OPTIMIZATION

Delivery 2: AStar

Sofia Almirante, 1527718

Pau Reig, 1527816

November 2022

AStar algorithm implementation

The objective of this work is to find the shortest path between any two points on a map through the implementation of the A* search algorithm. For this implementation, we have built two parallel linked lists. One of them acts as a priority queue and contains all the neighbours of the visited nodes in ascending order of the value $f = g + h$, while the other one stores all the visited nodes in a tree form with a father-son relation. Once the target node reaches the beginning of the cue, the shortest path is given by the expanded list.

At each step of the algorithm, we visit (and remove) the first element of the priority queue, and we add it to the tree linked list. Then we insert its neighbours in the priority queue according to its f value. If the node is already contained in the priority queue, we are going to keep only the one that minimizes f.

The $f(v)$ function of a node v is defined as the sum of its cost function $g(v)$ and its value of the heuristic function $h(v)$. The cost function represents the distance through the path from the origin node to v, while for the heuristic function we have decided to use the distance in the surface of a sphere, given by the Haversine formula, between the target node and v. This chosen heuristic function is admissible because the real distance through the path between any two nodes can't be lower than the distance computed in straight line over the surface of a sphere.

Code details

Reading the map

Due to the large size of the files we need to work with, we have considered it convenient to write a separate code to convert the provided files into binary files, so that the reading time during execution is reduced. This code reads the *.csv* file of the map, and stores this information as a list of nodes with the following structure definition:

```c
typedef struct Node{
    unsigned long id;
    char name[200];
    double lat, lon;
    unsigned short nsucc;
    unsigned long successors[9];
    //Afegit
    struct Node *father,*next,*previous;
    double f,g;
    int visited;
} node;
```

Here is a brief explanation of the attributes contained in this node structure, of which only the first six will store information in this first code:

- *id*: The unique identification integer number of a node.
- *name*: A string containing the optional name of a node.
- *lat* and *lon*: The latitude and longitude components of the coordinates of the node, stored as long floats.
- *nsucc*: The integer number of successors of the node.
- *successors*: A list containing the index of the successors of the node.
- *father*: A pointer to the previous node structure in the tree linked list. This is used at the end of the code, to be able to trace back the optimal path from the destination.
- *next*: A pointer to the next node structure on the priority queue.
- *previous*: A pointer to the previous node structure in the priority queue. This is helpful to be able to remove an element from the priority queue without going through all of its elements.
- f and g: The functions $f(v)$ and $g(v)$ as discussed before.
- *visited*: A binary attribute to know if a node has already been visited.

A* implementation

We can divide this second code in three sections: a first part that reads the binary file and stores the information, followed by the A* algorithm implementation and finally the output generation.

In this section, we are going to provide a description of the different components of this code. At the beginning, we find the declaration of three **constants**:

- *ORIGIN*: with the id value of the origin node for the assignment (240949599).
- *DEST*: ith the id value of the destination node for the assignment (195977239).
- *EARTH_RADIUS*: with the average value of the earth radius (6371000) in meters.

Then, we define two different **structures**:

- *node* structure: As discussed before.
- *queue* structure: This structures gives form to the priority queue an enable us to keep it consistent and modifiable.

```c
typedef struct {
    node *start;
    node *end;
}queue;
```

The **functions** created for this code are the following ones:

- *ExitError*: This function, also defined in the *readingmap* code, is useful to stop the code when an error occurs and to be able identify the error.

```c
void ExitError(const char *miss, int errcode) {
    fprintf (stderr, "\nERROR: %s.\nStopping...\n\n", miss); exit(errcode);
}
```

- *RealDist*: This function provides the distance between two node structures using the Haversine formula.

```c
double RealDist(node* A, node* B){
    double f1 = A->lat*M_PI/180;
    double f2 = B->lat*M_PI/180;
    double l1 = A->lon*M_PI/180;
    double l2 = B->lon*M_PI/180;
    double Df = f2 - f1;
    double Dl = l2 - l1;
    double a = sin(Df*0.5)*sin(Df*0.5) + cos(f1)*cos(f2)*sin(Dl*0.5)*sin(Dl*0.5);
    double c = 2*atan2(sqrt(a),sqrt(1-a));

    return EARTH_RADIUS*c;
}
```

- *DeQueue*: This function removes the first element of a queue structure.

```c
void DeQueue(queue* cua){
    node* primer = cua->start;
    cua->start = primer->next;
}
```

- *PutBetween*: This function inserts a target node structure between specified nodes A and B. It also modifies the g and f value of the introduced target node and sets its father, which will be the first element of the queue.

```
void PutBewteen(node* A, node* B, node* target, node* start, double g, double f){
    A->next = target;
    target->previous = A;
    target->next = B;
    B->previous = target;
    target->father = start;
    target->g = g;
    target->f = f;
}
```

- *PutAtEnd*: This function puts a node structure at the end of a queue structure. Like the previous function, it also modifies its f and g value and sets its father as the first element of the queue.

```
void PutAtEnd(queue* cua, node* target, double g, double f){
    cua->end->next = target;
    target->previous = cua->end;
    cua->end = target;
    target->father = cua->start;
    target->g = g;
    target->f = f;

}
```

- *SearchNode*: This function returns the index of a node inside the nodes list given its *id*.

```
unsigned long SearchNode(unsigned long id, node *nodes, unsigned long nnodes){
    unsigned long l = 0, r = nnodes - 1, m;
    while (l <= r){
        m = l + (r - l) / 2;
        if (nodes[m].id == id) return m;
        if (nodes[m].id < id)
            l = m + 1;
        else
            r = m - 1;
    }
    return nnodes+1;
}
```

The *main* function

Once we have defined the functions and structures, we can proceed to explain the *main* function of our program.

First, the program **reads the map** from a binary file.

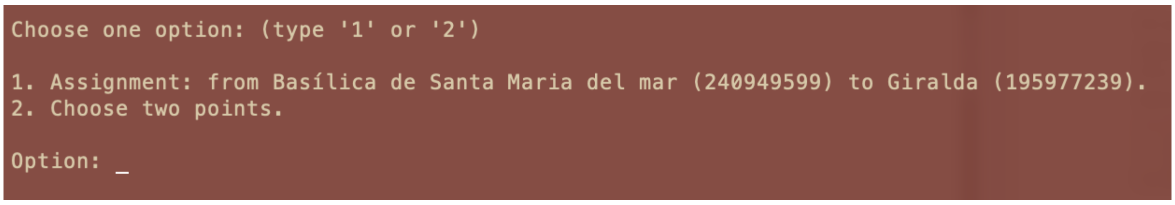
```
//READ BINARY FILE
printf("Reading map...\n");
if ((fin = fopen (argv[1], "rb")) == NULL){
  ExitError("the data file does not exist or cannot be opened", 11);
}
/* Global data         header */
if( fread(&nnodes, sizeof(unsigned long), 1, fin) + fread(&ntotnsucc, sizeof(
  unsigned long), 1, fin) != 2 ){
  ExitError("when reading the header of the binary data file", 12);
}
/* getting memory for all data */
if((nodes = (node *) malloc(nnodes*sizeof(node))) == NULL){
  ExitError("when allocating memory for the nodes vector", 13);
}
if((allsuccessors = (unsigned long *) malloc(ntotnsucc*sizeof(unsigned long))) ==
   NULL){
  ExitError("when allocating memory for the edges vector", 15);
}
/* Reading all data from file */
if( fread(nodes, sizeof(node), nnodes, fin) != nnodes ){
  ExitError("when reading nodes from the binary data file", 17);
}
for(int i = 0; i < nnodes; i++){
  if( fread(&nodes[i].successors, sizeof(unsigned long), nodes[i].nsucc, fin) !=
  nodes[i].nsucc){
    ExitError("when reading sucessors from the binary data file", 18);
  }
}
fclose(fin);
printf("Done!\n");
```

Where the following variables have been used:

```
FILE *fin;
unsigned long nnodes,ntotnsucc,*allsuccessors;
node *nodes;
```

Right after that we find the implementation of the A* algorithm. First we decided to create a little **visual interface** to give a more general utility to the code. There, one can choose either to do the assignment or to choose two points to find the optimal path between them. This is done by choosing '1' or '2', as can be seen in the Fig 1:

```c
while(1){
    printf("\nChoose one option: (type '1' or '2')\n\n1. Assignment: from Bas lica
     de Santa Maria del mar (240949599) to Giralda (195977239).\n2. Choose two
    points.\n\nOption: ");
    scanf("%s",decision);
    if(strncmp(decision,"1",1) == 0){
        origin = &nodes[SearchNode(ORIGIN,nodes,nnodes)];
        dest = &nodes[SearchNode(DEST,nodes,nnodes)];
        break;
    }
    if(strncmp(decision,"2",1) == 0){
        printf("Where do you want to go? (Introduce the node's Id)\n");
        scanf("%lu",&destId);
        printf("Where do you come from? (Introduce the node's Id)\n");
        scanf("%lu",&originId);
        origin = &nodes[SearchNode(originId,nodes,nnodes)];
        dest = &nodes[SearchNode(destId,nodes,nnodes)];
        break;
    }
    printf("\nInvalid option, try again!\n");
}
```



```
Choose one option: (type '1' or '2')

1. Assignment: from Basílica de Santa Maria del mar (240949599) to Giralda (195977239).
2. Choose two points.

Option: _
```

Figura 1: Visual interface from the terminal.

Right after that, we have defined the following variables:

```c
queue *cua;
node *dest,*origin,*n,*m,*son;
unsigned long destId,originId;
double fprov,gprov;
char decision[10];
```

- *cua*: A pointer to a queue structure that will represent the priority queue.
- *dest* and *origin*: Pointers to node structures that will be the destination node (dest) and the origin node (origin).
- n and m: Pointers to node structures that will help us to navigate through the priority queue. In particular, as we will see, m is the next element of n.

- *son*: A pointer to a node structure that represents the *son* or *successor* node of the top element of the priority queue.
- *originId* and *destId*: Two large integers that represent the id of the origin and destination nodes.
- *fprov* and *gprov*: Provisional values of f and g of a node before introducing it to the priority queue.
- *decision*: A string that represents the input of the user.

With all of this, we can build the **A* implementation** part in our program:

```
origin->g = 0;
origin->f = 0;
cua->start = origin;
cua->end = origin;
while(cua->start != dest){
    cua->start->visited = 1;
    for(int i = 0; i < cua->start->nsucc; i++){
        son = &nodes[cua->start->successors[i]];
        if(son->visited == 1){continue;}
        gprov = cua->start->g + RealDist(cua->start,son);
        fprov = gprov + RealDist(son,dest);
        if(son->f != 0){
            if(fprov < son->f){
                if(cua->end != son){
                    son->previous->next = son->next;
                    son->next->previous = son->previous;
                }
                else{
                    cua->end = son->previous;
                    son->previous->next = NULL;
                }
            }
            if(fprov >= son->f){continue;}
        }
        n = cua->start;
        while(n != cua->end){
            m = n->next;
            if(fprov <= m->f){PutBewteen(n, m, son, cua->start,gprov,fprov);break;}
            else{n = m;}
        }
        if(n == cua->end){PutAtEnd(cua, son, gprov, fprov);}
    }
    DeQueue(cua);
}
```

The functioning of this part of the code is provided below. First we put the origin node to the priority queue with g and f values to 0. Right after that, we enter to a *while* that we will only *break* once the first element of the priority queue is the destination node. In each iteration, we expand the first element of the priority queue. This means we change to 1 its *visited* attribute and then we check its successors. For every one of this sons, we will first check if it has been already visited (in case it has, we will directly proceed to the next son). Then, we will calculate the value of the functions f and g associated to it, and will check if the node is already in the priority cue. If it is, we will keep the one that minimizes the value of f, either keeping the preexisting one in the list and go to the next son, or removing it from the list. Due to the differences in the removing method, in the code we differentiate if the son is at the end or not.

Once we have done this check, we enter to the second *while* that is the responsible for allocating the sons in the correct place of the priority queue. We define two consecutive nodes (m and n) that are going to move forward one element in each iteration and checks if the desired node to introduce has a lower value of f than m. Once this is satisfied, we introduce the element between n and m with the *PutBetween* function descrived before and we *break* the *while*. Otherwise, the while will stop when the second of the two consecutive nodes (m) reaches the last element of the queue. In case the node has not been introduced the program will introduce the node at the end of the queue with the *PutAtEnd* function.

Once we have expanded the node, meaning we have visited it and introduced its sons in the priority queue, we remove it from the queue with the *DeQueue* function and move forward to the next one to do the same, until the first element on the priority queue is the destination node.

Finally we find the output part of the code, where we print the distance on the terminal and write a *.txt* file with the output format to be able to represent the path with the provided *mapplot.py* code.

```c
FILE *output;
node *a,**path;
int index;
double dist_loop;
double *distances;

path = (node**)malloc(nnodes*sizeof(node*));
distances = (float*)malloc(nnodes*sizeof(float));

distances[0] = 0.0;
index = 0;
a = dest;
while(a != origin){
    path[index] = a;
    a = a->father;
    index++;
}
```

```c
path[index] = origin;
dist_loop = 0;
for(int i = 0;i<index;i++){
    distances[i+1] = RealDist(path[index-i-1],path[index-i]) + dist_loop;
    dist_loop = distances[i+1];
}
printf("Path found from %lu to %lu with distance: %.2lf meters.\n",origin->id,dest
    ->id,dist_loop);
fclose(fopen("output.txt", "w"));
output = fopen("output.txt","w");
fprintf(output,"# Distance from %lu to %lu: %lf meters.\n# Optimal path:\n",origin
    ->id,dest->id,dist_loop);
for(int i = 0; i<index+1;i++){
    fprintf(output,"Id=%lu | %lf | %lf | Dist = %lf\n",path[index-i]->id,path[index
    -i]->lat,path[index-i]->lon,distances[i]);
}
fclose(output);
```

For this last part we have defined 6 new variables:

- *output*: A file where we will store the output format.
- *a*: A pointer to a node structure that we will use to build the path by starting at *dest* and moving up through the parent attribute.
- *path*: A list of the node pointers that define the optimal path.
- *index*: An integer that helps us to build the path list.
- *dist_loop*: The distance from the origin to the destination.
- *distances*: A list of the distance between the origin and the node in the path. This is useful to write in the output format.

Compilation instructions

To compile the codes, we have used the standard *gcc* compiler without any optimize option. Since the *math.h* library is used, it is necessary to add the *-lm* extension to the compiler.

Execution instructions

To execute the first program, *ReadMap-WriteBin.c*, it is necessary to introduce as an argument for the main function the *.csv* file that contains the desired map in order to convert it to a binary file. To execute the second program, *AStar.c*, it is necessary to introduce as an argument the binary file created by the *ReadMap-WriteBin.c* code.

As we have mentioned, there will appear an interface in which you will be able to choose between to options for the *AStar.c* code. In order for the program to run the assignment it is necessary to type '1'. To introduce by hand the *id* of the destination and origin nodes, it is necessary to type '2' and then introduce the desired *id*s.

Results

With our code implementing the A* algorithm, we have obtained an optimal path with distance 958815.01 meters. The resulting path is represented in Fig. 2, while the path decided by Google Maps is represented in Fig. 3. It is necessary to emphasize that we might find discrepancies between both paths, since Google Maps minimizes the travel time and we minimize the distance traveled (which are not always equivalent).

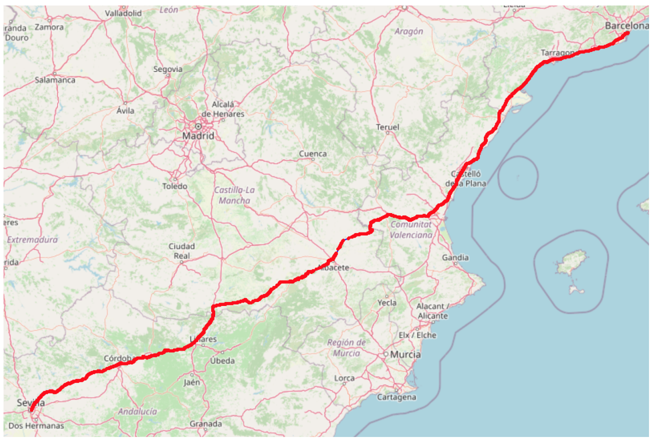


Figura 2: Path found by our A* program.

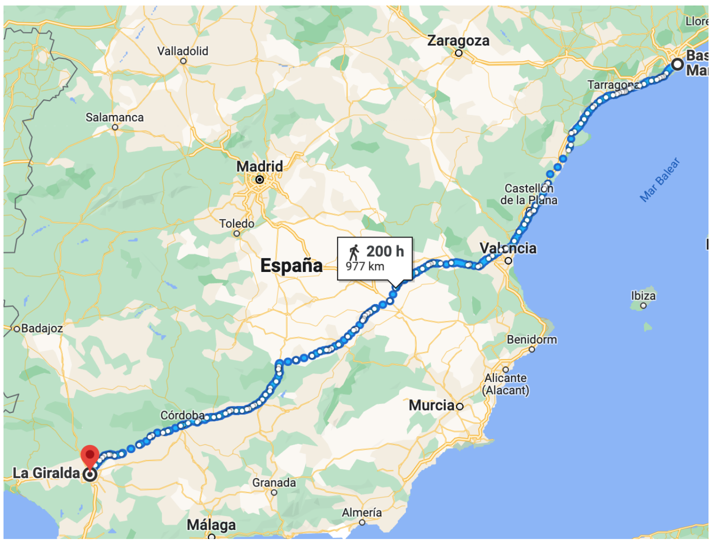


Figura 3: Path found by Google Maps.