Routing problems:

A* algorithm

Marina Berbel Palomeque (1388498)
Miquel Barcelona (1359817)

January 2020

Contents

1 Introduction

Looking for the shortest route from a point to another has been a problem since human kind have travelled. Trying to reduce days on a horse, or sailing nights, nowadays we can use computers to look for the fastest path. Translating a map to a graph for a mathematical treatment, Dijkstra algorithm can find the shortest route between a point and all others at a high cost. For an efficient, more practical approach an heuristic algorithm is needed; one that, just like a human, looks for a path in the direction of the goal and no other. This is the A* algorithm.

In this assignment, we use real maps made public by Google and try to find routes inside Spain. The objective is to find a shorter route than Google Maps, as their code minimizes time and we want to minimize physical distance. Another restriction is that Google Maps needs to follow roads and traffic, while we will just traverse a graph. Also we want our program to be efficient, not wasting memory and running fast.

The report is structured as follow: in section §2 we describe the A* algorithm in a generic way, with no mention to application or implementation. In section §3 we comment the code written for obtaining a graph from the map and for the A* itself. Here we will discuss the solution proposed for each detail of the algorithm. In section §4 we present the results obtained, the details about the route found and some graphical representations to visualize the output. Finally in section §5 we present our conclusions.

2 Algorithm

The A*, or A Star algorithm, is a variant of Dijkstra, more efficient and with an heuristic component. Given a graph and a function w that returns the cost of the edge between two nodes, it can construct a path from a source to a node. All adjacent nodes to a visited one are put in an open list. At each iteration, the node in this list with the minimum distance is extracted and visited, and put in a closed list at the end. The iterations continue until the goal node is extracted (success) or the open list is empty (an error occurred).

When a node is extracted, the distance from the source to each of its neighbours is computed and if this is less that the one previously achieved (if any), it's used as new distance for the adjacent node, and the extracted node set as its parent. This adjacent node could be in the open list, where we would just update the distance and the parent; in the closed list, so if the distance needs to be updated the node is moved again to the open list; or in an empty list, meaning that the node has never been explored before, so it is moved to the open list and the distance and parent assigned directly.

A clear summary of the algorithm can be seen in the pseudocode of Algorithm 1.

What is different about A* is that the cost to reach a node is not only computed by the w function, but also an heuristic function. The total cost of a node n, that is looked when popping it from the open list, is

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

where $g(n)$ is the actual distance from the source to the node n following the more optimal path found until that moment, and $h(n)$ is the value of the heuristic function for that node.

The heuristic function is problem dependent. It's an estimation of the optimal cost to reach the goal, from

Algorithm 1 A*, s as source node, g as goal node.

```
1: s ← OPEN
2: f(s) = h(s)
3: while OPEN not empty do
4:     u ← pop(OPEN)
5:     if u=g then
6:         break
7:     for v ∈ Adj[u] do
8:         cost = g(u) + w(u, v)
9:         if v ∈ OPEN then
10:            if g(v) < cost then
11:                continue
12:        else if v ∈ CLOSED then
13:            if g(v) < cost then
14:                continue
15:            v ← OPEN
16:        else
17:            v ← OPEN
18:            h(v)
19:        g(v) = cost
20:        parent(v) = u
21:    u ← CLOSED
```

a given node. It is used to guide the search towards those nodes that are nearer the destination, so the number of nodes that need to be visited is reduced. This translates into a reduction of the computation time.

The A* algorithm always finishes in finite graphs, as there exist a limited number of edges and therefore even exploring all them is a finite process. It is also complete, which means that it returns a path, given that there exist one. This is true even for infinite graphs [1]. What we need now is the certain that the returned path is optimal, or at least, *optimal enough*. This is provided by the heuristic function.

An heuristic function is admissible if the prediction of the cost from a given node to the goal is less or equal than the actual optimal distance between them in a graph. An admissible heuristic gives an admissible algorithm, that guarantees that the solution returned is optimal.

An heuristic returning the length of the optimal path guarantees that only nodes in this path are expanded. As we don't know this distance, an approximation is made and therefore more nodes are expanded. The better the heuristic, the faster the algorithm. In [1] proofs can be found about the algorithm expanding only nodes with cost less or equal than the estimation of the optimal distance, and that an heuristic that approximates better the actual optimal distance gives a better algorithm than a less accurate heuristic.

A property the heuristic may have is monotonicity:

$$h(v) \leq h(u) + w(u, v) \tag{2}$$

The Theorem 10 in [1] states that a A* using a monotone heuristic function finds the optimal cost for each node it expands. This means that once a node is in the closed list, it is never reopened. This makes the algorithm faster and much more efficient. The monotonicity can be check numerically in an

implementation of the algorithm, as if the heuristic has this property the extracted costs in the *pop* function should be non decreasing.

3 Implementation

We have coded a program that finds the optimal route, considering physical distance, inside a map of Spain.

The heuristic used measures distance over the surface of a sphere using geodesics. There are in this description two things to justify. In first place, we use the approximation of the Earth as a sphere because we are calculating distances just inside Spain, that is small enough to consider that the radius of the planet does not change. Also with this we are considering that the fastest route between two points follows a geodesic. This is true because Spain does not have any big, non passable hole, or a sea that can stand in the middle of a path. Our map can be seen as a continuous patch in the surface of a sphere.

The code has two well differentiated parts that we will comment in separate subsections. The first is to translate the given csv file into a graph we can store and access. The second is to actually implement A* to find routes inside that graph. This two parts are separated in two main functions corresponding to the codes in `write.c` and `main.c` files. The rest of the enclosed files contain the needed functions and definitions to compile the program and they are explained in the following sections. Both main files can be compiled and executed using the `Makefile` and `chou.sh` files respectively.

As we deal with a very big graph and a lot of memory is needed to store it and deal with it, we take extra care of the use of memory. We might sacrifice a little bit of execution time to save the need of extra variables, as their requirement is quickly incremented by the amount of nodes. For this reason we also have extra precaution with pointers and memory reserved. We make sure every bit assigned is freed at exiting the program, and a memory-tracing program confirms we have no memory leaks.

3.1 Obtaining a graph

The map is given in a csv file, where all nodes are presented along their information (name, coordinates, identification number...), and then sets of ways between the nodes are listed. The lecture and treatment of this file is slow because of its dimension. That is the reason why all information that we need from it is extracted and stored conveniently just once. In order to have access in different executions, all relevant data is stored in a binary file, that is faster to read, and it's organized so the recovery of the graph is easier. The initial csv file is treated in a way that can be divided in three different parts that are explained below.

3.1.1 Reading the csv file

A first lecture of the csv file is needed to know the number of nodes and ways we need to interpret. This lecture is executed by the file `readCsv.c`. Every line is read with the *getline* function and stored in a buffer that reallocates itself if it needs more memory. Then an auxiliary pointer copies the address of the buffer, so we can modify one of them without loosing the head of the memory block assigned. The different fields of each line, separated by the character "|", are read one by one by the *strsep* function.

This functions modifies the pointer passed to it and move it to the beginning of the first element after the first delimiter it can find. That's why the argument passed is the auxiliary pointer, so the buffer reserved is not corrupted. The treatment afterwards is different if we are reading nodes or ways.

Each node is stored in a struct that will be described later, but that in essence contains the identification number of the node, its name, its coordinates and the list of its neighbours. When reading nodes we fill the information that related just to the node, not its adjacents. The field id is converted from string to unsigned long by the *stroul* function. Then, if there is a name, the corresponding pointer in the struct is malloc to the size of that name, the string copied to it, and the null terminator added. Finally we skip the fields we are not interested in, and save the latitude and longitude converting the string to double by *strtod* function.

Then an array of nodes is constructed and the information in each of then filled. To have a graph, only the edges are left.

In the file, the edges are written as lists of nodes that are connected, in one way or both. These lists are tagged as ways. The approach is to read each line and write in the corresponding nodes the next in the list as a successor.

To avoid memory wasting, we increase the size of the successors vector of each node two by two. From the information given, the maximum number of neighbours is 16 and the mean is 2. So we create a vector of integer variables to keep track of the size of each array of successors, and only reallocate it when needed.

In order to know if a way has one direction or both the value is read in the file line and stored as a boolean value in an integer variable. The value assigned is such that when writing an edge, the opposite is also written because a loop is extended by the boolean.

In the ways, the nodes are referred to by id, but we are interested in the index of their position in the array of nodes in order to save execution time. To find an equivalence between both, we use a binary search, taking advantage of the id of the nodes being in an ascending order. The binary search works like a bisection method for a continuous function: it checks if the id that is looking for is inside the interval of ids that are inside two positions in the array. If it is, the interval is halved and then it checks again. If it is not, is the complementary interval the one that is halved. This runs in $\mathcal{O}(logn)$ in the worse case [2], faster than a linear comparison. Also we accelerate the process forcing the more used variables to be in the register memory of the computer.

But when creating an edge we also need to take into account that the file is not perfect, and maybe a way has less than two nodes, or some of them do not exist in the declaration of nodes. This cases are taken into account, using that the binary search returns a negative value if the node is not found in the array, and that the edge is only constructed when there are two nodes identified by index that should be connected.

When this process is done, the array of nodes contains all the information of the graph in the shape of an adjacency list. This information is stored in a specific structure that is explained below. But in order to not read all the file each time we want to try the program, after reading the csv file, the information is written in a binary file.

3.1.2 Writing the binary file

The file `writebin.c` contains the two main functions that corresponds to the writing and the reading of the binary file. As we have said before, once the array of nodes is built, we need to write this information in a binary file. This task is executed in three different steps.

First of all, we write the header of the file that will be used later when we reconstruct the whole information. This header contains the total number of nodes and the total number of successors. This second argument is needed because each node has his own array of successors written as a pointer, so we will be writing just the position in the memory of this array as a member of the structure. This position can change in any execution of the code so we need a way to reconstruct this content. After that, we can write all the nodes and then all the information stored in each node's pointer of successors. In this way, we write the successors of each node in a single block.

Thirdly, as the names are also stored as a pointer of char type, in order to be able later to reconstruct this information we need to write the lengths of the names for every node in our array. This information was not necessary when writing the successors because the node structure has already a member that contains the length of the array of successors. Finally, we just need to write the corresponding names.

After this, we are able to run the file `write.c` that is responsible of executing the first reading of the csv file passed by argument and then writing the correspondent binary file that will be used later.

3.1.3 Reading the binary file

The corresponding code to the reading of the binary file can also be found inside the file `writebin.c`. This function opens it and save the needed memory to store all the information that the binary contains. This means that it reconstructs the entire array of nodes in the same way that we wrote the information before.

The first thing is to read the header: the total number of nodes and the total number of successors. The first argument is used to declare and to malloc the required memory to store the array of nodes and the second one is used to build an auxiliary array. This array will contain all the written successors for the total number of nodes. After that, we use these arrays to store first the nodes and then all the successors. Later, we have to deal with the names of the nodes. For that reason, we read and save all the lengths of each name inside another auxiliary array of integers. This array is used to read and to store directly into the array of nodes each of the written names.

Finally, we need to set the pointers of each node to its own successors that are stored in the auxiliary array. Considering this purpose, we use the special variable of the node structure that gives us the total number of successors for the given node. This variable is already read and saved in the array of nodes. With this information, we just need to use pointer arithmetic to make the pointers of the structure point to the appropriate part of the auxiliary array.

This function is run in the `main.c` file that also contains the A Star algorithm as this execution is needed every time we want to try the algorithm.

3.2 Coding A*

In this section we deal with the part of the program that corresponds with the implementation of the A Star algorithm itself. First of all, before explaining the implementation of the problem, we need to introduce a set of definitions and structures that are declared and used to run the A Star in a fast and efficient way.

3.2.1 Structures and definitions

All the structures and definitions that are implemented through the main program are declared in the `auxFun.h` file. As we have mentioned before, the first one of them is the node structure. This structure is also needed during the first part of the program corresponding to the first reading and the writing of the binary. This structure contains seven members: the id of the node, the name, the latitude and the longitude, the total number of successors and an array with all of them and finally an element of the structure called list that will be explained later.

```
typedef struct {
    unsigned long id;
    char* name;
    double lat, lon;
    unsigned short nsucc;
    unsigned long *successors;
    list* lista;
} node;
```

The second structure is called AStarStatus and contains all the information related with the A Star algorithm for each one of the nodes. These are the actual value of the g function, the heuristic distance to the goal node, the index in the array of nodes of its parent node set by the algorithm and a type of char defined as Queue that says in which kind of list is stored the given node.

```
typedef struct {
    double g, h;
    unsigned long parent;
    Queue where;
} AStarStatus;
```

The definition of the char called Queue is used together with the definition of the enumeration data type called whichQueue. These two are used to make easier the way to identify in which queue of the Algorithm 1 is set each node.

```
typedef char Queue;

enum whichQueue {NONE, OPEN, CLOSED};
```

Then, in order to treat the part of the Algorithm implementation corresponding to the concept of the open

list we need to define the structure called list that is also used in the node structure. This open list, as will be explained in the following section, works as a priority queue. In this way, the list structure corresponds to one element of a linked list that defines a priority queue. For that reason, the structure contains as a member the value of the f function, which defines the priority feature. It also has two pointers of the same list structure that point to the next and the previous element in the list. In addition, we store the index of the given node in the total array to make easier the identification.

```
typedef struct List {
    double f;
    unsigned long index;
    struct List *next, *prev;
} list;
```

Finally, with the goal of making easier to change between heuristic function in equation 1, we define a new type called *heur* that represents a function with two different node structures as argument. This type is used to called the A Star algorithm using different heuristic functions.

```
typedef double (*heur)(node u, node s);
```

3.2.2 Priority queue

As we have explained in section §2, the algorithm needs to extract the element of a open list with the minimum value of the f function. For that reason we have defined the list structure. This list works as a priority queue implemented with a linked list. Thereby, we have to choose the proper way to order the elements of the list. This can be done during the pop (extraction) or during the push (introduction) of each of the visited nodes. However, as we have said before, there are cases in which the distance of a visited node set in the open list can need to be updated. These are the occasions when it is not possible to sort the list during the push because a node may need to be reordered.

However, we use a function that deletes the node from the list every time that we find a shorter distance for a node set in the open list. This can be done without executing any loop by using the previous and the next element from the list structure to update the corresponding list elements from the list structure. Moreover, this allows us to make the sort during the push. In this case, the Algorithm 1 will always enqueue the v node after setting its parent. During the sort it is not possible to avoid traveling through all the nodes of the list looking for the first element with a higher distance. When this is found we have to update the pointer list elements of the corresponding nodes.

Finally, as the list is already sorted we just need to pop the first element and update the list pointers of the second element from the list if it exists.

In addition, there is another push function that does not order the elements. This is only used to represent the path of the optimal found solution starting from the goal source and travelling through all the parents nodes. All this functions that deal with the list elements are declared in the `astar.c` file.

3.2.3 Cost of a node and heuristic functions

The priority queue explained above uses the value of the f function to sort its elements. This one is computed for the n node using the following expression,

$$f(n) = (1 - w) \cdot g(n) + w \cdot h(n) \cdot \left(1 + \epsilon \left(1 - \frac{d(n)}{N}\right)\right) \tag{3}$$

where $0 \leq w \leq 1$ is a constant called weight, $\epsilon \geq 0$ is another constant that guarantees that the found path will not be worse than the optimal path by a factor of $1 + \epsilon$ when $w = 1/2$, $d(n)$ is the depth of the given node and N is the desired depth of the goal node [1].

This is the easiest way to define the f function that allows us to modify the values of w and ϵ to try different versions of the algorithm looking for the most efficient. By default, we set $\epsilon = 0$ and $w = 1/2$ which means that we recover the definition introduced in (1), as a factor multiplying both term results in the same order of the costs, just with different numbers.

By keeping $\epsilon = 0$ and varying w between 0 and 1 one can observe how the algorithm behaves when the weight of the heuristic function changes. This modification makes the algorithm to explore more nodes or to change its performance. In this sense, when w is set to 0, the A Star algorithm becomes the Dijkstra method. The admissibility of the method is guaranteed when $0 \leq w \leq 1/2$ and this property may be lost in the remaining interval [1]. This will be shown in the results section.

In the same way, when we set $w = 1/2$ and $\epsilon \geq 0$, we will be applying the dynamical weight method. Which means that the weight of h will be less heavy as we will be closer to the goal node. Again, the performance of this method will be explained in the results section.

Going back to equation (3), we have defined 4 different heuristic functions with the same structure as the type heur presented above. The first one corresponds to the Dijsktra algorithm which is basically a function that returns always a 0. The remaining three are the following and are defined using [3] and assuming that R is the radius of the Earth and the n node can be entirely defined using its latitude φ_n and longitude λ_n.

Haversine distance The formula 4 computes the great circle distance between the node 1 and the node 2 over the earth's surface.

$$d = 2R \left\{ \arcsin \left(\sqrt{\left(\sin \frac{\varphi_2 - \varphi_1}{2}\right)^2 + \cos \varphi_1 \cos \varphi_2 \left(\sin \frac{\lambda_2 - \lambda_1}{2}\right)^2} \right) \right\}. \tag{4}$$

Spherical Law of Cosines The well known spherical law of cosines,

$$\cos c = \cos a \cos b + \sin a \sin b \cos C$$

can be also used when the estimated distance is as small as few metres on the earth's surface. In this case, the distance will be

$$d = R \arccos \left(\sin \varphi_1 \sin \varphi_2 + \cos \varphi_1 \cos \varphi_1 \cos \left(\lambda_2 - \lambda_1\right)\right). \tag{5}$$

Equirectangular approximation In this formula, again in the case of small distances, the Pythagora's theorem is applied:

$$d = R\sqrt{\left((\lambda_2 - \lambda_1)\cos\frac{\varphi_2 + \varphi_1}{2}\right)^2 + (\varphi_2 - \varphi_1))^2}. \tag{6}$$

This formula is also used to compute the distance $w(u, v)$ of the Algorithm 1 between the node u and v due to its high performance.

These four heuristic formulas are defined in the file `auxFun.c`.

3.2.4 A* implementation

After all of these definitions we are ready to execute the algorithm itself. In this case, the function responsible of its execution receives as argument the array of nodes and AStarStatus, the total number of nodes, the id of the goal and source node and the heuristic function that is going to be applied.

Before starting the algorithm, the function needs to initialize some variables. Firstly, an array of integers is declared and it will be used to store the depth of each of the visited nodes in order to compute the value of $d(n)$ in the equation (3). This array is initially set to infinity for every node except the source which is set to 0 by definition. Then, the binary search is called to find the index in the array of nodes of the 2 nodes. If any of the calls returns -1, then the function ends by returning a -1. After that, the status of the source node is initialized and the node is pushed into the open list. The status of the rest of the nodes is set to NONE and the element of the list from the node structure is defined as null.

Finally, two variables are declared: the value of f from the previous explored node and a boolean variable called monotone and initialized to 1. These two are used during the algorithm to check if the used heuristic function is monotone by comparing the value of f of the current explored node with the previous one. If there is any case where the previous value is bigger than the current one, then the heuristic function is not monotone and the boolean variable is updated to 0. At the end of the algorithm, if this variable is still 1, we will have proved that the heuristic is monotone.

After all of this, the Algorithm 1 is run. We start by extracting the first element of the open list as this is already sorted. We check the monotonicity of the heuristic function and if the node is the goal one, we break the while loop and return the index of the goal node in the array of nodes. If it is not the case, we have to explore all of the successors of the extracted node. We compute the depth of the successor if the node has not been already visited and then we calculate the distance between the parent and the actual node. In this case, we need to check if the computed distance between the parent u and its child v is not 0 because there are nodes in the csv file with the same longitude and latitude. If this verification is not done, then the algorithm wont find the optimal path. Finally, we compute the actual distance, $g + w(u, v)$, and if this is less than the computed before the node will be push into the open list and its g and parent will be updated to the current values. If the node was previously in the open list, then the node needs to be deleted from the list as we said before and re-pushed.

At the end of this function, the index of the last node, the goal, is returned. This implementation is computed in the `main.c` file. This file receives by parameter the binary file and optionally the value of w and ϵ if we want to try different weighted versions. Then it reads the binary file. Once the binary file is read, the program needs to receive by screen the id of the source and the goal node to be able to execute the just mentioned function. Here we present an example of execution for our particular problem.

```
echo 771979683 429854583 | ./astar spain.bin .5=weight .0=epsilon
```

3.2.5 Printing the path

Another function included in the `astar.c` file corresponds to the printing of the optimal solution found by the algorithm. In this case, we create another linked list starting by the goal node and traveling through all the parents. All of them are introduced by a new push function that sets the node at the beginning of this list. So at the end of the loop this will have the source node at the first position. So we just need to travel again through the list but now starting from beginning and using the next list pointer.

Besides some useful information about the performance of the algorithm that are introduced by a #, the function prints for each node from the path the current information:

```
Id | Distance | Name | Latitude | Longitude
```

3.2.6 Visualization of the path

Finally, a new file called `map.py` is written in Python language and it allows us to visualize in a map the optimal solution that is printed and save it in a file following the above structure.

This file uses the library `gmplot` to generate a Google map with the given route by using two arrays with the latitudes and the longitudes for each of the nodes included in the solution. This program needs to receive as argument a file with the explained structure where the third and fourth field are the latitude and longitude. After this, an html file is generated with the correspondent route and can be loaded with a common browser to see the route in an interactive map [4].

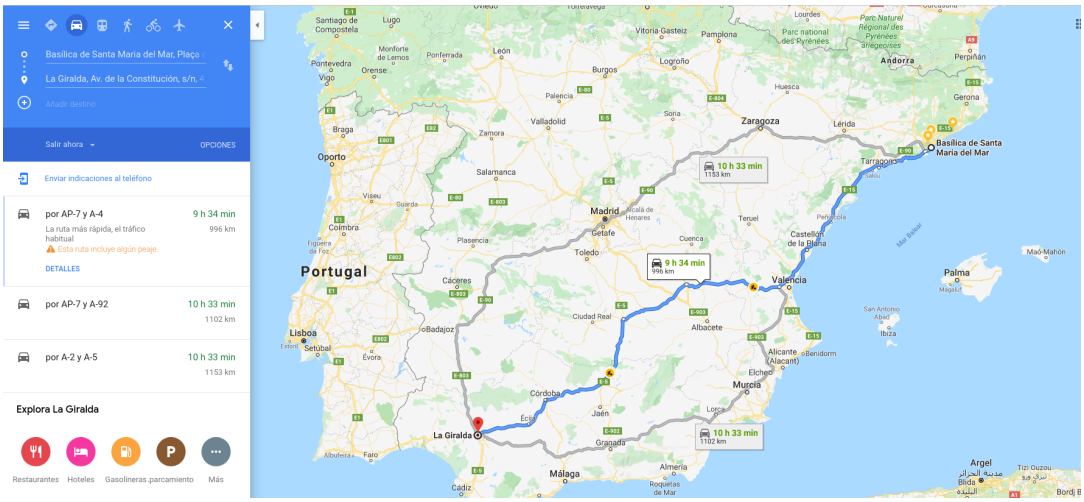
4 Results

In the following section we present the obtained results of the program explained above related with the routing problem of finding the optimal path between Santa Maria del Mar of Barcelona and la Giralda of Sevilla. These contains information about the optimal path found by the algorithm and also about the performance of the code.

The first thing that we need to do is to execute the `write.c` file and check how long takes the generation of the binary file. The obtained time was 30.92 seconds which is a good result considering the huge size of the raw file and the generated by the program, around 1.5 GB.

After that, we execute the `main.c` file to check how fast are the reading of the binary file and the execution of the algorithm itself and also its accuracy. Related with the first purpose, we have obtained that the binary reading task takes 1.35 seconds which is also an admissible result considering again the size of the file.

Regarding the second part of the main function, the performance of the A Star algorithm, in order to check how good is our solution we need to compare it with the one found by Google. On one hand,



(a) Optimal path by car.

(b) Optimal path on foot.

Figure 1: Google optimal paths for the given routing problem between Santa Maria del Mar and La Giralda.

Google gives a route by car which distance is about 996 kilometres. And, on the other hand, it also gives another route on foot with a distance about 980 kilometres. These two routes are presented in Figure 1.

On our case, by setting the w and ϵ to its default values and trying with the four discussed heuristic functions, we have obtained the same optimal path and distance corresponding to

$$D = 959883.540597\, m.$$

This means that the three heuristic functions (here we are not counting the one returning zero) are admissible, as the solution returned is optimal.

As we can see, this results improves the Google path by approximately 20 kilometres in the best case. As we have said earlier, this is due to the way that Google optimizes the routes: by minimizing the time. The obtained route can be visualized and compare it with the Google ones using the Figure 2. This figure, as mentioned before, has been generated with the `map.py` file and shows that the optimal route

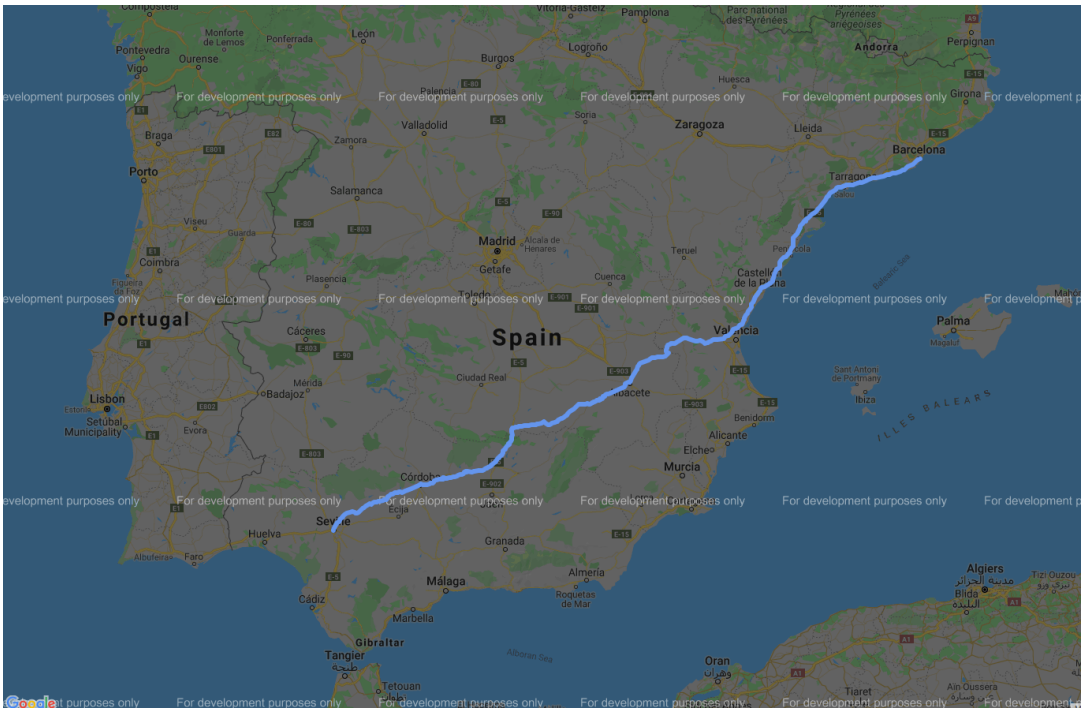


Figure 2: Optimal path found by the A* algorithm.

found by our algorithm is very similar to the one found by Google on foot.

Although the optimal distance found by each of the heuristic functions is the same, the computational time spent by executing the algorithm is different. We show the results in Figure 3. As we can see, the haversine, the spherical law of cosines and the equirectangular approximation presents similar results and the small differences can be caused by some slight fluctuations: after all, we are measuring elapsed time, not CPU time, so other processes of the computer may do the time vary. However, as expected, the Dijkstra algorithm presents a time execution higher than the rest. This is because the Dijkstra algorithm needs to expand more nodes.

Finally, we explore how to improve the performance by modifying the weight of the f function. First

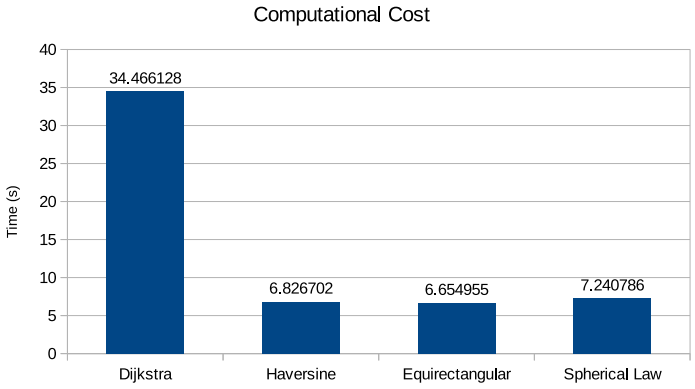


Figure 3: Computational time comparison using different heuristic functions. We can say A* is taking around 7 seconds.

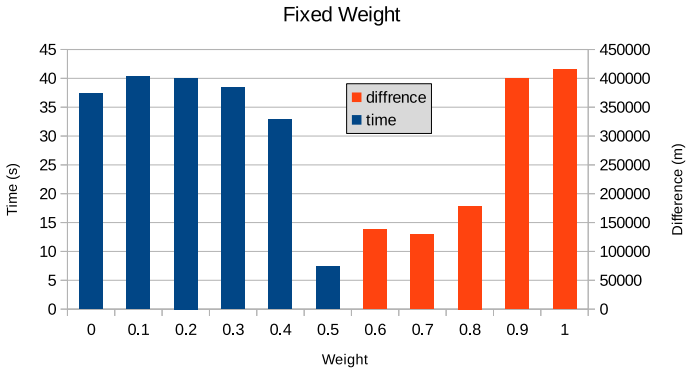


Figure 4: Performance results corresponding to computational time and difference distance with the optimal one for different fixed weights.

of all, by setting again ϵ to 0, we vary w between 0 and 1 and compute the obtained distance and the computational time. As we can see in Figure 4, between 0 and 1/2, the weighted heuristic is still admissible as the optimal path obtained by the new version of the algorithm is the same as before. In this case, only the computational time increases in the same way that the algorithm needs to explore more nodes. Nevertheless, for values larger than 1/2, the algorithm looses its admissibility as the difference between the found path and the optimal is larger than 0. In these cases, despite the decrease in accuracy, we obtained an improvement on the computational time which is now less than a second.

In order to solve this problem we deal with the dynamic weighted version of the problem by setting ϵ values different than 0. In this case, we choose again $w = 1/2$ and we modify the value of ϵ between 0 and 1 because we do not want to loose accuracy. The obtained results are observed and in this occasion, the distance and the computational time are the same as the first version of the code. So the dynamical weighted is not giving us any improvement on the performance and it is wasting some memory by saving the depth array.

5 Conclusions

In this project we have developed a version of the A Star Algorithm to solve route problems. In this particular case we have find the optimal path between the La Basilica de Santa Maria del Mar from Barcelona to La Giralda from Sevilla.

Our program has been divided in two different main functions due to the huge size of the data file that needs to be read. In this way, our main purpose is to optimize the computational time. At the end, the A Star Algorithm only takes almost 8 seconds to read and execute the algorithm which is a good and a fast performance.

In addition, we have checked some different heuristic functions looking for the best result but we have obtained the same distance and the same computational time. However, we could check how the Dijkstra Algorithm despite giving the same result looks in a worse performance than the heuristic cases. Likewise, we have tried with different weighted versions for the computation of the distance looking for better performances but without good results.

In any case, our program has been able to find a solution that improves the path given by Google which is consistent with our initial expectations.

References

[1] Judea Pearl; "Heuristics: Intelligent Search Strategies for Computer Problem Solving"

[2] https://en.wikipedia.org/wiki/Binary_search_algorithm

[3] https://www.movable-type.co.uk/scripts/latlong.html

[4] https://www.geeksforgeeks.org/python-plotting-google-map-using-gmplot-package/