IEE 598: Network Optimization and Algorithms

Instructor: Dr. Pitu Mirchandani

Final Project Report

on

"Optimizing Path for Restaurant Delivery Bots"

by

Aekansh Mishra- 1217857342

April 24th, 2021

Table of Contents

1. Aim/Objectives of the project:

In this project, we consider a company that provides robots for food delivery from restaurants to Residence Halls at ASU. The company makes more profits or breaks even only when the robots are efficient at making the greatest number of deliveries in the least amount of time. The clients to this company would be restaurants around campus, like Domino's, Papa John's, Subway, Starbucks, etc. The Residence halls can be Hassayampa, Tooker House, Adelphi Commons, etc.

We will consider three levels of complexity:

Level 0: The robot is already carrying a single food order and has to deliver it to a location.
Level 1: The robot needs to both acquire the food from a restaurant and then deliver it to the required destination.
Level 2 (future scope): The robot can see a list of orders to be fulfilled and their delivery locations. It then needs to choose which orders to pick so that it can optimize for delivery in the shortest time possible. This was an ambitious goal for the current project but can definitely be executed later.

The project includes the implementation of the below features/concepts:
a) Representation of a graph
b) Use of Data Structures (Lists, Tuples, Dictionaries, Queue, Stack, Heap)
c) Dijkstra's Algorithm to calculate shortest path
d) A-star Search Algorithm to calculate shortest path using a heuristic
e) Time Complexity analysis

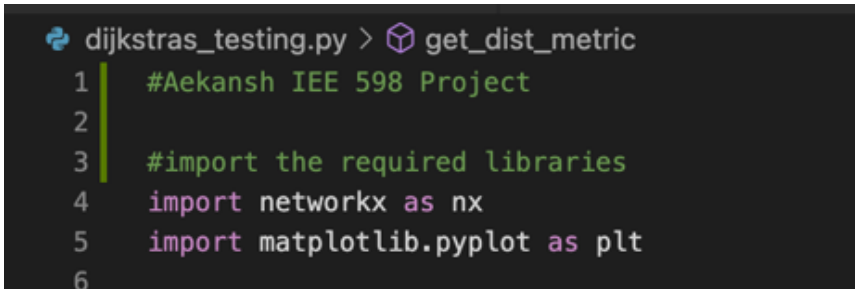
2. Introduction to NetworkX and foundation of the Project:

To optimize the path between the source (a restaurant) and the destination (a residence hall/student), we basically deal with a shortest path problem. We have used two algorithms, Dijkstra's Algorithm and A-Star Search Algorithm. Data Structures elements like Queue, PriorityQueue, and Heap have been used to implement it.

What is a Shortest Path Problem?

In Graph Theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of weights of its constituent edges is minimized.

When I started working on the project, I came across a dilemma of how to represent my Network graph using Python. This was bound to happen since I am not a student of computer science, and had a limited knowledge of Python, or programming, in general. So, constructing the graph with the help of 'Graph' class was an obvious choice. But after digging quite a bit deeper, I stumbled upon a powerful library on Python called NetworkX. NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It helps us visualize network graphs of several complexities, while giving us controls over how to represent nodes and edges, how to label them, and how to deal with the corresponding edge weights. Hence, we import the Networkx library at the start of the code. We also import matplotlib.pylot library in order to visualize the plots effectively.



Libraries imported

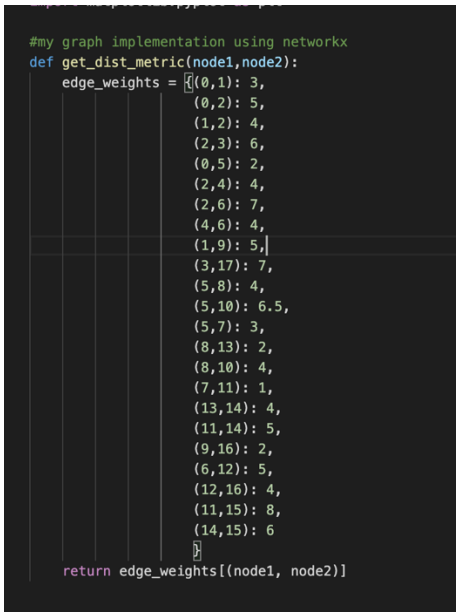
3. Dijkstra's Algorithm:

3.1 Graph Description:

We start programming the network graph by introducing a function *get_dist_metric()* which entails edge weights of the nodes connected to each other in the format *(node1, node2): weight*, where node1 refers to the first node, node2 refers to the second node, and weight refers to the edge weight connecting the two nodes. For instance, *(0,1): 3* means that the edge weight along the node 0 and node 1 (0 and 1 are the indices of the nodes) is 3.

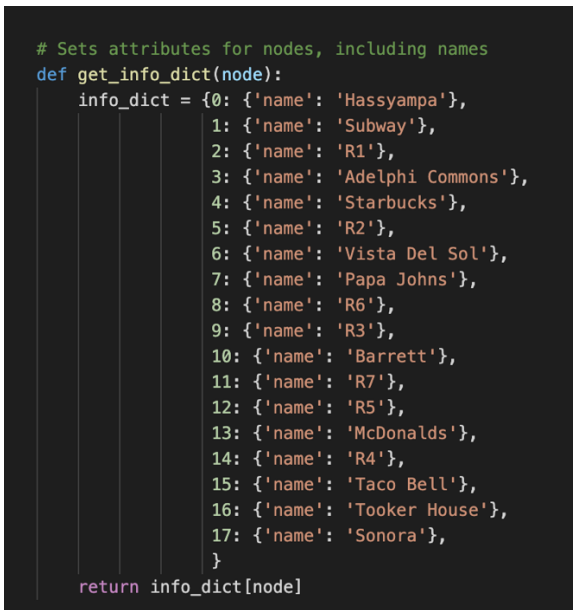
It could have been inserted without using the function, but it was done so, so that in a dynamic situation where we have to introduce other parameters or the edge weights were made to reflect other traffic conditions like wait time, we could do that easily using the function. It future proofs our program. We can dynamically call edge weights through a database possibly, or even some other random function, if the situation demands that. Later, we introduce another function *get_info_dict()* to add a dictionary where we add the names of all the nodes with respect to their indices. For example, Node 0 is Hassayampa Residence hall, while Node 1 represents Subway restaurant. There are 7 Random Points in the graph, from R1, R2,……. R7. Then we create the graph with help of instructions from the Networkx library, and add the nodes from the list *e*, with metric (weights) and labels from the functions we used above.

Here are the screenshots from VS Code Window showing the specific parts of the code:



```python
#my graph implementation using networkx
def get_dist_metric(node1,node2):
    edge_weights = {(0,1): 3,
                    (0,2): 5,
                    (1,2): 4,
                    (2,3): 6,
                    (0,5): 2,
                    (2,4): 4,
                    (2,6): 7,
                    (4,6): 4,
                    (1,9): 5,
                    (3,17): 7,
                    (5,8): 4,
                    (5,10): 6.5,
                    (5,7): 3,
                    (8,13): 2,
                    (8,10): 4,
                    (7,11): 1,
                    (13,14): 4,
                    (11,14): 5,
                    (9,16): 2,
                    (6,12): 5,
                    (12,16): 4,
                    (11,15): 8,
                    (14,15): 6
                    }
    return edge_weights[(node1, node2)]
```

giving edge weights



```python
# Sets attributes for nodes, including names
def get_info_dict(node):
    info_dict = {0: {'name': 'Hassyampa'},
                 1: {'name': 'Subway'},
                 2: {'name': 'R1'},
                 3: {'name': 'Adelphi Commons'},
                 4: {'name': 'Starbucks'},
                 5: {'name': 'R2'},
                 6: {'name': 'Vista Del Sol'},
                 7: {'name': 'Papa Johns'},
                 8: {'name': 'R6'},
                 9: {'name': 'R3'},
                 10: {'name': 'Barrett'},
                 11: {'name': 'R7'},
                 12: {'name': 'R5'},
                 13: {'name': 'McDonalds'},
                 14: {'name': 'R4'},
                 15: {'name': 'Taco Bell'},
                 16: {'name': 'Tooker House'},
                 17: {'name': 'Sonora'},
                 }
    return info_dict[node]
```

giving node labels

```python
# Code from networkx that creates the same graph
G = nx.Graph()
G.add_nodes_from([(0, get_info_dict(0)),
                  (1, get_info_dict(1)),
                  (2, get_info_dict(2)),
                  (3, get_info_dict(3)),
                  (4, get_info_dict(4)),
                  (5, get_info_dict(5)),
                  (6, get_info_dict(6)),
                  (7, get_info_dict(7)),
                  (8, get_info_dict(8)),
                  (9, get_info_dict(9)),
                  (10, get_info_dict(10)),
                  (11, get_info_dict(11)),
                  (12, get_info_dict(12)),
                  (13, get_info_dict(13)),
                  (14, get_info_dict(14)),
                  (15, get_info_dict(15)),
                  (16, get_info_dict(16)),
                  (17, get_info_dict(17)),
                  ])

e = [(0, 1, get_dist_metric(0,1)),
     (0, 2, get_dist_metric(0,2)),
     (0, 5, get_dist_metric(0,5)),
     (2, 3, get_dist_metric(2,3)),
     (2, 4, get_dist_metric(2,4)),
     (2,6, get_dist_metric(2,6)),
     (1,2, get_dist_metric(1,2)),
     (4,6,get_dist_metric(4,6)),
     (1,9,get_dist_metric(1,9)),
     (3,17,get_dist_metric(3,17)),
     (5,8,get_dist_metric(5,8)),
     (5,10,get_dist_metric(5,10)),
     (5,7,get_dist_metric(5,7)),
```

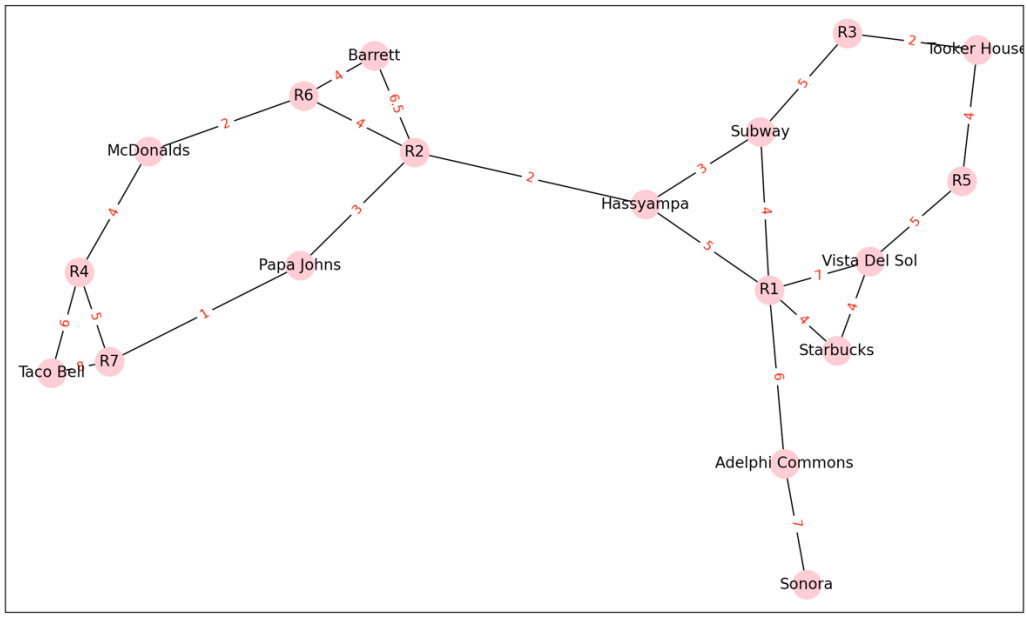
adding nodes from edge list

3.2 Drawing the Graph:

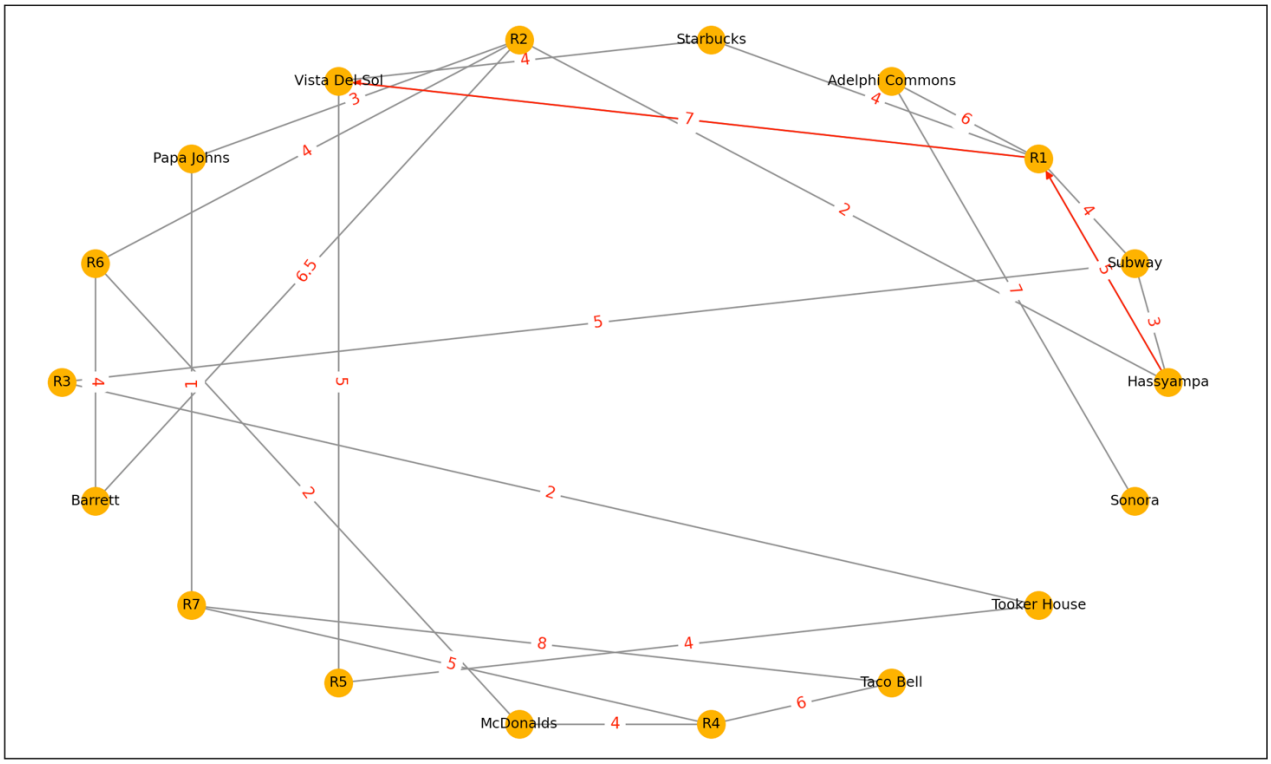
We choose the circular layout for Graph G. Then we proceed with assigning attributes like size and color for the nodes and edges using *nx.draw_networkx_nodes* and *nx.draw_networkx_edges* command. Then we give the attributes to the labels too. We use the function *nx.draw_networkx_edges* again to give a specific color to the shortest/optimal path. *Plt.show* is used to show the plot.

```
175
176  pos = nx.circular_layout(G)
177  # nx.draw_networkx_edge_labels(G,pos)
178  nx.draw_networkx_nodes(G, pos, node_size=300, node_color='orange')
179  nx.draw_networkx_edges(G, pos, edgelist=e, arrows=False, edge_color= 'grey')
180  nx.draw_networkx_edges(G, pos, edgelist=[(dist[node_y][1][idx], dist[node_y][1][idx+1], get_dist_metric(idx, idx+1))
181                                           for idx in range(len(dist[node_y][1]) - 1)], edge_color='red', arrows=True)
182  nx.draw_networkx_edge_labels(G, pos, edge_labels={(x,y): weight for x,y,weight in G.edges.data("weight")}, font_color='red')
183  nx.draw_networkx_labels(G, pos, labels=dict(G.nodes(data='name')), font_size=9)
184  plt.show()
185
```

Python Output (The Plot with 17 nodes):



Spiral Layout



Circular Layout

3.3 Implementation of Dijkstra's Algorithm:

We have 18 nodes in total, indexed from 0 to 17. It includes 7 Random Points in the graph along with 5 restaurants and 6 dorms.

According to Wikipedia, Dijkstra's Algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, rail networks. It was conceived by computer scientist Edsgar W. Dijkstra in 1956.

I have taken reference and inspiration from a Medium article from Micah Shute on Dijkstra's Shortest Path Problem, and successfully implemented networkx library features in it. We start by defining a function *Dijkstra_my()* having G, starting node, and end node as the variables. Distance is list of lists containing smallest from source node to each node in the network, which is defined as *dist = [None] * len(G.nodes()),* where it carries an initial value of infinity. We declare the distance from source node to itself as 0. Then we use the queue data structure to keep

track of the nodes. We set the numbers seen so far by using *set()* function. Then we iterate over queue to find the minimum distance using the while loop. Here, if the goal node is specified, then stop when minimum distance to goal node has been found using *break.*
Otherwise, we determine the node with the smallest distance from the current node. We then pop the node with minimum distance from queue and add it to seen *set().* We get the nodes connected to the minimum node by making a list of tuples called *connections* containing the component from the neighbor nodes. We then update the distance and path.

Enter the source node and the destination node value as *node_x* and *node_y.* Print the distance from source to every other node, and also print the distance from source to the destination node.

3.4 The Code for Dijkstra's Implementation:

```
# This is my Dijkstra code
"""
G: networkx graph
node_x: source node
node_y: destination node
"""
def dijkstra_my(G, node_x, node_y=None):
    # distance is list of lists containing smallest from source node
    # to each node in the network
    dist = [None] * len(G.nodes())
    for i in range(len(G.nodes())):
        dist[i] = [float("inf")]
        dist[i].append([node_x])
    # distance from source node to itself
    dist[node_x][0] = 0
    # Queue data structure to keep track of nodes
    queue = [i for i in range(len(G.nodes()))]
    # Set of numbers seen so far
    seen = set()
    # iterate over queue to find the minimum distance
    while len(queue) > 0:
        # if goal node is specified then stop
        # when min distance to goal node has been found
        if node_y in seen:
            break

        # determine node with the smallest distance from the current node
        min_dist = float("inf")
        min_node = None
        for n in queue:
            if dist[n][0] < min_dist and n not in seen:
                min_dist = dist[n][0]
                min_node = n

        # pop the node with min distance from queue
        # and add it to seen set()
        queue.remove(min_node)
        seen.add(min_node)
```

```
        # get nodes connected to min node
        connections = [(n, G[min_node][n]["weight"]) for n in G.neighbors(min_node)]
        # update the distance and the path
        for (node, weight) in connections:
            tot_dist = weight + min_dist
            if tot_dist < dist[node][0]:
                dist[node][0] = tot_dist
                dist[node][1] = list(dist[min_node][1])
                dist[node][1].append(node)
    return dist

node_x = 0  # source node
node_y = 6  # destination node
dist = dijkstra_my(G, node_x=node_x)

# print the distance from source to every other node
print('The distance returned by Dijkstras is:', dist)

dist = dijkstra_my(G, node_x=node_x, node_y=node_y)
# print the distance from source to destination node
print('The distance returned by Dijkstras is:', dist[node_y])
```

3.5 Results of Dijkstra's Implementation:

Scenario 1:

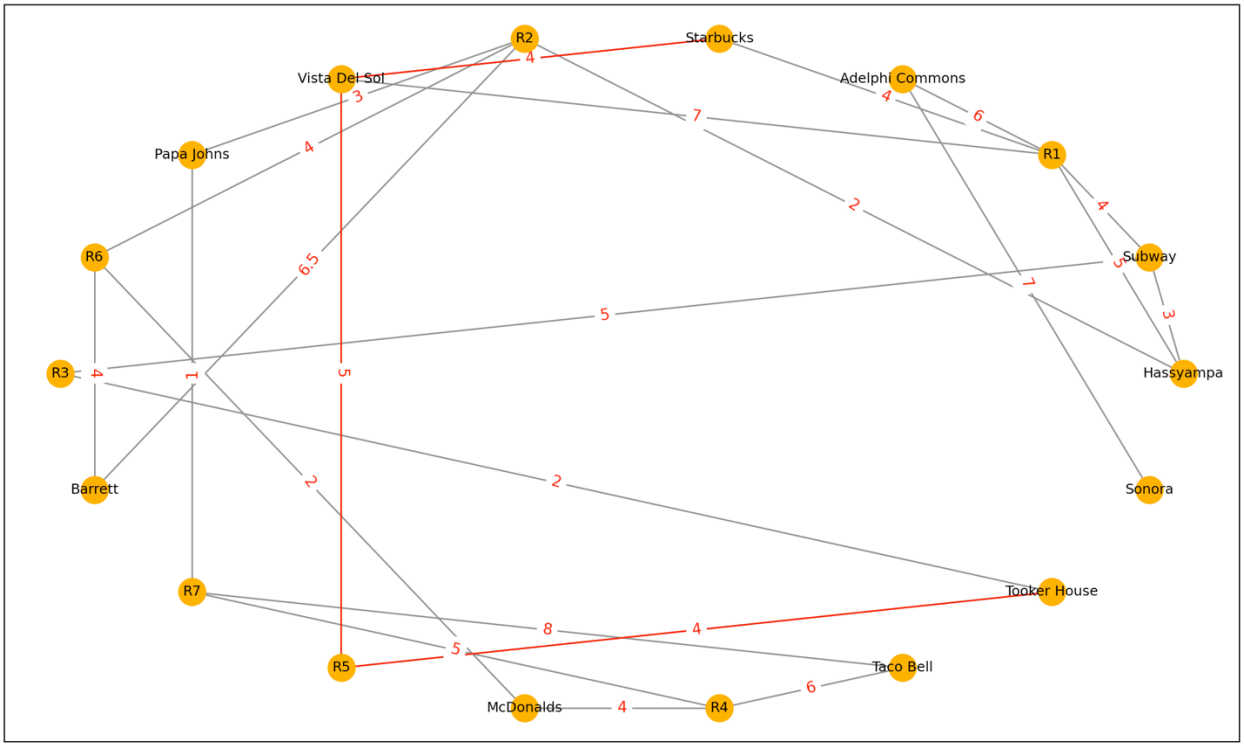
Level 0 Implementation: Food is delivered from Starbucks (4) to Tooker House (16)

```
#input source and destination nodes here

node_x = 4  # source node
node_y = 16  # destination node
dist = dijkstra_my(G, node_x=node_x)

# print the distance from source to every other node
print('The distance returned by Dijkstras is:', dist)

dist = dijkstra_my(G, node_x=node_x, node_y=node_y)
# print the distance from source to destination node
print('The distance returned by Dijkstras is:', dist[node_y])
```

```
aekansh@Aekanshs-MacBook-Air shortest-path-algorithms % python3 dijkstras_testing.py
The distance returned by Dijkstras is: [[9, [4, 2, 0]], [8, [4, 2, 1]], [4, [4, 2]],
 [10, [4, 2, 3]], [0, [4]], [11, [4, 2, 0, 5]], [4, [4, 6]], [14, [4, 2, 0, 5, 7]],
[15, [4, 2, 0, 5, 8]], [13, [4, 2, 1, 9]], [17.5, [4, 2, 0, 5, 10]], [15, [4, 2, 0,
5, 7, 11]], [9, [4, 6, 12]], [17, [4, 2, 0, 5, 8, 13]], [20, [4, 2, 0, 5, 7, 11, 14]
], [23, [4, 2, 0, 5, 7, 11, 15]], [13, [4, 6, 12, 16]], [17, [4, 2, 3, 17]]]
The distance returned by Dijkstras is: [13, [4, 6, 12, 16]]
aekansh@Aekanshs-MacBook-Air shortest-path-algorithms %
```

As we can see, the shortest distance given by terminal is 13. The path is 4-6-12-16. Or Starbucks--Vista Del Sol--R5--Tooker House.

Graphical Representation (shortest path given with red edges):



Scenario 2:

<u>Level 1 Implementation:</u> Food is picked from Hassayampa (4) from a point R7 (11) and then delivered to Vista Del Sol (6)

```
#input source and destination nodes here

node_x = 11  # source node
node_y = 0  # destination node
dist = dijkstra_my(G, node_x=node_x)

# print the distance from source to every other node
print('The distance returned by Dijkstras is:', dist)

dist = dijkstra_my(G, node_x=node_x, node_y=node_y)
# print the distance from source to destination node
print('The distance returned by Dijkstras is:', dist[node_y])
```

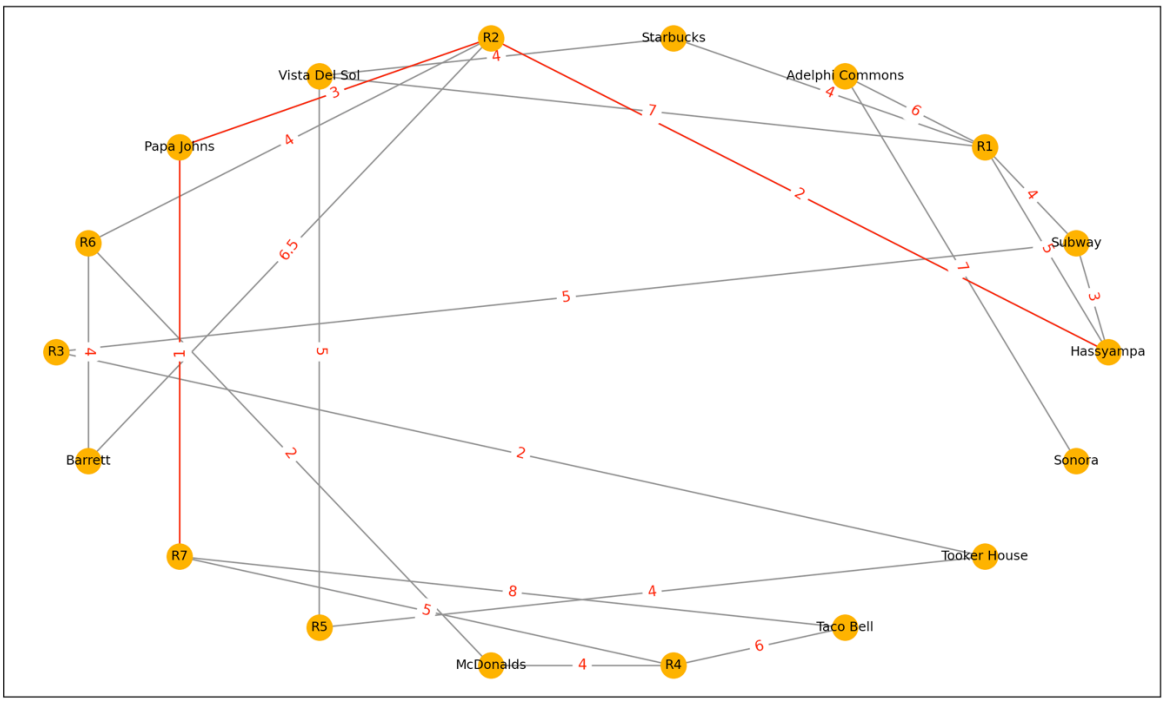
```
#input source and destination nodes here

node_x = 0  # source node
node_y = 6  # destination node
dist = dijkstra_my(G, node_x=node_x)

# print the distance from source to every other node
print('The distance returned by Dijkstras is:', dist)

dist = dijkstra_my(G, node_x=node_x, node_y=node_y)
# print the distance from source to destination node
print('The distance returned by Dijkstras is:', dist[node_y])
```

```
aekansh@Aekanshs-MacBook-Air shortest-path-algorithms % python3 dijkstras_testing.py

The distance returned by Dijkstras is: [[6, [11, 7, 5, 0]], [9, [11, 7, 5, 0, 1]], [
11, [11, 7, 5, 0, 2]], [17, [11, 7, 5, 0, 2, 3]], [15, [11, 7, 5, 0, 2, 4]], [4, [11
, 7, 5]], [18, [11, 7, 5, 0, 2, 6]], [1, [11, 7]], [8, [11, 7, 5, 8]], [14, [11, 7,
5, 0, 1, 9]], [10.5, [11, 7, 5, 10]], [0, [11]], [20, [11, 7, 5, 0, 1, 9, 16, 12]],
[9, [11, 14, 13]], [5, [11, 14]], [8, [11, 15]], [16, [11, 7, 5, 0, 1, 9, 16]], [24,
 [11, 7, 5, 0, 2, 3, 17]]]
The distance returned by Dijkstras is: [6, [11, 7, 5, 0]]
```

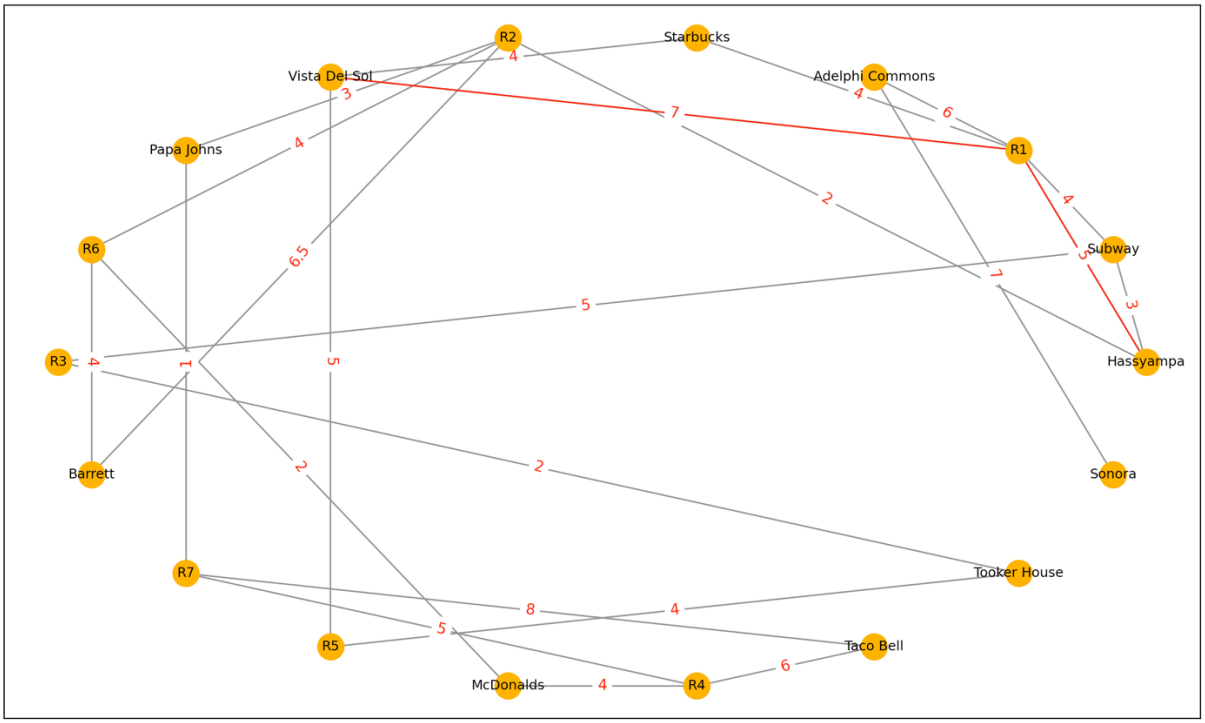
```
aekansh@Aekanshs-MacBook-Air shortest-path-algorithms % python3 dijkstras_testing.py

The distance returned by Dijkstras is: [[0, [0]], [3, [0, 1]], [5, [0, 2]], [11, [0,
 2, 3]], [9, [0, 2, 4]], [2, [0, 5]], [12, [0, 2, 6]], [5, [0, 5, 7]], [6, [0, 5, 8]
], [8, [0, 1, 9]], [8.5, [0, 5, 10]], [6, [0, 5, 7, 11]], [14, [0, 1, 9, 16, 12]], [
8, [0, 5, 8, 13]], [11, [0, 5, 7, 11, 14]], [14, [0, 5, 7, 11, 15]], [10, [0, 1, 9,
16]], [18, [0, 2, 3, 17]]]
The distance returned by Dijkstras is: [12, [0, 2, 6]]
aekansh@Aekanshs-MacBook-Air shortest-path-algorithms %
```

As we can see, the shortest distance given by terminal is 12 +6 = 18. The path is 11-7-5-0-2-6. Or R7—Papa John's—R2—Hassayampa—R1—Vista Del Sol.

Graphical Representation (shortest path given with red edges):



Shortest path while picking up the order



Shortest path while delivering the order

4. A* or A-Star Search Algorithm:

According to Wikipedia, A* or A-Star is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency.

As the A-Star Search Method involves working with a heuristic, and whatever the heuristic is, it will have a $O(b^d)$ space complexity, as it stores all generated nodes in memory. Hence, our previous 18-node graph would have made it a tedious execution at this level, and difficult to present on paper. Hence, we use a smaller, less complex graph for this algorithm. Also, where Dijkstra's was greedily approaching the goal, A-Star Search will use priority queue data structure technique to reach the goal more efficiently.

4.1 New Graph with the Code:

It must be noted that code for the graph is similar to the Dijkstra's one. Just another *get_heuristic()* function is added here which contains all the heuristic values between nodes in it, i.e., the straight line distances between them (unlike the walking distance stored in the *edge_weights* list). We have used Google Maps for this purpose. The values were scaled and dimensionally equalized for easy use.

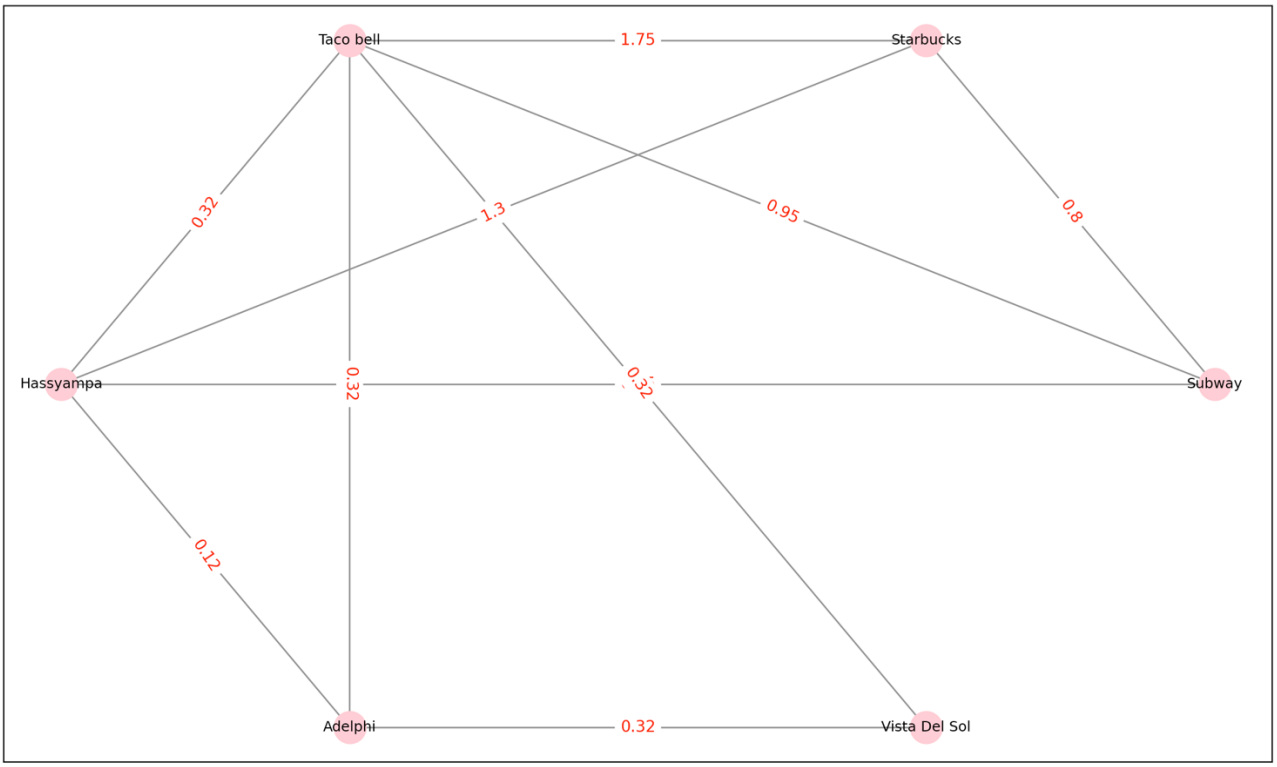
astar.py > a_star_search

```python
import networkx as nx
import matplotlib.pyplot as plt
from queue import PriorityQueue

#my graph for A Star Search using networkx
def get_dist_metric(node1,node2):
    edge_weights = {(0,1): 0.8,
                    (0,2): 0.95,
                    (0,3): 0.65,
                    (1,2): 1.75,
                    (1,3): 1.30,
                    (2,3): 0.32,
                    (2,4): 0.32,
                    (2,5): 0.32,
                    (3,4): 0.12,
                    (4,5): 0.32,
                    }
    return edge_weights[(node1, node2)]


# Sets attributes for nodes, including names
def get_info_dict(node):
    info_dict = {0: {'name': 'Subway'},
                 1: {'name': 'Starbucks'},
                 2: {'name': 'Taco bell'},
                 3: {'name': 'Hassyampa'},
                 4: {'name': 'Adelphi'},
                 5: {'name': 'Vista Del Sol'},
                 }
    return info_dict[node]

```

```python
#code for heuristic attribute if required
def get_heuristic(node, node_y):
    if node == node_y:
        return 0
    values = {(0,1): 0.525,
                (0,2): 0.77,
                (0,3): 0.48,
                (0,4): 0.675,
                (0,5): 0.56,
                (1,2): 1.29,
                (1,3): 1.02,
                (1,4): 1.21,
                (1,5): 1.13,
                (2,3): 0.335,
                (2,4): 0.235,
                (2,5): 0.368,
                (3,4): 0.16,
                (3,5): 0.238,
                (4,5): 0.2,
                }
    heuristic = values.get((node, node_y))
    if not heuristic:
        heuristic = values.get((node_y, node))
    return heuristic

# Code from networkx that creates the same graph
G = nx.Graph()
G.add_nodes_from([(0, get_info_dict(0)),
                  (1, get_info_dict(1)),
                  (2, get_info_dict(2)),
                  (3, get_info_dict(3)),
                  (4, get_info_dict(4)),
                  (5, get_info_dict(5)),
                  ])

e = [(0, 1, get_dist_metric(0,1)),
     (0, 2, get_dist_metric(0,2)),
     (0, 3, get_dist_metric(0,3)),
     (1, 2, get_dist_metric(1,2)),
     (1, 3, get_dist_metric(1,3)),
     (2,3, get_dist_metric(2,3)),
     (2,4, get_dist_metric(2,4)),
     (2,5,get_dist_metric(2,5)),
     (3,4,get_dist_metric(3,4)),
     (4,5,get_dist_metric(4,5))]
```

To Draw the Graph:

```python
pos = nx.circular_layout(G)
# nx.draw_networkx_edge_labels(G,pos)
nx.draw_networkx_nodes(G, pos, node_size=400, node_color='pink')
nx.draw_networkx_edges(G, pos, edgelist=e, edge_color='grey')
nx.draw_networkx_edge_labels(G, pos, edge_labels={(x,y): weight for x,y,weight in G.edges.data("weight")}, font_color='red')
nx.draw_networkx_labels(G, pos, labels=dict(G.nodes(data='name')), font_size=9)
plt.show()
```



Circular Layout Graph for A-Star

4.2 Implementation of A-Star Search Algorithm:

The program is similar to the one we used for the Dijkstra's Algorithm. The difference is with the kind of data structure used. Here we use the PriorityQueue instead of Queue, which was defined as frontier. Also, while updating the path and the distance, we have a component of heuristic added to it. The code looks as follows:

```python
# A* search algo
def a_star_search(G, node_x, node_y):
 # distance is list of lists containing smallest from source node
 # to each node in the network
    dist = [None] * len(G.nodes())
    for i in range(len(G.nodes())):
        dist[i] = [float("inf")]
        dist[i].append([node_x])
    # distance from source node to itself
    dist[node_x][0] = 0

# PriorityQueue data structure to keep track of nodes
    frontier = PriorityQueue()
    frontier.put(node_x, 0)
    # Set of numbers seen so far
    seen = set()

    # iterate over queue to find the minimum distance
    while not frontier.empty():
        # if goal node is specified then stop
        # when min distance to goal node has been found
        if node_y in seen:
            break
        # determine node with the smallest distance from the current node
        min_node = frontier.get()
        min_dist = dist[min_node][0]
        print(f"Min node: {min_node}")
        # add min node to the seen set
        seen.add(min_node)
        # Get nodes connected to min node
        connections = [(n, G[min_node][n]["weight"]) for n in G.neighbors(min_node)]
        # update the distance and the path using the heuristic
        for (node, weight) in connections:
            tot_dist = weight + min_dist
            if tot_dist < dist[node][0] and node not in seen:
                dist[node][0] = tot_dist
                dist[node][1] = list(dist[min_node][1])
                dist[node][1].append(node)
                frontier.put(node, tot_dist + get_heuristic(node, node_y))
    return dist
```

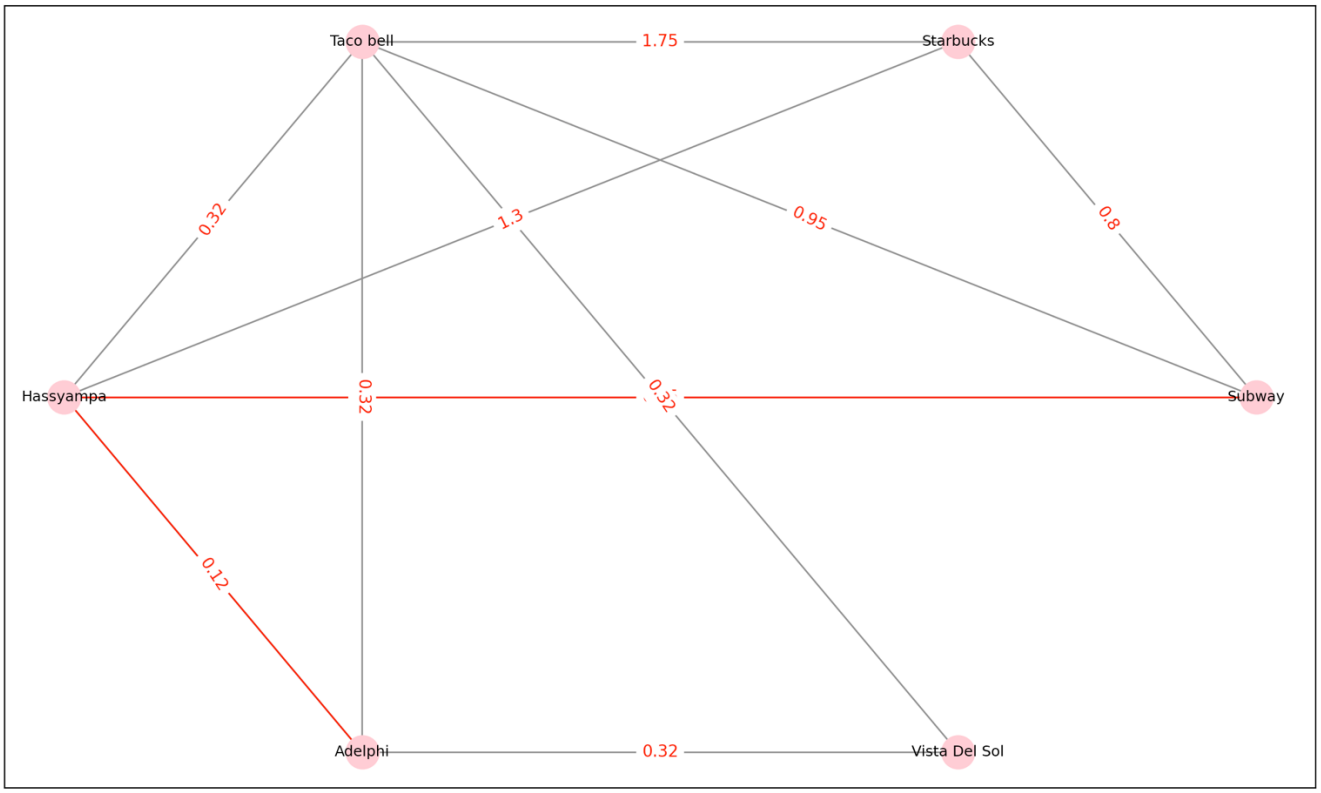
4.3 Results of A-Star Search Implementation:

Scenario: Food is delivered from Subway (0) to Adelphi (4)

```python
#input source and destination nodes here
node_x = 0 #source node
node_y = 4 #destination node

dist = a_star_search(G, node_x=node_x, node_y=node_y)
# print the distance from source to destination node
print('The distance returned by A* is:', dist[node_y])
```

```
[aekansh@Aekanshs-MacBook-Air shortest-path-algorithms % python3 astar.py
Min node: 0
Min node: 1
Min node: 2
Min node: 3
Min node: 4
The distance returned by A* is: [0.77, [0, 3, 4]]
aekansh@Aekanshs-MacBook-Air shortest-path-algorithms %
```

Hence, the shortest distance is 0.77, and the shortest path is 0-3-4, i.e., Subway—Hassayampa-Adelphi. The shortest path is also highlighted in red color in the graph:



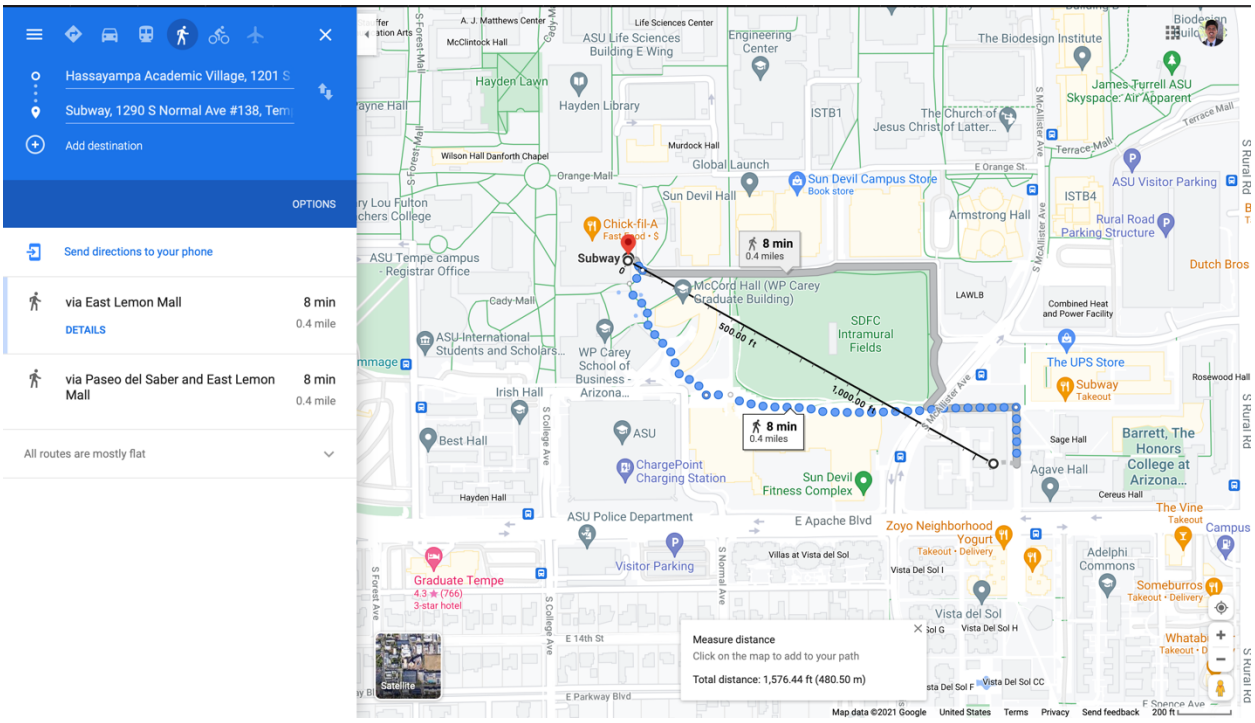
4.4 Choosing the heuristic:

We decide to choose the Straight-Line Distance as the metric to come up with a heuristic.

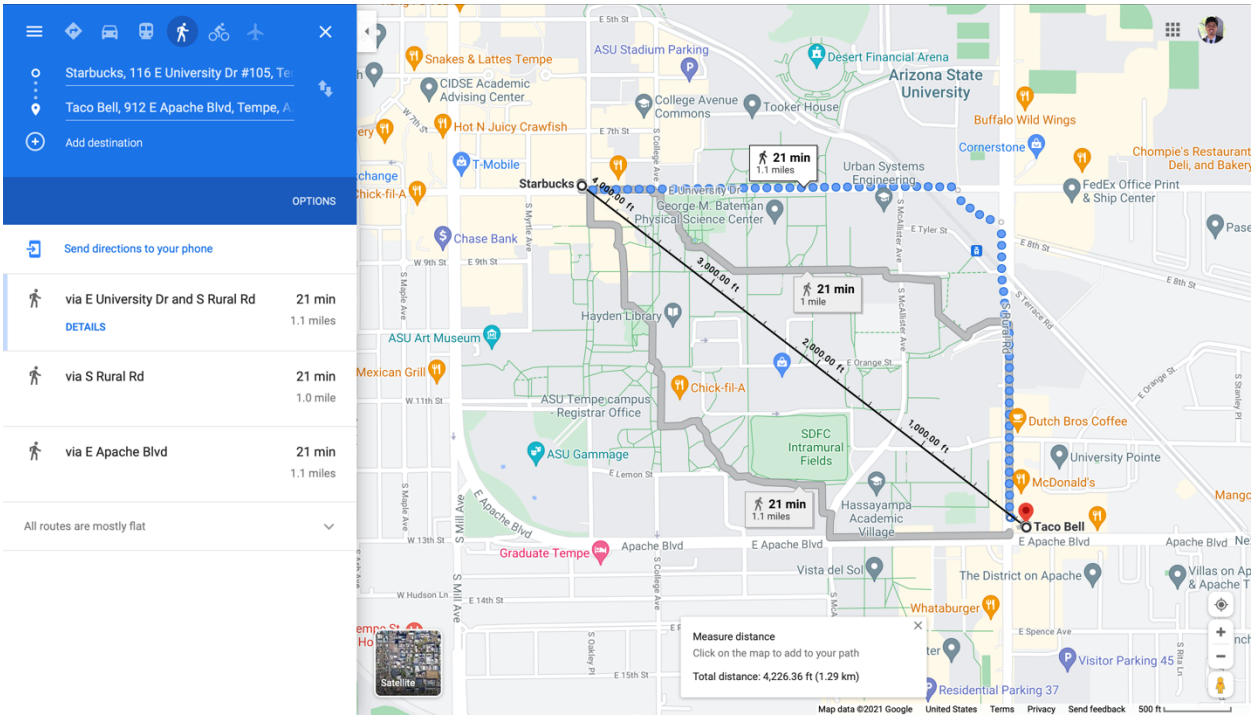
We take the walking distance from Google Maps between the restaurants and the dorms to create our network graph. In order to decrease the complexity of the network, and make sure it is presentable in the report, we decrease the number of nodes to 6, (3 restaurants and 3 dorms).

Below are some screenshots of how the data was collected using google maps.

1. Hassayampa Village to Subway:



2. Starbucks to Taco Bell:



4.5 Conditions for optimality: Admissibility and consistency

From the famous book on Artificial Intelligence by Peter Norvig and J.Russell, we learn that:

1. The first condition we require for optimality is that *h(n)* be an admissible heuristic. An admissible heuristic is one that *never overestimates* the cost to reach the goal. Since the straight-line distance will always be shorter than the walking distance via streets between two nodes, our heuristic can **never** overestimate. We visualized that on the google maps.
2. The second one is consistency. It is the traditional triangle inequality.

   i.e., $h(n) <= c(n,n') + h(n')$

   Since our heuristic is a straight-line distance, it will always obey this property.

5. Priority Queue (Heap) Implementation in Dijkstra's:

We observer that if we put the heuristic value = 0 in A-star search code, it basically becomes Dijkstra's algorithm. But this way, we implement Dijkstra's using priority queue (binary heap) data structure. The results are the same, but we use priority queue to find the min node. It makes our system a bit more efficient. Program Code can be found in the GitHub Repository.

6. Time Complexity Analysis:

Time Complexity of Dijkstra's Algorithm is $O(V^2)$ but with min-priority queue it drops down to $O(V + E*logV)$.
Time Complexity of A-star search Algorithm is $O(b^d)$ or $O(V)$.

7. Conclusion:

In conclusion, we can say that both the algorithms have their pros and cons. They both do the job well, in specific scenarios. Where Dijkstra's uses greedy approach, and gives good results, it sometimes isn't efficient. It can neither deal with negative edge weights.

On the contrary, A-star search is more efficient, as it uses the heuristic approach, but the drawback is that it takes more calculations to get to the goal (calculating heuristic value of every node from every node).

8. Future Scope:

We can successfully implement the level 2 as we planned during the initial proposal of the project. Where, ***Level 2 (future scope):*** *The robot can see a list of orders to be fulfilled and their delivery locations. It then needs to choose which orders to pick so that it can optimize for delivery in the shortest time possible. This is an ambitious goal for the current project but can definitely be executed later.*

If time wouldn't have been a binding factor, it could have been presented with the main project itself. For that purpose, we used functions to call edge weights, so we could anytime add other factors like waiting time at restaurant and number of orders into it or pick data from a given database. But in the future, it can definitely be implemented with the given code skeleton itself, without much hassle.

9. Bibliography:

1. Networkx.org
2. https://www.cantorsparadise.com/dijkstras-shortest-path-algorithm-in-python-d955744c7064
3. https://learnxinyminutes.com/docs/python/
4. https://www.redblobgames.com/pathfinding/a-star/implementation.html
5. Russell, Stuart, and Peter Norvig. "Artificial intelligence: a modern approach." (2002)
6. My Github Repository (https://github.com/aekansh7/shortest-path-algorithms)
7. Wikipedia