Optimization
A* Algorithm: Routing problem: Barcelona-Sevilla

Arturo del Cerro Vergara (NIU:1593930), Sharika Vijayakumar (NIU:1590768) and Jorge Piqueras Marqués (NIU:1583634)

January 10, 2021

Master's degree in Modelling for Science and Engineering

Abstract

The routing problem between two points is an issue that even nowadays is being researched. The objetive of this project is to solve the routing problem between Plaça de Santa María, Barcelona and Calle Mateos Gago, Sevilla (Iglesia de Santa María del Mar and Giralda respectively) improving the Google Maps' solution in terms of distance. The implemented code in C language is presented and described, and is intended to allow the user to experiment with various possible versions of the algorithm. Three different heuristic and cost functions have been tested, commented and compared, with the purpose of optimizing the efficiency of the algorithm. The optimal path with better performance is found by using weighted heuristics, obtaining a route that covers 958.81 km, and takes 3.91 s of A* algorithm execution.

Contents

1 Project description **4**
1.1 The `spain.csv` file 4

2 The code **5**
2.1 Creation of the graph 5
2.1.1 Reading the graph 6
2.1.2 Writing the binary file 8
2.2 Main code 9
2.2.1 Reading the binary file 11
2.2.2 Heuristic function 11
2.2.3 Weighting and dynamic weighting 13
2.2.4 A* Algorithm 14
2.3 Running the code 15

3 Results **16**
3.1 Non-weighted A* 16
3.2 Weighted A* 17
3.3 Dynamic weighted A* 19

4 Conclusions **21**

References **22**

1 Project description

The objective of this project is to find a path from *Basílica de Santa Maria del Mar* (Plaça de Santa Maria) in Barcelona to the *Giralda* (Calle Mateos Gago) in Sevilla, by means of the A* algorithm. To do this, we will be given a map in the form of a `spain.csv` file that we will have to preprocess in order to create and store a usable version of a map that an A* algorithm can use. The A* algorithm is a best-first search algorithm that is applied to weighted graphs and its goal is to find a path which minimises a cost function from the starting node to the goal node. Depending on the choice of this cost function, we might minimise the time, the distance, or many other variables. In our case we will be minimising distance, and the cost function for a given node will be $f(n) = g(n) + h(n)$. The distance $g(n)$ is the cost of the path followed from the starting node and node n, and $h(n)$ is called the heuristic function, and it can be chosen as many different formulas, as we will see in more detail on future sections. The heuristic function represents an estimate of the distance between the node n and the goal node. This way, A* algorithm selects a path from a node to the next one that minimises $f(n)$.

During this project, we will mainly use the resources provided by professor Lluís Alsedà on his personal page [1], including the csv file for the map, or the codes/pseudocodes for some of the functions we will use.

1.1 The `spain.csv` file

All the information of the map of Spain that we need to access in order to construct a graph is stored in the `spain.csv` file. This file contains about 1.3 GB of data, organized on a table with 25253790 rows, where the first three are just to explain the content of the rest of the rows, so they should be ignored. There are two relevant types of rows, and the shape and content are as follows:

```
node|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|node_lat|node_lon
way|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|membernode|membernode|...
```

The `node` type of rows contain the information of each of the 23895681 nodes on the map. In particular, we will be interested on the `id`, the `node_lat` (latitude of the node) and the `node_lon` (longitude of the node). We will also store the `name` of the nodes, even though most of them are empty, in order to create the outputs of the results. The node id will be useful mainly to tell one node apart from the rest, especially in the reading of the file. On the other hand, the latitude and longitude of a node will give us the tools to calculate the heuristic function given by some definition of a distance between nodes. The rest of the parameters are only useful for visualization purposes. On the other hand, the `way` rows describe the connections between nodes. They contain three interesting parameters: the `oneway` slot can be empty (meaning that the connection between the nodes goes both ways) or full (meaning that the two consecutive nodes are only connected on a way from the left-most one to the righ-most one), and the `membernode` slots can appear up to 9 times and contain a nodes's id. Then, for each two consecutive nodes in the `membernode` list, there is an edge connecting them.

All the slots inside the rows are separated by a '|' delimiter, which we will need to detect on the reading of the file in order to obtain the useful parameters. Also, we will need to presume that the file is imperfect: some ways have less than two nodes, some others have two consecutive nodes with the same id, or a way might have a node that is not on the node list. These outliers have to be detected and discarded on the reading.

2 The code

The implementation of A* algorithm is divided into two programs. The first one is `Write_bin`, and the second one `Astar_main`, which has to be executed together with `Astar_functions`. The objective of this separation is that reading the file from the `spain.csv` file can be time consuming, but only has to be done once. After one execution of `Write_bin`, the graph will be written on a binary file, which `Astar_main` will read in seconds and be ready to apply A* algorithm.

2.1 Creation of the graph

The first part of the code starts with the `Write_bin.c` file and its purpose is to access the `spain.csv` table, read the information from it, store the data in graph form and write it on a binary file. An efficient way to store the parameters of each node is to create the following structure:

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

Then, we will globally declare a `nodes` vector of type `node`, to store all the nodes. It will also be useful, as we will explain later, to define another structure that we will call `successor`:

```
typedef struct{
        unsigned long id;
        int position;
}successor;
```

We will make a global declaration of a `stored` vector of type `successor`, in order to temporarily store the `position` (index of a node in the nodes vector) and `id` of each of the nodes that appear on a single way, and therefore have to be connected. Once we do this, the main function of the program starts by doing a `malloc` of the nodes vector, an initialization with a `malloc` of the successors vector of the nodes, and then goes on to open the `spain.csv` file. After that, it calls the `reading` and `writing` functions which are explained on the following sections.

2.1.1 Reading the graph

The reading of the `spain.csv` file consists on getting the node information from each row of the table, and storing the successors. This is the most time consuming part of the program, and is performed by the following function:

```
void reading (FILE *fcsv)
{
    int type = 0;
    int n = 0;
    unsigned short oneway = 0;
    unsigned long third = 0;
    char *line = NULL;
    size_t size_ = 0;
    ssize_t bytes_read;
    char *piece = NULL;
    unsigned long id_prev = 0;

    if (fcsv == NULL) exit(EXIT_FAILURE);

    while((bytes_read = getline(&line, &size_, fcsv))!=-1)
    {
        if (third > 2){
        while((piece=strsep(&line,"|"))!=NULL)
        {
            if (n ==0)
            {
                type = line_type(piece);
                if (type!=-1) {
                    if(num>1) {
                        link_succ(oneway);}}

                        free(stored);
                        stored = NULL;
                        num = 0;
                        oneway = 0;
            }
```

```
            if (type==0 ) {get_node(n, piece);}
            else if(type==1 )
            {
                if (n==7 && strcmp(piece,"oneway")==0) oneway=1;
                if (n>=9) {
                    if (atol(piece) != id_prev){store_node(piece); id_prev =
                    ↪  atol(piece);} // checks for repeating nodes
                    else {type = -1;}
                }
            }
            n++;
        }
        id_prev = 0;
        k++;
        n = 0;}
        third++;
    }
    printf("number of lines read: %lu\n", third);
}
```

It uses the `getline` built-in function to read each line on the csv file and stores a pointer to its content. The '`if (third > 2)`' clause is included to ignore the three first lines, that are the descriptions of the file. Then, the inner `while` loop uses the `strsep` built-in function to separate each one of the slots (that we call pieces) of the lines, separated by the delimiter '|'. The integer `n` indicates the position of the piece that we are currently reading. As the first piece is always a string that informs of the type of the line, we call the `line_type` function, which returns a 0 if the line is a node, a 1 if its a way, 2 for relations, and -1 if anything else. Then, for the following pieces of a line, if the type is 0, we call the `get_node` function, which stores the node's `id` when the piece corresponds to $n = 1$, the node's `name` when $n = 2$, and the `latitude` and `longitude` when $n = 9$ and $n = 10$ respectively.

After all the node lines are completed, the ways begin, and then the type changes to 1. For the 7th piece, we check if the slot contains the 'oneway' parameter or not. If it does, we change the variable oneway from 0 to 1. Then, for n from 9 to the end of the line, we read all the member nodes on the way. We include an if statement that detects if two consecutive member nodes have the same id, in which case we need to discard the node. If this is not the case, we call the `store_node` function, which stores the `id` and `position` of the member node in the `stored` vector of type successor. The `position` corresponds to the index of the member node in the `nodes` vector, and this is obtained by calling the `binary_search` function (taken from [1]):

```
long binary_search (unsigned long key, node *list, unsigned long ListLength)
{
```

```
    unsigned long start=0UL, AfterEnd=ListLength, middle;
    unsigned long try;

    while(AfterEnd > start)
    {
        middle = start + ((AfterEnd-start-1)>>1); try = list[middle].id;
        if (key == try) return middle;
        else if ( key > try ) start=middle+1;
        else AfterEnd=middle;
    }
    return -1;
}
```

This function takes as an input the `id` of the member node (converted to unsigned long), a node type list (the `nodes` vector) and the `ListLenght` (which is the globally declared total number of nodes `nnodes = 23895681`). It uses the bisection method to find the position of the member node on the node vector and returns this parameter. If the node is not on the list, it returns -1.

After calling this function, `store_node` takes the returned `position` and stores it together with the `id`. Every time a member node is included on the `stored` vector of successors, we increase the counter `num` by one, and reallocate the needed memory. Finally, every time a new line starts (and n is set back to 0), we check if the number of member nodes on the `stored` vector is at least 2, and if so, we call the `link_succ` function. This function takes every consecutive member node and adds them to the successors vector inside the corresponding node. If oneway $= 1$, it only adds the $i+1$ member node as a successor of the node i; whereas if oneway $= 0$, the member node i is also added as a successor of member node $i+1$. After this process, the `stored` vector is freed and 'oneway' and 'num' are set back to 0.

2.1.2 Writing the binary file

Once the reading of the map is completed, the `writing` function [1] is called and a binary file with name `binary.bin` is created.

```
void writing(){
    FILE *fin;
    int i;

    unsigned long ntotnsucc=0UL;
    for(i=0;i<nnodes; i++)   ntotnsucc+=nodes[i].nsucc;


    if((fin=fopen("binary.bin","wb"))==NULL) printf("The binary Output file
    ↪  can't be opened\n");

    /* Global data---header */
```

```
    if((fwrite(&nnodes,sizeof(unsigned long),1,fin) +
    ↪  fwrite(&ntotnsucc,sizeof(unsigned long),1,fin))!=2) printf("When
    ↪  initializing the output binary data file\n");

    /* Writting all nodes */

    if( fwrite(nodes,sizeof(node),nnodes,fin)!=nnodes) printf ("When writing
    ↪  nodes to the output binary data file\n");

    //Writting successors in blocks
    for (i=0;i<nnodes; i++)
        if(nodes[i].nsucc)
        {
            if(fwrite(nodes[i].successors,sizeof(unsigned
            ↪  long),nodes[i].nsucc,fin)!=nodes[i].nsucc)
            printf("When writing edges to the output binary data
            ↪  file\n");
        }

    fclose(fin);
}
```

2.2 Main code

The `Astar_main.c` code performs a set of functions detailed in `Astar_functions.c` that ends with an output file with the solution for our problem. The most relevant functions based in [2] used are:

- `Reading_from_file, heuristic functions` and `Astar_algorithm` that will be detailed below.
- `Init_list`, which initializates the open list.
- `Init_values`, which sets initial values copying part of them from the binary file into a `Node_status` structure. Most important issue is the set of heuristic and g_cost values to infinity.
- `Pop_node`, which removes a node from the open list.
- `Insert_node_before`, which appends a node in the open list before a target node.
- `Insert_node_at_end`, which appends a node in the open list at the end of it.
- `Add_to_open_list`, which adds a specified node into the open list sorting it by the `f` value. Tie break rule: in case of the node to be added having the same `f` value as other in the open list, we decided to keep the older node in the open list.

- `Print_path`, which prints the path from start node to goal node in the `output_Astar.txt` file, giving the number of nodes in the path, the distance from each node to the starting point and the latitude and longitude coordinates. It also prints the time A* algorithm takes to find a path and the reading time of the binary file. Names of the nodes are not printed because most of them are not informed and nodes can be identified by its id.

- `Exit_error`, the function to exit with an error.

In addition, we need to define the following structures to work with the nodes inside the A* algorithm:

```
typedef struct Node{
    unsigned long id;
    double latitude, longitude;
    unsigned short num_succesors
    unsigned long *succesors;
    struct Node *prev, *next,*parent;
    Queue queue_status;
        double f, h, g;
}Node_status;
```

It contains the same information as the `Node` structure on the `Write_bin.c` code, but adding the previous, next and parent nodes, the values of the cost and heuristic functions, and a `queue_status` parameter of type `Queue`, defined as follows:

```
typedef char Queue;
    enum whichQueue { NONE, OPEN, CLOSED };
```

This definition is useful to control if a node is on the open or closed list. And finally we show the open list structure we work with below.

```
typedef struct{
  Node_status *head, *tail;
} List;
```

The `Astar_main.c` program contains the main function that executes the code. It starts by doing a `malloc` of the `Node_status` vector of structures, and then reads the binary file `binary.bin` that was written in the `Write_bin.c` program. After this, it initializes the values of the `Node_status` vector, and proceeds to start to ask the user to input the choice of the starting and goal nodes, and the cost function type. Finally, it executes the `Astar_algorithm` function and creates the output file with the optimized path obtained.

2.2.1 Reading the binary file

We use the `reading_from_file` function [3] that mainly opens the binary file, reads the header, allocates memory for the variables, reads and stores the data in this variables and closes the file. Once this is done we set pointers to the succesors.

```
void reading_from_file(){
        int i;
        static unsigned long * allsuccesors;
        unsigned long ntotnum_succesors=0UL;
        FILE * file_in;

  // Open file
  if ((file_in = fopen("binary.bin","rb")) == NULL)
    Exit_Error("The data file does not exist or cannot be opened\n",4);

 // Header
  if ((fread(&num_nodes,sizeof(unsigned long ),1,file_in) +
       fread(&ntotnum_succesors,sizeof(unsigned long ),1,file_in))!=2)
    Exit_Error("when reading the header oh the binary file\n ", 5);

  // Memory
  if ((nodes=malloc(sizeof(Node)*num_nodes))==NULL)
    Exit_Error("when allocating memory for the nodes vector\n ", 6);

  if((allsuccesors=malloc(sizeof(unsigned long )*ntotnum_succesors))==NULL)
    Exit_Error("when allocating memory for the edges vector\n ", 7);

  // Data
  if (fread(nodes,sizeof(Node),num_nodes,file_in)!=num_nodes)
    Exit_Error("when reading nodes from the binary data file\n ", 8);

  if (fread(allsuccesors,sizeof(unsigned long ),ntotnum_succesors ,file_in ) !=
  ↪  ntotnum_succesors)
    Exit_Error("when reading succesors from the binary data file\n ", 9);

  fclose(file_in);
  for (i=0;i<num_nodes; i++)
    if (nodes[i].num_succesors){
        nodes[i].succesors=allsuccesors;
        allsuccesors+=nodes[i].num_succesors;}
  }
```

With the O3 flag we achieved a 0.69 seconds time to read the binary file.

2.2.2 Heuristic function

As we mentioned in section 1, the choice of the heuristic function for the A* algorithm is key for its success. In general, an heuristic function in an algorithm is used to help make decisions regarding the path to take during the searching process. It represents an approximation of the final solution of the problem, and

lets the algorithm decide what is the route that gets closer to it. The cost function that is minimised during the algorithm is:

$$f(n) = g(n) + h(n). \tag{2.2.1}$$

The distance between the starting node and node n is given by $g(n)$, and $h(n)$ is the heuristic function, representing the estimate of the distance from node n to the goal node. For an heuristic function to be good, it should be monotone and smaller or equal to the optimal distance between the nodes. In this project, we will apply the code using three different equations for the heuristic function, based on the information found in [7].

☐ **Equirectangular approximated distance.**

With invention attributed to Marinus of Tyre, this distance is calculated by drawing meridians and circles of latitude as straight lines on the earth's map, and then applying Pythagoras' theorem. The equation of the equirectangular distance between two nodes of latitude ϕ_1 and ϕ_2, and longitude λ_1 and λ_2 is:

$$h_{\text{EQ}} = R\sqrt{(\Delta\lambda \cos\phi_m)^2 + (\Delta\phi)^2}, \tag{2.2.2}$$

where $\Delta\lambda = \lambda_2 - \lambda_1$, $\Delta\phi = \phi_2 - \phi_1$, $\phi_m = \frac{\phi_1+\phi_2}{2}$, and $R = 6371km$ is earth's mean radius. This formula is useful to implement when we don't particularly focus on precision, but on performance, so it's the distance we use to test the algorithm. The function we use for its implementation is:

```
double h_equir (Node_status * node_1, Node_status * node_2) {
    double phi1 = node_1 -> latitude/180*PI;
    double phi2 = node_2 -> latitude/180*PI;
    double lambda1 = node_1 -> longitude/180*PI;
    double lambda2 = node_2 -> longitude/180*PI;

    double dif_long = fabs(lambda1-lambda2);
    double dif_lat = fabs(phi1-phi2);

    return R*sqrt(pow(dif_long*cos((phi1+phi2)/2),2) + pow(dif_lat,2));
}
```

☐ **Haversine distance**

The Haversine formula calculates the great-circle distance, which is the minimum distance between two points in a spherical surface. Its based on the law of Haversine applied to a "triangle" of three points on a sphere surface. The formula for the distance is given by:

$$h_H = 2R \arctan\left(\sqrt{\frac{sin^2(\frac{\Delta\phi}{2}) + cos(\phi_1)cos(\phi_2)sen^2(\frac{\Delta\lambda}{2})}{1 - sin^2(\frac{\Delta\phi}{2}) + cos(\phi_1)cos(\phi_2)sen^2(\frac{\Delta\lambda}{2})}}\right). \tag{2.2.3}$$

This formula, oftenly used in navigation, is more precise than the equirectangular distance, but also harder to compute. The code we use in our program is:

```
double h_haversine(Node_status * node_1, Node_status * node_2 ) {
    double phi1= node_1->latitude/180*PI;
    double phi2= node_2->latitude/180*PI;
    double lambda1=node_1->longitude/180*PI;
    double lambda2=node_2->longitude/180*PI;

    double dif_long = fabs(lambda1-lambda2);
    double dif_lat = fabs(phi1-phi2);

    double a =
    ↪  sin(dif_lat/2)*sin(dif_lat/2)+cos(phi1)*cos(phi2)*sin(dif_long/2)*sin(dif_long/2);
    return R*2*atan2(sqrt(a),sqrt(1-a));
}
```

☐ **Spherical law of cosines distance**

It uses the law of cosines applied to a sphere's surface to calculate the distance between two points.

$$h_S = R \arccos\left(\sin(\phi_1)\sin(\phi_2) + \cos(\phi_1)\cos(\phi_2)\cos(\Delta\lambda)\right). \tag{2.2.4}$$

This formula is useful because it is a good approximation of the Haversine distance in small scales. The code used to implement it is:

```
double h_spherical (Node_status * node_1, Node_status * node_2 ){
    double phi1= node_1->latitude/180*PI;
    double phi2= node_2->latitude/180*PI;
    double lambda1=node_1->longitude/180*PI;
    double lambda2=node_2->longitude/180*PI;

    double dif_long= fabs(lambda1-lambda2);
    return R*acos( sin(phi1)*sin(phi2)+cos(phi1)*cos(phi2)*cos(dif_long));
}
```

2.2.3 Weighting and dynamic weighting

Besides the usage of different heuristic functions, we can add weights [6] on the cost function in order to control how much "importance" we give to the heuristic function on the algorithm. We can add the weight parameter w to the cost function as follows:

$$f_w = (1-w)g(n) + wh(n). \tag{2.2.5}$$

In this situation, by changing the parameter $w \epsilon [0, 1]$, we control the way the search is done in the algorithm. If we choose $w = 0$, then $f_w = g(n)$, and we eliminate the heuristic term, and therefore we are using Dijkstra algorithm, which computes all the nodes's costs before finding the solution path. On the other hand, if we choose $w = 1$, we eliminate the backwards cost g, $f_w = h(n)$, which represents a Greedy search algorithm. Any value of w between 0 and 1, finds a solution between these two methods, where $w = 0.5$ goes back to the non-weighted case. Choosing the right w can be interesting as it can improve performance, but in practise we need to be careful because a w too high might stop the convergence of the model. Usually, though, a value of w slightly above $w = 0.5$ improves efficiency.

In addition, we might find that the algorithm becomes slow at some parts of the map. To try to solve this, we can use dynamic weighting:

$$f_{dw} = g(n) + h(n) + \epsilon \left(1 - \frac{d(n)}{N}\right) h(n), \tag{2.2.6}$$

where $d(n)$ is the depth of the node n, and N is the anticipated depth of the goal node. In the earlier iterations of the algorithm, the fraction $d(n)/N$ is lower, and therefore the weight of $h(n)$ is higher, so the algorithm behaves as a Greedy algorithm, encouraging depth-first searches, making it faster. On the other hand, at the last stages of the path, the fraction is $d(n)/N \simeq 1$, and then we return to regular A* algorithm. The result of this, is an algorithm that is always "greedier" than A*, given that for any ϵ different than zero (if it's 0 then we go back to regular A*), the algorithm always adds a weight in the heuristic function that is greater than 1. Choosing to implement this can speed up the search, but you might run into sub-optimal final paths.

2.2.4 A* Algorithm

Based on peudocode from [4] the A* algorithm is presented here. Is uses the `Node_status` structure in order to track the parents, next and previous nodes in the path and to inform about the `g` and `h` costs. The `Astar_algorithm` takes as input the start and goal node, the evaluation for the `f` values (Default, Weighted or Dynamic Weighted) and the parameter for this `f` in case it is not choosen as default.

```
int Astar_algorithm (unsigned int start_node, unsigned int goal_node, int
↪  evaluation, double param){
    Node_status *current_node , *succesor_node;
    int i;
    double succesor_current_cost;
    List open_list; Init_list(&open_list);
    open_list.head = open_list.tail = &(node_status[start_node]);
    node_status[start_node].g = 0;
    node_status[start_node].h =
    ↪  h_equir(&(node_status[start_node]),&(node_status[goal_node]));

    while (open_list.head != NULL){
```

```
        current_node = open_list.head ;
        if ( current_node->id == node_status[goal_node].id) {return 1;}
        for ( i = 0; i < (current_node->num_succesors) ; i++){
            succesor_node = & node_status[*(current_node->succesors+i)];
            succesor_current_cost = current_node -> g +
            ↪  h_equir(succesor_node,current_node);
            if ( succesor_node -> queue_status == OPEN){
                if ( succesor_node -> g <= succesor_current_cost)
                    continue;
                pop_node(&open_list, succesor_node);}
            else if ( succesor_node -> queue_status == CLOSED){
                if ( succesor_node -> g <= succesor_current_cost)
                    continue;
                succesor_node -> queue_status = OPEN; }
            else{
                succesor_node -> queue_status = OPEN;}
            succesor_node -> g = succesor_current_cost;
            succesor_node -> h =
                h_equir (succesor_node, &(node_status[goal_node]));
            succesor_node -> f = evaluation_function(succesor_node, evaluation,
            ↪  param, start_node, goal_node);
            add_to_open_list ( &open_list, succesor_node);
            succesor_node -> parent = current_node;
        }
    current_node -> queue_status = CLOSED;
    pop_node (&open_list,current_node);
    }
    return -1;
}
```

As the way we constructed the path is by saving the parent of each node of the path backwards (not from start to goal), then in the function `print_node` this is fixed in order to print the path correctly.

2.3 Running the code

To run the code, the `spain.csv` must be in the current working directory. To compile the reading of the graph and the writing on a binary file we use gcc and then execute the program with a command the likes of:

```
gcc -g -Wall -o write_bin Write_bin.c -lm
write_bin
```

Once the execution has finalized (approximately 30 seconds long), the `binary.bin` file will be written and ready to use as many times as we want. We compile the main A* program that is composed of `Astar_main` and `Astar_functions` in order to modularize the code. The instructions for compiling and executing them are:

```
gcc -g -Wall -o Astar_main Astar_main.c Astar_functions -lm -O3
Astar_main
```

We have used the optimizer flag $-O3$ in order to improve performance. Once the `Astar_main` is executed it will start by asking the user to choose between running the program for the default start and goal nodes, or to choose them manually. Afterwards, it will ask the used whether to use the default cost function, a weighted version, or one with dynamic weighting. If we choose one of the last two options, the program will ask the user for the values of the weight w, or the parameter ϵ. The user can find a summary of the optimized path found, on the `output_Astar.txt` file that is created after the execution.

3 Results

In order to test the program's performance, we will study four parameters that inform of the efficiency of the results: execution time of A* algorithm, number of iterations, total distance covered by the optimized path and number of nodes expanded. The code runs by default using the heuristic as the equirectangular approximated distance, but we also try with the Haversine distance and Spherical. In addition, we will analyse the results by applying weights to the cost function, and also dynamic weighting, trying different values for the w and ϵ control parameters. The visualization of the paths we will include is made with a simple code using the `geoplot` function in `Matlab` (`https://es.mathworks.com/help/matlab/ref/geoplot.html`), which allows to add and join two coordinates on earth's map. To do this, we use the `output_Astar.txt` that is produced after an execution, importing it to `Matlab` and obtaining the latitude and longitude coordinates of every node that appears on the optimized path.

3.1 Non-weighted A*

The default version of the code implements A* algorithm with the regular cost function given by Equation 2.2.1. We can run the code for the three different heuristics and compare the results. In table 1, the performance for the default execution is shown.

Table 1: Performance for the default A* execution

Heuristic	Distance (km)	Expanded nodes	Nodes in path	A* execution time (s)
Equirectangular	958.815	2309499	6648	5.55
Spherical	958.815	2309499	6648	5.61
Haversine	958.815	2309499	6648	5.64

The time elapsed in the printing of the output file is around 0.6s independently of the type of heuristic or cost function. We notice that the equirectangular distance displays the best performance results, with an execution time of 5.55s, followed by the Spherical distance and Haversine. This is a reasonable and

expected result, as the Haversine forumula is the most accurate one, so it results in precisely optimized paths, but the time for the calculation is higher. Both the spherical and equirectangular distances are aproximations, with the equirectangular being the least accurate measurement. This is why the equirectangular is the fastest to execute, and also because from Barcelona to Sevilla there is not too much earth curvature so we can consider it as flat. In consequence of this results, we have chosen the equirectangular distance as the default heuristic. In Figure 1 we draw the path for the default non-weighted version of the program.

Figure 1: Optimal path (in red) obtained with the non-weighted A* algorithm, with equirectangular distance as heuristic function. The execution time of the algorithm is 5.55s and the distance of the path is 958.815 km.

3.2 Weighted A*

The selection of a weighted cost function, given by Equation 2.2.5, will result in the code asking the user to input the weight w. As we previously explained, this parameter controls if the algorithm is closer to Dijkstra (for lower values of w), or behaves as a greedy search algorithm (for values of w closer to 1). In Table 2, we have selected some different relevant values of w to study the performance of the code, and also tried it for the three heuristics. Some key results, are the choice of $w = 0$, obtaining the Dijkstra's algorithm imlementation, or the $w = 0.5$, where the code behaves like the regular default A*. As it is expected, for values of $w \epsilon [0, 0.5]$, the algorithm reaches the same optimal solution than the default version, but the number of expanded nodes, and therefore also the execution time, grows as we decrease w. For values of w closer to 1, we find that the algorithm is much faster, as it behaves as a greedy search, giving more importance to the heuristic and expanding less nodes, but also giving a sub-optimal solution. The

most efficient way to implement weights, is to choose w slightly above 0.5, as this will give enough importance to the heuristic function so that the execution time is reduced, but not so much that the path becomes sub-optimal.

Table 2: Performance for the weighted A* execution

Heuristic	w	Distance (km)	Expanded nodes	Nodes in path	A* execution time (s)
Equirectangular	0 (Dijkstra)	958.815	13071469	6648	31.98
Equirectangular	0.1	958.815	11784506	6648	33.91
Equirectangular	0.25	958.815	10521848	6648	35.22
Equirectangular	0.4	958.815	8083165	6648	27.30
Equirectangular	0.45	958.815	5888678	6648	26.31
Equirectangular	0.5 (A*)	958.815	2309499	6648	5.60
Equirectangular	0.51	958.815	1901666	6648	4.40
Haversine	0.51	958.815	1903229	6648	5.04
Spherical	0.51	1139.029	1840685	6666	4.46
Equirectangular	0.52	958.897	1743077	6659	3.91
Haversine	0.52	958.897	172974	6659	4.63
Spherical	0.52	1139.248	2042741	6723	3.63
Equirectangular	0.55	976.062	297720	7451	0.13
Haversine	0.55	975.777	301251	7438	0.23
Spherical	0.55	1162.001	460454	7585	0.13
Equirectangular	0.6	1085.911	92197	8835	0.04
Equirectangular	0.88	1429.223	61137	16817	0.02
Equirectangular	1	1663.521	21245	14905	0.01

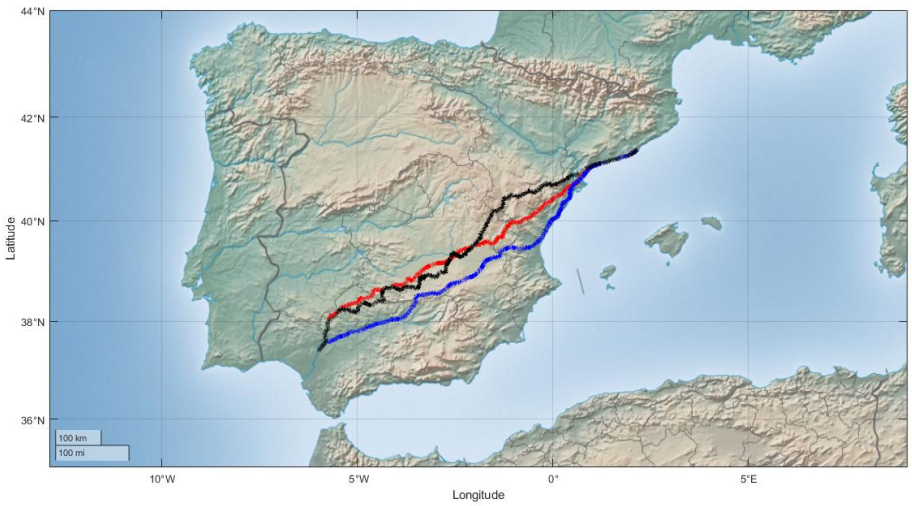

Figure 2: Graph with three sub-optimal paths using equirectangular distance as heuristic, and weighted cost function. In blue, the case $w = 0.55$, very close to the optimal path; in red, $w = 0.6$, a path that increases the distance to 1085.911 km, but reduces time to 0.04 s. In black, a very sub-optimal path with $w = 0.7$, that represents the usage of a weight too high which makes the algorithm too greedy.

Among the most relevant results we find that, for the value $w = 0.51$ and the equirectangular heuristic (in gray), the optimal solution is still found but the execution time is reduced a 21% from the default A*. Also, for value $w = 0.52$

(also in gray), the time is reduced a 30%, while keeping the distance very close to the optimal, differing by meters. As for the Haversine heuristic, we notice that it reaches the same distance values than the equirectangular, but generally in a higher time. The Spherical heuristic displays worse results in general, not finding the optimal path by a big margin. This might happen because, as we explained, this approximation better on small scales. In Figure 2, we can visualize the three different paths obtained for three values of w.

3.3 Dynamic weighted A*

Lastly, in Table 3, we show the performance for different cases of the application of the code with dynamic weights on the cost function, given by Equation 2.2.6. In general, for any value of the control parameter ϵ, dynamic weighting results on a greedy search algorithm, so the objective of applying this effect is mainly reducing the execution time even more for the early stages of the algorithm. By choosing $\epsilon = 0$, we can check that the algorithm behaves like the regular A*, whereas for high values of ϵ, the execution time decreases but the path is very sub-optimal. In fact, for $\epsilon > 0.1$, the path obtained becomes sub-optimal, so we are interested on values above this threshold, even though, if we want to be a bit more flexible in terms of distance accuracy, the result $w = 0.2$ is also a very good implementation, with a 67% decrease on execution time. The most interesting result in this case, is $w = 0.05$, with which we obtain the optimal path, but reduce the time from the default in 0.43 s.

Table 3: Performance for the dynamic weighted A* execution

Heuristic	ϵ	Distance (km)	Expanded nodes	Nodes in path	A* execution time (s)
Equirectangular	0 (A*)	958.815	2309499	6648	5.72
Haversine	0 (A*)	958.815	2314405	6648	7.36
Spherical	0 (A*)	1139.029	2266797	6666	5.55
Equirectangular	0.001	958.815	2301071	6648	5.81
Equirectangular	0.05	958.815	2342978	6648	5.13
Haversine	0.05	958.815	2327429	6648	6.90
Spherical	0.05	1139.029	3167435	6666	5.7
Equirectangular	0.01	958.815	2232466	6648	5.58
Equirectangular	0.1	958.815	23836989	6648	31.97
Equirectangular	0.2	970.675	2183489	7348	1.80
Haversine	0.2	970.675	2105253	7348	3.14
Spherical	0.2	1169.437	5901346	8944	2.11
Equirectangular	0.25	992.340	1873956	9618	1.06
Equirectangular	0.5	997.670	230973	9226	0.16
Haversine	0.5	999.244	239004	9442	0.33
Spherical	0.5	1285.863	2394854	9380	0.73
Equirectangular	0.75	1127.157	129672	9384	0.10
Equirectangular	1	1166.727	2063124	14305	0.75

In Figure 3, we show three cases of dynamic weighting where the trajectories are very close to each other at the end of the paths, but they differ in the first stages. This is the expected behaviour of dynamic weighting, which makes the algorithm behave in a greedier way at the beginning, and more A*-like at the end.

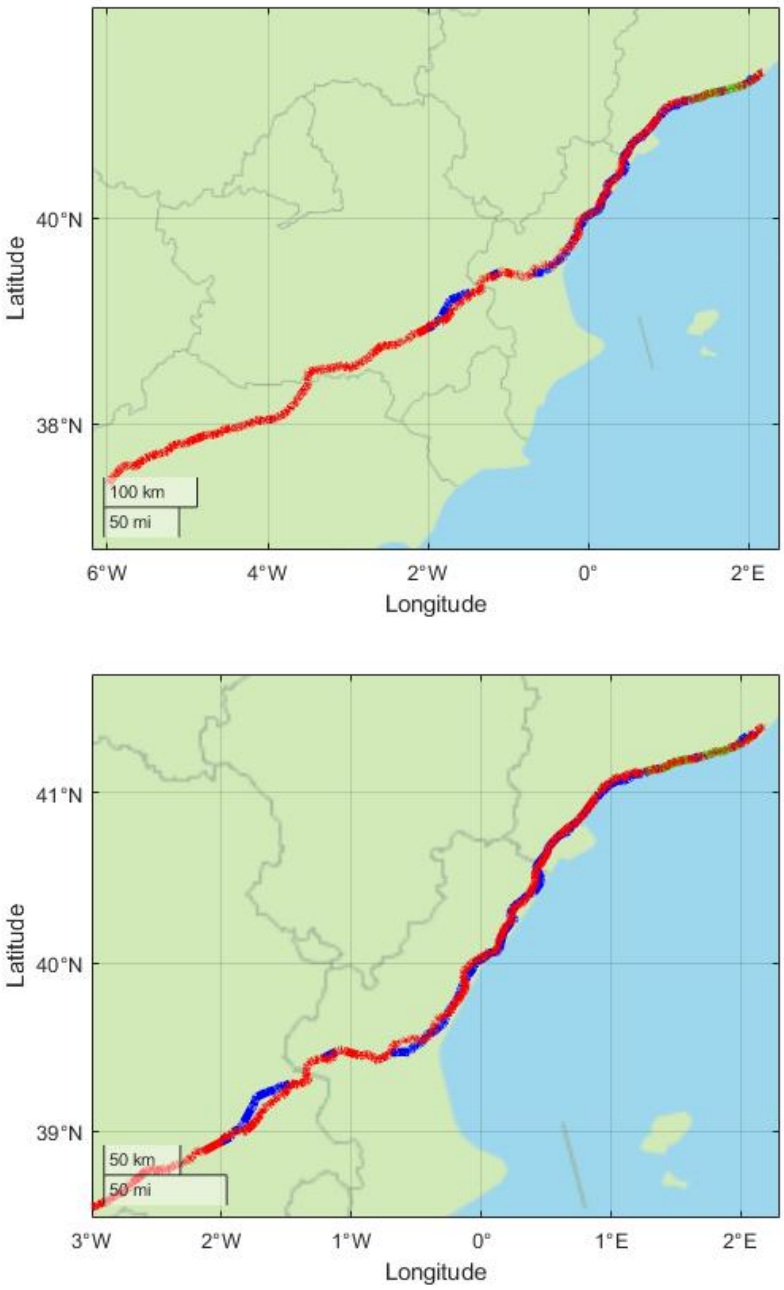

Figure 3: Visualization of the slight difference between the paths obtained by setting $w = 0.05$ (in green), $w = 0.2$ (in blue) and $w = 0.6$ (in red). The green path is the most optimal one obtained by dynamic weighting, and the blue one is very similar to it, differing only on about 12km. In the lower picture, we zoom in on the earlier stages of the paths, showing that they differ mainly on this portion of the algorithm.

4 Conclusions

For this project, we have developed an A* algorithm in C language for solving the routing problem given a starting and goal point. The particular case of interest was the route Plaça de Santa María (Barcelona) to Calle Mateos Gago (Sevilla). The reading of the `spain.csv` file has been done while dodging the imperfections of the file: we controlled the ways with two consecutive nodes with same `id`, which could result on the algorithm getting stuck on it, or the presence of ways with nodes that aren't on the nodes list. We have studied the importance of the heuristic function choice for the algorithm, implementing three different definitions for the distance: equirectangular, spherical and Heuristic. By analysing the performance through the executed time of A*, the distance covered by the path, and the explored nodes, we conclude that, if we are interested on having an optimal path in terms of distance, while reducing the time as much as possible, the equirectangular distance function is the best choice.

In addition, we have implemented different types of cost functions, as the non-weighted default one, weighted and dynamic weighted, obtaining that the default one reaches the optimal path for our code, but adding weights to the heuristic term can be very helpful to reduce the execution time. Besides, we have tested how the Dijkstra algorithm is way worse than A* algorithm in terms of execution time, as the number of expanded nodes is an order of magnitude bigger. Dynamic weighting proved to be useful if our aim was to make the program faster and didn't care a lot about finding a totally optimal path, specially on the earlier stages of the map. In general, increasing the weight reduces execution time on A* algorithm, but this might not always be the case. We found that setting the control parameter on the dynamic weighting version to $\epsilon = 0.1$ increases execution time in a strange way. The paper in [6] contains possible interpretations on why and when weighted A* is not always faster. Some of the explanations are that the execution time is not always reduced when the heuristic function error is big, has little correlation with the real distance, or when the heuristic contains local minima for many nodes. This means that probably, if the algorithm finds many "tie break" situations where two nodes minimize the distance, execution time increases, which might happen less the more accurate the heuristic is. This topic is highly discussed in literature and is still being studied.

In summary, considering all the possible versions of our code, the best choice for performance is to apply the weighted version with $w = 0.51$ (if we want total precision on the distance), or $w = 0.52$, with equirectangular distance as heuristic. This last version is executed in about 3.91 s, giving a 30% reduction from the default version. Although this is a good result, our algorithm is not as good as the one provided as example by professor Alsedà's, [1] as it takes 47 extra nodes in the final path. However we achieved other goal of the assignment: to improve the Google Maps' path in terms of distance. As Google Maps search for time optimization, not distance, our algorithm takes 958.81 km versus the 978 km Google Maps gives in the shortest distance option (walking).

References

[1] Alsedà, L. (2021). Optimisation. Master's degree in Modlling for Science and Engineering. http://mat.uab.cat/~alseda/MasterOpt/index.html. [Online; accesed 07-January-2021].

[2] Alsedà, L. (2015). Tipus Abstractes de Dades Lineals. http://http://mat.uab.cat/~alseda/MatDoc/TADL.pdf. [Online; accessed 08-January-2021].

[3] Alsedà, L. Reading and Writing Binary files in C. http://mat.uab.cat/~alseda/MasterOpt/ReadingWriting-bin-file.pdf. [Online; accessed 08-January-2021]

[4] Alsedà, L. A* Algorithm pseudocode. http://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf. [Online; accessed 08-January-2021].

[5] MATLAB. (2010). version 7.10.0 (R2010a). Natick, Massachusetts: The MathWorks Inc.

[6] Patel, A. (2020). Variants of A*. http://theory.stanford.edu/~amitp/GameProgramming/Variations.html. [Online; accesed 08-January-2021].

[7] Veness, C. (2020). Movable type scripts. Calculate distance, bearing and more between Latitude/Longitude points. https://www.movable-type.co.uk/scripts/latlong.html [Online; accessed 08-January-2021].

[8] Wilt, Chirstopher and Ruml, Wheeler (2012). When does weighted A* fail? Proceedings of the 5th Annual Symposium on Cobinatorial Search, SoCS 2012. pp 137-144.