Universitat Autònoma de Barcelona

The A* Algorithm

An Implementation Using C

Alfredo Hernández Ruth Kristianingsih

10th February 2018

Contents

1 Introduction

Finding shortest paths or optimal paths is a classic problem in graph theory that has been widely studied. It has enormous applications in many areas including computer science, operations research, transportation engineering, network routing network analysis, and specifically in vehicle routing. The use of Global Positioning System (GPS) and Google Maps has rapidly increased in recent days; finding the shortest path from one place to another place has become an ordinary daily task in modern society.

Several studies about shortest path search show the feasibility of using graphs for the purposes explained before. Dijkstra's Algorithm is one of the classic shortest path search algorithms. This algorithm is not well suited for shortest path search in large graphs. This is the reason why various modifications to Dijkstra's Algorithm have been proposed by several authors using heuristics to reduce the run time of shortest path search. One of the most used heuristic algorithms is the A* Algorithm, with the main goal of reducing the run time by reducing the search space.

This report shows the use of A* Algorithm in computing an optimal path from Basílica Santa Maria del Mar in Barcelona to the Giralda (Calle Mateos Gago) in Sevilla implemented in the C language. The A* Algorithm will be implemented as a function that will give the optimal path.

2 Theoretical Background

In this section, a short theoretical background of the topics studied in this research work will be introduced.

2.1 Single Source Shortest Path Problem

The Single-Source Shortest Paths (SSSP) problem, which calls for the computation of a tree of shortest paths from a given vertex in a directed or undirected graph with non-negative edge weights, is one of the most important and most studied algorithmic graph problems.

Let $G = (V, E, w)$ be a weighted directed graph with two special vertices, and we want to find the shortest path from a source vertex s to a target vertex t. That is, we want to find the directed path p starting at s and ending at t that minimizes the function:

$$w(p) \coloneqq \sum_{u \to v \in p} w(u \to v). \tag{1}$$

The shortest paths are unique, because any subpath of a shortest path is itself a shortest path. There are some algorithms that are usually used to solve SSSP problem, namely Dijkstra's Algorithm, Bellman–Ford Algorithm, Floyd-–Warshall Algorithm, A* Algorithm, etc.

2.2 The A* Algorithm

A* which is a slight generalization of Dijkstra's algorithm, is the most popular choice for path finding, because it is fairly flexible and can be used in a wide range of contexts. A* achieves better performance by using heuristics to guide its search. This heuristic was first described by Peter Hart, Nils Nilsson, and Bertram Raphael [1].

The secret to its success is that it combines the pieces of information that Dijkstra's Algorithm uses (favoring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, $g(n)$ represents the exact cost of the path from the starting point to any vertex n, and $h(n)$ represents the heuristic estimated cost from vertex n to the goal.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n) \tag{2}$$

The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node.

It is easier to understand the Algorithm using pseudocode. The following is the pseudocode of A* given by Prof. Lluís Alsedà [2]:

Algorithm 1: A* Algorithm Pseudocode

```
1  Put node_start in the OPEN list with f(node_start) = h(node_start)
2  while the OPEN list is not empty {
3    Take from the open list the node node_current with the lowest
4      f(node_current) = g(node_current) + h(node_current)
5    if node_current is node_goal {
6      we have found the solution; break
7    }
8    Generate each state node_successor that come after node_current
9    for each node_successor of node_current {
10     Set successor_current_cost = g(node_current) + w(node_current,
           node_successor)
11     if node_successor ∈ OPEN list {
12       if g(node_successor) ≤ successor_current_cost {
13         continue (to line 26 )
14       }
15     } else if node_successor ∈ CLOSED list {
16       if g(node_successor) ≤ successor_current_cost {
17         continue (to line 26 )
18       }
19       Move node_successor from the CLOSED list to the OPEN list
20     } else {
21       Add node_successor to the OPEN list
22       Set h(node_successor) to be the heuristic distance to node_goal
23     }
24     Set g(node_successor) = successor_current_cost
25     Set the parent of node_successor to node_current
26   }
27   Add node_current to the CLOSED list
28 }
29 if (node_current != node_goal) exit with error (the OPEN list is
       empty)
```

The goal node is denoted by `node_goal` and the source node is denoted by `node_start`. In A*, we maintain two lists, which are `OPEN` and `CLOSED`. `OPEN` consists on nodes that have been visited but not expanded (meaning that successors have not been explored yet). This is the list of pending tasks. `CLOSED` consists on nodes that have been visited and expanded (successors have been explored already and included in the open list, if this was the case).

This is the the algorithm in which we will base our implementation.

3 Implementation of the Program

An efficient approach to the problem (similar to what happens in GPS industry) is to write two programs:

- `write_bin.c`: reads the `.csv` files and computes the graph with binary search and stores the graphs in a binary file with arrows between nodes already determined. This file is thus very quick to read. This processes will be discussed in § 3.2
- `a-star.c`: a second program that reads this formatted binary file (getting the map already in graph form), asks for the start and goal nodes, and then performs the A* Algorithm.

3.1 Data Preprocessing

Unfortunately, the file is not consistently formatted. There are ways with less than two nodes (that have to be discarded) and there are nodes in ways that do not appear in the list of nodes. For this reason, to ease the process of creating a binary file, we have decided to preprocess the raw `spain.csv` file using AWK.

First thing we do is to is to understand the structure of the `.csv` file. There are three kind of records: `node`, `way`, and `relation`. Since we want to build a graph connecting nodes, we only care about nodes and ways. In particular, we are interested in the following fields:

- `$1`: record type (node or way).
- `$2`: `@id` of the node or way.
- `$3`: `@name` of the node or way.

For nodes we have the following exclusive fields:

- `$10`: latitude of the node.
- `$11`: longitude of the node.

Whilst for ways we have the following exclusive fields:

- `$8`: `@oneway`, a boolean indicating if the way is unidirectional.
- `$10`–EOL: member nodes connected by the way.

Having this in mind, we first use AWK to create a file containing only the nodes, whilst keeping the `|` separator:

Script 2: `get_nodes.awk`

```
BEGIN{
  FS = "|"
  OFS = "|"
}
FNR > 3 {
  if ($1 == "node") {
```

```
7       print $1, $2, $3, $10, $11
8     }
9   }
```

For the ways we create a similar AWK program, with the difference that we already make sure that the ways have more than one member node (`if (NF >= 11)`), so we have to resort to use `printf` instead of the standard `print`:

Script 3: `get_ways.awk`

```
1  BEGIN{
2    FS = "|"
3  }
4  FNR > 3 {
5    if ($1 == "way" && NF >= 11) {
6      printf("%s|%s|%s", $1, $2, $8);
7      for (i=10; i<=NF; ++i) printf("|%s", $i);
8      printf("\n")
9    }
10 }
```

To use the AWK programs we just need to execute them in the following way:

```
1 awk -f get_nodes.awk data/spain.csv > data/spain-nodes.csv
2 awk -f get_ways.awk data/spain.csv > data/spain-ways.csv
```

3.2 Creating a Binary File

Structure of the data

To store the data into a binary graph format, we need to use structures (or structs). A struct in the C programming language is a composite data type declaration that defines a physically grouped list of variables to be placed under one name in a block of memory, allowing the different variables to be accessed via a single pointer. Thus, it has the role that a class would have (although they can do much more than a struct) in object-oriented programming languages.

We will define the `Node` struct to store the variables contained in each of the nodes:

```
1 typedef struct {
2   unsigned long id;               // Node identification
3   unsigned short name_lenght;
4   char *name;
5   double lat, lon;                // Node position
6   unsigned short num_successors; // Node successors: wighted edges
7   unsigned long *successors;
8 } Node;
```

Since the nodes' `id`s do not follow a structured pattern, it is recommended to store the nodes in an array of node structures and refer to the nodes internally in the program by

the index in this vector (not by the `id`). This is way of naming nodes allows a quicker way of finding them.

This `nodes` array is defined in the following way:

```
Node *nodes;
unsigned long nnodes = 23895681UL;
nodes = (Node *) malloc(nnodes*sizeof(Node));
```

An array for storing the `ways` is initialized in a similar fashion:

```
unsigned long *ways;
ways = (unsigned long *)malloc(sizeof(unsigned long) * 5306);
```

Reading the `.csv` Files

Basically this process consists on reading the two preprocessed files we built with AWK:

- A file containing all the nodes (`spain-nodes.csv`).
- A file containing all the ways (`spain-ways.csv`).

For these operations we will work with a `FILE *file` variable which is dynamically opened and closed, and with the `str*` functions (from the `string.h` library) to work with tokens.

The process of reading the nodes can be seen in the snippet below:

```
file = fopen("data/spain-nodes.csv", "r");

while((bytes_read = getline(&buffer, &bufsize, file)) != -1) {
  field = strsep(&buffer, "|");
  field = strsep(&buffer, "|");
  if((nodes[i].id = strtoul(field, '\0', 10)) == 0) printf("ERROR: Conversion failed.\n");
  field = strsep(&buffer, "|");
  nodes[i].name_lenght = strlen(field) + 1;
  nodes[i].name = (char *) malloc(nodes[i].name_lenght*sizeof(char));
  strcpy(nodes[i].name, field);
  field = strsep(&buffer, "|");
  nodes[i].lat = atof(field);
  field = strsep(&buffer, "|");
  nodes[i].lon = atof(field);
  i++;
}
```

The process of reading the ways and getting the successors of each node can be seen in the snippet below. Notice that we use the `perform_binary_search()` function to check if a node is present in the list of member nodes of each way.

```
file = fopen("data/spain-ways.csv", "r");

unsigned long a, b, A, B, member, way_lenght, index;
```

```
4  bool oneway;
5
6  for (i = 0; i < nnodes; i++) {
7    if((nodes[i].successors = (unsigned long *)malloc(sizeof(unsigned
       long)*16)) == NULL) {
8      printf("Memory␣allocation␣for␣the␣successors␣failed␣at␣node␣
         %lu\n", i);
9      break;
10   }
11 }
12
13 i = 0;
14 while((bytes_read = getline(&buffer, &bufsize, file)) != -1) {
15   i++;
16   field = strsep(&buffer, "|");
17   field = strsep(&buffer, "|");
18   field = strsep(&buffer, "|");
19   if (strlen(field)>0) {
20     oneway = 1;
21   } else {
22     oneway = 0;
23   }
24
25   while((field = strsep(&buffer, "|")) != NULL) {
26     member = strtoul(field, '\0', 10);
27     if((index = perform_binary_search(member, nodes, nnodes)) !=
         ULONG_MAX) {
28       ways[n] = member;
29       n++;
30     }
31   }
32   way_lenght = n;
33   n = 0;
34
35   for (j = 0; j < way_lenght - 1; j++) {
36     A = ways[j];
37     a = perform_binary_search(A, nodes, nnodes);
38     B = ways[j + 1];
39     b = perform_binary_search(B, nodes, nnodes);
40
41     nodes[a].successors[nodes[a].num_successors] = b;
42     nodes[a].num_successors++;
43
44     if(oneway == 0) {
45       nodes[b].successors[nodes[b].num_successors] = a;
46       nodes[b].num_successors++;
47     }
48   }
49 }
```

Binary Search

Binary search is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful. If the search ends with the remaining half being empty, the target is not in the array.

The main reason to use this algorithm is that it runs at worst (and in average) in logarithmic time, specifically it is $\mathcal{O}(\log_2(n))$, being a much faster algorithm that a Sequential Search.

For this purpose, we have developed the following `perform_binary_search()` function:

```
unsigned long perform_binary_search (unsigned long key, unsigned
    long *list, unsigned long lenlist) {
  register unsigned long start = 0UL, afterend = lenlist, middle;
  register unsigned long try;
  while ( afterend > start ) {
    middle = start + ((afterend-start-1)>>1); try = list[middle];
    if (key == try) {
      return middle;
    } else if ( key > try ) {
      start = middle + 1;
    } else {
      afterend = middle;
    }
  }
  return ULONG_MAX;
}
```

Storing into a Binary File

This process is strongly based on the methods provided by Prof. Lluís Alsedà in [3]. In short, we write the following data into the `spain-data.bin` binary file:

(i) A header specifying the number of nodes, the total number of successors, and the total number of characters.

(ii) An array containing all the nodes (including their internal `Node` structure).

(iii) An array containing all the successors.

(iv) An array containing all the names.

These processes can be seen in the snippet below:

```
const char filename[] = "data/spain-data.bin";

unsigned long num_total_successors = 0UL;
for (i = 0; i < nnodes; i++) {
  num_total_successors += nodes[i].num_successors;
}
```

```
7
8  unsigned long num_total_name = 0UL;
9  for (i = 0; i < nnodes; i++) {
10   num_total_name += nodes[i].name_lenght;
11 }
12 printf("Starting␣to␣write␣the␣bin␣file.\n");
13
14 if ((file = fopen (filename, "wb")) == NULL) printf("The␣output␣
       binary␣data␣file␣cannot␣be␣opened.\n");
15
16 // Global data --- header
17 if( fwrite(&nnodes, sizeof(unsigned long), 1, file) +
18 fwrite(&num_total_successors, sizeof(unsigned long), 1, file) +
19 fwrite(&num_total_name, sizeof(unsigned long), 1, file)!= 3 ) {
20   printf("Error␣when␣initializing␣the␣output␣binary␣data␣file.\n");
21 }
22
23 // Writing all nodes
24 if( fwrite(nodes, sizeof(Node), nnodes, file) != nnodes ) {
25   printf("Error␣when␣writing␣nodes␣to␣the␣output␣binary␣data␣
         file.\n");
26 }
27
28 // Writing sucessors in blocks
29 for (i = 0; i < nnodes; i++) {
30   if (nodes[i].num_successors) {
31     if (fwrite(nodes[i].successors, sizeof(unsigned long),
           nodes[i].num_successors, file) != nodes[i].num_successors) {
32       printf("Error␣when␣writing␣edges␣to␣the␣output␣binary␣data␣
             file.\n");
33     }
34   }
35 }
36
37 // Writing names in blocks
38 for (i = 0; i < nnodes; i++) {
39   if (nodes[i].name_lenght) {
40     if (fwrite(nodes[i].name, sizeof(char), nodes[i].name_lenght,
           file) != nodes[i].name_lenght) {
41       printf("Error␣when␣writing␣names␣to␣the␣output␣binary␣data␣
             file.\n");
42     }
43   }
44 }
45
46 fclose(file);
```

3.3 Reading a Binary File

This process is also based on the methods provided by Prof. Lluís Alsedà in [3], and is the *inverse* operation of storing the graph into the binary file. In short the process consists on:

(i) Initializing the `file_in` file variable.

(ii) Opening the `spain-data.bin` file using `fopen` with reading permissions.

(iii) Reading the header of the binary, this is needed to allocate the memory.

(iv) Allocating the memory needed for the variables.

(v) Actually reading the data and storing into the variables defined before.

(vi) Once everything has been read without issues, closing the `file_in` file.

These processes can be seen in the snippet below:

```
1  FILE *file_in;
2
3  if ((file_in = fopen ("data/spain-data.bin", "r")) == NULL) {
4    printf("The␣data␣file␣does␣not␣exist␣or␣cannot␣be␣opened\n");
5  }
6
7  /* Global data --- header */
8  if ( fread(&nnodes, sizeof(unsigned long), 1, file_in) +
9  fread(&num_total_successors, sizeof(unsigned long), 1, file_in) +
10 fread(&num_total_name, sizeof(unsigned long), 1, file_in) != 3 ) {
11   printf("Error␣when␣reading␣the␣header␣of␣the␣binary␣data␣file\n");
12 }
13
14 /* getting memory for all data */
15 if ((nodes = (Node *) malloc(nnodes*sizeof(Node))) == NULL) {
16   printf("Error␣when␣allocating␣memory␣for␣the␣nodes␣vector\n");
17 }
18 if ((allsuccessors = (unsigned long *)
       malloc(num_total_successors*sizeof(unsigned long))) == NULL) {
19   printf("Error␣when␣allocating␣memory␣for␣the␣edges␣vector\n");
20 }
21 if ((allnames = (char *) malloc(num_total_name*sizeof(char))) ==
       NULL) {
22   printf("Error␣when␣allocating␣memory␣for␣the␣names\n");
23 }
24
25 /* Reading all data from file */
26 if ( fread(nodes, sizeof(Node), nnodes, file_in) != nnodes ) {
27   printf("Error␣when␣reading␣nodes␣from␣the␣binary␣data␣file\n");
28 }
29 if ( fread(allsuccessors, sizeof(unsigned long),
       num_total_successors, file_in) != num_total_successors ) {
30   printf("Error␣when␣reading␣sucessors␣from␣the␣binary␣data␣file\n");
31 }
32 if ( fread(allnames, sizeof(char), num_total_name, file_in) !=
       num_total_name ) {
```

```
33    printf("Error␣when␣reading␣names␣from␣the␣binary␣data␣file\n");
34  }
35  fclose(file_in);
```

After reading this, we need to input the `allsuccessors` and `allnames` arrays into the `nodes[i].successors` and `nodes[i].name` variables of the `Node` struct:

```
1  //Setting pointers to successors
2  for (i = 0; i < nnodes; i++) {
3    if (nodes[i].num_successors) {
4      nodes[i].successors = allsuccessors;
5      allsuccessors += nodes[i].num_successors;
6    }
7  }
8
9  //Setting pointers to names
10 for (i = 0; i < nnodes; i++) {
11   if (nodes[i].name_lenght) {
12     nodes[i].name = allnames;
13     allnames += nodes[i].name_lenght;
14   }
15 }
```

3.4 Heuristic Cost Function

For the heuristic cost function $h(n)$ we have decided to use the Haversine distance formula.

The haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes. Important in navigation, it is a special case of a more general formula in spherical trigonometry, the law of haversines, that relates the sides and angles of spherical triangles.

$$\begin{aligned} a &= \sin^2\left(\frac{\Delta\varphi}{2}\right) + \cos(\varphi_1)\cdot\cos(\varphi_2)\cdot\sin^2\left(\frac{\Delta\lambda}{2}\right) \\ c &= 2\cdot\operatorname{atan2}\left(\sqrt{a}, \sqrt{1-a}\right) \\ d &= R\cdot c \end{aligned} \tag{3}$$

where φ is latitude, λ is longitude, and R is the mean radius of the Earth ($R = 6371\,\text{km}$), and $h(n) = d$ is the distance between the two points (along a great circle of the sphere). The function $\operatorname{atan2}(x, y)$ returns the principal value of the arc tangent of y/x.

It is crucial to have in mind, this formula supposes the Earth as spherical, fact that in our range of distances is a very good approach, and does not account for irregularities in the surface

This heuristic has been implemented in the `distance()` function below:

```
double distance (Node *nodes, unsigned long node_start, unsigned
    long node_goal) {
  double R = 6371000;
  double lat1 = nodes[node_start].lat * (M_PI/180);
  double lat2 = nodes[node_goal].lat * (M_PI/180);
  double lon1 = nodes[node_start].lon * (M_PI/180);
  double lon2 = nodes[node_goal].lon * (M_PI/180);
  double delta_lat = lat2 - lat1;
  double delta_lon = lon2 - lon1;
  double a = sin(delta_lat/2) * sin(delta_lat/2) + cos(lat1) *
     cos(lat2) * sin(delta_lon/2) * sin(delta_lon/2);
  double c = 2 * atan2(sqrt(a), sqrt(1-a));
  return R * c;
}
```

3.5 Dealing with the Queue

As stated in § 2.2, we need to deal with a queue to track the status of each of the nodes. For this we use the `AStarStatus` struct and a `whichQueue` enumerated type:

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

typedef struct {
  double g_cost, h_cost;
  unsigned long parent;
  Queue queue_status;
} AStarStatus;
```

In the `main` algorithm the `AStarStatus` is used to define the `status` variable, with which we can easily update the values of $g(n)$, $h(n)$, and the status of the node in the queue:

```
AStarStatus *status;
status = (AStarStatus *)malloc(sizeof(AStarStatus)*nnodes);

status[node_start].g_cost = 0;
status[node_start].h_cost = distance(nodes, node_start, node_goal);
status[node_start].queue_status = OPEN;
```

To deal with the `OPEN` list, we used a linked list. A linked list is a data structure that consists of sequence of nodes. Each node is composed of two fields: data field and reference field which is a pointer that points to the next node in the sequence. This allows us to work with a dynamical `Node` structure:

```
typedef struct Node {
  double f_cost;
  unsigned long index;
  struct Node *next;
} DynamicNode;
```

To dynamically change the `OPEN` list we need to define the following functions: (i) `append()` to add items, (ii) `pop_first()` and `pop_by_index()` to remove items, and (iii) `index_minimum()` to find the item with with the minimum $f(n)$ (or `f_cost`) and return its index.

3.6 The A* Algorithm

Our implementation of the A* Algorithm as a function is nothing more than a fully-fledged version of Algorithm 1: it uses the `Node` struct to define the nodes, the `DynamicNode` struct and its linked list functions for the `OPEN` list, the `AStarStatus` to track the status of the Queue, the Haversine `distance()`, and last but not least the starting and goal nodes.

The `astar_algorithm()` function can be seen below:

```
1  inline void astar_algorithm (Node *nodes, DynamicNode *open_list,
       AStarStatus *status, unsigned long node_start, unsigned long
       node_goal, unsigned long nnodes ) {
2    double successor_current_cost;
3    unsigned long node_current;
4
5    int i = 0;
6    while ((node_current = index_minimum(open_list)) != ULONG_MAX) {
7
8      if (node_current == node_goal) {
9        break;
10     }
11
12     for (i = 0; i < nodes[node_current].num_successors; i++) {
13       unsigned long node_successor =
             nodes[node_current].successors[i];
14       successor_current_cost = status[node_current].g_cost +
             distance(nodes, node_current, node_successor);
15
16       if (status[node_successor].queue_status == OPEN) {
17         if (status[node_successor].g_cost <= successor_current_cost){
18           continue;
19         }
20       } else if (status[node_successor].queue_status == CLOSED) {
21         if (status[node_successor].g_cost <= successor_current_cost)
               continue;
22
23         status[node_successor].queue_status = OPEN;
24         append(&open_list, successor_current_cost +
               status[node_successor].h_cost, node_successor);
25       } else {
26         status[node_successor].queue_status = OPEN;
27         status[node_successor].h_cost = distance(nodes,
               node_successor, node_goal);
28         append(&open_list, (successor_current_cost +
               status[node_successor].h_cost), node_successor);
```

```
29        }
30
31        status[node_successor].g_cost = successor_current_cost;
32        status[node_successor].parent = node_current;
33      }
34      status[node_current].queue_status = CLOSED;
35
36      if ((pop_by_index(&open_list, node_current)) != 1) {
37        printf("Removal failed.\n");
38      }
39    }
40    if (node_current != node_goal) {
41      printf("OPEN list is empty.\n");
42    }
43
44    print_path(nodes, status, node_start, node_goal, node_current,
          nnodes);
45  }
```

To print the paths in a readable `.csv` format, we defined the function `print_path()` as follows:

```
1  inline void print_path (Node *nodes, AStarStatus *status, unsigned
       long node_start, unsigned long node_goal, unsigned long
       node_current, unsigned long nnodes) {
2    // Print the path
3
4  unsigned long *path;
5    unsigned long node_next;
6    int i = 0;
7
8    node_next = node_current;
9    path = (unsigned long *)malloc(nnodes*sizeof(unsigned long));
10   path[0] = node_next;
11   while (node_next != node_start) {
12     i++;
13     node_next = status[node_next].parent;
14     path[i] = node_next;
15   }
16   int path_length = i + 1;
17
18   // Header
19   printf("Node ID, Distance (m), Lat, Lon\n");
20   // Print nodes
21   for (i = path_length-1; i >= 0;i--) {
22     printf("%lu, %.2f, %.5f, %.5f \n", nodes[path[i]].id,
           status[path[i]].g_cost, nodes[path[i]].lat,
           nodes[path[i]].lon);
23   }
24 }
```

We will use the resulting `solution.csv` file to plot the solution into a illustrative map using the R script `create_maps.R`.

3.7 Compiling and Running the Program

Since some of the functions for the A* Algorithm have been defined in a separate `functions.c`, we need to include it in the compilation process with GCC. Compiling it using the `-Ofast` flag we get a quite faster compiled binary (nonetheless, in § 4 we will include the times for the base run and the version with `Ofast`).

To compile and run the program that writes the binary file, we use:

```
gcc -lm -Ofast write_bin.c -o write_bin
./write_bin
```

To compile and run the A* Algorithm, we use:

```
gcc -lm -Ofast functions.c a-star.c -o astar
./astar
```

Notice that the programs are executed in a quite powerful machine, but without an Solid State Disk. Therefore, a clear bottleneck will be the reading and writing speed of the Hard Drive Disk.

4 Results and Discussion

Results of the preceding implementation in § 3 will be presented in this section. Mainly, the objective of this analysis is finding optimal path from Barcelona to Sevilla. In fig. 1, it is shown the map of optimal path computed with the distance 958 815.01 m with a number of optimal nodes of 6649 as seen in table 1. As a performance indicator, we used `perf stat`; the program took around 11.972 s to obtain and print the optimal path to a `solution.csv` file.

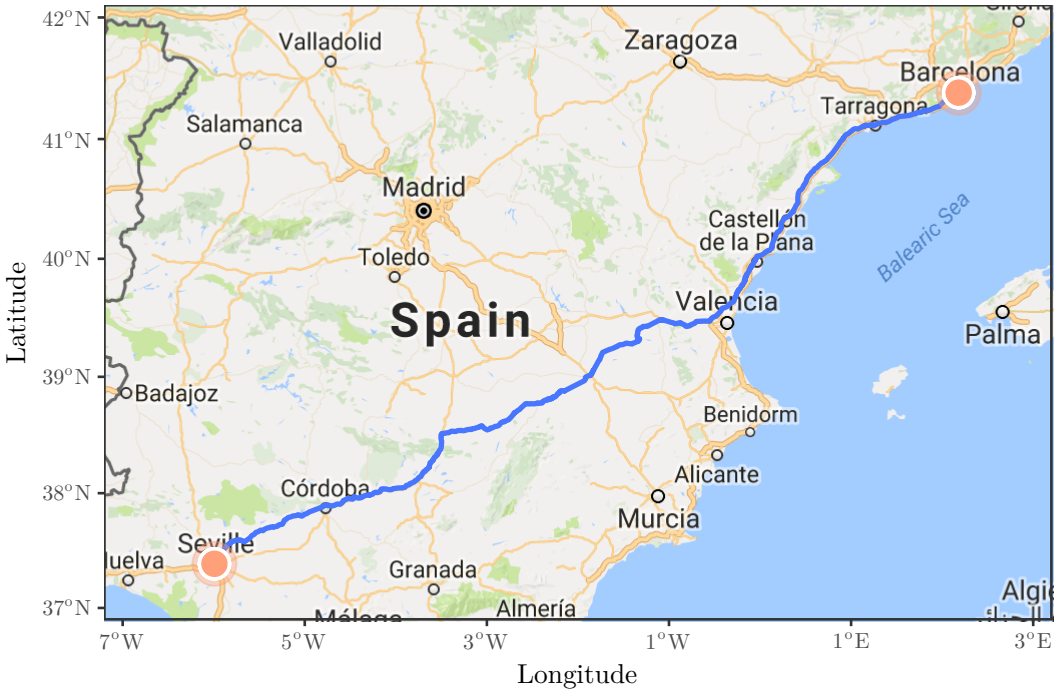

Figure 1: Optimal Path from Barcelona to Sevilla

Creating the binary file	28.817 s
Creating the binary file (`Ofast`)	28.732 s
Finding the optimal path	11.972 s
Finding the optimal path (`Ofast`)	8.599 s
Optimal path distance	958 815.01 m
Number of nodes	6649

Table 1: Information about the Performance and Results of Implemented Program

In case one wanted to apply the A* Algorithm to two arbitrary nodes in Spain, we have implemented another version of the main program (`a-star-input.c`) that allows the

user to select the start and goal nodes[1].

To test this interactive version of the program, we tried to find the optimal path in smaller area using the implemented algorithm from Barcelona (node 240949599) to UAB (node 255403621):

```
gcc -lm -Ofast functions.c a-star-input.c -o astar-input
./astar-input 240949599 255403621
```

The optimal path computed is illustrated in fig. 2. The distance is around 19 839.10 m with the number of nodes is 331, and the program need about 0.916 s to get the results as summarized in table 2.

Figure 2: Optimal Path from Barcelona to UAB

Finding the optimal path	0.916 s
Finding the optimal path (`Ofast`)	0.819 s
Optimal path distance	19 839.10 m
Number of nodes	331

Table 2: Information about the Performance and Results of Implemented Program

[1]One can easily find node IDs using `https://nominatim.openstreetmap.org`, or grepping the `.csv` database

4.1 Further Improvements

Some different experiments, for instance changing the heuristic functions have been left for the future work due to lack of time. There are some ideas that we would like to try during the implementation of A* algorithm. This analysis report mainly focused on finding path from Barcelona to Sevilla, leaving the finding optimal path from different places. For this reason, we propose some ideas to be tested and implemented as part of future work and further improvements as follows:

(i) Try different heuristic functions and compare the performance of the algorithm.

(ii) Include larger area to be tested to find optimal path, for instance whole area of Europe (we are not even sure the whole Spain area is included, as some of the nodes we tried to experiment with were not in the `.csv` file).

(iii) For the latter, we would like to define some of the variables in a less *hardcoded* way, especially when dealing with sizes memory allocation.

(iv) Trying to find an optimal path from Barcelona to Madrid the A* Algorithm got apparently stuck in a cyclic loop at some point. This is because a node in the `CLOSED` list could end up back again in the `OPEN` list. This is an issue that would be interesting to address.

(v) Improve the performance of the programs in terms of memory. Surely some variables could have been freed after not needing them anymore.

References

[1] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. (37):28–29, 1972. URL: http://portal.acm.org/citation.cfm?doid=1056777.1056779.

[2] L. Alsedà. A* Algorithm pseudocode. URL: http://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf.

[3] L. Alsedà. Reading and Writing Binary files in C. URL: http://mat.uab.cat/~alseda/MasterOpt/ReadingWriting-bin-file.pdf.

[4] C Linked List. URL: http://www.zentut.com/c-tutorial/c-linked-list/.

[5] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. 1984, pages 4–6. ISBN: 0201055945. DOI: 10.1016/0306-4573(85)90100-1.

[6] L. Sean. *Essentials of Metaheuristics: A Set of Undergraduate Lecture Notes*. 2010, pages 29–52. ISBN: 9780557148592. DOI: 10.1007/s10710-011-9139-0.

[7] J. Erickson. *Shortest Paths (Lecture Notes)*. 2014.