Solving a Routing Problem with the A* Algorithm Using Different Heuristic Distances

Martí Municoy and Daniel Salgado

Master's degree in Modelling for Science and Engineering
Optimisation, 2017/2018.

Abstract

In this report, we briefly describe the A* algorithm to implement and solve a routing problem given a graph, to find the optimal paths with respect to distance that join two given nodes. We study different strategies that are obtained by varying the heuristic distance and the distance that determines the weight of the graph edges, and compare them in terms of performance and possible path differences. We have focused on locations from Spain, given the data available.

Moreover, we introduce a program which makes use of the A* algorithm to solve general routing problems. We have added the possibility of choosing the locations by entering the latitude and longitude coordinates of the starting and goal locations. Then, the program finds the nearest existing nodes to these pairs of coordinates and computes the optimal path between them, if there exists any.

Our A* implementation in C language has been proved to be a reliable tool to solve a general routing problem to find paths that minimize distance, assuming that the data sets for the regions of interest are available and are formatted in a proper way. In addition, we have added a script in Python that is able to represent the paths graphically as it is done by Google Maps. The sources of all our work can be found in our GitHub repository under a GNU License [1].

UAB
Universitat Autònoma de Barcelona

CRM
CENTRE DE RECERCA MATEMÀTICA

Contents

1 Overview

In this project, we implement the so-called A* algorithm in order to compute an optimal path (according to distance) from Basílica de Santa Maria del Mar (Plaça de Santa Maria) in Barcelona to the Giralda (Calle Mateos Gago) in Sevilla. We take these two start and goal locations, but the algorithm implementation can be used to compute paths from arbitrary locations if they are contained in the available datasets [1]. All our work is available in our GitHub repository [1].

As the reference starting node for Basílica de Santa Maria del Mar (Plaça de Santa Maria) in Barcelona we will take the node with key (@id): 240949599 while the goal node close to Giralda (Calle Mateos Gago) in Sevilla will be the node with key (@id): 195977239. In Figure 1.1 we can visualize the starting point in Barcelona and the goal location in Sevilla we already mentioned, and also the paths suggested by Google Maps. In blue we have the optimal path with respect to time from the Google's algorithm.

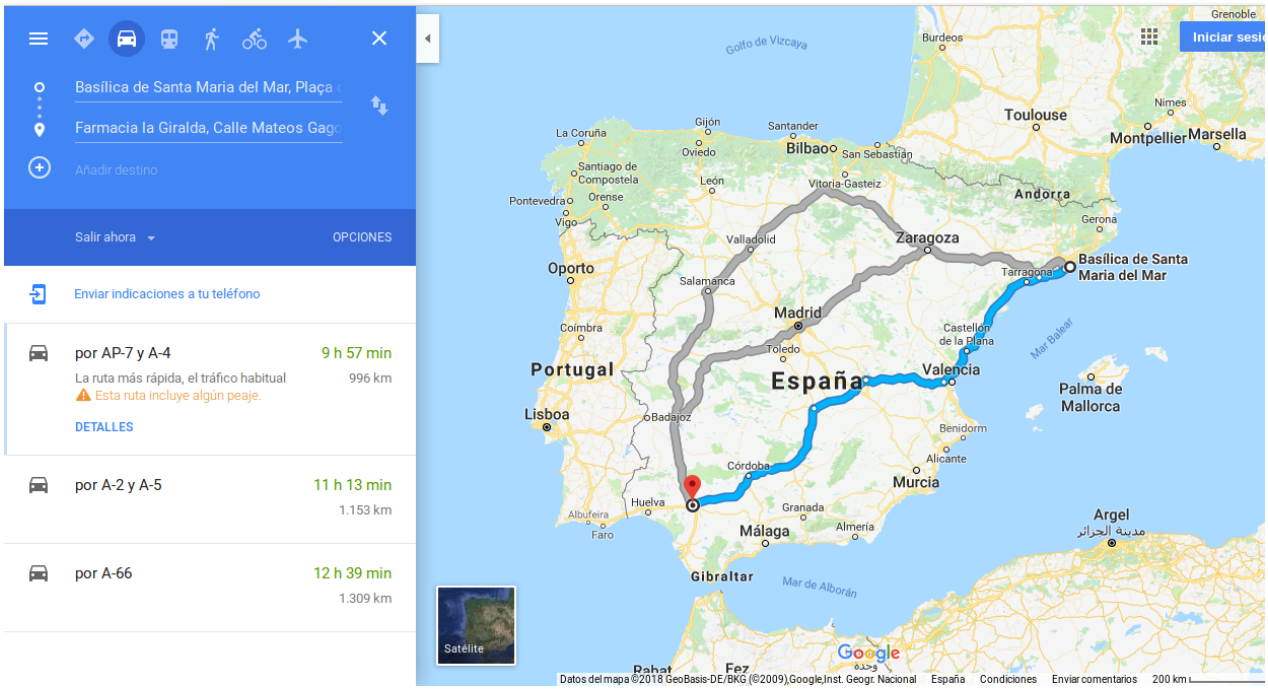


Figure 1.1: Paths suggested by Google Maps to go from Basílica de Santa Maria del Mar in Barcelona to Farmacia la Giralda (Calle Mateos Gago, 14, 41004) in Sevilla. The shortest path suggested is about 996 km long. The blue path is, according to Google, the optimal one (with respect to time).

Our goal is to find a path (or paths) similar to those from Figure 1.1 using the A* algorithm but such that they minimize the distance.

[1] This is an assignment from the Optimisation subject at Master's in modelling for Science and Engineering in the UAB. Professor Lluís Alsedà, 2017/2018. The datasets required can be downloaded from `http://mat.uab.cat/~alseda/MasterOpt/index.html`.

2 A* search algorithm

Many real world problems of importance are related to the general problem of finding a path through a graph satisfying some conditions such as optimality with respect to some property. To illustrate this idea, consider the directed graph from Figure 2.1.

In our context, the `nodes` (circles) represent locations available in the datasets we work with, and may represent particular points of streets. For us, nodes are characterized by a `name`, a integer `id`, its latitude φ, its longitude λ and its `successors` nodes, which are nodes that are reachable from the node in consideration (there is an arrow that points from the current node to each of its successors). Arrows (also called edges) can be bidirectional, which in reality means that there is a two-way street that connects two locations of the map (two nodes); or unidirectional, i.e., a one-way street.

Referring to the graph in Figure 2.1, the node with `id` $= 2$ is connected with the node `id` $= 3$ by a two-way street; whereas it is connected by a one-way street with its successor with `id` $= 4$. Moreover, it is the successor of the nodes with `id` $\in \{1, 3, 8, 9, 10\}$. The integer labels that are near each of the edges are the so-called weights w, which in our problem context it would be the distance that separates the two nodes joined by the edge. For instance, the distance to go from node 2 to node 3 is of 5 units, whereas the distance to go from 2 to 4 it is 10 units.

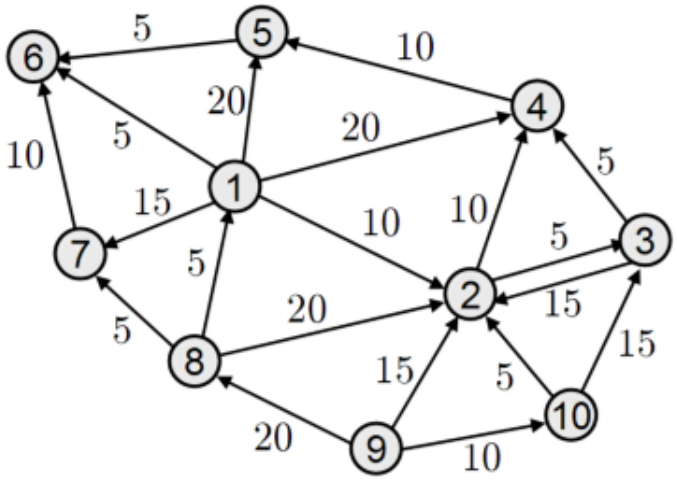


Figure 2.1: Directed graph example to illustrate the representation of a real map, where nodes represent locations that separate streets (for instance); arrows indicate the directions allowed to travel from a node to another; and the edge labels represent the weights or distance units that separate adjacent nodes.

Once we have understood this toy example, we can think of our problem graph as a graph that contains $N = 23.895.681$ nodes and $E = 1.417.363$ edges (ways) from many Spain's locations. Nodes are characterized by a tuple $(\texttt{id}, \texttt{name}, \varphi, \lambda, \texttt{successors})$ and edges are characterized by a tuple of the form $(A, B, \texttt{way\_type})$, where A and B are the nodes that are connected by the edge, and $\texttt{type} \in \{\text{one-way, two-way}\}$ determines if the connection is bidirectional or unidirectional (i.e., if $A \longleftrightarrow B$ or $A \longrightarrow B$, respectively).

2.1 Algorithm description

The A* ("A star") algorithm is an informed search algorithm. This means that it solves problems by searching among all possible paths from a starting point to the solution (goal point) for the one that incurs the smallest cost (least distance travelled, shortest time, or in general a weighted property w), and among these paths it first considers the ones that appear to lead most quickly to the solution. *It is formulated in terms of weighted graphs: starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node* [2].

Before presenting the A* algorithm pseudo-code 1, we introduce some notation. We make use of some concepts that we have not introduced here so we refer to [2] and [4] (Chapter 2) for further clarifications.

At each iteration of its main loop, A needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes* [2]

$$f(n) = g(n) + h(n) \tag{2.1}$$

where

- n denotes the last node on the path,
- $g(n)$ is the cost (total weight) of the path from the start node, we call `start_node`, to node n,
- and $h(n)$ is a heuristic (distance) function that estimates the cost of the cheapest path from n to the goal node, we call it `goal_node`.

The heuristic is problem-specific, and in our case we will propose different alternatives for h in section 2.2.

There is a very important condition that has to be satisfied in order to guarantee that the A* algorithm returns always an optimal path whenever such a path exists. This condition is given the so-called **admissibility** of the heuristic function: if h is admissible then the associated A* algorithm is also admissible [2]. The idea is the following: if $h^*(n)$ denotes the optimal cost to reach a goal from node n then

$$h \text{ is admissible if and only if for all nodes } n \text{ of the graph, we have that } h(n) \leq h^*(n) \tag{2.2}$$

Now lets focus on the A* star algorithm 1. First of all, two lists of nodes are defined to be present during the algorithmic procedure:

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

[2] For a more detailed theory about the properties of the A* algorithm see [4].

Algorithm 1 A* algorithm

1: **procedure** `AStar` (`start_node`, `goal_node`)
2: **OPEN**, **CLOSED** $\leftarrow \emptyset$
3: Put `start_node` in the **OPEN** list with $f(\texttt{start\_node}) = h(\texttt{start\_node})$ ▷ (Initialization)
4: **while** **OPEN** $\neq \emptyset$ **do**
5: Take from the open list the node `current_node` with lowest

$$f(\texttt{current\_node}) = g(\texttt{current\_node}) + h(\texttt{current\_node})$$

6: **if** `current_node` = `goal_node` **then break** ▷ We have found the solution.
7: **end if**
8:
9: **for** each `successor` of `current_node` **do**
10:
11: Set $\texttt{successor\_current\_cost} = g(\texttt{current\_node}) + w(\texttt{current\_node}, \texttt{successor})$
12: **if** `successor` $\in$ **OPEN then**
13: **if** g(`successor`) $\leq$ `successor_current_cost` **then continue** ▷ to line 25
14: **else if** `successor` $\in$ **CLOSED then**
15: **if** $g(\texttt{successor}) \leq$ `successor_current_cost` **then continue** ▷ to line 25
16: Move `successor` from **CLOSED** to **OPEN**.
17: **else**
18: Add `successor` to the **OPEN** list.
19: Set $h(\texttt{successor})$ to be the heuristic distance to `goal_node`.
20: **end if**
21:
22: Set $g(\texttt{successor}) = \texttt{successor\_current\_cost}$
23: Set the parent of `successor` to be `current_node`.
24: **end for**
25:
26: Add `current_node` to the **CLOSED** list.
27: **end while**
28:
29: **if** `current_node` $\neq$ `goal_node` **then** exit with error (the **OPEN** list is empty).
30: **else** An optimal path has been found
31: **end procedure**

If the algorithm ends the while loop and the current node is exactly the goal node, this means that a optimal path have been found.

2.2 Heuristic and weight distances

Since the information available for our nodes are pairs (φ, λ) of latitudes and longitudes, we are interested in using distances to quantify the weight of an edge that depend on these parameters. In the same way, we are going to use as heuristic function, some distance function to estimate the distance between a given node and the goal node. We are going to consider the three alternatives for distances between two nodes given by the latitudes and longitude pairs (φ_1, λ_1), (φ_2, λ_2), which are proposed in [5]. We do not analyse in detail their admissibility (or non-admissibility); as discussed in [8], in practice, for reasonable heuristics such as the ones we present, the A* algorithms finds optimal paths. We will see in section 4 that our implementation succeeds for many combinations of h and w.

Haversine distance

The Haversine distance derives from the so-called Haversine formula [6], and allow us to calculate the great-circle distance between two points, i.e., the shortest distance over the earth's surface, given their longitudes and latitudes:

$$d_H = 2R \cdot \arctan\left(\frac{\sqrt{a(\varphi_{1,2}, \lambda_{1,2})}}{\sqrt{1 - a(\varphi_{1,2}, \lambda_{1,2})}}\right), \qquad a(\varphi_{1,2}, \lambda_{1,2}) = \sin(\Delta\varphi/2) + \cos\varphi_1 \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2) \tag{2.3}$$

where $\Delta\varphi = \varphi_2 - \varphi_1$, $\Delta\lambda = \lambda_2 - \lambda_1$ and R is a radius which in principle would be the mean Earth radius. In addition, we propose to try to use a radius R dependent on the latitudes of the nodes by means of the geocentric radius formula:

$$R(\varphi) = \sqrt{\frac{(A^2 \cos^2\varphi) + (B^2 \sin\varphi)}{(A\cos\varphi)^2 + (B\sin\varphi)^2}} \tag{2.4}$$

where A is the mean equatorial radius and B the polar mean radius of the Earth. Then, in equation (2.3) we choose

$$R := \frac{R(\varphi_1) + R(\varphi_2)}{2} \tag{2.5}$$

Spherical law of cosines

An alternative for the Haversine formula,

$$d_C = R \cdot \arccos\left(\sin\varphi_1 \sin\varphi_2 + \cos\varphi_1 \cos\varphi_2 \cos\Delta\lambda\right) \tag{2.6}$$

where R can be considered in different ways as commented for the Haversine distance.

Equirectangular approximation

For small distances Pitagora's Theorem is useful to approximate distances between two nodes as a equirectangular projection:

$$d_E = R \cdot \sqrt{x^2 + y^2}, \quad x = \Delta\lambda \cos \varphi_m \, , \, y = \Delta\varphi \tag{2.7}$$

where $\varphi_m = \frac{\varphi_1 + \varphi_2}{2}$ and R is again as for the Haversine distance.

2.3 A^* strategies (w, h)

In section 4 we will discuss a bit about the performance and the optimal paths found by our A* implementation depending on the combination of heuristic and edge-weight distances used. We define a A* strategy as a pair (w, h) where h and w denote the heuristic function and the weight function of the edges, respectively.

In our implementation, eight distance functions are available to use and we identify them by an integer between 1 and 8:

- 1: Haversine (eq. (2.3)),
- 2: Spherical law of cosines (eq. (2.6)),
- 3: Equirectangular approximation (eq. (2.7)),
- 4: Haversine with variable Earth radius (eq. (2.3) and (2.5)),
- 5: Spherical law of cosines with variable Earth radius (eq. (2.6) and (2.5)),
- 6: Equirectangular approximation with variable Earth radius (eq. (2.7) and (2.5)),
- 7: Zero distance, equal to 0 for all pairs of nodes,
- 8: Uniform distance, equal to 1 for all pairs of nodes.

Thus, the strategies (w, h) we consider belong to the set $\{1, ..., 8\} \times \{1, ..., 8\}$.

In section 4 we will concrete the strategies that will be analysed and compared. For now, just note that strategy $(8, 7)$, where we have no heuristic ($h = 0$) and the weight between nodes is uniform and equal to 1 ($w = 1$) is the Breadth-first search (BFS) case. Since it is the greediest strategy, we will use it as a reference to study the speedups of other A* strategies.

3 Implementation in C

In this section, we are going to detail the implementation of the program. It was mainly developed in the C Programming Language. However, we also developed small scripts in Posix, Bash, HTML and Python. By following this section, one should be able to get a working copy of the repository programs installed in their computer.

3.1 Prerequisites

Depending on the native OS some external libraries will need to be included to run this program. See install section below.

This project supports the following C compiler versions:

- Apple gcc v4.2.1
- Ubuntu (16.4) gcc v5.4.0

Other regular C compilers should work as well but have not been tested yet.

For a graphical representation of the paths via Google Maps, Python 2.7 needs to be installed and the following (additional) packages are required:

- `matplotlib`
- `requests`

3.2 Getting started

3.2.1 Make Installation

Setting a running copy of this project with `Make` is straightforward. Make can build the project executables according to the user preferences. From the main directory of the project you can run several `Make` commands with different predefined built in behaviours. They are summarized below:

- Type `make` to install all executables
- Type `make install` to install only the main executables of the program
- Type `make test` to only install the testing executable of the program
- Type `make clean` to remove all executables

In principle, testing executables do not offer useful or reliable data. They do not offer a good user experience and their use is recommended only for development purposes.

Besides the installation with `Make`, this program includes a Python script which can represent graphically the paths on a Google Maps map. To run this script a proper Python environment needs to be configured. There are different ways to get this Python script working. We recommend you two different methods depending on your system architecture.

3.2.2 Getting Python libraries on OSX

The easiest way to set the required packages up is by installing Conda. Download the suitable binary file from their official website and follow the instructions from there.

With a working Python 2.7 Conda environment set up, you can run the following commands to install the required packages from above:

```
conda install matplotlib
```

```
conda install requests
```

3.2.3 Getting Python libraries on Linux

Additionally, you can get a proper Python environment by using the Python Package Index tool (`pip`). It can be installed in a Debian/Ubuntu based system by using the following command:

```
sudo apt install python-pip
```

Then, we just need to ask pip to install the required Python packages:

```
pip install matplotlib
```

```
pip install requests
```

3.3 Execution instructions

The default `Make` installation is the following one:

```
make install
```

It will compile 4 executables which are listed and briefly explained below:

- **bincreator.exe**: it parses a suitable input map file (*.map*) and saves the corresponding graph to a binary file (*.bin*).
- **cmapcreator.exe**: it parses a suitable input map file (*.map*) and saves the corresponding graph to a compressed binary file (*.cmap*).
- **routing.exe**: it takes the graph previously parsed by either **bincreator.exe** (*.bin*) or **cmapcreator.exe** (*.cmap*) and uses it to find the shortest route between the two input points.
- **routeprinter.exe**: it plots the route obtained by **routing.exe** (*.out*) on a map from Google Maps (it requires a proper Python 2.7 environment).

bincreator.exe

This program reads a suitable input map file (*.map*) and, from its information, it creates a graph. The resulting graph is saved in a binary file (*.bin*). To read the map properly, **bincreator.exe** needs that the input map file satisfies the format rules defined below, in the corresponding section. It is executed in the following way:

```
bincreator.exe [file] [...]
```

It requires a mandatory argument `[file]` which is the directory to the input map file. Additionally, it accepts the following optional arguments:

> `-o directory`: which is the path to the directory where the resulting binary file is going to be saved.
>
> `-f`: which makes the program to create the graph faster. To achieve this it does not minimize graph inconsistencies and the performance when working with the resulting graph will be lower.
>
> `-h`: which prints out a brief description of all these arguments.

cmapcreator.exe

This program reads a suitable input map file (*.map*) and, from its information, it creates a graph. The resulting graph is saved in a compressed binary file (*.cmap*). It is much slower than **bincreator.exe** but the output binary file is much smaller. To read the map properly, **cmapcreator.exe** needs that the input map file satisfies the format rules defined below, in the corresponding section.

It is executed in the following way:

```
cmapcreator.exe [file] [...]
```

It requires a mandatory argument `[file]` which is the directory to the input map file.

Additionally, it accepts the following optional arguments:

> `-o directory`: which is the path to the directory where the resulting compressed binary file is going to be saved.
>
> `-f`: which makes the program to create the graph faster. To achieve this it does not minimize graph inconsistencies and the performance when working with the resulting graph will be lower.
>
> `-h`: which prints out a brief description of all these arguments.

routing.exe

This program is the one that performs the AStar Algorithm itself. It reads a graph from a suitable binary file, either compressed or not (*.bin* or *.cmap*), and uses its information to find the shortest

route between two of its nodes. The starting and ending nodes need to be specified as command-line arguments. It is also able to perform the AStar Algorithm by calculating distances between nodes with different approaches.

The program is executed in the following way:

```
routing.exe [file] [-s id/lat,lon] [-e id/lat,lon] [...]
```

It requires a mandatory argument `[file]` which is the directory to the input binary file which can be a regular *.bin* file or a compressed *.cmap* file. Take into account that if a compressed *.cmap* file is used the execution of the program will take more much time. Two more arguments must be included when executing the program:

`-s id/lat,lon`: which sets the starting node.

`-e id/lat,lon`: which sets the ending node.

Both nodes can be defined in two different ways. First, they can be defined by their id from the graph if a single number is given. They can also be defined by entering the latitude and longitude coordinates of a place. In this case, the program will search in the graph for the node which is closest to this location. Notice that these two values must be written without any white space between them and using only a comma (,) to separate them.

Additionally, **routing.exe** accepts the following optional arguments:

`-o directory`: which is the path to the directory where the route information is going to be saved.

`-d number`: which is the selection of the method used to calculate the heuristic distance by the AStar Algorithm.

`-w number`: which is the selection of the method used to calculate the weight between the edges of the graph.

`-h`: which prints out a brief description of all these arguments.

routeprinter.exe

This program makes us of Python 2.7, matplotlib and requests libraries and gmplot to plot the resulting route on a map from Google Maps. It works with the output file from **routing.exe** program (*.out*). Its execution is straightforward:

```
routeprinter.exe -f routes/path.out
```

As a result, the default web browser will open up a windows with the route.

3.4 Format rules of compatible map input files

The two programs that read map files to construct a graph are **bincreator.exe** and **cmapcreator.exe**. They need to work with compatible map input files which have to contain information about the nodes and the ways of the graphs according to some rules. Nodes are the points which are used to construct the paths of the map and the ways contain information about the connectivity between nodes. Those map input files which do not fulfil these rules will not be compatible with these programs. They are the following:

1. Each line of the file contains information about a node or a way.
2. Node lines must start with the word *node*.
3. Way lines must start with the word *way*.
4. Node lines must have the following format:
   node/@id/@name/@place/@highway/@route/@ref/@oneway/@maxspeed/node_lat/node_lon
5. The only mandatory parameters are: *@id*, *node_lat* and *node_lon*.
6. The program is also able to parse the name of the node if a *@name* is given,
7. Way lines must have the following format:
   way/@id/@name/@place/@highway/@route/@ref/@oneway/@maxspeed/member nodes/...
8. The only mandatory parameter is 'member nodes' which is a list of all the nodes that are sequantially connected.
9. If *@oneway* parameter is given, the link between connected nodes will be in a single direction according to the original submitted order.

3.5 Distance functions available

The **routing.exe** program is able to calculate distances between nodes with different approaches. The user can choose one of the following methods through either command-line arguments or while running the program. Below there is a summary of all the methods that are available:

1. Haversine
2. Spherical law of cosines
3. Equirectangular approximation
4. Haversine with variable Earth radius
5. Spherical law of cosines with variable Earth radius
6. Equirectangular approximation with variable Earth radius
7. Zero distance (equal to 0.0)
8. Uniform distance (equal to 1.0)

3.6 Test running

A simple test can be run by performing the following steps.

First of all, we need to download a proper map input file. You can get a map from Spain here.

Save it into a known folder. It is recommended to save it into a folder inside the repository (it is not mandatory but the access to the file will be faster). For instance, a good choice is:

```
path_to_the_repository/AStar-Algorithm/routing/inputs/spain.csv
```

Then, you can execute **bincreator.exe** to read the input map file, create the graph and save it into a binary file.

```
bincreator.exe inputs/spain.csv
```

Unless it is specified through a command-line argument, the output binary file (*.bin*) will be saved into `bin/folder`.

Afterwards, you can execute **routing.exe** to calculate the shortest route between two nodes from this map. These two nodes correspond to the **Basílica de Santa Maria del Mar** (Plaça de Santa Maria) in Barcelona, *(@id) 240949599*, and to the **Giralda** (Calle Mateos Gago) in Sevilla, *(@id) 195977239*.

```
routing.exe bin/map.bin -s 240949599 -e 195977239
```

Unless it is specified through a command-line argument, the output route file will be saved as `routes/path.out`.

Finally, we can represent graphically the route on a map from Google Maps by using the last script.

```
routeprinter.exe -f routes/path.out
```

A plot of the resulting route should be displayed in your default web browser (to see the map an Internet connection is necessary).

4 Results

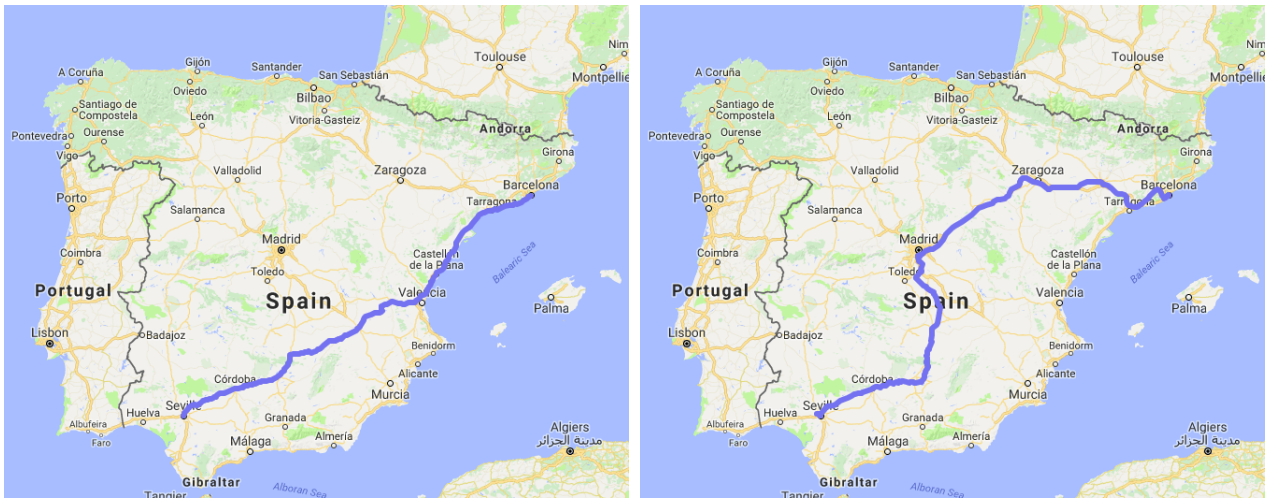
Following the notation we have introduced in section 2.3 here we are going to discuss about the performance and the optimal paths found by our A* algorithm implementation for the following sets of strategies (w, h):

- For all $d \in \{1, 2, 3, 4, 5, 6\}$ we consider the strategies $(w, h) = (d, d)$, which use the same distance for the heuristic and for the edge weights.
- With the expectations that the radius varying distances (4,5 and 6) would be more precise when working locally than when using the mean Earth radius for all nodes, we consider strategies where the weight distance $w \in \{4, 5, 6\}$. On the other hand, for the heuristic distance h we consider the corresponding w versions but that use the mean Earth radius, that is: $(w, h) \in \{(4, 1), (5, 2), (6, 3)\}$.
- Next, we also consider the greedy approaches where there is no heuristic, i.e., $h = 0$ for all nodes, and just vary the edge weight distances. That is, $(w, h) \in \{(w, 7) \,|\, w \in \{1, 2, 3, 4, 5, 6, 7\}\}$. The case where $w \equiv 8$ is the particular case of the breadth-first search strategy.

Therefore, we are considering a total of 16 different strategies among the 64 possibilities that we could consider by using the 8 distances we have proposed. We denote $L(w, h)$ the distance (which is determined by w) of the optimal path found by the A* algorithm using strategy (w, h) (if a optimal path has been found).

4.1 Optimal paths found with different strategies

In Figure 4.1 we have depicted the two different optimal paths found by A*. In Figure 4.1b we can see the path found by the breadth-first search strategy $(8, 7)$, and in Figure 4.1a the optimal path found by the rest of the strategies introduced above; it has been checked with an script that the paths, node by node, are exactly the same.

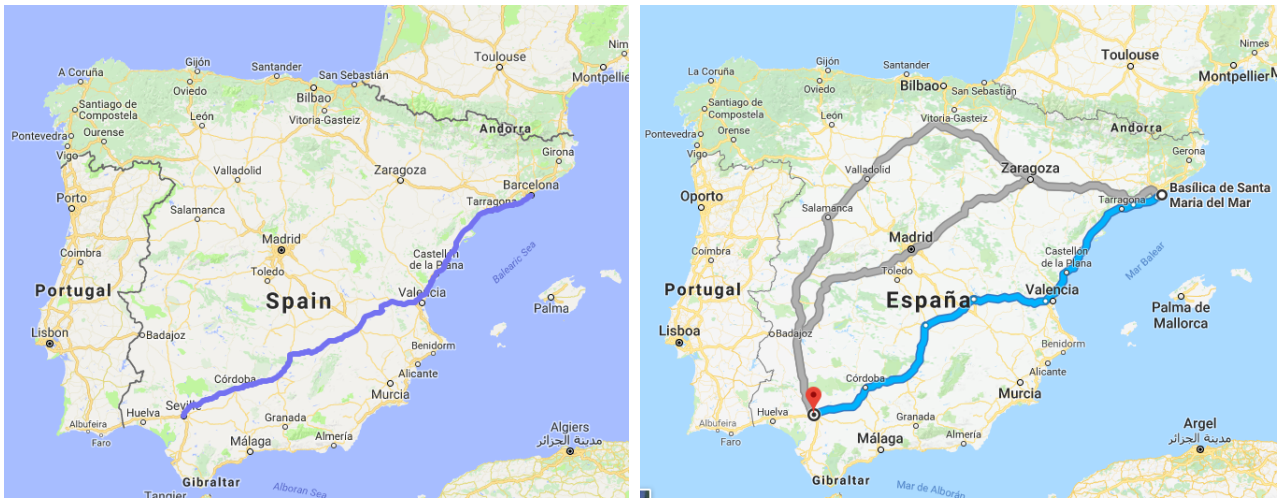


(a) (b)

Figure 4.1: Line in blue: optimal paths for (b) Breadth-first search (a) All the other 15 A* strategies studied.

Observe that the breadth-first search strategy finds a path that is clearly quite more longer than the one found by the other A* strategies with realistic edge weights. In fact, the path from the breadth-first search depends on the order in which nodes are expanded during the algorithm.

Finally, from Figure 4.2, note that the optimal path with respect to distance found by A* (Figure 4.2a) is slightly different from the path suggested by Google Maps, which in principle minimizes with respect to time (Figure 4.2b). The bigger differences are in the middle of the paths, between Valencia and Córdoba.



(a) (b)

Figure 4.2: Barcelona to Sevilla. Line in blue: optimal paths for (a) All the A* strategies studied except for the breadth-first search case (with respect to distance). (b) Google Maps own strategy (with respect to time).

4.2 A* performance metrics with different strategies

Now we proceed to study some performance metrics for the different strategies. We focus on four different parameters:

A. The total CPU time spent to complete the A* algorithm, in seconds (Figure 4.3 A),

B. The A* algorithm Speedup with respect to the time ($t_{BFS} \approx 360$ seconds) that the Breadth-first search algorithm lasts (Figure 4.3 B):

$$\text{Speedup} = \frac{t_{BFS}}{t_{A^*}(w,h)}. \tag{4.1}$$

C. The total number of iterations of the main while loop of the A* algorithm that is performed until the **OPEN** list is empty or an optimal path has been found (Figure 4.3 C),

D. And the relative path distance of each strategy with respect to strategy $(1,1)$ (Figure 4.3 D), which is the one with the smallest path distance L [3]:

[3]We do not account the breadth-first search case, since in this case the edge weight distance is not realistic.

$$\Delta L(w) := L(w,h) - L(1,1) \geq 0. \tag{4.2}$$

Note that the path distances are determined by the distance measure w and the heuristic does not play a role.

Now, referring to Figure 4.3 the results are the following:

A. First, we can see that the time spent by A* in strategies that use the trivial heuristic ($h = 0$) spent about x3 and x4 more time. The fastest strategies seem to be the ones that use either as w or h, the Spherical law of cosines (with and/or without varying radius, i.e., $h, w \in \{2, 5\}$). Among those, the fastest strategy is $(w, h) = (2, 2)$ where the Spherical law of cosines distance is used both as heuristic and as edge weight distance. Among the strategies where there is no heuristic, the best one seems to be the one that uses as w the Equirectangular approximation (without varying Earth radius).

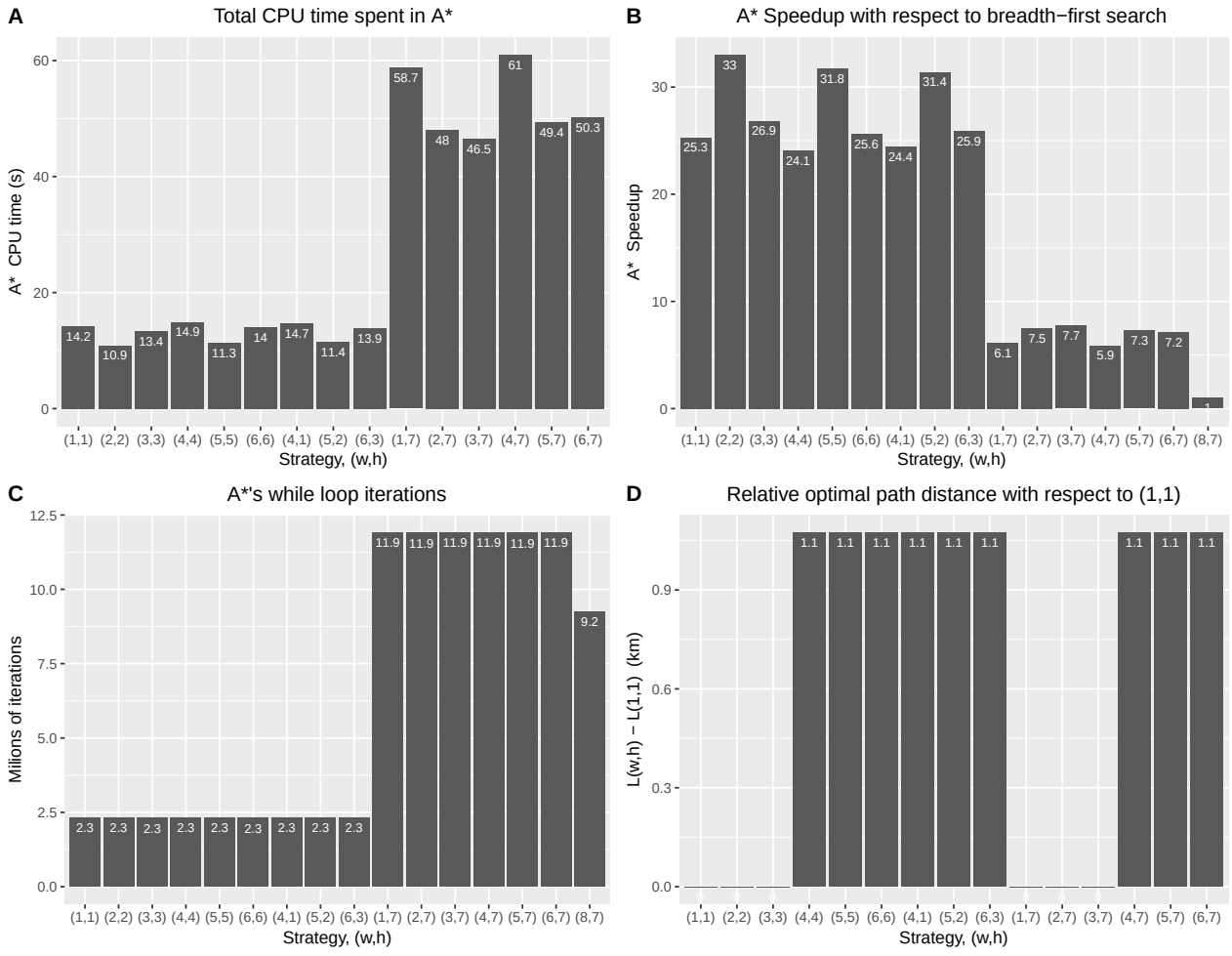


Figure 4.3: **A**: Total CPU time spent by the A* algorithm . **B**: Speedup of each strategy with respect to the breadth-first search strategy $(8, 7)$. **C**: Total number of iterations of the A*'s main while loop. **D**: Relative optimal path distance with respect to strategy $(1, 1)$, i.e., $L(w, h) - L(1, 1) > 0$. In Table A.1 from Appendix A there is all the data required for building these plots.

B. Now we look at the speedup with respect to the breadth-first search approach that lasted about 360 seconds. With this graphic now it is more clear the difference between strategies: the strategies that combine both Spherical law of cosines approaches (radius varying and non-varying) lead to an speedup above 31.4, which means that these strategies are more than 31 times faster than the most greedy approach we could consider. Among those, the best one is strategy $(2, 2)$, as we concluded above, which is 33 times faster than the BFS.

C. By comparing the number iterations in millions we have that strategies that use a non-trivial heuristic require about 2.3 millions (the differences between them are small, they differ at most by 20.000 iterations). When using the trivial heuristic with a realistic weight distance the number of iterations increases a bit more than 5 times and reaches almost 12 million iterations. For the BFS approach the total number of iterations is slightly reduce to 9.2 millions (note that the path found now is different from the other strategies).

D. Finally we can make some comments about the lengths of the optimal paths found by the different strategies. The main conclusion is that for the cases $w \in \{1, 2, 3\}$ where the Earth (mean) radius is constant the three different distances are in fact quite equivalent since the differences are found in the order of centimetres. If we compare these distances with the same ones but with a varying Earth radius(i.e. $w \in \{4, 5, 6\}$), we see that the difference in path lengths is about 1.1 kilometre; a distance which is quite small when compared to the total length of the paths which is about 959 kilometres.

4.3 Finding other paths changing start and goal nodes

Since our implementation, given starting and ending pairs of latitudes and longitudes, i.e., $S = (\varphi_s, \lambda_s)$ and $E = (\varphi_e, \lambda_e)$, allows us to find the corresponding the available (in the dataset) starting and ending nodes that are more closer to the pairs S and E by means of the Quadrance measure (the euclidean distance squared), in this section we consider some additional examples of optimal paths found by our A* algorithm, and compare them with the paths proposed by Google Maps. To find latitude and longitude pairs of the desired locations we have used the web-page app from [9].

We consider the following latitude-longitude pairs (φ, λ) for some locations:

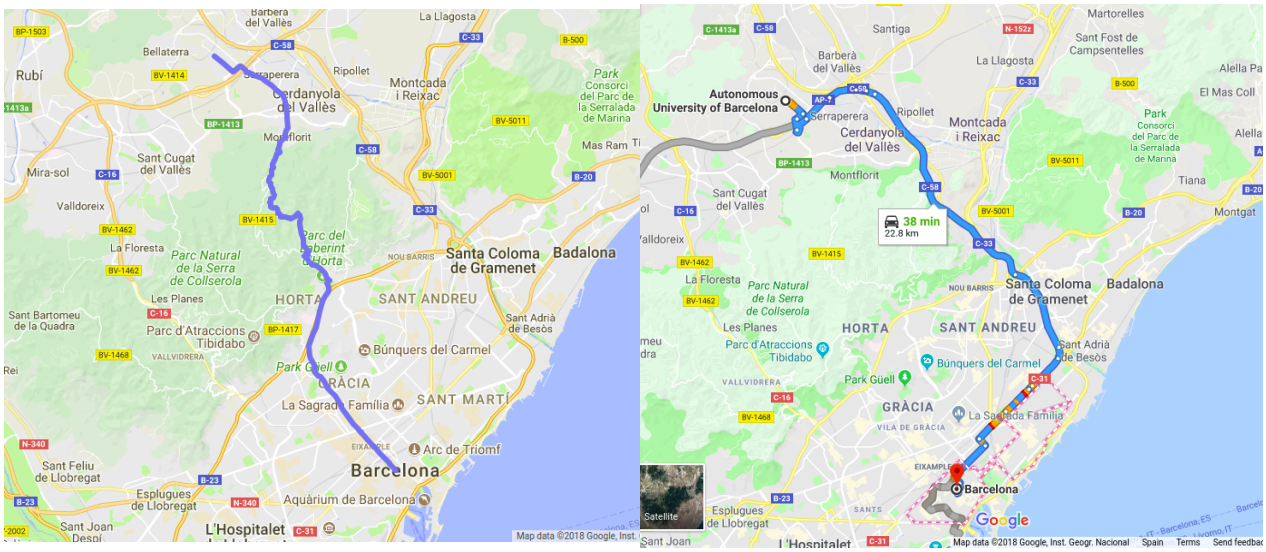
- Manresa, Barcelona, Spain. $(\varphi, \lambda) = (41.716389, 1.822082)$.
- Autonomous University of Barcelona Campus de la UAB, Plaça Cívica, 08193 Bellaterra, Barcelona, Spain. $(\varphi, \lambda) = (41.501926, 2.104854)$.
- Madrid, Spain. $(\varphi, \lambda) = (40.416775, -3.70379)$.
- Barcelona, Spain. $(\varphi, \lambda) = (41.385064, 2.173403)$.
- Santiago de Compostela, A Coruña, Spain. $(\varphi, \lambda) = (42.8782132, -8.544844499999954)$.

The routes we consider and the commands required to execute the algorithm that compute the optimal path are the following:

- Start from UAB and End in Barcelona: `./routing.exe bin/map.bin -s 41.501926,2.104854 -e 41.385064,2.173403`. See Figure 4.4.

- Start from Barcelona and End in Madrid: `./routing.exe bin/map.bin -s 41.385064,2.173403 -e 40.416775,-3.70379`. See Figure 4.5.
- Start in Manresa and End in Santiago de Compostela: `./routing.exe bin/map.bin -s 41.716389,1.822082 -e 42.8782132,-8.544844499999954`. See Figure 4.6.

In order to compute the paths we follow the strategy $(w, h) = (2, 2)$ since it was the most efficient one as we saw in section 4.2.

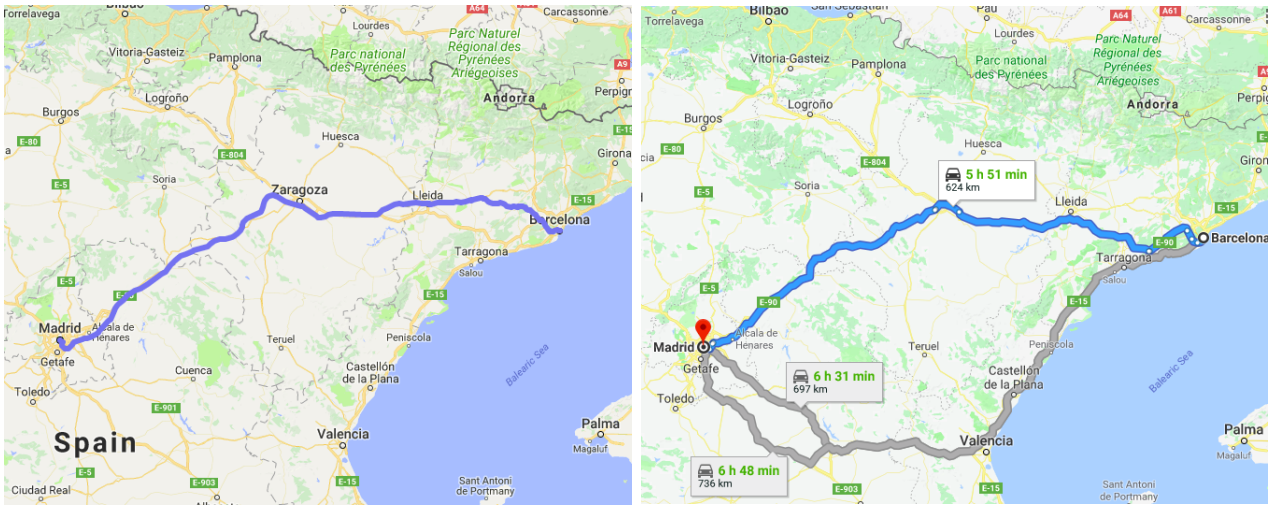


(a) (b)

Figure 4.4: UAB to Barcelona. Line in blue: optimal paths for (a) All the A* strategies studied except for the breadth-first search case (with respect to distance). (b) Google Maps own strategy (with respect to time).

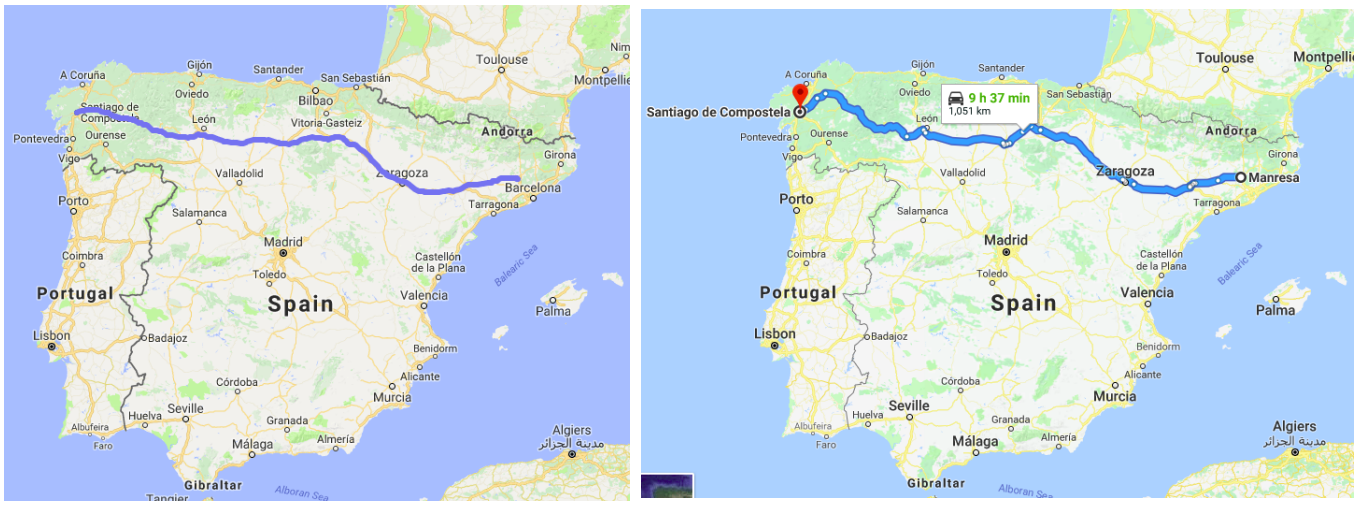
By comparing the paths fund by our A* implementation and Google Maps for the routes considered above we can see that:

- UAB to Barcelona: the A* path seems to be quite shorter than the one proposed by Google Maps, which make sense since in Google Maps it minimizes in time, and as it can be observed, the A* path passes through a mountainous route, which obviously will be much longer in terms of time.
- In the case of Barcelona to Madrid, both paths seem to coincide pretty well, probably because the route involves superhighways so that the optimal path in terms of time and distance more or less coincide.
- For the case of Manresa to Santiago de Compostela, as in the previous case the paths seem to be very similar, but one can note that near Santiago the Google Maps' path seems to be quite larger in terms of distance, although the one found by A* would be probably shorter in terms of time.



(a) (b)

Figure 4.5: Barcelona to Madrid. Line in blue: optimal paths for (a) All the A* strategies studied except for the breadth-first search case (with respect to distance). (b) Google Maps own strategy (with respect to time).



(a) (b)

Figure 4.6: Manresa to Santiago de Compostela. Line in blue: optimal paths for (a) All the A* strategies studied except for the breadth-first search case (with respect to distance). (b) Google Maps own strategy (with respect to time).

5 Conclusions

In this work we have developed in C an A* algorithm that is able to solve a general routing problem. The first example we have considered and the one we have used to analyse the performance of the different strategies considered (by varying heuristic and edge-weight distances) is the route that goes from Basílica de Santa Maria del Mar (Plaça de Santa Maria) in Barcelona to the Giralda (Calle Mateos Gago) in Sevilla. We have tried to use three different distances: the Haversine, the Spherical law of cosines, and the Equirectangular approximation approaches, both by taking as Earth radius the mean Earth radius or by considering a varying Earth radius depending on the latitude by means of the geocentric radius formula.

We have also compared the strategies that make use of these distance variants with the A* algorithm without heuristic and also with the Breadth-first search approach. We have been able to appreciate the powerfulness of the heuristic function, since it speedups up to 33 times the performance that the more greedy breadth-first search approach has.

In addition, we have shown the paths found by our algorithms and compared them with those from Google Maps, for some routes inside Spain, such that Barcelona to Madrid, UAB to Barcelona and Manresa to Santiago de Compostela. Generally, all paths were quite in agreement with those from Google Maps, although there were some differences between them given the fact that in Google what is minimized is the time, and in our A* approach what is minimized is the distance.

Finally, we have realized that the Spain's graph built from the data available seems to have more than one connected component so that, locally, there are groups of nodes that are isolated from the rest. This has been seen when trying to compute some paths between locations that are in small and rural towns. We have started to develop a function able to clean the graph in some way so that the number of connected components is reduced in order to avoid the mentioned problem when trying to find optimal paths between some concrete locations. This improvement of our work is left as future work.

References

[1] A*-Algorithm GitHub repository, Martí Municoy & Daniel Salgado, https://github.com/dsalgador/AStar-Algorithm.

[2] A* search algorithm. (2018, January 28). In Wikipedia, The Free Encyclopedia. Retrieved 18:34, February 7, 2018, from https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=822855961

[3] Admissible heuristic. (2017, December 7). In Wikipedia, The Free Encyclopedia. Retrieved 19:25, February 7, 2018, from https://en.wikipedia.org/w/index.php?title=Admissible_heuristic&oldid=814117910.

[4] Heuristics: Intelligent Search Strategies for Computer Problem Solving, Judea Pearl. Addison-Wesley Pub (Sd), ISBN: 0201055945, 1984-04.

[5] Calculate distance, bearing and more between Latitude/Longitude points, Chris Veness. http://www.movable-type.co.uk/scripts/latlong.html.

[6] Haversine formula. (2017, December 31). In Wikipedia, The Free Encyclopedia. Retrieved 00:31, February 8, 2018, from https://en.wikipedia.org/w/index.php?title=Haversine_formula&oldid=817871096.

[7] Earth radius. (2018, January 22). In Wikipedia, The Free Encyclopedia. Retrieved 15:45, February 7, 2018, from https://en.wikipedia.org/w/index.php?title=Earth_radius&oldid=821830040

[8] P. E. Hart, N. J. Nilsson and B. Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, in IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968. doi: 10.1109/TSSC.1968.300136. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4082128&isnumber=4082123.

[9] Latitude, longitude and address of any point on Google Maps, https://www.gps-coordinates.net/.

A Tables

Table A.1: Total CPU time observations (3 in total) and its average, the path distance $L(w, h)$, and the number of iterations required for each strategy.

Distance, w	Heuristic, h	t_1 (s)	t_2 (s)	t_3 (s)	t_{avg} (s)	Path distance, $L(w)$	Iterations
1	1	14.58	13.96	14.09	14.21	958815.013	2307242
2	2	10.89	10.92	10.85	10.89	958815.036	2307243
3	3	13.14	13.46	13.56	13.39	958815.013	2302274
4	4	14.86	15.02	14.93	14.94	959888.752	2307243
5	5	11.31	11.25	11.44	11.33	959888.775	2307244
6	6	14.09	13.99	14.02	14.03	959888.752	2302274
4	1	14.77	14.72	14.67	14.72	959888.752	2319882
5	2	11.28	11.42	11.65	11.45	959888.775	2319882
6	3	13.93	13.74	13.97	13.88	959888.752	2315176
1	7	57.33	59.66	59.22	58.74	958815.013	11905449
2	7	45.96	48.11	50.02	48.03	958815.036	11893494
3	7	46.34	46.5	46.58	46.47	958815.013	11905449
4	7	59.93	62.14	61.03	61.03	959888.752	11905449
5	7	47.05	49.98	51.13	49.39	959888.775	11893494
6	7	49.15	50.9	50.74	50.26	959888.752	11905449
8	7	332.04	380.75	366.4	359.73	2995	9248953