Juliana Zweng (juzw3153)
Annelise Thorn (anth6800)

DSA Group Project 3 Report

Scenario One- Average Time

Description of Approach

The base scenario implements Dijkstra's Algorithm to find the fastest route from Twin Gates to the Golden Gate Bridge in San Francisco. We mapped 10 popular tourist locations in the city, and used Google Maps to find the average time to drive between them. This data was then translated into a dictionary that contains each node with connected nodes and the distances between them. To find the fastest routes, we created a Graph class that contains a shortest_distance function. The shortest_distance function returns a dictionary containing each node with the shortest distance from the starting point. In this case, we were looking for the shortest distance from node A (Twin Gates) to node J (Golden Gate Bridge). Once the data is passed into the Graph class and the shortest_distance function is called, the resulting dictionary of shortest distances can be indexed for any node. We indexed using the "J" key and printed the corresponding distance value.

Description of the Data Structure Used and the Algorithm Implementation

The starting data gathered from Google Maps is held in a dictionary. Within the Graph class, an initiation function was necessary to create an instance of the class and apply the shortest_distance attribute to our data. The shortest_distance attribute uses a variety of data structures including dictionaries, priority queues, min heaps and sets. First, a distances dictionary is created with keys as each node and values as the smallest distance from the starting point, or source. The initial values are set to zero for the source and infinity for all other distances. Then, we initialized a priority queue containing the source node using the heapify function in heapq. We also initialized an empty set to be filled with visited nodes.

Once all the data structures were set up, we implemented an iterative algorithm to check all distances from the source to each node, returning the minimum distance between them. First, the minimum value is popped from the priority queue and set as the current node and distance. Then, each neighbor node is checked. If the new path from the source to the neighboring node is shorter than the current distance, the value is replaced and pushed into the priority. This process repeats until every path has been checked and the shortest distances are returned in a dictionary.

Juliana Zweng (juzw3153)
Annelise Thorn (anth6800)

DSA Group Project 3 Report

Manual Calculation for the Base Case Scenario

Figure 1: Base Case Scenario Map with driving times between nodes

Using the driving times in Figure 1, a manual calculation can be performed to find the fastest route from A to J. An initial visual calculation indicates the fastest route is 20 minutes from A to D to I to J. The table below confirms this result by a systematic elimination of other routes:

Route	Time (minutes)	Analysis
A – D – I – J	20	Current shortest possible route

Check remaining directions from D:

A – D – E	22	> 20 mins (longer than current shortest possible route, eliminate direction
A – D – H	28	> 20 mins

Check paths towards J beginning with A to B:

A – B – E	32	> 20 mins
A – B – I	28	> 20 mins
A – B – C	21	> 20 mins, eliminate all paths beginning with A to B

Juliana Zweng (juzw3153)
Annelise Thorn (anth6800)

DSA Group Project 3 Report

Check paths towards J beginning with A to C:

A – C – B	21	> 20 mins
A – C – I	32	> 20 mins
A – C – H	26	> 20 mins
A – C – F	25	> 20 mins, eliminate all paths beginning with A to C

Table 1. Systematic manual calculations to prove shortest route

The manual calculations show that the shortest time is 20 minutes from A – D – I – J. All other routes are eliminated because once the time is greater than 20 minutes, there are no possible ways forward to reach J in a shorter time.

Simulation of Base Case Scenario with Results

The results of the manual calculations match the results of the coded simulation. See the attached Jupyter Notebook file to see the complete code. The following figures show the test code and the printed results:

```
#test code
distances = G.shortest_distances("A")
print(distances, "\n")

to_J = distances["J"]
print(f"The shortest distance from A to J is {to_J} mins")
```

```
{'A': 0, 'B': 16, 'C': 16, 'D': 10, 'E': 22, 'F': 25, 'G': 33, 'H': 26, 'I': 17, 'J': 20}

The shortest distance from A to J is 20 mins
```

Figure 2. Base Case Results

Results show a dictionary with the shortest time from node A to all other nodes. The dictionary can be indexed to find the shortest time from a specific node, in this case node J.

Discussion and Analysis of Results

As confirmed in both the manual and algorithmic simulation, the shortest time from A to J when there is no traffic is 20 minutes. Due to the nature of San Francisco, this was intuitive

considering this path took the straightest line. However, other maps that involve freeways may result in a less linear but more efficient route.

In this case, a manual/ visual calculation was a reasonable approach because there were only 10 nodes. Even so, Dijkstra's Algorithm was a more efficient and effective tool, and will become increasingly so as the data set grows. A further step could be to implement an algorithm that prints the route, in addition to the time. This would be helpful for users looking for directions or if they are choosing a route based on what they will see. In this case, the most efficient route hits 4 major San Francisco landmarks. An additional feature of the algorithm may suggest the next efficient route with different landmarks.

Scenario Two- High Traffic

Description of Approach

To represent a high traffic scenario, we expanded upon scenario one and added a function that randomly adds weight using a normal distribution with a mean of 2 of a standard deviation of 0.5. Additionally we added a level of uncertainty with a high impact and low probability event from occurring using a discrete distribution. Each time the algorithm runs, different traffic factors and high impact events are calculated which impacts the rush hour weight creating a modified graph.

Description of the Data Structure Used and the Algorithm Implementation

For scenario two, we used the same data structures as described in scenario one, but added a function, apply_rush_hour_traffic(), to take rush hour traffic into account. Apply_rush_hour_traffic() first creates an empty dictionary to store its modified graph, then it loops through each node and its neighbors in the original graph, next it loops each neighbor of the current node and adjusts the weight using a rush hour traffic factor. The rush hour traffic factor is generated randomly following a normal distribution with a mean of 2, a standard deviation of +- 0.5 and a minimum factor of 1 to ensure the rush hour factor never falls below 1. We then multiplied the weight by the traffic factor to get the rush hour weight. To account for a 20% chance of a high impact event, we applied the random.choices function with the choices set as 0 and 1 and the set weights as 0.8 and 0.2. If a high impact event does occur, a random weight between 0.5 and 2 will be multiplied by the original weight and will further increase the rush hour weight. The rush hour weight will then be rounded to the nearest whole number and the modified graph will be returned.

Juliana Zweng (juzw3153)
Annelise Thorn (anth6800)

DSA Group Project 3 Report

Simulation of Rush Hour Scenario with Results

When the rush hour scenario is run, one possible example of the traffic factors, high impact events, impact factors, and the final rush hour weight is as seen below:

```
Edge A -> B: Base Weight = 16   Traffic Factor = 2.12   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 33
Edge A -> C: Base Weight = 16   Traffic Factor = 2.71   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 43
Edge A -> D: Base Weight = 10   Traffic Factor = 1.55   High Impact Event = 1   Impact Factor = 1.48   Final Weight = 23
Edge B -> C: Base Weight = 5    Traffic Factor = 2.07   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 10
Edge B -> E: Base Weight = 16   Traffic Factor = 2.06   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 32
Edge B -> I: Base Weight = 17   Traffic Factor = 1.17   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 19
Edge C -> F: Base Weight = 9    Traffic Factor = 2.59   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 23
Edge C -> H: Base Weight = 10   Traffic Factor = 1.47   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 14
Edge C -> I: Base Weight = 16   Traffic Factor = 1.95   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 31
Edge D -> E: Base Weight = 12   Traffic Factor = 1.54   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 18
Edge D -> H: Base Weight = 18   Traffic Factor = 1.22   High Impact Event = 1   Impact Factor = 2.55   Final Weight = 56
Edge D -> I: Base Weight = 7    Traffic Factor = 2.55   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 17
Edge E -> I: Base Weight = 12   Traffic Factor = 1.37   High Impact Event = 1   Impact Factor = 2.25   Final Weight = 37
Edge F -> G: Base Weight = 8    Traffic Factor = 1.90   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 15
Edge F -> I: Base Weight = 22   Traffic Factor = 2.25   High Impact Event = 1   Impact Factor = 1.33   Final Weight = 66
Edge G -> H: Base Weight = 7    Traffic Factor = 1.38   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 9
Edge G -> I: Base Weight = 16   Traffic Factor = 1.62   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 25
Edge H -> I: Base Weight = 14   Traffic Factor = 2.16   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 30
Edge I -> J: Base Weight = 3    Traffic Factor = 1.67   High Impact Event = 0   Impact Factor = 0.00   Final Weight = 5
```

With these updated weights, we can update our map to look like:

A – Twin Peaks
B – Painted Ladies
C – City Hall
D – Golden Gate Park
E – Lands End
F – Oracle Park
G – Ferry Building
H – Fisherman's Wharf
I – Presidio
J – Golden Gate Bridge

Then to get the shortest travel time from point A (Twin Peaks) to point J (Golden Gate Bridge) we can run the code in figure 3 and see that the shortest travel time is 51 mins.

```
#test code
distances = G.shortest_distances("A")
print(distances, "\n")

to_J = distances["J"]
print(f"The shortest travel time from A to J is {to_J} mins")
```

```
{'A': 0, 'B': 21, 'C': 35, 'D': 25, 'E': 40, 'F': 66, 'G': 90, 'H': 55, 'I': 45, 'J': 51}

The shortest travel time from A to J is 51 mins
```

Juliana Zweng (juzw3153)
Annelise Thorn (anth6800)

DSA Group Project 3 Report

Discussion and Analysis of Results

The shortest path to get from point A (Twin Peaks) to point J (Golden Gate Bridge) follows A -> D -> I -> J. This is visually very easy to see, because it creates a straight line from point A to point J and takes the shortest route at each point, however, this is not always the case. In some cases, it will be more beneficial to take a route with a greater weight at one step or to take more steps than the most direct path to your point of interest. For this case, edges A -> C, C -> F, and D -> I endured high traffic factors (< 2.25) and edges A -> D (1.48), D -> H (2.55), E -> I (2.25), and F -> I (1.33) endured high impact events with their impact factors in parentheses. Even though edge A -> D endured a high impact event it was still included in the shortest route to point J, but this is because its overall weight ended up being less than A -> B and A -> C with traffic factors and impact factors considered. Additionally, routes off of node D were more simplistic than routes off of B and C. Overall, the rush hour traffic scenario did a good job of representing traffic in San Francisco; most paths were slightly impacted by rush hour traffic and some paths were significantly impacted by high impact events. It would be a real pain to have to get across the city from D -> H or I -> F, but rush hour does not significantly impact other paths like going from E -> D.