A-star vs. Dijkstra for Path Finding

Sparsh Bansal and HK Rho

May 6, 2020

1 Motivation

The idea for our final project was developed from the moment when our professor, Alice, briefly went over the different performance of path-finding algorithms from a variety of platforms such as Google Maps and Bing. Initially, we immensely scaled this idea so that we would compare which path-finding algorithm from different platforms performs the best at given situations; however, with feedback, we scaled down to comparing the performance between Dijkstra's algorithm and the A-Star Heuristic algorithm.

Our learning goals for this project were to observe how to program and interpret the heuristics, specially those that go into path-finding algorithms like traffic and journey time optimization. We were also interested to look at the implementation of the heuristics and data processing to prepare information for the algorithm to use.

2 Background Research

With this implementation-based project, we wanted to explore the difference in results between Dijkstra's algorithm and the A-Star Heuristic algorithm for path-finding applications by exploring different kinds of heuristic functions that are talked about in detail in the algorithm section. The dataset used to generate a graphical representation of the streets of Boston and to query their latitude and longitudinal coordinates was from a street-sweeping schedule. This dataset contained information about the starting street and ending street for the sweeping, and the distance between them. We also looked at the peak traffic times and the traffic-heavy areas in Boston city, and implemented the traffic heuristic accordingly. The references to the datasets that we used are towards the end of this report.

3 Algorithm Description

Our GitHub repository is linked here.

Dijkstra's Algorithm

The pseudo-code for Dijkstra's algorithm can be found below:

```
        def dijkstra(start, end):
            # initialize dict() to store visited/unvisited status
            visited_status = dict()
            # initialize dict() to store previous node
            prev = dict()
            # initialize dict() to store distance to that node
            dist = dict()
```

```
# initialize priority queue to store the unvisited nodes
queue = []

# initialize the dictionaries
for node in total number of nodes:
    visited_status[node] = False
    prev[node] = None
    dist[node] = 100000.0

# update information for start node
queue.append(start)
dist[start] = 0.0
visited_status[start] = True

# iterate until the queue is empty
while len(queue) > 0:
    curr = queue.pop(0)
    visited[curr] = True
    queue.append(neighbors of curr)

    for i in range(len(queue)):
        if dist[curr] + edge(curr, i) < dist[i]:
            # update distance
        else:
            # keep the original distance
            queue.pop(i) # get rid of the analyzed nodes

    # find all the neighbors of currently analyzing node
    curr_neighbors = G.neighbors(curr)

    for next_node in curr_neighbors:
         # find the closest neighbor
         # add the closest neighbor to the priority queue

    # update dictionary of visited status

```

A-Star Heuristic Algorithm

The pseudo-code for the A-Star Heuristic algorithm can be found below:

```
def astar(start, end, heuristic):
# initialize dict() to store visited/unvisited status
visited_status = dict()
# initialize dict() to store previous node
prev = dict()
# initialize dict() to store distance to that node
dist = dict()
# initialize priority queue to store the unvisited nodes
queue = []

# initialize the dictionaries
for node in total number of nodes:
    visited_status[node] = False
    prev[node] = None
    dist[node] = 100000.0

# update information for start node
queue.append(start)
dist[start] = 0.0
```

```
            visited_status[start] = True

            # iterate until the queue is empty
            while len(queue) > 0:
                # end when the current node == destination node
                if curr == end:
                    # return the path

                curr = queue.pop(0)
                visited[curr] = True
                queue.append(neighbors of curr)

                for i in range(len(queue)):
                    if dist[curr] + edge(curr, i) < dist[i]:
                        # update distance
                    else:
                        # keep the original distance
                        queue.pop(i) # get rid of the analyzed nodes

                # find all the neighbors of currently analyzing node
                curr_neighbors = G.neighbors(curr)

                for next_node in curr_neighbors:
                     # find the closest neighbor (including the heuristic)
                     # add the closest neighbor to the priority queue

                # update dictionary of visited status

```

The pseudo-code for A-Star Heuristic algorithm is almost identical to the pseudo-code for Dijkstra's except two differences: the A-Star Heuristic algorithm stops as soon as the currently analyzing node is the destination node, and it includes a heuristic to find the closest next step. The heuristic has two parts to it - one is the most common path-finding heuristic, the Euclidean distance between the potential node and the destination node. The other is the traffic heuristic which returns a traffic index for the time of the day when the algorithm is being used, and offsets the optimality accordingly.

4 Results and Interpretation

4.1 Test Case I

For a particular test case, we tried to find the optimal directions from Diston Street to Pinckney Street in Boston. The output from different algorithms is summarized below.

Output from Djikstra's algorithm:

```
Directions using Djikstra Algorithm: ['Ditson St', 'Geneva Ave', 'Bowdoin St',
'Somerset St', 'Tremont St', 'Beacon St', 'Ashburton Pl',
'Cambridge St', 'Mount Vernon St', 'Pinckney St']
```

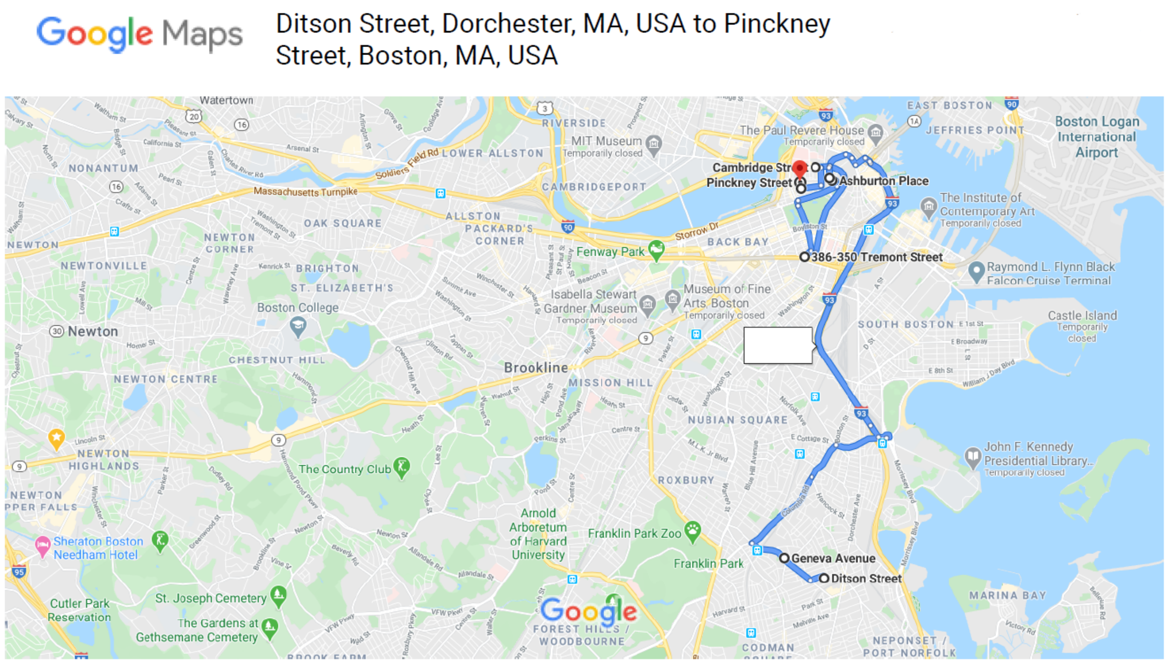

Figure 1: Node Spread from Ditson Street to Pinckney Street on Google Maps for Dijkstra's Algorithm for the rush hour period in the city

```
Directions using A-star Algorithm, with Euclidean Distance and Traffic
Conditions as a Heuristic: ['Ditson St', 'Geneva Ave', 'Dorchester Ave',
'Congress St', 'Blackstone St', 'Cambridge St', 'Mount Vernon St',
'Pinckney St']
```

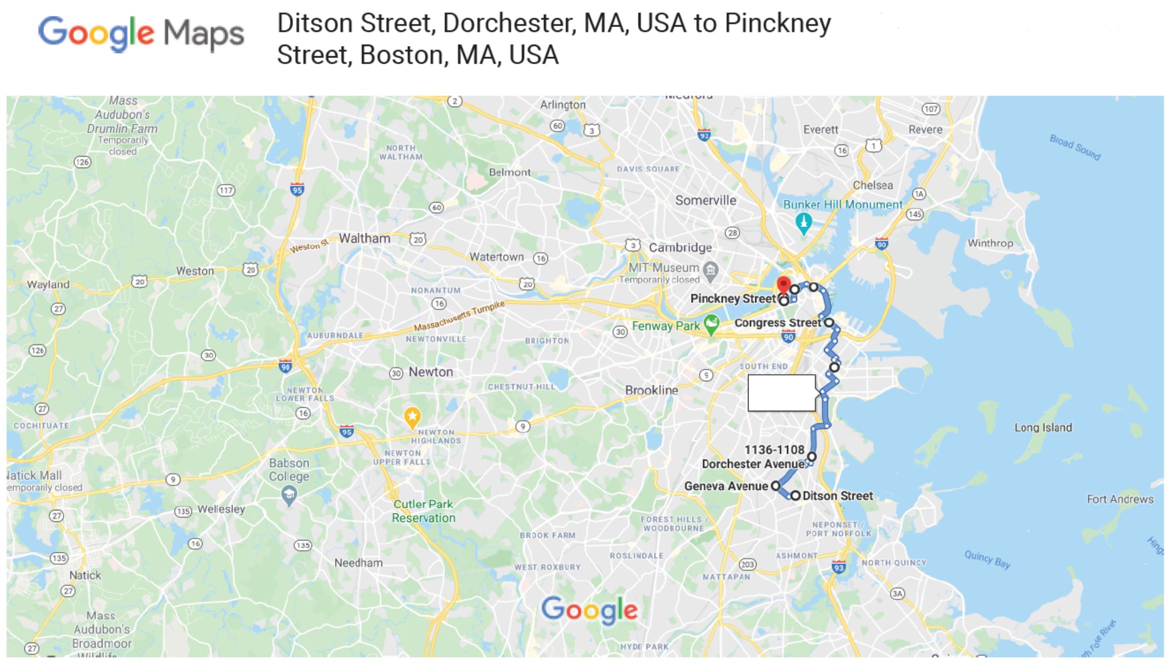
Google Maps
Ditson Street, Dorchester, MA, USA to Pinckney Street, Boston, MA, USA
Pinckney Street
Congress Street
1136-1108 Dorchester Avenue
Geneva Avenue
Ditson Street

Figure 2: Node Spread from Ditson Street to Pinckney Street on Google Maps for A-star Pathfinding Algorithm with Euclidean Distance and Traffic Condition Heuristics for the rush hour period in the city

Using the map figures, we can observe that the A-star heuristic algorithm is modifying the shortest distance path to incorporate the traffic conditions. The heuristic offsets the path so as to avoid the streets that directly cross the city (for our algorithm, it is the streets within a radius of 1 mile of the city).

For this particular result, we can observe that the A-star algorithm chooses to set the path around the city as much as possible before entering the heart of the city. It avoids going through Somerset Street, Tremont Street, Baecon Street, and Ashburton Place to avoid passing through the center of the city.

If we run the algorithm during the time of the day when there is lower traffic, we observe increasing similarity between the Dijkstra's algorithm and the A-star algorithm with decreasing traffic. If we consider no traffic conditions, the outputs are identical for both algorithms.

4.2 Test Case II

For a particular test case, we tried to find the optimal directions from Saratoga Street to Perkins Street in Boston. The output from different algorithms is summarized below.

Output from Djikstra's algorithm:

```
Directions using Djikstra Algorithm: ['Saratoga St', 'Princeton St',
'Chelsea St', 'Moulton St', 'Lowney Way', 'Tremont St',
'Beacon St', 'Ashburton Pl', 'Cambridge St', 'Perkins St']
```

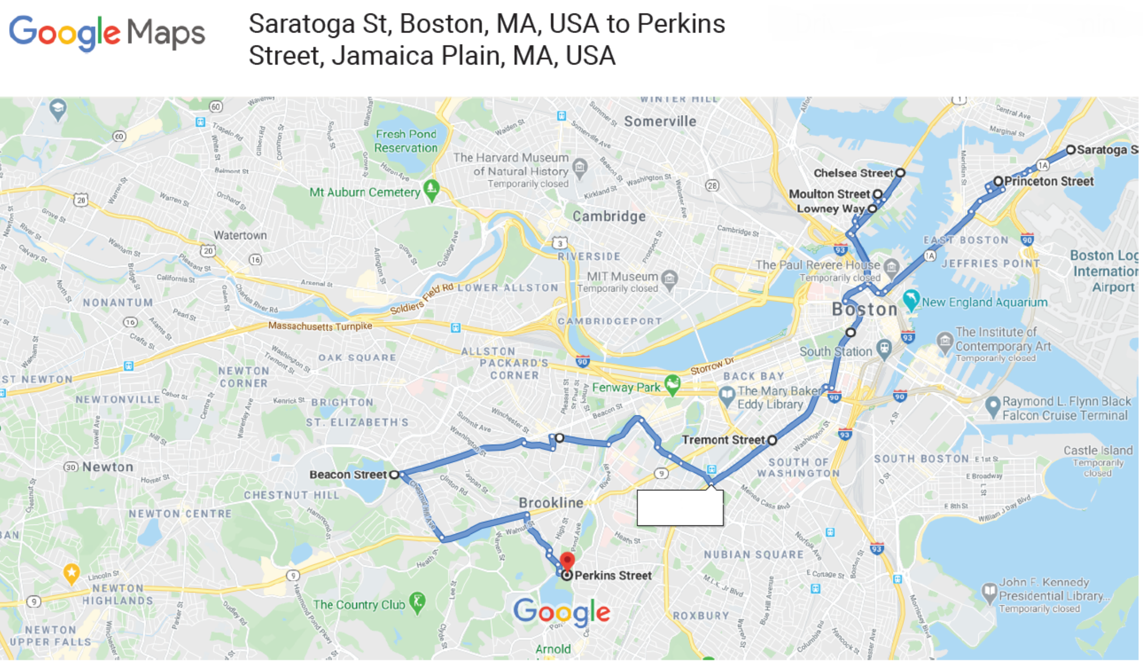

Figure 3: Node Spread from Saratoga Street to Perkins Street on Google Maps for Dijkstra's Algorithm for the rush hour period in the city

```
Directions using A-star Algorithm, with Euclidean Distance and Traffic
Conditions as a Heuristic: ['Saratoga St', 'Princeton St',
'Meridian St', 'Sumner St', 'Bakersfield St', 'Dorchester Ave',
'Washington St', 'Lamartine St', 'Perkins St']
```

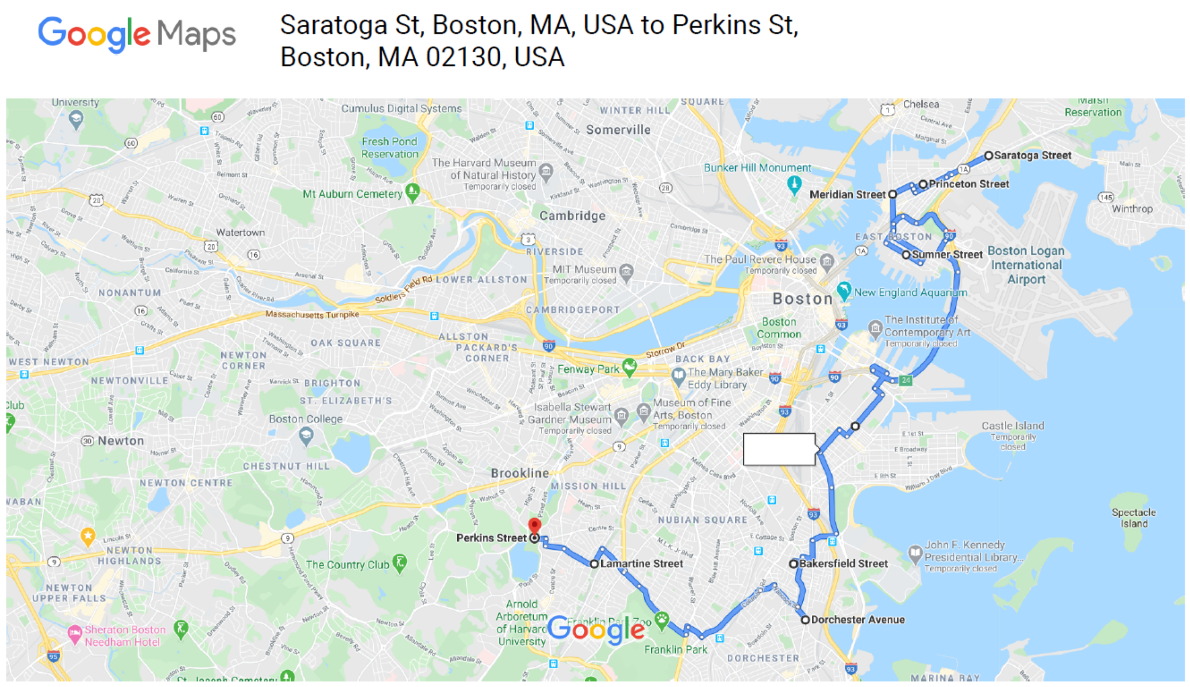

Figure 4: Node Spread from Saratoga Street to Perkins Street on Google Maps for A-star Path-finding Algorithm with Euclidean Distance and Traffic Condition Heuristics for the rush hour period in the city

Using the map figures, we can observe that the A-star heuristic algorithm is modifying the shortest distance path to incorporate the traffic conditions. The heuristic offsets the path so as to avoid the streets that directly cross the city (for our algorithm, it is the streets within a radius of 1 mile of the city). According to the algorithm, we will travel more distance using this method, but save on time.

4.3 Runtime Analysis

In order to compare the results of the algorithms, we have recorded the runtimes in ms (RT) for 1000 cycles, in the table below.

Performance Table				
Route	Dijkstra	NetworkX	A-star (EU)	A-star (EU+TC)
Washington St. - Charlotte St.	23.505	41.166	4843.088	2499.924
Washington St. - George St.	43.433	31.998	4452.326	2224.429
Ditson St. - Pinckney St.	26.063	61.019	2105.646	2253.431
Saratoga St. - Perkins St.	30.964	70.001	2071.651	2275.247

EU - Euclidean Distance Heuristic
TR = Traffic Condition Heuristic

For our implementation of the Dijkstra's algorithm, we found out that it is generally faster than the optimized version of the shortest path algorithm by NetworkX. Looking at the program repository maintained by NetworkX, we assumed that their program spends more time on computation compared to ours. Additionally, for a small percentage of the total tests that we conducted on our dataset, we observed slight differences in the optimal paths.

Secondly, we expected the heuristic algorithm to take a much longer runtime than the Dijkstra's algorithm implementation. In general, the a-star algorithm that we implemented took about two orders of magnitude more time relatively. We observe this behaviour for both the heuristic types that we have recorded in the Performance Table.

Thirdly, comparing the two heuristics (using Euclidean Distance with and without the Traffic Conditions) we noticed an irregular correlation. For some cases the traffic conditions would not cause the runtime to change by much. While for others, the runtime is comparable. One explanation for this could be that the high traffic streets have an effect of being ruled out from the possibility of making the quickest route, which could argue for reducing the total time, or even covering the time cost of calculating the traffic conditions on top of that.

5 Limitations

The first limitation of the performance of the algorithms that we implemented would definitely be the discrepancy between the dataset used to output the most optimal paths for our algorithms and the actual representation of the Boston streets. Not only was the dataset we used incomplete–did not encompass all the streets in Boston, but also had a good amount of insufficient information–rows that contained streets that were "Dead Ends", streets that had no distance between themselves, or empty street names. This lack of data resulted in the optimal paths output from our algorithms taking a longer route due to the absence of certain streets in our graph

The second limitation arose from the way we handled the errors when querying the street locations from a Python library, geopy. For we were unavailable to figure out why geopy was having trouble querying certain streets at certain times with our understanding, we ended up defaulting the location of a query-failed street to the center of Boston. This accounted for another difficulty in representing an accurate model of the Boston streets; therefore, resulting in a less accurate output.

The last limitation, traffic, is another factor that decreases the accuracy of our implementation. We have set arbitrary traffic values for different times during the day to see the different paths taken by the algorithms under the influence of traffic. However, we would like to emphasize again that these values were arbitrary, which does not reflect the actual traffic in the Boston streets.

6 Next Steps

To better improve our understanding of the performance of the Dijkstra's and A-Star Heuristic algorithm that we implemented, a logical next step for us would be looking into the optimality gap of our algorithms. As of now, we do not have a clear idea to find the correct, most optimal solution to the pathfinding problem as the solution is subject to change depending on two factors–traffic and time of day–right now in the implementation. However, we expect that exploring more into this area of finding the optimality gap would be as equally interesting.

7 References

Google Maps: `https://www.google.com/maps`
GeoPy Library: `https://geopy.readthedocs.io/en/stable/`
NetworkX Library: `https://networkx.github.io/documentation/stable/`
Dataset Source: `https://data.boston.gov/dataset/street-sweeping-schedules/resource/9fdbdcad-67c8-4b23-b6ec-861e77d56227`
Github Link: `sbansal22/A-star-versus-dijkstra-for-pathfinding`