Contents

1 Project Overview 2

2 Instructions on how to run code 4

3 Performance Comparison 6

4 Conclusion 10

1 Project Overview

For our project we implemented to A* algorithm and compared its performance against Dijkstra's algorithm. The data we chose to evaluate was flight costs to and from various airports. The data was downloaded as a *.csv file from the U.S. Bureau of Transportation Statistics. We used the flight data from Quarter 1 of 2021, which gave us over three million flights across 429 airports. To simplify the data, we took the average price between each airport which reduced the graph space to 59,241 edges across 429 vertices. To make our A* heuristic more relevant, we then removed all flights with a cost under 5 dollars , leading to a final graph containing 429 vertices and 58,974 weighted, directional edges. The final graph represents the average cost of a flight between each airport during Q1 of 2021. The pruning and organization of the data was done in SQL and exported as *.csv file, which was then imported into Python.

Implementation of Dijkstra's Theoram

We first used Dijkstra's algorithm as a baseline to compare the cheapest itinerary between various destinations. The pseudocode for the algorithm can be seen below:

Input = (G = (V, E), start node, end node)
Unvisited = list of all nodes
Shortest path = cost dictionary
Previous nodes = path list dictionary
For all u in V:

dist.(u) = inf

dist.(s) = 0

While Unvisited is not empty:

u = node with smallest Fscore in list

if u is (end node):

return path and cost

else:

if u has neighbors:

Fetch list of edges (u, v) from adjacency matrix

For all edges (u, v) in E:

If dist(v) > dist (u) + w(u, v):

Cost_dictionary(v) = dist(u) + w(u, v):

Add u to path list of v

Remove(u) from unvisited list

Implementation of A star

Next we implemented the A* algorithm using the minimum number of hops between two airports (obtained using a BFS search) multiplied by the cheapest flight in the dataset (dollar 5 for our data set) as the heuristic function. This guarantees that the heuristic function will always be equal to or less than the true cheapest path between two given airports. The pseudocode for the algorithm can be seen below:

Input = (G = (V, E), start node, end node)

Unvisited = list of all nodes

Shortest_path(G(x)) = cost dictionary

F_score = cost dictionary including heuristic

Previous _nodes = path list dictionary

For all u in V:

dist.(u) = inf

dist.(s) = 0

While Unvisited is not empty:

u = node with smallest F_score in list

if u is (end node):

return path and cost

else:

if u has neighbors:

Fetch list of edges (u, v) from adjacency matrix

For all edges (u, v) in E:

G(x) = dist. (u) + w (u, v)

H(x) = BFS (v, end node) * min_flight_cost

F(x) = G(x) + H(x)

If G(x) $\geq dist.(u) + w(u, v)$

Cost_dictionary(v) = dist.(u) + w(u, v):

F_score dictionary(v) = F(x)

Add u to path list of v

Remove(u) from unvisited list

2 Instructions on how to run code

We implemented the algorithms in Python, using the pandas library to read data from the *.csv file, and the queue library to easily implement a queue for the BFS search. To run the program, make sure your Python environment has those libraries imported, and make sure that the path to the two source *.csv files ("Flight_DataGreater_Than _5.csv" and "Nodes.csv"), located at the bottom of the code, are correct.

```
218  #reading the data form the *.csv files
219  flight_data = pd.read_csv (r'C:\Users\patri\PycharmProjects\Dijkstra\Flight_Data_Greater_Than_5.csv')
220  nodes = pd.read_csv(r'C:\Users\patri\PycharmProjects\Dijkstra\Nodes.csv')
```

The parameters affecting which source and destinations are compared are located at the bottom of the *main.py* file, in the following section:

```
#assinging the source and destination nodes
start = "XNA"
end = "YUM"
```

Simply change the text to a three letter identifier located in the node list or "Nodes.csv" to change which nodes are evaluated.
To run either A* or Dijkstra's algorithm, you first call the "a star" or "dijkstra" function, and then call "print_result" in order to display the results and besides this also print the stopwatch to measure elapsed time:

```
218  #reading the data form the *.csv files
219  flight_data = pd.read_csv (r'C:\Users\patri\PycharmProjects\Dijkstra\Flight_Data_Greater_Than_5.csv')
220  nodes = pd.read_csv(r'C:\Users\patri\PycharmProjects\Dijkstra\Nodes.csv')
```

A note on execution time: Dijkstra's algorithm typically takes around 10 seconds to execute, however A* can take around 1-2 minutes to display the result. This is because it is running BFS at every node it is evaluating.

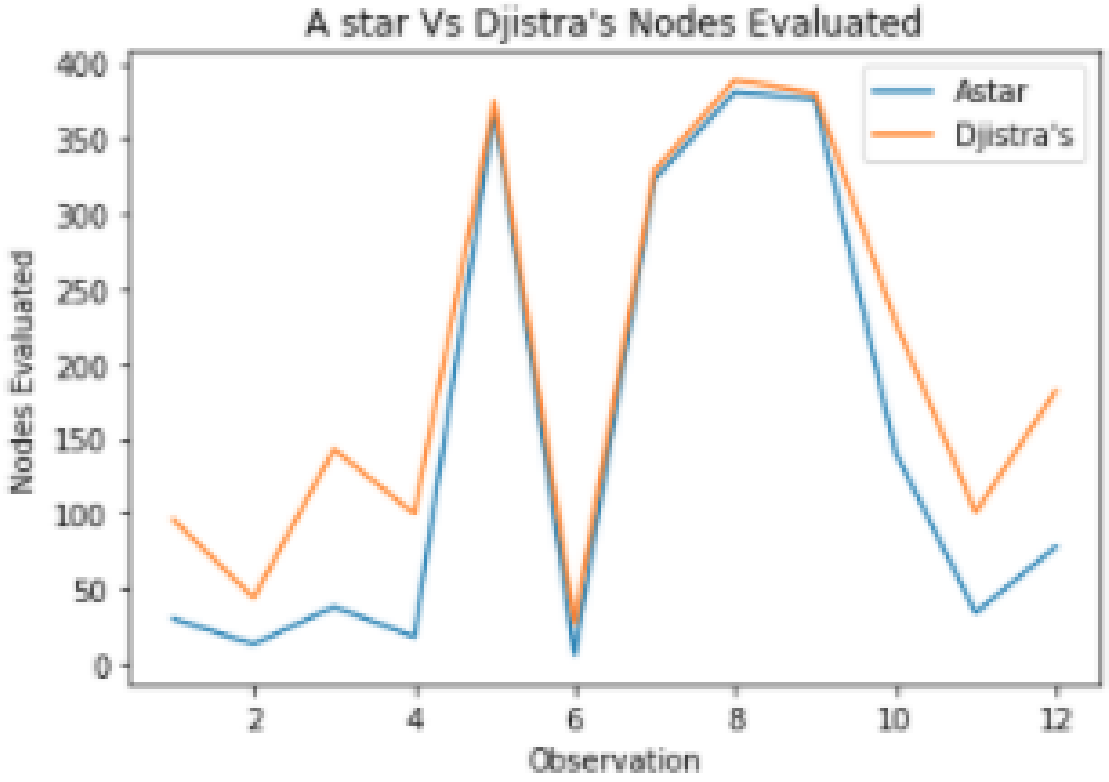
After executing above commands following output format we can see:

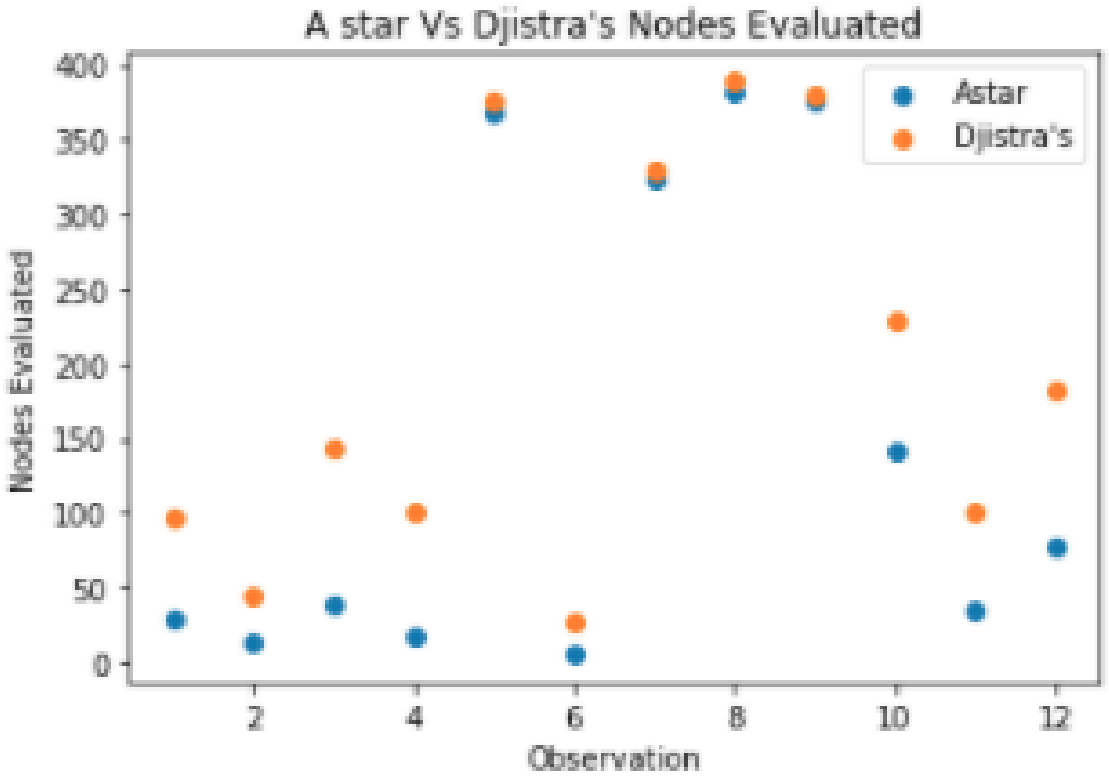
```
A* nodes evaluated = 78
The cheapest itinerary is $90.92
XNA -> PAE -> GUC -> CLL -> MYR -> CWA -> YUM
Time Diff 224.76465892791748
start 1639787551.551501
end 1639787776.31616
Dijkstra nodes evaluated = 182
The cheapest itinerary is $90.92
XNA -> PAE -> GUC -> CLL -> MYR -> CWA -> YUM
0.01880192756652832
Time Diff 0.01880192756652832
start 1639787776.3164952
end 1639787776.335297
```

3 Performance Comparison

Nodes Evaluated

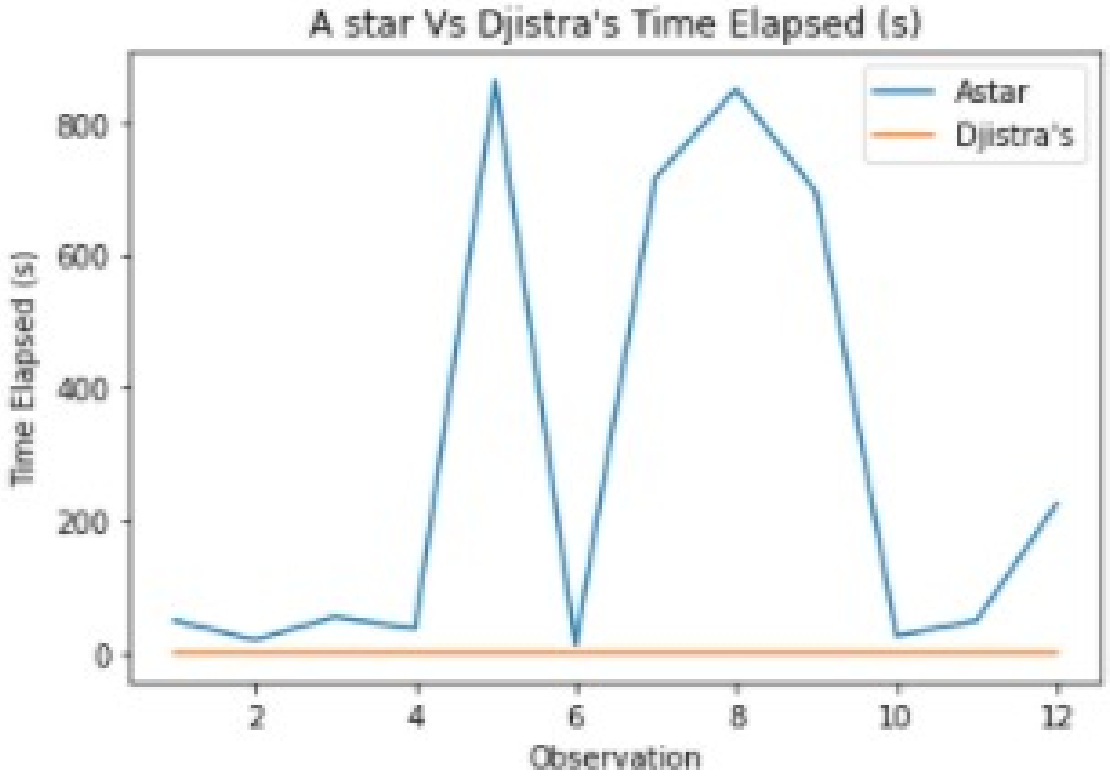
In order to compare the performance between the two algorithms, we ran twelve instances of each one between random selected source and destination nodes. As the graphs display indicate, A* was able to find the destination node in all iterations, with the degree of the improvement varying on a case by case basis. This makes sense, as the heuristic should be guiding the search in the right general direction a good percentage of time.

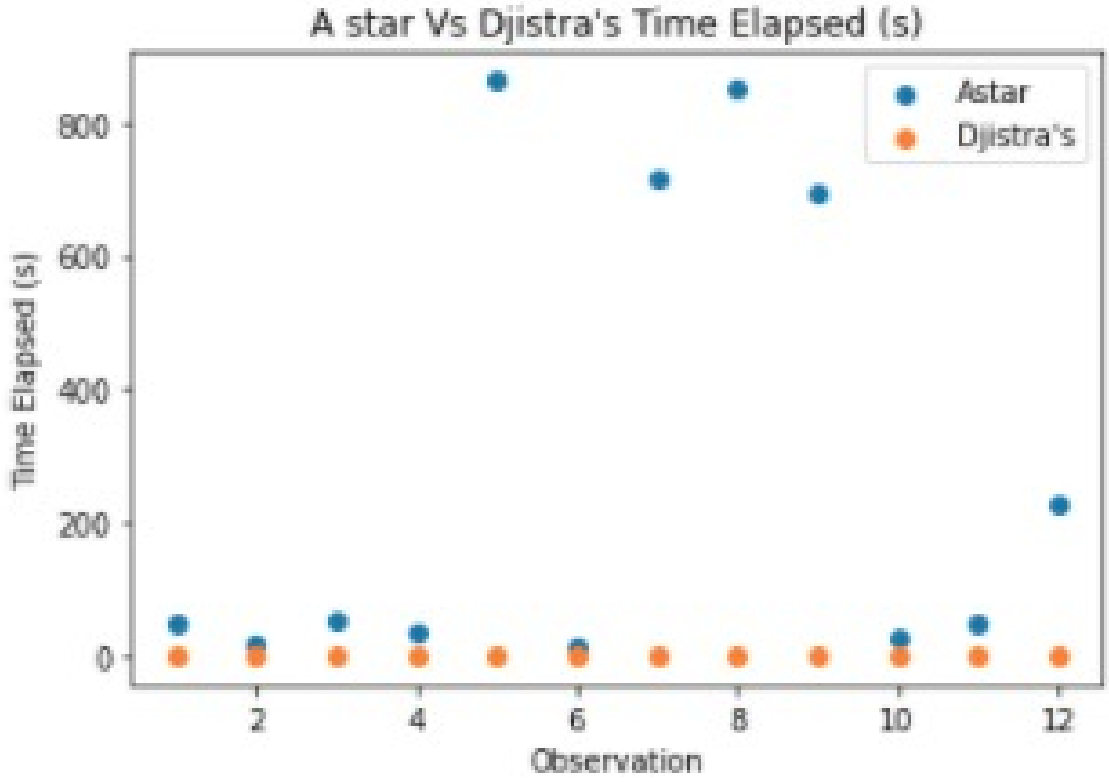




Time Elapsed

Using the same twelve instances as above, we also measured the time performance of the two algorithms. Here, Dijkstra's algorithm significantly outperformed A* in every iteration, particularly those where many nodes had to be evaluated. The reason for this is that the cost of the heuristic (running a BFS for each node being evaluated) outweighed the benefit of reducing the total number of nodes being searched.


A star Vs Djistra's Time Elapsed (s)
Astar
Djistra's
Time Elapsed (s)
0
200
400
600
800
2
4
6
8
10
12
Observation


A star Vs Djistra's Time Elapsed (s)
Astar
Djistra's
Time Elapsed (s)
0
200
400
600
800
2
4
6
8
10
12
Observation

Time and Space Complexity

Dijkstra's Time Complexity:

O(VE log(V))

V: Number of vertices
E: Maximum number of edges attached to a single node

Dijkstra's Space Complexity:

O(V)

V: Number of vertices

A Star Time Complexity:

$O(b^m)$:

b: Branching Factor
m: Height of the tree

A Star Space Complexity:

$O(b^m)$:

b: Branching Factor
m: Height of the tree

4 Conclusion

While A* can lead to a performance increase over Dijkstra's algorithm, it is important to consider whether to heuristic chosen is worth the benefit it provides. One way to improve the performance of this would be to calculate the BFS distance from each node to every other node beforehand, and store this in a lookup table. That would be memory intensive, however would have significantly improved the time performance.