while Dijkstra's time cost rises only slightly. With a fixed

number of nodes, as the number of edges increases, both algorithms' time costs rise, with Bellman-Ford's time cost being higher. Overall, Bellman-Ford has significantly higher time and space costs compared to Dijkstra.

4 Floyd-Warshall Algorithm (Part 3)

The **Floyd-Warshall algorithm** is a dynamic programming technique used to solve the *All-Pairs Shortest Path (APSP)* problem in a weighted graph. It computes the shortest path distances between every pair of vertices by iteratively improving the path using intermediate nodes. The key idea is that for each pair of nodes (i, j), the algorithm checks whether including an intermediate node k can provide a shorter path:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$$

The Floyd-Warshall algorithm works for graphs with both positive and negative edge weights, as long as there are no negative weight cycles. Its time complexity is $\Theta(V^3)$, and its space complexity is $\Theta(V^2)$, where V is the number of vertices.

Comparative Complexity Analysis:

Dijkstra's algorithm is a single-source shortest path algorithm with a complexity of $\Theta(E+V \log V)$ when implemented with a priority queue (heap), or $\Theta(V^2)$ in the case of dense graphs where $E = \Theta(V^2)$. Bellman-Ford, another single-source shortest path algorithm, has a complexity of $\Theta(VE)$, which becomes $\Theta(V^3)$ in dense graphs.

To solve the all-pairs shortest path problem, we must run each of these single-source algorithms once for every vertex:

In a dense graph, the number of edges E approaches the maximum possible, which is:

$$E = \Theta(V^2)$$

- Floyd-Warshall for all vertices: $\Theta(V^3)$ for dense graphs, because there are three nested for loops.
- Bellman-Ford for all vertices: $\Theta(V \times VE) \rightarrow \Theta(V^4)$ for dense graphs.

Conclusion: Given the above analysis, we conclude that the Floyd-Warshall algorithm, which solves APSP directly in $\Theta(V^3)$ time regardless of graph density, is the most suitable algorithm for dense graphs. It is not only simpler to implement but also supports negative edge weights, making it a more efficient and robust choice compared to repeatedly applying Dijkstra or Bellman-Ford.

5 Implementation and Inquiries on the A* Algorithm (Part 4)

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

Dijkstra's algorithm is relaxing all the edges in the graph, even the ones that may have nothing to do with the path we're trying to compute. This is not necessarily a problem for single source or all-pair pathfinding methods, but for a determined source and destination, we waste the computer resources doing irrelevant computations.

A* is as if putting a rat in a maze and a cheese at the destination. The rat will then follow the paths where the cheese smell is stronger. In the same way, A* defines a heuristic on the neighbor nodes, determining their likelihood to unveil a cheaper path to destination. This way, a big chunk of the irrelevant edges will be low in priority, and thus we'll ideally find the destination well before having to check those.

In summary, A* offers an alternative to Dijkstra's algorithm that implements dynamic programming principles to make significant performance increases to the shortest path problem.

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

We would have to compare the algorithms based on two metrics; time complexity, and optimality of the solution. I would randomly generate a source and destination within the graph nodes for N iterations, then I would run both algorithms and measure their runtime independently.

Assuming that both algorithms provide a valid path to destination, I would sample a random graph N times and run the pathfinding algorithms on the same graph. Then, I would compute the mean for

both measurement lists, allowing me to compare the average runtime difference between the two algorithms.

Given that the paths may be longer or shorter based on the proximity of the source and destination nodes, so I would make sure to sample from many different lengths by taking $N \geq 100$, given that there's around 320 nodes in the provided dataset.

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

The whole point of the heuristic function is to have an approximate relation between a node and destination, allowing us to have a rough distinction between relevant and irrelevant nodes. The better the approximation the better the distinction.

However, if we randomize the heuristic function, the algorithm efficiency would be roughly indifferent from Dijkstra's (in the general case) since the heuristic values would not have any correlation with the destination node.

In the case of a completely irrelevant (or opposite) heuristic generation, the compute will consistently match the worst-case complexity of Dijkstra's algorithm.

d. What applications would you use A* instead of Dijkstra's?

A* is extremely popular in source-to-destination pathfinding applications. This includes many fields such as robotics, non-playable characters in video games, and navigation applications due to its optimality and well resource management. It is also more easily pairable with machine learning applications where the heuristic could be optimized as a model.

In the above listed use cases, Dijkstra's algorithm would remain slow and inflexible. Also, A* particularly remains very efficient in navigation-based applications since the good heuristic choice becomes more important in denser graphs where there's more choice for edge traversal.

6 Performance Analysis of the A* and Dijkstra's Algorithm (Part 5)

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

The purpose of using A* is to make better decisions and relax the edges that have a higher chance of being shorter. Thus, A* outperforms Dijkstra's Algorithm especially in denser graphs, where a heuristic becomes more critical to making good path choices. To prove the efficiency of A* over Dijkstra's, we plotted the runtime results in as follows:

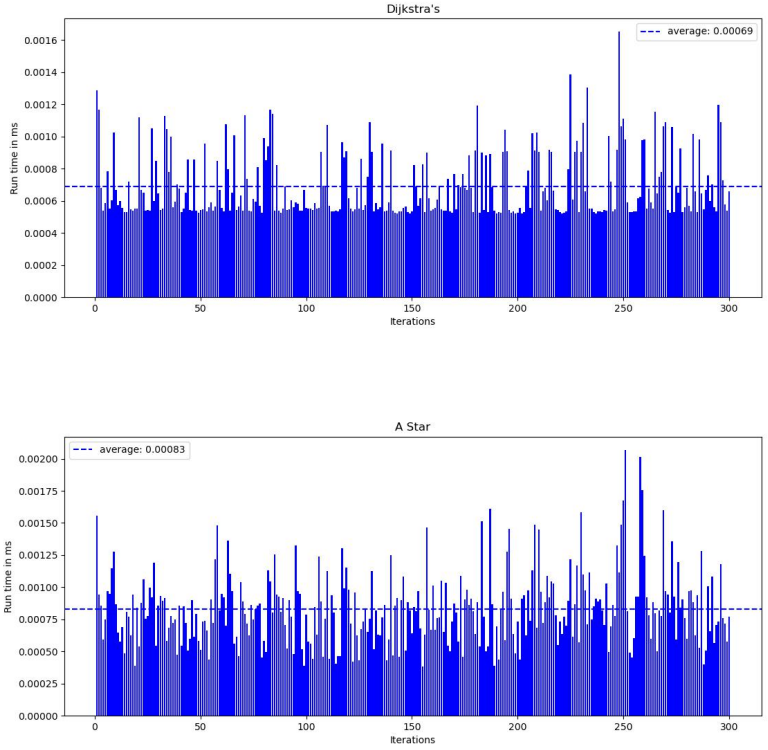

Figure 10: Dijkstra's vs. A* on London Dataset

When we ran the above experiment on the **London Dataset**, we've

seen that Dijkstra's Algorithm still outperforms A* by a notable amount. As seen in the plot, Dijkstra's algorithm performs around $0.00069ms$ while A* was around $0.0083ms$, which is more than Dijkstra's algorithm.

We have developed a theory as to why this may be the case. In the provided London data set, the connections between the stations (the edges) are not dense enough for A * to outperform Dijkstra's, therefore making a good choice of path with heuristic becomes marginally less beneficial compared to Dijkstra's relaxation of each neighbor.

If this theory is correct, then A* should be much faster than Dijkstra's in graphs that get denser than the given dataset. How do we test this theory?

To test this theory, we have written a random graph generator that takes number of nodes, and number of edges as input. We have also written a random heuristic generator that generates latitude and longitude values within a provided range. One of our doubts about this experiment was that, since the generated information is random and without context, the graph and the heuristic might not represent a practical basis for algorithmic comparison and therefore still favor Dijkstra over A *. However, running the experiment gave us promising results.

Keeping the number of nodes to be $n = 100$, we ran a density-based experiment with edge sizes varying from 10 to 10000, and here's the result:

From the experiment results, we can draw some important conclusions:

- A* performs significantly better than Dijkstra's at higher graph densities.
- Dijkstra's time cost scales up linear to the graph density, where A* time cost scales up logarithmically.

Running this experiment has solidified our theory, which states that A* is particularly much more efficient than Dijkstra's especially in denser graphs.

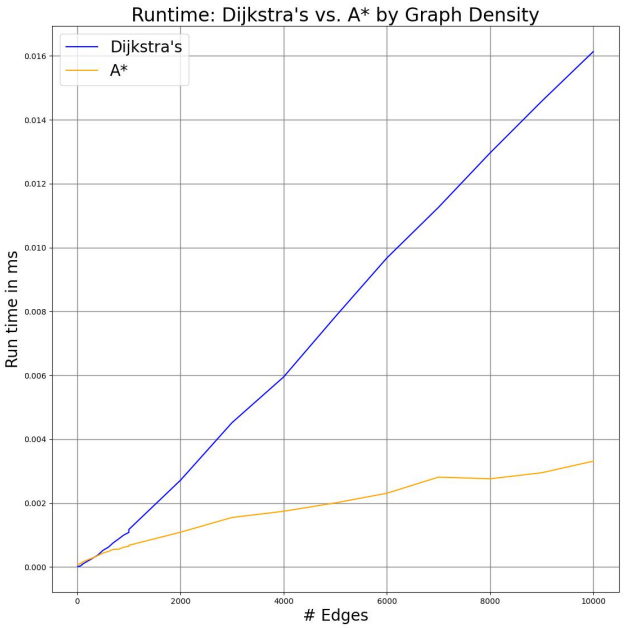

Figure 11: A* & Dijkstra Performance by Density

b. What do you observe about stations which are (1) on the same lines (2) on the adjacent lines (3) require several line transfers?

To compare the runtimes of source-destination pairs that are (1) on the same lines, (2) on the adjacent lines, (3) distinct lines with multiple transfers, we decided to run an experiment on the **London Dataset**. We have first written a piece of code that would populate source-destination pairs into the three categories. Then we have tested each source-destination pair on both Dijkstra's and A* algorithms to see which one of the 3 groups would be faster computed by the Algorithms. The experiment can be described in 3 steps:

- Step 1: Select random pairs of source and destination nodes, and compute the shortest path between them
- Step 2: Label the shortest path into one of the 3 described categories, done by computing the number of distinct lines traversed at each edge.

- Step 3: After examples are populated for each 3 categories, reduce the number of examples to the length of the category with the minimum examples. This ensures all categories have equal number of examples.
- Step 4: Run A* and Dijkstra's algorithm on each of the populated source-destination categories, and plot the results.

After running this experiment, the resulting graphs are as follows:

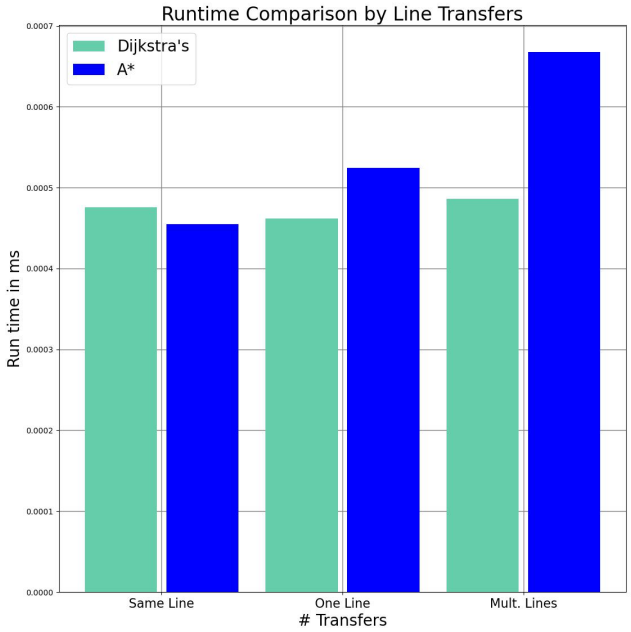

Figure 12: A* & Dijkstra's Performance by Req. Line Transfers

Some valuable observations:

- Runtime performance of Dijkstra's Algorithm remains roughly the same across different number of required line transfers between source and destination.
- Runtime cost of the A* Algorithm increases as more line transfers are needed between source and destination.

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

The number of line transfers depends on the shortest path between source and destination, determined by the algorithm. To compute different lines of transfers of the shortest path both for A* and Dijkstra's algorithm, We've made a method called `test_lines()` inside the `Tests` class located in `part_5/part_5_experiments.py`.

This method gets the edges on the shortest path and retrieves the line on which they locate. Doing this for each edge and reducing the list to all the distinct values, the method can compute the number of line transfers found on the shortest path.

This method is also used in the above experiment, and provides a valid algorithm that computes how many lines the shortest path uses.

7 UML Diagram and Implications (Part 6)

Design Patterns and Principles in the UML Diagram

The UML diagram applies several useful object-oriented design principles to make the system more flexible and easier to extend. One clear example is the **Strategy Pattern**. The `SPAlgorithm` interface defines a shared structure for different shortest path algorithms, including `Dijkstra`, `Bellman_Ford`, and `A_Star`. This setup allows the main class (`Short Path Finder`) to choose and use different algorithms without changing its own internal logic.

The design also follows the **Open-Closed Principle**, meaning it is open to extension but closed to modification. For instance, if a new algorithm needs to be added, it can be done by creating a new subclass of SPAlgorithm without editing any of the existing code in `Short Path Finder`. This makes the system much easier to update or maintain.

We also see the **Adapter Pattern** used in the `A_Star` class. Since A* uses additional information like heuristics, it doesn't behave exactly like the other algorithms. But by ensuring it still fits the same interface as the others, we can plug it into the rest of the system with no major changes.

In short, the design is clean, extendable, and easy to work with—qualities that are very helpful in any project involving algorithms and multiple moving parts.

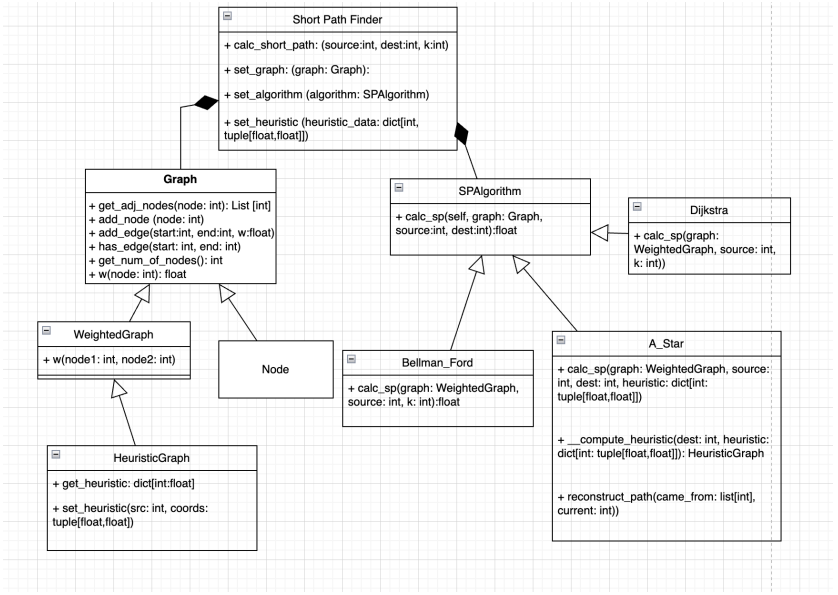

Figure 13: A* & Dijkstra Performance by Density

Notice that Figure 13 (above UML diagram) shows `Node` as a separate class instead of integer value. This is because, ideally, you would want the nodes of the graph to be able to support various data types to allow more flexibility in the program. This way, the graph can be compatible with a wider range of applications that don't necessarily use integers to store values at locations.

To fit this need, the `Node` class would consist of

- **a key** (could also be a comparable data type)
- **a value** (either a built in type, or custom data type/structure).

After this final tweak, the graph and its associated algorithms could be used in a wider pool of applications suited to support various data types as nodes.

8 Appendix: Code Navigation

Below are the file directories you should be able to run from the terminal as specified. (**Make sure to first install the requirements as specified in** `README.md`)

- `main.py` includes a playground to test custom cases in algortihms and ensure they work. Run 'main.py' by clicking the IDE run button (intuitive) or execute 'python -m main' on your terminal from the project root directory.

- `part_5/part_5_experiments.py` includes the experiment code and scripts that evaluate and compare the performances of the A* and Dijkstra's. To run the file, simply execute 'python -m part_5.part_5_experiment' on your terminal from the project root directory.

- `part_2_3/part_2_experiment.py` includes the experiment script associated with the section 2.3 of the project chapter 2. Run by simply executing 'python -m part_2_3.part_2_experiment' from the project root directory on the terminal.

Here are where some of the essential files are located:

- `Algorithms/..` folder is where the essential A*, Dijkstra's, Bellman-Ford algorithms are located. The Floyd-Warshall algorithm for the all-pairs path finding is also located here.

- `Dataset/..` folder is where the **London Dataset** files are located.

- `DataLoader.py` is the file that contains the module responsible for reading and filtering the `.csv` data from the tables.

- `Graphs.py` is where the graph and heuristic data structures are located.

- `ShortPathFinder.py` is the intermediary module that mitigates the shortest path computation calls.