University of Messina

DEPARTMENT OF ENGINEERING
MASTER'S DEGREE COURSE IN ENGINEERING AND COMPUTER SCIENCE

Distributed Systems

Optimization of the food delivery problem from the sustainable mobility point of view

Student
Marco Garofalo - 516005

Professor
Antonio Puliafito

Academic year 2021/2022

Contents

Introduction

Since 2014 the use of mobile applications for food delivery by restaurants has begun to be massive, thanks to the widespread use of smartphones and applications in mobile stores. Today the food delivery industry generates around 26.5 billion dollars and it is estimated that in 2024 it could reach 32.5 billion dollars, indicating how fast this sector is growing.[1]

During the Covid-19 pandemic, it could be expected that the food delivery sector, as it is linked to the catering sector which has undergone several restrictions due to the gathering of groups of people, has gone down. But in contrast to this statement, it is the demand for food ordered online that has grown by 300%[2].

All these considerations and more, imply that such a large sector involves a lot of instrumentation that generate collective effects, just think of the large amount of vehicles used and their CO2 emissions.
Although CO2 emissions have been reduced due to mobility restrictions due to Covid-19, according to some statistical data it was expected that in 2021 Italy would suffer an increase in greenhouse gases, with an increase in emis-

sions of 0.3%. The estimate is developed by the Istituto Superiore per la Protezione e la Ricerca Ambientale (Ispra) which - based on the first available data on the trend of the year 2021 - reports that the cause of the increase in emissions is the ”consequence of resumption of economic activities”. This estimate is achieved thanks to the reduction of CO2 emissions for the production of electricity (-1.4%), with an increase in emissions in other sectors, such as industry (+ 2.7%) and heating (+ 1.5%)[3].
Furthermore, climate and energy policies are undergoing a phase of deep revision following the signing of the Paris Agreement, where the goal for the European Union is to reduce greenhouse gas emissions by at least 40% over the year. 1990, by 2030[4].
Even Italy, one of the member states of the United Nations Framework Convention on Climate Change, in January 2021 published the Italian long-term strategy on the reduction of greenhouse gas emissions, which emphasizes the need for structural and technological changes and behaviors that minimize greenhouse gas emissions in the medium and long term.
Sustainability is assuming an increasingly important role within any sector on the Italian territory, and therefore on the basis of these estimates and considerations an interesting case study is the optimization of the problem of food delivery by restricting its domain: by setting a time window in which to place orders and by optimizing the route and stops of n electric vehicles used to collect and deliver orders.

The food reservation and transport service that has been proposed will be structured on the basis of a client-server infrastructure, in this report we

describe the server core.
The server will calculate the optimized route for the collection of orders based on the number of electric vehicles available, their battery status, the number of items of each orders and the location of the restaurants involved.

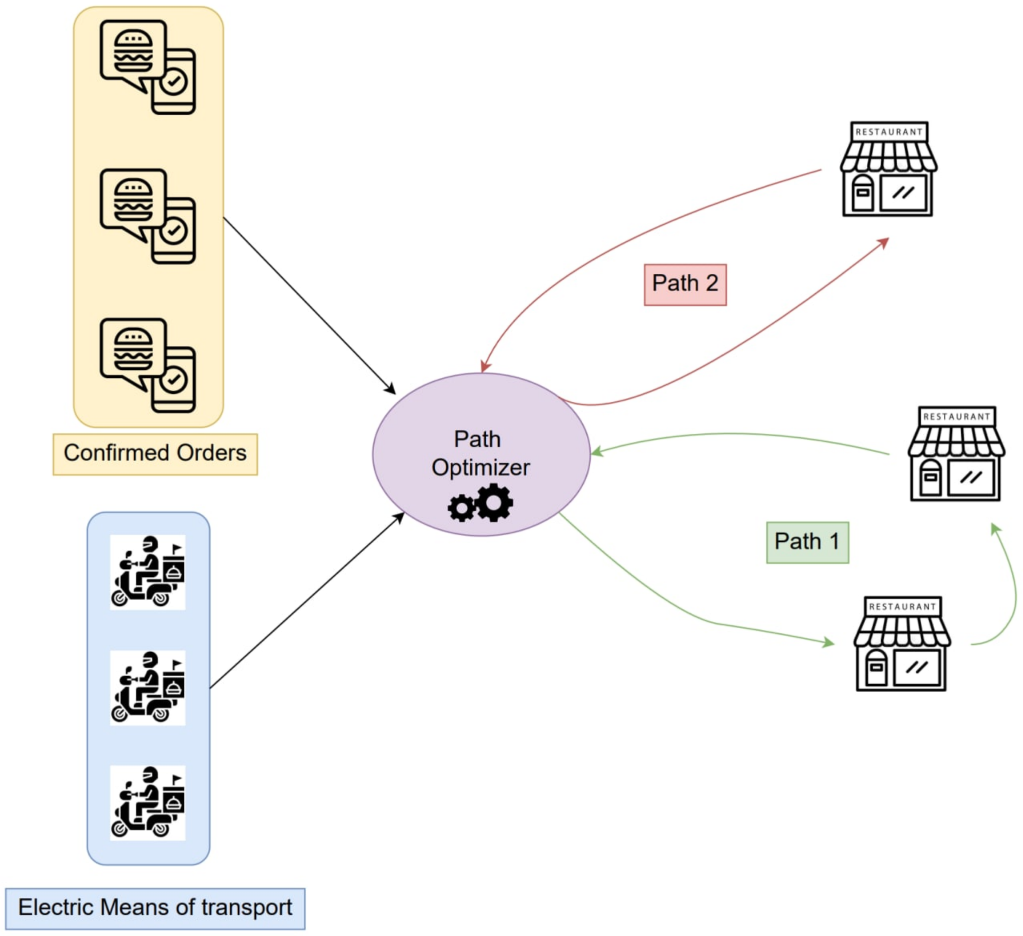


Figure 1: `Diagram representing server the optimization flow`

So in a basic scenario it is possible that the server returns one or more paths depending on the riders involved.

Chapter 1

Design

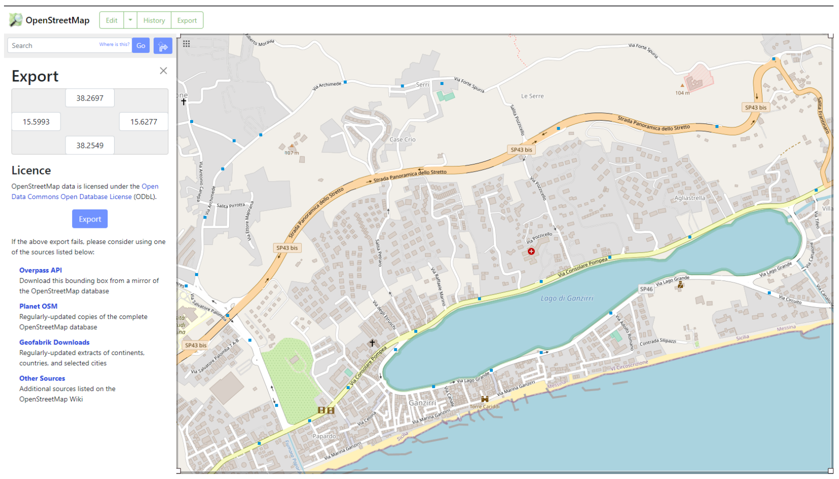
The very first challenge of the project was to obtain a dataset, required to study the domain of the problem and to develop the right techniques. The leader in the field of data related to road maps is Google Maps, but since the service is not completely free, other alternatives have been investigated.

1.1 Open Street Maps

OpenStreetMap[5] (OSM) is a collaborative project to create a free editable geographic database of the world. The geodata underlying the maps is considered the primary output of the project. The creation and growth of OSM has been motivated by restrictions on use or availability of map data across much of the world, and the advent of inexpensive portable satellite navigation devices.

Figure 1.1: `Open Street Map logo.`

By taking a look on the OSM website, it's very easy to extract data in the osm format. It is indeed possible to manually select an area and extract the coordinates of the selected rectangle.



38.2697
15.5993
15.6277
38.2549

Licence

OpenStreetMap data is licensed under the Open Data Commons Open Database License (ODbL).

If the above export fails, please consider using one of the sources listed below:

Overpass API
Download this bounding box from a mirror of the OpenStreetMap database

Planet OSM
Regularly-updated copies of the complete OpenStreetMap database

Geofabrik Downloads
Regularly-updated extracts of continents, countries, and selected cities

Other Sources
Additional sources listed on the OpenStreetMap Wiki

Figure 1.2: `Selected Area from OSM`

1.2 OSMnx

The Python programming language is very popular and has a huge community of contributors in the science field. Since the main purpose of the project is to work with a road map seen as a graph, python it's the best choice in terms of supporting libraries. A useful module for this purpose is OSMnx[6], a python module that implements a bridge between OSM and python throughout the data structures provided by the NetworkX[7] module. With OSMNX it's possible to retrieve a graph given the coordinates in the real world:

```
import osmnx as ox
G = ox.graph_from_place(  'Messina, Italy', network_type = 'drive' )
ox.plot_graph(G)
```

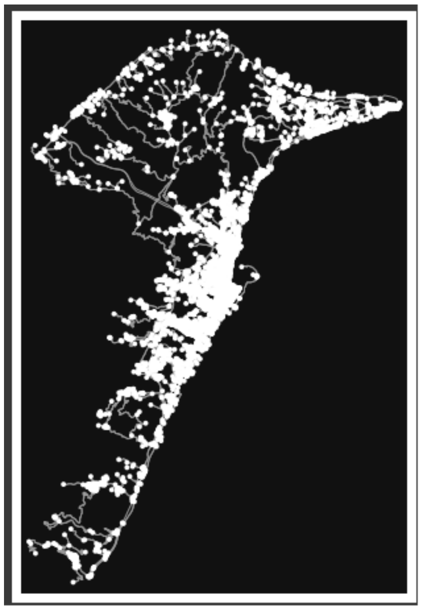


Figure 1.3: Graph of the city of Messina

In the above code, we obtained the the map of Messina as a graph and plotted it.
For the purpose of this project we want to apply an optimization technique on a subgraph of the city of Messina, so we can explicitly give the boundaries of the map rectangle we want to consider:

```python
G = ox.graph_from_bbox(north, south, east, west, network_type = 'drive')
ox.plot_graph(G)
```

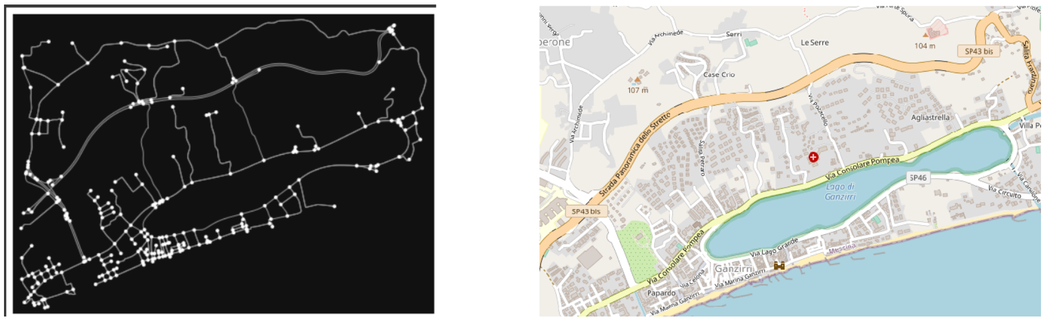


Figure 1.4: Subgraph of Messina

1.3 Open-Elevation API

Once we got the dataset to work with, the next step is to enrich the data in order to evaluate them from the point of view of sustainable mobility. The selected strategy aims to find the elevation of each node that will allow us to compute the slope of each edge. Again in this field the leader of this service is Google Elevation API, but since it's a paid service, we evaluated other alternatives open source. Open-Elevation API[8] provides a free and open-

source alternative, very easy to setup since it's possible to host the service autonomously or just doing some HTTP requests specifying the coordinates of each node to get their elevation in meters.
From their documentation[9] it's possible to do an HTTP GET request in the form:
`https://api.open-elevation.com/api/v1/lookup?locations=`$\{node_1|node_2|...|node_n\}$
where each node is a tuple composed by it's coordinates in latitude and longitude.

1.4 Weight Function

In order to enrich the information contained in the graph, we designed a function able to compute the new weight associated to each edge, taking into account a compromise between the length of the street and its slope.

$$new\_weight(e) = \alpha * length(e) + (1 - \alpha) * slope(e)$$

We can observe that the greater is α, the greater the importance given to the length while the importance given to the slope of the street decrease. So to give more importance to the slope it will be enough to provide small value of α.

1.5 Consume Function

To be able to give a quantitative measure of the consumption associated with an edge. A study was conducted on the consumption of electric scooters and on the calculation of the road slope. The source of informations considered were some details provided by the companies who sell electrical scooters.

1.5.1 Electrical Scooter Consumption

According to vaielettrico.it [10], modern electric scooters have an autonomy range between $70km$ and $100km$, this means that roughly a modern scooter consumes 1% of battery for $1km$. Also, again according to vaielettrico.it, modern scooters can travel on roads with a maximum gradient of 30%

1.6 Slope expressed in percentage

So if we can compute the difference of height of a path from A to B as the difference between the elevation of B and the elevation of A:

$$diff\_height(A, B) = B - A$$

we can then express this value in percentage with respect of the length of the street:

$$slope\_perc(A, B, L) = \frac{B - A}{L} \cdot 100$$

1.6.1 Slope of the street

According to [11], if we consider a path from point A to point B of $100cm$, if the differerence in height between the two points is of $30cm$, we have a slope of 30%, of course this calcolous does not consider the instant slope in each point between A and B but the mean slope.

We can formalize a cost function by defining two intervals referring to $1km$ of street:

- the domain of the slope
- the domain on the consumption

The cost function will represent the relation between the slope percentage of an edge and the consumption associated to it.
We assume that in a negative slope there will be always a value of consumption even if relatively small, for this reason the inferior extreme of the consumption interval will be 0.5% of battery, we know also that for $1km$ of street the consumption is roughly 1% of battery and by observing the graph considered, we saw that we never reach a value as 30% of slope so we will consider it as maximum value of the slope interval, assuming a relation little bit more steep than linear, assuming that at 30% of slope correspond 33% of consumption.
So far we have then defined our intervals:

- $[-40, 30]$ for the percentage of slope associated to an edge
- $[0.5, 33]$ for the percentage of consumption associated to a percentage of slope.

We can then plot our function, recalling that at 0 slope of a $1mk$ path we associated 1% of battery consumption. And we computed it as:

$$\begin{cases} 0.01x + 1 & \text{if } -40 \leq x < 0 \\ 1.08x + 1 & \text{if } 0 \leq x < 30 \end{cases}$$

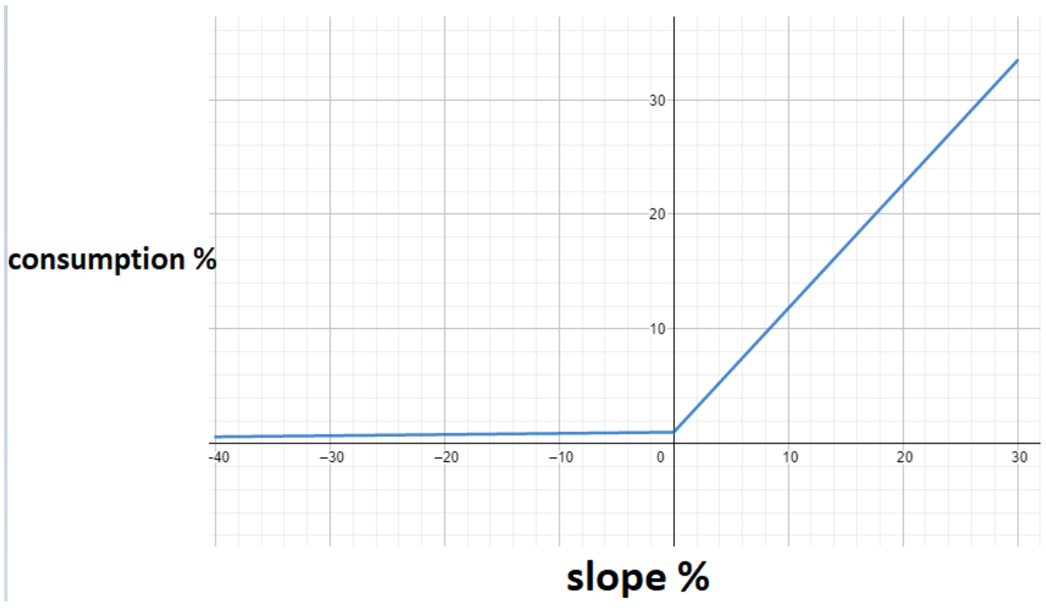


Figure 1.5: Consumption Function

1.7 Distance between two coordinates

Once we got all the information we wanted, for the algorithm purpose the only missing part is to compute the distance as the crow flies. According to movable [12], the haversine formula is a great tool to accomplish this task:

$$a = \sin^2(\Delta\phi/2) + \cos\phi_1 \cdot \cos\phi_2 \cdot \sin^2(\Delta\gamma/2)$$

$$c = 2 \cdot atan2(\sqrt{a}, \sqrt{1-a})$$

$$d = R \cdot c$$

where ϕ is latitude, γ is longitude and R is earth's radius (roughly $6.371km$)

Chapter 2

Implementation

Once we got all the data and designed the functions, it was necessary to translate everything in the python programming language. For this project Google Colaboratory has been used in order to avoid any kind of issues releated to the local environment.

2.1 Add elevations

We wanted to add the elevation of each node ad a property of the corresponding object in the data structure. The problem encountered in this part was that the subgraph selected contained 279 nodes, and because of that the URL formatted to do a GET HTTP request to Open-Elevation API was too long, return an error code of 414. So, empirically, we found out that the request handled at most 160 nodes roughly, so we developed a function that was partitioning the nodes of the graph in partition of 160 nodes each, sending the request and combining the results:

```python
def add_elevations(G,simulate=False):
  elevations = []
  URL = 'https://api.open-elevation.com/api/v1/lookup?locations='

  partitions = math.ceil(len(G.nodes())/160)

  if not simulate:

    inf_extreme = 0
    sup_extreme = 0
    for i in range(1,partitions+1):
      query_coords = ""
      inf_extreme = sup_extreme
      if i == partitions:
        sup_extreme = len(G.nodes())
      else:
        sup_extreme = 160*i
      for node in list(G.nodes())[inf_extreme:sup_extreme]:
        query_coords += str(G.nodes[node]['y'])+','+str(G.nodes[node]['x'])+'|
      query_coords = query_coords[:-1]
      x = requests.get(URL+query_coords)

      data = x.json()

      #Costruisco l'array delle elevazioni
```

```
   for el in data['results']:
    elevations.append(el['elevation'])
 else:
  for el in G.nodes():
   elevations.append(random.randint(0,113))
 #Aggiungo elevation ai nodi
 i = 0
 for node in G.nodes(data=True):
  x = node[1]
  x['el'] = elevations[i]

  i = i + 1

 return elevations
```

2.2 Updating the weights

Since we wanted to enrich the content of the weights of each edge with the elevation between the extremes, we defined all the functions necessary to reach this objective. We observed different kind of values for elevations and length of street, and since we didn't want to prioritize the impact of a term in an uncontrolled way, we normalized them between 0 and 1.

```
def norm_slope(x, lowest=min_slope, highest=max_slope):
 return (x-lowest)/(highest-lowest)

```

```python

def norm_distance(x, lowest=min_dist, highest=max_dist):
  return (x-lowest)/(highest-lowest)
```

Hence the function to change the weight has been formalized

```python
def new_weight(length,elevation_A,elevation_B,alpha=0):
  return
        (alpha * norm_distance(length) +
        (1-alpha) * norm_slope(elevation_B - elevation_A))
```

2.3 Finding the route

To find the shortest path between two points we used the function provided by NetworkX that internally uses the Dijkstra algorithm by default, but we can explicitly say to use the Bellman-Ford algorithm.

```python
def find_route(graph,origin,destination,alg='dijkstra'):
  origin_node = ox.distance.nearest_nodes(graph,*origin)
  destination_node = ox.distance.nearest_nodes(graph,*destination)
  route = nx.shortest_path(graph, origin_node, destination_node,
                                weight = 'weight',method=alg)

  return route
```

2.4 Computing the consumption

Finally we implemented all the function discussed in the design part:

2.4.1 Computing the percentage of elevation

```python
def perc_elevation(el_A,el_B,length):
  return ((el_B-el_A)/length)*100
```

2.4.2 Computing the percentage of consumption

```python
def get_consumption(perc_elevation):
  if perc_elevation < 0:
    return 0.01*perc_elevation + 1
  else:
    return 1.08*perc_elevation + 1
```

2.4.3 Computing the percentage of consumption of a route

```python
def get_route_consumption(G,route):
  consumption = 0
  for i  in range(0,len(route)-1):
      consumption += G[route[i]][route[i+1]][0]['perc_consumption']
  return consumption
```

2.4.4 Computing the air distance

```python
def get_air_distance(pos_A,pos_B):

  phi_1,phi_2 = pos_A[1]*(math.pi/180),pos_B[1]*(math.pi/180)
  delta_phi = (phi_2 - phi_1)
```

```
    delta_gamma  = (pos_B[0] - pos_A[0])* (math.pi/180)
    a = math.sin(delta_phi/2) * math.sin(delta_phi/2) + math.cos(phi_1) *
        math.cos(phi_2) * math.sin(delta_gamma/2) * math.sin(delta_gamma/2)
    c = 2 * math.atan2(math.sqrt(a),math.sqrt(1-a))
    R = 6371e3 #earth ray in meters



    d = R * c
    return d
```

2.5 The algorithm

The algorithm provided aims to optimize the paths done by the rider using electrical vehicles, considering the feasibility of accomplish an order. The idea is to try to find a solution for each order, where with solution we mean the feasibility for the rider to go towards the restaurant, get the order and go to the university. We follow a greedy approach by assuming that the best option is always to schedule the maximum number of order to the rider with the greater percentage of battery left. So the first step is to get the rider with the greater percentage of battery and the compute the air distances between the selected rider and the location of the orders left. We select the closest one and we compute the consumption of the forward path and the backward one. If the rider has still battery percentage left, we go to the next order refunding the consumption of the backward path, otherwise we select the rider with the maximum capacity and loop again over the orders.

The algorithms ends when all the orders have been scheduled or when all the riders have been considered for each order. Also we assume that a rider has a finite capacity of items that he can transport, so we evaluate if the rider is able to get all the food from a restaurant to continue, otherwise we mark the order as ”excluded” for this rider.

```python
def optimize_path(G,S,riders,orders):
  associations = { }
  riders_orders =  { }
  riders_copy = copy.deepcopy(riders)
  orders_copy = copy.deepcopy(orders)
  visited = []
  excluded_orders = []
  old_rider = None
  is_capacity_considered = True
  for r in riders_copy:
    associations[r['id']] = []
    riders_orders[r['id']] = []

  while orders_copy and len(visited) != len(riders):
    R = get_max_capacity(riders_copy)

    if R['id'] not in visited:
      visited.append(R['id'])

    s = R['start']
    old_cons_b = 0
    old_route_b = []

    while R['battery'] > 0 and orders_copy:

      distances = get_distances(s,orders_copy)

      if R['id'] not in visited:
        max_D_index = np.argmin(distances)
        D = orders_copy.pop(max_D_index)
      else:
        min_D_index = np.argmin(distances)
        D = orders_copy.pop(min_D_index)



      if R['order_capacity'] < D['capacity'] and is_capacity_considered:
        excluded_orders.append(D)
        continue

      route_f = find_route(DG,s,D['coords'])
      cons_f = get_route_consumption(DG,route_f)
      route_b = find_route(DG,D['coords'],S)
      cons_b = get_route_consumption(DG,route_b)

      if (R['battery'] - cons_f - cons_b) > 0:
        s = D['coords']
        if associations[R['id']]:
          associations[R['id']].pop()
          R['battery'] = R['battery'] + old_cons_b

        R['battery'] = R['battery'] - cons_f - cons_b
        R['order_capacity'] = R['order_capacity'] - D['capacity']
        associations[R['id']].append(route_f)
        associations[R['id']].append(route_b)
        riders_orders[R['id']].append(D)
        old_route_b = route_b
        old_cons_b = cons_b
      else:
        orders_copy.append(D)
        break
    orders_copy += excluded_orders
    excluded_orders = []
  return associations,riders_orders
```

The algorithm returns the associations between the riders and the orders. To evaluate the rider with the maximum percentage of battery left a function has been developed:

```
def get_max_capacity(riders):
  max_capacity = 0
  j = 0
  for i in range(0,len(riders)):
```

```python
    if max_capacity < riders[i]['battery']:
      max_capacity = riders[i]['battery']
      j = i

  return riders.pop(j)
```

Chapter 3

Results

Some experiments have been conducted on the algorithm proposed by generating three riders with three different starting positions and three orders located in different points of the graph.
The cases have been observed

- Fixing the number of items contained in one order and varying the percentage of battery
- Fixing the percentage of battery of the riders and varying the number of items of the orders.

3.0.1 Fixing the number of items contained in one order

The initial data are structured as follow:

```
riders = [
    {
        'start': (15.6190,38.2590),
        'id': 'rider-1', #1,3
        'battery':30,
        'order_capacity':5
    },
    {
        'start': (15.6077,38.2559),
        'id': 'rider-2', #2
        'battery':10,
        'order_capacity':12
    },
    {
        'start': (15.6052,38.2677),
        'id': 'rider-3',
        'battery':10,
        'order_capacity': 12
    },
]

orders = [
      {
          'id' : 1,
          'capacity': 2,
          'coords': (15.6090,38.2595)
      },
      {
          'id' : 2,
          'capacity': 7,
          'coords':  (15.6076,38.2647)
      },
      {
          'id': 3,
          'capacity': 3,
          'coords': (15.6105,38.2569)
      }
    ]
```

The first rider has more percentage of battery, so what we expect is that he will be able to do two orders if the two location are close enough each other. The results shown as predicted: the first rider has been associated to the third order since is the closest to his starting position and to the first order. The second order, instead, has been associated to the third rider since the second one didn't have enough battery to cover the forward path and the backward path.

```
{
'rider-1':
    [
        {'id': 3, 'capacity': 3, 'coords': (15.6105, 38.2569)},
        {'id': 1, 'capacity': 2, 'coords': (15.609, 38.2595)}
    ],
'rider-2': [],
'rider-3':
     [
          {'id': 2, 'capacity': 7, 'coords': (15.6076, 38.2647)}
     ]
}
```

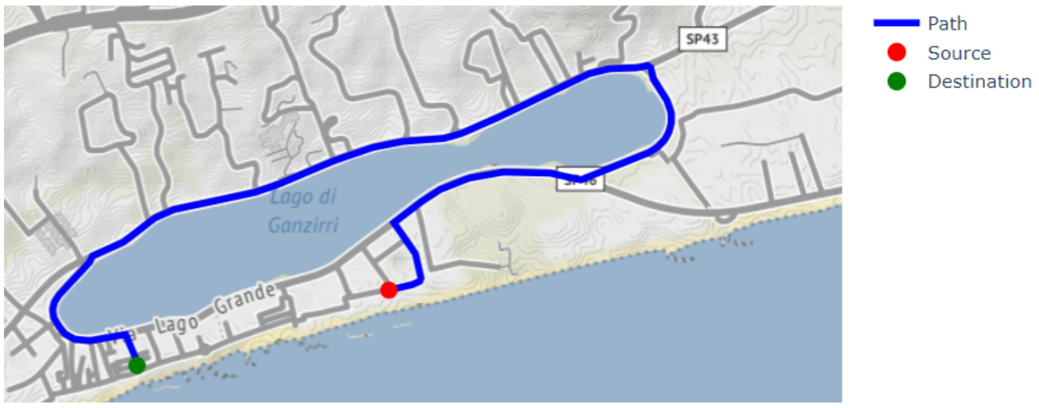


Figure 3.1: Rider-1 doing path between his starting position and order-3

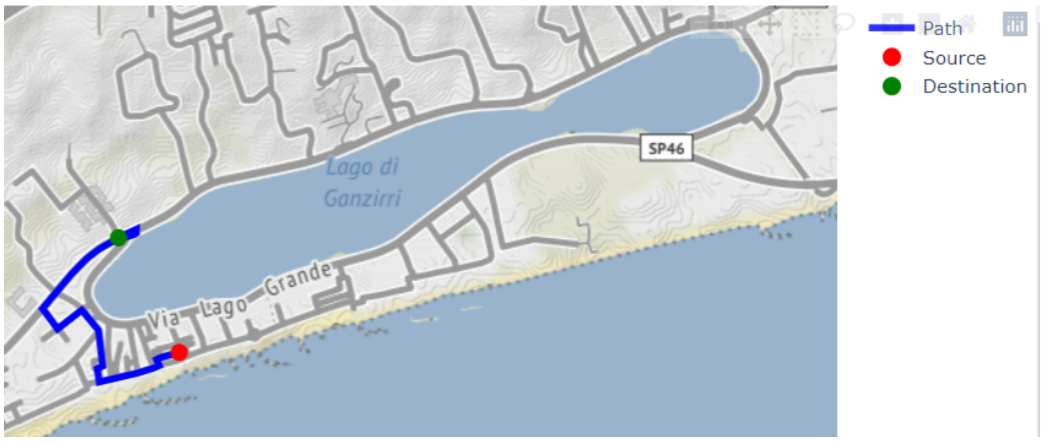


Figure 3.2: Rider-1 doing path between his new position (order-3) and order-1

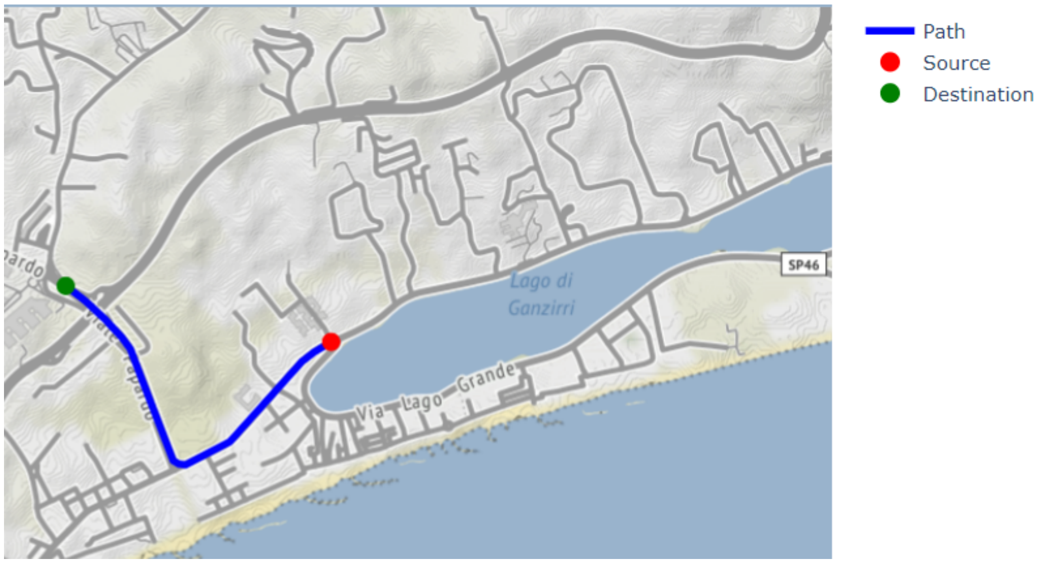


Figure 3.3: Rider-1 doing path between the order-1 and the university (backward)

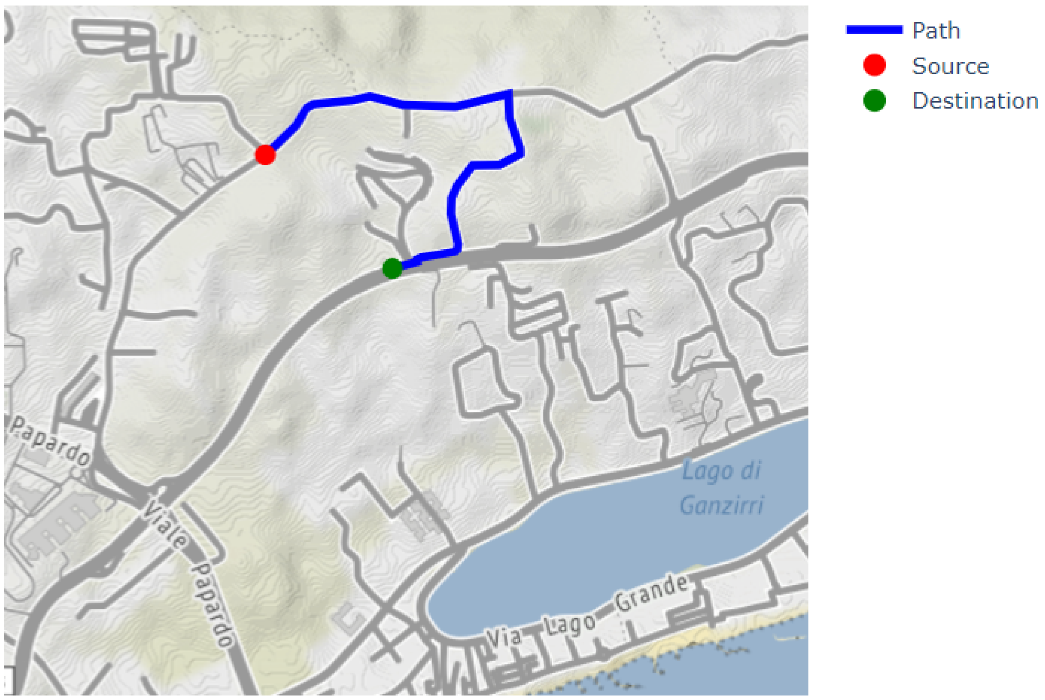


Figure 3.4: Rider-3 doing path between his starting position and order-2

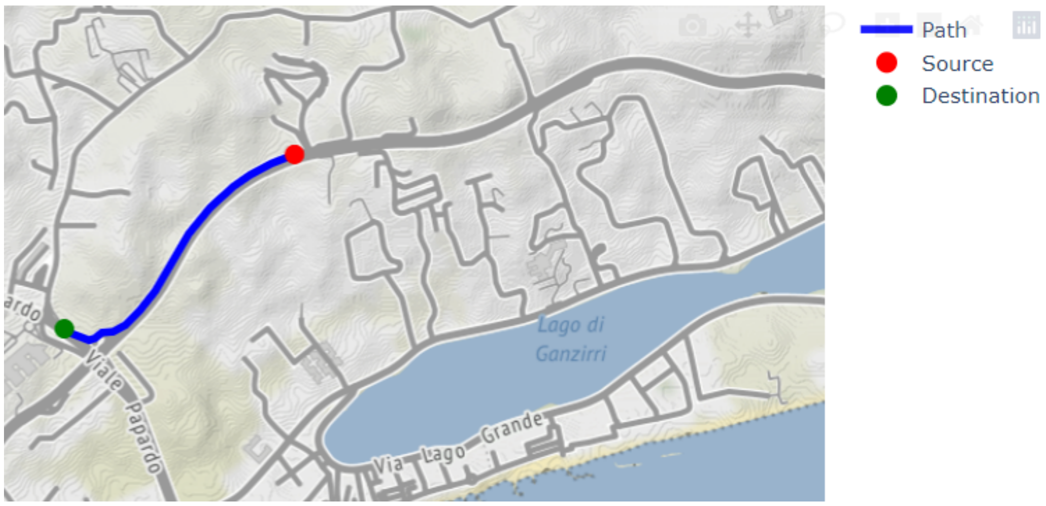


Figure 3.5: Rider-3 doing path between his order-2 and the university (backward)

Changing the percentage of battery lead to another result, again predictable, if we put same percentage of battery, small enough for each rider to accomplish at least one order, each rider will be assigned to one order.

```
riders = [
    {
        'start': (15.6190,38.2590),
        'id': 'rider-1', #1,3
        'battery':20,
        'order_capacity':5
    },
    {
        'start': (15.6077,38.2559),
        'id': 'rider-2', #2
        'battery':20,
        'order_capacity':12
    },
    {
        'start': (15.6052,38.2677),
        'id': 'rider-3',
        'battery':20,
        'order_capacity': 12
    },
]

orders = [
      {
          'id' : 1,
          'capacity': 2,
          'coords': (15.6090,38.2595)
      },
      {
          'id' : 2,
          'capacity': 7,
          'coords':  (15.6076,38.2647)
      },
      {
          'id': 3,
          'capacity': 3,
          'coords': (15.6105,38.2569)
      }
    ]
```

The above initial data return the following associations:

```
{
    'rider-1': [{'id': 3, 'capacity': 3, 'coords': (15.6105, 38.2569)}],
    'rider-2': [{'id': 1, 'capacity': 2, 'coords': (15.609, 38.2595)}],
    'rider-3': [{'id': 2, 'capacity': 7, 'coords': (15.6076, 38.2647)}]
}
```

Where each rider has been assigned to one different order, depending on the closeness of the starting position, indeed rider-1 still get the third order since is the closest to him.

3.0.2 Fixing the percentage of battery

The second experiment has been done observing the results when the riders have the maximum percentage of battery but the capacity of the order changes. Considering the following initial conditions:

```
riders = [
    {
        'start': (15.6190,38.2590),
        'id': 'rider-1', #1,3
        'battery':100,
        'order_capacity':2
    },
    {
        'start': (15.6077,38.2559),
        'id': 'rider-2', #2
        'battery':100,
        'order_capacity':12
    },
    {
        'start': (15.6052,38.2677),
        'id': 'rider-3',
        'battery':100,
        'order_capacity': 12
    },
]

orders = [
      {
          'id' : 1,
          'capacity': 2,
          'coords': (15.6090,38.2595)
      },
      {
          'id' : 2,
          'capacity': 7,
          'coords':  (15.6076,38.2647)
      },
      {
          'id': 3,
          'capacity': 3,
          'coords': (15.6105,38.2569)
      }
    ]

```

We can observe that rider-1 is able only to satisfy the order-1 due to the his capacity and that rider-2 and rider-3 are able to satisfy both the orders 2 and 3, we expect in this case that riders 2 will satisfy the remaining two orders since he has full battery.

```
{
'rider-1': [{'id': 1, 'capacity': 2, 'coords': (15.609, 38.2595)}],
'rider-2': [
    {'id': 3, 'capacity': 3, 'coords': (15.6105, 38.2569)},
    {'id': 2, 'capacity': 7, 'coords': (15.6076, 38.2647)}
    ],
'rider-3': []
}
```

If we change the capacity of the rider-2 to 9, we expect that he will be able to satisfy only one of the two remaining orders and the last order will be assigned to rider-3:

```python

riders = [
    {
        'start': (15.6190,38.2590),
        'id': 'rider-1', #1,3
        'battery':100,
        'order_capacity':2
    },
    {
        'start': (15.6077,38.2559),
        'id': 'rider-2', #2
        'battery':100,
        'order_capacity':9
    },
    {
        'start': (15.6052,38.2677),
        'id': 'rider-3',
        'battery':100,
        'order_capacity': 12
    },
]

orders = [
      {
          'id' : 1,
          'capacity': 2,
          'coords': (15.6090,38.2595)
      },
      {
          'id' : 2,
          'capacity': 7,
          'coords':  (15.6076,38.2647)
      },
      {
          'id': 3,
          'capacity': 3,
          'coords': (15.6105,38.2569)
      }
    ]
```

Having the result as expected:

```
{
'rider-1': [{'id': 1, 'capacity': 2, 'coords': (15.609, 38.2595)}],
'rider-2': [{'id': 3, 'capacity': 3, 'coords': (15.6105, 38.2569)}],
'rider-3': [{'id': 2, 'capacity': 7, 'coords': (15.6076, 38.2647)}]
}
```

Chapter 4

Conclusions

The aim of this project was to optimize a routing problem related to the delivery food field. We decomposed the problem starting from the data required and then we went forward developing a technique able to enrich data extracting information from the mean slope between two points. We built a cost function useful to approximate the consumption associated to each edge of the graph and we developed a greedy algorithm that takes into account the battery percentage, the capacity of transported orders and the starting position of each rider to associate each order if possible. The results are promising and we showed how the algorithm act with different initial conditions. In the future we could improve the optimization by considering more variables and observing the algorithm on a bigger dataset. Also we showed how is possible to reach very interesting results just using open source technologies.
The complete repository of the project can be found at the link:
https://github.com/MarcoGarofalo94/FoodDeliveryPathOptimization

Bibliography

[1] Online food delivery is usa. https://www.statista.com/outlook/dmo/eservices/online-food-delivery/united-states.

[2] Sweetgreen to pilot first digital-only pickup location in washington d.c. https://www.nrn.com/fast-casual/sweetgreen-pilot-first-digital-only-pickup-location-washington-dc.

[3] Emissioni co2 in italia. https://www.rinnovabili.it/ambiente/cambiamenti-climatici/emissioni-co2-in-italia-2021/.

[4] Le emissioni di gas serra in italia alla fine del secondo periodo del protocollo di kyoto: obiettivi di riduzione ed efficienza energetica. https://www.isprambiente.gov.it/files2022/pubblicazioni/rapporti/rapporto_362_2022-completo-13_04.pdf.

[5] Openstreetmap. https://www.openstreetmap.org/.

[6] Osmnx. https://osmnx.readthedocs.io/en/stable/.

[7] Networkx. https://networkx.org/.

[8] Open-elevation api. https://open-elevation.com/.

[9] Open-elevation api documentation. https://github.com/Jorl17/open-elevation/blob/master/docs/api.md.

[10] vaielettrico.it. https://www.vaielettrico.it/,.

[11] alke.it. https://www.alke.it/calcolo-pendenza-salita-veicolo-elettrico,.

[12] Calculate distance, bearing and more between latitude/longitude points. https://www.movable-type.co.uk/scripts/latlong.html,.