Universitat Autònoma de Barcelona

CENTRE DE RECERCA MATEMÀTICA

A Star Algorithm

Optimisation

Pol Pastells Vilà and Narcís Font Massot
Master's degree in Modelling for Science and Engineering

January 2021

1 Introduction

The aim of this project is the implementation of an A* algorithm to solve a routing problem. Given a graph, find the optimal path between two designated nodes, with respect to the distance. We are going to use different heuristic strategies and compare them in terms of performance.

We have build a program that implements the A* algorithm using C language. It can be used to solve general routing problems given the coordinates of the start and the goal location, with the data formatted in an appropriate way.

1.1 Motivation

We are going to compute the optimal path (according to distance) from *Basílica de Santa Maria del Mar (Plaça de Santa Maria)* in Barcelona to *Giralda (Calle Mateos Cago)* in Sevilla using the A* algorithm.

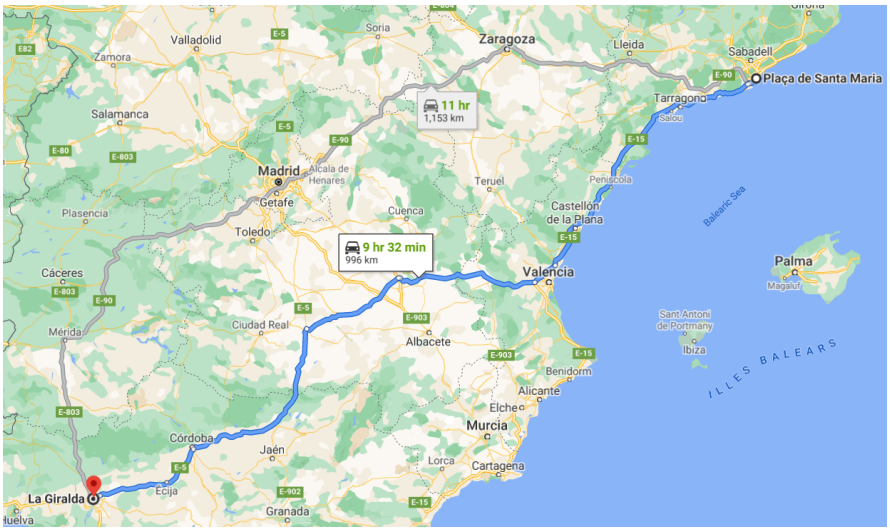

Figure 1: Paths suggested by Google Maps. The blue path is the shorter one. Note that Google minimizes the time in order to find these paths.

Our goal is to find a path similar to the given by Google Maps (Figure 1), but minimizing the distance instead of the time. We will be using a data map of Spain in csv format that contains all the nodes and connections information. This dataset can be downloaded from [1]. The starting node key we will be using its (@id): 240949599 and the final node key (@id): 195977239.

2 A* Search Algorithm

The A* Algorithm is based on Dijkstra's Algorithm. It builds upon it by adding an heuristic function to estimate the distance between two nodes. For it to work, we need some extra information, like the geographical position of each node. Both algorithms are formulated in terms of weighted graphs, where the weight between two nodes represents the (positive) distance.

2.1 Algorithm description

A* Algorithm is an informed search algorithm [2], or best-first search. This algorithm solves problems by searching among all possible paths from the starting point to the goal point (Solution) and chose the ones with smallest cost. At each iteration of the A* main loop, we need to determine which of the possible paths expand. It does this based on an estimation of the cost function until the goal node is reached. A* selects the path that minimizes:

$$f(n) = g(n) + h(n)$$

Where n is the last node of the path, $g(n)$ is the cost (total weight) of the path from the start node, and $h(n)$ is the heuristic function that estimates the cost of the cheapest path from n to the goal node. The pseudocode of the algorithm can be fount at the annex of this document.

2.2 Heuristics

In order to minimize the distance, we are going to use as heuristic function some distance functions that estimate the distance between the given node and the goal node. The heuristics that we are going to use are the following [3]:

- No heuristic (Dijkstra)
- Haversine formula
- Spherical law of cosines
- Equirectangular approximation

3 Code Implementation

In this section, we are going to detail a few aspects of the program implementation. It was mainly programmed using C Programming Language, however, we developed a small python script to display the resulting path in a map. The versions used are the following:

- gcc 9.3.0, using -Ofast flag
- Python 3.8.5

In order to achieve a better code efficiency we separate the code into two parts. The first one is for prepossessing purposes. It reads the file and computes the graph using binary search. Then stores it for the main program to read. The second, the main program, runs the A* algorithm. Additionally, a utilities and auxiliary files are used to define functions and structures.

To execute the program, we first need to place the csv file with all the map information inside the *data* folder, and then compile it using `compile.sh`. The detailed instructions on how to run the program are in the `README.md` file for the project.

3.1 Binary file

Due to the complexity of reading the data file, that contains lots of information (1,4 GB), it is useful to preprocess the data and create a binary file (`binary.bin`) with all the map information. With this, we can read directly the data stored in the binary file and archive a better performance for multiple runs of the program. The creation of the binary file takes about $30s$.

3.2 Visualization

To get a better understanding of the results obtained, we have created a python script that displays the solution on a map (`visualization.py`). To run this script is necessary to install the *gmplot* library. All the plots can be found on the *results* folder.

4 Results

Now, we are going to discuss the performance and the optimal paths obtained with our A* algorithm implementation using different heuristic strategies.

4.1 Performance

The execution time is divided between the time spend during the binary file reading and the time elapsed by the A* algorithm.

Heuristic / Time (s)	Binary file reading	Algorithm	Total
Dijkstra (no heuristic)	1.81	993.48	996.29
Haversine formula	0,53	5,70	6,98
Spherical law of cosines	0,73	5,72	7,18
Equirectangular approximation	0,62	5,31	6,68

Table 1: Execution time for the different heuristic strategies.

If we take a look on the results exposed in the Table 1, we can appreciate that the execution times are pretty similar for the different heuristic strategies. As expected, the majority of time is spent on the algorithm execution. The case with no heuristic takes two orders of magnitude more time to run the algorithm. The time spent reading the binary file should be the same for all heuristics, given that the code is exactly the same, they are different due to the machine they were ran on.

4.2 Optimal path

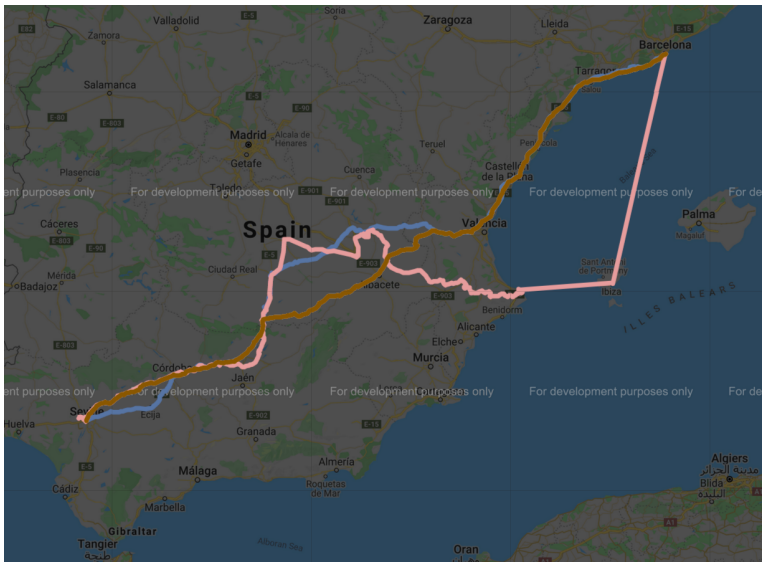

Figure 2: Optimal path obtained for the heuristic strategies (orange), Dijsktra path (pink), Google Maps shortest path by time (blue), obtained via Google Maps UI.

The optimal path obtained using the different strategies is the same for all three cases with an heuristic, and has a length of $960km$. The solution for the Dijkstra algorithm goes through the sea. Although we didn't expect it, it is not really a bad solution given the collection of nodes we fed the algorithm. That said, the solution is much longer, $1,279km$.

Comparing the result obtained (Figure 2) with the one using Google Maps (Figure 1), we see that the path we have found is approximately $30km$ shorter. We remark that this doesn't mean our path is faster, as previously exposed, we are minimizing distance and Google Maps minimizes time.

5 Conclusions

We developed an A* algorithm in C using three different heuristics. We use the trajectory for going from a point in Barcelona to one in Sevilla given by Google Maps as a benchmark, given that it is minimized for travel time by car, and ours minimizes distance. As expected, our solution is shorter, by about $30km$, not depending on the heuristic used. The case without heuristic, Dijkstra algorithm performs way worse. It's solution is almost $300km$ longer and took two orders of magnitude more time to converge.

References

[1] Lluís Alsedà, *Optimisation*, http://mat.uab.cat/~alseda/MasterOpt/index.html , January 2021.

[2] Wikipedia, *A* search algorithm* , https://en.wikipedia.org/wiki/A*_search_algorithm , January 2021.

[3] Chris Veness, *Movable Type Scrips* , http://www.movable-type.co.uk/scripts/latlong.html , January 2021

Appendix

A* pseudocode

The goal node is denoted by `node_goal` and the source node is denoted by `node_start`

We maintain two lists: **OPEN** and **CLOSE**:

OPEN consists on nodes that have been visited but not expanded (meaning that sucessors have not been explored yet). This is the list of pending tasks.

CLOSE consists on nodes that have been visited *and* expanded (sucessors have been explored already and included in the open list, if this was the case).

```
1  Put node_start in the OPEN list with f(node_start) = h(node_start) (initialization)
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 we have found the solution; break
6    Generate each state node_successor that come after node_current
7    for each node_successor of node_current {
8      Set successor_current_cost = g(node_current) + w(node_current, node_successor)
9      if node_successor is in the OPEN list {
10       if g(node_successor) ≤ successor_current_cost continue (to line 20)
11     } else if node_successor is in the CLOSED list {
12       if g(node_successor) ≤ successor_current_cost continue (to line 20)
13       Move node_successor from the CLOSED list to the OPEN list
14     } else {
15       Add node_successor to the OPEN list
16       Set h(node_successor) to be the heuristic distance to node_goal
17    }
18     Set g(node_successor) = successor_current_cost
19     Set the parent of node_successor to node_current
20   }
21   Add node_current to the CLOSED list
22 }
23 if(node_current != node_goal) exit with error (the OPEN list is empty)
```