Analyzing Path finding Algorithms

COMP 4202 Project Report

Due: December 6th, 2022

Dawson Theroux

101106602

Table of Contents

Introduction

Algorithms and graph theory are fundamental pillars of Computer Science. Understanding how algorithms work along with their running times can be simple at times from a general point of view, but it can be difficult to fully visualize their advantages and disadvantages on a real dataset. The goal of this project was to implement many common path finding algorithms used by route planning apps such as google maps to analyze their average running times on real world data along with their search areas. In this case, the search area is defined as the edges and nodes explored by the path finding algorithm while finding the desired path in the graph. The algorithms that were implemented and tested where Dijkstra's Algorithm, A* search with Euclidian distance heuristic, A* search with Manhattan distance heuristic, Breadth Frist Search, Depth First search and Greedy Heuristic search. These algorithms were implemented in python and analyzed on road data for Ottawa, Toronto, and New York City. Although the results of this analysis were predictable from the start, it was very interesting to see the search area created by each of the different algorithms, as well as seeing the running time of each algorithm on the real-world road data.

About the path finding algorithms

Path finding algorithms are exploratory algorithms in which a graph comprised of edges and nodes is explored until some end goal is achieved [1]. In this case, the end goal is to find the destination node after while starting at some start node. Some path finding algorithms such as Breadth First Search, Dijkstra's Algorithm, and A* search can guarantee that they find the shortest path in the graph because they are able to guarantee that the shortest possible path is always the one being explored at a given time [2]. Graph search algorithms work by constructing a frontier of nodes to be explored after the current node and all search algorithms typically differ from each other by changing the way that the frontier works and is sorted.

Breadth First Search

Breadth First Search (BFS) works by starting at the start node and adding all its neighbors to the frontier, which is implemented as a FIFO (First In First Out) queue. It then pops the next node in the FIFO, sets itself as the parent of the node chosen, and explores the node from the FIFO queue by adding all its neighbors to the FIFO queue. This is repeated until the end node is achieved and it is then simply a task of reversing through the parent nodes of all the nodes starting from the end node to get the path from start to finish. This implementation can be found in *BreadthFirstSearch.py.* Since a node at depth 2 is only explored after all nodes at depth 1 are explored, BFS can guarantee that the path found from the start node to the end node traverses the least number of edges, since no other path in the graph with less edges remains unexplored. For this project, this does not necessarily mean that it will find the shortest path since it does not take edge length into account. Instead, it means that it will find the path that traverses the least number of edges.

Dijkstra's Algorithm

Dijkstra's Algorithm works similarly to Breadth First Search, but with one small difference. Instead of using a FIFO queue as its frontier, it uses a priority queue to store the frontier, which keeps the edges sorted by the path cost to get a given node. This priority queue allows Dijkstra's algorithm to guarantee that the shortest path between two nodes (based on the sum of edge weights in the path) will be found. This is guaranteed in a similar way to BFS in that since no unexplored shorter path exist in the graph, it is impossible that Dijkstra's finds a path that is not the smallest from the start node to the end node [2]. See *DijkstrasWithExplored.py* for its implementation in this project.

A Search Algorithm*

The A * search algorithm differs the most from the other algorithms as it is a heuristic-based path finding algorithm. In this case, a heuristic is a function which can predict the quality of a node in relation to the end goal. For example, the heuristics explored in this case are the Manhattan distance to the goal and the Euclidean distance to the goal. These heuristics enhance the priority queue by being added to the path cost when adding them, meaning that nodes are sorted by path cost + heuristic in the Priority Queue. Not all heuristics are the same though, as they can be Admissible and Consistent. Admissible heuristics are ones which for all nodes, the heuristic function returns a value that is less than or equal to the remaining path cost to the goal. Consistent heuristic functions are one in which for any node n, for all subsequent nodes n' the heuristic function will return a value less than the path cost from n to n' plus the heuristic of n'. All consistent heuristics are admissible, but not all admissible heuristics are consistent [3]. Consistent heuristics are regarded as optimal as they can guarantee that the shortest path between two points is always found because all paths shorter than the path currently being explored have already been explored previously. For this project, two heuristics which are both admissible and consistent heuristics were explored, the Manhattan distance and the Euclidian distance. See *AStarWithExplored.py* for its implementation in this project.

Greedy Heuristic Search

Not all path finding algorithms can find the shortest paths, instead some are optimized for search time. One such case is greedy heuristic search, a search algorithm that leverages a priority queue sorted only by a heuristic function in order to find the path from a start to an end node. This algorithm leads to very small, explored sets and very quick running times at the cost of not finding the shortest path. See *GreedyHeuristicSearch.py* for its implementation in this project.

Depth First Search

Finally, the last search algorithm explored was Depth First Search. This search algorithm was added to explore the effects of search algorithms that do not find the shortest paths and how that effects the running time of the algorithms. Depth First Search (DFS) works in the opposite way of BFS in that the frontier is implemented using LIFO (Last in First Out). This means that DFS will start by exploring the start node and adding all its neighbors to the frontier. It will then explore the first node in the frontier and add all its neighbors to the frontier, and so on. Since DFS uses a FIFO, it ends up exploring to the deepest depths of the graph before ever checking the neighbour of the root node. This leads to some interesting running times on the different datasets since this algorithm does not necessarily find the shortest path. See *DepthFirstSearch.py* for its implementation in this project.

Implementation

The implementation of this project was achieved using Python, some Python libraries, and ArcGIS for dataset acquisition. The project repository can be broken down into three components: The input data, the Path finding algorithms, and the *PathAnalyzer*. All the data is stored in shapefiles in the "OttawaData", "NYCData" and "TorontoData" project folders. These files are read by the *PathAnalyzer.py* program script and are used to generate the output data, such as algorithm execution time, and explored area images. The path finding algorithms are implemented though five python files: *BreadthFirstSearch.py, DepthFirstSearch.py, DijkstrasWithExplored.py, GreedyHeuristicSearch.py, and AStarWithExplored.py*. These files all contain the respective path finding algorithm and return the path from the start to the end node as well as the search area explored while finding that path. Finally, the *PathAnalyzer.py* is a script which reads in all the Shapefile data, runs the path finding algorithms, and produces the output data and images.

Since the implementation was meant to use real world road data, Road data had to be found and modified to fit this project. To save computing time, I used the ArcGIS catalog online portal to download the road data, then trimmed them down to a specific area of interest. Once the datasets where of an acceptable size, I used the export features tool to save it to my project repository. Now, with generated input data, the *PathAnalyzer.py* script was written to take in shapefile data.

PathAnalyzer.py is the main Python file which manages reading in all the Shapefiles, generating their graphs, running the various path finding algorithm on the generated graphs, and displaying/outputting the results. All this functionality is implemented through the *runTests()* function. The *runTests()* function starts by calling the *generateGraphFromShape()* function, which uses a package called NetworkX to read the Shapefile data and generate a NetworkX graph [4]. Using this graph, the *findStartAndEnd()* function is called to determine a set of start and end nodes for the graph.

When determining start and end points, the goal was to maximize the analytical value of the produced graph instead of mapping paths between two real locations. In this case, the starting node is determined to be the node that is the closest to the average Y value of the graph and has 10% of the graph between itself and the left side of the graph. The end node is determined the same way, except it has 10% of the graph to its right. This maximizes the path length while also allowing us to visualize how the algorithm back tracks when finding the path. With the start and end nodes determined, *runTests()* can run the path finding algorithms to generate the shortest paths and explored graph.

Although many libraries already exist which include the path finding algorithms implemented as part of this project, none of them allow you to access the explored sequence of nodes and edges. Therefore, these algorithms had to be reimplemented to include this functionality and was managed by creating a NetworkX Graph object at the start of each path finding algorithm and add the nodes and edges to this graph object whenever they are explored. The path finding algorithms return the explored NetworkX Graph objects along with the list of nodes in the path which allows the *runTests()* function to start outputting the results from the test.

The *plotMap()* function takes in the full map Graph object, the explored Graph object, and the path Graph object and saves an image of all these maps overlayed on each other. This is accomplished using matplotlib library which can generate images from graphs. The plot generates the figure by first adding the entire roadmap graph in black. It then overlays the explored graph in red on top of the entire roadmap. Finally, it adds the path by the algorithm in green on top of all the previously added data. This results in an image containing the entire road data, the explored area, and the path all in the same image.

Dataset	**Number of nodes**	**Number of edges**
Ottawa	18884	25645
Toronto	46073	68353
New York City	55126	90789

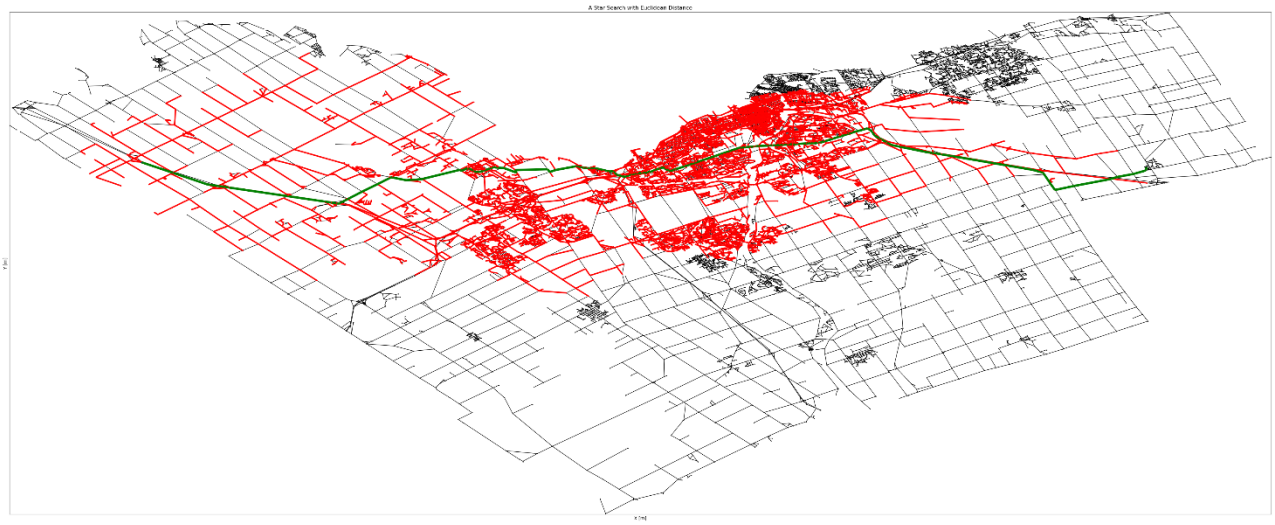
PathAnalyzer.py Output

The following is the output from the *PathAnalyzer.py*. All images are included in the submission in full resolution in the output file and are organized by dataset. The following table is the running time in seconds of each of the algorithms on each of the datasets.

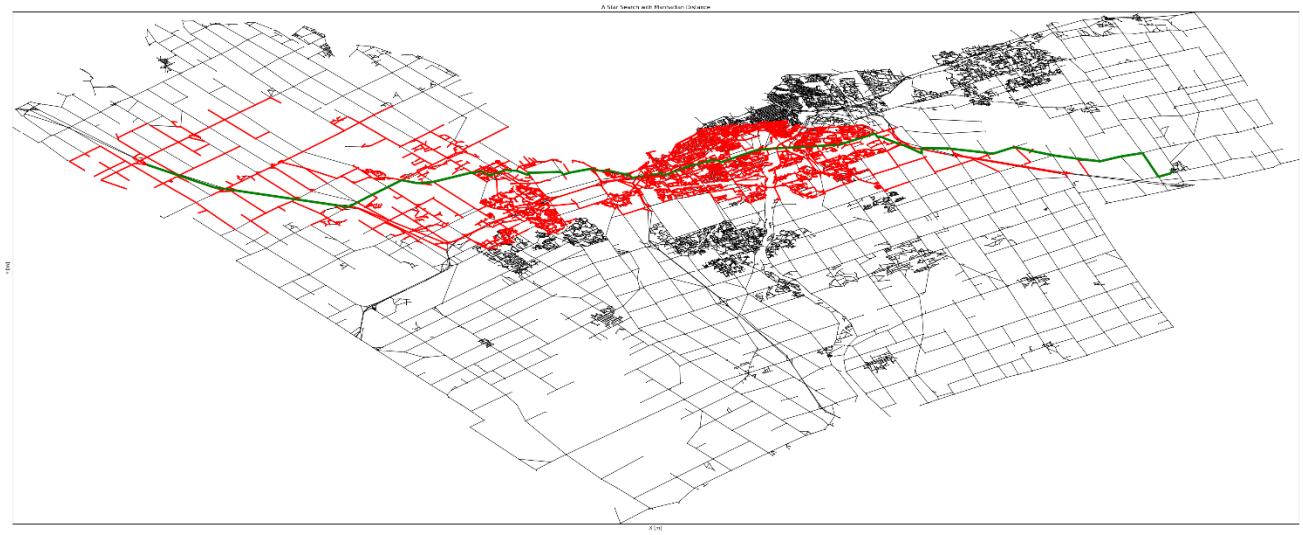
Algorithms:	**Datasets:**		
	Ottawa	Toronto	New York City
A* (Euclidean)	12.415	51.979	50.022
A* (Manhattan)	7.090	31.799	18.504
Dijkstra's Algorithm	15.446	163.226	170.309
Breadth First Search	15.370	213.446	281.728
Greedy Heuristic Search (Euclidean)	0.153	0.808	3.109
Greedy Heuristic Search (Manhattan)	0.308	1.747	5.450
Depth First Search	4.227	10.275	84.516

Ottawa Dataset

A* search with Euclidean Distance Heuristic function

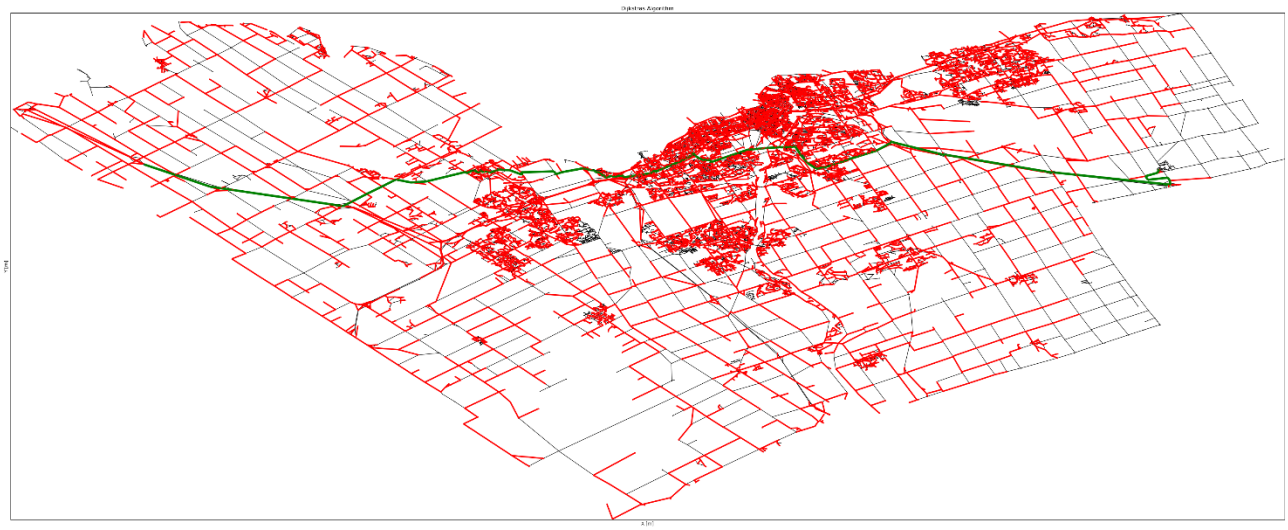


A* search with Manhattan Distance Heuristic function

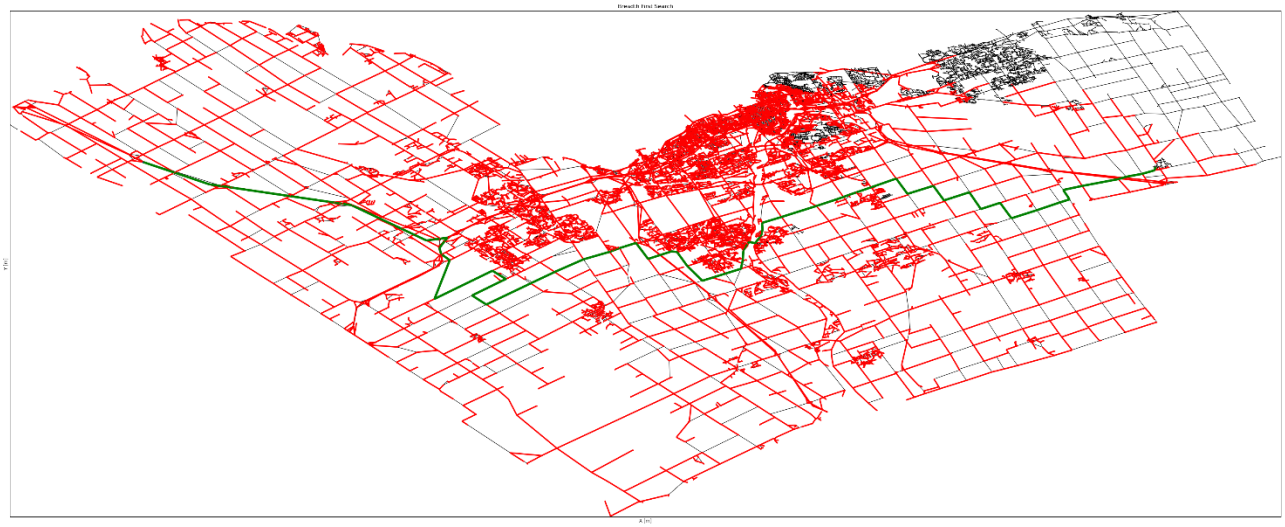


Ottawa Dataset:

Dijkstra's Algorithm:



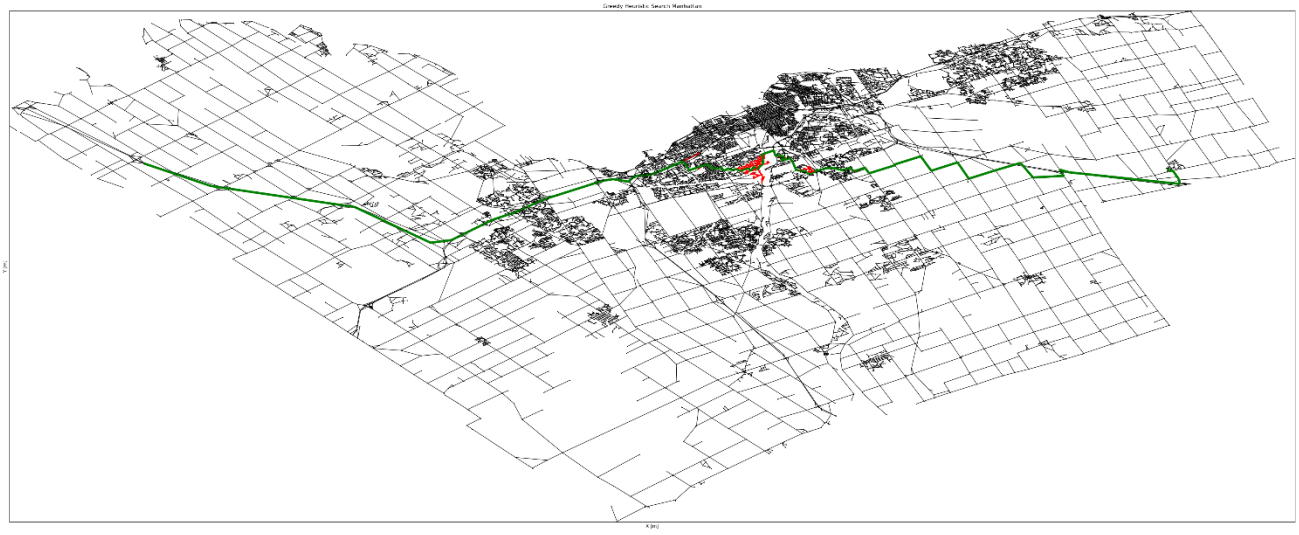
Breadth First Search:



Ottawa Dataset:

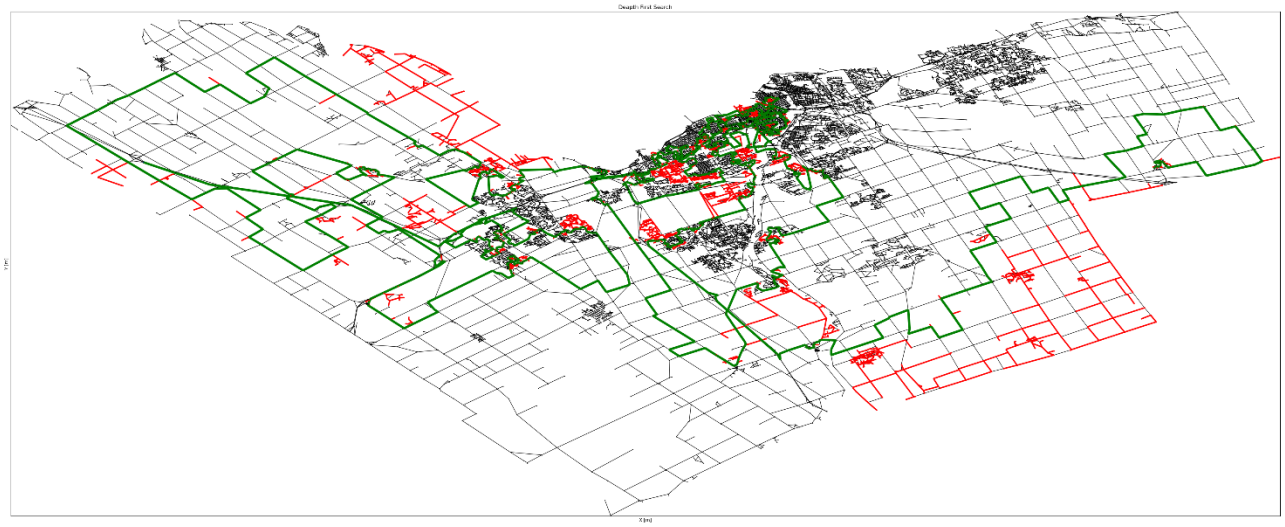
Greedy Heuristic Search with

Greedy Heuristic Search (Manhattan Distance):



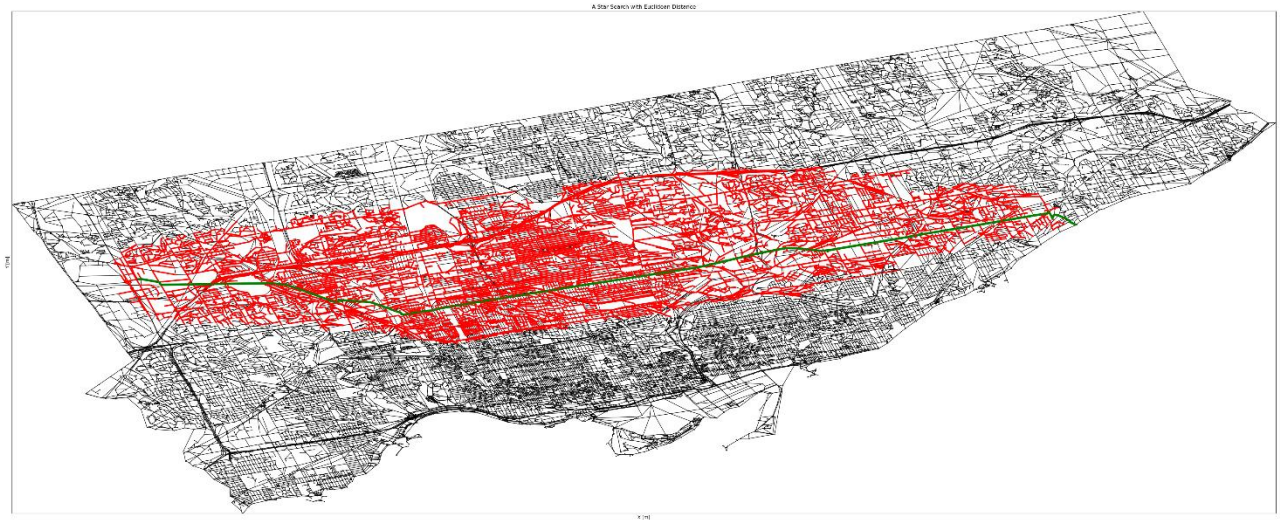
Ottawa Data:

Depth First Search:



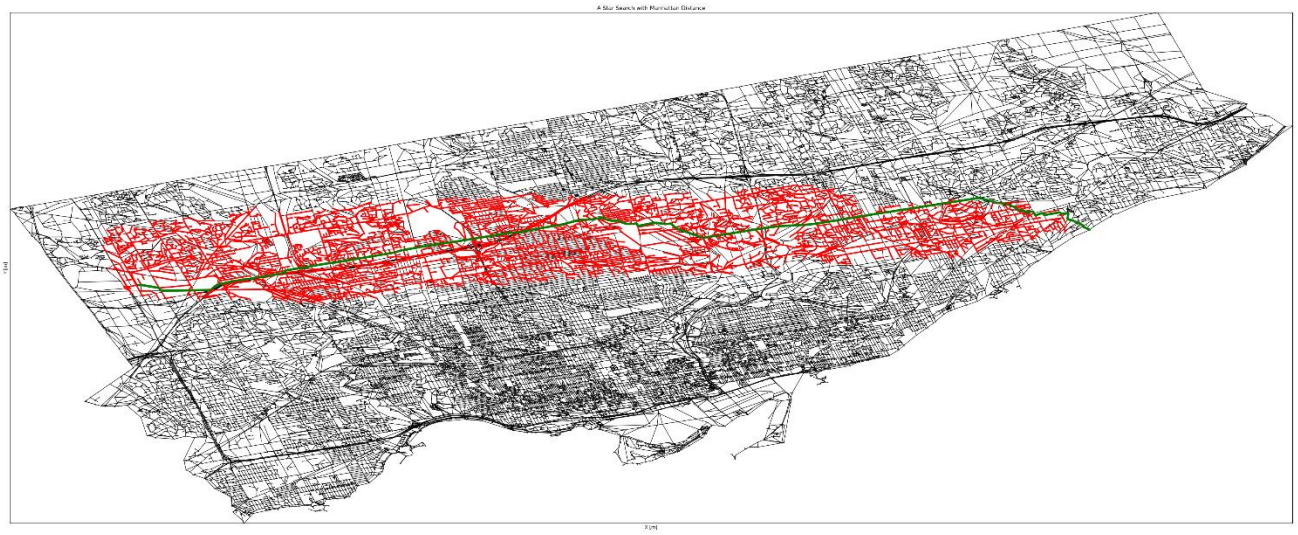
Toronto Dataset

A* Search with a Euclidean Distance Heuristic Function:



Toronto Dataset:

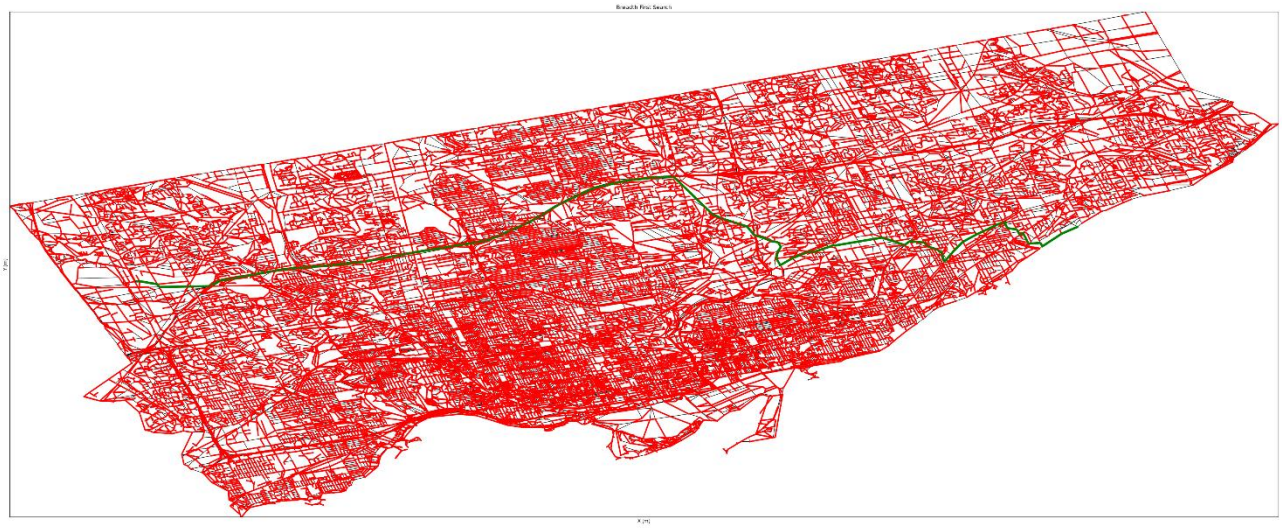
A* Search with Manhattan Distance Heuristic:



Dijkstra's Algorithm:

Toronto Dataset:

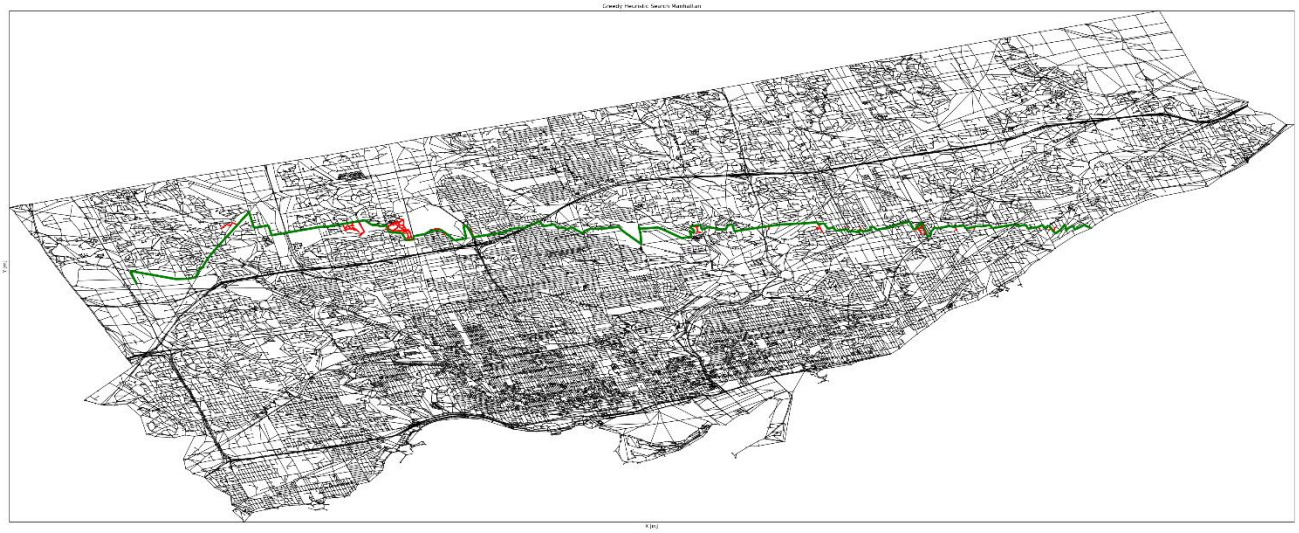
Breadth First Search:



Greedy Heuristic Search with Euclidean distance Heuristic function:

Toronto Dataset:

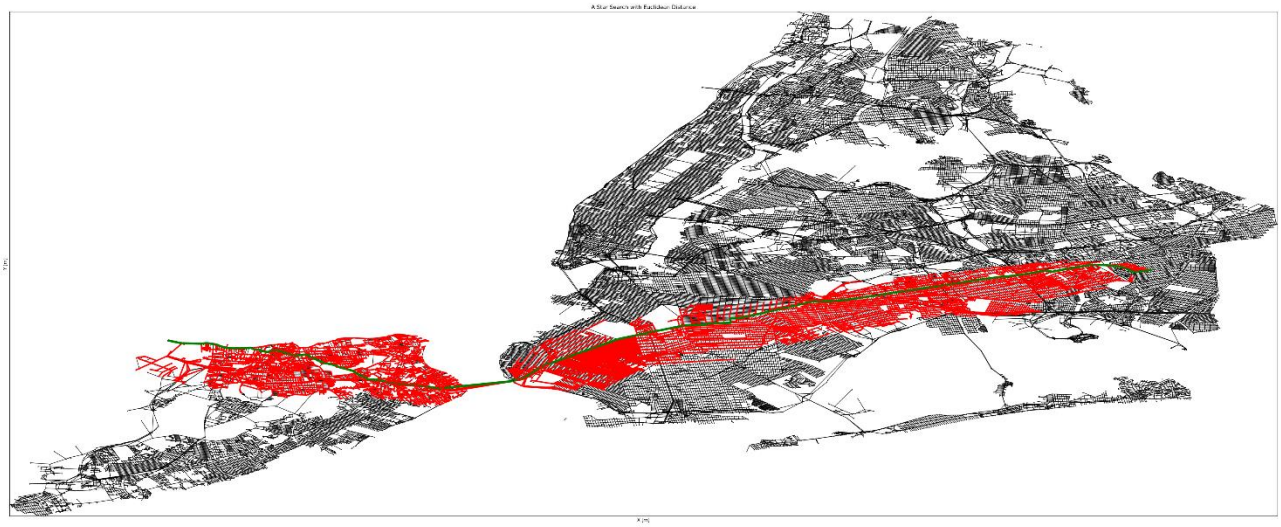
Greedy Heuristic Search with Manhattan Distance Heuristic function:



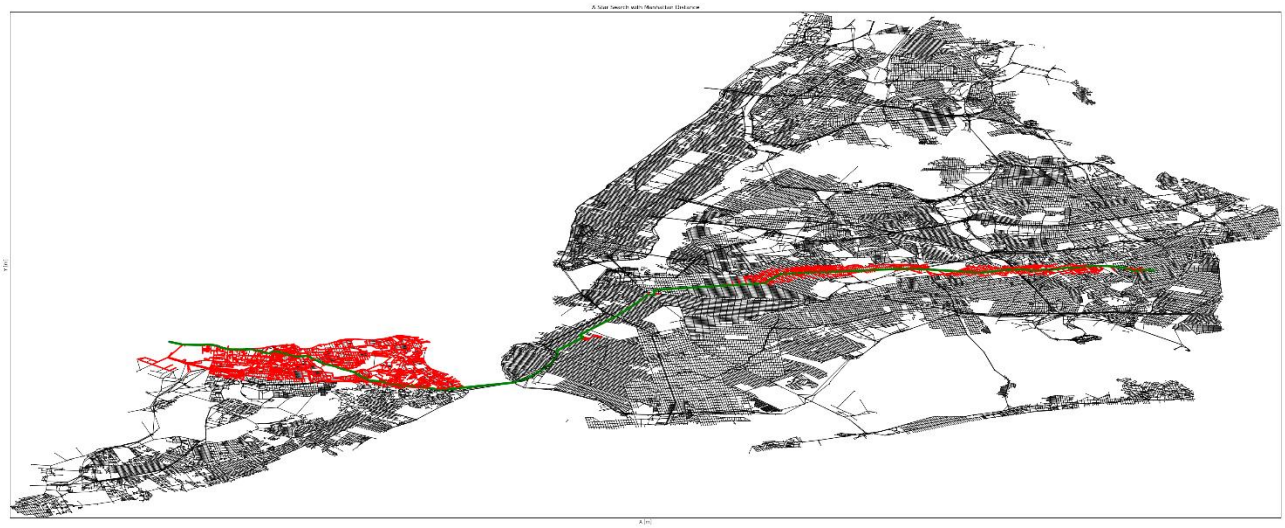
Depth First Search:

New York City Dataset

A* Search with Euclidean Distance Heuristic:

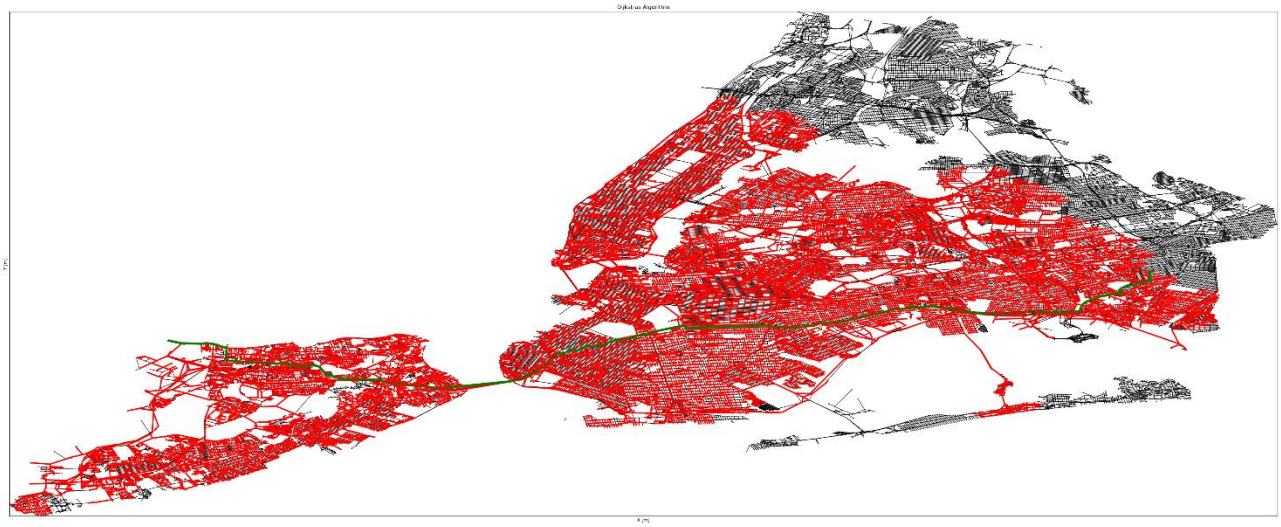


A* Search with Manhattan Distance heuristic function:

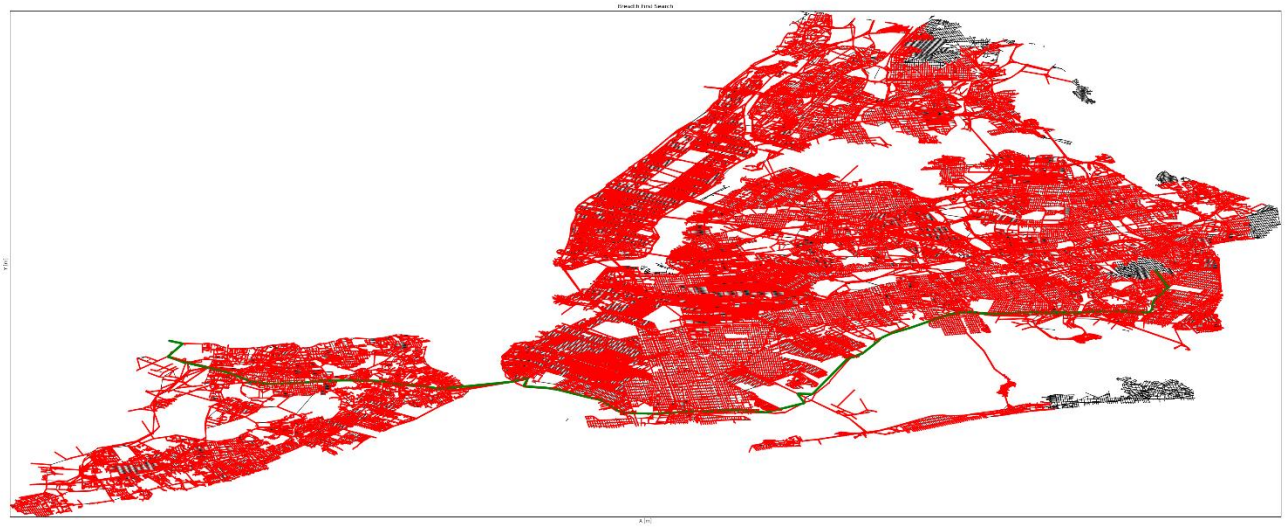


New York City Dataset:

Dijkstra's Algorithm:



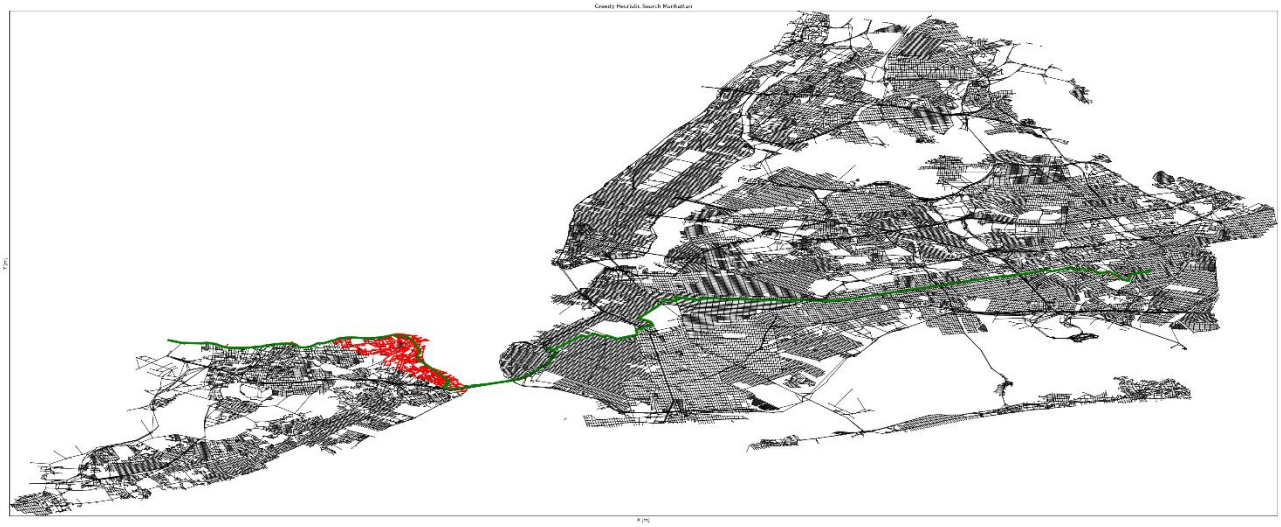
Breadth First Search:



New York City dataset:

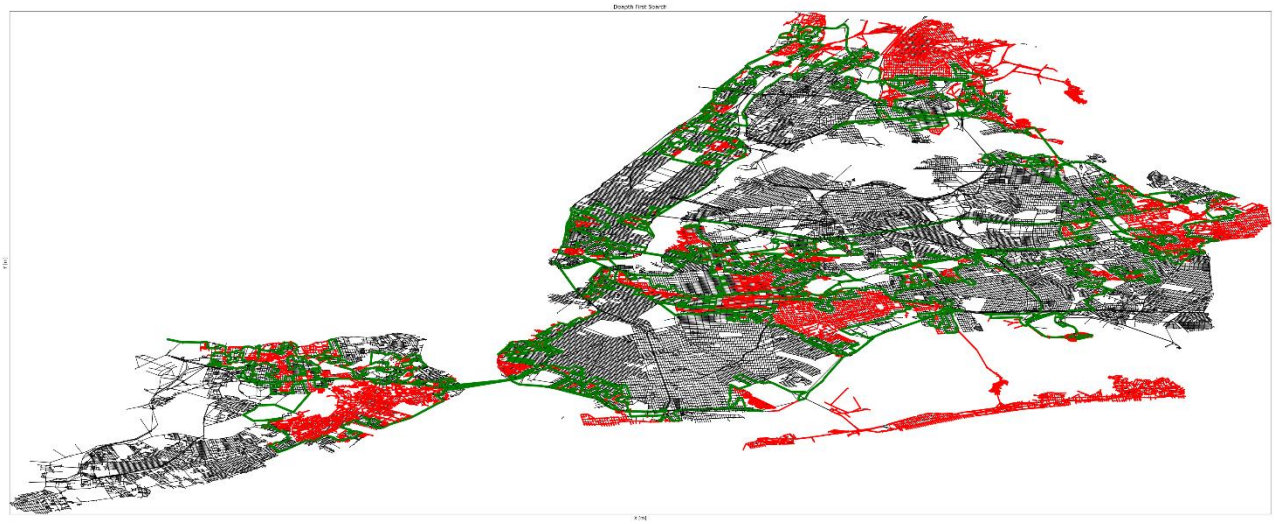
Greedy Heuristic Search with Euclidean Distance heuristic:

Greedy Heuristic Search with Manhattan Distance heuristic:



New York Dataset:

Depth First Search:



Analysis

The following analysis was done on the output data of the *PathAnalyzer.py* as seen in the section above. The full output of the program can be found in the output subdirectory of this submission.

Non-shortest Path Algorithms

Guaranteeing that the shortest path is found can greatly affect the efficiency of a path finding algorithm. By this, I mean that the explored area and running time of the Greedy Heuristic Search is magnitudes smaller than the fastest guaranteed shortest path algorithm, A* Search. The explored area of Greedy Heuristic search that is not used towards the final path is extremely minimal and is mainly comparable only partially comparable in size to Depth first search, while still maintaining a relatively short path to the goal. Even in its worst case in the New York City data, greedy heuristic search with the Euclidean distance heuristic finds a path from the start to the end node six times faster than A* search with the Manhattan distance heuristic.

Another standout result in the data is how little Depth First Search must back track before finding a path from the start to the end node in the dataset. I believe this is the case because the start and end points lie at close to maximum depth from the start node and since Depth First Search searches at depth first, these examples are close best case for this algorithm. This hypothesis is even further confirmed when comparing the results from DFS on Toronto data vs. New York City data. Since the Toronto dataset start and end points lie on the edge of the graph, the explored edges are even smaller than in the New York City data where the points are slightly more centralized. Another aspect of DFS that surprises me is the "shape" of the paths found. It is evident that the nodes added to the frontier are in random order because the path zigs and zags across the entire dataset resulting in a very long twisting and turning path. Most of the running times in the table are close to the average running time of the algorithm except for the DFS running time. The running time of the New York City dataset is magnitudes larger than for the other datasets. This is most likely the case because as the data gets larger, it is less likely that DFS will get lucky and find a path in an efficient manner. This means that in a real-world case (a dataset containing road data for more than just one city), it would presumably be almost impossible for DFS to find a path.

Dijkstra's and Breadth First Search

Breadth First Search and Dijkstra's algorithm both seemed to produce similar results, which is understandable because they are both very similar algorithms. What is a little bit surprising though is how the running time scales with the number of nodes and edges in the map. It appears Dijkstra's is slightly more efficient in that it can handle the larger datasets slightly better than BFS because its running time increases more smoothly as the dataset size increases. One possible reason for this is the difference in how path cost is calculated for both algorithms. Since Dijkstra's uses the distance travelled so far instead of the number hops on the path, it will be slightly more efficient on large dataset that can possibly have many hops because it can eliminate some of the unnecessarily long possible hops. i.e., the 401 in the Toronto dataset. Interestingly, this fact worked in favour of BFS in the Ottawa data where the complex nature of downtown meant that it avoided searching too far because the number of hops was too large. This led to the running time of BFS and Dijkstra's close to the same for this dataset.

Since the term “Shortest Path” differs for Dijkstra’s and BFS, in that Dijkstra’s takes into consideration the edge length while BFS only considers the number of hops. This fact is very evident in the paths they produce through the data. In the Toronto and NYC datasets, it appears as though BFS is avoiding certain parts of the city. Presumably, this is because it is avoiding areas in which there are many intersections, which means that the path cost would increase very quickly for BFS. On the other hand, Dijkstra’s will gladly take the mode direct path through the center of the city where all the intersections are because it only cares about the distance from the start to the end, and not how many nodes it has visited.

A* Search and Heuristic functions

A* search is presumed to be one of the main algorithms used by google maps in order to generate the routes when querying a path from some start point to some destination. One possible reason for this could be the fact that an admissible and consistent heuristic results in little wasted time exploring paths that are evidently wrong. It is also possible that the route planning apps can tune their heuristic functions to consider past traffic records to be able to cater the path finding algorithm to the shortest travel time. No matter the case, A* search is evidently the most efficient shortest path algorithm explored in this project solely because it can guarantee that the shortest path is computed and while keeping the explored areas so small. On all the datasets, A* Search never explores backwards from the starting point and never deviates too far of the optimal path. The heuristic function almost acts like a magnet to bias the priority queue enough in the direction of the goal while still allowing enough new candidate paths to be explored to guarantee the shortest path. This is further magnified when using the Manhattan distance heuristic which produces an even smaller explored area than the Euclidean distance heuristic, which in turn leads to a faster execution time. This is extremely evident in the New York City dataset where the explored area only deviates off the path twice for a long while after exiting the Verrazzano-Narrows Bridge. When deciding which datasets to use to test these algorithms, I thought it would be interesting to use New York City because I am testing a Manhattan distance heuristic and Manhattan is in New York City. Interestingly, this dataset resulted in some of the most compelling data for the Manhattan distance heuristic in regard to the A* algorithm, presumably because of the relation between Manhattan distance and the many city blocks of the city. Looking at this small sample set of data, I would guess that the Manhattan distance heuristic is the better heuristic than Euclidean distance for A* search because it outperforms the Euclidean distances for all the datasets.

Although the Manhattan distance heuristic seems to work better with A* search, the case was not the same for Greedy Heuristic search. Unlike A* Search, Greedy Heuristic Search entirely relied on the heuristic to make its way towards the goal. This results in extremely quick running times at the cost of not finding the shortest path, which may be useful when first determining if there is a path between two nodes. Since Greedy Heuristic search relies entirely on the heuristic function, it makes sense that a more accurate representation of the distance from the current node to the end goal would result in more efficient running time and a smaller explored area. It is for that reason that I believe we can see better running times and less unnecessary explored edges when Greedy Heuristic search uses Euclidean distance as its heuristic instead of Manhattan distance. This also explains why the path found with greedy heuristic search with the Manhattan heuristic is less straight than the one found with Euclidean distance. This fact is extremely evident in the New York City Dataset where the path after the Verrazzano-Narrows Bridge heads north for much longer with the Manhattan distance heuristic function

compared to with the Euclidean distance heuristic function because of the way Manhattan distance is affected by traveling directly North or South.

Conclusion

While implementing this project I learnt a lot about the inner workings of many different path finding algorithms. Such as the way various ways that the frontier implementation can affect the running time and demeanor of a search algorithm. Like how if you implement the frontier with a FIFO queue, it results in Breadth First Search, a search algorithm that can find a path between two nodes in a graph in the shortest number of hops. Then, if you implement that same path finding algorithm with a LIFO queue, it results in Depth First Search, an algorithm that does not find a useful path in exceedingly more time as the dataset grows. Prior to this project, I could not have guessed how the search areas could differ in shape between path finding algorithms, or even the same path finding algorithm with different search heuristics. Now, after implementing the different algorithms and visualizing how these small changes effect the running time and search area, I believe I understand at a deeper level how they are all able to find the shortest path in their running time. Plotting the explored graphs between Manhattan and Euclidean distance also demonstrated how Euclidean distance and Manhattan distance increase with respect to the location of both the points. Finally, it was shocking to see the running time of Greedy Heuristic Search. I briefly explored this algorithm in a previous class, but never implemented myself and was surprised at how it was able to find the path between two nodes while rarely having to backtrack. Implementing, visualizing, and timing these 5 graph search algorithms has enabled me to gain a deeper understanding of graph search and graph theory as a whole.

Running Instructions

Although running the script will net the same results (if no values are changed) here are the instruction to run it. By default, it will run on the Ottawa dataset as it is the quickest to run. Running another dataset can be done by uncommenting code from the main function.

1. Install Python 3.10
2. Install the conda environment: https://conda.io/projects/conda/en/stable/user-guide/install/download.html
3. Install the correct conda environment with the command:
    a. “conda install -c conda-forge gdal”
4. Install the NetworkX python package with pip:
    a. “pip install networkx”
5. Install Scipy python package with pip:
    a. “pip install scipy”
6. Enter the conda environment:
    a. “conda activate OSMNX”
7. Run the PathAnalyzer.py file with python:
    a. “python PathAnalyzer.py”

Bibliography

[1]“Path finding algorithms - developer guides,” *Neo4j Graph Data Platform*. [Online]. Available: https://neo4j.com/developer/graph-data-science/path-finding-graph-algorithms/. [Accessed: 06-Dec-2022].

[2] “Dijkstra algorithms,” *Dijkstra Algorithms - an overview | ScienceDirect Topics*. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/dijkstra-algorithms. [Accessed: 06-Dec-2022].

[3] “Admissible heuristic,” *Engati*. [Online]. Available: https://www.engati.com/glossary/admissible-heuristic. [Accessed: 06-Dec-2022].

[4] “Reference#,” *Reference - NetworkX 3.0rc2.dev0 documentation*. [Online]. Available: https://networkx.org/documentation/latest/reference/index.html. [Accessed: 06-Dec-2022].