Computational Thinking: Final Report

Lucas HERAL
SN: 17072444
Word count: 1997 words

1) System requirements and business problem

a) Business problem

The food delivery market is booming all over the world. Moreover, Uber Eat is the fastest growing meal delivery service in the US[1]. Looking at the issues in the food delivery service industry, it is easy to see that ensuring rapid delivery is a key challenge[2]. Moreover, companies like Uber Eat want to be efficient and deliver as much as possible. Therefore, the delivery guys often have to pick up two orders consecutively. Therefore, an objective for our algorithm is to find the best 2nd order to pick up which would make the least waste of time when delivering the first customer. We also considered that if the detour for picking up the 2nd order takes more than five minutes, it is not worth it as it lowers the quality of our customer experience. The algorithm should tell us if the delivery guy should pick up a 2nd order and if yes, the position of the 2nd order.

b) Why Dijkstra?

Dijkstra's algorithm is an algorithm which finds the shortest route between two points on a weighted graph. This algorithm tries every possible path, replacing paths when a shorter one is found. Therefore, we can see we could use it for Uber Eats' problem. In fact, Uber Eat needs to know the which second order could add the least time to the 1st order. Using time as weight for the edges, we can definitely see how this algorithm could help solve our business problem.

I will try to use Dijkstra's algorithm to Uber Eat's problem by writing a well-structured code that will give an accurate answer. I will build my code on econchick's algorithm found on GitHubGist.[3]

---

[1] 'Uber Eats and the $6 billion bookings run rate: The AI success story no one is talking about' RB, Oct 2018.

[2] '8 Issues To Consider When Setting Up Your Food Delivery Service', Lior Sion, Oct 2017

[3] 'Python implementation of Dijkstra's Algorithm' by econchick on GitHubGist.

c) Functional requirements

To run the Dijkstra algorithm you need a graph and an initial node as the inputs. In the case Uber Eat, the graph represents a map of the area of the orders where the edges are the streets, the nodes represent the position of the delivery guy, the intersections of the streets but also the restaurants for the next orders and the position of the first client. Moreover, the edges are oriented and weighted with the streets' durations (only positive values). What the Dijkstra function does in the code is that it computes all the shortest times to go to each node of the graph given a departure node(output). When running the function, it thus returns the time of the shortest path to go to each node from the departure node path and the predecessors, which are the nodes we need to go through to minimize the time.
The second part of the algorithm calls the first function to calculate the time of fastest route between the departure and the arrival but also when making a detour to pick up the next order. Therefore, the inputs are the departure (rider's position), the arrival (first client's position), the positions of the restaurants' next orders and the same graph as in the previous function. The code will output the decision to make concerning the second order. In fact, it will say whether it is worth or not to pick it up on the way to the 1st customer.

d) Non-functional requirements

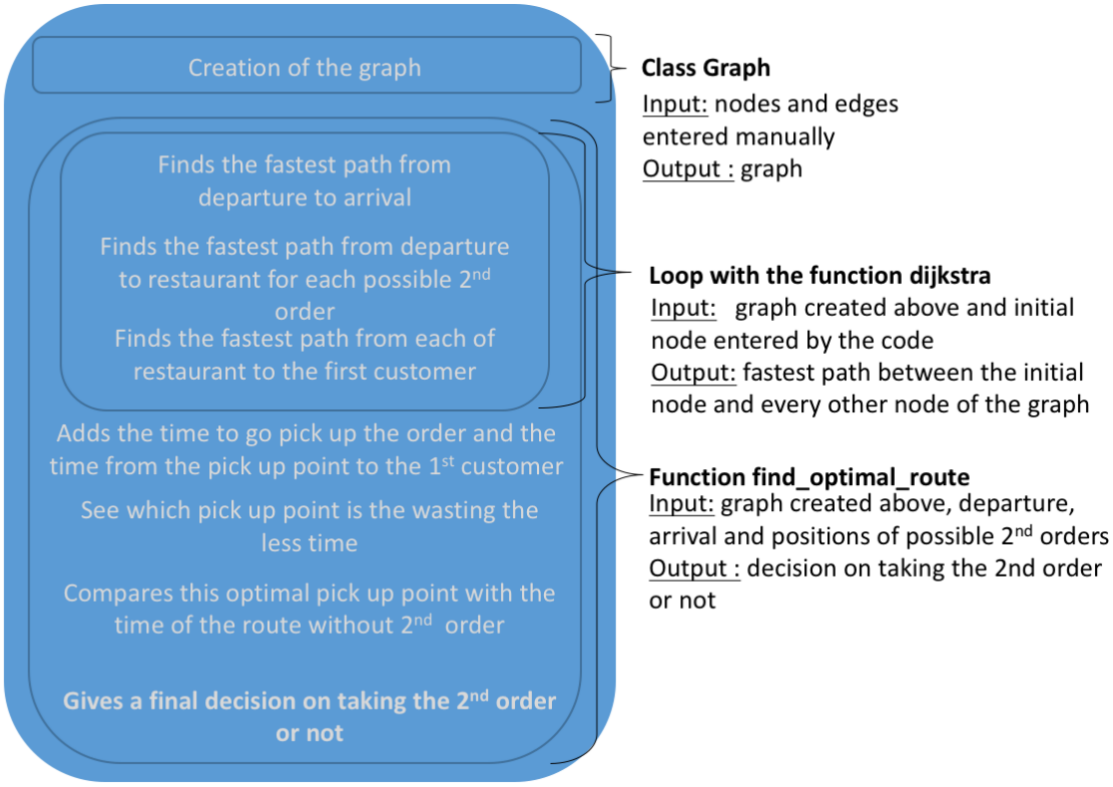
When your graph is made, the algorithm is quite easy to use, you just need to assign the departure, the arrival and the positions of the possible second orders then run the code and it will give you straight away if the rider should take a second order or not. Moreover, I think that this code could be easily adaptable to other business problems as it gives which intermediary point is optimal. For example, this could also be used by companies operating shared rides like Uber or Lyft.

The running time is quite fast and supports different sizes of data. The software also has no bugs and should always perform as required.

2) Explanation of the code

a) Overall structure

The first part of the code is the class *Graph* that contains three functions that are used to create the graph which is an input of the two main functions. The next function in the code is the *Dijkstra* function. It is used by the function *find_optimal_route* to find shortest paths. Then, based on its outputs, it decides if the rider should go to the next pick up point. Therefore, the function *find_optimal_route* is the main one as it is the one that as for output the answer to our business problem.
Here is a representation of the code's structure:



b) Flow of the code

Let's now see the flow of the program. First of all, python notices the first step which is the definition of the function *dijkstra*. Therefore, it goes into all the following lines until it is no longer part of the function. The function *dijkstra* is created. Secondly, the main function *find_optimal_route* is defined. After that, python notices the function *find_optimal_route* to execute.

It goes back to its definition and starts executing the lines but find the *dijkstra* function in it. Therefore, it goes back to the definition of this function and execute it. It is composed of a *while* loop that executes the *for* loops as long as there are still nodes that have not been visited. The first *for* loop aims at finding the nodes that minimize the path to each of the non-visited node. This loop is executed once for every non-

visited node. The second *for* loop updates the cumulated weight and predecessors until the optimal path is found for the node. Then loop breaks when every node has been visited.

Moreover, if n is the number of possible pick up points for 2nd orders, the *dijkstra* function is executed n+1 time. The first one is with the departure as the initial position which gives us the time between departure and arrival but also between departure and each pick up positions. Then the other times are when looking for the time between each pick up points and the first customer: each pick up location is then an initial node for the function at some point.

When we have all that, it executes the end of the *find_optimal_route* function. It sums up all the information gathered for each route with a *for* loop and then print the final output based on a condition generated by *if/else*.

c) Generalization

My code is generalized because it can be used in different scenarios. As mention earlier, it could be used to coordinate shared rides. Moreover, the Dijkstra function can be used separately to solve simpler short path problems as it gives the shortest time to the departure node and the path taken for every node. However, a graph should be created beforehand as it is a key input of both functions.

d) Data

In my code, data is represented in form of a graph containing nodes and weighted edges because we can represent the map of a city with such graph. The graph should be created before running the algorithm as it is part of the input. Finally, this graph must be weighted with only positive values in order to run properly.

e) Improvement of original code

I used econchick's work on Dijkstra's algorithm found on GitHubGist to code the class *Graph* and the function *Dijkstra*. However, I have linked the names of certain attributes to our problem and add simplicity in order to have a well-designed code. I also added comments to my code so that it is easier to follow the flow. Moreover, I coded the rest of the algorithm myself.

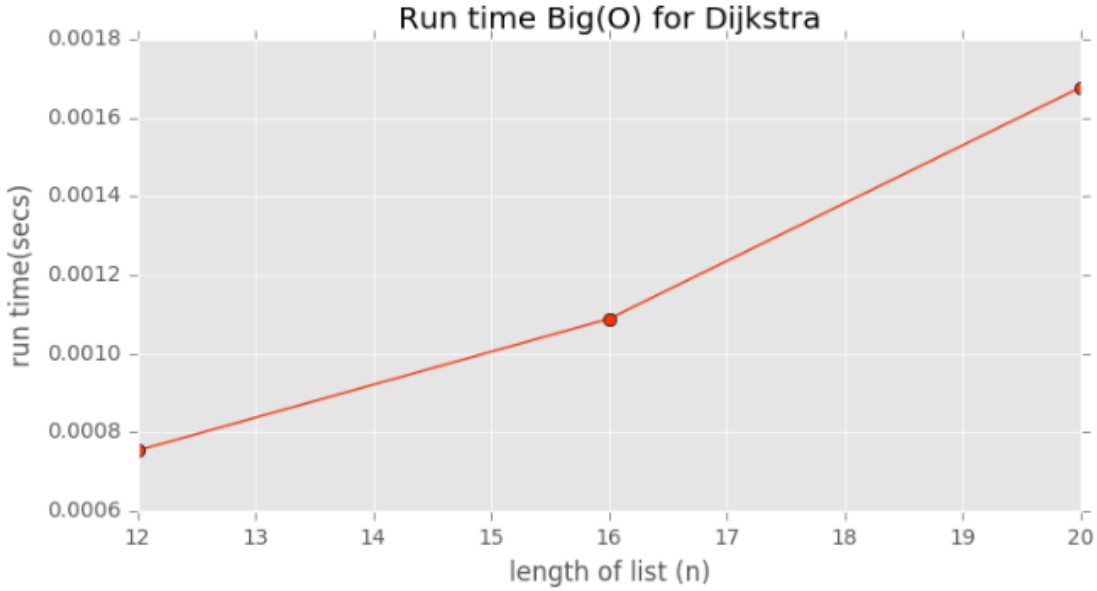
3) Complexity

When assessing the complexity of my code, I decided to run it with different input sizes. In fact, I created 3 graphs with 12, 16 and 20 nodes. I calculated the running time for each one and created the Big (O) plot below:

```
%matplotlib inline
from matplotlib import pyplot as plt
plt.style.use('ggplot')
plt.figure(figsize=(8,4))
times=[]
graphs=[graph2,graph3,graph4]
n_nodes=[len(list(g.edges))for g in graphs]

for g in graphs:
    start_time=time.time()
    find_optimal_route(departure,arrival,next_orders,g)
    end_time=time.time()
    times.append(end_time-start_time)

plt.xlabel('length of list (n)')
plt.ylabel('run time(secs)')
plt.title('Run time Big(O) for Dijkstra')
plt.plot(n_nodes,times,'o-');
```

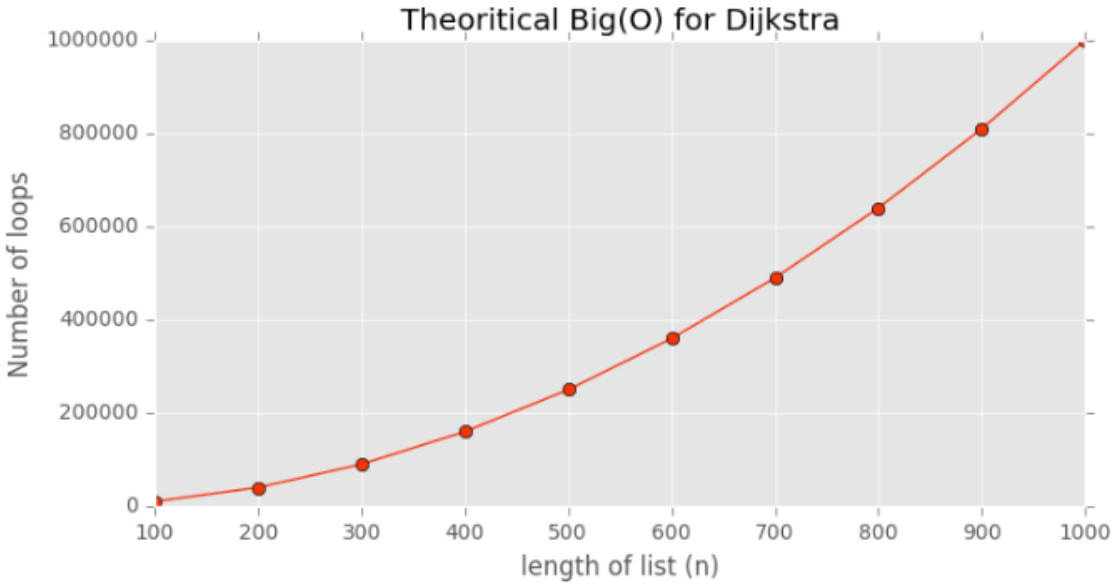


I also found that the complexity of the Dijkstra algorithm is O(n^2)[4]. Therefore, I decided to plot the theoretical Big(O) for Dijkstra that you can see below.

```
%matplotlib inline
from matplotlib import pyplot as plt
plt.style.use('ggplot')
plt.figure(figsize=(8,4))
nvals=[n for n in range (100,1001,100)]
loopvals=[n**2 for n in nvals]


plt.xlabel('length of list (n)')
plt.ylabel('Number of loops')
plt.title('Theoritical Big(O) for Dijkstra')
plt.plot(nvals,loopvals,'o-');
```

[4] 'Short Paths Problems' Gauthier Stauffer, University of Bordeaux

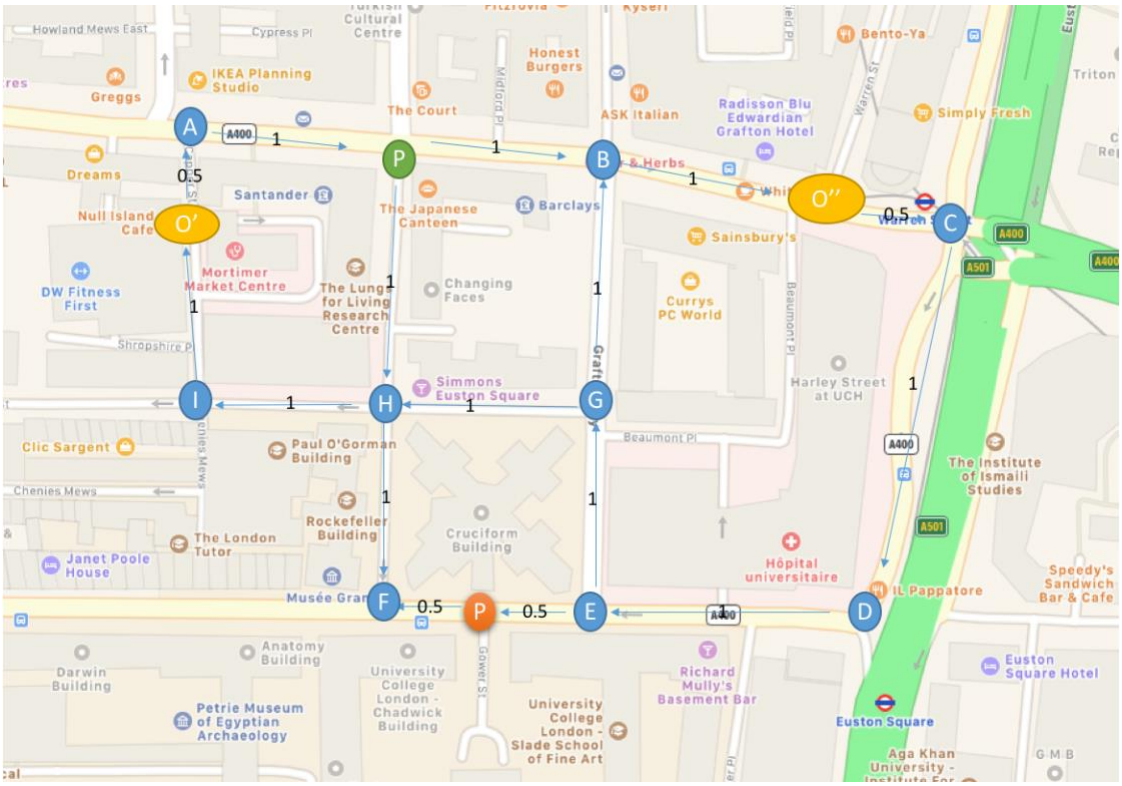


When comparing the two plots, we can see that they both follow the same trend. An increase in the size of the input has a considerable impact on the run time. In fact, increasing the size of the graph will make the algorithm test more paths, increasing the number of loops and the runtime. The fact that our algorithm has this level of complexity is a problem. In fact, I only used small graphs to test the algorithm but if you apply that to the representation of a city, the size of the input will be colossal. For example, there are more than 60 000 streets in London[5], which would represent a massive graph with the same number of edges that the code would have to go through to find the shortest path. Therefore, in this case and with this level of complexity, the algorithm might have a much longer runtime. Therefore, the runtime of the code relies a lot on the size of the input, varying in number of nodes and edges.

[5] 'The knowledge' The London Taxi Experience.

4) The data

I chose to create the graph myself because we are working on a fictive example but I think that a weighted graph can easily represent a city and its streets. In fact, you can see above how we could create a graph from a map: the green node is where the delivery guy is, he picked up an order at The Japanese Canteen for the customer located in front of the UCL main quad (orange node). Moreover, the yellow nodes are possible next orders and the blue nodes are street intersections.



The choice of generating myself the data also gave me flexibility when testing the algorithm. In fact, I generated different datasets (graphs) and input them to test my code, mainly to test the running time of my code. To analyse the running time complexity of my code, I increased the size of my data by adding nodes and edges.

5) Conclusion

I think that my code follows good design principles because of its good structure. Moreover, it is easy to read through as the comments make the code simple to follow. The fact that some of the print statements at the end also show what values were calculated and compared to take the decision makes the objective of the algorithm very clear. Furthermore, I used names consistent with the business problem which add simplicity to the reading. I also think that I minimalized my code as the core of the algorithm is made out of two functions only. Finally, I think that my code is well-designed as it answers the business problem with a specific recommendation for every possible situation.

I think that many other algorithms can solve the shortest path on a graph. For example, the Belman-Ford algorithm could also solve our business problem. In fact, this algorithm is very similar to Dijkstra's but with the difference that the weight of edges on the graph can be negative values. However, in our business case we do not need negative values. Therefore, Dijkstra seems to fit it best.

I think that my code could be improved by creating a new way to generate a graph for the input. In fact, it would be really useful to have a function that could integrate a real map.

I think that my code is generalized. In fact, the fact that the weight of the edges can represent different variables such as: fuel consumption, cost, distance. In fact, using other units could make the code applicable to other business problem like operating shared rides but it could also help any companies requiring shortest paths between two locations.

Finally, I think that one limitation of my algorithm is that in real life, the weight of the edges between nodes would have to be updated constantly in real time according to the traffic.

APPENDIX

```
#We first need to import the two following librairies which will be used in this code.
```

```
import numpy as np #Array processing for numbers, strings, records and objects.
```

```
import collections #collections will be used to import sets, lists, and dictionaries.
```

```
#The first step is to create the class Graph which contain the functions that allow us to create a graph.

class Graph:

    def __init__(self): #This first function of the class allows to initialize the objects needed for our graph:
                        #nodes, edges, and times.
        self.nodes = set()
        self.edges = collections.defaultdict(list)
        self.times = {}



    def add_node(self, value):#This function allows to add to the graph a node and a value for this node.
                              #The value will be used to find the total time to go to the node from the initial
                              #node when taking the shortest path.
        self.nodes.add(value)



    def add_edge(self, from_node, to_node, time):#This functions adds edges to the graph and a time to the edge.
        self.edges[from_node].append(to_node)
        self.times[(from_node, to_node)] = time
```

```
def dijkstra(graph, position):# Dijkstra function that allows to find the shortest time to go from the inital node

    m = {position: 0} #Set the value of the initial node at 0.
    predecessor = {}

    non_visited = set(graph.nodes)


    # We now create a loop so that the code goes through every possible route.
    while non_visited:

        min_node = None
        for node in non_visited:
            loop_counter+=1
            if node in m:
                if min_node is None:
                    min_node = node
    #We replace the value of the min node to always focus on the quickest route.
                elif m[node] < m[min_node]:
                    min_node = node

        if min_node is None:

            break #At this step, every node has been visited through every possible route, we can stop the loop.

        non_visited.remove(min_node)
        current_weight = m[min_node]

        for neighbor in graph.edges[min_node]:
            weight = current_weight + graph.times[(min_node, neighbor)] #Compute the weight at the neigbor by
                                                                        #adding the weight at the origin node
                                                                        #and the weight on the edge


            if neighbor not in m or weight < m[neighbor]:
                m[neighbor] = weight #Updating the weight
                predecessor[neighbor] = min_node #Updating the predecessor to min_node

    return m, predecessor
```

```
'''The following function is the function that will give us the final output. In fact, it is the function that will
tell us wether or not the delivery guy should stop at the restaurant to pick up the next order on his way to the
first client. Therefore, the functions needs a departure, an arrival, a list including the positions of the next
orders and a graph representing a map of the area.'''

def find_optimal_route(dep,arr,next_orders,graph):

    m_dep, predecessor_dep = dijkstra(graph,dep)
    #We use the previous function 'dijkstra' to compute the time needed to go from the departure to each nodes.

    dijkstra_results = {}
    order_time = []

    for order_point in next_orders:

        dijkstra_results[order_point], _ = dijkstra(graph,order_point)#Now we do the same but we take the
                                                                      #positions of the next orders as initial node.

    #Now we have all the information needed to compute the total time needed for each order

    no_order_time = m_dep[arr] #Time needed if the delivery guy does not pick up one more order but
                               #go straight to the client
    print("Without picking up another order, the journey time to the first customer is {}min.".format(no_order_time))
    #The following lines calculate the total time for each order. (time to go pick up the next order plus time to go
    #to the client from the restaurant of the next order)
    for order_point in next_orders:

        dep_order = m_dep[order_point]
        order_arr = dijkstra_results[order_point][arr]
        total_time = dep_order + order_arr
        order_time.append(total_time)
    optimal_order = next_orders[np.argmin(order_time)] #Find the order which adds minimum time to the delivery
                                                       #of the first order.
    optimal_time = min(order_time) #Find the total time for this optimal next order.

    '''Now we say that if it adds more than 5 minutes to the first delivery to pick up the next order,
    the delivery guy should go straight to the customer without picking up the next order. '''

    print("Here are the journey times to go to the first customer if we pick up another order on the way:")
    for i, order_point in enumerate (next_orders):

        print ("If we pick up {}, the journey time will be {}min.".format(order_point, order_time[i]))

    if optimal_time>no_order_time+5:
        print ("Therefore, the delivery guy should go straight to the client at {}, the estimated travel time is {}min.'
    else:
        print ("Therefore, the delivery guy should go pick up the order of the next client at {}, then deliver the first
```

```
#We create a graph to test our program
graph = Graph()

#We add the nodes
graph.add_node('A')#departure
graph.add_node('B')#order2
graph.add_node('C')
graph.add_node('D')
graph.add_node('E')#order1
graph.add_node('F')#order3
graph.add_node('G')
graph.add_node('H')#arrival



#We add the edges
graph.add_edge('A','C',3)
graph.add_edge('A','B',1)
graph.add_edge('B','D',2)
graph.add_edge('B','E',3)
graph.add_edge('C','D',1)
graph.add_edge('C','G',2)
graph.add_edge('D','F',1)
graph.add_edge('D','H',2)
graph.add_edge('E','F',2)
graph.add_edge('E','G',1)
graph.add_edge('F','H',1)
graph.add_edge('G','H',2)


#Then finally we define the departure, the arrival which is the position of the first customer and
#the pick up points of next orders.
departure='A'
arrival='H'
order1='E'
order2='D'
order3='G'
next_orders=[order1,order2,order3]
```

```
#Now we can run our algorithm
find_optimal_route(departure,arrival,next_orders,graph)
```

```
Without picking up another order, the journey time to the first customer is 5min.
Here are the journey times to go to the first customer if we pick up another order on the way:
If we pick up E, the journey time will be 7min.
If we pick up D, the journey time will be 5min.
If we pick up G, the journey time will be 7min.
Therefore, the delivery guy should go pick up the order of the next client at D, then deliver the first order at H.
The estimated travel time is 5min.
```

```
#We create a graph to test our program
graph2 = Graph()

#We add the nodes
graph2.add_node('A')#departure
graph2.add_node('B')#order2
graph2.add_node('C')
graph2.add_node('D')
graph2.add_node('E')#order1
graph2.add_node('F')#order3
graph2.add_node('G')
graph2.add_node('H')#arrival
graph2.add_node('I')
graph2.add_node('J')
graph2.add_node('K')
graph2.add_node('L')



#We add the edges
graph2.add_edge('A','C',3)
graph2.add_edge('A','B',1)
graph2.add_edge('B','D',2)
graph2.add_edge('B','E',3)
graph2.add_edge('C','D',1)
graph2.add_edge('C','G',2)
graph2.add_edge('D','F',1)
graph2.add_edge('D','H',2)
graph2.add_edge('E','F',2)
graph2.add_edge('E','G',1)
graph2.add_edge('F','H',1)
graph2.add_edge('G','H',2)
graph2.add_edge('C','I',1)
graph2.add_edge('I','G',1)
graph2.add_edge('A','J',1)
graph2.add_edge('E','K',3)
graph2.add_edge('K','H',2)
graph2.add_edge('K','F',2)
graph2.add_edge('D','L',1)
graph2.add_edge('L','G',1)
```

```
    import time
    start_time=time.time()
    find_optimal_route(departure,arrival,next_orders,graph2)
    end_time=time.time()
    print("__{} seconds___".format(end_time-start_time))
```

```
Without picking up another order, the journey time to the first customer is 5min.
Here are the journey times to go to the first customer if we pick up another order on the way:
If we pick up E, the journey time will be 7min.
If we pick up D, the journey time will be 5min.
If we pick up G, the journey time will be 7min.
Therefore, the delivery guy should go pick up the order of the next client at D, then deliver the first order at H.
The estimated travel time is 5min.
__0.0018978118896484375 seconds___
```

```
#We create a graph to test our program
graph3 = Graph()

#We add the nodes
graph3.add_node('A')#departure
graph3.add_node('B')#order2
graph3.add_node('C')
graph3.add_node('D')
graph3.add_node('E')#order1
graph3.add_node('F')#order3
graph3.add_node('G')
graph3.add_node('H')#arrival
graph3.add_node('I')
graph3.add_node('J')
graph3.add_node('K')
graph3.add_node('L')
graph3.add_node('M')
graph3.add_node('N')
graph3.add_node('O')
graph3.add_node('P')


#We add the edges
graph3.add_edge('A','C',3)
graph3.add_edge('A','B',1)
graph3.add_edge('B','D',2)
graph3.add_edge('B','E',3)
graph3.add_edge('C','D',1)
graph3.add_edge('C','G',2)
graph3.add_edge('D','F',1)
graph3.add_edge('D','H',2)
graph3.add_edge('E','F',2)
graph3.add_edge('E','G',1)
graph3.add_edge('F','H',1)
graph3.add_edge('G','H',2)
graph3.add_edge('C','I',1)
graph3.add_edge('I','G',1)
graph3.add_edge('A','J',1)
graph3.add_edge('E','K',3)
graph3.add_edge('K','H',2)
graph3.add_edge('K','F',2)
graph3.add_edge('D','L',1)
graph3.add_edge('L','G',1)
graph3.add_edge('C','M',3)
graph3.add_edge('M','H',1)
graph3.add_edge('D','M',1)
graph3.add_edge('E','N',1)
graph3.add_edge('N','H',1)
graph3.add_edge('A','O',2)
graph3.add_edge('O','K',2)
graph3.add_edge('B','P',2)
graph3.add_edge('P','K',3)
```

```
    import time
    start_time=time.time()
    find_optimal_route(departure,arrival,next_orders,graph3)
    end_time=time.time()
    print("__{} seconds___".format(end_time-start_time))
```

```
Without picking up another order, the journey time to the first customer is 5min.
Here are the journey times to go to the first customer if we pick up another order on the way:
If we pick up E, the journey time will be 6min.
If we pick up D, the journey time will be 5min.
If we pick up G, the journey time will be 7min.
Therefore, the delivery guy should go pick up the order of the next client at D, then deliver the first order at H.
The estimated travel time is 5min.
__0.0025010108947753906 seconds___
```

```
graph4 = Graph()

#We add the nodes
graph4.add_node('A')#departure
graph4.add_node('B')#order2
graph4.add_node('C')
graph4.add_node('D')
graph4.add_node('E')#order1
graph4.add_node('F')#order3
graph4.add_node('G')
graph4.add_node('H')#arrival
graph4.add_node('I')
graph4.add_node('J')
graph4.add_node('K')
graph4.add_node('L')
graph4.add_node('M')
graph4.add_node('N')
graph4.add_node('O')
graph4.add_node('P')
graph4.add_node('Q')
graph4.add_node('R')
graph4.add_node('S')
graph4.add_node('T')


#We add the edges
graph4.add_edge('A','C',3)
graph4.add_edge('A','B',1)
graph4.add_edge('B','D',2)
graph4.add_edge('B','E',3)
graph4.add_edge('C','D',1)
graph4.add_edge('C','G',2)
graph4.add_edge('D','F',1)
graph4.add_edge('D','H',2)
graph4.add_edge('E','F',2)
graph4.add_edge('E','G',1)
graph4.add_edge('F','H',1)
graph4.add_edge('G','H',2)
graph4.add_edge('C','I',1)
graph4.add_edge('I','G',1)
graph4.add_edge('A','J',1)
graph4.add_edge('E','K',3)
graph4.add_edge('K','H',2)
graph4.add_edge('K','F',2)
graph4.add_edge('D','L',1)
graph4.add_edge('L','G',1)
graph4.add_edge('C','M',3)
graph4.add_edge('M','H',1)
graph4.add_edge('D','M',1)
graph4.add_edge('E','N',1)
graph4.add_edge('N','H',1)
graph4.add_edge('A','O',2)
graph4.add_edge('O','K',2)
graph4.add_edge('B','P',2)
graph4.add_edge('P','K',3)
graph4.add_edge('P','Q',3)
graph4.add_edge('Q','G',1)
graph4.add_edge('D','R',1)
graph4.add_edge('R','M',1)
graph4.add_edge('I','S',2)
```

```
    import time
    start_time=time.time()
    find_optimal_route(departure,arrival,next_orders,graph4)
    end_time=time.time()
    print("__{} seconds___".format(end_time-start_time))
```

```
Without picking up another order, the journey time to the first customer is 5min.
Here are the journey times to go to the first customer if we pick up another order on the way:
If we pick up E, the journey time will be 6min.
If we pick up D, the journey time will be 5min.
If we pick up G, the journey time will be 7min.
Therefore, the delivery guy should go pick up the order of the next client at D, then deliver the first order at H.
The estimated travel time is 5min.
__0.0017206668853759766 seconds___
```

Reference list

‘Uber Eats and the $6 billion bookings run rate: The AI success story no one is talking about’ RB, Oct 2018.
https://venturebeat.com/2018/10/02/uber-eats-and-the-6b-bookings-run-rate-the-ai-success-story-no-one-is-talking-about/

‘8 Issues To Consider When Setting Up Your Food Delivery Service’, Lior Sion, Oct 2017
https://www.bringg.com/blog/latest/8-issues-to-consider-when-setting-up-your-food-delivery-service/

‘Python implementation of Dijkstra's Algorithm’ by econchick on GitHubGist.
https://gist.github.com/econchick/4666413

‘Short Paths Problems’ Gauthier Stauffer, University of Bordeaux
https://www.math.u-bordeaux.fr/~gstauffe/cours/MSE3211A/MSE3211A_3.pdf

‘The knowledge’ The London Taxi Experience.
http://www.the-london-taxi.com/london_taxi_knowledge