EC504 Project Report

Project 7 - A Route Recommendation Application

Section 1: Team Information

- Ken Krebs
  - BU-ID: U87040597
  - SCC User Name: kenkrebs
- Santiago Gomez
  - BU-ID: U37299983
  - SCC User Name: santi09
- Apollo Lo
  - BU-ID: U61388879
  - SCC User Name: apollol
- John (Wiley) Hunt
  - BU-ID: U32633283
  - SCC User Name: whunt

Section 2: Abstract

In our project we developed a Python-based route recommendation application to generate a shortest path for a vehicle between a user-provided starting and ending location within a user-selected city. This application provides a simple (command-line) interface for a user to select a city, input the latitudes and longitudes of the start location and destination, and then select the algorithm to use (Djikstra's, Bellman-Ford, or A*). Our program then computes the shortest path between these locations, and displays the following to the user on an HTML page:

- Total distance between the points (along the route)
- Algorithm used (and time required for calculation)
- A visual representation of the map between the two points with the route highlighted.
- Step by step directions (with street names, travel lengths, bearings, and turns)

Notably, this application connects with the Open Street Maps (OSM) API using the *osmnx* and *networkx* libraries to gather the information on street networks within a city (including street names, lengths, bearings, etc.), produce a graph structure, and subsequently provide a visual representation to the user.

As discussed in greater detail in *Section 5: Testing Analysis and Sample Results of Application*, we performed an in-depth stress-testing on the various algorithms (as well variants of some of these algorithms) used by our program. We tested the performance of each algorithm on multiple routes within small (<1000 NODES/ <2500 EDGES), medium (~10,000-25,000 NODES/~25,0000-50,000 EDGES), and large(> 50,000 NODES/ > 130,000 EDGES) graphs. Ultimately, we found that across small, medium, and large graphs, our A* Search algorithm and

our version of Djikstra's (using a custom priority Queue) provided the best overall performance (with no real separation between the two algorithms).

Section 3: Instructions for Running the Code

Running Locally

- Clone our Repository at https://github.com/whunt1965/EC504MapApp
- Download and install miniconda if it is not already installed.
- Setup Miniconda Environment using the provided shell script
    - Run *./buildEnv.sh* in your terminal in the cloned directory
- Run the Program (in the same cloned directory)
    - Run *conda activate ox* in your terminal in the cloned directory
    - Run *./runAppLocal* in your terminal in the cloned directory (note: the commands in this script will work for Mac/Linux users, but may need to be updated for Windows users)
    - Enter a location, source latitude, source longitude, destination latitude, destination longitude, and choose an algorithm
    - Results will be displayed in your browser as an HTML page

Running on the SCC

- To run our application on the SCC, an X-Forwarding Application is necessary to view the GUI output provided by the Application. Please consult the appropriate section for your operating system:
    - Windows
        - An X-Server can be installed and configured on a Microsoft Windows system using MobaXterm. MobaXterm has an X-server built-in, so after you login to the SCC you should be able to launch a remote graphical application without any extra work.
    - Apple OS X
        - First install the XQuartz/X11 application (if not already installed), which is available at http://xquartz.macosforge.org/landing.
        - Once XQuartz is installed, connect to the SCC using: ssh –Y yourBULoginName@scc1.bu.edu
    - Linux
        - X Forwarding should be built-in on your Linux machine but you will need to include the -X flag with your ssh command.ssh –X yourBULoginName@scc1.bu.edu
- Log in to SCC1 and navigate to /projectnb/ec504/whunt/Project/EC504MapApp
- Setup a Miniconda Environment using the provided shell script
    - Run *./buildEnv.sh* in your terminal (Note: if the first installation fails, run the script again and choose to uninstall the previously installed environment)

- Run the Program
    - Run *./runAppSCC.sh* in your terminal to run the program
    - Enter a city (eg, Boston, MA, USA) , source latitude, source longitude, destination latitude, destination longitude, and choose an algorithm
    - Results will be displayed in your browser as an HTML page

Testing with sample Inputs

- We have provided several sample input files which may be used to test the program in the *input* folder.
- To test these files, use the appropriate runScript (either for the SCC or local machine, depending on which system you are using to test) and run the script as follows:
    - SCC
        - ./RunAppSCC < (Full path to input file)
    - Local
        - ./RunAppLocal < (Full path to input file)
- These tests will generate the results in an HTML page (output.html) which can be compared to the respective city's output.html file in the *Project/output/<city name>* folder for validation. These sample output files can be viewed by simply right-clicking the desired output.html file and loading in the browser (or, from the command line in the SCC, navigating to the directory containing the output folder and using the following command: *firefox output.html)*
- Note: The time displayed on the html page represents the time required to run the algorithm itself. For an initial trial on a given city, additional time is needed to download the graph from OSM and for each trial, and some time is needed to convert the .graphml file (the format provided by OSM) into the object structure required by the algorithm for processing.

Section 4: Overview of Algorithms and other Application Components

Discussion of Algorithms Implemented

To calculate the shortest path between the source and destination locations, we implemented and tested several different shortest path algorithms (and in some cases, variations of these algorithms). These algorithms include:

- ***Bellman Ford***
    - We implemented the Bellman Ford shortest path algorithm using the same implementation from Homework 4 (simply translated into Python with some modifications to compensate for our lack of pointers in the Python language).
    - For this implementation of Bellman Ford, we used a built-in Python Queue as the "working Queue" used to hold/extract nodes for processing.
- ***A* Search***
    - We implemented the A* Search algorithm (based on the pseudocode provided in class) which utilizes a specialized HeapQ data structure to hold/extract nodes for processing.
    - This HeapQ data structure (which is based on the HeapQ implemented in Homework 2, though translated to Python and modified to get around our lack of pointers) used priorities based on distance from the source and a heuristic estimated distance to the destination.
    - To calculate the heuristic (ie, the estimated distance to the target node), we calculated the Euclidean distance between each node in the graph and the destination node (based on the latitude and longitude of the two nodes) as part of our pre-processing stage (essentially, when translating the data extracted from OSM into usable Node/Edge objects).
    - This algorithm attempts to search in a more "targeted" manner than Djikstra's by using this heuristic as a guide to explore edges which get us closer to the destination node (and of course, end the search when the destination node is found)
- ***Djikstra*'s Shortest Path Algorithm**
    - We implemented several different versions of the Djikstra algorithm in order to explore relative performance based on the different implementations. These implementations include:
        - *Optimized Djikstra with a HeapQ*
            - Based on our implementation in HW4, we implemented a version of Djikstra using our specialized HeapQ data structure(with priorities based on distance from the source) to hold/extract/update priorities of nodes during the search.
            - In addition, we implemented an additional optimization to terminate the "search" when we discover the destination node (rather than traversing the entire graph).

        - Ultimately this proved to be our fastest Djikstra algorithm (as discussed in our testing results below) and is the version implemented in our main application.
    - *Optimized Djikstra with an Array*
        - We also implemented a version of Djikstra using an array (rather than a Queue) as was implemented as an example in Homework 4.
        - In addition (as in the case of the Djikstra version with a HeapQ described above), we implemented an additional optimization to terminate the "search" when we discover the destination node (rather than traversing the entire graph).
    - *No-break Djikstra with a HeapQ*
        - To compare performance with our "optimized" Djikstra with a HeapQ implementation, we created another version which does not terminate the algorithm when we discover the destination node and instead traverse the entire graph.
    - *No-break Djikstra with an Array*
        - To compare performance with our "optimized" Djikstra with an Array implementation, we created another version which does not terminate the algorithm when we discover the destination node and instead traverse the entire graph.

A discussion of the performance of these algorithms is included in "Testing Results Section."

Other Components Implemented

In order to translate these algorithms into a full user application, we implemented additional components to handle the user interface as well as provide utilities for parsing data received from OSM and extract useful information (street names, bearings, etc.) from our shortest paths to relay to the user. Broadly, these can be categorized into ***Utilities*** and ***User Interface*** Components.

- **Utilities** - In order to make our application useful for a user (and easier to code from an algorithmic perspective), we implemented several utility functions, including:
    - Graph Parsing: As the OSMNX library provides complex data structures to represent nodes and edges (which are very difficult to process from an algorithmic perspective), we implemented utility functions and classes to traverse the raw OSMNX data, and break the graph into custom node and edge objects (which store important information such as ID, street names, weights (distances), and bearings).
    - Step by Step Directions: As we wanted our application to be able to provide a user with step-by-step directions between a start and end location, we implemented functions to traverse through the nodes along our shortest path and extract key information such as street names, distance travelled along the

selected edge, and bearing (used to indicate direction of travel and identify "turns" as they occur in a path).
    - Output Helpers: Finally, we implemented helper functions to take the data collected by the algorithms and other utility functions and build an easily understandable report in HTML format.

- **User Interface -** Within main.py, we provide a command line interface for a user to enter a city name, start/end latitude and longitude, as well as select an algorithm to use for computing a route. Once the route is calculated, we use the Output helper utilities to produce an output page (as an HTML page which can be loaded in the browser), with the following information:
    - Start and End locations
    - Total distance between the points (along the route)
    - Algorithm used (and time required for calculation)
    - Step by step directions (with street names, travel lengths, bearings, and turns)
    - A visual representation of the map between the two points with the route highlighted.

Section 5: Testing Analysis and Sample Results of Application

Testing Strategy and Performance Analysis

Testing Strategy

To better understand the performance of the algorithms (and variations thereof) that we implemented, we implemented a rigorous testing strategy. As part of this strategy, we sought to understand how these algorithms performed on 3 different categories of graphs: small (<1000 NODES/ <2500 EDGES), medium (~10,000-25,000 NODES/~25,0000-50,000 EDGES), and large(> 50,000 NODES/ > 130,000 EDGES).

For each of these categories of graphs, we selected sample cities with the requisite structure and then used Google Maps to identify 3-4 routes within each city which would require traversing the longest path within the city (i.e., we selected starting and ending points at opposite ends of the city).

Subsequently, we ran each of our algorithms on each of these routes (these were run locally on a Windows machine) in order to identify the total time required to calculate the route. To identify total runtime, we simply captured the system time before calling the algorithm and the system time after the algorithm returned (and measured the delta). A table showing the results from this testing is included on the following pages.

In addition, to verify correctness of each route produced by our algorithms, we used the built-in shortest path library function (within the *networkx* library, which is utilized by the osmnx library to create graph structures from street networks) to extract a shortest path and compare with results. As part of this comparison, we compared both the distance produced by the library function to our calculated distance as well as the actual path itself (ie, the nodes traversed). Overall, we had a 95% success rate in our results matching the built-in computations. However, in two (out of 37) of the test cases, all of our algorithms produced a path which did not match the built-in path (this occurred for 1 path in Boston and 1 path in Key West). While all of our algorithms produced the same path, the built-in function found a slightly shorter path which was in ~200-300 meters in length of our computed path). This may be due to the fact that the built-ins are able to operate on the raw GraphML structure (produced by the OSM API). Meanwhile, we transform this structure (by extracting a subset of useful features) into a set of custom node and edge objects (to make it easier for our algorithms). There therefore may be some cases where our “transformation” excludes certain features present in the GraphML which could lead to a slightly shorter path.

Figure 1: Table of Test Cases and Timing Results

	Graph Data				Algorithmic Performance (seconds required to compute shortest path) (Fastest and Slowest algorithms are highlighted for each test case)					
Graph Type	City	# Nodes	# Edges	Density (Edges/Node)	BF	A*	Dij Simple	Dij Heap	NB Dij Simple	NB Dij Heap
Small	Key West, FL, USA	786	2258	2.872773537	0.017156363	0.009058237	0.082759857	0.007978201	0.084004641	0.00803137
Small	Key West, FL, USA	786	2258	2.872773537	0.011967421	0.00802803	0.06990242	0.007930994	0.073149681	0.00798321
Small	Key West, FL, USA	786	2258	2.872773537	0.015103102	0.007991791	0.069441319	0.008975983	0.080919027	0.00797915
Small	Key West, FL, USA	786	2258	2.872773537	0.013958931	0.006830692	0.052062511	0.005611658	0.072337627	0.00790262
Small	Bath, ME, USA	529	1387	2.621928166	0.008029699	0.004985571	0.030098438	0.005037069	0.032993555	0.00498605
Small	Bath, ME, USA	529	1387	2.621928166	0.008980036	0.005984306	0.032938004	0.005017042	0.032008171	0.00503111
Small	Bath, ME, USA	529	1387	2.621928166	0.011024714	0.00498414	0.027939796	0.005031109	0.032185316	0.00502443
Small	Bath, ME, USA	529	1387	2.621928166	0.008980989	0.004986286	0.033919573	0.005494356	0.030971527	0.00602841
Small	Stowe, VT, USA	512	1132	2.2109375	0.003988981	0.005984068	0.029920101	0.004986525	0.030650377	0.00502825
Small	Stowe, VT, USA	512	1132	2.2109375	0.007692099	0.005084991	0.030041695	0.004975319	0.031060696	0.00504827
Small	Stowe, VT, USA	512	1132	2.2109375	0.00498724	0.002990961	0.016007185	0.002994061	0.030909061	0.00494623
Small	Stowe, VT, USA	512	1132	2.2109375	0.007603645	0.003988743	0.027080774	0.003980398	0.030069113	0.00404716
Medium	Boston, MA, USA	10987	25322	2.304723764	1.048258305	0.176236391	19.0099206	0.218380451	25.2355361	0.18712425
Medium	Boston, MA, USA	10987	25322	2.304723764	0.874254942	0.21523881	20.67712021	0.181763887	23.45306134	0.26574564
Medium	Boston, MA, USA	10987	25322	2.304723764	1.203927517	0.200387001	23.02081227	0.186010361	21.11501503	0.23269725
Medium	Boston, MA, USA	10987	25322	2.304723764	1.01192975	0.165515423	18.31893969	0.168062449	20.47313452	0.17855978
Medium	Seattle, WA, USA	15638	45690	2.921729121	3.280170679	0.317967653	54.46555066	0.322723866	56.75458407	0.28526497
Medium	Seattle, WA, USA	15638	45690	2.921729121	1.078494072	0.056551218	3.45673275	0.050911903	51.97557378	0.27215385

	Graph Data				Algorithmic Performance (seconds required to compute shortest path) (Fastest and Slowest algorithms are highlighted for each test case)					
Graph Type	**City**	**# Nodes**	**# Edges**	**Density (Edges/Node)**	**BF**	**A***	**Dij Simple**	**Dij Heap**	**NB Dij Simple**	**NB Dij Heap**
Medium	Seattle, WA, USA	15638	45690	2.921729121	1.080896378	0.187848806	26.45530152	0.156202793	53.31482863	0.27835703
Medium	Seattle, WA, USA	15638	45690	2.921729121	1.898926258	0.167996645	24.81438136	0.155269146	52.12922382	0.30521894
Medium	Nashville, TN, USA	21850	54557	2.496887872	2.11976552	0.461115122	105.2917368	0.406281471	111.7562373	0.43609643
Medium	Nashville, TN, USA	21850	54557	2.496887872	8.250644445	0.366239071	86.94050598	0.413599253	106.6472681	0.4481523
Medium	Nashville, TN, USA	21850	54557	2.496887872	3.735167503	0.15784359	28.51339126	0.16299367	105.7657139	0.41314435
Medium	Nashville, TN, USA	21850	54557	2.496887872	2.189629316	0.257078409	57.44206071	0.258980513	104.0281298	0.43489099
Large	New York, New York, USA	55314	140557	2.541074592	48.78854871	1.166575909	702.2584939	1.131014347	773.6170604	1.29598808
Large	New York, New York, USA	55314	140557	2.541074592	26.75201058	1.360488176	808.7817562	1.280940533	817.6586323	1.26896763
Large	New York, New York, USA	55314	140557	2.541074592	24.22308421	1.50980401	666.2146616	1.084590197	809.9255695	1.23750401
Large	New York, New York, USA	55314	140557	2.541074592	19.77525091	1.240972519	698.652107	1.08201766	807.0998838	1.89881325
Large	Los Angeles, CA, USA	50141	137738	2.747013422	35.67507935	1.090646744	602.9589405	1.211746454	658.707732	1.04485726
Large	Los Angeles, CA, USA	50141	137738	2.747013422	13.23227048	1.186382055	644.4495893	2.483489752	722.8119066	1.40804386
Large	Los Angeles, CA, USA	50141	137738	2.747013422	16.47552466	1.246623039	691.7709625	1.175914526	718.9448085	1.01651168
Large	Los Angeles, CA, USA	50141	137738	2.747013422	20.61626172	1.10900569	608.1965241	1.012811422	638.5591574	1.05451393
Large	Houston, TX, USA	59619	149258	2.503530754	32.73928094	1.41967988	1005.161289	1.78493166	1152.624957	2.11014009
Large	Houston, TX, USA	59619	149258	2.503530754	16.04284453	1.73444891	1015.145248	1.599303484	1120.215214	2.49342227
Large	Houston, TX, USA	59619	149258	2.503530754	33.01124907	3.551515818	929.6173952	1.619872808	1141.676867	1.69640851

Analysis of Results

Small Graphs

For small graphs, we found that in 6 of the 12 test cases, our “optimized” Djikstra with a HeapQ (which breaks when finding the destination node) provided the best performance. Meanwhile, A* Search provided the fastest result in 4 of 12 test cases, while “No-break” Djikstra and Bellman Ford provided the fastest result in 1 case each. As these are small graphs, the delta between the fastest and next fastest algorithms tended to be within a fraction of a millisecond, so the difference between them may not be significant enough to determine which one was truly “fastest.”

Meanwhile, as expected, the worst-performing algorithms were the No-break Djikstra with an array (worst performer in 10 of 12 test cases) and “optimized” Djikstra with an array (worst performer in 2 of 12 test cases).

Medium Graphs

For medium graphs, we found that in 6 of the 12 test cases, our “optimized” Djikstra with a HeapQ (which breaks when finding the destination node) provided the best performance. Meanwhile, A* Search provided the fastest result in 5 of 12 test cases, while “No-break” Djikstra provided the fastest result in 1 case. As in the case of the small graphs, the delta between the fastest and next fastest algorithms tended to be within a fraction of a second.

Meanwhile, as expected, the worst-performing algorithms were the No-break Djikstra with an array (worst performer in 11 of 12 test cases) and “optimized” Djikstra with an array (worst performer in 1 of 12 test cases).

Large Graphs

For large graphs, we found that in 6 of the 11 test cases, our “optimized” Djikstra with a HeapQ (which breaks when finding the destination node) provided the best performance. Meanwhile, A* Search provided the fastest result in 3 of 11 test cases, while “No-break” Djikstra provided the fastest result in 2 cases.

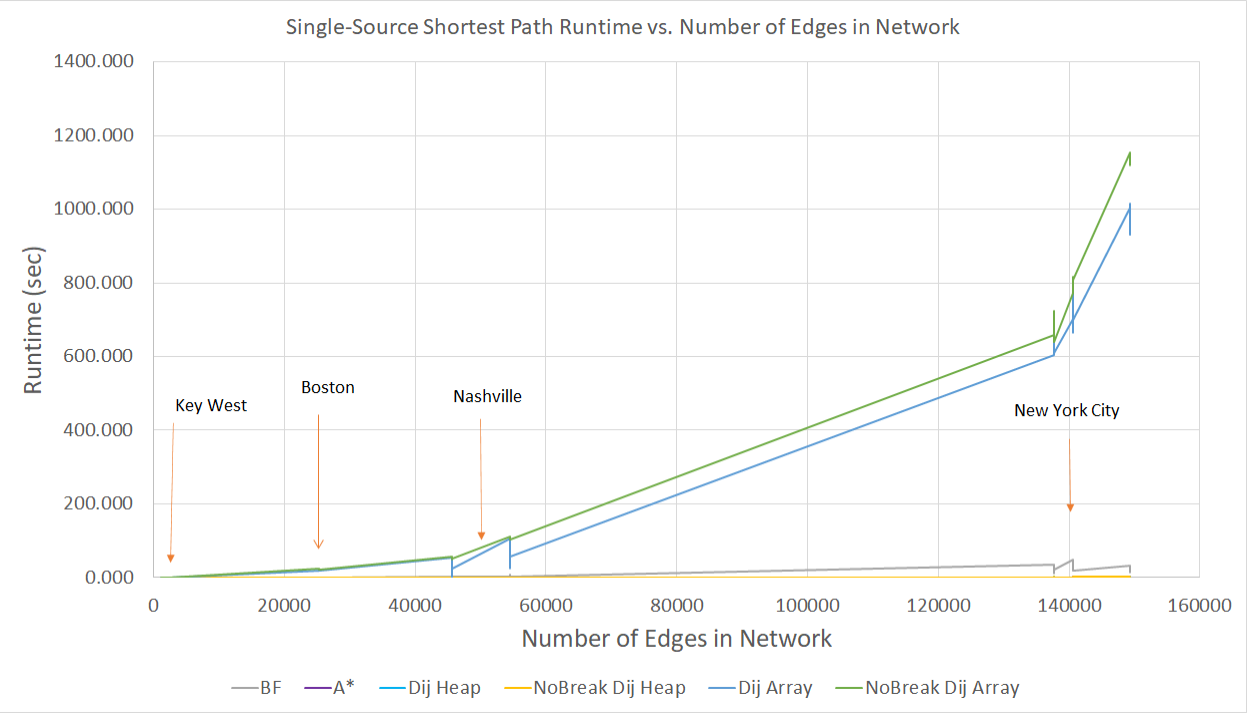
Interestingly, in the case of large graphs, we expected to see greater separation between A* search and Djikstra. However, perhaps because of our choice of start and end locations (at the edges of the city graph), there is very little separation between the two. If we had chosen a starting location in the middle of the graph (in which case, Djikstra might spend more time searching in the wrong direction), we might see better relative performance out of A* search. Moreover, we were surprised that in some cases, our no-break Djikstra outperformed the “optimized” Djikstra. This could be due to the fact that the additional check we implement at the top of the loop (which checks whether the node extracted from the heap is the destination) provides a performance hit that is significant enough to overshadow the performance speedup provided by breaking when we find the destination node.

Meanwhile, as expected, the worst-performing algorithms were the No-break Djikstra with an array, which was the worst performer in all 11 test cases.

Additional Charts and Discussion of Performance

Below, we provide a series of charts generated from this data which provide further discussion into the relative performance of the various algorithms.

Figure 2



In Figure 2, we can see that the worst performing algorithms are Dijkstra with Array and No-break Dijkstra with Array. While they could perform in under .08 seconds small networks, these two algorithms quickly pass the 1 minute threshold with medium networks. They take an eternity (over 10 minutes) with large networks.

Figure 3

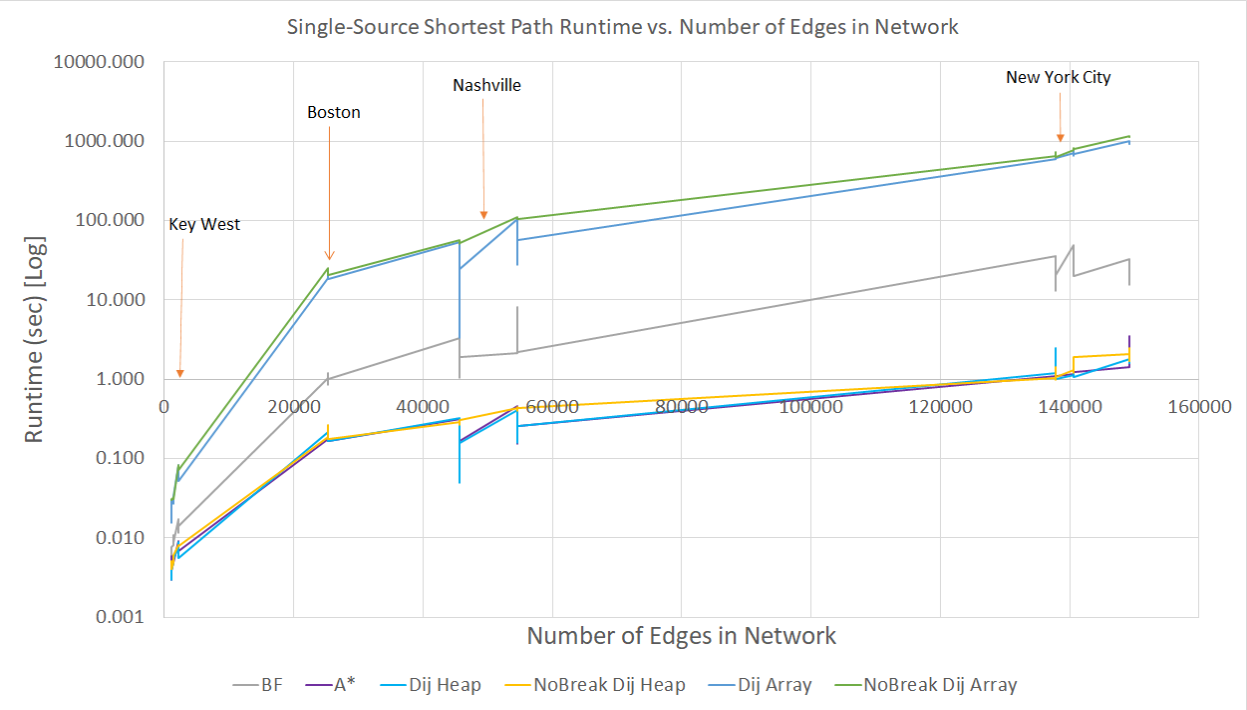
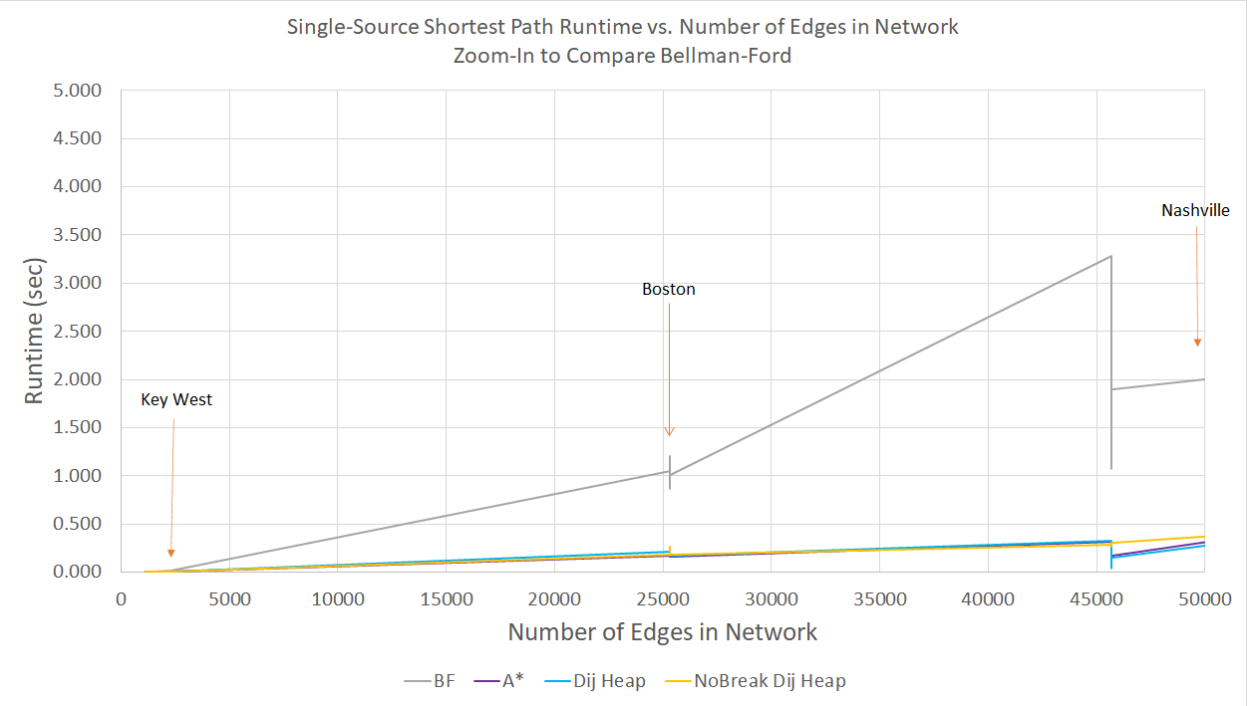


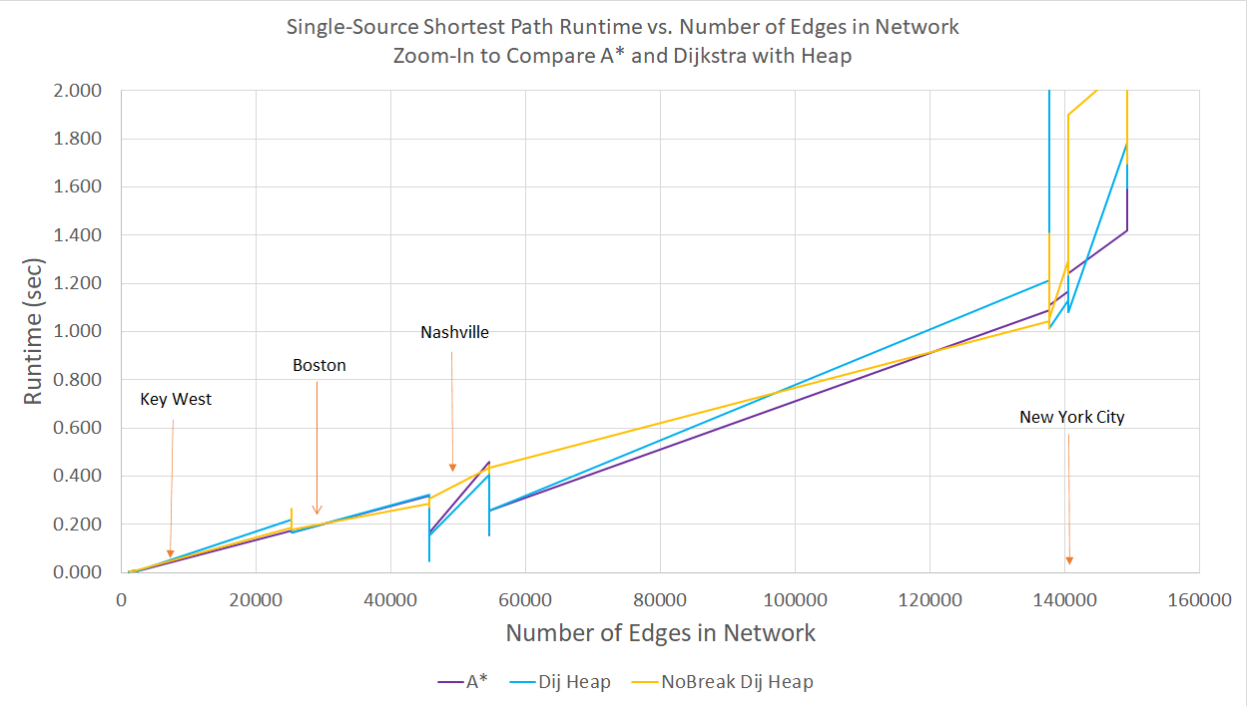
Figure 3 is the same as Figure 2 but the y-axis is in logarithmic form so that all the algorithms are visible on the graph.

Figure 4



In Figure 4, we zoom in to see the performance of Bellman-Ford in relation to the faster algorithms. Bellman-Ford can perform in under .01 seconds with small networks. In a medium network like Boston that is around 30,000 edges, Bellman-Ford can perform at about 1 second. Above 50,000 edges, Bellman-Ford's runtime balloons in comparison to A* Search and Dijkstra with a Heap Queue. In very large networks like New York City, Bellman-Ford ranged between 30-50 seconds. By contrast, A* Search and Dijkstra with Heap Queue found their routes within 2 seconds.

Figure 5

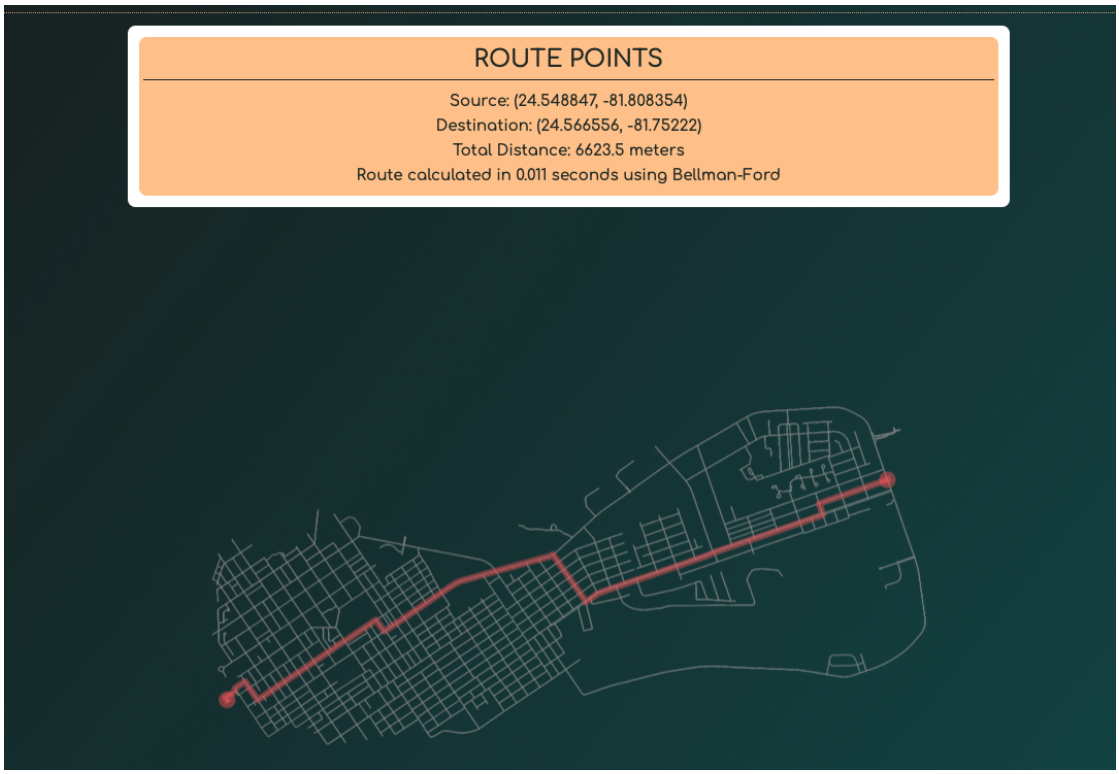


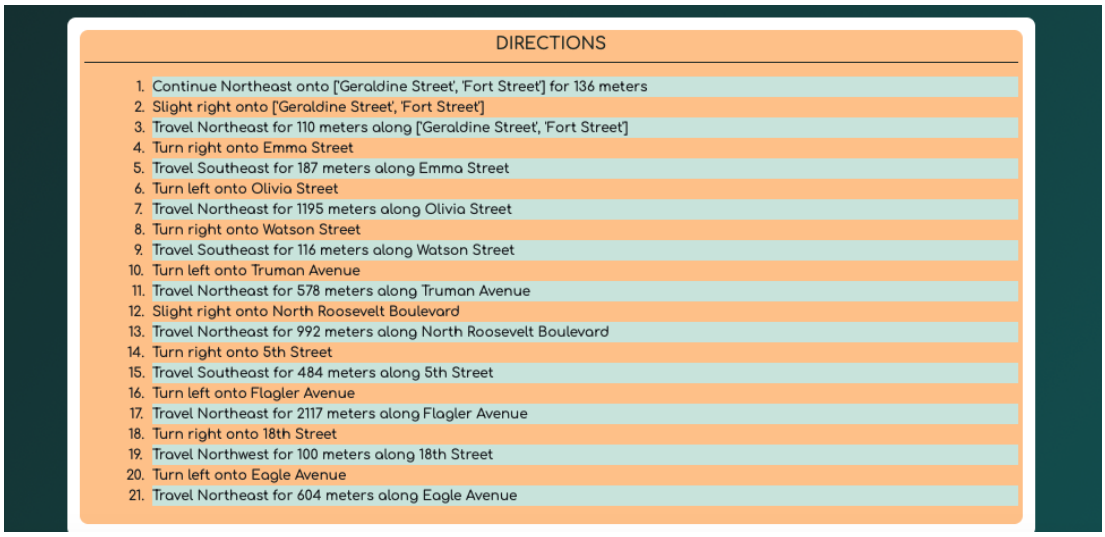
We zoom in even further to see A* Search, Dijkstra with Heap, and No-break Dijkstra with Heap. These algorithms outperformed their counterparts by orders of magnitude. In very large networks like New York City and Houston, Texas, these algorithms were running between 1 to 3 seconds. While we expected this with the Dijkstra with Heap Queue that terminates upon finding the destination, we did not expect the non-optimized Dijkstra that has to compute distances to all vertices to perform just as well. Ultimately, A* Star Search and Djikstra with a HeapQ consistently ran the fastest. For applications dealing with large graphs, these algorithms provide the best overall options.

Sample Results of Application

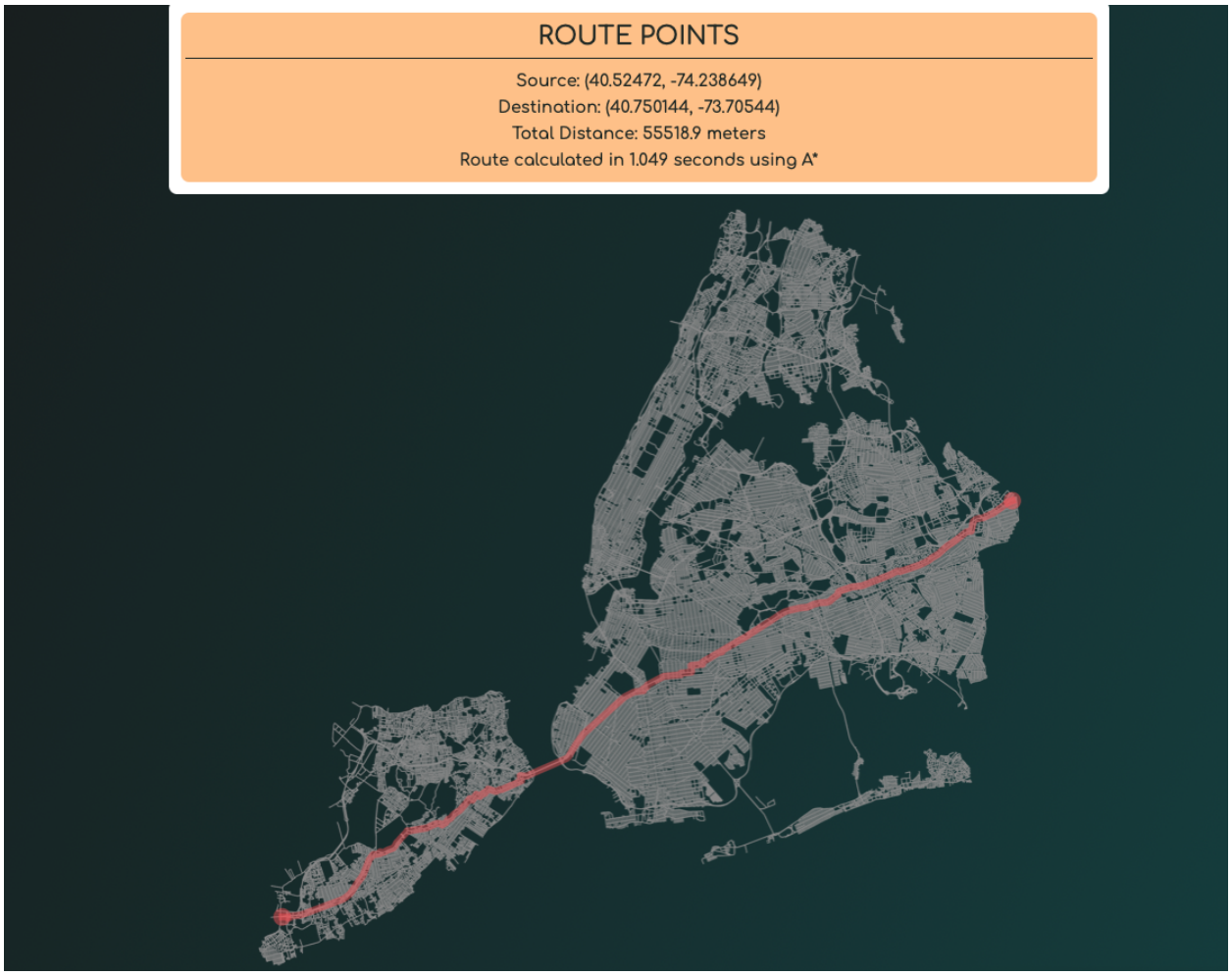
Below, please find two sample results produced by our system. Additional examples are available in the output folder (and can be viewed by loading the “output.html” for a given city in the browser)

Sample 1:
City: Key West, Florida, USA
Start Location: (24.548847, -81.808354)
End Location: (40.750144, -73.70544)
Algorithm Used: Bellman-Ford





Sample 2:
City: New York, New York, USA
Start Location: (40.52472, -74.238649)
End Location: (40.750144, -73.70544)
Algorithm Used: A*

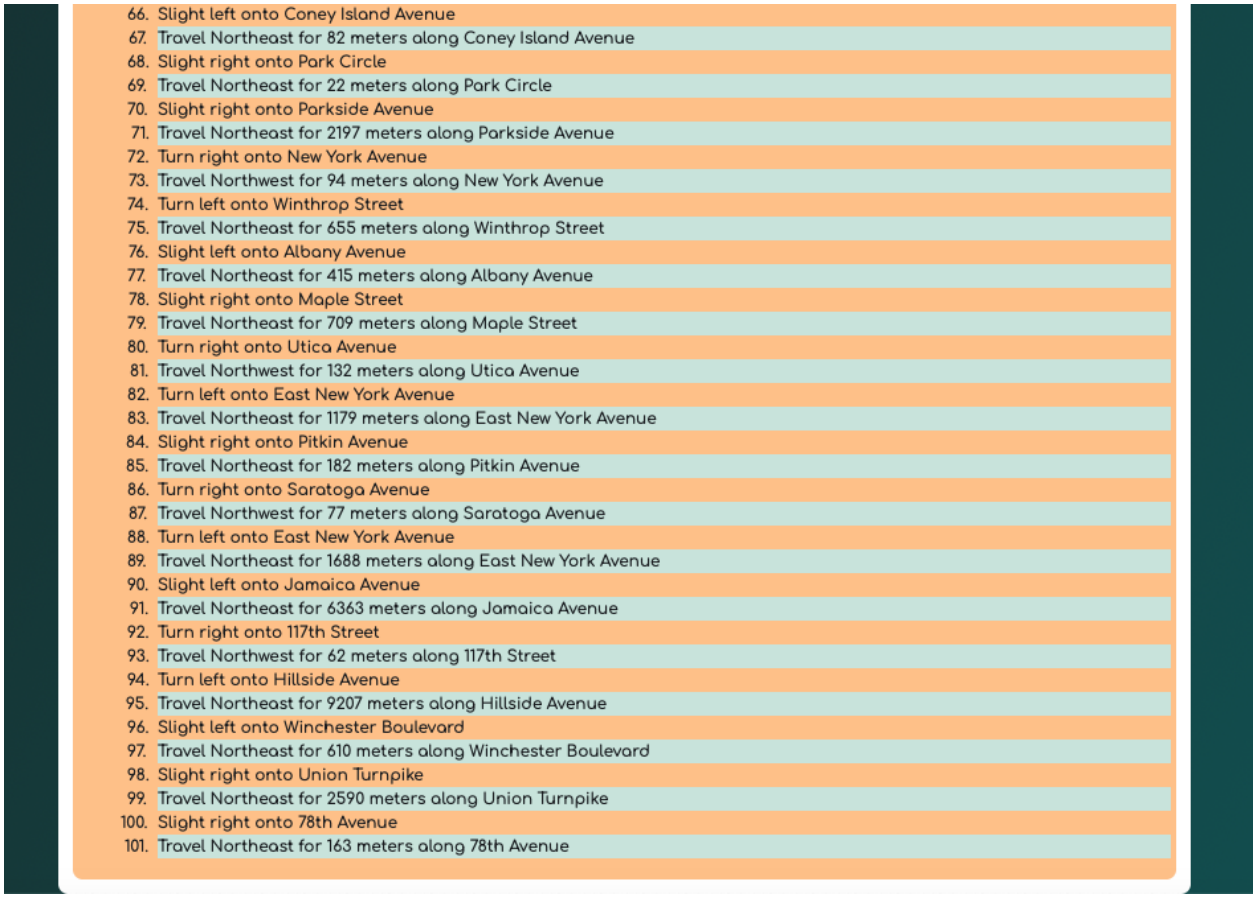


DIRECTIONS

1. Travel Southeast for 338 meters along South Bridge Street
2. Turn left onto Boscombe Avenue
3. Travel Northeast for 185 meters along Boscombe Avenue
4. Slight left onto Korean War Veterans Parkway
5. Continue Northeast onto Korean War Veterans Parkway for 549 meters
6. Slight right onto Korean War Veterans Parkway

6. Slight right onto Korean War Veterans Parkway
7. Travel Northeast for 4289 meters along Korean War Veterans Parkway
8. Slight left onto Drumgoole Road East
9. Continue Northeast onto Drumgoole Road East for 358 meters
10. Slight left onto Drumgoole Road East
11. Travel Northeast for 1627 meters along Drumgoole Road East
12. Turn right onto Wainwright Avenue
13. Continue Southeast onto Wainwright Avenue for 24 meters
14. Slight right onto Wainwright Avenue
15. Travel Southeast for 18 meters along Wainwright Avenue
16. Turn left onto Drumgoole Road East
17. Travel Northeast for 179 meters along Drumgoole Road East
18. Slight right onto Gurley Avenue
19. Travel Northeast for 239 meters along Gurley Avenue
20. Turn right onto Getz Avenue
21. Travel Northwest for 111 meters along Getz Avenue
22. Turn left onto Arthur Kill Road
23. Travel Southeast for 2254 meters along Arthur Kill Road
24. Turn left onto Richmond Road
25. Travel Northeast for 1686 meters along Richmond Road
26. Slight left onto Morley Avenue
27. Travel Northeast for 214 meters along Morley Avenue
28. Turn right onto Richmond Road
29. Travel Southeast for 2726 meters along Richmond Road
30. Slight right onto Buel Avenue
31. Travel Southeast for 189 meters along Buel Avenue
32. Turn left onto Jefferson Street
33. Travel Northeast for 448 meters along Jefferson Street
34. Turn right onto Alter Avenue
35. Travel Southeast for 254 meters along Alter Avenue
36. Turn left onto North Railroad Avenue
37. Travel Northeast for 311 meters along North Railroad Avenue
38. Turn right onto Burgher Avenue
39. Travel Southeast for 76 meters along Burgher Avenue
40. Turn left onto South Railroad Avenue
41. Travel Northeast for 360 meters along South Railroad Avenue
42. Turn right onto Reid Avenue
43. Travel Southeast for 897 meters along Reid Avenue
44. Turn left onto McClean Avenue
45. Travel Northeast for 1317 meters along McClean Avenue
46. Turn right onto Lily Pond Avenue
47. Travel Northwest for 168 meters along Lily Pond Avenue
48. Turn left onto Staten Island Expressway
49. Continue Northeast onto Staten Island Expressway for 350 meters
50. Turn right onto Staten Island Expressway

51. Travel Southeast for 62 meters along Staten Island Expressway
52. Turn left onto ['Verrazzano-Narrows Bridge', 'Staten Island Expressway']
53. Travel Northeast for 2720 meters along ['Verrazzano-Narrows Bridge', 'Staten Island Expressway']
54. Slight left onto Gowanus Expressway
55. Travel Northeast for 1321 meters along Gowanus Expressway
56. Slight left onto Fort Hamilton Parkway
57. Continue Northeast onto Fort Hamilton Parkway for 603 meters
58. Slight right onto Fort Hamilton Parkway
59. Travel Northeast for 4010 meters along Fort Hamilton Parkway
60. Slight right onto Caton Avenue
61. Travel Northeast for 631 meters along Caton Avenue
62. Turn right onto Ocean Parkway
63. Travel Northwest for 124 meters along Ocean Parkway
64. Turn left onto Caton Place
65. Travel Northeast for 283 meters along Caton Place



Section 6: References

To better understand the osmnx (and networkx) library functions and data structures, we used the following sources:

- For understanding networkx graphs: https://networkx.org/documentation/latest/tutorial.html
- For retrieving data from OSM: https://automating-gis-processes.github.io/CSC18/lessons/L3/retrieve-osm-data.html
- For understanding OSMNX data structures: https://automating-gis-processes.github.io/site/notebooks/L6/network-analysis.html
- For examples on graph parsing with OSMNX and other useful features of the library: https://github.com/gboeing/osmnx-examples/blob/main/notebooks/00-osmnx-features-demo.ipynb