AStar Assignment – A Routing Problem

Pablo Gallego Adrián
Elena Fraile Valdés

December 12, 2021

1 Introduction

The main objective of this assignment is to compute an optimal path which minimizes the distance from *Basílica de Santa María del Mar* (Plaça de Santa María) in Barcelona to the *Giralda* (Calle Mateos Gago) in Seville, by using the AStar algorithm.

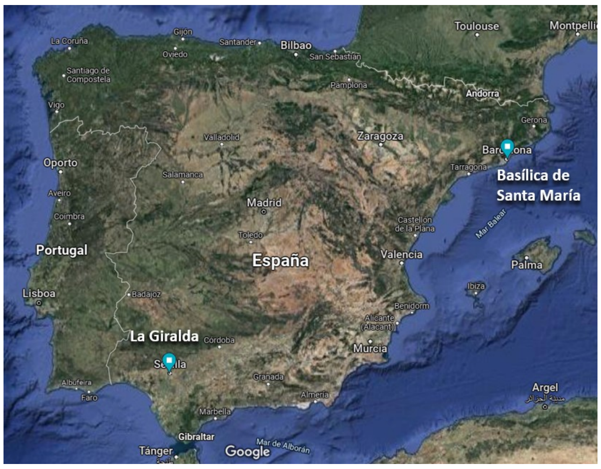


Figure 1: Map showing the the target node (*Basílica de Santa María del Mar*) and the goal node (*Giralda*)

AStar is an informed search algorithm, that is, it uses information about path cost and heuristics to find the optimal path between a starting node and a goal node. The path founded by AStar has the smallest cost (in this case, least distance travelled). In subsection 3.1, a more detailed explanation about AStar is provided. In order to solve this routing problem, we have written two C programs:

- The first program, called *AStar_write*, reads the csv file where all the nodes and ways are stored, and builds the graph. This same program stores the graph in a *binary file*.
- The second program, called *AStar_heap*, is able to read this binary file, thus getting the map already in graph form. Then, it asks for the start and goal nodes and performs AStar algorithm, which has been implemented using a *binary_heap*, and returns the optimal path between this two nodes.

In order to run the whole program, we advise the lector to read section 5.

2 First program: AStar_write

Before understanding the code of this part, let's take a look into the CSV, in order to organize the main objectives that we want *AStar_write* to achieve.

2.1 The map

All the map is contained in the a csv file called *Spain.csv* which our program has to read. In this file, there are three kind of fields: nodes, ways and relations; but we will be focusing on the first two:

- **Nodes**:
  Nodes specify a single point in the map. In fig. 2.A, the structure of nodes' field can be seen.

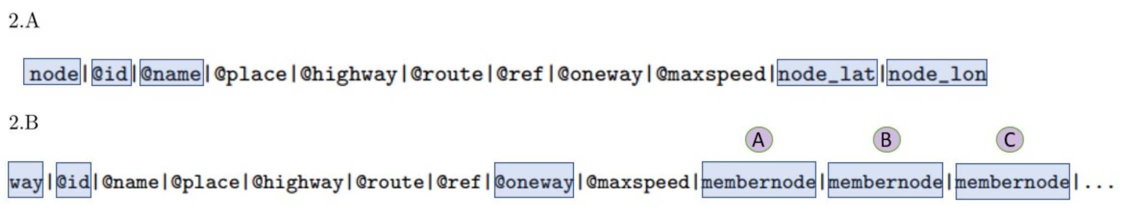

2.A
node|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|node_lat|node_lon
2.B
A
B
C
way|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|membernode|membernode|membernode|...

Figure 2: **A** shows the general structure for nodes field. **B** shows the general structure for ways field. In both cases, the main tokens of interest have been highlighted with a blue box.

In order to store the nodes' information, it is useful to create a structure like the following:

```
typedef struct a_node {
    unsigned long id; // Node identification
    char *name; // Node name
    double lat, lon; // Node position
    unsigned long *successors;
} node;
```

Figure 3: Node struct used in *Astar_write*

Therefore, one of the objectives of our program is to creat a node's vector in which we can store, for each node, its id, name, latitude, longitude, the number of successors and who are those succesors. In addition, we will be refering to each node throughtout the program by its index in this vector node, and not by its id (beacuse it is too large).
Now, the question is: how are we able to know each nodes' successors? Is at this point where ways become important.

- **Ways**:
  A way is a list of nodes and specify the relations between them, just like edges in a graph. The main structure of a way field can be seen in 2B.

  As shown in this figure, for each way line, there is a certain number of nodes A, B, C, etc., which are connected (A with B, B with C, etc.). Also, we need to take into account the *@oneway* field, which takes to possible values: *oneway* or *empty*. In the first case, *oneway* specifies that A is linked to B, but B is not linked to A. In the second case, A is linked to B and B is also linked to A.
  Therefore, another objective of this first program is to read all ways' lines, and set arrows between the nodes that are connected, constructing he graph this way.

In summary, this first program is structured following the nexts tasks:

- 1) Create a function that is able to count the number of nodes and the number of ways, in order to know how many memory we need to allocate.
- 2) Read nodes' lines from the file. In this process, store each node's id, name, latitude and longitude in the vector of node structures.
- 3) Read ways' lines from the file. In this process, set arrows between nodes that are connected. This means that for each node, we should save its number of successors (inside node[ ].nsucc) and what is the index of those successors in the vector of node structures. These indexes should be stored inside node[ ].successors.
- 4) Be able to, somehow, check that tasks 2) and 3) have been computed successfully.
- 5) Write a binary file that stores the graph, in order to save time when dealing with it.

In the next section, we are going to take a short look into our code, highlighting the most important points in it.

2.2 The code

Firstly, after declaring and opening the file in an usual way, our code calls function ***counter***.
This function iterates over the lines of the csv and counts the number of nodes and ways. In *counter*, two important functions are introduced: first, the getline function, which is the best method for reading lines of text: geline reads an entire line from a stream (our csv file), up to and including the next newline character, and stores it inside a char variable *line, automatically enlarging or reducing the block of memory needed to store the read line.
The second function is strsep, which splits the line when finding a '|'. While this first token is equal to 'node', counter adds one to the number of nodes, and same happens with ways.
After compiling Astar_write using -Ofast comand, function *Counter* successfully counts the number of nodes (23895681) and number of ways (14173639) in a total time of 8.77 CPU seconds.

Secondly, after allocating memory for the struct vector of nodes, we call ***Constructing_vector_nodes***. This function iterates over nodes' lines, and for each node, stores its id, name, latitude and longitude inside the vector of node structs.

The main body of this function works as follows:

1. While our counter is less than number of nodes, and while there is still file to read, do:

```
while (linecounter < nnodes && getline(&line, &max_len, stream) != -1) {
```

2. While there is line to split, split it in tokens, using strsep.

```
while((token = strsep(&line, "|")) != NULL){
```

3. Then, we use an int variable called n_field to identify in which token we are placed:

   If n_field equals 1, we are placed over id field, then store it as an unsigned long.
   If n_field equals 2, we are placed over name field, then allocate memory for the name and store it.
   If n_field equals 9, we are placed over latitude field, then store it as a double.
   If n_field equals 10, we are placed over longitude field, then store it as a double.

```
if (n_field == 1) nodes[i].id = strtoul(token, NULL, 10); // Id
if (n_field == 2){ // Name
    nodes[i].name = (char *) malloc((strlen(token) + 1) * sizeof(char);
    strcpy(nodes[i].name, token);
    }
if (n_field == 9) nodes[i].lat = atof(token); // Latitude
if (n_field == 10) nodes[i].lon = atof(token); // Longitude
n_field++;
```

Once again, compailing using -Ofast, *Constructing_vector_nodes* takes a total time 19.51 CPU seconds to run.

Thirdly, function ***Create_arrows*** is called. This function iterates over way lines, and for each node living in a way, its successors (named by its position in the node struct of vectors) are stored.

1. Create_arrows iterates over all the way lines. For each way, while there is line to split:

   2. Save *oneway* field, in order to know how to place arrows between nodes.

```
bool oneway = false;
while ((token = strsep(&line, "|")) != NULL) {
    if (n_field == 7 && strcmp(token, "oneway") == 0)
    oneway = true;
```

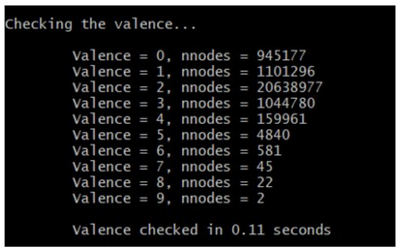
   3. For the first node in the way, let's say A, call ***binarysearch*** function. If A exists in the node struct of vectors, that is, if *binarysearch* returns true, then save its position. Then, set A to be previous node, and skip to the present one, let's say B.
   4. As for the first node, call *binarysearch*. Then, having both A and B positions, call ***NodesConnect*** function, in order to connect A $\rightarrow$ B. The procedure in *NodesConnect* is as follows:
      If A is already connected to B, return.
      If A has no successors, allocate memory for one successor (B).
      Else A has already successors, then reallocate space in order to add one more.
      Last, store B's position inside A.successors, and increment A.nsucc by one.

5. Finally, if *@oneway* is empty, then also connect B → A.
6. Then, set B to be a previous and repeat the process.

```
else if (n_field > 8) { // Connect the nodes
    if (binarysearch(strtoul(token, NULL, 10), nodes, nnodes, &present_pos)) { //
     if the node exists in the vector
    if (previous_pos != ULONG_MAX) {// If there is a previous
    NodesConnect(previous_pos, present_pos, nodes); //always connect A --> B
        if (!oneway) NodesConnect(present_pos, previous_pos, nodes); // if no
    oneway then also B --> A
        }
        previous_pos = present_pos; // Update
    }
}
```

Fourthly, we call ***Check_valence*** function. The purpose of this function is to simply check if we have correctly constructed the graph. In the assigment, a table showing the number of nodes having a certain valence was provided, so we wanted to recreate this table. As can be seen in fig. 4, this table was correcly reproduced, and took only 0.11 seconds.

Valence	# of nodes
0	945177
1	1101296
2	20638977
3	1044780
4	159961
5	4840
6	581
7	45
8	22
9	2



```
Checking the valence...

        Valence = 0, nnodes = 945177
        Valence = 1, nnodes = 1101296
        Valence = 2, nnodes = 20638977
        Valence = 3, nnodes = 1044780
        Valence = 4, nnodes = 159961
        Valence = 5, nnodes = 4840
        Valence = 6, nnodes = 581
        Valence = 7, nnodes = 45
        Valence = 8, nnodes = 22
        Valence = 9, nnodes = 2

        Valence checked in 0.11 seconds
```

Figure 4: On the left, the valence table provided in the assigment. On the right, our result after calling *Check_valence* function.

Finally, our program *AStar_write* ends by calling the function ***WriteBinaryFile***, whose code has been provided at the end of the assignment. The main purpose of this function is to store the graph in a binary file, that is, as a series of bits (zeros and ones). The main advantage is to save time when reading the graph, because binary formats are easily to access (in terms of speed) and also use much less memory.

The whole first part took a total of 41.77 CPU seconds, and the function who took the most time to run was *Constructing_vector_nodes*, with a total of 19.53 seconds, which represents a 46.8 % of the total.

3 Second program: AStar_heap

In this section we present our code to perform AStar and our results, but before that, we should understand how AStar works.

3.1 AStar algorithm

AStar algorithm is a graph traversal and path search algorithm for solving the routing problem: starting from a specific target node of a graph, it aims to find a path to a given goal node having the smallest cost (least distance travelled, shortest time, etc.). A* is one of the most successful search algorithms, as it is able to achieve optimality and completeness.

The problem with other traversal techniques such as Dijkstra, is that they only take into account the cost from the target node to the current one, without paying any attention to what direction they are going. For this reason, it is quite intuitive that a good informed search algorithm should take into account both factors, the cost of the already travelled path but also of the path that remains to be covered, and that is exactly what A* does.
Specifically, A* selects the path that minimizes $f(v) = g(v) + h(v)$ where v is the next node on the path, $g(v)$ is the cost of the path from the starting node to v, and $h(v)$ is a heuristic function that estimates the cost of the cheapest path from v to the goal.

The function $g(v)$ computes the shortest distance from the starting node to the current one, but since this distance is not well defined, we will explore different ways of doing it in 3.3. The choice of the heuristic function depends on the problem, and for this assignment, this choice will also be discussed in subsection 3.3.

3.1.1 Code for A*

Firstly, the program *AStar_heap* starts reading the binary file that has been created in the previous program. The code for reading the binary file was also provided in the assignment, and took 1.31 seconds to run.

Moving to A*, to implement the algorithm, we have used a binary heap, which is explained in 3.2. This binary heap acts as an Open Queue. In each step of the algorithm, the node with the lowest f function is removed from this queue, and its neighbours are added, once its f and g values have been updated. The algorithm ends when the removed node is the goal node. In addition, to control the information of each node, we have created the following struct:

```c
typedef struct a_node {
    double g,h;
    bool isOpen;
    node *my_node;
    struct a_node *parent;
} a_node;
```

Figure 5: Struct created in *AStar_heap*

For each node, this struct stores its cost (g) and its heuristic value (h); a bool *isOpen*, which is true if the node has already been visited; a pointer **my_node* which points to the adress memory where the current node's id, latitude, longitude, etc. are stored (see fig. 3); and the same *a_node* struct for the current node's parent.

To find the optimal path we have created a function called ***AStar***, which takes the graph, the total number of nodes, the position of the starting and the goal node; ** result, where the optimal path is going to be stored, and finally the desired heuristic function. A* computes the optimal path and returns a bool, being true if a solution is found. The pseudocode for this function is as follows:

1. Initialize the open list.
2. For all the nodes in the graph:

```
set isOpen = false;
set g, h = inf;
set parent = NULL;
set *my_node to point correctly
```

3. Put the starting node on the open list. Compute h and g (which is 0) for the starting node.
4. Pop the current node off the open list (which firstly is the starting node) and store its position inside a variable called current. This conditions are inside the following while loop:

```c
while(dequeue(open, &to_deq_curr) && (current = (a_node *) to_deq_curr) != goal){
```

Basically, this loop works while the opened list is not empty: condition one returns true while the process of dequeue worked, and condition two returns true while the dequeued node is not the goal. For the node inside current:

5. For each of its successors:

    a) Get a_node struct for the current successor.
    b) Compute its cost g:
    successor.g = current.g + distance between successor and current.

    i) If the computed g for the successor is bigger than the one he already had (condition 1), or if the successor is already in the open list (condition 2) then skip this successor and move to the next one.

```c
if (g >= successor -> g || successor -> isOpen) continue;
```

    ii) If we haven't computed yet the heuristic cost, then do it, using the corresponding heuristic function:

```c
if (successor -> h == DBL_MAX) successor -> h = fdistance(successor -> my_node
    , goal -> my_node);
```

iii) Set the computed g to be the cost of the actual successor.
iv) Set current node to be the parent of the actual successor.
v) Add actual successor to the open list and end for loop. Then repeat the whole process.

As said before, A* ends when the dequeued node is the goal node. For this function, the running time depends on the chosen heuristic function, as we will see in 3.3.

3.2 Binary heap

The performance of the A* algorithm is strongly dependent on the chosen algorithm for the priority queue. One approach could be using a linear queue, in which each time we insert an element we compare it sequentially with each element until we find its position.
This, however, is very inefficient cause the time complexity of inserting an element is O(n), where n is the number of elements in the queue. There are different algorithms we can apply to a priority queue. One of the most efficient ones is the binary heap, which has time complexity O(log(n)).

A binary heap is a structure that organizes the data in the form of a binary tree with 2 constrains or properties:

- **Shape property**: it must be complete, what means that all levels of the tree, except possibly the last one (deepest) are fully filled; and it also must be balance: the number of insertions on the left must be equal or greater by just one than the number of insertions of the right.
- **Heap property**: every element on the tree has a priority and the priority of an element must be always bigger on equal to its parent's priority, causing the root node to be the one with the smallest priority. To keep these two properties, we must take them into account every time we enqueue or dequeue an element. In return, the number of comparations we must do is much smaller than in a linear queue.

Every element of the heap is a struct that contains a float with its priority, a pointer with the address of the data we want to store on that node, two pointers with the directions of its left and right descendants and some integers that keep the count of how many direct descendants the element has and the number of total descendants to its left and right.

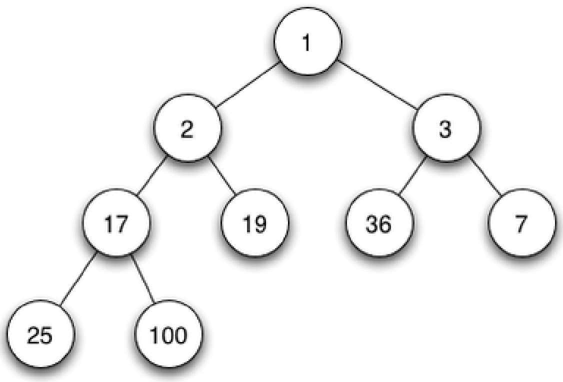


Figure 6: Structure of a binary heap.

The queue consists of a struct with the address of the root element and a variable to store the size (total number of elements stored) of the queue. The two main functions that it uses are the enqueue and dequeue functions. We will now describe the procedure of both functions:

- *Enqueue*
    1. To preserve the shape, we look for a new position on the last level that maintains the shape constrain.
    2. Insert the new element on that new position and update the information of the elements above about their number of descendants on each side.
    3. To preserve the heap (priority) property, from the new node we compare the every node with its parent until we reach the root, and exchange them if they are violating the hierarchy.
    4. We update the variable that contains the size of the queue.

- *Dequeue*
  1. If the enqueue has been done correctly, the root node will be the one with the smallest priority, so we store the root element data on a variable that has been introduced to the dequeue function and we delete the root node.
  2. Now, to preserve the shape, we take one element of the last level and we put it on the top so that the balance property is preserved.
  3. As in the enqueue function, we check for the heap constrain, this time from the root node comparing it with its smaller children and exchanging them if necessary, all the way down to the last level.
  4. We update the variable that contains the size of the queue.

Every time we insert of get an element of the queue we use one of these functions, and the number of operations we need to do are proportional to the number of levels of the tree. As the tree is always complete and every node has 2 branches, the number of levels are approximately $log_2(n)$ where n is the number of elements stored on the tree, so the complexity of the enqueue and dequeue operations on this queue is always proportional to $log_2(n)$.

In addition to these two functions, we can create other auxiliary functions such as one that create a new queue, one that prints all the elements in the heap and one that returns the size of the queue.

All these functions are implemented with other internal functions on the queue.c file and the user has to accede to them through the header queue.h.

Execution performance

We have measured the time it takes to the A* function to calculate the optimal path with both a sequential an a binary queues. With the sequential queue implemented in the class slides, our program takes 4.15 s, with the optimized compiler -Ofast and the heuristic function "*Haversine*", which will be discussed later.

With the binary heap queue and the same compiler it takes 2.03 s, which means a reduction of more than half of the time.

This shows us that the selection of an efficient algorithm on the priority queue its crucial to get a good performance of the algorithm.

3.3 The heuristic function

Besides the algorithm for the priority queue, the other element that is crucial to the A* result is the heuristic function, which is responsible for evaluating the importance of each node on our search.

In this case, we have decided the heuristic function to be the shortest straight distance on the earth surface between two points, given the latitude and longitude of each point (we are not considering the altitude). This distance is not well defined, so there are different ways of computing it.

Due to that, we are going to expore three different distance formulas: *Haversine formula*, *Spherical Law Cosines* and *Equirectangular approximation.*

- **Harvesine**: calculates the great-circle distance between two points on a sphere given their longitudes and latitudes, where the great circle is the intersection of the sphere and a plane that passes through the center point of the sphere.

$$a = sin^2(\Delta\phi/2) + cos(\phi_1) * cos(\phi_2) * sin^2(\Delta\lambda/2) \tag{1}$$

$$c = 2 * atan2(\sqrt{a}, \sqrt{(1-a)}) \tag{2}$$

$$d = R * c \tag{3}$$

  Where here ϕ_i means latitude of point i, λ means longitude, Δ represents difference between both points and R is the radius of the Earth.

- **Spherical Law of Cosines**: geodetic form of the law of cosines rearranged from the canonical one so that the latitude can be used directly, rather than the colatitude.

$$d = acos(sin(\phi_1) * sin(\phi_2) + cos(\phi_1) * cos(\phi_2) * cos(\Delta\lambda)) * R \tag{4}$$

- **Equirectangular approximation**: used for small distances where performance is important and accuracy is less important, computes the distance over a flat surface.

$$x = \Delta\lambda * cos(\phi_m) \tag{5}$$

$$y = \Delta\phi \tag{6}$$

$$d = R * \sqrt{(}x^2 + y^2) \tag{7}$$

  Where in this case ϕ_m is the average latitude.

For implementing these functions, we pass the heuristic function we want to use as an argument to the A Star function. This allows us to modify the heuristic function and try different alternatives without modifying the code of the principal A Star function.

```
bool AStar(node *nodes, unsigned long NNODES, unsigned long node_start, unsigned long
    node_goal, a_node **result, double (*fdistance)(node *, node *));
```

Furthermore, we can make it even more easily by creating a *define* directive that substitutes the name of the function and call the A Star function using this directive, so we can modify the heuristic function just by changing the name if the function on the *define*.

```
#define HEURISTIC haversine;

bool r = AStar(nodes, nnodes, node_start, node_goal, &result, HEURISTIC);
```

Now we will discuss the performance of these three functions in our code.

Execution performance and comparison

The corresponding final path lengths and execution time of the A Star function are the following:

- **Harvesine**
  Total path length: 958815.0128 m
  Finding path time: 1.89 secs.

- **Spherical Law of Cosines**
  Total path length: 958815.0222 m
  Finding path time: 3.58 secs.

- **Equirectangular approximation**
  Total path length: 958815.0134 m
  Finding path time: 1.55 secs.

We can see that the final distance is almost the same with all the three methods, with the differences appearing on the second decimal. This differences are minimal, since they mean few centimeters on a path of hundreds of kilometers.
The main difference between the methods lies on the execution time, being the Equirectangular approximation the most efficient formula.

4 Conclusions

Through this project we have learned how to read and manage a large amount of data, how to store and read information on a binary file, and how to implement efficiently the path search algorithm A* on C language. We have also compared different algorithms to implement a priority queue and several heuristic functions to measure distances on the surface of a sphere.

Talking about speed, the whole process takes around 45 seconds to complete: 42 seconds for the first part, and 3 seconds for the second one. The most time consuming task of the whole process is the storage of all the nodes on a vector of structs with all the information of each node. This takes 19.53 s, which is about 43% of the total time.
As we can see, the algorithm part takes much less time in comparison with the first part, where all the data is organized. Because of that reason, once we have created our binary file it is relatively inexpensive to run the algorithm as many times as necessary. This shows us the importance of reading and storing our information on a efficient way such as the binary file, so it can be accessed later more quickly.

Moving to the A* algorithm, we have been able to successfully compute the distance from *Basílica de Santa María del Mar* in Barcelona to the *Giralda* in Seville, obtaining a final result of 958815 m. As we have seen, A* is really an optimal an complete algorithm, since it has been able to search an optimal path along a very big amount of nodes in a very short time. The distance obtained in this assigment is shorter than the one found by Google maps (1.014 km) since we are minimizing distances instead of time.

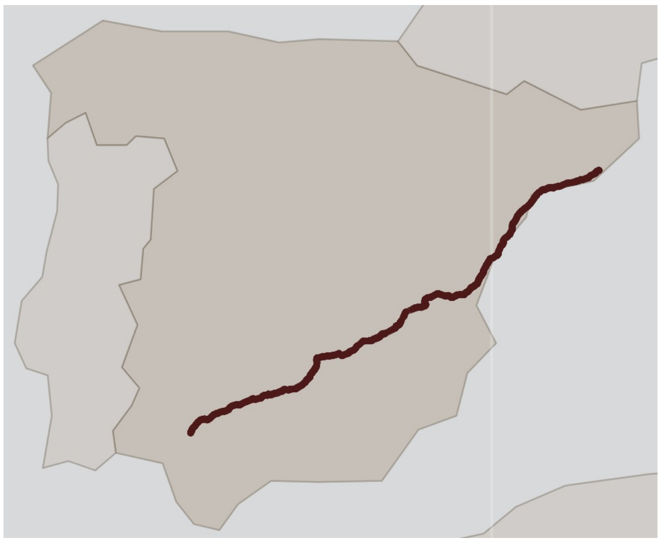


Figure 7: Optimal path founded by A* between *Basílica de Santa María del Mar* and the *Giralda.* The vertical white line represents the Greenwich meridian.

With respect to the queues algorithms, we have seen that organizing our priority queue in the shape of a binary tree is much more efficient than using a traditional sequential queue, since this transforms operations with linear complexity to operations with logarithm complexity. In terms of speed, this has been proven, since performing A* with a binary tree (and using Harvesine) took only 2.03 seconds, while using a sequential queue took around 4.15 seconds.

Last but not least, we have explored different heuristic functions in order to see how this affected to the algorithm. To do this, we have tested different ways of measuring distances on the surface of the Earth and we have seen that all of them give a very similar distance result (around 958815 m.), since the distances between our nodes are relatively small. Nevertheless, the main difference is the execution time. If we are looking for a fast program and we know that we are working with very close points on the map, we can use the Equirectangular Aproximation, since this one was the fastest (it only took 1.55 secs.). On the other hand, if the distances are larger and great precision is needed, then the Haversine formula is the best option.

5 Compiling and executing requirements

In order to compile the first program *AStar_write.c*, you also have to give as an argument the C file with all the necessary functions of the first part *write_functions.c* which is connected with the main program through the header *write_functions.h.*
To run it, you must specify the name of the binary file to be created, for example *binary.bin.* You can also run the program without specifying this argument, but the file won't be created. It is also important to keep both the csv file called *spain.csv* and the program in the same folder.

For the second program *Astar_heap*, you must give as an argument the files *astar_functions.c* for the auxiliary functions of the A* algorithm, and *queue.c* for the implementation of the binary heap.

To run the program *Astar_heap* you must specify the binary file previously created (since *Astar_heap* needs to read this file to get the graph), and as optional, you can specify the name of a file where the path will be stored, for example *result.csv.*

To simplify this, we submit two bash files that execute all the specified above automatically.
The first one *run_astar.sh* makes the compiling and execution of all the procedure, and the second *run_astar_part2.sh* only compiles and executes the second part, when the file *binary.bin* has already been created.