

Tokyo Metro System Optimization

Contributors: Shivalika Gupta, Kirin Kambon

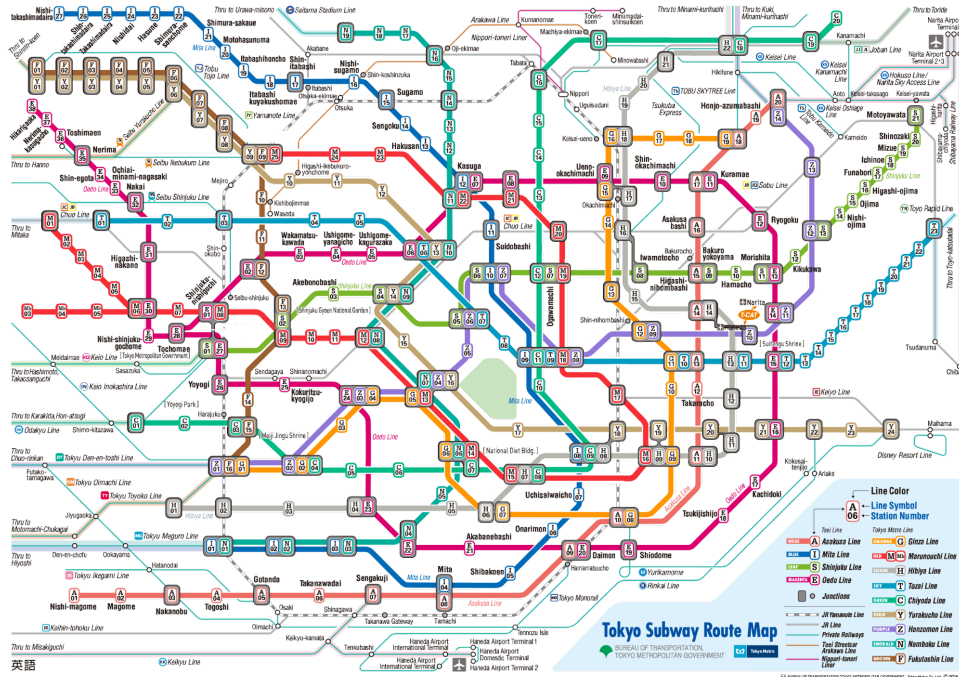


Figure 1: Map of Tokyo Metro System

The Tokyo Metro System is very large and complicated as shown in Figure 1. It is difficult to navigate it, especially if it is your first time using it. With our knowledge of path planning, we decided to optimize the path for users based on their preferences. We created a graphical interface where the user could input their starting station, their ending station, and whether they would like to optimize the distance traveled, the number of stops, and/ or the number of transfers. Our algorithm will then calculate a path based on this information and output the stops along this path. It will also highlight the corresponding stations on a map of Tokyo in black and output the statistics of the path. This includes the number of stops, the number of transfers, and the total distance traveled in kilometers. It will also determine if adding a connection between two of the stations would further optimize the distance. If this is true, it will output this information and show the percentage of how much the distance would be improved.

In order to create this algorithm, we first searched for a dataset that would provide detailed information in order to execute the path planning algorithm. After extensive research, we were able to find a github with data that included the connections, the distance, and the

station name/ station IDs [3]. The downside of this dataset is that it does not contain time information as time is real-time data that varies based on various conditions on different days. Hence, we could not also optimize based on the travel times between stations. This is why we decided that optimizing on stops and transfers along with distance was important as these are important factors in travel time. We also had to combine this dataset with a dataset of the station names and their corresponding longitude and latitude [5]. We preprocessed these datasets in order to extract only the useful information using python scripts and the Pandas library. We then created a python script to iterate through both datasets and match stations from the first dataset to stations in the second dataset, so that we could have one centralized dictionary with all of the required information. This matching was done according to the station “id” which consists of a letter referring to the line a station belongs to and a number indicating which station along the line our current station is. We created a dictionary to store the connections along with the corresponding distance between the two stations. We also created a dictionary for the longitude and latitude data to make it easier to calculate distances.

There were many challenges we faced with the datasets. After doing some testing, we realized that portions of the data containing the connections were incorrect. For instance, some stations have multiple corresponding IDs, since one station connects multiple lines, and some of these IDs were incorrect on the connection mapping. As a result, we had to go through a map of the Tokyo Metro System and correct the IDs manually. This was a time consuming and tedious process, but we felt that it was necessary to do this, since we were unable to find another dataset containing the connections between the stations. Furthermore, some of the longitude and latitude information of the stations were missing, so we had to manually add these in by researching these stations. While this process was tedious it was the most concrete way of ensuring that our data is 100% accurate as we are manually checking it over, since any inconsistencies in our data can cause our generated path to be suboptimal.

In addition to the connection data and missing location data, we also realized that there were inconsistencies in our dataset for the existing location data. Particularly, some of the locations of the stations were from different stations with the same names from various other cities in Japan. Therefore, we ran a script that iterates through all the stations line by line and searched for large jumps or abnormalities within a line (any increase of more than 22 km from one station to another in one line was flagged). A list of suspicious stations was printed out by the algorithm and we manually went through the list to fix the offending location data for each of

the stations. This was necessary as our algorithm would sometimes make strange decisions when pathfinding, which we found was induced from the errors of our dataset. These errors would sometimes cause the priority of a successor station to be abnormally high (one of the train stations had location data from a matching named station in Osaka instead of Tokyo) causing it to never be picked in the path even if it was closer to the goal state.

Once we found the datasets, we considered various Path Planning algorithms, which include Dijkstra and the A* Search algorithm. Since our dataset is finite, both of these algorithms would work for our goal. One important thing that we had to consider is which algorithm would produce faster results for users, especially since the dataset is very large. For this, the A* Search algorithm seemed to reduce the cost since Dijkstra requires looking at every station in our dataset. Additionally, the A* Search algorithm would allow us to experiment with various heuristic functions and apply them based on the user's preferences.

To ensure that our decision to choose the A* Search Algorithm over Dijkstra's Algorithm was better for our goal, we implemented both and ran a series of tests. We checked how long it took for the algorithms to run, the overall distance, the number of stops, and the number of transfers. One such example is finding the path from station A01, Nishi-Magome, to N10, Iidabashi. As a result, Dijkstra took 1993400 nanoseconds to run whereas the A* Search Algorithm took 1000600 nanoseconds to run. Clearly, A* Search is about 2 times faster than Dijkstra's algorithm in this example. Another thing we noticed was that Dijkstra's algorithm produced a path with distance 13.8 kilometers whereas A* Search produced a path with a distance of 14.7 kilometers. Although Dijkstra's algorithm produced a path with shorter distance, the A* Search Algorithm was also able to optimize on stops and transfers too. Specifically, the A* Search Algorithm's path is ['A01', 'A02', 'A03', 'A04', 'A05', 'N02', 'N03', 'N04', 'N05', 'N06', 'N07', 'N08', 'N09', 'N10'], which contains 14 stops and one transfer. On the other hand, Dijkstra's path is ['A01', 'A02', 'A03', 'A04', 'A05', 'A06', 'A07', 'A08', 'I04', 'I05', 'I06', 'I07', 'I08', 'Y18', 'I09', 'I10', 'Z07', 'Z06', 'T07', 'T06', 'N10'], which contains 21 stops and 6 transfers. The number of transfers and stops is very important for our goal because we are allowing our users to optimize this along with the distance, since in the real world, stops and transfers are very time consuming factors.

Generally, the difference between the two algorithms is not extremely apparent in a dataset of our size, specifically the region of Tokyo within Japan. However, we believe that if our

scope was widened to include other neighboring cities or even the country of Japan as a whole we would begin to see more of a benefit with A* search over Dijkstra's algorithm. We believe this is due to the fact that "Dijkstra and A* almost have the same performance when using it to solve town or regional scale maps, but A* is better when using it to solve a large scale map.[2]" This makes sense as our algorithm is meant to decrease the number of total stations searched when looking for the optimal path which starts to see more of an effect as the size of the search space increases. Additionally, A* has easy support for modularity in that we can create our heuristic in many different ways to fit what the user wants to optimize. The algorithm doesn't change if we are trying to minimize transfers instead of stations or distance, we would just use a different heuristic that weights transfers higher than non transfer stations. However, with Dijkstra's we would have to hardcode the weights based on the users preferences, which would be inefficient and less versatile.

In the algorithm, the stations represent the nodes and the connections between them represent the edges. Each edge has a weight corresponding to the distance and whether it is a transfer based on what the user chooses to optimize. At each step of our algorithm, a successor node is picked according to a value $f(n)$, which is a function of $g(n)$ and $h(n)$. $g(n)$ is the cost of moving from the starting station to the current station n on the graph following the path generated to get there. $h(n)$, also known as the heuristic function, is the estimated cost to move from the current station n to the final goal station. We add these two values in order to obtain $f(n)$, which allows us to pick the station that has the least overall cost estimate.

If the heuristic function, $h(n)$, were to be 0, our algorithm would be Dijkstra's algorithm and it would not be as time efficient as the algorithm would have a larger search space. This is why it is important to incorporate this value into the $f(n)$ function. It was also important to ensure that the heuristic function is always lower than or equal to the cost of moving from station n to the goal station. The more accurately we estimate the heuristic, the faster the algorithm would be as the search space will be smaller.

We split up our heuristic into three parts: distance, stops, and transfers. For the distance, we considered the euclidean distance, diagonal distance, and the manhattan distance. The euclidean distance is simply the distance between the current station and the goal station. The Manhattan distance is the sum of absolute values of differences in the goal station's x and y coordinates and the current station's x and y coordinates respectively. The diagonal distance is

the maximum of the absolute values of differences in the goal station's x and y coordinates and the current station's x and y coordinates respectively. The issue with the Manhattan distance was that there was a possibility of it overestimating the distance that was left, which would cause the result of our algorithm to be inaccurate. We decided to choose to implement the euclidean distance over the diagonal distance because we believed that the euclidean distance more accurately estimated the distance that was left from the current station to the goal station, since it incorporates both the horizontal and vertical positions whereas diagonal only incorporates one of the two directions

For the stop heuristic, we considered two options. The first option was to calculate the number of stops we have crossed so far in our path over the total distance traveled times the estimated euclidean distance left to travel. The other option was to estimate the number of stops per kilometer by calculating the total number of stops on the map divided by the total distance in kilometers on the map between all stations. With this number, we calculated the number of stops left by multiplying it by the estimated distance left to the goal station. We ended up choosing the second option because we felt that the first option was more likely to overestimate the number of stops if there was a concentration of stops in the beginning of the path.

For the transfers heuristic, we added an additional weight if the next stop that was being considered was a transfer. This way, the algorithm would be less likely to choose the stop if it was a transfer. However, while testing we realized that the algorithm was not changing much when optimizing the distance versus the transfers. Initially, we added a weight of 2 to a transfer, but we increased this to 5, so that the algorithm would be more likely to find a path with less transfers.

Based on what the user would like to optimize, we considered the corresponding heuristics. If the user wanted to optimize multiple things, the heuristics would be added together to produce $h(n)$. The cost from the starting station to the current station, $g(n)$, would also be updated according to what the user is trying to optimize. If it is multiple parameters that are being optimized, $g(n)$ would be the sum of these weights correspondingly. As a result, the function to choose the next node in the path, $f(n)$, would be optimized to fit the user's preferences.

An additional component that our algorithm considers is whether adding a new connection between two stations further optimizes the path. In order to do this, we created a new hashmap with our data containing the original connections and new added connections. Since the dataset we obtained did not contain information about the most traveled paths, we generated hypothetical data and decided which paths could benefit from a new connection. Our code runs the A* search algorithm on this new dataset along with the old dataset and then compares the distance of the two paths. If the path generated from the new dataset is better, we know that adding a new connection is beneficial. We then calculate the percentage that the distance is improved by and this is shown to the user.

To present the results of our algorithm to the user, we decided to create a website. For this we had to use Flask to connect our backend components with our frontend components. Our backend components were coded in python, while the frontend components were coded in HTML, CSS, and JavaScript.

Initially, the user is presented with a form to input their preferences as shown on figure 2. This includes the starting station, the ending station, and whether they want to optimize the distance, the number of stops, and/ or the number of transfers. Then the user can click on the “Find a route” button which causes all the calculations to take place.

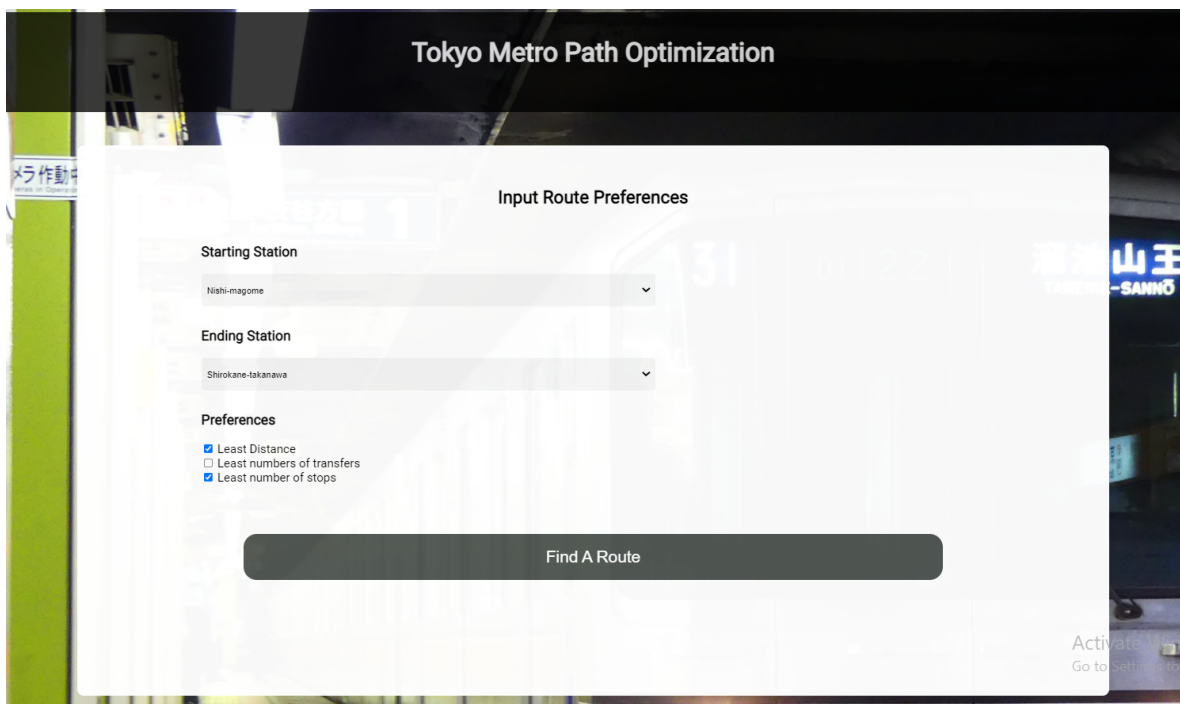
The image shows a web browser displaying a form titled "Tokyo Metro Path Optimization". The form is titled "Input Route Preferences" and contains three main sections. The first section, "Starting Station", has a dropdown menu with "Nishi-magome" selected. The second section, "Ending Station", has a dropdown menu with "Shirokane-takanawa" selected. The third section, "Preferences", contains three checkboxes: "Least Distance" (checked), "Least numbers of transfers" (unchecked), and "Least number of stops" (checked). At the bottom of the form is a large, dark grey button labeled "Find A Route". The background of the website is a blurred image of a Tokyo Metro station platform with a green and white striped pillar on the left and a blue sign with white text on the right.

Figure 2: Form presented to users on our website

After the user clicks on the button, they will be redirected to another page which will show the results of the algorithm. The first thing that the user will see is the parameters that the algorithm is trying to reduce on as shown in figure 3.

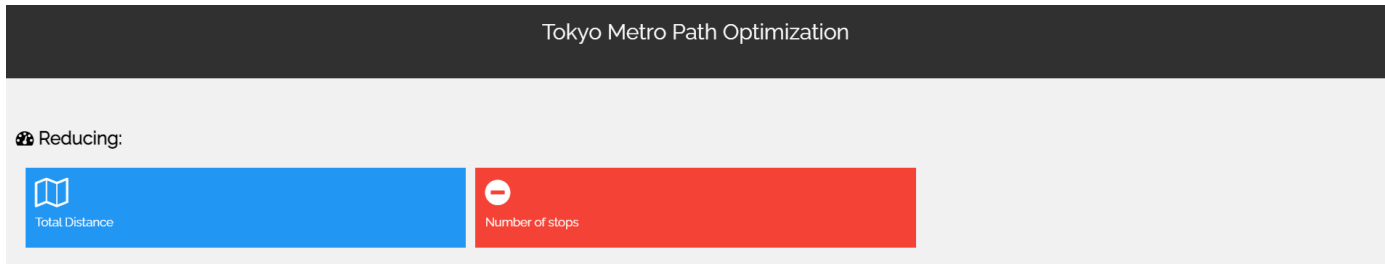


Figure 3: Portion of website showing what parameters are being optimized

The next thing that the user is presented with is the map of Tokyo with the stations as nodes. The nodes are all colored in relation to the line that the station belongs to. This is seen by the legend that is above the map. The stations along the path are colored in black, have a higher opacity, and a larger radius, so that it is more visible to the user. This map was created using Javascript's d3 and topojson packages and a dataset we found on github [4]. Alongside this map, the user is also presented with a table of the station IDs and the corresponding station names that are along the calculated path. This visualization can be seen on Figure 4.

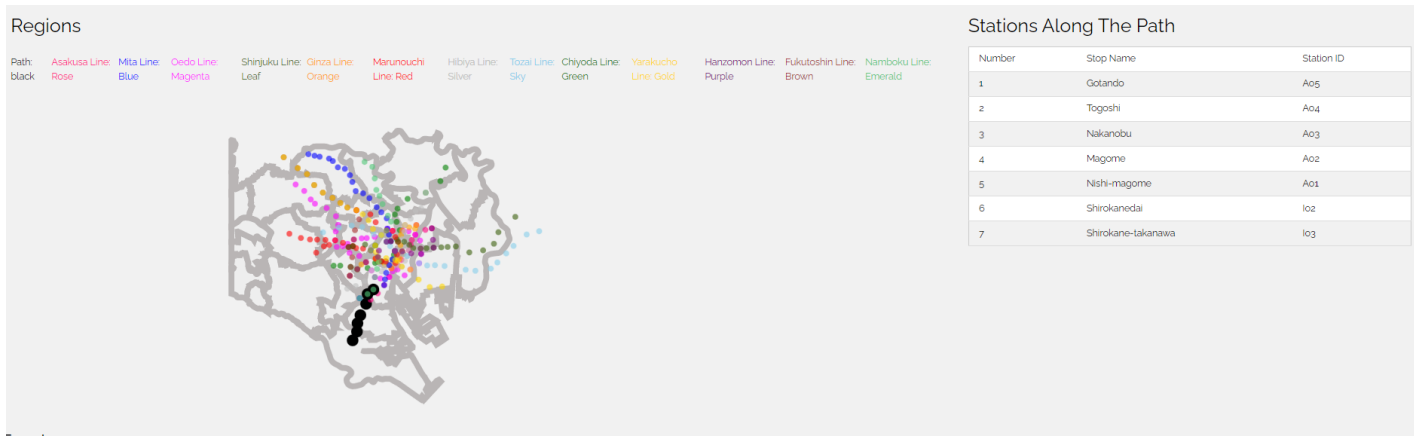


Figure 4: Map of Tokyo and Stations Table

Below the map, the user is presented with the statistics of the path including information on the total distance, the number of stops, the number of transfers, and whether a new connection should be added. An example of this can be seen in Figure 5 shown below.

Feeds		
	Total Distance:	7.010000000000001
	Number of stops:	7
	Number of transfers:	1
	Add a new connection:	Yes

Figure 5: The Table containing the Statistics of the Path

If the path added a new connection between stations to further optimize the distance traveled, the user will be presented with a bar that shows the percentage by which the distance has decreased. Additionally, the user can then click on the new path button at the bottom of the page to go back to form and input two new stations or to update the preferences on their current path. Figure 6 shows the visualization of the bar that shows the statistics of adding a new connection and the button the user can click on to take them back to the form.

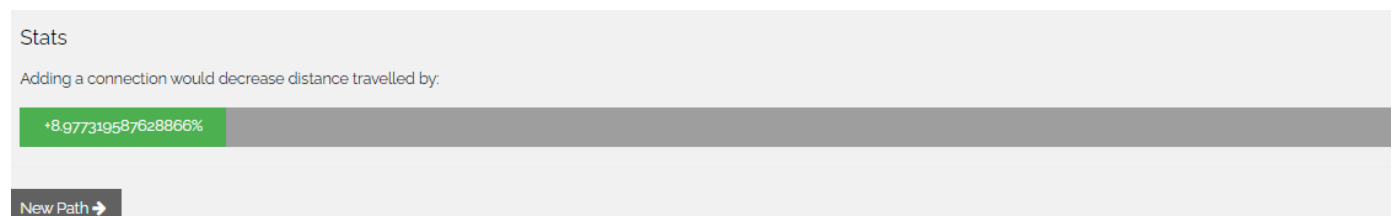


Figure 6: Statistics about the new Connection

Below we have listed examples that demonstrate the correctness of our algorithm in optimizing either distance, number of transfers, number of stops, or a combination of the three. These examples were generated using a test function that runs our algorithm on all possible paths with different combinations of parameters on which metric we are trying to optimize: distance, transfers, or stops. The syntax of the following examples compares running the algorithm optimizing the first bolded metrics versus running the algorithm optimizing the second bolded metrics. Then comparing the results to see if the two runs actually made a difference in the metrics that were meant to be optimized.

Optimizing number of **stops** vs Optimizing number of **transfers**

In the example pathway from H01 (Naka-meguro) and T15 (Minami-sunamachi), we can see that our stop-optimizing algorithm chose a path that had a stop count of 17 stops vs the 18 stops for the second path. Additionally, our transfer-optimizing algorithm chose a path that had a transfer count of 1 vs the 2 transfers in the first path. Since distance was not optimized, we also had two different paths of the same distance (16.1 km)

Optimizing number of **stops** vs Optimizing number of **transfers and stops**

In the pathway from C01 (Yoyogi-Uehara) to G04 (Aoyama-ichtome), we can see that our stop-optimizing algorithm chose a path that had the same minimized stop count as the second path (7 stops). However, since our second algorithm also was optimizing transfers, we have a lower transfer count of 1 compared to the first path which took 2 transfers. Again, since distance was not optimized, both paths had a distance of 4.5 km.

Optimizing number of **stops** vs optimizing **distance and stops**

In the path from A01 (Nishi-magome) to E05 (Ushigome-kagurazaka) the number of stops was equally optimized at 20 stops, but the algorithm that optimized distance was able to route a path of distance 15.899 km vs the first path which had a distance of 17.2 km. Transfers were not optimized for either path and both took 3 transfers to complete.

Optimizing number of **stops** vs optimizing **distance, transfers, and stops**

The route from A01 (Nishi-magome) to S11 (Morishita) both have optimal stop counts of 18, however since the second path is also optimizing distance and transfers, it was able to route a path that spans 15.9 km instead of 16.8 km and 1 transfer instead of 2.

Optimizing number of **transfers** vs optimizing number of **transfers and stops**

In the pathway from A12 (Takaracho) to C10 (Nijubashime), we can see that our distance is both unoptimized at 2.3 km, and both paths return an optimized transfer count of 2 transfers. However, since the second algorithm is also optimizing the number of stops it was able to route a path that takes 6 stops instead of 7.

Optimizing number of **transfers** vs optimizing **distance and transfers**

The route from A01 (Nishi-magome) to G12 (Mitsukoshimae) both have unoptimized stop counts of 15 stops, but both optimized transfers and had 1 transfer in their paths. However, the path that is optimizing distance spanned a distance of 13.4 km while the other path spanned 13.6 km.

Optimizing number of **transfers** vs optimizing **distance, transfers, and stops**

The path from A01 (Nishi-magome) to C05 (Nogizaka) both had optimal transfer counts of 2 transfers. However, the path that is optimizing distance and stops as well had better distance values of 13.8 km vs 16.9 km and 16 stops to 19 stops, respectively.

Optimizing **distance** vs optimizing **distance and stops**

The path from A09 (Daimon) to C04 (Omote-sando) both had optimal distances of 5.9 km and suboptimal transfer count of 3 transfers. However, the path that is optimizing stops was able to take 9 stops instead of 10.

Optimizing **distance** vs optimizing **distance and transfers**

The path from A10 (Shimbashi) to C01 (Yoyogi-uehara) both had optimal distances of 8.1 km and unoptimized stops of 12 stops. However the path that is also optimizing transfers was able to route 2 transfers instead of 3.

Optimizing **distance** vs optimizing **distance, transfers, and stops**

The path from A01 (Nishi-magome) to N08 (Yotsuya) both have optimal distances of 14.1 km but the second path that optimizes transfers and stops was able to route 2 transfers vs 3 transfers and 16 stops instead of 17 stops.

Optimizing **distance and stops** vs optimizing **distance, transfers, and stops**

The path from A10 (Shimbashi) to C01 (Yoyogi-uehara) in this case has an optimal distance of 8.1 km and optimal stops of 12 stops, but the path that is optimizing transfers took 2 transfers instead of 3.

Optimizing **distance and transfers** vs optimizing **distance, transfers, and stops**

The path from A10 (Shimbashi) to Z01 (Shibuya) has an optimal distance of 6.3 km and optimal transfers of 2 lines, but the path that is optimizing stops routed 9 stops instead of 10.

Optimizing number of **transfers and stops** vs optimizing **distance, transfers, and stops**

The path from A01 (Nishi-magome) to C08 (Kasumigaseki) both have optimal transfer and stop counts of 2 transfers and 15 stops. However, the path that is optimizing distance spans 12.1 km instead of 12.7 km.

These examples all show that our different optimization options can truly get different results based on the user's preferences. Further, these examples shows cases where if both paths are not optimizing an attribute, they will return the same unoptimized value and if both paths are optimizing an attribute they will return the same optimized value. There are also a few

paths in this case where the optimal and suboptimal paths with respect to one attribute are equal which could be due to overestimation or the fact that the path unintentionally optimized another attribute even if the user didn't explicitly select it. This is explained by the fact that our stop estimation heuristic uses the distance heuristic to calculate the distance remaining. So paths that are optimized for stops are also partially optimized for distance as well, which is expected behavior since generally it takes more stops to go longer distances.

To summarize the key findings of our project as a whole we can see that while A* search is a good algorithm for using heuristics to make greedy choices on which stations to try and add to our path, the benefits of this are not fully captured in medium to small sized datasets. However, there is a strong benefit of A* in the fact of its modularity and its general structure that allows for any variation of heuristic functions as long as they are admissible. This modularity is what helps us easily implement the different heuristics for distance, transfers, and stops. Additionally, in this project we found that our new proposed connections to help mitigate overall traffic and improve the efficiency of certain paths were actually beneficial. Any time we run the pathfinding algorithm, we automatically check to see if any of these new connections were used and we found that oftentimes they were used in order to calculate an optimal solution.

In the future, we would try to optimize our algorithm further. We would try to reduce the overall runtime by using different data structures or by trying other heuristics. Other heuristics could also potentially further optimize our results. Additionally, we could use real passenger data on traffic, so that we could determine better new connections to add that would be helpful to actual users. Furthermore, we would like to compare the A* Search algorithm to the Bellman-Ford algorithm to analyze if our algorithm is the most optimal for our goal.

Works Cited

- [1] Daniel Foeaad, Alifio Ghifari, Marchel Budi Kusuma, Novita Hanafiah, Eric Gunawan, A *Systematic Literature Review of A* Pathfinding*, Procedia Computer Science, Volume 179, 2021, Pages 507-514, ISSN 1877-0509, <https://doi.org/10.1016/j.procs.2021.01.034>.
(<https://www.sciencedirect.com/science/article/pii/S1877050921000399>)
- [2] Dian Rachmawati and Lysander Gustin 2020 *J. Phys.: Conf. Ser.* 1566 012061 DOI 10.1088/1742-6596/1566/1/012061(<https://iopscience.iop.org/article/10.1088/1742-6596/1566/1/012061>)
- [3] Jugendhackt (2020) Tokyo Metro Data [Source code].
<https://github.com/Jugendhackt/tokyo-metro-data>.
- [4] n1n9-jp (2014) Data of Japan [Source code]. <https://github.com/dataofjapan/land>
- [5] Piuccio (2018) List of Japanese Railway Stations [Source code].
<https://github.com/piuccio/open-data-jp-railway-stations>