A* algorithm assignment

Optimization

Nieves Montes Gómez (1393150)

07/01/2020

1 Introduction

The purpose of this assignment is to implement the A* algorithm [1] to find an optimal path from Santa Maria del Mar church in Barcelona (41.3833411°N, 2.1817741°E) to Giralda in Sevilla (37.3860989°N, 5.9917423°W). The objective is to find an optimal path by minimizing the distance that is shorter than that provided by Google Maps. The algorithm is applied on a graph that is constructed from a provided .csv file.

In order to implement A*, it is necessary to define a distance function. In this case, I have used the Haversine formula, which corresponds to the great-circle distance between two points on a sphere [3]. The role of this formula is to compute both the cost of edges (*i. e.* the distance between adjacent nodes) and the heuristic function of the nodes $h(n)$, which is calculated as the Haversine distance to the destination node.

2 Implementation

2.1 Program

The code that implements the solution is organized into different files, all of which are included in the `Code` folder:

- `write_main.c` contains the main function of the program that reads the map .csv file, builds a graph from it and exports it to a .bin file.
- `Astar_main.c` contains the main function that reads the graph from the .bin file, and directs the user to choose the IDs of starting and destination nodes as well as the type of evaluation function he/she wishes to use. There are three option: `default`, `weighted` and `dynamic weighting`. If the choice is `weighted` or `dynamic weighting`, the user must also input the parameter `w` or `e` (ϵ) respectively. More information on the different evaluation functions implemented is provided in a later section. Finally, it calls the `AStar()` function to compute the optimal path.
- `Astar_header.h` contains all the necessary `include` calls to standard C libraries, the defined macros (the radius of the earth in km `R` and the π number `pi`), custom structures to represent a node in the graph (`typedef struct node`), the g and f functions for a node as it is updated while A* is running (`typedef char Queue`, `enum whichQueue` and `typedef struct AStarStatus`) and a node in the `OPEN` list (which is implemented as a linked list, `typedef struct open_node`). Finally, it contains all necessary functions

that are called both by the main function in `write_main.c` and by the `AStar()` function. Following is a summary of such functions:

  - `ExitError()` is the standard function to exit with error.
  - `process_node()` takes in a line of type `node` in the .csv file and extracts its ID, name (if there is one), latitude and longitude.
  - `binary_search()` locates a node in the `nodes` vector by its index given its ID. If there is not a node with the given ID, it returns -1.
  - `update_nsucc()` takes in a line of type `way` in the .csv file and updates the successors counts in the nodes involved, according to whether the way is bidirectional or not.
  - `update_successors()` takes in a line of type `way` in the .csv file and updates the adjacency lists of the involved nodes accordingly, taking into account whether the way is bidirectional or not.

    For the last two functions (`update_nsucc()`, `update_successors()`), when a node is encountered that is not in the `nodes` vector, it is skipped. In order not to interrupt paths, if node b (not in the nodes vector) is between nodes a and c, the program updates the successors and creates an edge between a and c.
  - `haversine()` computes the Haversine distance between two given nodes.
  - `evaluation_function()` computes the f function for a node of given index according to the choice of evaluation function, its parameter (if any), the current g and h values for the node and the information of the source and destination nodes. The last two arguments, however, are only utilized if the evaluation type is of type dynamic weighting.
  - `print_OPEN()` is an auxiliary function to keep track of the `OPEN` list as the algorithm proceeds. It is recommended only to use when testing the program between two very close-by nodes.
  - `insert_to_OPEN()` computes the f function of a given node and inserts it in the `OPEN` linked list by preserving the ascending ordering of f. If two nodes have the same f values (which in practice is extremely rare when dealing with floating-point values), the new node is inserted after the one already residing in the `OPEN` list.
  - `delete_from_OPEN()` deletes a node from the `OPEN` list given its index. In practice, this function is used in tandem with `insert_to_OPEN()` to update the value of f of nodes already residing in `OPEN` and their position in the list.
  - `is_path_correct()` takes in a vector of nodes' indices and checks that the path they describe is indeed correct, *i. e.* that all the nodes in the path are in the adjacency list of the previous one.
  - `path_to_file()` creates a "|"-separated file containing the information of all nodes that make up a path. The name of such file has the following format: (spain or cataluna)_(ID of source)_(ID of destination)_(evaluation function)_(parameters if any).csv. The produced file contains the following information for all the nodes that make up the path: position in path, ID, latitude, longitude, optimal g computed, h and name (if any).

- `Astar_func.h` only contains the definition of the `AStar()` function, which actually implements the algorithm. It takes in the vector of nodes (*i. e.* the graph that the algorithm is to be applied on), the IDs of source and destination, the number of nodes in the graph, the name of the .bin file from which the graph has been read, the evaluation function to be employed and its parameter (if any). After allocating all necessary memory and

declaring all the involved variables, it reads the CPU time right before entering the main `while` loop. Then, right after the loop is exited, it reads the CPU time again.

If the computation is successful and a path has been found, the function displays the optimal distance that has been found (*i.e.* g(destination)), the number of nodes that have been expanded from the `OPEN` list, the time it has taken the `while` loop to execute and the number of nodes in the path. Next, it checks that the computed path makes sense by calling the `is_path_correct()` function. If so, it calls the `path_to_file()` function and finally frees all the allocated memory that has not been previously liberated.

Note: Some nodes have duplicate successors. However, this is not an issue when running the algorithm, because all nodes fill in only one position of the `nodes` vector. Hence, if when running A* it happens to be updating the second copy of a duplicate successor, the algorithms will find that the successor is either in the `OPEN` or `CLOSE` lists. If it is in the `OPEN`, it will not update it because its cost is already the one obtained if the successor is reached from the current node (provided that this is the lowest cost found for that successor at that point in the program). If the duplicate successor is in the `CLOSE` list, it will stay there (more on that in the next section).

2.2 Implications of a monotone heuristic function

Despite not being explicitly checked, it is assumed that the heuristic function used in this problem:

$$h(n) = \text{Haversine distance}(n, \text{destination}) \tag{1}$$

is monotone. Intuitively, it makes sense that this is the case. The assumption of a monotone heuristic has some simplifying computational implications:

- By Theorem 10, Chapter 3 in [2], a monotone heuristic leads A* to find optimal paths to all expanded nodes. Therefore, once a node has been pushed into the `CLOSE` list after expansion, it will stay there until the algorithm halts. In terms of coding, once a successor is found to be in the `CLOSE` list, the A* `while` loop just skips to the next successor.
- By Theorem 11, Chapter 3 in [2], "Monotonicity implies that the f values of the sequence of nodes expanded by A* is non-decreasing". This statement simplifies the `insert_to_OPEN()` function, as we need not consider the insertion of a node at the begining of the list, only at the end or at intermediate positions.

2.3 On the evaluation function

This implementation of A* allows the user to choose between three evaluation functions:

- Default: This corresponds to the typical case:

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

  Assuming A* to be admissible (*i.e.* it is guaranteed to find an optimal solution if one exists), the solution provided by using default evaluation will be set as reference for comparison.

- Weighted evaluation: This depends on a weight parameter w:

$$f_w(n) = (1 - w) \cdot g(n) + w \cdot h(n) \tag{3}$$

Provided A* is admissible, the same algorithm with weighted evaluation is guaranteed to also provide an optimal solution for $0 \leq w \leq \frac{1}{2}$ [2, page 87]. However, such statement might not be applicable to range $w > \frac{1}{2}$.

- Dynamic weighting: It is an evolved version of weighted evaluation where the heuristic part $h(n)$ progressively looses influence as the search approaches the destination. It depends on the parameter ϵ, and it originally takes the following form:

$$f(n) = g(n) + h(n) + \epsilon \cdot \left[1 - \frac{d(n)}{N}\right] \cdot h(n) \tag{4}$$

where $d(n)$ is the depth of the node and N the depth of the destination. Again, assuming A* is working with an admissible heuristic, the same algorithm with dynamic weighting is guaranteed to find a path of maximum length $(1+\epsilon) \cdot$ (length of optimal path). Hence, the parameter ϵ can be considered to be a tolerance for error in the computed length, traded-off for faster convergence.

Given that the depth of the destination node is unknown *a priori*, the program implements a different version of dynamic weighting, where the last term in equation (4) is substituted by:

$$f(n) = ... + \epsilon \cdot \left[1 - \frac{dist(n, \text{source})}{dist(n, \text{source}) + dist(n, \text{goal})}\right] \cdot h(n) \tag{5}$$

where $dist(n, \text{source})$ and $dist(n, \text{goal})$ are the Haversine distances from the node in question to the source and goal nodes respectively.

3 Results

All static maps presented in this report are available in an interactive version in the folder `Maps`. The raw results of all the computed paths presented in this report are also provided in .csv files ('|'-separated) in the folder `Paths`. Whenever two paths are claimed to be equal, their raw output .csv files have been compared with the bash command `cmp` and have been found to be indeed identical. Both programs (`write_main.c` and `Astar_main.c`) have been compiled with the `-O3` optimization flag.

3.1 Default evaluation function

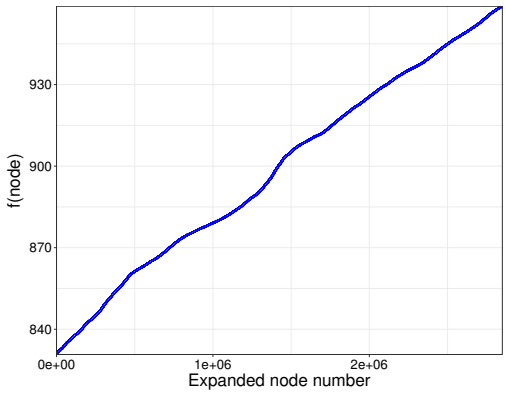

Figure 1: Sequence of f values for the expanded nodes of A* with default evaluation. This plot shows that the sequence of f values is non-decreasing, which is compatible with the heuristic function being consistent.

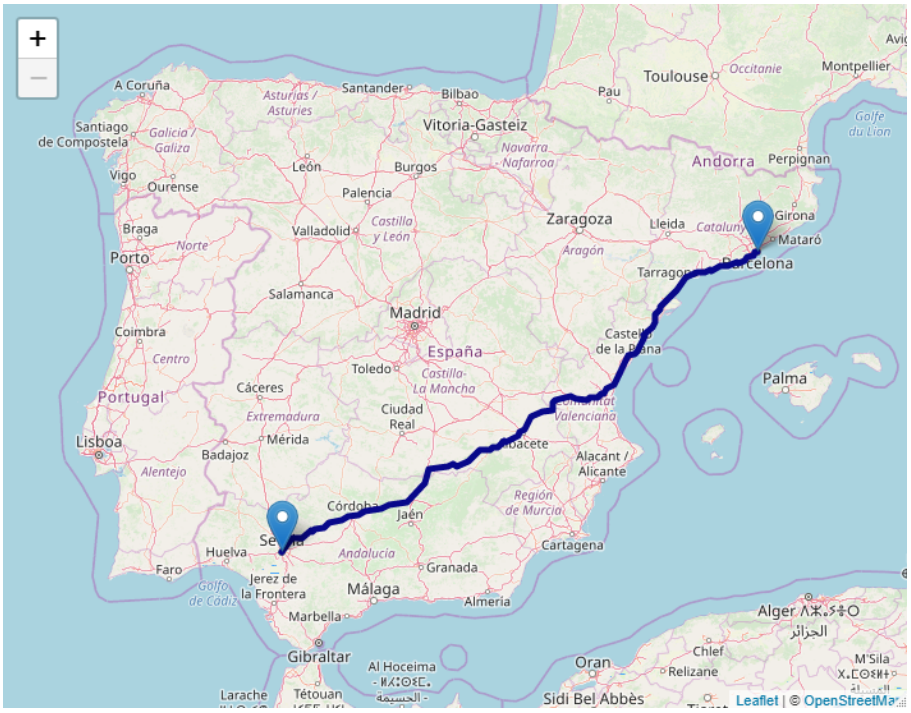

Figure 2: Computed route with A* using default evaluation.

3.2 Weighted evaluation

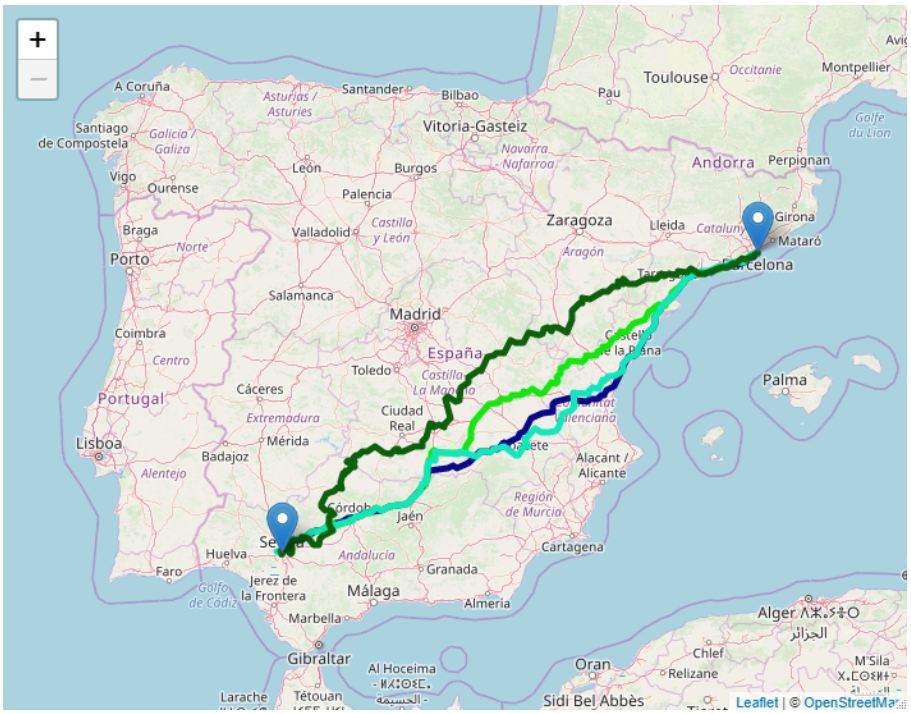

Figure 3: Computed route with A* using weighted evaluation. Legend: dark blue for $w = 0, 0.25, 0.5$; bright green $w = 0.60$; light blue $w = 0.75$; dark green $w = 1$. Only distinguishable routes have been included. The interactive version of the map contain the routes for all the explored w parameters (see Table 1).

3.3 Dynamic weighting

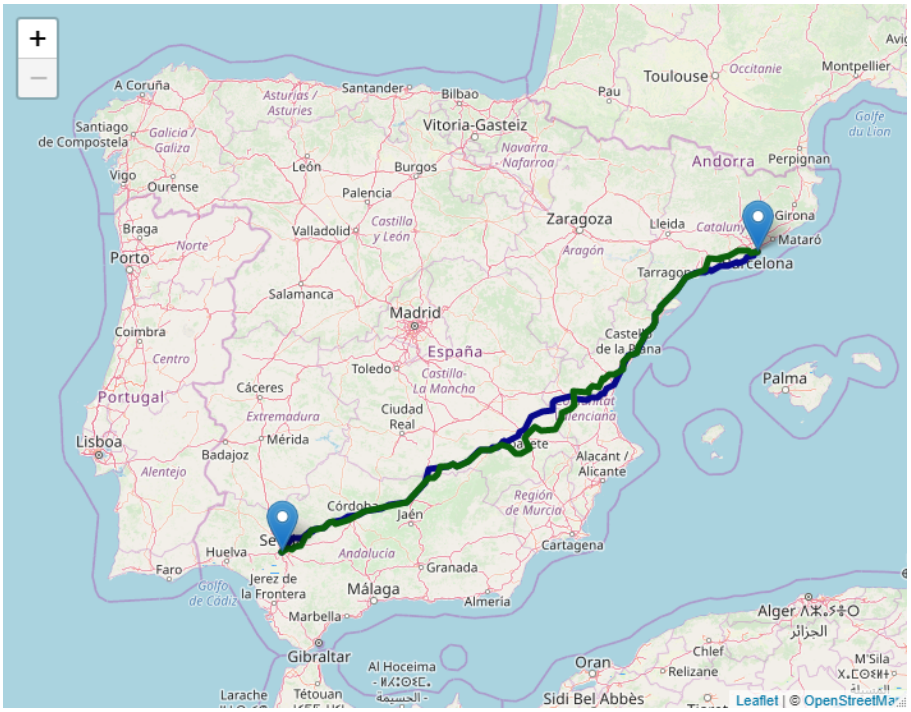

Figure 4: Computed route with A* using dynamic weighting. Legend: dark blue for $\epsilon = 0.0001, 0.001$; light blue $\epsilon = 0.30$; dark green $\epsilon = 1$. Only distinguishable routes have been included. The interactive version of the map contain the routes for all the explored ϵ parameters (see Table 1).

3.4 Summary

The results of many program executions using the various evaluation methods and parameters can be found in Table 1. Execution with default evaluation is considered to be the optimal solution. The optimal distance it calculates (≈ 959 km) is shorter than that suggested by Google Maps on foot (which the shortest among all transportation forms). Also, as it is displayed in Figure 1, the sequence of values of f for the expanded nodes is non-decreasing. This observation is consistent (however does not provide confirmation) with the heuristic function being monotone.

Weighted evaluation with $w \leq 0.5$ computes the same path as default evaluation. However, for strictly $w < 0.5$ the Dijsktra part $g(n)$ is the largest term in the f function, the algorithm is more exploratory and the number of expanded nodes greatly increases. Consequently, so does the computation time. Execution with $w = 0.5$ is equivalent to default evaluation. For $w > 0.5$, weighted evaluation makes A* loose its admissibility, as expected. The distance found then is further from the optimal as the weight of the heuristic function increases. Due to the increasing heuristic component, the number of expanded nodes also diminishes, and so does the computation time. Given these results, weighted evaluation is recommended for w values moderately above 0.5 (0.5 - 0.57).

Dynamic weighting with $\epsilon \leq 0.001$ produces the same path as the default. A few less nodes are expanded, however, regarding the execution time, this effect is cancelled given that the evaluation function is now computationally more costly. Setting ϵ to 0.1-0.2 is a good compromise to find a slightly longer route expanding many less nodes. Finally, $\epsilon = 1$ finds a far-from-optimal

Table 1: Summary of the results of running A* with the different evaluation functions and various parameters.

Evaluation	Parameter	Distance (km)	Nodes in path	Expanded nodes	Time (s)
Default	None	958.81501	6,649	2,850,435	4.34
Weighted	$w = 0$	958.81501	6,649	12,343,811	19.78
	$w = 0.25$	958.81501	6,649	10,567,296	17.46
	$w = 0.50$	958.81501	6,649	2,850,435	3.83
	$w = 0.52$	959.79674	6,732	1,809,632	2.78
	$w = 0.55$	968.96538	7,030	112,066	0.0714
	$w = 0.57$	982.63609	7,638	37,291	0.0134
	$w = 0.60$	1,039.65369	7,396	26,951	0.00925
	$w = 0.75$	1,105.59375	8,000	12,468	0.00471
	$w = 1$	1,374.97015	12,298	14,041	0.00531
Dynamic weighting	$\epsilon = 0.0001$	958.81501	6,649	2,849,455	4.90
	$\epsilon = 0.001$	958.81501	6,649	2,841,657	4.70
	$\epsilon = 0.01$	958.81577	6,639	2,762,840	4.49
	$\epsilon = 0.10$	963.68211	7,592	2,100,058	4.22
	$\epsilon = 0.15$	966.41284	6,661	1,384,531	2.14
	$\epsilon = 0.20$	968.01862	6,750	398,296	0.449
	$\epsilon = 1$	1,067.01809	8,788	40,558	0.0286

path in almost no time. Again, the strong heuristic component in this case leads to the algorithm being highly exploitative and expanding very few nodes. Despite the resulting distance not being optimal, it is still far from the worst case scenario ($(1 + \epsilon) \cdot C^* \sim 1900$ km).

4 Conclusions

In this assingment, the A* algorithm has been applied to compute the route between two given points. To do so, three types of evaluation function have been implemented: the classical default, weighted evaluation and dynamic weighting. The result obtained for default evaluation is considered to be optimal and a reference for comparison of the other methods. Further, the sequence of f values of the expanded nodes does not contradict the heuristic function (Haversine distance to the goal node) being consistent. Up to this point, the objective of the assingment has been fulfilled, as the calculated route is shorter than that provided by Google Maps. Alternative choices of the evaluation function are recommended with a reasonable parameter choice, so that the heuristic aspect of the algortihm does not overshadow the exploratory component.

References

[1] Lluís Alsedà. *A* algorithm pseudocode. Lecture notes in Optimization.*

[2] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984, p. 399. ISBN: 0-201-05594-5.

[3] Wikipedia the Free Encyclopedia. *Haversine formula.* 2019. URL: https://en.wikipedia.org/wiki/Haversine%7B%5C_%7Dformula.