THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

# CSCI 6511 – A.I.

## Term Project - Smart Metro Pathfinding

CSCI 6511 Nicholas Capurso nickcapurso@gwmaill.gwu.edu

# Table of Contents

# Introduction

The Metrorail system in Washington, D.C. is probably the most widely used transportation system in the city and this likely also stands for subway systems in other cities, such as NYC. For many new residents or tourists, learning the Metro system may seem intimidating or foreign. For these people, a welcome tool will be one that will guide users on the trains to take and where to transfer in order to get where they want to go. On the other hand, experienced riders may have an idea of what lines are "good" to take, at what times, and may use different paths to get to the same destination based on the current situation. However, experienced riders can only make these decisions based on the knowledge they have at the beginning of the trip – at their entry station – whereas a computer application has the power to make a decision based on a "bird's eye" view of the Metro.

Although there are many applications that attempt to do pathfinding through the D.C. Metro to a desired location, there are many cases where humans would prefer to take a different path than what is typically suggested. Additionally, pathfinding in the Metro is not an easy task. The "best path" between two Metro stations is not always the shortest path; a good algorithm should also consider train delays, station or line closures, and train arrival times. Additionally, people do not like to transfer trains if there is a long wait (or just in general, due to convenience).

As a result, for my AI project I wanted to take a graph-based and algorithmic approach to solving the Metro pathfinding problem. Metro or subway systems, in general, can naturally be represented as a graph and metrics such as train delays or waiting times can be factored into some of the graph-searching algorithms we learned this semester, such as A*. The Washington Metropolitan Area Transit Authority (WMATA) provides an API [1] to request station, line, train, and delay information, but does not provide pathfinding across stations that do not share a line (i.e. paths that require a transfer). In the past, I had worked with the WMATA API in a project that suggested *trivial* Metro paths based on the fact that every line in the D.C. Metro intersects every other line at least once. While this does produce a solution, it is often not optimal.

# Evaluation of Current Offerings

There are many applications that Metro riders may use to do pathfinding to a destination. Some of the preliminary work I did in my project was to look at some of these applications to figure out use cases where they would not perform correctly, or yield an irrational path. In this case, a rational path refers to a path that a human would consider taking under normal circumstances. This preliminary "research" also provided additional motivation for a new type of pathing solution. In this section, I will look at three applications: the WMATA online trip planner [2], Google Maps [3], and the Transit Android app [4].

## Case 1: Foggy Bottom to Columbia Heights

One case I tested was taking the Metro from Foggy Bottom / GWU to Columbia Heights. As shown below, these stations do not share a line and an obvious solution would be to find the intersection between the lines servicing both stations (L'Enfant Plaza) and transfer there. However, the typical path I take (and others I have asked) is to transfer once at Metro Center onto the Red Line, then transfer again at Gallery Place to Yellow or Green, which saves a total of 4 stops, but adds one transfer. This path is also the shortest in terms of distance traveled. In general, this path saves time when if there are not long train delays, and it is conceivable that an application with a Bird's Eye View of the Metro system would be able to suggest the path.


*Figure 1. Foggy Bottom (OR, SV, BL) and Columbia Heights (YL, GR). [5]*

However, in general, the case of recognizing a path that requires the rider to transfer twice is algorithmicly more complex than recognizing a path that requires the user to transfer once (which is just the simple intersection of two lines), assuming that no prior knowledge about the D.C. Metro, its stations, or its lines, are known.

WMATA Website

An example of the trip returned by the WMATA website is shown below, which suggests the single transfer at L'Enfant Plaza. Other possible choices simply include the different lines to take to/from L'Enfant Plaza. Other two-station transfer tests also fail with the WMATA's website's planner.

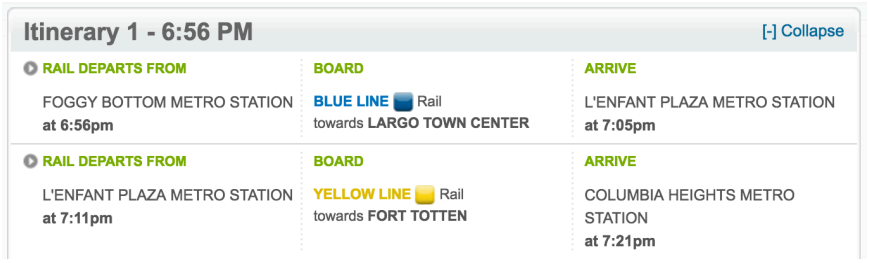
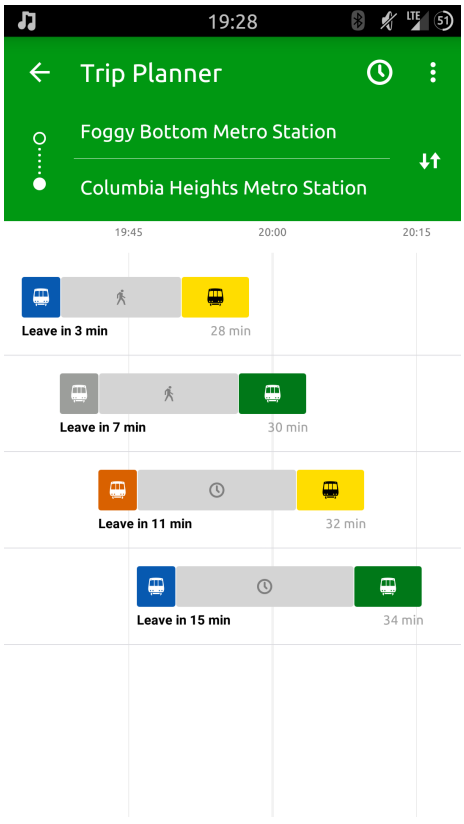



*Figure 2. WMATA Website Results.*

## Transit App

Although the Transit app is concerned with mass transit beyond just the Metro, it still represents one of the most popular applications for Metro pathing on Android's app store. Interestingly, by default, it suggests to take a line to McPherson or Metro Center, then take the bus. When forced to give paths only in terms of the Metro, the results are shown to the left. Interestingly, their algorithm suggests the user to walk the ~7 mins from Metro Center to Gallery Place (both of which lie on the Red Line). In reality, this is not feasible because the user would have to pay twice to use the Metro.




Based on other tests I have done, I have attributed this result to their algorithm being unable to consider a path where the user has to transfer twice.

*Figure 3. Transit App Results.*

## Google Maps

Of the three applications I have surveyed, Google Maps (as expected) performs the best. Google Maps also displays the simple intersection path where the user transfers at L'Enfant Plaza. In the below screenshot, this particular trip had an estimated time of 25 minutes (though different times had ~30 minute estimations), which is shorter than the paths listed in the itineraries of the two previous applications. As mentioned previously, aside from time, this path is also the shortest in terms of distance.

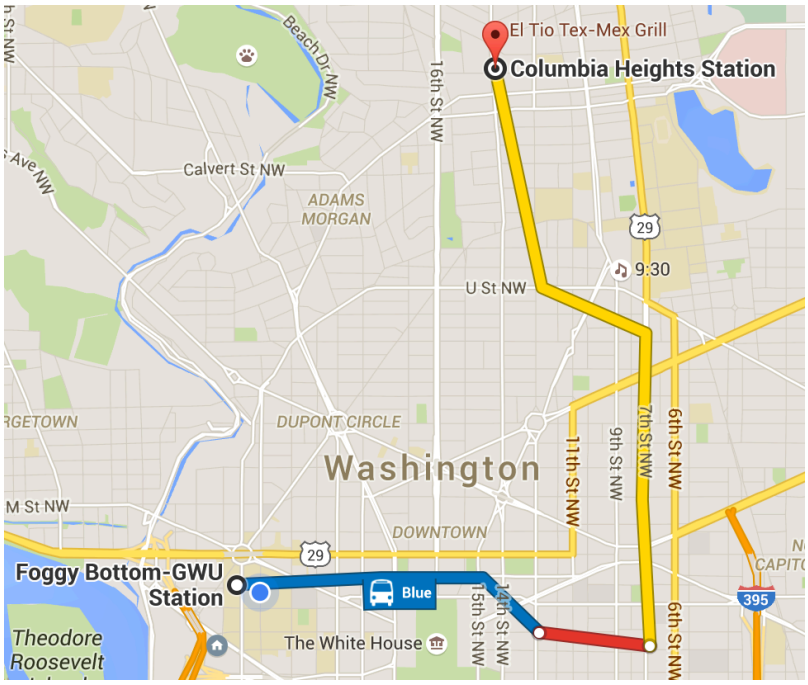



*Figure 4. Google Maps results.*

## Case 2: L'Enfant Plaza to Van Dorn Street

The next case example is taking the Metro from L'Enfant Plaza to Van Dorn Street, in the general case where it is not rush hour (i.e. Van Dorn is not on the Yellow Line). This scenario is interesting because the "direct path" of taking the Blue Line is not the shortest path, in distance nor time. It is faster (Google Maps reports at least 6 minutes) to take the Yellow Line, then transfer to Blue.


*Figure 5. L'Enfant Plaza (YL, GR, OR, SV, BL) and Van Dorn St. (BL, YL (rush-hour)). [5]*

In fact, as I also discovered while programming my application (and will explain later), this case causes two "seemlingly reasonable" general heuristics to fail: 1) that direct paths are always better and 2) a user will never need to transfer at a station where no new lines are presented in relation to the station they are coming from.

To put 2) in a different way: L'Enfant Plaza and Pentagon both are on the Blue and Yellow lines, but taking Blue takes longer than taking Yellow. In most every other case in the D.C. Metro, lines run together, so two stations that lie on the same lines can usually be reached in the same amount of time regardless of which one you take.

## Applications' Performance

Rather than present all three applications separately, the three results are shown below. The WMATA planner and the Transit app only suggest taking the Blue line (likely a fault of taking a direct path when available), while Google Maps correctly suggests the correct Yellow-Blue transfer.

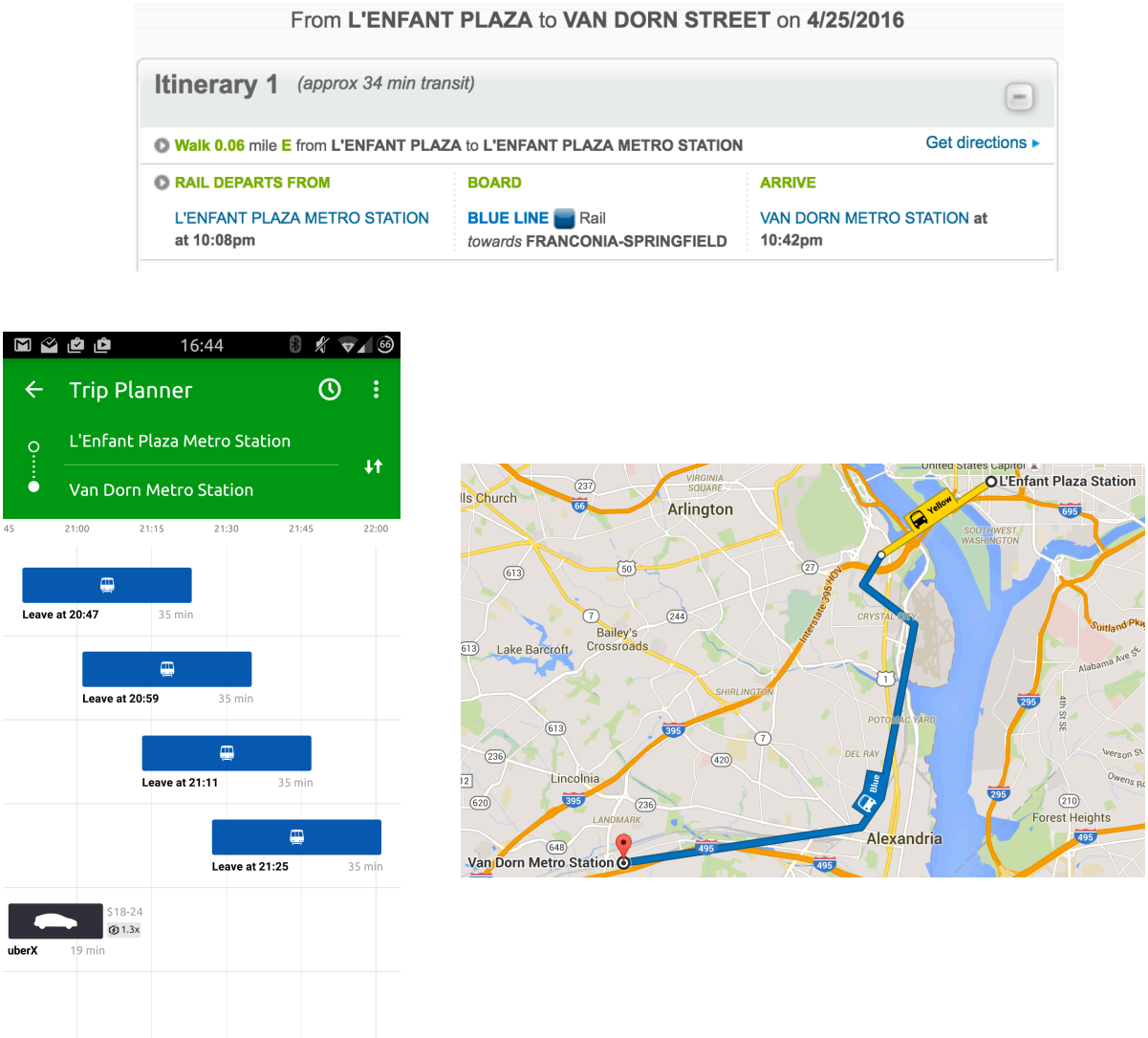



*Figure 6. Results from all three applications.*

If we model the Metro system as an undirected graph and apply an intelligent graph-searching algorithm that keeps track of a "cost traveled so far" metric, such as A* or Uniform-Cost Search, it is likely that the direct Blue Line path would never be chosen as a "best" path.

# Project Design

Given the problems in the current offerings for Metro pathing as presented in the previous section, my project was write an algorithm, using the graph-searching techniques we learned in class, to correctly suggest paths through the D.C. Metro.

## Overview

The core of my algorithm uses the A* algorithm for pathing. Where the "cost so far" (g(n)) is the distance (by track) travelled along the Metro to the *nth* station and the heuristic (h(n)) is the approximate distance from the *nth* station to the goal. Any delays can be factored into *h(n)* by multiplying the time delay by the estimated speed of the train.

Although the algorithm is simple to explain, it is also the result of playing with the WMATA API, the data it can provide, and also with the graphical representation of the Metro system. The application also does not utilize any special information about the Metro system and is written in a generic fashion.

Originally, the plan was to write an Android application that implemented the algorithm, but too much time would have to be devoted to user interface design. As a result, the current implementation is in Python, where the algorithm itself was the majority of the focus. From this point, the code can be converted from Python to a Java implementation for Android.

## WMATA API

Although having a readily available API to easily request information about the Metro was certainly welcome, it imposed limitations on the algorithm design. For one, there is a limit on the number of requests allowed per second (10); and, two, I could only work with what information was provided. The first limitation can be explicitly problematic as any sort of operation that has to make O(n) or $O(n^2)$ API requests will take unacceptably long. Fortunately, there are a few ways within the WMATA API to return information about *all* stations in a single request.

When my project application starts up and collects source / destination information from the user, it hits the WMATA API and builds the Metro graph from scratch – i.e. all the lines and stations are queried – no information is known about the Metro ahead-of-time.

## APIs Used

In order to get a sense for the information I had to work with, the following is a description of the API endpoints that were utilized [1], the information that is returned, and how the information is used in my project. All data is returned in a JSON format.

The **Lines** API is used to query the lines available on the D.C. Metro and what stations they begin and end at. The starting and ending stations are actually significant because they can be used to infer what relative direction you are "travelling" in as nodes are traversed.

The **Station List** API is used to fetch the stations along each line returned by the Lines API. Stations are uniquely identified with "station codes." Extra processing must be done at this stage for stations that have multiple platforms to avoid duplicates in the graph – as these are actually listed as "separate" stations with separate codes. For example, Metro Center has a single platform for the Orange, Silver, and Blue lines and another platform (on the upper level) for the Red Line. As a result, Metro Center has two separate station codes: "A01" for the Red Line and "C01" for the Orange, Silver, and Blue lines. Following this stage, the Metro graph can be constructed using the stations as nodes and the connecting lines as edges.

Interestingly, the Station List API does not return stations in order of their location along their lines, but according to their unique station code. As will be discussed shortly, the ordering of stations is significant in order to connect the nodes of the Metro graph.

The **Station-to-Station** API provides the core of the A* heuristic h(n) as it provides approximate distances and times (by track) from one station to another station and can also provide estimates for all stations to a single station in a single network request. If queried using the starting and ending stations of each line, the result will be the

approximate distance and times, by track, of each line – which can be used to approximate the speed of the trains, as used by my A* heuristic. It is not stated, but the distances and time seem to correspond to the shortest path, in terms of distance.

Next, the **Path Between Stations** API is used. This cannot be used to do pathfinding to stations that do not share a line (otherwise I would not need to do a project). However, for stations that are on the same line, it provides the ordered sequence of intermediate stations. This is signifcant because, if queried using the starting and ending stations of each line, will return the *ordered* sequence of stations for each line – which is cruicial for building the edges of the final Metro graph.

Finally, the **Real-Time Predictions** API can be used to determine train arrival times given a particular station (or, for all stations).

## Developing the A* Algorithm

After the Metro graph had been built, then the task fell to the actual implementation of the A* algorithm which has three core components: g(n), h(n), and the expansion of a node into the fringe/frontier.

### Time vs. Distance

The metric for g(n) and h(n) must agree and the two options I had readily available was to use distance or time on the Metro. As mentioned in the last section, the Station-to-Station API provided distance and time measurements between any pair of stations or from every station to a single station (or, inversely, a single station to every station).

One challenge that had to be overcome was that there was no easy way to keep track of the "cost travelled so far," as determining the distance or time between each pair of stations along a path took far too many network requests and only 10 requests were allowed per second. To compensate, I used the fact that one could use the "every station to a single station" option to determine the distance or time from every station to the *desired destination* station. Therefore, the "cost so far" became the "cost so far, relative to the goal" and the distance/time between two stations was calculated by the difference between their two approximate distance/time to the destination.

To evaluate the feasibility of this direction and to also compare between using distance vs. time, I wrote a script to graph the distances or times from every station on a given line to a destination (i.e. visualizing and analyzing different choices). The graphs below show the distances (left) or times (right) from every station on the Orange line to Van Dorn Street station. Note: the Y-axis starts from 20 on the time graph and the station points begin at 0 on both graphs.

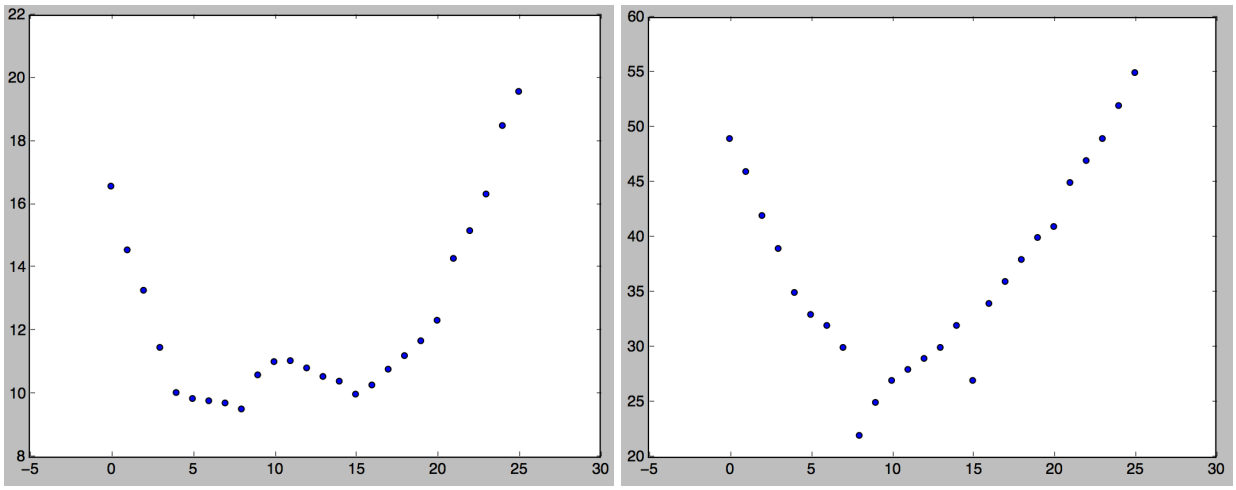



*Figure 7. Distance (left) or Time (right) from every station on the Orange line to Van Dorn St. station.*

The two minima represent the "closest" stations on the Orange line to Van Dorn Street (again, in terms of distance or time by track): Rosslyn and L'Enfant Plaza. Although there are similarities between the two graphs, I have hypothesized that the differences in the time graph are due to the API factoring in transfer times – stations before Rosslyn require a transfer to the Blue line (explains the jump between the 8th and 9th station points) and L'Enfant Plaza's time (16th station) is shorter because of taking the Yellow Line and transfering to Blue.

As a result of this "extra factor" included in the time approximations, I decided to use the **distance** approximations as the base metric for the A* algorithm as it is a more stable and predictable metric, regardless of whether or not a transfer is required.

## G(n) and H(n)

With the distance metric chosen, the calculation of g(n) simply became the distance travelled so far (based on the absolute value of the differences of station distances to the destination).

H(n) became the approximate distance by rail from the *nth* station to the goal. The speed of the train on each line was determined as mentioned in the WMATA API section; any delays at the *nth* station could be multiplied by the speed of the corresponding train and added to the approximate distance. In this way, "waiting" for a train is equivalent to getting on the train instantly, but the overall track distance is longer.

This heuristic allows h(n) to be admissible, or optimistic, since it will be equal to the actual in the case where the user catches their transfer, and will less than the equal in the case where the user misses their transfer, but will never be greater than the actual.

## Defining Node Expansion

In the A* algorithm, the node with the lowest f(n) = g(n) + h(n) is chosen to be expanded and its children are added to the frontier or fringe.

Initially, in an attempt to minimize the amount of computation done, I defined expansion of a node *n* as, "Any station that shares a line with *n* and also lies on a line that is not shared with *n*." In other words, the children of a node are any transfer stations that are reachable. For example, children of Foggy Bottom would include Metro Center (adds the Red line) and L'Enfant Plaza (adds the Yellow and Green lines), but not McPherson Square (adds no new lines). However, as this expansion does allow certain paths to be found successfully, it causes the algorithm to fail (as I found) in cases like the L'Enfant Plaza to Van Dorn Street example, where lines do not run parallel in all cases.

Given this, I refined my expansion function to *any* station that shares a line with the *nth* station and is not already in the frontier (to avoid duplicates). Alternatively, I could allow duplicates in the fringe if the station is being expanded from a different parent –

assuming there was some mechanism to detect graph cycles. Insofar, I have not found a case in the D.C. Metro where using this altenative would yield a better path, but it may be better in general.

Checking for the goal state (the destination) is done following expansion of a node. Since the goal state may be expanded from any node which shares a line with the destination, the final solution is not accepted unless the expanded node is the current best (minimal f(n)), following general usage of the A* algorithm.

## Current Results

The screenshot below shows the algorithm performing under the same test cases which I evaluated the other current offerings under: 1) L'Enfant Plaza to Van Dorn Street and 2) Foggy Bottom to Columbia Heights.

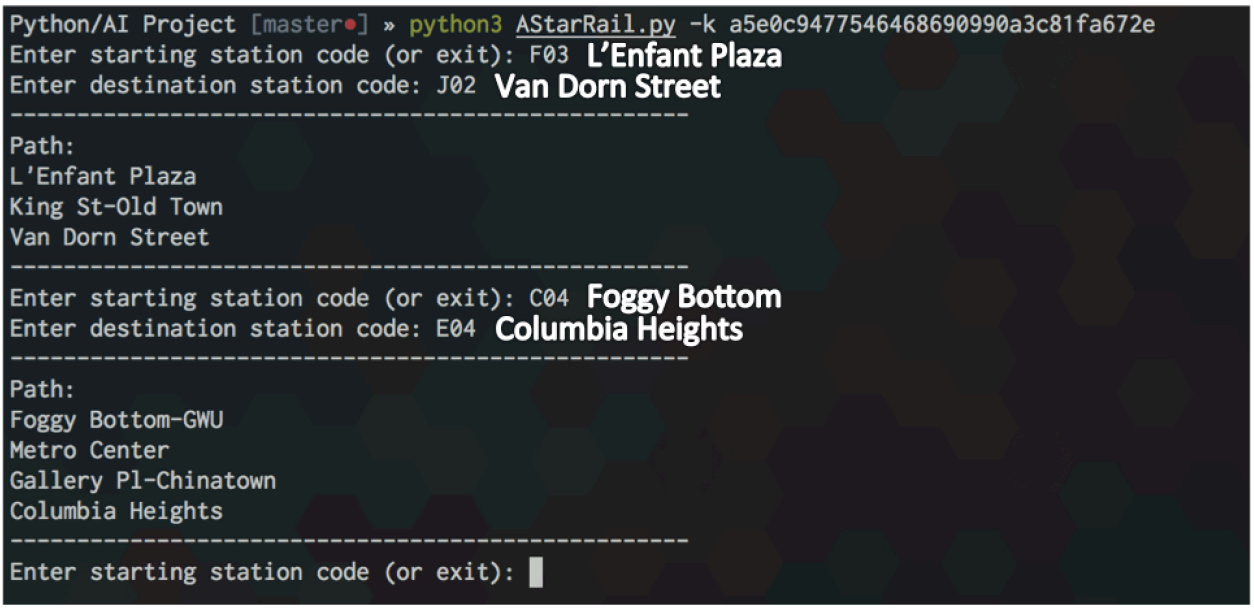



*Figure 8. Project results under the two previous test cases.*

The output is a path of stations **to transfer at** (usually lines can be inferred). In the first test case, the algorithm correctly suggests travelling to King St-Old Town (the Yellow Line), then transferring to get to Van Dorn Street. In the second test case, it suggests the correct two-transfer path: once at Metro Center and then at Gallery Place/Chinatown.

## Future Work & Preliminary Results

At this time, delays and station closures are not yet built-in *dynamically* due to the additional parsing and processing required. However, to simulate this, I can manually "close" stations or lines by altering h(n) of specific stations. In the example below, I run the Foggy Bottom (C04) to Columbia Heights (E04) test again, but "close" Metro Center by increasing its h(n) to an unreasonable amount (ex. the Red Line delayed by 20 minutes):

```
Python/AI Project [master●] » python3 AStarRail.py -k a5e0c9477546468690990a3c81fa672e
Enter starting station code (or exit): C04
Enter destination station code: E04
--------------------------------------------------
Path:
Foggy Bottom-GWU
L'Enfant Plaza
Columbia Heights
--------------------------------------------------
Enter starting station code (or exit):
```

*Figure 9. Foggy Bottom - Columbia Heights with delays on the Red line.*

In this case, the "second best" path is the one in which the user transfers once at L'Enfant Plaza. Finally, as another example, I perform the test again, but "close" both Gallery Place and L'Enfant Plaza (but leave Metro Center open). The results are below:

```
Python/AI Project [master●] » python3 AStarRail.py -k a5e0c9477546468690990a3c81fa672e
Enter starting station code (or exit): C04
Enter destination station code: E04
--------------------------------------------------
Path:
Foggy Bottom-GWU
Metro Center
Fort Totten
Columbia Heights
--------------------------------------------------
Enter starting station code (or exit):
```

*Figure 10. Foggy Bottom - Columbia when Gallery Place and L'Enfant Plaza are closed.*

As shown, in the case where the user cannot transfer to the Yellow or Green line at Gallery Place/Chinatown or L'Enfant Plaza, the next best way to Columbia Heights is to

take the Red line up to Fort Totten to transfer. As a follow up: if we also closed Fort Totten here, the algorithm would then suggest going from Foggy Bottom down to Pentagon to reach the Yellow line.

Further future work would automate this process – parsing delays and warning messages issued by WMATA into a time-delay and lines affected. As mentioned previously, we estimate the speed of each line's trains by the line's length and the overall time for the train to move from end to end. Thus, any delays multiplied by the approximate speed of the train would be factored in to h(n). The same concept goes for the train arrival times at a given station.

Additionally, based on the current results in Python form, it seems promising that the algorithm could be implemented into a full smartphone application.

Finally, the current program works uses Metro station *codes* for the source and destination locations. This is because station codes uniquely identify a station + its platform. In a "real" implementation, I might take a location (GPS or address) and perform a lookup of nearby Metro stations to use.

# Conclusion

The project I worked on for AI was a smarter pathfinding algorithm for the D.C. Metro. Current offerings in Metro pathfinding suggest a need for improved subway pathfinding algorithms and served as one of the main motivations for my project. The Metro system was modeled as a graph and the A* graph-searching algorithm was applied using distance metrics. Results show that the algorithm can correct determine the correct paths and will still operate well when manually applying delays or closing stations outright.

# Appendix A: Running the Algorithm

The requirements to run the project:

- `python3` (Note, the `python` command defaults to `python2`).
- `requests` library for `python3` (can install with `pip3 install requests`)

To run the application, open a terminal and navigate to the directory containing the project and run: `python3 AStarRail.py -k a5e0c9477546468690990a3c81fa672e`

The `-k` argument specifies the API key to use (in this case, mine).

The application will prompt for the starting and ending *codes* of stations. This is because station codes uniquely identify a Metro station + platform. In a full implementation, I would instead use the user's GPS location and automatically determine the closest station.

Commonly used station codes:

| **Station** | **Code** |
|---|---|
| Columbia Heights | E04 |
| Foggy Bottom | C04 |
| Dupont Circle | A03 |
| L'Enfant Plaza | D03 |
| New Carrollton | D13 |
| Pentagon City | C08 |
| Reagan Airport | C10 |
| Tysons Corner | N02 |
| Union Station | B03 |
| Van Dorn Street | J02 |

# References

[1] Washington Metropolitan Area Transit Authority API, "*APIs,*" Internet: https://developer.wmata.com/docs/services/

[2] Washington Metropolitan Area Transit Authority API, "*Trip Planner,*" Internet: http://www.wmata.com/rider_tools/tripplanner/tripplanner_form_solo.cfm

[3] Google Inc., "*Google Maps,*" Internet: https://www.google.com/maps

[4] Transit App, "Transit App," Internet: http://transitapp.com/

[5] Washington Metropolitan Area Transit Authority API, "*Metro System Map,*" Internet: http://www.wmata.com/rail/docs/color_map_silverline.pdf