University of Castilla-La Mancha

Universidad de Castilla-La Mancha

Escuela Superior de Informática

Algebra & Discrete Mathematics

Incremental Laboratory Practice 2021-22 (Bilingual Group)

Navigation System

Professors:
José Ángel Martín Baos (joseangel.martin@uclm.es)
José Luis Espinosa Aranda (josel.espinosa@uclm.es)
Juan Romero Cañas (juan.romero2@alu.uclm.es)
Ricardo García Ródenas [Theory] (ricardo.garcia@uclm.es)
José Ángel López Mateos [Theory]

Important things to keep in mind:

1. The practice must be done individually.
2. This practice is evaluated over 2 points of the overall course, which is equivalent to approximately 57% of the laboratory part of the course.
3. It is not necessary to complete all the milestones in order to submit the practice.
4. For the evaluation of each of the sections, both the clarity of the code and the comments included in it will be taken into account.
5. A copy-detection system will be used to correct the practice. When the similarity between two or more practices is out of the permissible range, all of them will be qualified with a 0, which will result in a failure of the course in both, the ordinary and the extraordinary call.
6. Deadline: May 22nd 2022. An assignment will be enabled on the virtual campus to upload the practice.
7. The codes must be delivered compressed in a single .zip file with the student's name. Do not forget to attach all the necessary files for its correct execution, including the files provided in the virtual campus.

Objectives of the practice:

A navigation system is a computer system whose purpose is to assist navigation (whether by land, sea or air vehicles). These navigation systems may be completely installed inside the vehicle they control or they may be located in a different place and make use of radio transmission or any other communication method to send control commands to the vehicle.

Among the possibilities of these navigation systems, satellite navigation devices stand out in everyday use. Modern navigation devices use the signal from the GPS (USA), GLONASS (Russia), BDS (China) or Galileo (Europe) satellite constellation, among others, to determine the location of a vehicle and give directions on the route to follow.

In this practice we will focus on the study and design of a small part of a GPS navigation device. More specifically, we will focus on the problem of representing the street network of a city on a computer and the design of a program to obtain the optimal route that a vehicle should follow to get from point A to point B.

Modern devices have an internet connection, which allows them to know the traffic in a city in real time. For this reason, in this practice we will simulate vehicle traffic between different points in a city. Using this simulated traffic, we will be able to carry out an analysis of interventions in the road network, thus answering questions such as: which street should I cut if we have a construction project, is it convenient to change a street into a two-way street, or is it convenient to reverse the direction of another street? Finally, we will improve the design of the GPS program to obtain an optimal route between two points, taking into account the traffic within the city.

1 Building a city street network

The first step in building a navigation device for a city is to store the city's street network in the device. In order to do this, it is necessary to study how the map of the street network can be represented in the computer system. The most common way of working with these street and road networks is through the use of **graphs**.

Recall that a graph is a mathematical object made up of a set of objects called vertices (or nodes) that are connected by links called edges (or arcs). These edges allow us to represent the binary relations between different elements of the set of vertices. In our case, the edges represent the roads, while the vertices are the intersections, i.e. all the points where it is possible to choose which path to take. Figure 1 shows an example of the representation of a map using a graph. In this figure the red circles represent the vertices and the brown lines represent the edges.

Figure 1: Example of the representation of a map using a graph. (Source: OpenStreetMap)

In this practice we will use as a basis the maps of OpenStreetMap, a collaborative project to create editable and free maps that are distributed as vector data under an open licence. To do this, we can go to the project's web page https://www.openstreetmap.org from where we can download both the image of the desired map (in JPG, PNG, SVG format) and export the map data in OpenStreetMap's own format (OSM). This OSM format contains all the information about the downloaded map (streets, roads, buildings, services, parking, public transport, parks, etc).

From the data exported in OSM format it is possible to filter only the streets and to infer which will be the vertices and which will be the edges in our graph. This is possibly the most complicated part and for this reason it is solved in this practice. However, if the student is interested in knowing how to filter this data, it has been used the Python package OsmToRoadGraph[1]. This package allows, given a map in OSM format as input, to obtain two

[1] https://github.com/AndGem/OsmToRoadGraph

output files: one with the different vertices and edges and another file with the name of the streets along which the edges run (this will be explained later). The exact command used to obtain the data for this practice is:

```
$ python3 run.py -f downloaded-map.osm -n c
```

where `downloaded-map.osm` is the map that has been downloaded in OSM format. **This process has already been done and should not be done by the student** and is detailed only to explain the complete process followed to obtain the map.

Once the above process has been carried out, four files are obtained for each map, which are provided as material for the student. All these files can be found in the folder `/data` of the ZIP file provided together with the statement of this practical assignment in the Virtual Campus. For example, for the map called **ESI**, the files provided are:

1. `ESI.osm`. This is the original OSM file from which all the data necessary for the practice have been extracted. It is provided only for the student's curiosity, and **it is not necessary for the development of the practice**.
2. `ESI.png`. This file contains the above map in PNG image format. It will be used as a background when constructing the graph, thus providing visual context to the drawn streets.
3. `ESI.pycgr`. This file has been obtained from the Python package above and contains the vertices and edges of the graph.
4. `ESI.pycgr_names`. This file contains the name of the street traversed by each edge.

Next, we will explain the structure of the `.pycgr` files. The first 7 lines of these files start with a hash sign (#) and contain a human readable summary of the file structure. The first line without the # sign will be an integer containing the number of vertices (let denote this number as V). The next line contains the number of edges in the file (let denote this value as E). Next we have V lines with the structure:

```
<id> <lat> <lon>
```

where each line represents each of the vertices of the graph and contains 3 numeric values: `<id>` is an integer value containing the vertex identifier; and `<lat>` and `<lon>` are two decimal values (in floating point) containing the latitude and longitude of the vertex, respectively, i.e. its position on the map. Finally, we have E lines with the structure:

```
<source_node_id> <target_node_id> <length> <street_type> <
   max_speed> <bidirectional>
```

where each line represents each of the edges of the graph and contains 6 numeric values: `<source_node_id>` and `<target_node_id>` are integer values containing the source and destination vertex identifiers, respectively; `<length>` is a decimal value containing the length of the edge in metres; `<street_type>` is a road type identifier[2]; `<max_speed>` indicates the maximum speed allowed on the road (in km/h); and `<bidirectional>` is a value that will be 0 if the road is one-way (from origin vertex to destination vertex) or 1 if the road is two-way.

[2] https://wiki.openstreetmap.org/wiki/Key:highway

In the case of the files `.pycgr_names`, they contain E lines in which line e contains the name of the street traversed by edge e.

The function `load_pycgr` is provided, which the student can use to load the above files into MATLAB. This function receives as input parameters the directory in which the data is contained (for example: `'data/'`) and the name of the file containing the map, without extension (for example: `'ESI'` or `'RondaCiudadReal'`). This function will return 4 variables (please check the implementation of this function in the `.m` file): the number of vertices, a structure with all vertices, the number of edges and a structure with all edges.

For example, assuming that the vertices have been stored in a structure called `edges`, if we run:

```
>> edges.length
```

we will obtain a vector with the length of each of the vertices, where for example:

```
>> edges.length(40)
```

will contain the length of the edge 40.

Using these structures and the concepts learned in the graph sessions, we can build an **undirected graph** that we will use to visualise the streets. To do this, the steps are as follows: First build the undirected graph; second create a figure in MATLAB and a *axes* object from this figure; then we will show on the newly created figure the map of the city (in PNG format) using for this purpose the function `showmap` provided in Virtual Campus; finally we will draw the generated graph on top of the map and position each of the vertices using longitude and latitude.

Note that when you plot the graph all the vertices and edges must be in its correct location of the map, if not, possibly you are making a mistake when establishing the coordinates of the vertices.

Note that the street network in a city is actually a directed graph, where the direction of the edges represents the direction of traffic on each of the streets. The construction of the undirected graph is a simplification that will allow us to draw it correctly in MATLAB, however, we must define a second graph (which will be directed) and will be used in the subsequent milestones for the calculation of routes and traffic flows. If we do not do this, we run the risk of sending vehicles down forbidden streets.

To perform the directed graph, we will focus on the `bidirectional` field of the structure containing the edges. If this field has a value of 0, it indicates that the edge is one-way, and the direction of the edge will be from the origin vertex to the destination vertex. Therefore, we could directly create the digraph in MATLAB by specifying the source and destination vertices. However, we also have edges with the value of the `bidirectional` field at 1, which represents two-way streets. Therefore, before creating the digraph, we will have to obtain the two-way edges and for each of them add a new edge in the opposite direction. To do this, we can add new components to the vectors that we have obtained from the edge structure that we loaded from the .pycgr file, in which we will copy the same values as the original edges but interchanging the source and destination vertices.

Note: Both graphs need to be stored: the undirected graph will be used throughout the different milestones to perform the visualisations, while the directed graphs will be used to run all algorithms.

1.1 Milestone 1 (0.25 points)

a) Using the `Hito1.m` file provided as a basis, a MATLAB script called `Hito1.m` must be implemented to generate the street network in a city and represent it on a map. To do this, an undirected graph (call it `G_visual`) and a directed graph (call it `G`) must be created following the steps explained in this section. Finally, the undirected graph will be used to visualise the streets on top of the city map.

b) Using the script `Hito1.m`, display the three maps provided in the folder `data/` (**ESI**, **RondaCiudadReal** y **CiudadReal**).

2 Designing a simple GPS

In Milestone 1 we have seen how the street network of a city can represented mathematically, and how we can visualise it on our computer. Next, we are going to make a simple program that will allow us to calculate the optimal route between two locations on the map.

When we have to choose between two possible paths, we can use several metrics to help us make a decision: we can choose the fastest path to the destination, the shortest, the one with the fewest intersections, the one with the lowest cost (e.g. imagine having to pay a toll for using a certain road), etc. Usually, we use to take into account mostly the road that takes us to the destination first. There are many factors that come into play: the maximum speed, the size of the road, the presence of traffic lights, the amount of traffic, etc. We can summarise them all in a single value, the (estimated) travel time on that section. This information is integrated into the edges of our graph in the form of weights and, consequently, we will have a weighted (or labelled) graph. For example, if a "4" is written on the edge connecting vertex X and vertex Y, this indicates that we estimate it takes 4 minutes to go from intersection X to intersection Y.

Google Maps (and all other GPS tools) is based on a very simple but incredibly efficient algorithm: Dijkstra's algorithm. This algorithm is named after its inventor, Edsger Dijkstra, one of the founding pioneers of modern computing. But what does Dijkstra's algorithm do? Given a weighted graph, a starting vertex and a destination vertex, the algorithm finds the 'minimum path' connecting the two points, i.e. the sequence of edges that minimises the sum of the (positive) weights and thus, in the case of maps, minimises the estimated travel time.

So, going back to our graph, the first step is to add weights to each of these edges. Recall that for each edge we have its length, the type of road, the maximum speed, whether it is a two-way road and its name. Using the distance and maximum speed, we can infer the time that a vehicle would take to travel along the edge under optimal conditions (assuming low traffic density, that the car is travelling at the maximum speed allowed, green traffic lights, no traffic jams, etc.). To do this, we only need to operate with the length and speed vectors. **Note** that the length is expressed in metres and the speed in kilometres per hour. Assuming constant travel at 90% of the maximum speed, **the estimated time for each edge, expressed in minutes, must be calculated**. Once the travel time for each edge has been calculated, we will construct a new digraph (similar to the one constructed in Milestone 1) but using this magnitude as the weight for the edges.

If we look at the structure of the newly created digraph, we can see that it contains two tables, one of edges (`Edges`) and another of vertices (`Nodes`). The table of edges only contains

the origin and destination vertices of each edge and its associated weight (remember that the weight has been defined as the travel time in minutes circulating at 90% of the maximum speed). What we will do now is to add to this table of edges of the digraph some new columns with the values of **length** (`length`), **maxspeed** (`maxspeed`) and **street name** (`name`) from the edges structure loaded from the `.pycgr` file, which will be useful later on. However, we have to be careful because the order in which are listed the edges extracted from the `.pycgr` file is different from the order of the edges in the table contained on the digraph, as MATLAB reorders the edges when creating the digraph.

To solve this, we will use the MATLAB function `findedge(G, source, target)`[3] where, given a source vertex (or a vector of vertices) in a graph (`source`) and a target vertex (`target`), we obtain the index (or vector of indices) of the table in which these edges are located, that is, the row in which the edges from `source` to `target` are located. Using these indexes, we can look up each of the edges loaded from the .pycgr file in the digraph edges table and associate them with the corresponding length, maximum speed and street name values (by creating new columns in the `G.Edges` structure).

Note: As an example, if we want to assign to the edge with index number 5 (i.e. the one in row 5) value `length`$= 2.25$, we will use:

```
1 >> G.Edges.length(5) = 2.25
```

It is also possible to work with vectors instead of individual values (avoiding you to use loops).

Once the graph has been constructed, we only need to locate the index of the source and destination vertices and then apply Dijkstra's[4] algorithm to obtain the minimum path between both vertices. **Be careful**, as we are working with a directed graph, the minimum path will be different if we permute the source and destination vertices.

Once the minimum path is obtained, we will use an **undirected graph** to construct a figure showing the map of the city and its streets (as was done in Milestone 1), **clearly highlighting the origin vertex, the destination vertex and the minimum path followed**.

Finally, the estimated travel time (corresponding to the value of the total weight of the path obtained using Dijkstra) and the approximate distance of this path must be shown by console, for which the distance of each of the edges traversed in this minimum path must be added up.

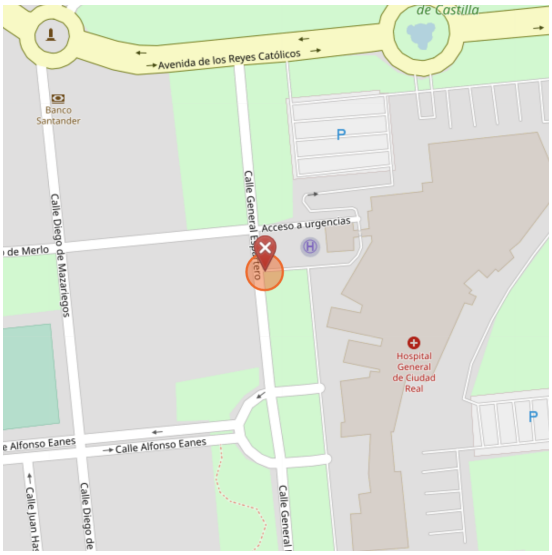
2.1 Milestone 2 (0.5 points)

a) Using the `Hito1.m` file you created earlier as a template, create a copy called `Hito2.m`. This script should search for the optimal path between two points (based on what it is explained in this section) and represent graphically this path. In addition, the distance of the optimal route (in kilometers) and the estimated travel time (in minutes) should be displayed on the command line.

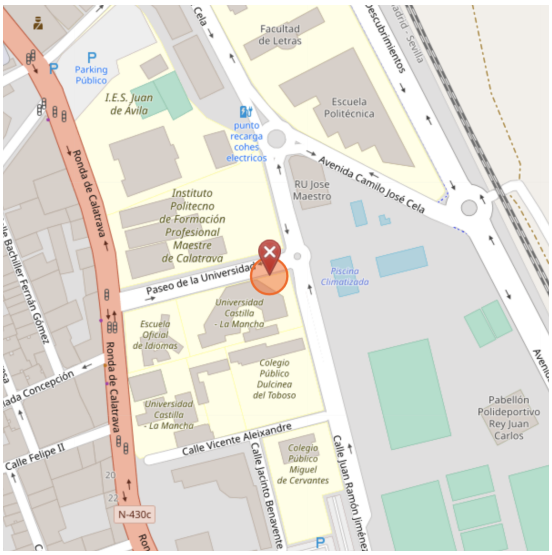
b) The script `Hito2.m` must obtain the optimal route for each of the following pairs of origin and destination points, using the map named **CiudadReal**:

[3] Check: https://es.mathworks.com/help/matlab/ref/graph.findedge.html
[4] Check: https://es.mathworks.com/help/matlab/ref/graph.shortestpath.html

1. Route from the emergency door of the Hospital General Universitario de Ciudad Real to the main entrance of the Escuela Superior de Informática (ESI). See Figures 2a y 2b to help you identify source and destination vertices.

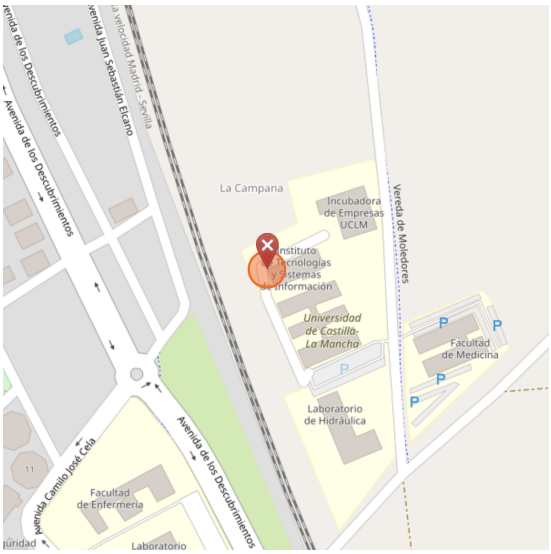


(a) Emergency door of the Hospital

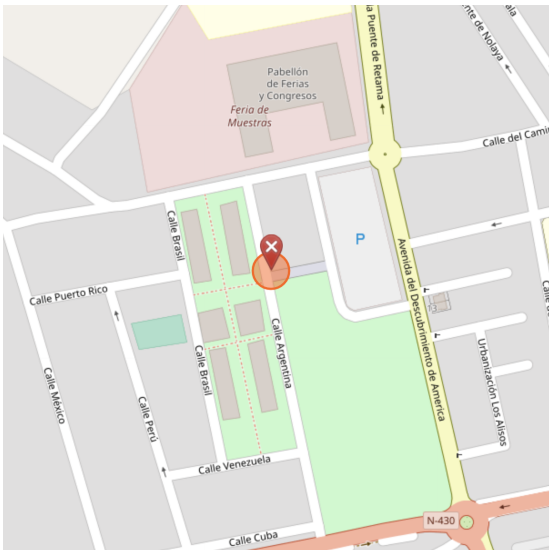


(b) ESI

2. Get the route from the Instituto de Tecnologías y Sistemas de Información (ITSI) to the Auditorio de la Granja. See the Figures 3a y 3b to help you identify source and destination vertices.



(a) ITSI



(b) Auditorio de la Granja

3. Get the reverse route to the previous one, that is, from the Auditorio de la Granja to the ITSI.

Note: Create a new figure for each route. Do not forget to highlight the optimal route, as well as the origin and destination points. Finally, display on the command line the distance and the estimated travel time for each of these routes. You can obtain the index of the origin and destination vertices by hovering the mouse cursor over it them until it appears.

3 Simulating city traffic

At Milestone 2 we have built a simple GPS navigator. Although its operation is the basis of more complex systems such as Google Maps, it is still a simplification of these systems. In this milestone, we will add a new dimension that will add realism to our system.

Additional information, such as traffic information or the presence of traffic jams, can be added by simply modifying the weights of the graph. But how does Google know the traffic information? It uses data collected by other users who are using the application in real time. In fact, every time you use Google Maps, or drive around a city with an Android device, Google is collecting your position, as well as the speed at which you are moving, and anonymously sends this data to its servers. Then Google process this data in real time to determine the congestion at each point in the city. Since we cannot access this information, we are going to simulate the city's traffic.

The level of congestion in a traffic network is the result of the interaction within the road network (supply) and the demand on the network. In traffic planning models, the demand is represented by the so-called *origin-destination (O-D) matrix*. This matrix is constructed in a multi-stage process. In the first stage (generation/attraction) the city is divided in zones and the potential of each of these zones to attract or generate trips is measured. These values will depend on the resident population, the number of jobs located in the area, school places, etc. After evaluating these characteristics, the number of trips that each of these zones is capable of generate and attract is estimated. These zones are abstracted by means of a set of vertices *centroids* that act as sources/sinks that inject/receive traffic flow from the network. In the attached file **ODmatrix.xlsx** this information has been collected for the map **CiudadReal**. The first column contains the id of the vertex acting as centroid, the second column the number of trips that is attracting and in the third column the number of trips it generates.

In the second stage (distribution), the aim is to find out how many trips are made from a given area to another, and in the last stage (modal split), the modes of transport used to make these trips are determined. In this practice we will consider that the origin-destination matrix for trips made by private vehicle are obtained by means of the expression:

$$d_{ij} = \frac{G_i A_j}{D} \text{ con } i \neq j \tag{1}$$

where d_{ij} is the demand from zone i to zone j, G_i is the number of trips generated in zone i, A_j is the number of trips attracted by zone j and D is the total demand, i.e. $D = \sum_i G_i = \sum_j A_j$. Note that the intra-zone trips d_{ii} (trips with same origin as destination) have been eliminated. Expression (1) assumes that trips are being made randomly from one area to another.

The superposition of all the paths used in the network will determine the level of use of the network edges. We call this operation the assignment of the $O-D$ matrix to the network. We assume that **vehicles always choose the minimum path** (with respect to travel time) and that for each pair (i, j) of the $O-D$ matrix this minimum path p_{ij} has been calculated. We define the parameter

$$\delta_{a,p_{ij}} = \begin{cases} 1 & \text{if the edge } a \text{ is in the path } p_{ij} \\ 0 & \text{the opposite case} \end{cases}$$

then the number of vehicles passing along the edge a is expressed by

$$f_a = \sum_{(i,j)} \delta_{a,p_{ij}} d_{ij} \text{ for all edge } a \tag{2}$$

The expression (2) indicates that the traffic flow on edge a is the sum of the flow of all paths passing through a. If the minimum path does not pass through the edge we add 0 but if the path passes through the edge we add its flow d_{ij}. The best way of doing this is to create a new column in the `Edges` field of the digraph G called `Flow` and containing the value f_a.

To visualise the flow of each edge in the map we must copy this `Flow` field from the digraph to the undirected graph we use for the visualisations. However, the order of the `Edges` table **is different** in both graphs (as MATLAB reorders the edges when creating a graph). Using the `findedge` function we can find which edges of the digraph correspond to the edges of the undirected graph, and so copy these values from the `Flow` field. **Be careful** as there will be edges in the undirected graph that correspond to two edges in opposite directions in the digraph (two-way streets). In these cases, the value of the `Flow` field in the undirected graph will be the sum of both edges in the digraph.

3.1 Milestone 3 (0.5 points)

a) Calculate the source-destination matrix d_{ij} using the expression (1) and the attached file *ODmatrix.xlsx*. Remember that the MATLAB instruction to read excel files is *xlsread*.

b) Assign this matrix to the network created using the map **CiudadReal** and calculate the traffic flow on the edges of the network (using the directed graph constructed in the previous milestones).

c) Represent the edges of the network proportionally to the traffic flow they have (the higher the flow, the larger the width of the edge).

Note: To modify the width of the edges in a MATLAB visualisation, you can change the parameter `LineWidth` of the plot[5].

4 Network intervention analysis

Using the concepts of the previous milestones, in this section we are going to use our system to evaluate a series of interventions to the street network of Ciudad Real, analysing the consequences that these would have.

The City Council of Ciudad Real is studying a series of alternatives to reduce the pollution produced by vehicles in the city centre. Three possible scenarios are currently being proposed:

1. Pedestrianise the streets "Calle Elisa Cendreros" and "Calle de la Paloma".
2. Reduce the maximum speed of the entire "Ronda" to 30 Km/h.
3. Change the traffic direction on the street "Calle Toledo".

[5] https://es.mathworks.com/help/matlab/ref/graph.plot.html#buzf5jn

Figure 4 shows the streets affected by each of these scenarios: Scenario 1 (red colour), Scenario 2 (magenta colour), Scenario 3 (green colour).

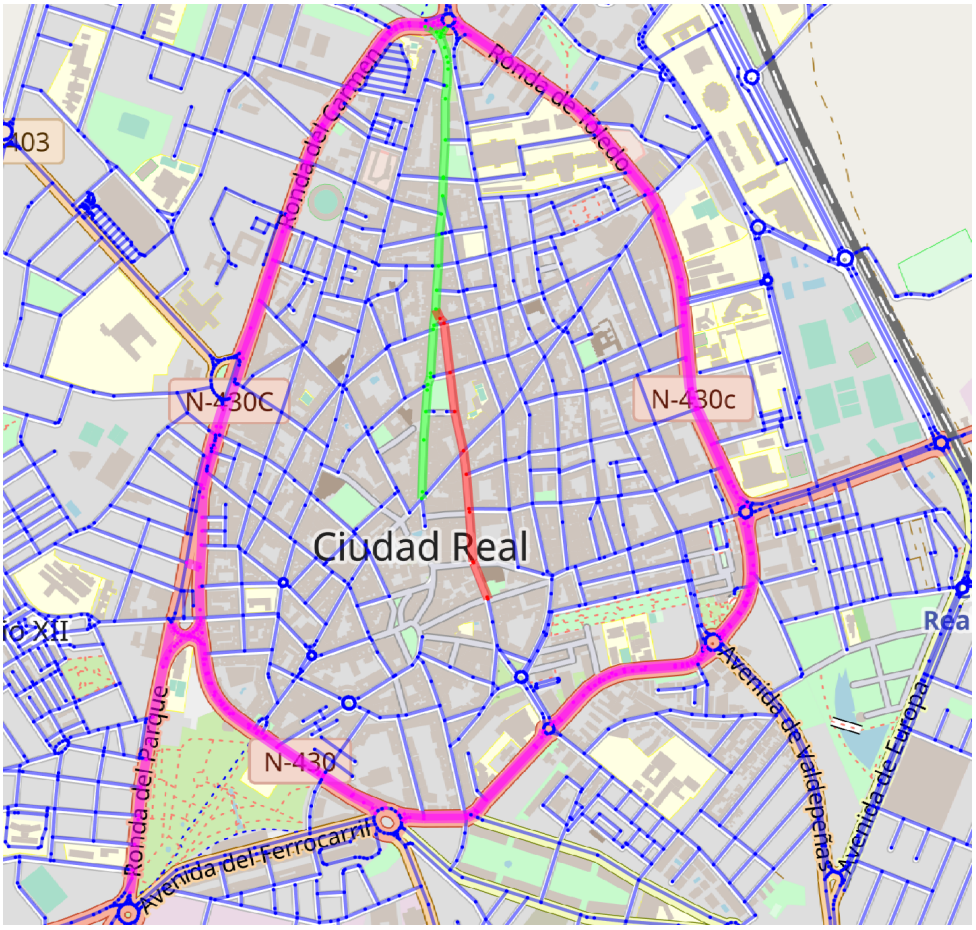


Figure 4: Streets affected by Milestone 4 actions. (Source: OpenStreetMap)

To implement these scenarios, it must be obtained the edges traversed by the affected streets. Then, it must be performed any necessary change to the digraph in order to implement each individual scenario. To obtain the edges of the affected streets, a possible alternative is proposed, which consists of using the field "`name`" with the name of the street that we stored in the edge structure of the digraph in Milestone 2 to search for all the edges that belong to these streets. To locate the streets "Calle Elisa Cendreros", "Calle de la Paloma" and "Calle Toledo" you can use the command `find` and the operator ==. In the case of the *"Ronda"*, as it is composed by several streets (*Rondas*) with different names, the command `startsWith` can be used together with `find` to locate all the streets that begin with the word "Ronda".

To invert the direction of an edge (Scenario 3), one option is to use `flipedge`, although edges can also be modified manually by swapping the source and destination vertices.

Be careful: Create a new digraph for each scenario to be implemented, so that each change does not affect the original digraph or the other digraphs created for each scenario.

4.1 Milestone 4 (0.5 points)

a) Based on the above milestones, create a new script called `Hito4.m` in which you implement each of these new scenarios.

b) The total travel time on the graph is calculated by the expression:

$$T = \sum_a f_a t_a \tag{3}$$

where t_a is the time to cross the edge a (travelling at 90% of maximum speed) and f_a is the traffic flow on the edge a (calculated using the expression 2). The above interventions will change the flow and/or travel time in the network and therefore the total time. Using MATLAB, fill in the following table[6] with the total travel times for the different interventions and the initial situation and display it by console when executing `Hito4.m`. Argue which of the proposed 3 scenarios has the least impact on the network.

Scenario	T
Initial case	-?-
Scenario 1	-?-
Scenario 2	-?-
Scenario 3	-?-

5 Designing an improved GPS

The travel time on the edges depends on the flow that travels through them. As the flow is approaching the maximum capacity of the edge, the edge starts becoming congested, increasing the travel times of the vehicles travelling through it. We introduce the following expression to model this effect and obtain the new travel time of the edge a:

$$t_a(f_a) = t_a^0 * \left(1 + 0.2 * \left(\frac{f_a}{k_a}\right)^4\right) \tag{4}$$

where f_a is the flow of the edge a, t_a^0 is the travel time when edge has no congestion at all and vehicles are travelling at the 90% of the maximum speed (calculated in previous milestones), and k_a is the capacity of the edge. We assume that all edges in the network have a capacity of $k_a = 500$ vehicles/hour.

The effect of congestion causes the appearance of interesting routes that were not originally considered because they were not the minimum routes. Assume that our GPS navigation system is able to estimate the current traffic flows in the city, simulate how the proposed routes obtained in Milestone 2 would change using the congestion calculated on Milestone 3.

5.1 Milestone 5 (0.25 points)

a) Calculate the travel times (edge weights) taking into account the congestion through the expression (4).

b) Create a new script called `Hito5.m` where Milestone 2 is repeated but considering these new travel times. Given the traffic simulation in Milestone 3, would there be a change of route for a new user wishing to take these routes? Which routes are altered and why? Does the travel time of the unaltered routes change? Why?

[6] https://es.mathworks.com/help/matlab/ref/table.html