Universitat Autònoma de Barcelona
Facultat de Ciències
Master in Modelling for Science & Engineering

A* assignment — A routing problem

Optimisation

Martha Domhoefer (NIU: 1586407) & Isidre Mas Magre (NIU: 1428185)

December 12, 2021

Contents

1 Introduction

This assignment consists in computing and writing the shortest 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 by implementing the A* algorithm, using as resources the provided data files and indications explained in the assignment's description [1].

Clearly, both points are connected via a huge amount of paths and even an infinite amount of paths if closed loops are allowed. To find the minimal path, i.e. the shortest path, we implemented two algorithms, namely A* and Dijkstra. The latter is a predecessor of the A* algorithm, and for this reason it will be briefly explained, although it was implemented mainly for comparison reasons. The algorithms definitions and pseudocodes provided in the following sections were extracted from the slides of this part of the course [2].

1.1 The problem as a graph

To represent the map of Spain, we used a directed weighted graph $G = (V, E)$ where the vertices V and edges E were read from a data set extracted from OpenStreetMap according to the format specified. Specifically, the vertices (also referred to as nodes) V of the graph refer to specific locations in the map with attached geographical coordinates of latitude and longitude. The edges E, refer to the connections between two nodes, they can have one direction (arrow/ directed edge) or two directions (two arrows/ two directed edges) and they may represent different transport environments such as streets, roads, highways, railroads, etc. The arrows are weighted by the distance between them obtained from the Haversine formula which assumes the geographical coordinates lying on a sphere [3].

1.2 Dijkstra algorithm

The Dijkstra algorithm was conceived by computer scientist Edsger W. Dijkstra in 1956. It is based on a (controlled) greedy strategy and was designed to solve the single-source shortest paths problem by computing a minimal spanning tree. It can also solve the routing problem by stopping the algorithm once the shortest path to the destination node has been determined. A quick way to understand how it works is analysing the pseudocode in **Algorithm 1** which implements the general Dijkstra algorithm for a graph G:

Algorithm 1 Dijkstra algorithm

```
1:  procedure DIJKSTRA(graph G, source)
2:      Pq ← EmptyPriorityQueue
3:      expanded[G.order] ← initialized to false
4:      dist[G.order] ← initialized to ∞
5:      parent[G.order] ← uninitialized
6:      dist[source] ← 0
7:      parent[source] ← ∞
8:      Pq.add_with_priority(source, dist[source])
9:      while not Pq.isEmpty do
10:         node ← Pq.extract_min()
11:         expanded[node] ← true
12:         for each adj ∈ node.neighbours and not expanded[adj] do
13:             dist_aux ← dist[node] + ω(node, adj)
14:             if dist[adj] > dist_aux then
15:                 if dist[adj] = ∞ then Pq.add_with_priority(adj, dist_aux)
16:                 elsePq.decrease_priority(adj, dist_aux)
17:                 dist[adj] ← dist_aux
18:                 parent[adj] ← node
        return dist, parent
```

It should be noted that Dijkstra not only finds the shortest path from a starting point to a goal but it finds the minimum path to all accessible nodes from a source. The algorithm returns two vectors (dist & parent) on which each component corresponds to a node, one with the minimum distances and the other with the parent to each node needed to reconstruct the path of minimum distance. The pseudocode includes several functions which will be addressed in further detail when presented the C implementation.

1.3 A* ALGORITHM

The A* algorithm was first published in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael. It can be seen as an extension of the Dijkstra algorithm to solve the routing problem, achieving a better performance by using heuristics to guide its search. It is an informed search algorithm, or best-first search, which means it keeps a tree of paths originating at the start node extending it one arrow at a time until its termination criterion is satisfied. At each iteration of its main loop, A* needs to determine which of its paths to extend. The determination is based on the cost of the path and an estimate/heuristic of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

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

where v is the next node on the path, $g(v)$ is the cost of the path from the start node to v, and $h(v)$ is an heuristic function that estimates the cost of the cheapest path from v to the goal.

As we did before with Dijkstra, the complete algorithm is presented in **Algorithm 2** and all functions will be explained in further detail when the C implementation is discussed.

Algorithm 2 A* algorithm

```
1:  procedure ASTAR(graph G, start, goal, h)
2:      Open ← EmptyPriorityQueue
3:      parent[G.order] ← uninitialized
4:      g[G.order] ← initialized to ∞
5:      g[start] ← 0
6:      parent[start] ← ∞
7:      Open.add_with_priority(start, g, h)
8:      while not Open.isEmpty do
9:          current ← Open.extract_min(g,h)
10:         if current is goal then return g, parent
11:         for each adj ∈ current.neighbours do
12:             adj_new_try_gScore ← g[current] + ω(current, adj)
13:             if g[adj] > adj_new_try_gScore then
14:                 parent[adj] ← current
15:                 g[adj] ← adj_new_try_gScore
16:                 if not Open.BelongsTo(adj) then Open.add_with_priority(adj, g, h)
17:                 elseOpen.requeue_with_priority(adj, g, h)
18:
        return failure
```

2 Methodology

In this part, the organisation and instructions to execute the scripts in the repository will be presented and explained. Moreover it will be explained in greater detail how the implementation of A* and Dijkstra algorithm was done in C, and how the code was adjusted to our specific problem.

2.1 Program operation & repository structure

The source code includes several scripts that organize the code following some strategies that attempt to improve re-usability and efficiency of code execution. In **Figure 1** we display the repository structure.

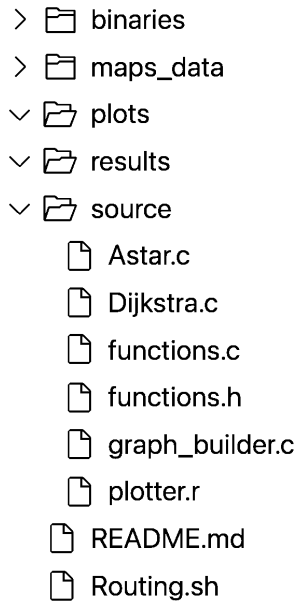

Figure 1: Repository folders and source code files.

A bash file (Routing.sh) initiates the process and binds together the compilation and execution of all the scripts in a self-contained program that can be controlled through the following input arguments:

- -f: Name of the map to be used. By default is spain.csv
- -o: Origin node. The argument can be a valid node ID or coordinates in the format "LAT,LON". By default is set to the node ID 240949599.

- -d: Destination node. The argument can be a valid node ID or coordinates in the format "LAT,LON". By default is set to the node ID 195977239.
- -a: Algorithm to use, either Dijkstra or Astar. By default is set to Astar.
- -c: Color to draw the solution route in the leaflet plot. By default is set to red.

An example execution with all the mentioned arguments would look like this:

```
bash Routing.sh -f cataluna.csv -o 41.61733,0.62554 -d 240949599 -a
   Dijkstra -c blue
```

However, to execute the baseline program with default values only the following command needs to be executed:

```
bash Routing.sh
```

Initially, the bash script will compile all the scripts if they aren't compiled yet. It will then check if a binary file containing the graph structure already exists. If not, it will execute the program graph_builder. Then it will execute the algorithm requested (either A* or Dijkstra) and finally plot the results in an interactive map.

2.2 Reading data and graph structure

The script graph_builder.c is in charge of reading the data from the formatted data files and storing relevant information in the following structure that all the scripts will use to codify the map as a graph:

```
typedef struct node{
    unsigned long id; // Node identification
    char *name; // Node name
    double lat, lon; // Node geographical position
    unsigned short nsucc; // Number of node successors; i. e. length of
   successors
    unsigned *successors; //Stores the indices of successors
}node;
```

To begin reading the data, the script opens the file and sets a pointer called *nodesdata to the beginning of it. The main procedure to access the different rows and fields in the file is by using the functions getline() and strtoke(). The first stores a whole line of the data file in a buffer and jumps the position of the pointer *nodesdata to the next line. The latter is a custom implementation of the tokenizer strtok() included in the standard library string.h. In

our implementation it returns an empty string when it encounters an empty field instead of NULL. It was implemented in the following way:

```
char* strtoke(char *str, const char *delim)
{
  static char *start = NULL;
  char *token = NULL;
  if (str) start = str;
  if (!start) return NULL;
  token = start;
  start = strpbrk(start, delim);
  if (start) *start++ = '\0';
  return token;
}
```

With those tools and a flag called field which is used to keep track of the field we are retrieving it is relatively easy to implement the logic necessary to retrieve the data from the file and store it conveniently in an array of allocated node structures. To know how many elements we need to allocate and also for iteration purposes, first we set up a counter called nnodes in which we store the total number of nodes by counting the rows of the file which contain the keyword "node" in their first field.

Once we have the total number of nodes we can allocate the necessary memory for them. Then the pointer *nodesdata rewinds to the beginning of the file and we start again iterating over each "node" line retrieving relevant fields while storing the values in the corresponding variables (id, name, lat & lon) within the list of nodes.

Once we have iterated over all the nodes we can start iterating over the "way" lines without rewinding the file as they are listed immediately after the "node" lines. Here the logic to read out the information of a field must be adapted as memory had to be dynamically allocated for the *successors of a node. Starting from field 9, the correspondent successors for each pair of consecutive nodes could be read out. In field 8, the information if the nodes of a line are connected in only one way or both ways could be retrieved. The number of successors nsucc was also updated every time a new successor got assigned to a node. The nodes are specified with their id in the "way" lines. because we later wanted to access nodes in our implementation via their indices in the list of nodes, we decided to use a binary search algorithm to efficiently find the index corresponding to a given node id. To address the inconsistencies in the file, some strategies were applied, specifically, when a node in a way could not be found using binary search it is skipped and the next valid node is used in the process of assign the successors. Pre-

viously we tried to discard the adjacent pairs when a node was missing but that resulted in less connections than the ones displayed in the valence distribution provided in the task description.

Finally, the graph structure codified in the nodes list was written into a binary file to be read later by the scripts Astar.c and Dijkstra.c. The values accessed through pointers inside the node structure were written separately as their values will not be carried along the node structure. Only the pointers were actually written.

2.3 Binary Search

As mentioned before, we used binary search instead of a sequential search algorithm to find the index of a node with its corresponding node id. Binary search has a time complexity of O(log n), which is way better than the time complexity of O(n) that sequential search has. To achieve that, binary search sequentially splits a sorted search space in two regions checking if the split point (usually located in a position half the number of elements in the search region) is the searched element, otherwise it compares it's value to decide the next region in which to make the split. Effectively, on each iteration it halves the search space in which a random search is performed, proving to be more efficient than sequential search when the number of elements is large. We implemented this function with this declaration:

```
1 bool binarysearch(unsigned long ident, node *nodes, unsigned n, unsigned *
    index);
```

The function takes as arguments the id to be searched (ident), the node structure in which to look for the node (*nodes), the total number of nodes (n) and a pointer to the location of memory where the found index is to be stored (*index). It returns a boolean with the value true if the id is found and false if it wasn't found.

2.4 Binary Heap

Both algorithms Dijkstra and A* make use of a priority queue to keep track of the next paths to be explored in the priority order. For efficiency reasons a binary heap structure seemed more adequate for this task than a linked list sorted by priority as its computation time for updating priorities is of order $O(\log n)$ instead of $O(n)$. Thus, to implement a binary heap in our program the following structure was used:

```c
typedef struct Heap{
    double *priority;
    unsigned *node_index;
    int count;
    int capacity;
}Heap;
```

The structure keeps the count of elements in the binary heap (count), the total capacity (capacity), and two allocated arrays (*priority & *node_index) that will keep the values of the priority and the node index in an order that maps a binary heap using a set of arithmetic rules implemented in the supplementary functions.

The function CreateHeap() returns an empty heap structure with a given capacity:

```c
Heap *CreateHeap(unsigned capacity);
```

The function heapify_bottom_top() takes a node of the binary heap and compares its priority with the parent's priority swapping its position with the parent if it has less priority, doing that repeatedly from bottom to top until all the binary heap is reordered:

```c
void heapify_bottom_top(Heap *h,int index);
```

The function heapify_top_bottom() takes a node of the binary heap and compares its priority with its childrens swapping its position with the children with more priority, and does that repeatedly from top to bottom until all the binary heap is reordered:

```c
void heapify_top_bottom(Heap *h, int parent_node);
```

The function insert() inserts a new node with a given index and priority at the end of the binary heap and heapifies the tree from bottom to top until it's ordered again by priority.

```c
void insert(Heap *h, double priority, unsigned node_index);
```

The function PopMin() pops the element at the top of the binary heap (higher priority) and replaces it with the last element of the binary heap, then heapifies from top to bottom to recover the priority order.

```c
unsigned PopMin(Heap *h);
```

The function decreasePriority() updates the priority of a node already existing in the binary heap and heapifies either top to bottom or bottom to top to recover the order by priority of the binary heap.

```c
void decreasePriority(Heap *h, double priority, unsigned node_index);
```

2.5 Heuristic and cost function

In our graph, the cost of each arrow is the distance between the nodes. To calculate this distance we used the geographical coordinates of latitude and longitude in the Haversine formula.

For the A* algorithm, the heuristic function needed to estimate the cost of a possible minimum path from every node to the destination without overestimating it. Since the map of Spain is mainly land and most of it's surface is connected, we could argue that a good estimate of a minimum path would be the distance from each node to the destination node following the earth surface. The situation would be different if big surfaces of forbidden terrain were located between regions of the map (water bodies, deserts, etc.). To simplify things, we took the same function we used to calculate the distance (cost) of an arrow connecting two nodes.

The choice of this function was critical both to correctly weight the edges and to use an adequate heuristic function. The earth doesn't have a regular shape, it is an ellipsoid and it's surface has many geological protuberances. However, it seems reasonable to approximate it's surface as an sphere and take as distances between two nodes the arc of the sphere given by the Haversine formula, which we implement with the following function:

```
1  double get_distance(double lat1, double lon1, double lat2, double lon2)
2  {
3      double const R = 6371e3;
4      double phi1 = lat1 * M_PI/180;
5      double phi2 = lat2 * M_PI/180;
6      double delta_phi = (lat2 - lat1) * M_PI/180;
7      double delta_lambda = (lon2 -lon1) * M_PI/180;
8      double a = pow(sin(delta_phi/2),2) + cos(phi1)*cos(phi2)*pow(sin(
       delta_lambda/2), 2);
9
10     return R * 2 * atan2(sqrt(a), sqrt(1-a));
11 }
```

2.6 Dijkstra

To implement the Dijkstra Algorithm presented in **Algorithm 1** the following structure was defined to store the current state of paths visited:

```
1 typedef struct DijkstraState{
2   double *g;
3   unsigned *parent;
4   bool *expanded;
5 } DijkstraState;
```

In the structure called DijkstraState each of the variables will allocate memory to store as many values as nodes on the graph (nnodes). Specifically, *expanded points to boolean values for each node to keep track of the nodes extracted from the priority queue Open (line 2 of **Algorithm 1**) and will be initialized to false. The pointer *g points to the values of the updated distance of a given node to the start node while different paths are visited, it will be initialized to ∞ for all nodes except for g[origin_index] which will be initialized to 0. The pointer *parent will point to the node index of its parent as found by the Dijkstra algorithm, it is uninitialized except for parent[origin_index] which will be initialized to ∞ (in practice a number outside of the range of indices for the nodes, for example nnodes).

The Dijkstra algorithm was implemented in a function that returns an entity of the previous structure as the output, and it can be found in the source code Djikstra.c with the following declaration:

```
1 DijkstraState Dijkstra(node *nodes, unsigned nnodes, unsigned index_origin
    , unsigned index_destination)
```

This function accepts as arguments a node structure (*nodes) which contains all the nodes read from the data file in the structure defined previously, the total number of nodes (nnodes), the index of the origin node (index_origin) and the index of the destination node (index_destination).

2.7 Astar

To implement the A* algorithm presented in **Algorithm 2** the following structure was defined to store the current state of paths visited:

```
1 typedef struct AStarState{
2   double *g, *h;
3   unsigned *parent;
4   bool *expanded;
5 } AStarState;
```

In the structure called AStarState each of the variables will allocate memory to store as many values as nodes there are on the graph (nnodes). Specifically *expanded points to boolean values for each node that keep track of the nodes extracted from the priority queue Open (line 2 of **Algorithm 2**) and will be initialized to false. The pointer *g points to the values of the updated distance of a given node to the start node while different paths are visited, it will be initialized to ∞ for all nodes except for g[origin_index] which will be initialized to 0. The pointer *h points to the value of the heuristic function for a given node which is the distance calculated using the Haversine formula from each node to the destination. The pointer *parent will point to the node index of of its parent as found by the A* algorithm, it is uninitialized except for parent[origin_index] which will be initialized to ∞ (in practice a number outside of the range of indices for the nodes, for example nnodes).

The A* algorithm was implemented in a function that returns an entity of the previous structure as the output, and it can be found in the source code Astar.c with the following declaration:

```
1 AStarState Astar(node *nodes, unsigned nnodes, unsigned index_origin,
    unsigned index_destination);
```

This function accepts as arguments a node structure (*nodes) which contains all the nodes read from the data file in the structure defined previously, the total number of nodes (nnodes), the index of the origin node (index_origin) and the index of the destination node (index_destination).

2.8 R plotter

The R script (plotter.r) simply takes the arguments necessary to read the formatted .CSV file which contains the resulting path in form of latitude and longitude data of the consecutive nodes and plots them in an interactive map using the leaflet library. It creates a widget which is then stored in a .html file.

3 Results

3.1 Results Binary Graph

After the creation of the binary graph we got the valence distribution displayed in **Figure 2**:

```
Nodes with valence 0: 945177
Nodes with valence 1: 1101296
Nodes with valence 2: 20638977
Nodes with valence 3: 1044780
Nodes with valence 4: 159961
Nodes with valence 5: 4840
Nodes with valence 6: 581
Nodes with valence 7: 45
Nodes with valence 8: 22
Nodes with valence 9: 2
```

Figure 2: The valence distribution

One can observe that the valence 2 is the most frequent one. The graph has a maximum valence of 9 and the mean valence is 1.932635. As the valence distribution is the same as the one presented in the task we could assume that our binary graph file was correctly read.

3.2 CPU/ Time of execution

We ran the complete code on two different machines, namely a HP Laptop with a windows 10 64-bit operating system, a 16GB RAM and a Intel(R) Core(TM) i7-8750H CPU with a clock speed of 2.21 GHz. The other machine was a MacBook Air M1 using Mac OS with a clock speed of 3.2 GHz and 8GB of RAM. The execution times for each script can be compared in the tables 1, 2 & 3.

Table 1: Execution times in CPU sec for the script graph_builder.c on two different machines with Mac OS and Windows 10.

CPU sec	Count n nodes	Read node lines	Read way lines, assign succ	Write graph into binary file	Total time to build & write graph
Mac OS	1.162888	5.074912	7.469525	2.505327	16.212854
Windows 10	34.463	53.486	23.381	9.47	120.8

Table 2: Execution times in CPU sec for the script Astar.c on two different machines with Mac OS and Windows 10.

CPU sec	Read bin file	Search indices of origen & dest.	Perform Astar	Save results in output file	Total computation time Astar
Mac OS	0.486382	0.000007	0.571535	0.002708	1.060705
Windows 10	0.398	0.000000...	1.613	0.012	2.026

Table 3: Execution times in CPU sec for the script Dijkstra.c on two different machines with Mac OS and Windows 10.

CPU sec	Read bin file	Search indices of origen & dest.	Perform Dijkstra	Save results in output file	Total computation time Dijkstra
Mac OS	0.839992	0.000080	2.279133	0.002709	3.122334
Windows 10	0.413	0.000000...	5.139	0.011	5.567

Of course the exact time differed from one execution to another as it is also dependant on the overall usage of the machine. What can be observed anyway is, that the Mac OS system performed way better than Windows 10. Moreover Astar was performing faster on both machines than Dijkstra.

3.3 Main result

The shortest distance found from Basílica de Santa Maria del Mar (Plaça de Santa Maria) in Barcelona to the Giralda (Calle Mateos Gago) in Sevilla was 958815.012846 m. With the script plotter.r, we were able to plot the route on a map of spain as seen in the **Figure 3**.

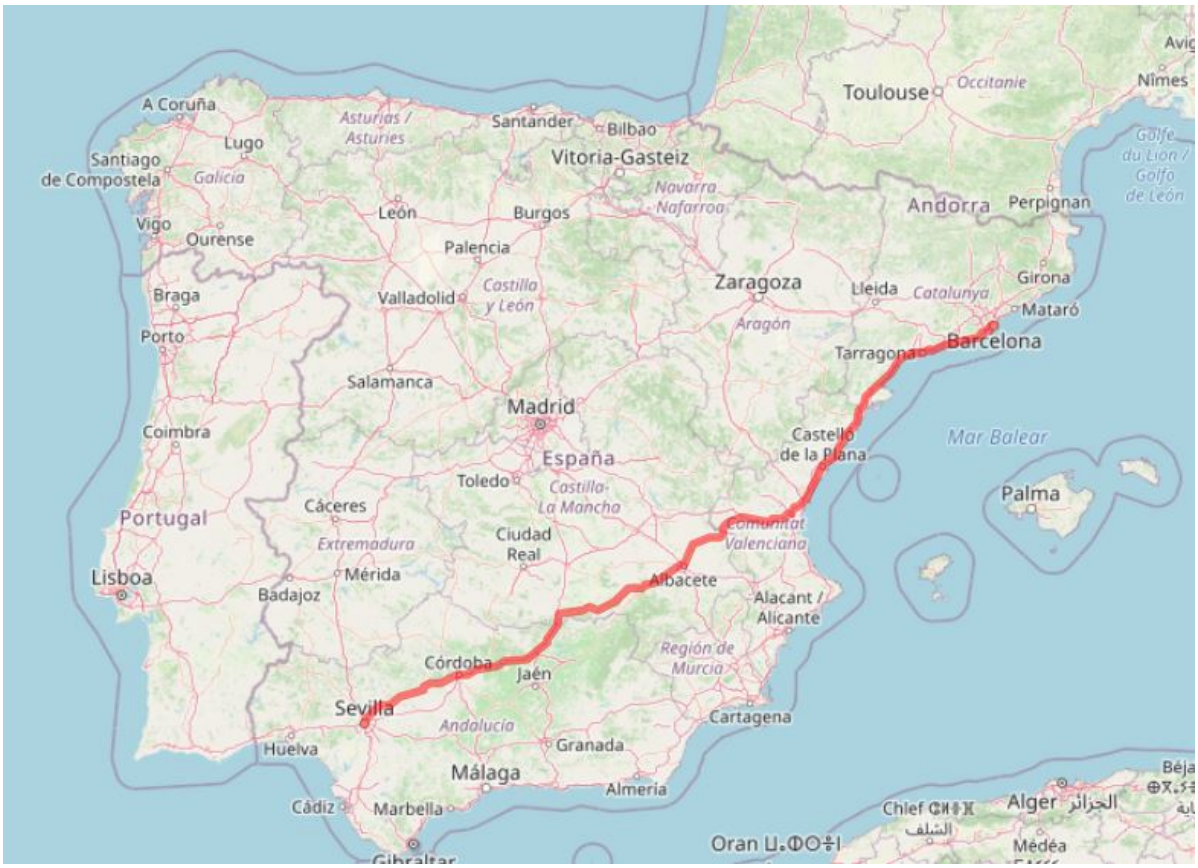

Figure 3: The map of Spain, red line showing the shortest path between Basílica de Santa Maria del Mar (Plaça de Santa Maria) in Barcelona to the Giralda (Calle Mateos Gago) in Sevilla.

3.4 Special cases

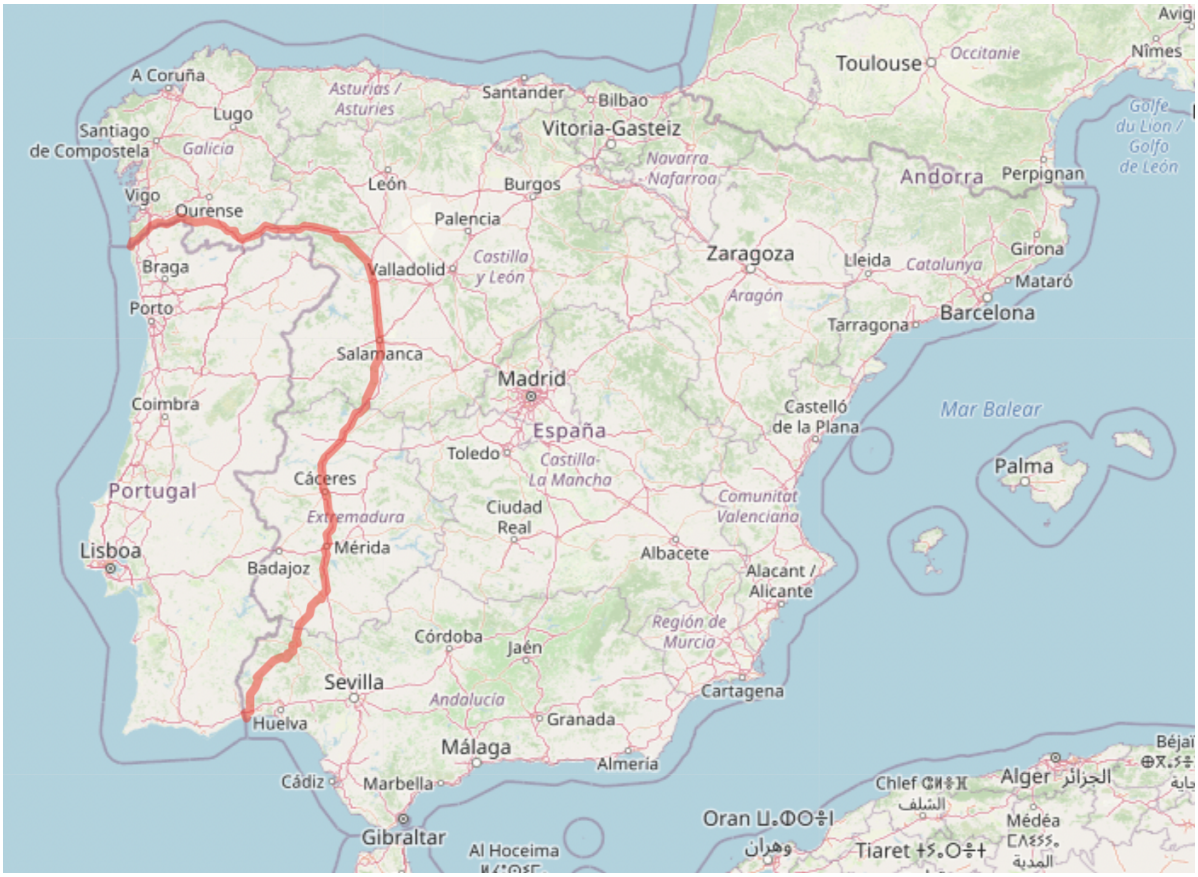

Figure 4: Minimal route found from the outlet of the Minho river in Galicia to the oultet of the Guadiana river in Andalusia.

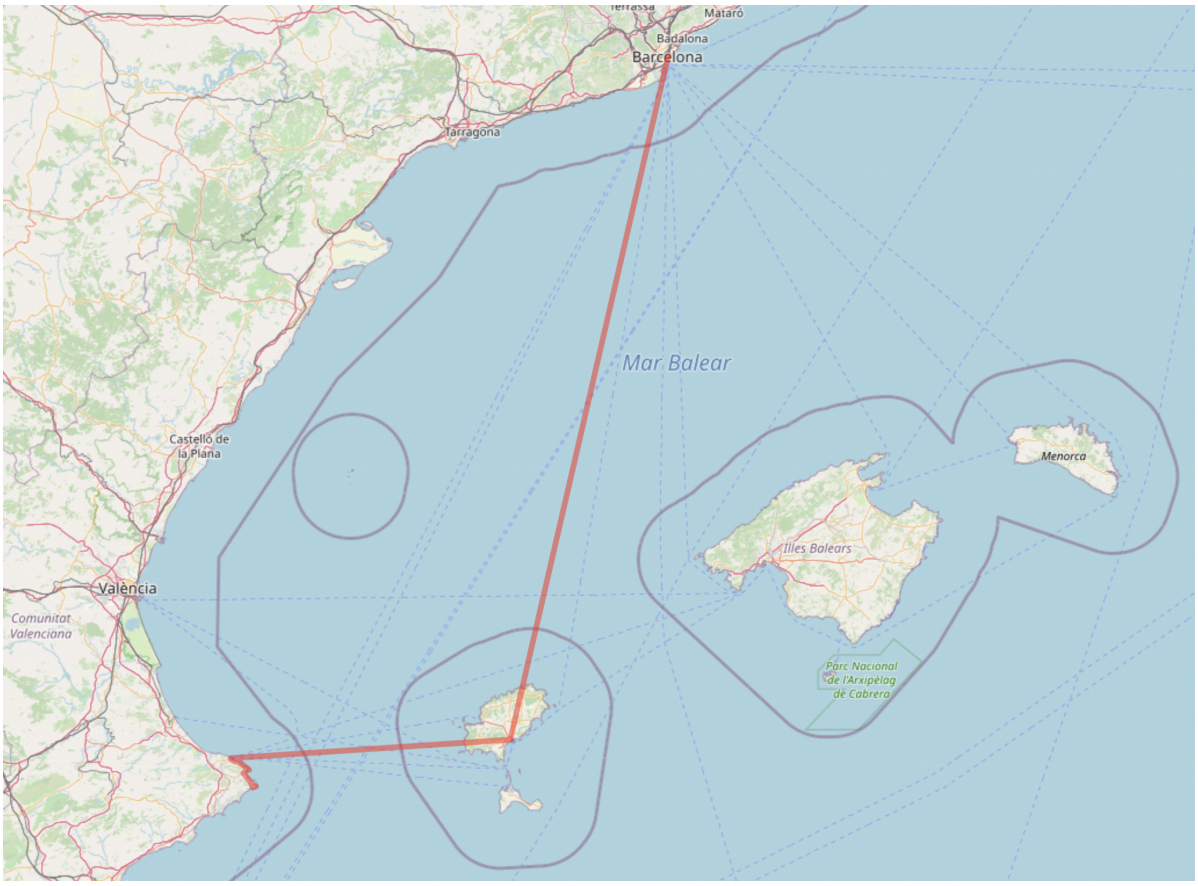

Figure 5: Minimal route found from Barcelona to Cap de la Nau (Cape of the Ship) going through a sea route passing by Ibiza.

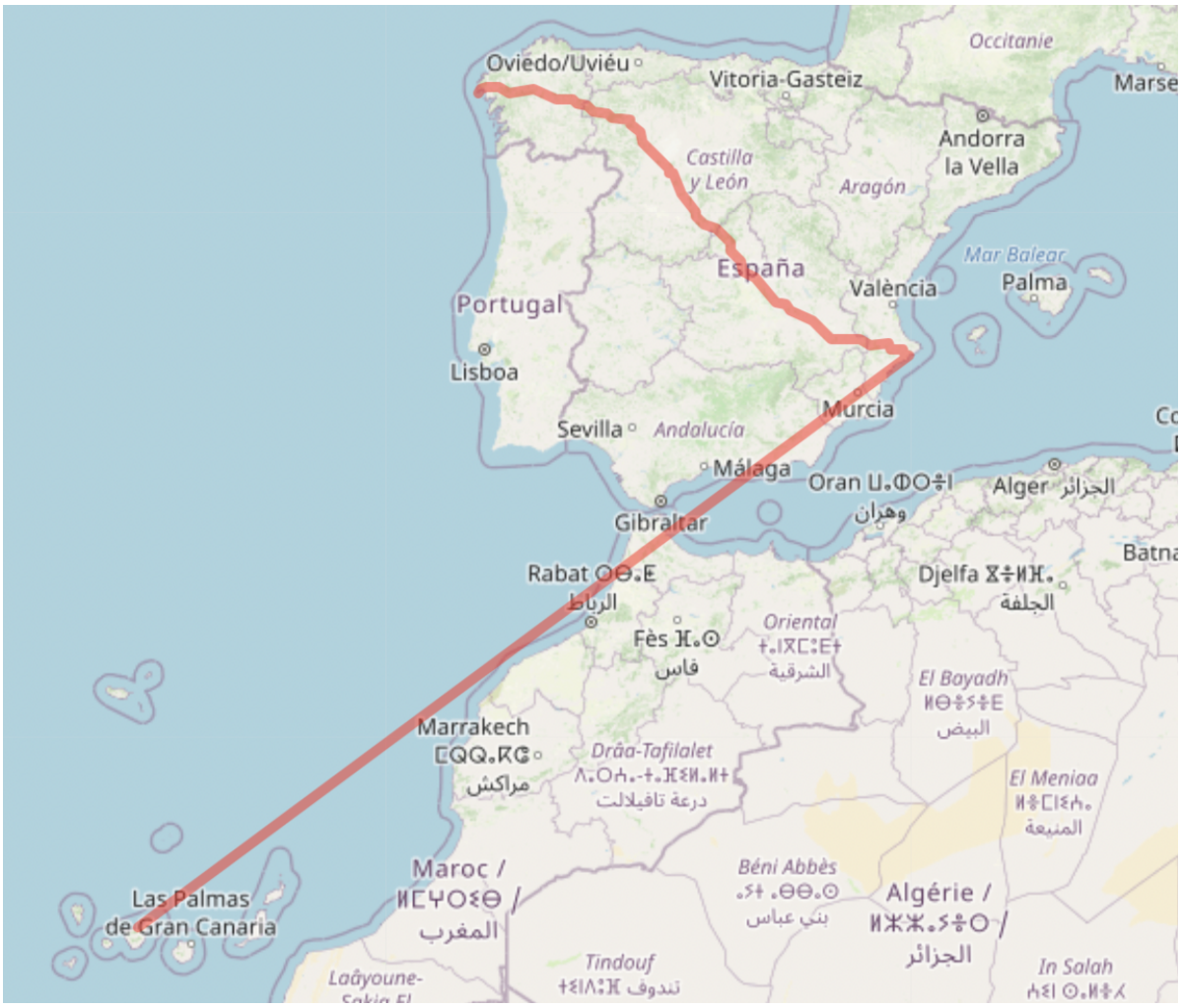

Figure 6: Route found going from Fisterra (Galicia) to Tenerife island.

4 Discussion

As mentioned earlier Astar is often described as the extension of the Dijkstra algorithm. The main difference is the heuristic distance function. The heuristic function helps that the paths followed i.e. the nodes on top of the priority queue are more closer to the destination and therefore the algorithm is faster as less successor nodes have to get visited.

4.1 Possible improvements

A possible way to improve the speed could be to store the weight (distances) of the successors while building the graph instead of calculating it while the algorithm runs.

Another improvement that could be pursued is changing the heuristic function. We used the Haversine function to measure the distance from a successor node to the destination. As already argued in section 2.6 this estimation works just fine as in Spain's mainland we cannot find large areas of "forbidden terrain" like desserts or oceans that cannot be crossed. The algorithm will not find a path between nodes trying to go this direction but still due to the heuristic function some nodes might get visited/preferred that would provide a shorter path if the "forbidden terrain" was cross able. To avoid this problem one could introduce an additional punishment in the heuristic function to avoid big areas without connection and to more easily go around them. Also, we would argue that the Haversine function for our problem is practically just fine, even though the earth is no perfect sphere but an ellipsoid/ spheroid. Therefore, if one wants to become more precise, one could adapt the radius to the region, taking into account that the radius is different in the polar region compared to the equatorial region. To get even more precise but probably also use extensive computational power, which might not be adequate to the problem size in general, one could dynamically adjust the radius using the earth topography i.e. taking elevation data into account.

If we attempted to generate a competitor to google maps, not only finding the shortest but also the fastest path one could punish smaller over bigger paths, take speed limits into account etc. But as mentioned before this might also exceed the problem size and scope.

One major part that was improved throughout the programming of the two algorithms, was switching from a Priority Queue as sorted linked list to a binary heap. The time complexity for adding with priority or decrease priority is $O(logn)$ for the binary heap and $O(n/2)$ for the linked list, where n is the average number of elements in the queue during the algorithm.

Binary search is really efficient when creating the binary graph with a runtime of $O(logn)$ where n denotes the number of nodes. Of course a HashTable with $O(1)$ would improve the runtime but would use up more memory space. As we are dealing with integers as node id, one could try to implement a Y-fast trie which makes use of a smaller HashTable in combination with binary search or fusion trees, but implementing all these concepts would exceed the scope of this report.

4.2 Special cases comments

As can be observed in **Figure 4**, when mapping the shortest path from the Minho river in Galicia to the oultet of the Guadiana river in Andalusia, which both lie very close to the border of Portugal but on opposite ends, we get a path that does not seem the shortest. That is because in out data set the map of Portugal was not included.

In **Figure 5** a path crossing the sea was found. Apparently the nodes Barcelona to Cap de la Nau (Cape of the Ship) and a node on Ibiza island are successors of each other to account for sea routes. If the path could actually be followed is unclear.

When going from Fisterra (Galicia) to Tenerife island, like in **Figure 6**, the path becomes very weird. First we cross the country to the opposite direction and from there, seems to exist a connection node to Tenerife island but the total distance found is infinite. Of course this is not the shortest path, especially if flights were an option. This example and the previously described leaves some questions open of how nodes on islands are connected to the land area. We are aware that the data set might not include all possible connections and also gives no detail about the type of connection i.e road, boat or flight.

4.3 Difference MacOs and Windows

As can be observed in the results the Mac OS system performed way better than the Windows 10. This is mainly due to the M1 processors having a higher clock speed compared to average laptop devices that therefore increase performance tremendously. Another problem we ran into were the different functions that existed for C with a OS system but not with a Windows system, for example the asprinf.h function, which we had to implement by hand. Also when allocating space for a character in Windows one would have to add another +1 for the '\0' at the end of a string, which is not a problem on the Mac system.

Last but not least, when executing a bash file, the Windows system needs different commands. First, when using Windows one has to find the Rstudio.exe file in their Program Files and state it explicitly in the code, rather than just calling Rstudio. Moreover, the command "open" in Windows is "start" to open an .html file.

REFERENCES

[1] Lluís Alseda. *Astar assignment - A routing problem*. URL: http://www.lluis-alseda.cat/MasterOpt/Assignment-AStar-2020.pdf.

[2] Lluís Alseda. *Shortest paths algorithms in weighted graphs*. URL: http://www.lluis-alseda.cat/MasterOpt/RoutingInGraphs.pdf.

[3] Movable Type Scripts. *Calculate distance, bearing and more between Latitude/Longitude points*. URL: https://www.movable-type.co.uk/scripts/latlong.html.