

Team Wish Upon A***Artificial Intelligence MTA Trip Planner****Table Of Contents**

- **Introduction**
- **Introduction to Route Planning(2)**
- **State of the Art (7)**
- **The Data and Representation (8)**
- **Google Maps AI(19)**
- **The Algorithm (20)**
- **Terminal User Interface.....(24)**
- **Sample Routes and Results Evaluation. . . (25)**
- **Known Issues / Future Steps(32)**
- **Paper Links(33)**

Artificial Intelligence (AI) is a growing field with endless applications. Its subfields range from machine learning, in which algorithms are utilized to predict outcomes in a multitude of issues, including medical treatments, cancer detection, and image recognition, to constraint satisfaction problem solving, which are used to optimize timetable management, handle hardware configuration, and so much more. Our group decided to focus on what is arguably the most important basic tenet of AI: search. Inspired by the many search problems we solved in class, including the N-Queens problem and the game 2048, we wanted to continue our learning by expanding to a larger, more applicable problem: route planning. As native New Yorkers, no doubt the best problem to tackle is the Metropolitan Transit Authority (MTA)'s vast subway system, underfunded and overwhelmed.

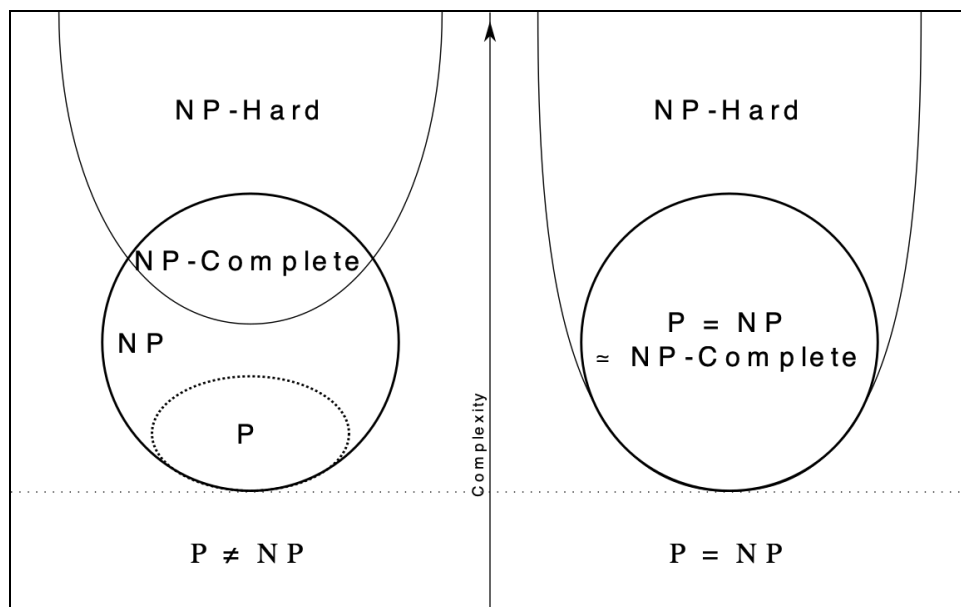
Our goal was to build an AI-powered MTA Trip Planner, a simpler version of the MTA's online Trip Planner, using search algorithms. The minimum viable product we aimed for was a trip planner that gave directions for an optimal route between two subway stations in New York City using first depth-first search and then the A* algorithm. We decided to limit our project complexity to only two modes of transportation: subway and walking, of which we will only be using search algorithms to determine the subway route, but not the walking route. Our decision to choose these limits was partly due to scope, but also partly due to the lack of adequate or clean data available. The project was further limited to the size of the system itself, only operating in more or less the five boroughs. Initially, we planned to only focus on the borough of Manhattan, but we decided to expand it to all five boroughs. After accomplishing the MVP, we slowly added new constraints and new features.

We have constrained the problem to only the subway. The routing will feature the ability to travel to accessible stops, simply route between two stops, or routes between two addresses. It will route with express and local trains in mind, and the minimization of transfers. All aspects will be discussed throughout the paper, but to begin, we will dive into the nuances of transportation planning as a whole, and in the context of our problem.

Introduction to Route Planning

Transportation planning, scheduling, and route planning are enormous, highly complex fields. On a large scale, these problems often contain tens or hundreds of variables to consider. If

formulated as graph problems, many often have thousands if not tens or hundreds of thousands of nodes. Most all Computer Science students are familiar with the frustration of the traveling salesman problem, the poster child of an NP-Hard problem. These questions are defined as those problems that are at least as hard as those in NP, but are not necessarily elements of NP, and may not necessarily be decidable (predictions in machine learning often are undecidable). Luckily for us, our problem is much more constrained than that of the traveling salesman, and does not consist of solving an undecidable NP-hard problem. Rather, in the simplest sense of route planning, we are simply looking for the shortest path between A and B, and if we need to have multiple routes, we would simply run this problem multiple times. A simple shortest path problem of the sort is actually of type P, which means it can be solved in polynomial time.



Now that we know our problem can be solved deterministically, we can begin to think about how to solve such a problem. If there exists a multiple-stop route, we can simply run the original single shortest path algorithm multiple times, as each is its own independent problem, not dependent on the other, so we can simply focus on solving a single shortest path between two

nodes. The natural inclination is to lean directly towards an algorithm that does exactly what we need, find the shortest path to all possible nodes from our starting node, and we could just pick the path we so desire. This is one of the first search algorithms any computer science student learns, Dijkstra. Dijkstra's algorithm was initially published in 1959. Today, it is the basis of many other search algorithms. In non-mathematical terms, it functions by:

- Beginning at a starting node
- Initialize all distances to infinity
- Visit each of the neighbors of that node
 - Calculate the distance to each neighbor
 - Update the distance to that neighbor if the distance is smaller than the labeled
- Mark that node as explored
- Repeat with neighbors of that node until the process has been completed for all nodes.

It is a very simple algorithm to understand and code. In theory, it would quite directly solve our problem. However, using Dijkstra brings up two very important issues, time and space complexity.

The nature of Dijkstra's algorithm requires the visitation of every single node in the graph. This is infeasible for large scale transportation problems. If one were to have a network of hundreds of thousands of bus stops, it would be incredibly time and space consuming to run Dijkstra on such a large network. The worst case runtime on Dijkstra's algorithm is that of a complete graph, $O(|E| + |V| \log |V|)$. While practically this wouldn't actually be an issue given our limitation to only the NYC Subway system, we decided it would be more true to the

spirit of a transportation routing problem to avoid having complexity of the sort, especially if we were to eventually update our program with the ability to route between bus stops.

The first collection of searches that we experimented with were uninformed searches, namely Depth First Search (DFS) and Breadth First Search (BFS). While both would provide us with a path that would get us to the final destination, both DFS and BFS are not optimal. They have no given information that would lead them to choosing one path over another, but only the nature of how they are built: one expands all the way to the end of its path first then backtracks (DFS), while the other expands every immediate node it has access to. While useful if we simply need to find a particular node, they both ignore the primary desire in route planning, to determine an optimal route. They did serve a small purpose in our project however, as DFS was used as the baseline search in the initial development of our graph and other data structures, as the search is simple and easily predictable.

As these searches have little benefit to us other than being easily predictable, the final product would not integrate them. Therefore, we transitioned to informed search. Unlike uninformed search, informed search algorithms use a calculated heuristic value to determine which nodes they should expand first. In our research on route planning, a consensus of results, papers on the topic, articles regarding searches, spoke of A* as a common algorithm that is applied to transit routing problems. This choice makes intuitive sense as opposed to a basic greedy best-first search.

Greedy best-first searches and A* are both best-first search algorithms. The key difference is that A* is not considered greedy. The term greedy, in the context of search, means that the search algorithm only looks at the nodes available to expansion in order to determine where to go next. This is represented by the calculation $f(n) = h(n)$ where $h(n)$ is the heuristic function applied to each of the nodes to be expanded. If implemented using a min-heap, that with the minimum heuristic value would be the “best” node to be expanded. Using a search like this however dramatically limits our abilities in a problem like transportation planning. This type of search is not guaranteed to be optimal, as not enough information is provided. An general example is we may reach our goal earlier, even though a longer path with more nodes actually has a lower weight, but our heuristic directed us towards the other, shorter path, but with a higher gain in the immediate expansion.

Thus, we selected A* as our search algorithm. A* is optimal best-first search (assuming an admissible heuristic). Unlike greedy best-first search, A* tries to minimize a slightly but significantly different function, $f(n) = g(n) + h(n)$ where $h(n)$ is the heuristic function as described earlier, but we now have the extra component of $g(n)$, which represents the cost from the starting node to the current node. This is critically important in the route planning problem, as there are many possible pathways to the final destination, and many of the significant portions of information are not available until much further from the starting node. This will actually allow our search to provide optimal results given out desired weights and route preferences. In summary, in contrast to a basic greedy best-first search, A* evaluates the path up

to and including the current node plus the heuristic estimation, while greedy best-first search only evaluated the heuristic estimation at the current node.

State of the Art

Before we dive into our implementation, we wanted to expand on the state of the art. While we used A* and constrained our problem in various ways, we did not need to perform much, if any pre-processing, besides the data structure generation that the next section will describe. As mentioned in our discussion of Dijkstra, for comprehensive route planning applications, there will exist tens if not hundreds of thousands of nodes. This amount is too much for efficient real-time calculation of shortest paths. Therefore many state of the art algorithms use graph segmentation techniques. Linked at the end of the paper is a fantastic summary paper of all of the major techniques used in transportation route planning, compiled by some of the most prominent creators of these algorithm modifications, such as Andrew Goldberg, the developer of Landmark A*. Landmark A* is one potential search modification to improve routing, where distances are calculated to dynamically placed or chosen landmarks, depending on the state of the search, and averaged in some form if multiple are used at a time. This allows for better routing than a simple geographical distance based heuristic. A common pre-processing method includes node clustering, a method discussed in several papers discussed in our research. The purpose of such clustering is to have a collection of pre-processed distance lookups between node groupings, so that searches can be split up into smaller problems, run inside each group. An interesting method found involved using a k-d tree for such segmentation. It allows for smaller

clusters near concentrated hub areas, and larger clusters around simpler, spread about suburban regions. In said paper, k-d partitioning resulted in on average 6x faster results than Dijkstra, or 5.34x faster than A* with basic euclidean distance. Other state-of-the-art algorithms include using arc length, and even machine learning to and data analysis of their users in order to shift weight biases towards commonly desired/used paths. If we were using a dataset of such a large magnitude, it would have been quite an interesting challenge to implement these optimizations. If we return back to the project, we will definitely have to if comprehensive bus data is added.

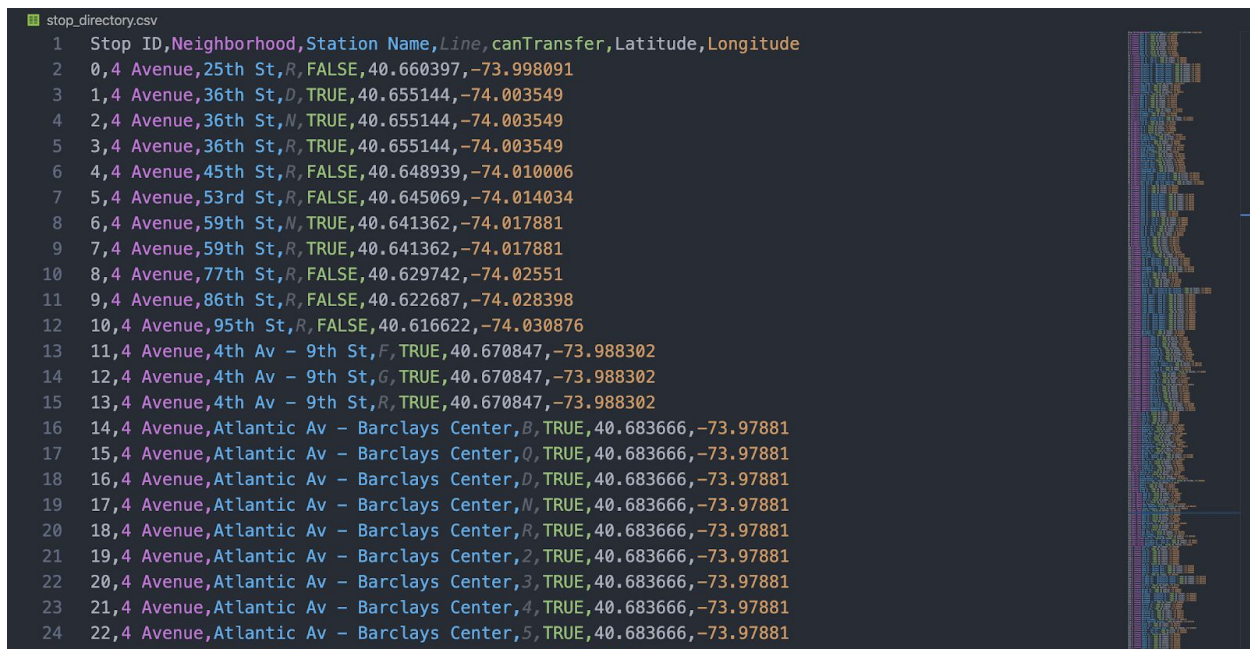
The Data and Representation

Now that we have established our algorithm of choice, A*, we must plan how to actually represent the data that the search will be traversing. This was a much more complex task than we originally realized. We began our process by seeing if there exists any official MTA datasets of stop lists, lines, stop coordinates, transfer information, etc. This was a very tedious process and yielded very little. The official MTA API is written in a format called GTFS. Unfortunately, we had quite a lot of trouble accessing the MTA's API, the static feeds simply weren't able to be accessed. Furthermore, it may be better that we did not try to incorporate their direct data, as due to new scheduling and management under the Essential Service Plan because of the coronavirus, there are dramatic modifications to service all around the system.

So, we began the process of formulating our own dataset for this project. We managed to find an old (2012) dataset containing information similar to what we wanted. We needed a dataset of every single stop on the subway, containing its corresponding train line, latitude and longitude coordinates, name, stop_ID, and whether or not one can transfer to another line at that

given stop. The original dataset contained some of that information, but some was outdated, and so we had to manually go through and fix the mistakes, removing stops that no longer exist, changing a few names to the most recent ones, and reformatting it to give each stop its own independent row in the dataset. We also added additional information for later implemented features, such as whether the stop is wheelchair accessible, or whether or not the stop is on an express or local line. The resulting file contains 743 stops, 300 more than the official MTA count as every line's stop was tabled independently.

Stop_directory.csv



Stop ID	Neighborhood	Station Name	Line	canTransfer	Latitude	Longitude
0	4 Avenue	25th St	R	FALSE	40.660397	-73.998091
1	4 Avenue	36th St	D	TRUE	40.655144	-74.003549
2	4 Avenue	36th St	N	TRUE	40.655144	-74.003549
3	4 Avenue	36th St	R	TRUE	40.655144	-74.003549
4	4 Avenue	45th St	R	FALSE	40.648939	-74.010006
5	4 Avenue	53rd St	R	FALSE	40.645069	-74.014034
6	4 Avenue	59th St	N	TRUE	40.641362	-74.017881
7	4 Avenue	59th St	R	TRUE	40.641362	-74.017881
8	4 Avenue	77th St	R	FALSE	40.629742	-74.02551
9	4 Avenue	86th St	R	FALSE	40.622687	-74.028398
10	4 Avenue	95th St	R	FALSE	40.616622	-74.030876
11	4 Avenue	4th Av - 9th St	F	TRUE	40.670847	-73.988302
12	4 Avenue	4th Av - 9th St	G	TRUE	40.670847	-73.988302
13	4 Avenue	4th Av - 9th St	R	TRUE	40.670847	-73.988302
14	4 Avenue	Atlantic Av - Barclays Center	B	TRUE	40.683666	-73.97881
15	4 Avenue	Atlantic Av - Barclays Center	Q	TRUE	40.683666	-73.97881
16	4 Avenue	Atlantic Av - Barclays Center	D	TRUE	40.683666	-73.97881
17	4 Avenue	Atlantic Av - Barclays Center	N	TRUE	40.683666	-73.97881
18	4 Avenue	Atlantic Av - Barclays Center	R	TRUE	40.683666	-73.97881
19	4 Avenue	Atlantic Av - Barclays Center	2	TRUE	40.683666	-73.97881
20	4 Avenue	Atlantic Av - Barclays Center	3	TRUE	40.683666	-73.97881
21	4 Avenue	Atlantic Av - Barclays Center	4	TRUE	40.683666	-73.97881
22	4 Avenue	Atlantic Av - Barclays Center	5	TRUE	40.683666	-73.97881

This, however, is not nearly enough data to service our transportation planning. It provides no information about how the stops are actually connected. It also is a processing power consuming reference for the transfer information, as searching would be required. Therefore, using this dataset and the official MTA we map, by hand and by script we generated two more data files providing us with the information we would need. The first is a table of route order. This is manually done by tracing through the MTA's map, as we could find no other datasets that

provided easily parseable and reliable files. For each train line, the stop order follows with the list of stopID's. The resulting file is stop_order.csv. The columns in stop_order also define what we refer to as “uptown” and “downtown” stops. As the index of stop_order's columns increases, stops get further “uptown,” and vice versa. Thus, the first stop on any line is the first stopID associated with it, while the furthest uptown stop on any line is the last stop listed.

We then wrote a script that took the information from stop_directory.csv and compiled another file, stop_transfers.csv. Compiling these files helps reduce computational complexity and minimizes data searching, while also making it much easier to understand that data we are actually working with. The files described are displayed below.

```

stop_order.csv
1 1,141,545,540,536,542,534,543,539,246,513,519,520,521,126,524,318,525,526,529,530,531,503,535,504,505
2 2,687,689,683,685,695,693,691,574,571,578,569,19,583,580,548,552,562,558,335,537,247,522,127,527,532,
3 3,677,682,678,676,679,680,681,588,582,587,575,572,579,570,20,584,581,549,553,563,559,336,538,248,523,
4 4,589,576,21,585,550,666,674,560,670,136,602,112,661,650,618,621,636,623,624,632,625,628,626,629,630,
5 5,688,690,684,686,696,694,692,577,22,586,551,667,675,561,671,137,603,113,662,651,619,622,716,744,748,
6 6,672,101,673,269,665,138,653,654,655,604,658,114,659,660,663,664,647,648,649,652,697,698,701,702,703,
7 7,592,129,256,605,616,607,596,614,615,610,599,609,617,600,462,597,598,608,590,591,611,601
8 A,430,427,426,428,402,396,420,412,404,105,338,554,332,327,343,179,309,119,314,290,296,509,304,305,306
9 A1,422,424,425,429
10 A2,746,482,480,499,490,489,488,487,486,497
11 A3,752,483,481,500,495,491,493,484,501
12 B,48,63,53,56,50,60,41,14,24,279,265,344,75,252,257,446,315,319,321,323,325,286,330,288,291,294,297,298
13 C,403,419,423,411,397,417,415,421,408,413,205,401,409,405,106,339,555,333,328,341,345,180,307,310,120

```

stop_order.csv (left → right = downtown → uptown)

```

stop_transfers.csv
1 Stop ID,Transferable Stops
2 0
3 1,2,3
4 2,1,3
5 3,1,2
6 4
7 5
8 6,7
9 7,6
10 8
11 9
12 10
13 11,12,13
14 12,11,13
15 13,11,12
16 14,15,16,17,18,19,20,21,22
17 15,14,16,17,18,19,20,21,22
18 16,14,15,17,18,19,20,21,22
19 17,14,15,16,18,19,20,21,22

```

stop_transfers.csv

Using this newly compiled data, we created the two main classes that would drive our graph representation and A* search: `Subway_System()` and `Stop()`.

We decided to take an object-oriented approach to solving the problem because the subway system as a whole acts a lot like a super class with many objects that have similar characteristics and functions. This seemed to be the best approach to creating a search space and graph that represented how the algorithm would conduct its search. We were fortunate enough to have a subway dataset ready for processing; however, there wasn't an easy way to use that data to build the subway system objects. This problem is what led us to reformatting the dataset in the first place. We initially pictured the subway system as a graph where the nodes represented stations, the edges were weighted and represented the time between stations, and stations that had transfers would have more edges to represent those transfers. In the graph, there was no easy way to establish which edges belonged to which train lines. However, we were able to change that when we established the classes. The graph served as a good outline and jumping point for our information structure.

Stop Class Essential Variables

```

# Stop Class, a.k.a. the Node
class Stop():
    # Variables
    stopID = -1      # Unique Stop ID for every stop on every train Line
    neighborhood = '' # Neighborhood it operates in
    station_name = ''
    line = ''        # Train name
    transfers = []    # List of Stop objects.
    latitude = 0
    longitude = 0
    prevStop = 0      # Reference to previous stop's node
    nextStop = 0      # Reference to next stop's node
    lastVisited = None # Reference to last stop visited (set by search algorithm)
    express = ""       # Determines whether this is an express/local train
    accessibility = "" # Determines the extent the stop is ADA accessible

    #heuristic uses the number of times the algorithm has "transferred"
    #and the number of stops left to the goal if the current and ending stops are on the same Line
    transferCount = 0
    #initialized as a large number to encourage staying on a train that stops at the goal stop
    stopsToEnd = 100

```

The first class we built was the Stop class and each Stop object represented a stop on a train line. Each stop had a unique stopID, the train that stopped there, the name of the stop, the neighborhood the train operated in, the coordinates of the stop, a reference to the last Stop Node visited by the algorithm, a reference to the previous Stop Node in the train line, a reference to the next Stop Node in the train line, a list of transfers available at that stop, whether it was an express stop or a local stop, how many transfers have occurred up until that stop, how many stops until the destination stop, and whether it was ADA accessible and to what extent.

The stopID was used to make the data relational. With the ID, we could quickly identify stops during major steps, such as when adding next Stops to the frontier or identifying what stop the user was trying to input. The neighborhood and station name was used mainly for the user interface so that the directions were readable. The line name was used for determining if the algorithm was on the right path to the destination stop.

The coordinates of the stop were used as identifying factors apart from the stopID. While the stopID was a useful identification system, it separated trains that stopped at the same station. For example, the 4/5/6 trains all stop at Grand Central - 42nd Street. However, each train has its own Stop object for Grand Central and, thus, its own unique stopID. We could have made an additional dataset for linking stops to shared stations; however, that data wasn't readily available or easily obtained and would have required us to manually enter almost all the data. An easier alternative was to use the coordinates to identify Stops that shared the same station. To do so, we concatenated the latitudes and longitudes for each Stop when we needed to compare Stops and if they were equal, they shared a station and thus, were transferable to one another. To expedite this, we wrote a script that grouped Stop objects by their "stations" so that Stops that shared a station were intrinsically linked together. The end result was a dataset of each stopID followed by the stopIDs that shared a station with it. This is where the transfers list came into play. The transfers list for each Stop object is taken directly from this generated dataset. Therefore, when dealing with any given Stop, we had quick access to other possible subroutes that could be taken on different lines and were connected with all the other Stops at that station. While the coordinates were useful for grouping Stops by station, they proved more useful for mapping user input to stopIDs. We used the Google Maps API to determine what station the user was most likely referring to because we assumed user input would be incomplete or incorrect at times. Alternatively, if the user's origin or destination was not at a subway station, then we would provide walking instructions to/from the nearest subway stations. Once we determined the Stop the user was referring to, semantically, we still needed to identify its stopID within our

directory. For any given stop, our coordinates did not always match Google's coordinates exactly and this could cause a problem because they would not be seen as equal even when they are. As a result, we solved this problem by using distance rather than equal coordinates to connect a stop identified by Google with a stop in our system. Since our dataset theoretically includes all stops in the subway system and the result returned by the Google Maps API is guaranteed to be a subway stop (more on how this is accomplished in the API section), then the Stop in our system that was the closest to the stop provided by Google would be the correlating Stop. For this, we used a coordinate distance formula we wrote into a function in each Stop object.

Next, each Stop object has references to its last visited Stop Node, the previous Stop Node in the train line, and the next Stop Node in the train line. The lastVisited reference is set as the search algorithm explores the search space. It's used to keep track of the direction the route is moving in and to prevent the algorithm from moving backwards, i.e. taking a train two stops backwards transferring to another train and taking it three stops forward. The previous Stop Node and the next Stop Node references are mainly used to allow movement and expansion of the search space. It is meant to represent the functionality of a doubly linked list, for easy access to the previous and next Node for pre-emptive consideration before choosing a route.

Then, we have the express indicator, the transfer count, and the number of the stops to the end. The express indicator is a string variable that is set to either "express" or "local." The transfer count is an integer variable used by the algorithm to keep track of how many transfers are required from the start stop to reach this stop. The number of stops to end is the number of

stops from this stop to the destination stop, only if this stop is on the same train line as the destination stop. These three variables are used for the heuristic function. Generally, there is a preference for express trains depending on the number of stops to the end and a lower number of transfers.

Lastly, there is the accessibility variable. This variable is used to determine if a stop is ADA accessible. The possible values for this variable are “UPTOWN,” “DOWNTOWN,” “BOTH,” or “NEITHER.” Accessible stations, unfortunately, are few and far between in New York City. This issue has been raised many times by the disabled living in NYC, but the MTA has yet to scratch the surface. Several of the stations are incomplete, leaving only one platform accessible and the other not. Therefore, stops marked as “UPTOWN” only have an accessible uptown platform and the same rule applies to stops marked as “DOWNTOWN.” Stops marked as “BOTH” have both uptown and downtown platforms accessible and “NEITHER” have none. This variable triggers an “accessible stops only” search that enforces only accessible start stops, transfers, and end stops. The Stop class functions include accessors and mutators, distance calculation, comparison, and heuristic calculation. As the algorithm traverses the stops, it uses these functions to calculate the heuristic and weights that guide its decision-making.

```

# =====
# Initialization Data
# Stop Directory: connects Line names to route details
directory_data = open('stop_directory.csv','r').read().split('\n')
# Transfers Directory: connections stopIDs to available transfers at the same Location
transfers_data = open('stop_transfers.csv', 'r').read().split('\n')
# Stop Order Directory: connects Line names to stops they visit
stop_order_data = open('stop_order.csv', 'r').read().split('\n')

# Subway System Class
class Subway_System():
    # Initialize components of subway system and call the necessary functions to set up the three main data variables
    def __init__(self, directory, transfers, train_lines):
        self.transfers = self.setupTransfers(transfers) # Dictionary: key stopID -> value List of transferable stops
        self.directory = self.setupDirectory(directory) # Dictionary of Stop Nodes: key stopID -> Stop Node
        self.system = self.setupSystem(train_lines) # Dictionary of Routes: key trainLine -> List of Stop Nodes in order
        self.total_stops = len(self.directory) # Total node(stops) in the search space
        # Changes all the stopIDs in the transfers var from ints to references to their associated Stop Nodes
        self.addNodeTransfers()
        # Determines all the prevStop and nextStop values in all the Stop Nodes in system var
        self.addPrevNext()
        # Read in API Key:
        try:
            file = open("api_key.txt")
            api_key = file.readline()
            file.close()
        except:
            sys.exit("No api_key.txt found.")
        # Set up Google Maps Client for data requests:
        self.gmaps = googlemaps.Client(key=api_key)

```

Subway_System Class Essential Variables and Initialization Steps

After we set up the Stop class, we integrated it into a larger network: the Subway_System class. Before we create an instance of the Subway_System, we prepare the datasets by reading them into the script and splitting them by line. Then, we pass them into the constructor of the Subway_System, in which they are initialized. The constructor takes care of the bulk of the initialization. There are five major steps to initialization that must be performed in order because they are dependent on each other:

1. Set up the transfers dictionary
2. Set up the Stop directory
3. Build the system(search space)
4. Convert stopIDs in Stop Node transfers variable to references
5. Connect the Stop Nodes on each line

The first step is setting up the transfer dictionary. This is a class dictionary. Each stopID has a key in the dictionary and its value is initially set to a list of the stopIDs of the Stops that share a station with that Stop. This dictionary is set up with the transfer dataset that was passed in. It will be used to determine what transfers are available at each Stop. The second step is to set up the Stop directory. This dictionary is built with the stop_directory dataset that was passed in. The format of this dictionary is the stopID as the key and a reference to the associated Stop Node as the value. This dictionary is used as a general reference for all the stops. Then, the third step is to build the system itself. With the stop_order dataset, we create a dictionary with the key as the train line and the value as a list with references to all the Stop Nodes on that train line in order from southernmost to northernmost stop. We considered using a doubly linked list instead of a list, but we decided since we had a finite dataset that wasn't very large (~800 Nodes), it would not cost much, if at all, to use a list. With a list, we'd have constant time access and quick traversal at the cost of relatively little memory. The next step is to convert the stopIDs in each Stop Node's transfers list to references to those Nodes. This step needs to be completed after the stop directory is set up because the directory will be used to get the references needed. Lastly, the final step is to set prevStop and nextStop for each Stop Node. This step needs to come after the system setup because it relies on the order of the stops to determine which Stop comes before and after it. Setting these references are done simply by traversing each train line and getting the item before and after it. For this, the easiest is to use indexes. Usually, having an array would make it easy to get the previous and next Node, but that would require using an iterator or keeping track of the indexes. We thought it would be best to create a data structure that

combined arrays and doubly-linked lists. This would allow us to get the next and previous node without the extra steps mentioned.

The Subway_System is also mainly where the Google Maps API is utilized, so in initialization, we also read in the Google Maps API Key obtained from the Google Cloud Platform. Then, we use the key to establish a Google Maps Client to use the API throughout our code.

There are several functions in the Subway_System class. Most of the functions are related to establishing information for the algorithm and helping it along the way. The transferStopsToEnd function calculates the number of stops and transfers until the destination stop. The findStop function takes the user's input and returns the possible stops that match the query. This function is used at the beginning of the program. After the user is asked what their start and end stops are, this function is used to provide suggestions for the user to confirm which stop they mean since there may be several stops with the same name. Then, these stops are set as the origin and destination and passed over to the search algorithm. This function ties in with the startToStation and stationToEnd functions. These functions are used when the start and/or end address is not at a subway stop. They determine what subway stop is nearest to the user's inputted addresses, provides walking directions from the address to that subway stop, connects the external stop to a stop within our system, and passes the information to the search algorithm. These functions use the Google Maps API to pinpoint the user-inputted address, find the nearest

stop, and get walking directions between the two. Overall, we are pretty happy with our search space structure. It is easily scalable, efficient, and versatile.

The Google Maps API

There were several features that we were able to implement, but there were also many features we weren't yet able to implement. In these places, we used the Google Maps API to aid us with directions and coordinates. Out of all the Google Maps APIs, we used the Directions API and the Places API. In the Directions API, we used the `directions()` function to get walking directions between two points. We used the `find_nearest()` and `place()` function calls in Places API in order to find the nearest subway station to a point and pinpoint a location based on user input. The use of these specific APIs are limited to the `findStop`, `startToStation`, and `stationToEnd` functions in the `Subway_System` class. The API was brought in to resolve extraneous issues with user input and walking directions. Our project did not yet have the capability to determine the best routes by walking. We needed this accommodation to give users the option to input start and end addresses that didn't need to be tied to a subway stop. In the interest of time, we decided to use the API to "convert" start and end addresses to start and end subway stops. We thought that the focus of the project was not on user interface or handling user input, so using the API for such an issue was not out of the question. We did include filters for the API calls, however, to narrow our search and increase the likelihood of finding the user's intended address. For the place API function call, we limited the search space New York City based on New York City coordinates that could be found online. This was then used to find the start and end addresses, which didn't necessarily have to be subway stations. Once we established the user inputted addresses, if they were subway stations we used the `find_nearest` api

function call to find the nearest subway station. This api call was also restricted to New York City and specifically results of type *subway_station* as determined by Google. This was extremely useful in narrowing down the results. The directions api we used last to connect the whole route. For this api function call, we limited it to walking directions only, departing now.

The Algorithm

Unfortunately, the NYC subway system as a whole is much more complex than most subway systems in the world, save for a few in Asia (especially Japan). Most subway systems are designed in a model known as the “hub and spokes,” where there is a central hub area, and lines radiate outward into the suburbs. New York City’s system is different. Instead, we have a collection of hub points that are dispersed across the system. These hub points contain much more complex criss-crosses of lines and transfers than the distant outward spokes. Even the outward spokes aren’t so simple, with the ends sometimes converging in various points that are quite far away from the city center, such as Coney Island - Stillwell Ave and Broadway Junction.

In addition to their being multiple hubs, lines often don’t stick to their neighbors. New York City is one of the only subway systems in the world to integrate express and local trains into a single train network. Also, the areas in which these trains are considered express and local are very inconsistent. For example, the F train is local in Brooklyn when it runs alone and when it runs alongside the G, local in Manhattan when it travels alone or alongside the B, D, and M trains, and express in Queens when it travels with the E, M, and R trains. Thus, trains like the F vary in the lines that they run parallel to. This makes the problem significantly more complex.

As New Yorkers, we have an intuitive sense of when we would like to take a particular train over another: how long we are willing to wait, whether we risk waiting for the express or just take the local, or whether we transfer at a certain stop. We sought to quantify this subjective thought process in our cost function and heuristic. This function tracks:

- costs based on the number of transfers made
- costs based on the number of transfers to be made
- the number of stops made before the current stop
- the number of stops remaining before the destination
- whether next stop is express or local
- the geographical (latitude/longitude) distance from the stop to the final destination
- whether the current stop's train stops at the goal stop

The weights on each of these metrics are also subjective: they were decided largely by trial and error. We manually reviewed hundreds of simple and complex (up to four or five or transfers) routes set by various heuristic weights and decided on the one that led to the most “optimal” results. Like we emphasized in our presentation, this optimality was shaped by our own bias. A more objective measure, which would have compared our algorithm's chosen routes to those of Google or Apple Maps, was not available due to the service changes imposed by COVID-19, which we did not consider. Examples of evaluation routes and what we considered optimal or sub-optimal is discussed in the evaluation and results section.

Our algorithm used a very similar structure to the A* we designed for the first assignment. We put our starting station into our frontier (a priority queue), initialized an empty explored set (which we filled as more stop nodes were popped from the frontier), checked for the goal state, and added all valid child nodes to the frontier. We also made sure to add all transfers of previously-visited stations (visited stations are connected by the lastVisited variable, using a doubly-linked list structure) to the explored set each time a new node was popped from the frontier. In other words, transfers of previous stops were not valid destinations, so no circular routes were produced.

Our algorithm managed accessibility by modifying the process by which the list of child nodes (neighbors) was generated. The most difficult constraint that the MTA's accessibility posed was the lack of "bidirectional" accessibility: certain stations are only accessible on the uptown or the downtown side. Our algorithm was not able to detect future stop selections; we could not predict whether it would choose to move uptown or downtown. Therefore, we separated the accessibility checks for the next stop and the previous stop. CurrentStop.nextStop would be added to the list of valid neighbors if either accessibility was not selected, the current stop's line and the next stop's line were the same (then the physically impaired commuter would stay on the train they are already on), or if currentStop's accessibility value was 'BOTH' or 'UPTOWN.' Similar logic was applied to currentStop.prevStop, except with 'DOWNTOWN' instead of 'UPTOWN' being accepted. For transfers, the requirement for "frontier consideration" was less stringent: any accessibility value other than 'NEITHER' was accepted. This light check

is possible because if they were expanded, the transfer's child nodes would check themselves for proper accessibility directions, and no invalid route would be created.

Finally, we had to make a special exception for the A train's divergent paths in eastern Queens. The real-world A train has three stops that it can potentially start at: Ozone Park - Lefferts Blvd, Far Rockaway - Mott Avenue, and Rockaway Park - Beach 116 St. Queens-bound A trains are designated to stop at one of these stops. We called the three A train extensions A1 (Ozone Park start), A2 (Far Rockaway start), and A3 (Rockaway Park start), respectively. All three converged at a single ending stop, Rockaway Blvd (A/A1/A2/A3), at which point a single A train continued to Manhattan. This structure posed a problem for accessibility, as Rockaway Blvd is not an accessible station. Even though no transfer is really occurring (no A1/A2/A3 distinction is made by the MTA), we instructed our accessibility transfer logic to accept transfers between lines with the same first character (A). We believe that the errors resulting from many accessible routes around northeastern Brooklyn and eastern Queens are related to the faulty transfer permissions on the Rockaway lines.

The A train divergence also required an adjustment in our stop directory and stop order data sets. On its Rockaway route, the A train stops at two different stations—but not both—depending on whether it is going uptown or downtown. The Rockaway-bound A stops only at Aqueduct - North Conduit Av, while the Manhattan-bound A stops only at Aqueduct Racetrack instead. For the sake of our graph structure, we avoided this strange set of rules, consolidating both stops into a single “Aqueduct.”

In the end, because we were unable to predict the direction the route would choose, we designed our accessibility logic so that all starting stops were bidirectionally accessible. Intermediate stops were selected as transfers depending on their direction matches. Since the end stop is an intermediate stop, the same logic was applied there as well.

Terminal User Interface

Seeing that this is a project focused on learning artificial intelligence, we did not give a lot of weight to the terminal UI. We decided to make the user experience simple and streamlined, but not completely technical. That is why we scrapped passing all the arguments in as extra parameters at the time of the program execution. Instead, we made each necessary parameter an input and gave the program a nice structure of questions with the persona of an android named Greg, a friend of ours in high school who was very passionate and knowledgeable about the MTA. While we generally wanted to assume the user would input perfectly formatted and correct data, we thought it would be better to at least integrate a little realism that could be extended easily to a proper UI if we made one in the future.

The first question asked the user if they needed accessible stations or not, to which the user would reply “yes” or “no.” Based on their response, we would set a boolean to determine if the user needed accessible stops. Then, the next question asked for the name or address of the starting point. Based on the input, we would try to find the exact stop that the user was referring to by comparing the input to the stop names in our Subway_System. If the user input returned

possible matches to a subway station within our subway, then the program would return a list of those options with their associated indices. The user would then pick the one that was their intended station. If we are unable to find any matches within our system, we then call the Google Places API to try and take a guess as to what station the user is referring to. The default assumption necessary here is that the api will try to guess what *subway stop* the user intended, not a place. If it still can't find any results, then we'll return an error message in the form of a dictionary and print it for the user. If the api does find a station, we will inform the user and our program will proceed with finding the stop in our station that corresponds to the stop chosen by the api. More about how this mapping is done can be found in the api section. The same process is then applied to get the ending point. It is important to remember, however, that while there are protections against incorrect/invalid input, if a person intentionally tries to break the program with corrupted input, they will succeed. On this, we hope you will look favorably upon our decision to prioritize other more important tasks. Once all the inputs have been collected, the route calculation function can be called and directions retrieved.

Sample Routes and Results Evaluation

We will now display a few representative demonstrations of our route generation. The selected routes will vary in the user input, number of requisite transfers, the accessibility designation, and their general direction (uptown vs. downtown).

We will begin by showing an accessible route from “brooklyn college” to “av x.” Our program detects “brooklyn college” as being a substring in a station name. We select the 5 train,

but since the two stations are equivalent in this case (they are transfers for each other), this choice is not actually relevant. Later, we will display an example where the user's choice here is relevant.

```
brooklyn college
Which of these do you mean? Please select one by inputting its index.
0 : Flatbush Av - Brooklyn College (2)
1 : Flatbush Av - Brooklyn College (5)
1
Starting Point: Flatbush Av - Brooklyn College (5)
```

```
av x
Let me try and figure out what station you mean...
I estimated your end station to be Coney Island - Stillwell Av (D)
```

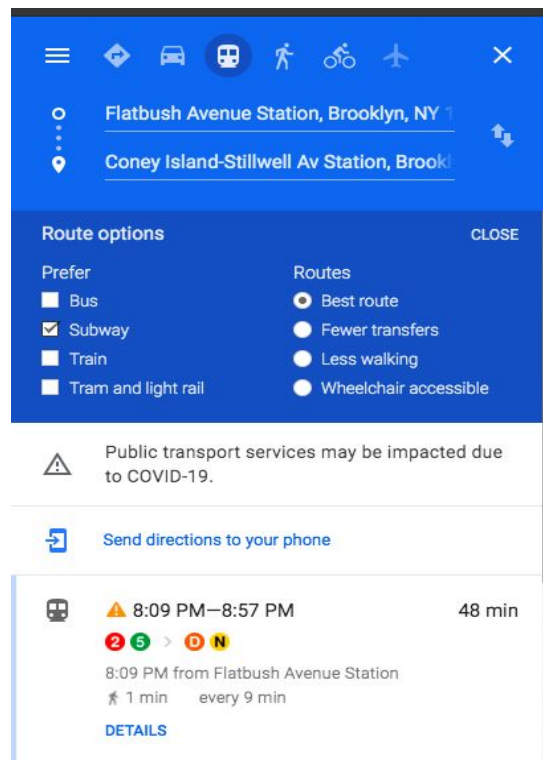
Although “av x” is also a substring of a station name, that station (Av X [F]) is not ADA accessible, so the algorithm ignores it. Instead, the algorithm uses the Google Maps API to detect the nearest accessible station, which is Coney Island - Stillwell Av (the train is not important, since there are four equivalent stops there). The route follows:

```
Intermediate Stops (Accessibility = True):
5, Newkirk Av
5, Beverly Rd
5, Church Av
5, Winthrop St
5, Sterling St
5, President St
5, Franklin Av
5, Atlantic Av - Barclays Center

Transfer at Atlantic Av - Barclays Center to the (N) train at Atlantic Av - Barclays Center.

N, Atlantic Av - Barclays Center
N, 36th St
N, 59th St
N, 8th Av
N, Fort Hamilton Parkway
N, New Utrecht Av
N, 18th Av
N, 20th Av
N, Bay Parkway
N, Kings Highway
N, Av U
N, 86th St
N, Coney Island - Stillwell Av
```

We determine that this is the optimal subway route between these two stations. Google Maps agrees:



Next, we will show an inaccessible route between “hunter main campus” and “lehman college campus.” Once again, the API is used to determine the nearest starting and stations:

```
hunter main campus
Let me try and figure out what station you mean...
I estimated your start station to be 68th St - Hunter College (6)
```

```
lehman college campus
Let me try and figure out what station you mean...
I estimated your end station to be Kingsbridge Rd (4)
```

The user did not select specific subway stations, so we include basic walking directions from the inputted locations to the stations we suggested:

```
Start at: hunter main campus. Walking instructions to subway:

STEP 0: Head southeast on E 68th St toward Lexington Ave. Destination will be on the left.

Board at: 68th St – Hunter College (6)

Intermediate Stops (Accessibility = False):

6, 77th St
6, 86th St

Transfer at 86th St to the (4) train at 86th St.

4, 86th St
4, 125th St
4, 138th St
4, 149th St – Grand Concourse
4, 161st St – Yankee Stadium
4, 167th St
4, 170th St
4, Mt Eden Av
4, 176th St
4, Burnside Av
4, 183rd St
4, Fordham Rd
4, Kingsbridge Rd

STEP 0: Head northeast on Jerome Ave toward W 195th St.
STEP 1: Turn left onto Lehman Walking Roads. Destination will be on the right.
Arrive at: lehman college campus
```

This is one of our favorite test cases. It is a difficult, multi-transfer, shuttle and Rockaway-inclusive, accessible route from Southern Brooklyn to Howard Beach - JFK Airport (A2/A3). The results from “parkside av” to “howard jfk” follow: (on next page, images should be viewed in tandem).

```
Start at: parkside av. Walking instructions to subway:

STEP 0: Head south toward Parkside Ave.
STEP 1: Turn right onto Parkside Ave.
STEP 2: Turn right onto Ocean Ave.
STEP 3: Turn right onto Lincoln Rd. Destination will be on the left.

Board at: Prospect Park (S)

Intermediate Stops (Accessibility = True):

S, Botanic Garden
S, Park Place
S, Franklin Av

Transfer at Franklin Av to the (C) train at Franklin Av.

C, Franklin Av
C, Nostrand Av
C, Kingston – Throop Aves
C, Utica Av
C, Ralph Av
C, Rockaway Av
C, Broadway Junction
C, Liberty Av
C, Van Siclen Av
C, Shepherd Av
C, Euclid Av
```

```
Transfer at Euclid Av to the (A) train at Euclid Av.

A, Euclid Av
A, Grant Av
A, 80th St
A, 88th St
A, Rockaway Blvd
A, Aqueduct
A, Howard Beach – JFK Airport

STEP 0: Head west.
Arrive at: howard jfk
```

This is the type of case where the subjective optimality issue comes to light. The fastest route would necessitate a transfer from the local C to the express A at the earliest accessible convergence of the two lines, Utica Av. However, in order to avoid excessive express train favoring (such as short backward trips on local trains just to get onto express trains), we raised the priority on local trains. This causes the algorithm to stay on the C train longer: until Euclid Av, where the C terminates and only the A continues. This is an illustration of the issues posed by the “express/local” dichotomy.

As we adjusted the heuristic weights, our exhaustive testing covered hundreds of varying routes.

The routes belonged to one or more of these general categories:

- Single line route, no transfer options
- Single line route transfers available
- Multi line transfers available
- Multi line with directional change
- Single and multi-transfer station hubs (Coney Island - Stillwell Av, Times Square - 42nd St, Atlantic Av - Barclays Center, Jackson Heights - Roosevelt Av, Broadway Junction, etc.)
- Long distance single line
- Long distance multi line
- Complex multi-transfer long distance route.

Once the algorithm consistently found optimal routes for simpler input configurations, most of the evaluation was focused on balancing in complex multi-transfer long distance routes. The mid-long range routes were most affected by the express vs. local train bias. We spent the most time testing these problem instances. Adjusting and testing for this express vs. local bias was the

biggest trouble in our evaluations. We settled on weights that tend to pick express routes for short-mid range routes, and where the determination for whether one should take an express route can be made closer to the current node. There are certain examples, like that of from Bergen St. on the F to Grand Ave on the M/R in Queens, where we would still get a local option rather than a faster express one. The reason for this is that the F is considered a local train at the beginning and throughout the majority of this route. The only area it is considered express is in Queens, near the end stop. Based on the biases we currently have, the search would determine that 1 transfer away is the possibility of getting on the R line, on which the destination exists. It would then prefer this, as express bias is only calculated from current stop, with no calculations at the destination.

Intermediate Stops (Accessibility = False):

F, Jay St - Metrotech

Transfer at Jay St - Metrotech to the (R) to

R, Jay St - Metrotech
R, Court St
R, Whitehall St
R, Rector St
R, Cortlandt St
R, City Hall
R, Canal St
R, Prince St
R, 8th St - NYU
R, 14th St - Union Square
R, 23rd St
R, 28th St
R, 34th St - Herald Square
R, Times Square - 42nd St
R, 49th St
R, 57th St - 7th Av
R, 5th Av - 59 St
R, Lexington Av - 59th St
R, Queens Plaza
R, 36th St
R, Steinway St
R, 46th St
R, Northern Blvd
R, 65th St
R, Jackson Heights - Roosevelt Av
R, Elmhurst Av
R, Grand Av - Newtown

Have a safe trip!



These examples illustrate the complexity of the task heuristic weight evaluation for a system like the NYC subway. It was evaluations like these that would help us consider special cases and tune our functions or add new evaluative metrics. Ideally, we would be able to create a more objective evaluation set where we could quickly compare routes between our project and the gold standards, Google and Apple Maps, but as mentioned, the coronavirus schedule changes make such evaluations impossible.

Known Issues

Here is a summary of currently known issues in our program:

- The A train splits off into 3 types in the Rockaways

- Transfers between A trains are treated as negligible
- Printing only shows a single A train
- Must be split back into its 3 types for optimal use.
- At "Aqueduct Racetrack", there exists the only instance of a "split stop"
 - North and South directions are independent stops on a single line (our structure cannot handle that)
 - Currently handled by combining into a single stop, "Aqueduct"
- The heuristic balance between choosing an express train or local train is very sensitive
 - Some routes will favor express routes as desired
 - Some will switch to an express train at a seemingly odd point, or not at all
 - Currently favors express slightly
 - May need more stern evaluation or implementation of heuristic adjustments across different cases
- Accessibility constraints may crash on certain routes.
 - There exists an issue in adding some stations to the priority queue when feature is on
 - Usually occurs when traversing A stations in the Rockaways
 - Priority queue empties before destination is reached.
 - Need to adjust station adding to make sure this never happens.
 - Otherwise, results appear optimal.
- Walking directions are limited to picking the closest station to the API's interpretation of the input address
 - No multiple starting point selection
 - No address interpretation options
- Printing walking directions has a slight formatting issue.

Future Steps

Below is a summary of future steps to be taken if we are to continue working on this project.

- Have algorithm better detect which train to start/end if the user inputs stops with many available trains
 - Currently randomized
- ensure that the user does not need to enter the exact name of desired stops
 - there is presently no way to distinguish between stops with the same station name
 - Ex: "7th Av" is a stop in Brooklyn on the F/G, a different Brooklyn stop on the B/Q, and a Manhattan stop on the B/D/E
 - the current method of relating user-inputted station names to stopIDs (mta.findStop) cannot distinguish between these different stations
 - current solution: allowing keywords (algorithm accepts substrings of station names)
 - potential improvement: use a drop-down GUI?
- Continue Fine-tuning heuristic
- Integrate buses
- Integrate ai-powered walking directions
- Add Timetable
- Add Real-Time Updates
- Resolve A train issues
- Route is sometimes not picking the shortest path
 - Needs confirmation
- Include railroads

Paper Links

Comprehensive Route Planning Paper - <https://arxiv.org/pdf/1504.05140.pdf>

K-d tree paper - https://www3.cs.uic.edu/pub/Bits/TransitGenieDocs/tm_thesis.pdf

More shortest path optimization -

https://www.researchgate.net/publication/221131640_Acceleration_of_Shortest_Path_and_Constrained_Shortest_Path_Computation

<https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/GH05.pdf>

Landmark A* -

<http://www-or.amp.i.kyoto-u.ac.jp/members/ohshima/Paper/MThesis/MThesis.pdf>