Universitat Autònoma de Barcelona

OPTIMIZATION

MARC FUSTER & CARLOS MOUGAN

A Star Algorithm

Submitted To:
Lluis Alseda
Optimization Professor
Mathematics Department

Submitted By :
Marc Fuster
Carlos Mougan
MSc in Modelling
CRM

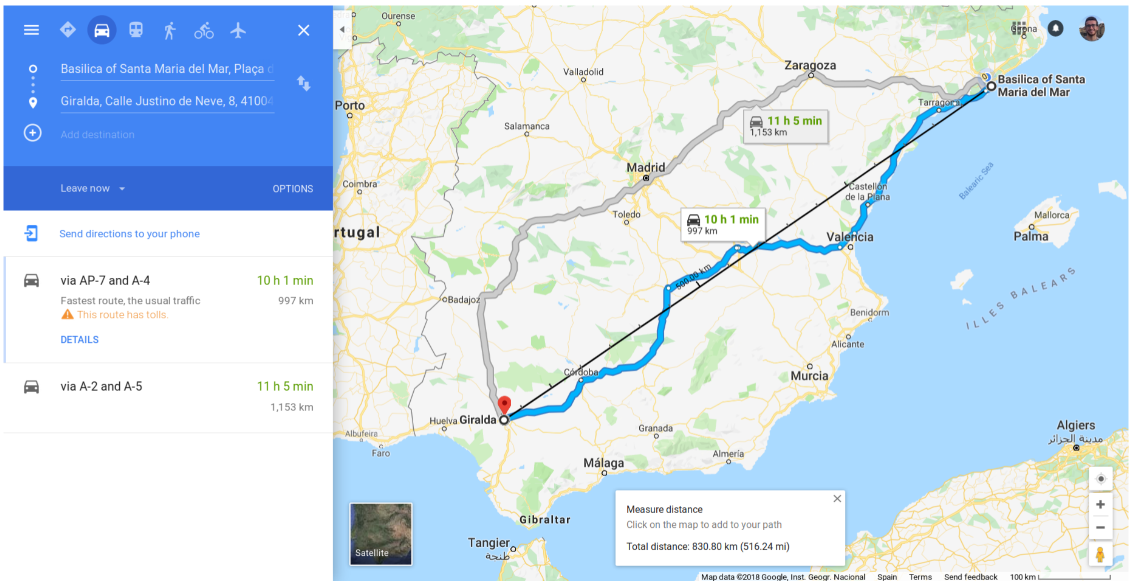
Abstract

The routing problem from Barcelona to Sevilla has been solved using an A* algorithm in C. Different heuristic strategies as haversine distance, spherical law of cosines, rectangular approximation and no heuristic function (Dijkstra) have been used. We have found that there is no difference between the different heuristic functions, so there is no need to research more on different heuristic approaches from this perspective. We also found that the route from Dijkstra is longer and way worse than A* (958Km against 1279km). The execution of A* is relatively fast, it takes around 8 seconds to compute the solution. Moreover, the fact that our solution is shorter than the solution of Google Maps (958Km against 998km) confirms that our results are correct since Google Maps minimizes the trip time and we minimize the trip distance.

Contents

1 Motivation

Our challenge for this project is to find the optimal route in terms of distance from Basílica de Santa Maria del Mar in Barcelona to the Giralda in Sevilla.


Basilica of Santa Maria del Mar, Plaça d

Giralda, Calle Justino de Neve, 8, 41004

Add destination

Leave now

OPTIONS

Send directions to your phone

via AP-7 and A-4 — 10 h 1 min — 997 km

Fastest route, the usual traffic

This route has tolls.

DETAILS

via A-2 and A-5 — 11 h 5 min — 1,153 km

Measure distance

Click on the map to add to your path

Total distance: 830.80 km (516.24 mi)

Figure 1: Google Maps route from Barcelona to Sevilla

In *Figure 1*, we can see that the fastest route that Google Maps finds is 998km and proposes other routes within the range of 1095-1155km. We will try to find the shortest way since we will only take distance into consideration. Our aim is to minimize the travel distance and not the travel time. The distance in a straight line is 830 km, so that is the best possible physical distance.

We will achieve this by using an **A* algorithm** [2]. These algorithms are designed to solve routine problems as: what is best, or close to best, a path to go from point A to point B.

2 Project Description

We are given a data map of Spain that has been extracted from the OpenStreetMap1 API in .csv format. This API identifies different kinds of reference points as churches, streets, roads, cultural places, etc, and assigns them a node identifier which provides its longitude and latitude. In reference to our problem, we find the road nodes with the connections among them. We will suppose the following:

- There is always a straight line between two points which will not cross any sea or, at first, any mountain
- That the Spain map has no altitude. Due to the fact that the map in the CSV file does not contain information about the altitude of the points, our algorithm can not take into consideration this variable. Of course, this is a simplification of the model, because there is an important difference between two paths that have the same distance. However, the difficulties to implement the altitude (scraping the information, implementation in A*, ...) have confirmed us the importance to neglect the altitude.

For our problem, the **Nodes** (N = 23.895.681) represent the different locations available in the dataset we are given. Characterized by a name, an integer id, its latitude (ϕ), its longitude (λ) and its successor's nodes. **Edges** (E = 1.417.363) can be bidirectional, which means that we can drive both ways between two nodes or unidirectional where you can just go from one node to the other.

2.1 A* Description

A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (defined function)[6].

We say that A* is both **complete** (finds a path if one exists) and **optimal** (always finds the shortest path) [3] if you use an Admissible (it never overestimates the cost of reaching the goal) heuristic function.

For every iteration of the main loop, A* needs to determine which of its partial path to expand [7]. It does it based on an estimate of the cost still to go to the goal node. The algorithm selects the path that minimizes the function of equation 1. Where $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal

$$f(n) = g(n) + h(n) \tag{1}$$

2.1.1 Heuristic Functions

Heuristic, is an approach to problem-solving, learning, or discovery that employs a practical method, not guaranteed to be optimal, perfect, logical, or rational, but instead sufficient for reaching an immediate goal.

An important condition for a heuristic function to be used in A* star algorithm is that this function must always underestimate the real distance[5]. This requirement is due to the fact that if the heuristic function overestimates the real distance, one cannot assure convergence. This is due to the fact that when you are close to the final node, if the heuristic function is larger than the real distance, A* will not reach the final node. To sum up, an ideal heuristic function should never overestimate the distance but being as close as possible to the real distance.

The trade-off criteria for deciding whether to use a heuristic for solving a given problem is optimal, completeness, accuracy and execution time. In this project, we will compare the trade-off of different heuristics functions.

Haversine distance

The Haversine distance (eq 2) allows us to calculate the difference between two points given its latitudes and longitudes. Where

$$d = 2r \cdot \arcsin \sqrt{\sin^2(\frac{\phi_2 - \phi_1}{2}) + \cos(\phi_1)\cos(\phi_2)\sin^2(\frac{\lambda_2 - \lambda_1}{2})} \tag{2}$$

d is the Haversine distance between two points.

r is the Earth radius that we will consider spherical and approximate it to 6.371 km.

ϕ_n Will be the latitude of the n points

λ_n Will be the longitude of the n points

Spherical Law of Cosines

As an alternative for Haversine distance, the simple spherical law of cosines formula (cos c = cos a cos b + sin a sin b cos C) gives well-conditioned results down to distances as small as a few meters on the earth's surface. The notation is the same than in Haversine distance.

$$d = r \cdot \arccos(\sin(\phi_1)\sin(\phi_2) + \cos(\phi_1)\cos(\phi_2)\cos(\Delta\lambda)) \tag{3}$$

Rectangular approximation

For small distance, we can approximate the earth to flat, and use Pythagoras theorem.

$$d = R\sqrt{x^2 + y^2} \tag{4}$$

In this equation R is the radius of the earth and $x = \Delta\lambda \cos\phi_m$ and $y = \Delta\phi$

Null Heuristic Funcion

If we consider the heuristic function as constant and equal to zero,we have the Djikstra algorithm just by itself.

$$f(n) = g(n) + h(n) \tag{5}$$

2.1.2 Algorithm pseudocode

Now, let's focus on the A* star algorithm. First of all, we define two lists of nodes which will be used:

- **CLOSED**: Contains the nodes that have been visited and expanded (successors have been explored already and included in the open list, if this was the case).
- **OPEN**: Contains the nodes that have been visited but not expanded (successors have not been explored yet).

```
Put node_start in the OPEN list with f (node_start) = h(node_start) (initialization)
while the OPEN list is not empty {
    Take from the OPEN list the node node_current with the lowest
        f (node_current) = g(node_current) + h(node_current)
    if node_current is node_goal we have found the solution; break
    Generate each state node_successor that come after node_current
7       for each node_successor of node_current {
8           Set successor_current_cost = g(node_current) + w(node_current, node_successor)
9           if node_successor is in the OPEN list {
10              if g(node_successor) $\leq$ successor_current_cost continue (to line 20 )
11          } else if node_successor is in the CLOSED list {
12              if g(node_successor) $\leq$ successor_current_cost continue (to line 20 )
13              Move node_successor from the CLOSE list to the OPEN list
14          } else {
15              Add node_successor to the OPEN list
16              Set h(node_successor) to be the heuristic distance to node_goal
17          }
18          Set g(node_successor) = successor_current_cost
19          Set the parent of node_successor to node_current
20      }
21      Add node_current to the CLOSED list
22  }
23  if (node_current != node_goal) exit with error (the OPEN list is empty)
```

2.2 Code description

The code has been divided in two parts: the preprocessing and the main algorithm. Those parts are represented in “Preprocessing.c” and “Main.c” and moreover the functions which are used in “Main.c” are located in “functions.c”

2.2.1 Why we need two different scripts?

This division of the project in two parts is due to the complexity of reading a file which contains a lot of data (1.4 GB) of different structure. With this division, the preprocessing code reads the file and writes the nodes registered with their properties and connections in a binary data file. In other words, the preprocessing code builds the graph. The graph is written as a binary field because reading from binary is faster than reading normal text. Moreover, it is just required to build the graph once because once is saved, it can be used from the Main.c as many times as required. As we will see, the Preprocessing.c executions require more or less 25 seconds while the Main.c requires around 8 seconds. Therefore, separating our project in two codes speeds up the overall project.

2.2.2 Preprocessing.c

This script starts by creating some structures required to build and store the items of the graph:

```
typedef struct {
    unsigned long id;  //node identification
    char *name;
    double lat, lon;  // node position
    unsigned short nsucc; //number of node successors
    unsigned long *successors;
}node;

// Stores both the index and the id
typedef struct{
    unsigned long id;
    int index;
}box;
```

These structures are used to store the graph information. Afterwards, the script contains two adaptations of the reading and writing functions that have been obtained from [4]. Once the most important functions have been introduced, let's start with the main part. The main starts with an initialization of the structure "nodes". Afterwards, it reads the "spain.csv" file and finally, it writes the information in the binary file.

2.2.3 Main.c

As said before, the functions of the script "Main.c" are located in "functions.c". The functions used are: BinarySearch (returns the node index from the node id in an optimal algorithm), `A_star` (which computes the whole A* algorithm described in section 2.1.2) and some functions to add and delete nodes from the OPEN and CLOSED list. Moreover, on "functions.h" we have declared 3 types of structures: The first one is "node" and is exactly the same as in "Preprocessing.c". The second structures is "nodes_complete" and it is an extension of the structure "node" because it contains the same information plus:

```
struct node *prev, *next,*parent; // next and prev for the open
//list structure. parent to know the final path
char list; // 0: nothing, 1: open, 2: closed
double heuristic; // heuristic distance+ g distance
double g; // g distance: from source to this node
```

This structure contains all the information required to run the A* algorithm. Finally, the third structure is called "ourlist" and is the structure used for the OPEN list.

Once the functions used have been introduced, let's describe the main script. It starts with the declaration and initialization of variables such as the goal node, the start node, variables to compute the elapsed time, other auxiliary variables and the nodes structure. Afterwards, the A_star function is executed until the path is obtained. Finally, the path is printed both in the terminal and in a file.

2.2.4 Visualization script

In order to better understand the result that we have obtained, we have decided to plot our result on a map. We have decided to plot it on Google Maps using the simple Python library "gmplot". This library only requires two lists (one of the latitudes and the other of longitudes) of the points to be plotted. Therefore, in the file "visualization.py" that we attach, we first read the "output.txt" and we extract the latitudes and longitudes. Afterwards, we define the center of the map and an altitude from where to see it. In our case, we have set the center of the map in Albacete because it is more or less in the middle of the route.

The typical output is ...

2.2.5 How to run the code

2.2.6 Preprocessing

Compilation:

```
gcc -g -Wall -o Preprocessing Preprocessing.c -lm
```

Execution:

```
./Preprocessing
```

2.2.7 Main

Compilation:

```
gcc -g -Wall -o Main Main.c functions.c -lm
```

Execution:

```
./Main
```

2.2.8 Visualization.py

Since Python is an interpreted language, we only require to execute the code. It can be executed from any IDLE or much easier from the command line. In order to select the txt to be printed, we should modify line 19. By default, it executes 'output_a_star.txt'. The code is based on the gmaps Python package [1].

3 Results

After researching, developing and testing the algorithm explained in the last sections. We were able to achieve the following results.

3.1 Methods

In the heuristic functions implementation we choose 3 different methods: **haversine, equirectangular & spherical law of cosines** and we run the algorithm without any heuristic function (Dijkstra).

For our surprise, we found out that the three heuristic functions select the same route while Dijkstra takes longer (as expected). In order to calculate the total distance traveled we calculate the Haversine, equirectangular and Spherical law of cosines distance and average them.

The path that A* chooses is indeed shorter than the one from Google maps, this actually makes sense since we are optimizing distance and they optimize travel time. We can appreciate that in, per example: Albacete, our path goes through there while Google doesn't. The reason is that Google takes into account traffic, speed & stoplights that make the route more time-consuming. Maps minimizes time whereas we minimize distance.

3.2 Performance

The computational cost of the different strategies is more or less equal except for the Dijkstra algorithm. The total computational cost is divided into reading from the binary file and the computation of the A* algorithm. Since all the reading from the

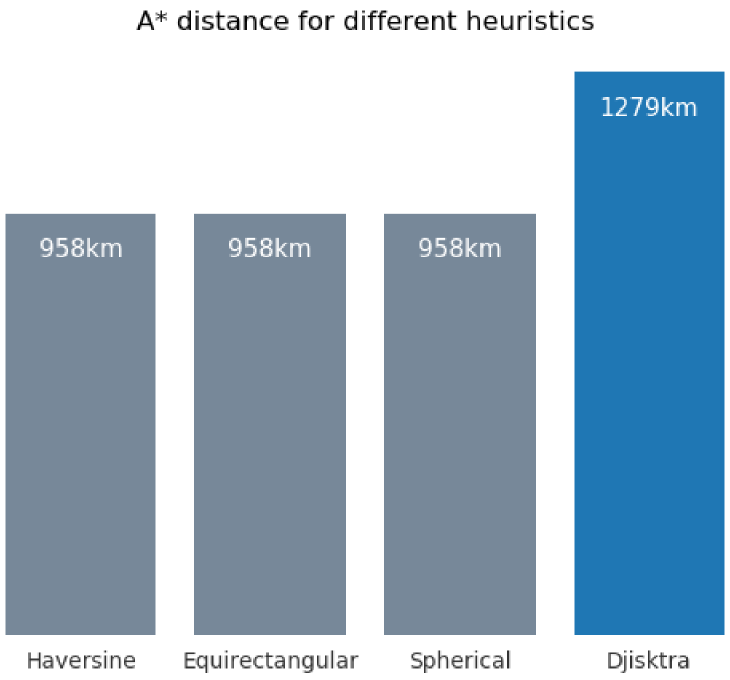


Figure 2: Distance of the path taken by the A* with different heuristic functions

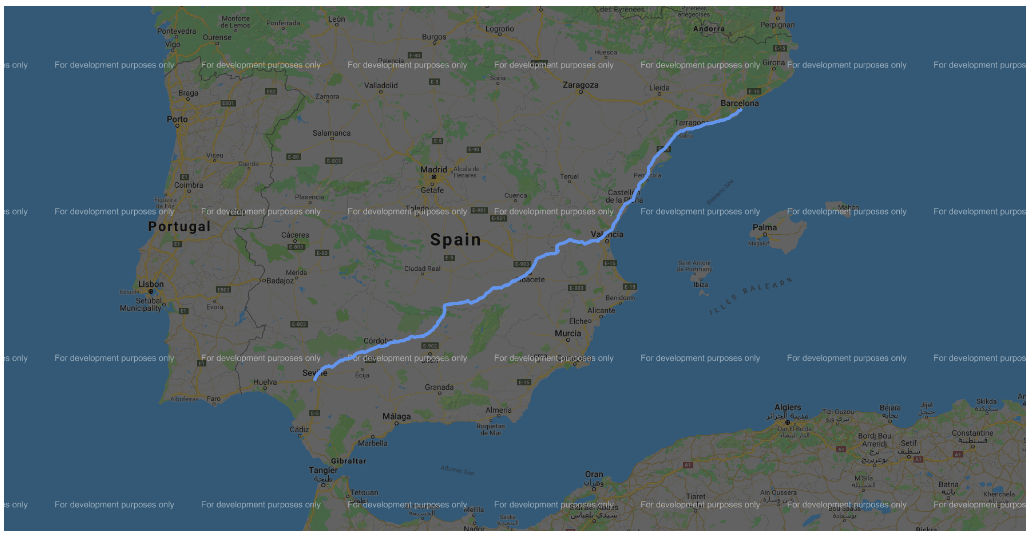


Figure 3: Route taken by the A* algorithm

binary file spends 0.68 seconds, we will only focus on the computational time used in the A* algorithm. The computational times are presented in the following figure: While Haversine, Equirectangular and Spherical spend more or less around 6 seconds,

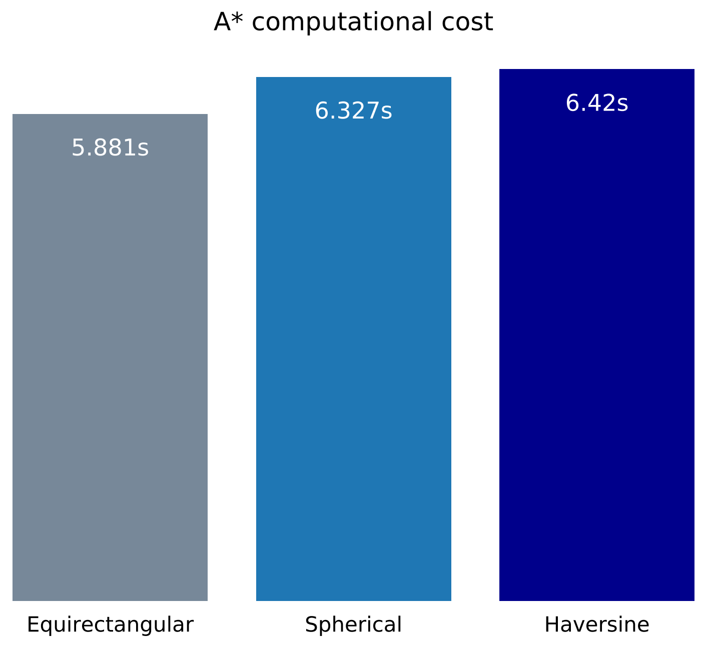


Figure 4: Time spent of the algorithm for different heuristic functions

the Dijkstra algorithm spends 849 seconds, 140x times more, in order to keep the bar plot visual we have decided not to plot it. This result is not surprising since Dijkstra visits much more nodes because it does not have the intuition of the heuristic function. Focusing on the three best methods, the differences in computational cost are due to the fact that some models are faster to compute than others. Haversine is the most expensive and Equirectangular the less expensive and it can be easily understood by the complexity of equations 2-4.

3.3 Curiosities

3.3.1 Distribution of steps distance

After reviewing some of the results we found that the largest distance traveled for A* is between two nodes close to Albacete, the distance that separates them its 5km.

We also found that sometimes the distance between nodes is 0.00 meters, that means

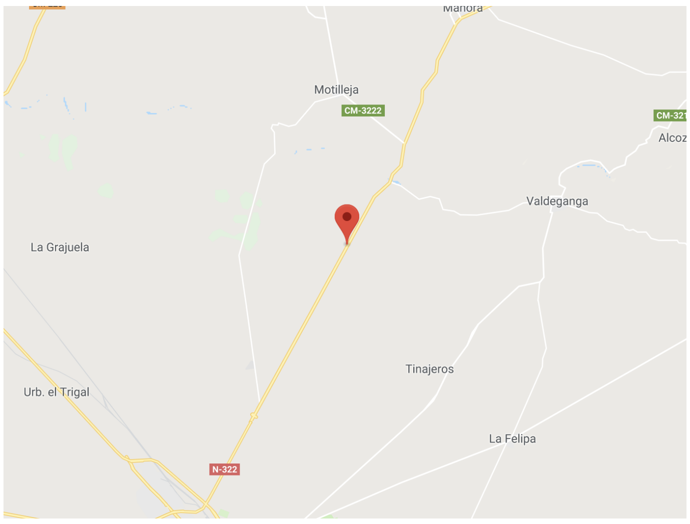


Figure 5: Maximum distance between nodes for A* in a road close to Albacete

that nodes are repeated, or that there are two 'different' nodes with the same latitude and longitude.

From this information we can extract that nodes are really close together sometimes even overlapping. The average distance between nodes is 144 meters.

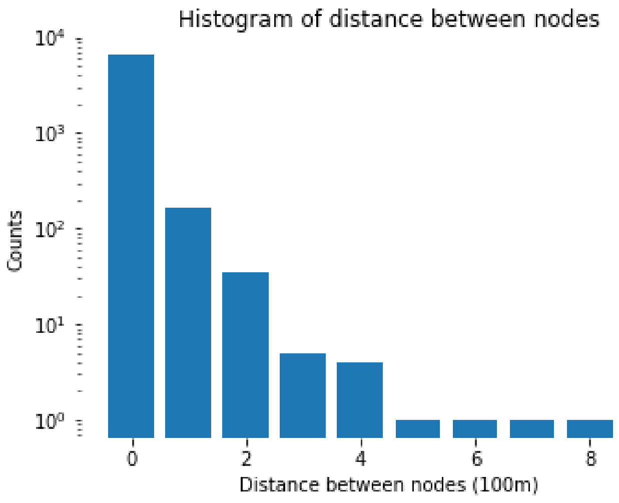


Figure 6: Histogram of the distance between the chosen nodes

3.3.2 Maritime routes

Another interest fact that we found when plotting the route that Djikstra took, is that it goes through the sea. This means that in the Spanish map of nodes, there are nodes connected through maritime transport as we can see on the following image.

This actually makes us aware of our first assumption. If we would have computed any distance with the sea in between, the A* algorithm would have to take the shortest path regardless if it had a drivable road.

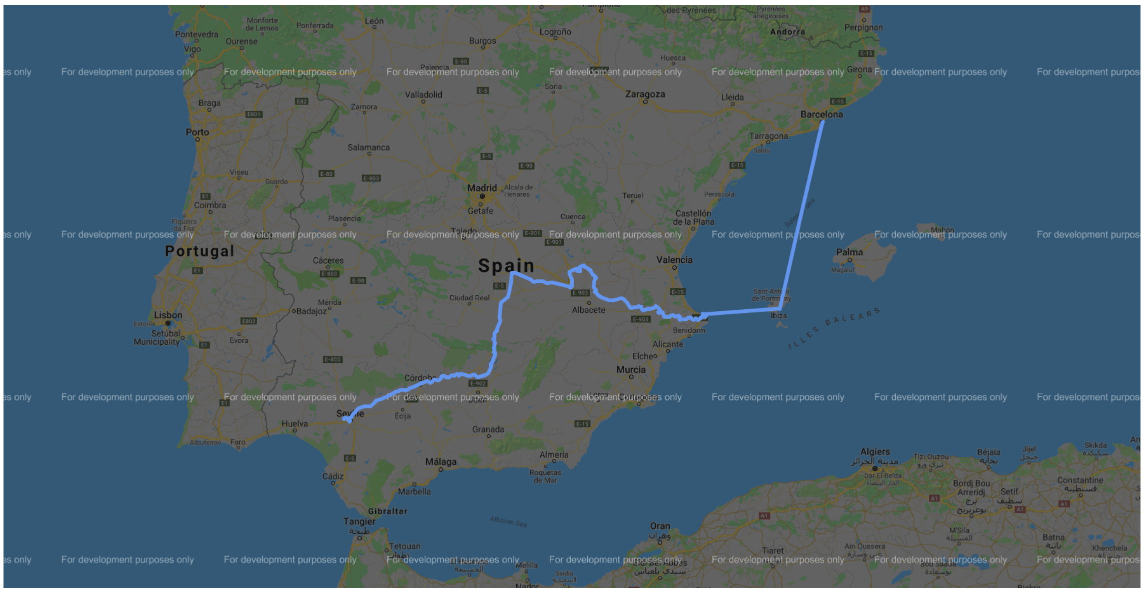


Figure 7: Maximum distance between nodes for A*

Those maritime routes are not commercial routes, they are available for regular passengers and the route followed is Barcelona-Eivissa and Eivissa-Dènia.

4 Conclusions

For this project, we have developed an A* algorithm in C code that is able to solve route problems. In this particular case, we have found a route from Barcelona to Sevilla. We have used different heuristic strategies as haversine distance, spherical law of cosines, rectangular approximation and no heuristic function (Dijkstra).

The code used is divided into two scripts. The first one which is the Preprocessing algorithm is devoted to extract the information about the CSV file and build a graph in a binary file so as to increase the speed. Afterwards, it comes the main code, where we compute the A* algorithm. The execution of A* is relatively fast, it takes around 8 seconds to compute the solution.

We found there is no difference between the different heuristic functions, so there is no need to research more on different heuristic approaches from this perspective. We also found that the route from Dijkstra is longer and way worse than A*. The fact that our solution is shorter than the solution of Google Maps (958Km against 998km) confirms that our results are correct since Google Maps minimizes the trip time and we minimize the trip distance.

References

[1] Gmap package python. https://github.com/googlemaps/google-maps-services-python.

[2] A star algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm.

[3] CLRS 24.3. Lecture 9: Dijkstra's shortest path algorithm. http://mat.uab.cat/~alseda/MasterOpt/MyL09.pdf. Accessed: 01-12-2018.

[4] Lluís Alsedà. Routing problem: Barcelona-seville. http://mat.uab.cat/~alseda/MasterOpt/index.html. Accessed: 01-12-2018.

[5] Jeff Erickson. Lecture 21: Shortest paths. http://www.cs.uiuc.edu/~jeffe/teaching/algorithms. Copyright 2014 Jeff Erickson.

[6] Wikipedia. Graph theory. https://en.wikipedia.org/wiki/Graph_theory.

[7] Wikipedia. Tree traversal. https://en.wikipedia.org/wiki/Tree_traversal.