Optimal Control of Energy Systems – Final Project

Washington Metro Stochastic Dynamic Programming

Authors: Daniel Hearn, Ryan McNeal, Gino Rospigliosi

ENME489A

Instructor: Dr. Hosam Fathy

Date of Submission: 12/21/2021

Table of Contents

Introduction

In this report, the energy management problem of Metro Line Trip Optimization is covered. This optimization problem falls in the category of dynamic programming for transportation scheduling algorithms. The system of interest in this problem is the Washington Metro, a rapid transit system of the United States of America serving the District of Columbia, Maryland, and Virginia.

Figure 1: Washington Metro System Map [4]

The goal of this problem is the minimization of time between stations and consequently the power consumed by each trip. This system is initially reduced to a subset of the Washington Metro, which accounts for all of the paths between College Park station and Tysons corner. Intermediate stops from the superset of the model are not included to reduce computational complexity while still demonstrating critical functionally through deterministic dynamic programming. The model is then extended to Stochastic Dynamic Programming where the full set of the Washington Metro stops are used. Trip times

between all critical points such as Fort Totten, Gallery Place, etc were compiled using the Washington Metro Trip Planner (Washington Metropolitan Area Transit Authority 2021).

The inputs to this model are the dynamic input of an initial and final state representing stations. The outputs to this model are a discrete solution of transition states that correspond to an optimal metro route, accounting for each available path and its transitions. The important dynamics of the system are the time between states, the transition between states, and stochastic variables such as delays.

Relevant tools from literature covered in this class of optimal control relate to dynamic programming. Other optimization tools such as KKT conditions and extremum seeking control are methods more difficult to formulate due to the structure of data intended to optimize and would require more effort and a reinterpretation of the data to be suitable. A relevant principle that is used to formulate this optimization is Bellman's principle of optimality. This principle states that every sub trajectory of an optimal trajectory must be optimal and guarantees a derivation for an optimal plan from the initial state, x_k to the final state x_N. Considered tools for this optimization included the Value-Iteration method, Policy-Iteration method, and Dijkstra's Algorithm.

The first optimal control tool considered is the Value-Iteration Method to find the optimal plan from initial station state x_k to final station state x_N .

$$\text{Minimize } \Sigma_k^N \ \{ L(x_k , u_k) + \phi(x_k)\}$$
$$G^*(x) = \min_u \{ L(x_k , u_k) + G^*(f(x_k , u_k) \}$$

In this algorithm, L denotes a transition cost of the state vector x_k and input variables u_k. To initialize the algorithm, the terminal state condition x_N and terminal cost $\phi(x_N)$ is given for a terminal state. Then, G*(x) recursively minimizes the optimal cost-to-go from through backward value iteration. The issue with this approach is the curse of dimensionality. First introduced by Richard Bellman, this states the number of samples needed to estimate an arbitrary function with a given level of accuracy grows exponentially with respect to the number of input variables. Assessing this algorithm, the time Computational Complexity $O(d^n e^m N)$, where d is the number of discretizations per state, n is the number of state variables, e is the number of discretizations per input, m is the number of input variables, N is number of time steps. Similarly to the value-iteration method, the policy iteration method which is covered later in this report, works by iteratively updating cost-to-go values on the state space in which the choice of possible plans has been restricted to a single plan. As a result of this feature and qualities of the Washington Metro, a different optimization approach is taken.

The next optimal control method is Dijksta's Algorithm. This algorithm finds the minimized path between initial station state x_k to final station state x_N in a graph data- structure.

function Dijkstra(Graph, source):

create vertex set Q
for each vertex v in Graph:
    dist[v] ← INFINITY
    prev[v] ← UNDEFINED
    add v to Q
    dist[source] ← 0

    while Q is not empty:
     u ← vertex in Q with min dist[u]

     remove u from Q

```
for each neighbor v of u still in Q:
        alt ← dist[u] + length(u, v)
if alt < dist[v]:
dist[v] ← alt
        prev[v] ← u
return dist[], prev[]
```

This algorithm is analogous to the DDP Value-Iteration method, where Dijkstra's Algorithm minimizes the optimal cost-to-go G*(x) weighted edges through a forward value iteration in a nonlinear data structure. The algorithm initially sets the initial node state cost to zero and unvisited node states as infinity. Then, the set of visited vertices is checked recursively while updating the cost-to-go G*(x) to find the minimized cost path. This algorithm is chosen over other algorithms because of the lower time computational complexity $O(V^2)$ using an adjacency list implementation and where V is the number of vertices.

Formulation

Input Variables (x_k: Next Station and Line)

The input variables, u_k, are the next station and line to take. Other important parameters affecting sequential inputs are the metro line transition time (min), the time to get from one station to another, the delay at each station (min), and the delay for each line (min).

State Variables (x_k: Stations and Line Color)

The state variables, x_k, are each individual metro station and their corresponding lines that pass through them. For example, the College park station has two lines passing through it, yellow and green. Each of these are their own individual state variables. These variables are represented as a node or vertex of a graph structure where transition costs are represented by weights. Below is a set of the state variables modeled in our optimization problem.

DDP

We first modeled this DC Metro environment in one of our class application challenges. The goal was to create a program that would find the shortest distance between two stations by using discrete dynamic programming. By using a modified version of Dijkstra's algorithm, the program iterated through each station, gathering information on the cost to go from the starting station to the end. The environment creates a unique challenge when modeling since two stations can be connected by multiple metro lines. This means that each state in the environment is represented by a state and line variable. The inputs are then the next station you will travel to, and by which line you will travel. This representation gave us the following node network:

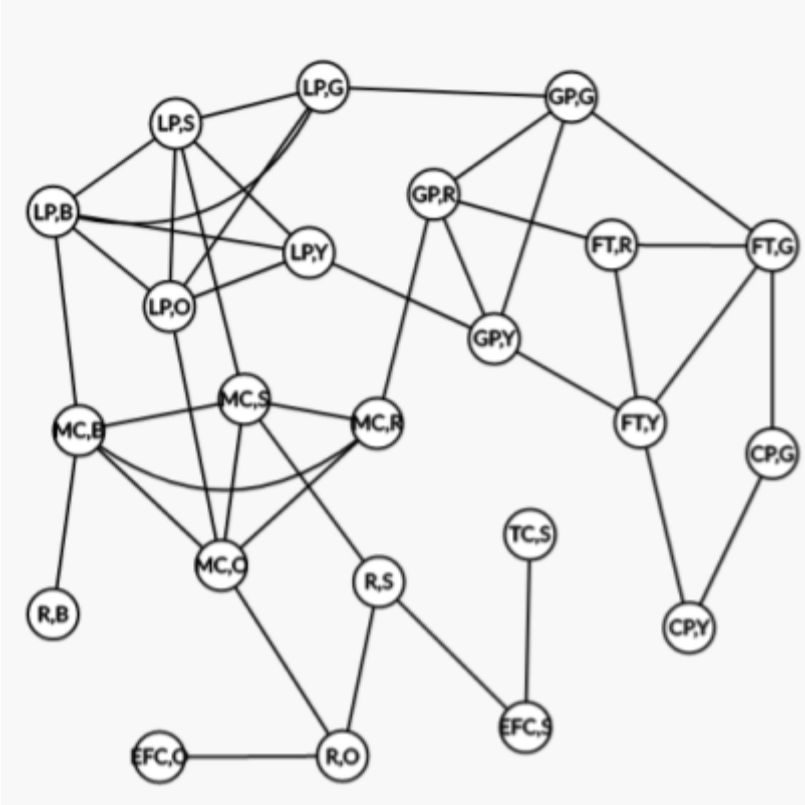

Figure 2: Simplified Graph used for first Application Challenge

For sake of time, we only implemented eight stations while the actual DC metro has 91. To replicate random delays of different lines, we randomly assigned delays to each metro line which were set at the beginning of the program. We also added additional delays when the program changed lines along with random delays at each station, replicating the time it would take to change from one train to another rather than just staying on. Using these state variables and additional constraints we were able to define the environment's objective function:

Minimize Σ_k^N { Cost(Station$_k$,t_{k+1})+ Transition_Cost(Station$_k$, $t_{k,\text{Station Delay}}$, $t_{k,\text{Line Delay}}$) }

Subject To:

Station$_0$ = Starting Station, Station$_N$ = Final Destination

Random Time Delay of Metro Line: $0 \leq t_{\text{Line Delay}} \leq 20$

Random Time Delay at Station$_k$: $5 \leq t_{\text{Station Delay}} \leq 15$

if(Transitioning lines): Transition_Cost = $t_{k,\text{Station Delay}} + t_{k,\text{Line Delay}}$

else: Transition_Cost = 0

Using Python and Jupyter Notebooks, we were able to implement this problem and see how the program worked with our set of stations. The result was a low-level model of the DC Metro, which outputted the steps to get from one station to another while minimizing the cost of travel.

```
Line: Green, Line Delay: 14
Line: Red, Line Delay: 5
Line: Silver, Line Delay: 7
Line: Orange, Line Delay: 12
Line: Blue, Line Delay: 2
Line: Yellow, Line Delay: 10
```

```
Starting Station: College Park, Line Taken: Yellow, Cost: 0
Starting Station: Fort Totten, Line Taken: Yellow, Cost: 30
Starting Station: Gallery Place, Line Taken: Red, Cost: 63
Starting Station: Metro Center, Line Taken: Red, Cost: 85
Starting Station: Rosslyn, Line Taken: Blue, Cost: 105
Starting Station: East Falls Church, Line Taken: Silver, Cost: 133
Starting Station: Tysons Corner, Line Taken: Silver, Cost: 162
```

```
Staion: College Park, Station Delay: 15
Staion: Fort Totten, Station Delay: 11
Staion: Gallery Place, Station Delay: 14
Staion: L'Enfant Plaza, Station Delay: 5
Staion: Metro Center, Station Delay: 15
Staion: Rosslyn, Station Delay: 9
Staion: East Falls Church, Station Delay: 9
Staion: Tysons Corner, Station Delay: 13
```

Figure 3: Program Outputs from First Application Challenge

While this model does a great job of finding our desired path, it is far from an accurate representation of the DC Metro system. First of all we are only using 8 of the 91 stations. For our next model we will use all 91 stations alongside times to go between each station and line. Additionally, our use of random delays are far from how real Metro Delays occur. Finally, our model does not account for potential probabilities of switching in between train cars. By taking into account these various lessons learned from our first application challenge we set out to create a new model which created a more accurate representation of the DC Metro system.

Research

Through our research we were unable to find examples of dynamic programming used to optimize metro trips. However, we were able to find a few key sources which identified the information we needed to create our own. In 2020, Yap and Cats published a study about predicting disruptions in the metro system and their corresponding delays that are caused. They used the DC metro line as their case study, making the analysis extremely relevant to our project. They Identified five categories of disruptions: Rail Car, Operations, Public, Infrastructure, and Other, each with different types of disruptions within. Figure 4 below shows the disruption breakdown.

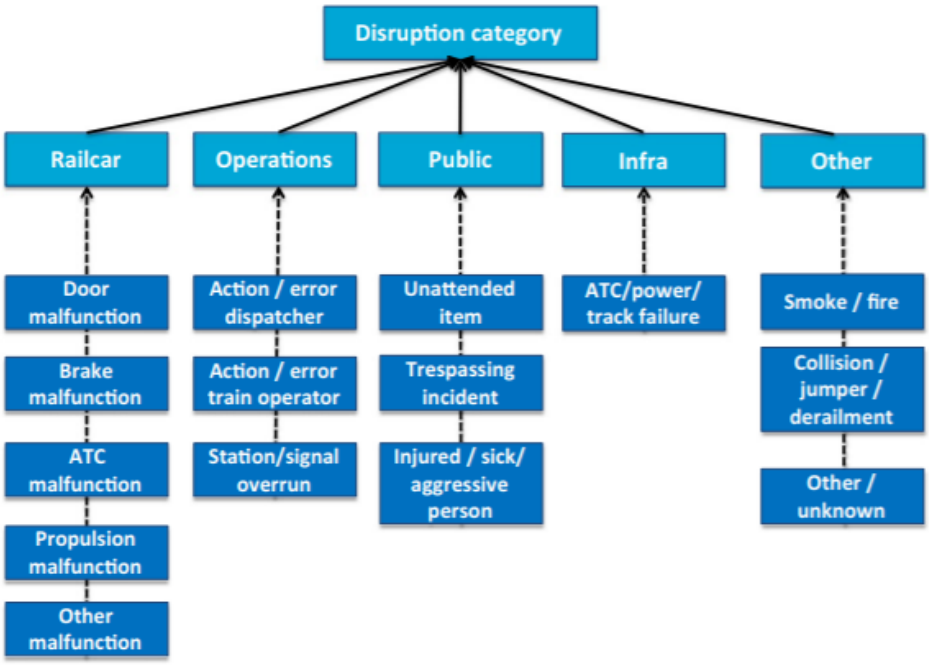

Figure 4: Disruption Categories

WMATA provided them with disruption log data, timetable data about train frequencies and passenger journey times from September 2017 to March 2018, a 6 month period. They were able to use this information to break down the frequencies of each type of disruption and disruption category. This information was utilized in our model to decide the relative frequency of disruptions that will occur.

Coincidentally, Cats was an author in another one of our research articles alongside Van Lint and Krishnakumari. Their goal was to identify patterns in day to day and seasonal passenger delays on the DC metro system. They utilized a years' worth of passenger smart card data between August 2017 to August 2018. During this period, the DC metro had an average of 438,000 rides per day with an average travel time of 28 minutes. In addition, 14% of the passengers experience delays longer than 6 minutes. Cats et al used the data from this period to map delay distributions and match them by month and day of week. For example, during weekdays they found strong peaks of delays during rush hours (Around 8am and 5pm).

WMATA's website page itself had useful information about the DC metro system's operations. One of the most interesting pieces of information we found is that each line had a different frequency of rail cars passing through. The Red line has cars operating every 12 minutes, Green and Yellow every 20 minutes, and Silver/Orange/Blue lines operating every 30 minutes. We will utilize this information in our transition probabilities to simulate how close the connecting lines are at any given stop.

Our SDP Program

We started by adding in a probability that the program will change in between stations. This probability is determined based on the last line that the program takes, and gives the highest probability that the rider will stay on the track instead of switching lines. The probability that the rider stays on the same line is 50%. From there, the rest of the lines are given diminishing probabilities based upon the frequency in which trains run through that station. As mentioned before, the Red line runs every 12 minutes, the Green and Yellow lines run every 20 minutes, and the Orange, Blue, and Silver lines run every 30 minutes. Using these frequencies, we can build a transition probability table for every station.

For example, say the rider comes into Metro Center station on the Orange line, and we need to look at the probability of transitioning to the Silver, Blue, or Red line, or even staying on the Orange. Based on our model, the probability for staying on the Orange would be 50%. The next highest probability would be transition to the Red line, with Silver and Blue with the same lowest probability. These remaining probabilities are determined off of weights. The Red line is 3 times more likely than the Silver, Orange, and Blue lines, and the Green and Yellow lines are 2 times more likely than the slowest triplet of lines. Again looking at Metro Center, this means the transition probability from Orange to Orange would be 50%, from Orange to Red would be 30%, from Orange to Blue would be 10%, and from Orange to Silver would be 10%. This representation of transition probability is not perfect, but does a fair job of replicating the decision to switch lines based on train frequency. The next addition to the previous model includes realistic delays. As mentioned previously, there are various percentages that represent the various causes for metro delays. We use these percentages to determine what type of delay is affecting each line. This is done by generating a random number between 1 and 100. Based on the number which is randomly selected we can select the delay. For example if 27 was generated, we would assign the Railcar Malfunction delay since 27 is in between 1 and 45. We also use a random number generator to assign the actual delay time as well. Since each delay has an associated time range, we allow the number generator to pick a value between the delay upper and lower bound. The issue is that not every line is delayed every time you take a trip on the metro. We used our intuition to assume that at least 1 line is delayed per trip, with the chances of an additional line being delayed decreasing with each new line. We again use a random number generator to select the number of lines that are delayed every time the program is run. These delays are applied to each line, and are added as a negative value when the value of each state is calculated.

With the delays and transition probabilities set, we are able to go through the program. Since we included transition probabilities for our improved model, we are not able to use discrete dynamic programming anymore, and must use stochastic dynamic programming. This is done through the use of a process called policy iteration. The overall theory behind stochastic dynamic programming is to store values at each state which represent how favorable the state is when compared to the end goal in the environment. The program iterates through each state, determining these stored values. When the program is done running, these stored values can be compared to determine the optimal path from any point to the end state. The process of policy iteration contains three main steps, the initialization, policy evaluation and policy improvement, and can be seen in the following graphic.

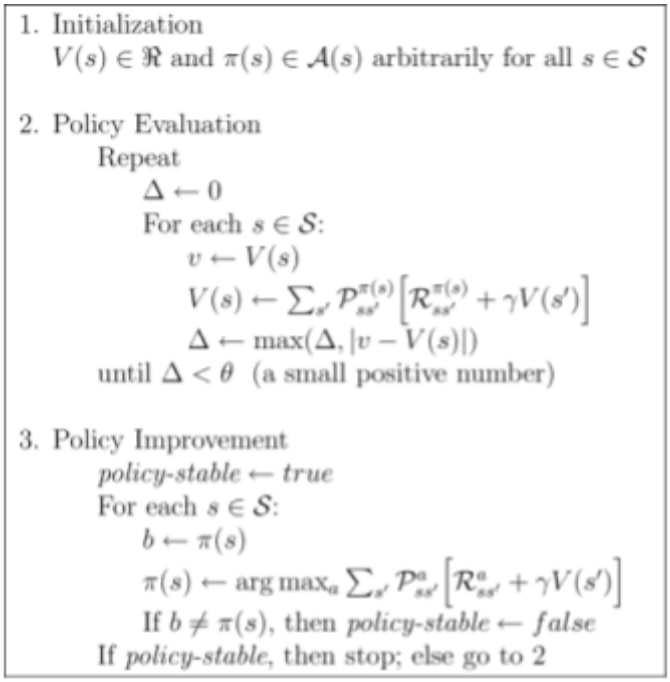

Figure 5: Policy Iteration Algorithm

The initialization step goes through and sets the V, or stores value of each state, to an arbitrary value. This will be changed and updated as the program iterates through each step. A policy is also initialized to arbitrary actions. The policy is the set of best actions based upon the current state. The policy evaluation iterates through the entire list of states and determines the V by summing the transition probabilities to go to each adjacent state multiplied by the reward to transition to that adjacent state added to the V of that adjacent state. In our environment, the transition cost, or reward, is the time it takes to go from one station to another. The end state, or destination, is given a positive reward of 10, while all other transition costs are negative values of the time to travel. This means that as the program iterates through and determines the V of each state, the program will be penalized for making additional stops. In addition, there is an extra -5 that is added to the reward if a line is changed. This accounts for any additional time that it would take to change from one line to another which could be used in transit on a line. Before this V value is calculated, the previous V of the state is stored. After the new V is found, the difference of these two V values are taken, and compared to a set delta value. If the difference of V values is larger than the delta, the program continues to loop. What this does is make sure that the current policy is thoroughly being used across the environment. When the difference in V values is lower than the set delta, it can be assumed that the program iterates through the state variables enough for the V values to converge. Once this convergence occurs, then the policy evaluation step takes place. The policy evaluation once again iterates through each state but this time finds the max V value of each state for each action that is possible. The max V will represent the best possible action for that state. Once this max is found, it will be compared to the current policy for that state. The policy is changed to this new max V, so that the policy represents the best action to take at every step. If the policy that is stored for every state does not match to max V that is found, the entire process starts from step 2. This process of evaluation and improvement creates a loop that results in a set of policies that represents the best possible action throughout every state of the environment. Running through policy from College Park to Metro Center gives the following moves:

```
Start Station and Line:  College Park   Yellow
Next Station and Line Prince George's Plaza   Green
Next Station and Line West Hyattsville   Green
Next Station and Line Fort Totten   Green
Next Station and Line Brookland-CUA   Red
Next Station and Line Rhode Island Ave   Red
Next Station and Line Noma-Gallaudet U   Red
Next Station and Line Union Station   Red
Next Station and Line Judiciary Sq   Red
Next Station and Line Gallery Place   Red
Next Station and Line Metro Center   Red
Destination
```

Figure 6: Program Output from College Park to Metro Center

The output from policy iteration is shown here:

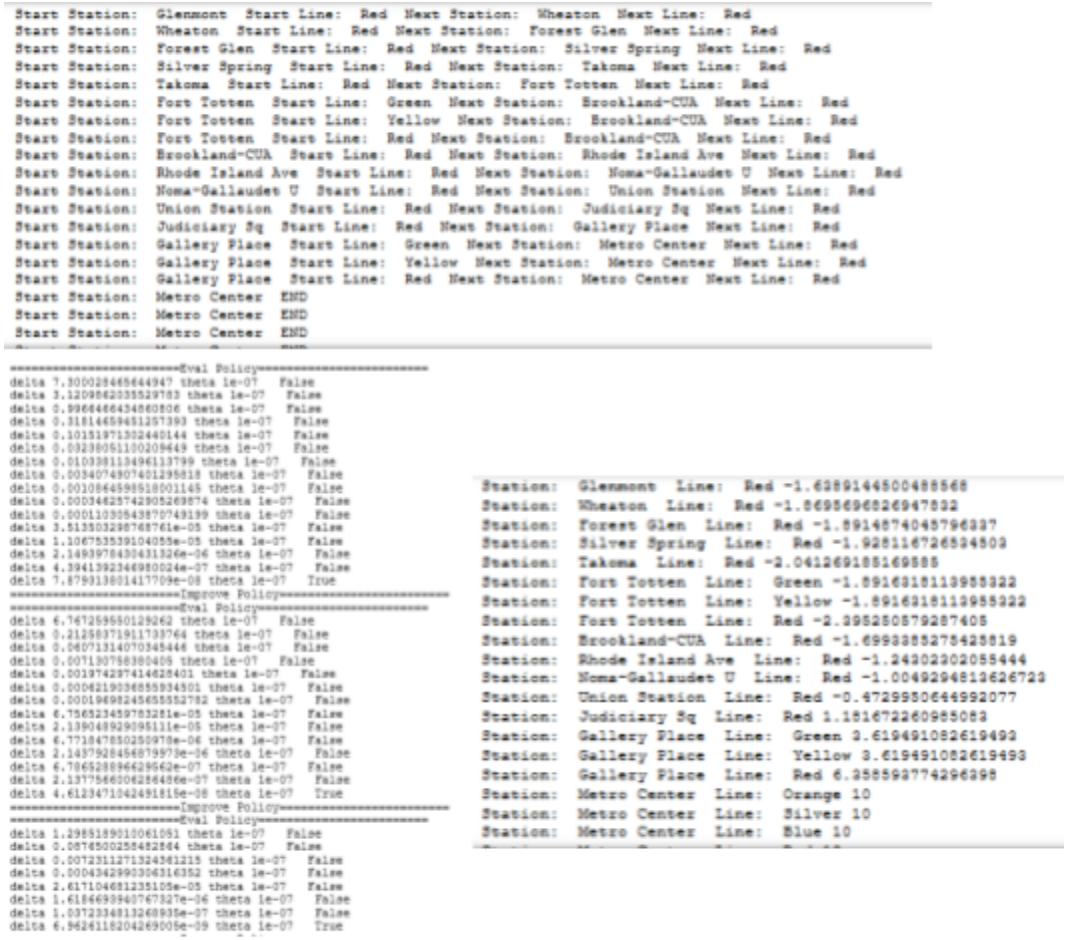

Figure 7: Program Outputs of Training Deltas, V values, and Example Policies

Again, using jupyter notebooks and python we were able to successfully implement this model. The code is attached as a text file at the end of the document to look at. Using a similar structure as the previous model, the environment uses a series of objects to represent the stations, lines, and overall graph.

The station holds the name, list of lines, and transition probability. The delay of each line is also stored in the line class. Giving the program a destination of Metro Center, the output is a list of all the V values for each state. The program is also given a delta value of 1e-7 and a gamma value of .9. The gamma value represents how much weight the next state V has on the current state V value. The 1e-7 value was selected after numerous trial and errors. If the delta was too high, the V values would not converge enough. If the V values were too low then the program would not be able to converge. Because of the complexity of the overall environment, there were quite a few challenges when developing the program. First was developing the code architecture. Trying to design the multiple objects where each station has the accurately stored data which interacts correctly with all the other stations took a lot of time and patience to figure out. Next, was implementing the policy iteration. Most examples of policy iteration use simple environments such as a 4x4 grind where the "rider" picks where direction in the 4x4 grind to move. These examples are quite similar, and it was hard to transfer the implementation policy iteration in those examples to the larger and more complex environment which we built. The last major challenger was debugging the code to make sure it was functioning properly. The code is still not 100% functional, and has a hard time finding the best path every time the code is run. The reason this occurs is still unknown, but the result is an infinite loop where the current V State is higher than the V state of the next station on the connecting line. This means that the rider will switch between two stations continuously without continuing down the track. The solution would be to continue debugging the code to determine the source of this issue.

Further Improvements

Debugging and optimization of the code will be a continuous process as we continue development. There are also several improvements and additions we would like to add in the future. First of which is a more accurate representation of the frequency and range of disruption delays. Currently we have a random number generator deciding how many rails will be disrupted and given a random delay from a given range. This means that there is a strong possibility that all 6 lines will have a disruption, something that is incredibly unlikely to occur in actual operations. Implementation of an accurate rate of disruptions for a passenger would make our trip prediction closer to real scenarios. In addition, we currently have a somewhat arbitrary range of time delay for each of our disruptions. More information on the distribution of delays that occur from each disruption would make our model much more realistic. The delays caused by disruptions are just one factor we implemented in our model. We would like to add other external factors such as delays correlated with the day of the week and time of day. As mentioned previously, traffic is much heavier during rush hour times during weekdays. Similarly, Saturdays have the highest number of riders per day. Ideally, we would include these factors into our model and at the beginning of the scenario we would be able to select our starting/ending station as well as the starting time of day and the day of the week. Lastly, we wanted to implement a time dependent disruption. This was something we discussed early on in the project, but were unable to fit it in given the time constraints. We envision that when our program determines if disruptions will occur for a given line, it will also decide at what time it will occur. For example, the program would randomly decide that a railcar malfunction will occur on the red line 15 minutes after the beginning of our trip. The program would then utilize the red line normally until the trip time had accumulated to 15 minutes and then would consider the delays in its optimization objective.

Conclusion

The work completed on our DDP application challenge laid the foundation for our stochastic model. It began as a simplified model of the DC metro system containing only a fraction of the stations using static, randomized variables to determine the optimal path from one station to another. This model was expanded into a stochastic model, including all 91 stations in the DC metro system and incorporated research-based random disruptions which are considered during optimal trip planning. The SDP program

uses railcar frequencies as the state transition probabilities, the colors operating more frequently having a higher probability of transition. The program determines the optimal path through a policy iteration process. The results in Figure 6 show the optimal path that was determined between College Park and Metro Center stations. Unfortunately, there are bugs in the code that create infinite loops in some scenarios. We plan to continue debugging the code and adding in some external delay factors in the future to make a more reliable and realistic program.

Code

SDP Final Code.pdf

References

[1] De-Yu, Chao. “Fundamentals of Reinforcement Learning: Value Iteration and Policy Iteration with Tutorials.” Medium, May 12, 2021. https://levelup.gitconnected.com/fundamentals-of-reinforcement-learning-value-iteration-and-policy-iteration-with-tutorials-a7ad0049c84f.

[2] Dennybritz. “Reinforcement-Learning.” GitHub, n.d. https://github.com/dennybritz/reinforcement-learning/blob/master/DP/README.md.

[3] Krishnakumari, Panchamy, Oded Cats, and Hans van Lint. “Day-to-Day and Seasonal Regularity of Network Passenger Delay for Metro Networks.” arXiv.org, July 7, 2021. https://arxiv.org/abs/2107.14094.

[4] “Maps.” Washington Metropolitan Area Transit Authority, n.d. https://www.wmata.com/schedules/maps/.

[5] “Metrorail Ridership Grew by 20,000 Trips per Weekday in 2019.” Washington Metropolitan Area Transit Authority, n.d. https://www.wmata.com/about/news/2019-Metrorail-ridership.cfm.

[6] “Navigating in Gridworld Using Policy and Value Iteration.” Data Science Blog: Understand. Implement. Succed., January 10, 2020. https://www.datascienceblog.net/post/reinforcement-learning/mdps_dynamic_programming/.

[7] piyush2896. “Policy Iteration from Scratch in Python.” GitHub, n.d. https://github.com/piyush2896/Policy-Iteration.

[8] “Trip Planner.” Washington Metropolitan Area Transit Authority, n.d. https://www.wmata.com/schedules/trip-planner.

[9] Yap, Menno, and Oded Cats. “Predicting Disruptions and Their Passenger Delay Impacts for Public Transport Stops.” *Transportation* 48, no. 4 (2020): 1703–31. https://doi.org/10.1007/s11116-020-10109-9.

OCES Final Project Code: DC Metro Stochastic Dynamic Programming

Dan Hearn, Ryan McNeal, Gino Rospigliosi

ENME489A:Dr. Hosam Fathy

Code written from scratch, with inspiration taken from mentioned code references in paper

In [38]:
```
import random
import pandas as pd
from datetime import datetime
random.seed(datetime.now())
```

Loading DataFrame with Data

In [39]:
```
df = pd.read_excel('Tracks.xlsx',nrows = 282)
df
```

Out[39]:

	From	To	Rail	Time
0	Glenmont	Wheaton	Red	-3
1	Wheaton	Forest Glen	Red	-3
2	Forest Glen	Silver Spring	Red	-3
3	Silver Spring	Takoma	Red	-3
4	Takoma	Fort Totten	Red	-3
...	...	...	...	...
277	Ronald Reagan	Crystal City	Blue	-2
278	Braddock Rd	Ronald Reagan	Blue	-5
279	King St-Old Town	Braddock Rd	Blue	-2
280	Van Dorn St	King St-Old Town	Blue	-6
281	Franconia-Springfield	Van Dorn St	Blue	-6

282 rows × 4 columns

Creating Objects

Station Object

In [40]:
```
class Station:

    def __init__(self, name):
        self.name = name
        #Connection stations
        #Stored as (next station, time, line)
        self.adj_stations = []
        #Variable used to store station lines
        self.existing_lines = set()
        #Dict storing each transition prob for each line
        self.prob = dict()

    #Adds connecting station
    def add_edge(self,station,weight,line):
        self.adj_stations.append([station,weight,line])
        self.existing_lines.add(line)

    def set_prob(self,prob):
        self.prob = prob

    def ret_name(self):
        return self.name
```

Line Object

In [41]:
```
class Line:

    def __init__(self,name):
        self.delay = 0
        self.delay_type = None
        self.name = name

    #Sets random delays for each line
    def set_delay(self):
        number_delay = [("Rail Car Malfunction",15,60),
                        ("Public", 1,10),
                        ("Operations",5,15),
                        ("Infrastructure",15,30),
                        ("Other",60,90)]
        weights = (45,27,18,6,4)
        (self.delay_type,low,upp) = random.choices(number_delay,weights,k=1)[(
        self.delay = random.randint(low,upp)
```

Graph Object

In [42]:

```
#Object which houses environment
class Graph:
    def __init__(self):
        self.stations = []
        self.names = []
        self.line_lst = []

    def add_stations(self,station):
        self.stations.append(station)

    def ret_station(self,name):
        for i in self.stations:
            if name == i.name:
                return i

    def has_station(self,name):

        if self.ret_station(name) == None:
            return False
        else:
            return True

    def has_line(self,name):

        if self.ret_line(name) == None:
            return False
        else:
            return True

    def ret_line(self,name):
        for i in self.line_lst:
            if name == i.name:
                return i

    def add_line(self,line):
        self.line_lst.append(line)

    #Selects random lines to set delay
    def add_delay(self):
        delay_lines = random.choices(self.line_lst,weights = (1,1,1,1,1,1),k =
        for line in delay_lines:
            line.set_delay()
```

Reading in Data and Building Map

In [43]:
```
#Reads in DataFrame and creates stations and lines
def start_alg(end):
    graph = Graph()
    for index, row in df.iterrows():
        if row['To'] == end:
            r = 10
        else:
            r = row['Time']

        s = graph.ret_station(row['From'])

        if graph.has_line(row['Rail']):
            line = graph.ret_line(row['Rail'])

        else:
            line = Line(row['Rail'])
            graph.add_line(line)

        if graph.has_station(row['From']):

            graph.ret_station(row['From']).add_edge(row['To'],r,line)

        else:

            station = Station(row['From'])

            station.add_edge(row['To'],r,line)
            graph.add_stations(station)


        for station in graph.stations:
            station.set_prob(create_prob(graph,station.existing_lines))


    return graph
```

Station Delays

In [44]:
```
delay_df = pd.DataFrame(columns=["Station","Line","To Station","Delay Type"])
for s in graph.stations:
    for (a,time,line) in s.adj_stations:
        temp_df = {'Station': s.name, 'Line': line.name, 'To': a, 'Delay': tir
        delay_df = delay_df.append(temp_df, ignore_index = True)
delay_df
```

Out[44]:

	Station	Line	To Station	Delay Type	Delay	To
0	Glenmont	Red	NaN	None	-3.0	Wheaton
1	Wheaton	Red	NaN	None	-3.0	Forest Glen
2	Wheaton	Red	NaN	None	-3.0	Glenmont
3	Forest Glen	Red	NaN	None	-3.0	Silver Spring
4	Forest Glen	Red	NaN	None	-3.0	Wheaton

	Station	Line	To Station	Delay Type	Delay	To
...	...	...	...	...	...	...
276	Federal Center SW	Blue	NaN	None	-2.0	Capitol South
277	Vienna	Orange	NaN	None	-4.0	Dunn Loring
278	Addision Rd	Silver	NaN	None	-3.0	Morgan Blvd
279	Addision Rd	Blue	NaN	None	-3.0	Morgan Blvd
280	Franconia-Springfield	Blue	NaN	None	-6.0	Van Dorn St

Transition Probabilities

Generating Prob

In [45]:

```
#Function which generates custom prob for each station
def create_prob(graph,station_line_lst):
    prob = dict()
    num_of_stations = len(station_line_lst)
    line_dict = {'Red':3,'Green':2,'Yellow':2,'Orange':1,'Blue':1,'Silver':1}
    line_post = {'Red':0,'Green':1,'Yellow':2,'Orange':3,'Blue':4,'Silver':5}
    prob_sum = 0

    for line in station_line_lst:

        prob_sum += line_dict[line.name]
        tmp_prob = []
        for l in line_dict.keys():
            tmp_prob.append(0)
            prob[line] = tmp_prob

    for line in station_line_lst:
        for l in station_line_lst:
            if l.name == line.name:
                prob[line][line_post[line.name]] = .5
            else:
                prob[line][line_post[l.name]] = .5*(line_dict[l.name]/(prob_su
    return prob
```

Example Probabilities

In [46]:

```
prob_df = pd.DataFrame(columns=["Station","Line","To Line","Prob"])

line_post = {'Red':0,'Green':1,'Yellow':2,'Orange':3,'Blue':4,'Silver':5}

for s in graph.stations:
    for pr in s.prob.keys():
        prob = s.prob[pr]
        for line in line_post.keys():
            temp_df = {'Station': s.name, 'Line': pr.name, 'To Line': line, 'P
            prob_df = prob_df.append(temp_df, ignore_index = True)
prob_df
```

Out[46]:

	Station	Line	To Line	Prob
0	Glenmont	Red	Red	0.5
1	Glenmont	Red	Green	0.0
2	Glenmont	Red	Yellow	0.0
3	Glenmont	Red	Orange	0.0
4	Glenmont	Red	Blue	0.0
...	...	...	...	...
907	Franconia-Springfield	Blue	Green	0.0
908	Franconia-Springfield	Blue	Yellow	0.0
909	Franconia-Springfield	Blue	Orange	0.0
910	Franconia-Springfield	Blue	Blue	0.5
911	Franconia-Springfield	Blue	Silver	0.0

912 rows × 4 columns

Helper Functions

In [47]:

```
#Sets all V states to 0
def set_states():
    states = []
    for station in graph.stations:
        for line in station.existing_lines:
            states.append((station,line))

    v_station = dict()
    for s in states:
        v_station[s] = 0
    return states, v_station
```

In [48]:

```
#Used to generate V for a passed in state
def get_v(v_station,graph,station,line,gamma):
    if gamma > .1:
        line_post = {'Red':0,'Green':1,'Yellow':2,'Orange':3,'Blue':4,'Silver
        tmp = 0
        pr = station.prob[line]
        for s,r,l in station.adj_stations:
            tmp += pr[line_post[line.name]]*(r+(gamma*v_station[(graph.ret_sta
        return tmp
    else:
        return 0
```

In [49]:
```
#Creates end
#Makes sure that the end state is an abosorbing state
def set_end(v_station,name,graph):
    for s in graph.stations:
        for s_,r,a in s.adj_stations:
            if s_ == name:
                r = 20
    station = graph.ret_station(name)
    station.adj_stations = []
```

In [74]:
```
#Prints out V for each state
def print_v(v_station):
    itter = 0
    for (s,l) in v_station.keys():
        print("Station: ", s.name, " Line: ", l.name, v_station[(s,l)])
        itter += 1
        if itter > 15:
            break
```

In [75]:
```
#Prints out the policy for each state
def print_pi(pi,end):
    itter = 0
    for s,l in pi:
        s_,l_ = pi[(s,l)]
        if s.name == end:

            print("Start Station: ", s.name,
                  " END")
        else:
            print("Start Station: ", s.name,
                  " Start Line: ", l.name,
                  " Next Station: ", s_.name,
                  " Next Line: ", l_.name)
        itter += 1
        if itter > 15:
            break
```

In [76]:

```
#Function used to print out best path from start to end stations
def print_path(v_station,pi,start,end):
    max_v = -9999
    start_s = None
    start_l = None
    last_l = None
    last_s = None

    itter = 0
    visit_stations = set()
    for s,l in v_station.keys():
        if s.name == start:
            if v_station[(s,l)] >= max_v:
                start_s = s
                start_l = l
    print("Start Station and Line: ", start_s.name, " ", start_l.name)
    run = True
    visit_stations.add(start_s)

    while run:
        next_s,next_l = pi[(start_s,start_l)]
        visit_stations.add(start_s)
        if next_s in visit_stations:
            max_temp_v = -99999
            for (s,r,l) in start_s.adj_stations:
                stat = graph.ret_station(s)
                if s != next_s.name:

                    if v_station[(stat,l)] > max_temp_v:
                        max_temp_v = v_station[(stat,l)]
                        next_s = stat
                        next_l = l




        print("Next Station and Line", next_s.name, " ", next_l.name)
        start_s = next_s
        start_l = next_l

        itter += 1
        if itter >= 10:
            print("Loop Detected: Terminating Search")
            run = False

        if start_s.name == end:
            print("Destination")
            run = False
```

Main Function

In [93]:

```
def policy_iter(graph,gamma, theta,end, start):

    # 1. Initialize

    graph.add_delay()
    line_post = {'Red':0,'Green':1,'Yellow':2,'Orange':3,'Blue':4,'Silver':5}

    states,v_station = set_states()

    #Creating initial policy, first station in adj_station list
    pi = dict()
    for (s,l) in states:
        s_,r,a_ = s.adj_stations[0]
        pi[(s,l)] = (graph.ret_station(s_),a_)

    set_end(v_station,end,graph)

    run = True
    while run :

        # 2. Policy Evaluation
        print("========================Eval Policy========================")
        d_check = True
        while d_check:
            delta = 0
            #going through each state
            for (s,l) in states:

                #getting best action
                (s_,l_) = pi[(s,l)]
                v = 0
                if s.name != end:
                    #iterating through each connected station
                    for s_a,r,l_a in s.adj_stations:

                        stat = graph.ret_station(s_a)
                        v_next = v_station[(stat,l_a)]
                        pr = s.prob[l]

                        #adding -5 if the action changes lines
                        if l.name != l_a:
                            r += -5

                        #prob that the best action is taken is .7 while the re
                        if stat == s_ and l_ == l_a:
                            v+= .7*pr[line_post[l_a.name]]*((r)+gamma*v_next)
                        else:
                            v+= .1*pr[line_post[l_a.name]]*((r)+gamma*v_next)
                    #calc delta
                    delta = max(delta, abs(v - v_station[(s,l)]))
                    #setting new V
                    v_station[(s,l)] = v
                else:
                    #keeping V of the destination the same
                    v_station[(s,l)] = 10
            print("delta", delta, "theta", theta, " ", delta < theta)
            #comparing delta and set threshold
            if delta < theta: d_check = False
```

```
        #going through each state
        for (s,l) in states:
            old_action = pi[(s,l)]
            max_act = old_action
            max_v = v_station[old_action]
            v_temp = 0

            #iterating through all connected states
            for s_a,r,l_a in s.adj_stations:
                stat = graph.ret_station(s_a)
                v_next = v_station[(stat,l_a)]
                pr = s.prob[l]

                v_temp = get_v(v_station,graph,stat,l_a,gamma)

                #storing the max V as new best action
                if v_temp > max_v:
                    max_v = v_temp
                    max_act = (graph.ret_station(s_a),l_a)

            pi[(s,l)] = max_act

            #comparing new and old action
            #if different keep iterating
            if old_action != max_act:
                policy_stable = False



        if policy_stable:
            run = False

    return v_station, pi
```

Example: College Park to Metro Center

Building Policy and V values

In [94]:

```
end = "Metro Center"
graph = start_alg(end)
threshold = .0000001
gamma = .9
st = "College Park"
v,pi = policy_iter(graph,gamma,threshold,end,st)
```

```
=======================Eval Policy=======================
delta 5.757123767962174 theta 1e-07   False
delta 2.8070678368900763 theta 1e-07   False
delta 0.885276718342954 theta 1e-07   False
```

```
delta 0.2988705729806931 theta 1e-07   False
delta 0.1107065692567577 theta 1e-07   False
delta 0.04069642786799932 theta 1e-07   False
delta 0.014841741879520143 theta 1e-07   False
delta 0.005384295708240394 theta 1e-07   False
delta 0.0019320249719916305 theta 1e-07   False
delta 0.0006606286116666027 theta 1e-07   False
delta 0.0002300684231419936 theta 1e-07   False
delta 7.867434274544394e-05 theta 1e-07   False
delta 2.0576822196360922e-05 theta 1e-07   False
delta 4.684307233659979e-06 theta 1e-07   False
delta 9.223418722115184e-07 theta 1e-07   False
delta 1.612972706510618e-07 theta 1e-07   False
delta 2.584147207329579e-08 theta 1e-07   True
========================Improve Policy=======================
========================Eval Policy=======================
delta 7.928216693289629 theta 1e-07   False
delta 0.24685320712131364 theta 1e-07   False
delta 0.0573241969725764 theta 1e-07   False
delta 0.006732726939786993 theta 1e-07   False
delta 0.0009286526236538251 theta 1e-07   False
delta 0.0003343152543546779 theta 1e-07   False
delta 0.00012035366592577645 theta 1e-07   False
delta 4.33273893385433e-05 theta 1e-07   False
delta 1.4623020162751743e-05 theta 1e-07   False
delta 4.979141696281886e-06 theta 1e-07   False
delta 1.5791883765103876e-06 theta 1e-07   False
delta 3.2757451773335333e-07 theta 1e-07   False
delta 5.6478699050899195e-08 theta 1e-07   True
========================Improve Policy=======================
========================Eval Policy=======================
delta 0.5764129563423195 theta 1e-07   False
delta 0.025938583035404417 theta 1e-07   False
delta 0.0011672362365935385 theta 1e-07   False
delta 3.5454800686451904e-05 theta 1e-07   False
delta 1.076939570410218e-06 theta 1e-07   False
delta 3.318697050502806e-08 theta 1e-07   True
========================Improve Policy=======================
```

Sample V Values

In [95]:

```
print_v(v)
```

```
Station:  Glenmont  Line:  Red -4.306718530752043
Station:  Wheaton  Line:  Red -4.7832334309588695
Station:  Forest Glen  Line:  Red -5.035935417155012
Station:  Silver Spring  Line:  Red -5.145047342101152
Station:  Takoma  Line:  Red -5.455334121679926
Station:  Fort Totten  Line:  Yellow -5.0122219409108055
Station:  Fort Totten  Line:  Green -5.0122219409108055
Station:  Fort Totten  Line:  Red -6.424784099318643
Station:  Brookland-CUA  Line:  Red -4.865411784741332
Station:  Rhode Island Ave  Line:  Red -4.369195238958703
Station:  Noma-Gallaudet U  Line:  Red -4.127783202048703
Station:  Union Station  Line:  Red -3.591014019985907
Station:  Judiciary Sq  Line:  Red -1.9214723044244923
Station:  Gallery Place  Line:  Yellow 1.0519433428476082
Station:  Gallery Place  Line:  Green 1.0519433428476082
```

```
Station:  Gallery Place  Line:  Red 3.301978814205947
```

Sample Polcies

In [96]:
```
print_pi(pi,end)
```

```
Start Station:  Glenmont  Start Line:  Red  Next Station:  Wheaton  Next Line:
Red
Start Station:  Wheaton  Start Line:  Red  Next Station:  Glenmont  Next Line:
Red
Start Station:  Forest Glen  Start Line:  Red  Next Station:  Silver Spring  N
ext Line:  Red
Start Station:  Silver Spring  Start Line:  Red  Next Station:  Takoma  Next L
ine:  Red
Start Station:  Takoma  Start Line:  Red  Next Station:  Fort Totten  Next Lin
e:  Red
Start Station:  Fort Totten  Start Line:  Yellow  Next Station:  Brookland-CUA
Next Line:  Red
Start Station:  Fort Totten  Start Line:  Green  Next Station:  Brookland-CUA
Next Line:  Red
Start Station:  Fort Totten  Start Line:  Red  Next Station:  Brookland-CUA  N
ext Line:  Red
Start Station:  Brookland-CUA  Start Line:  Red  Next Station:  Rhode Island A
ve  Next Line:  Red
Start Station:  Rhode Island Ave  Start Line:  Red  Next Station:  Noma-Gallau
det U  Next Line:  Red
Start Station:  Noma-Gallaudet U  Start Line:  Red  Next Station:  Union Stati
on  Next Line:  Red
Start Station:  Union Station  Start Line:  Red  Next Station:  Judiciary Sq
Next Line:  Red
Start Station:  Judiciary Sq  Start Line:  Red  Next Station:  Gallery Place
Next Line:  Red
Start Station:  Gallery Place  Start Line:  Yellow  Next Station:  Metro Cente
r  Next Line:  Red
Start Station:  Gallery Place  Start Line:  Green  Next Station:  Metro Center
Next Line:  Red
Start Station:  Gallery Place  Start Line:  Red  Next Station:  Metro Center
Next Line:  Red
```

Best Path from College Park to Metro Center

In [97]:
```
print_path(v,pi,st,end)
```

```
Start Station and Line:  College Park   Green
Next Station and Line Prince George's Plaza   Green
Next Station and Line West Hyattsville   Green
Next Station and Line Fort Totten   Green
Next Station and Line Brookland-CUA   Red
Next Station and Line Rhode Island Ave   Red
Next Station and Line Noma-Gallaudet U   Red
Next Station and Line Union Station   Red
Next Station and Line Judiciary Sq   Red
Next Station and Line Gallery Place   Red
Next Station and Line Metro Center   Red
Loop Detected: Terminating Search
Destination
```

Example: Code Getting Stuck

In [98]:
```
end = "Wiehle-Reston East"
graph = start_alg(end)
threshold = .0000001
gamma = .9
st = "Largo Town Center"
v,pi = policy_iter(graph,gamma,threshold,end,st)
```

```
========================Eval Policy========================
delta 5.757123767962174 theta 1e-07   False
delta 3.166012993499327 theta 1e-07   False
delta 1.001675285604231 theta 1e-07   False
delta 0.3165414438954448 theta 1e-07   False
delta 0.11070736549727034 theta 1e-07   False
delta 0.04069848581750346 theta 1e-07   False
delta 0.014845947542095672 theta 1e-07   False
delta 0.005398088943987744 theta 1e-07   False
delta 0.0019371550238229673 theta 1e-07   False
delta 0.0006803892952325086 theta 1e-07   False
delta 0.0002309858714051316 theta 1e-07   False
delta 8.10056104185719e-05 theta 1e-07   False
delta 1.949194741701632e-05 theta 1e-07   False
delta 3.2235356233201173e-06 theta 1e-07   False
delta 1.0630141575873608e-06 theta 1e-07   False
delta 3.689308885057585e-07 theta 1e-07   False
delta 1.3028847778429054e-07 theta 1e-07   False
delta 4.3375909086762476e-08 theta 1e-07   True
========================Improve Policy========================
========================Eval Policy========================
delta 0.182245658603021 theta 1e-07   False
delta 0.05740738245995214 theta 1e-07   False
delta 0.005812497295907271 theta 1e-07   False
delta 0.0005901631314690903 theta 1e-07   False
delta 5.996755651604957e-05 theta 1e-07   False
delta 6.09503408099954e-06 theta 1e-07   False
delta 6.195589188351391e-07 theta 1e-07   False
delta 6.298097687817972e-08 theta 1e-07   True
========================Improve Policy========================
```

V Values

In [99]:
```
print_v(v)
```

```
Station:  Glenmont  Line:  Red -4.306724701021682
Station:  Wheaton  Line:  Red -4.783252998791469
Station:  Forest Glen  Line:  Red -5.0363270648953264
Station:  Silver Spring  Line:  Red -5.146287872798413
Station:  Takoma  Line:  Red -5.459216364690182
Station:  Fort Totten  Line:  Red -6.43693146163284
Station:  Fort Totten  Line:  Green -5.02110066325757
Station:  Fort Totten  Line:  Yellow -5.02110066325757
Station:  Brookland-CUA  Line:  Red -4.897109499601604
Station:  Rhode Island Ave  Line:  Red -4.468087567708409
Station:  Noma-Gallaudet U  Line:  Red -4.4371988578614125
```

```
Station:  Union Station  Line:  Red -4.559158467665196
Station:  Judiciary Sq  Line:  Red -4.9507445049569325
Station:  Gallery Place  Line:  Red -6.176451980673013
Station:  Gallery Place  Line:  Green -4.732761758450576
```

Best Policies

In [100…

```
print_pi(pi,end)
```

```
Start Station:  Glenmont  Start Line:  Red  Next Station:  Wheaton  Next Line:
Red
Start Station:  Wheaton  Start Line:  Red  Next Station:  Glenmont  Next Line:
Red
Start Station:  Forest Glen  Start Line:  Red  Next Station:  Silver Spring  N
ext Line:  Red
Start Station:  Silver Spring  Start Line:  Red  Next Station:  Takoma  Next L
ine:  Red
Start Station:  Takoma  Start Line:  Red  Next Station:  Fort Totten  Next Lin
e:  Red
Start Station:  Fort Totten  Start Line:  Red  Next Station:  Brookland-CUA  N
ext Line:  Red
Start Station:  Fort Totten  Start Line:  Green  Next Station:  Brookland-CUA
Next Line:  Red
Start Station:  Fort Totten  Start Line:  Yellow  Next Station:  Brookland-CUA
Next Line:  Red
Start Station:  Brookland-CUA  Start Line:  Red  Next Station:  Rhode Island A
ve  Next Line:  Red
Start Station:  Rhode Island Ave  Start Line:  Red  Next Station:  Noma-Gallau
det U  Next Line:  Red
Start Station:  Noma-Gallaudet U  Start Line:  Red  Next Station:  Union Stati
on  Next Line:  Red
Start Station:  Union Station  Start Line:  Red  Next Station:  Judiciary Sq
Next Line:  Red
Start Station:  Judiciary Sq  Start Line:  Red  Next Station:  Gallery Place
Next Line:  Red
Start Station:  Gallery Place  Start Line:  Red  Next Station:  Metro Center
Next Line:  Red
Start Station:  Gallery Place  Start Line:  Green  Next Station:  Metro Center
Next Line:  Red
Start Station:  Gallery Place  Start Line:  Yellow  Next Station:  Metro Cente
r  Next Line:  Red
```

Code having to be stopped since infinite loop

In [101…

```
print_path(v,pi,st,end)
```

```
Start Station and Line:  Largo Town Center   Blue
Next Station and Line Morgan Blvd   Silver
Next Station and Line Addision Rd   Silver
Next Station and Line Morgan Blvd   Silver
Next Station and Line Largo Town Center   Silver
Next Station and Line Morgan Blvd   Silver
Next Station and Line Largo Town Center   Silver
Next Station and Line Morgan Blvd   Silver
Next Station and Line Largo Town Center   Silver
Next Station and Line Morgan Blvd   Silver
```

```
Next Station and Line Largo Town Center   Silver
```