A* Algorithm

Author: Oriol Moreno, Joan Pareras

(Dated: December 12, 2021)

Abstract: Route optimization is the process of determining the most cost-efficient route. The main goal of this project is to find the optimal path between two nodes in a graph. In this case, the nodes represent two different locations in Spain. ***A**** is the graph traversal algorithm used to solve this problem. In this report, the performance of this algorithm in achieving the given task will be studied.

I. INTRODUCTION

A* is a graph traversal and best-first search algorithm. First presented in 1968 by Peter E. Halt, Nils J. Nilsson and Bertram Raphael [1], it is computationally efficient, complete and finds the optimal solution. This algorithm can be seen as an improvement of Dijkstra algorithm, achieving better performance thanks to an heuristic function which, if admissible, will find the best solution.

From the start node until arriving to the end node, the algorithm visits the nodes allocated in the graph following a tree-descend order, considering the weight associated to each node. This weight is determined by the minimum path from the starting node to the actually visited path summed to the weight given by an heuristic function.

II. *A** FUNDAMENTALS

As previously mentioned ***A**** algorithm is a graph traversal algorithm, based on best-first search algorithms. This algorithm can be seen as an improvement on Dijkstra algorithm.

While Dijkstra algorithm computes the distance from a starting node to an end node, without taking into account if the distance to the end node is increasing, ***A**** algorithm adds a certain weight to each node finding the optimal path as quick as possible. This weight is added using an heuristic function. The visited node at each step is that with minimum value of $f(v) = g(v)+h(v)$, where v is the node visited, $g(v)$ is the cost of the path from starting to actual node and $h(v)$ is the heuristic function.

Despite being computationally efficient, its major drawback is that it stores all visited nodes in memory having a worst case space complexity of $O(b^d)$ where b is the branching factor and d the depth of the solution.

As mentioned earlier, ***A**** will find the optimal path, but only if the heuristic is admissible. In order to be admissible the heuristic can never overestimate the cost of reaching the goal from the current point in the path.

The heuristic function is problem-specific, meaning that it will return the minimal path from start to goal taking into account the criterion implemented in the heuristic function, when it is admissible. For instance, as we will see in the following sections, the criterion followed in this case, is to minimize the geodesic distance between visited nodes and the goal.

Usually, ***A**** uses a priority queue to select the nodes with minimum cost to expand. At each step of the algorithm, the first node of the priority queue (with lowest f) is removed from the queue and its neighbours are updated accordingly to the values of f and g, and added or replaced in the queue. This iterative process continue until the node to be expanded is the goal node. Notice that the goal node will fulfill $f(v) = g(v)$, since h at the goal is zero in an admissible heuristic.

In order to keep track of the path, each node must save its parent predecessor. At the end, each node will point to its predecessor, until arriving to the starting node.

A. Pseudo-code

In this section, the Pseudo-code for ***A**** is presented as an schematic visual reference.

```
AStar(graph G, start, goal)
  PQ ← EmptyPriorityQueue
  parent[G.order] ← uninitialized
  g[G.order] ← initialized to ∞
  PQ.add_with_priority(start)
  while PQ not empty
    current ← PQ.extract_min()
    if (current is goal)
    then return g, parent
    for adj ∈ current.neighbours do
      adj_new_g ← g[current] + h(G, current, goal)
      if adj_new_g < g[adj] then
        parent[adj] ← current
        g[adj] ← adj_new_g
        if not PQ.BelongsTo(adj)
        then PQ.add_with_priority(adj, g)
        else PQ.requeue_with_priority(adj)
        end if
      end if
    end for
  end while
```

B. Comments on the Pseudo-code

The algorithm uses as variables a graph containing all the nodes information and a starting and ending nodes.

In this case, the graph contains the node id, the location defined by its latitude and longitude coordinates, the number of successors and a list of successors direction in memory.

At first, the priority queue is initialized along with two vectors of length equal to the number of nodes in the graph. The first vector contains a list of parent nodes of each visited node, and the other contains the distance from the visited node to the starting node following the path. As it will be explained later the distances have been computed using the Haversine formula.

Then, the starting node is added to the priority queue. As it is the first node, it is allocated as the first element on the queue.

Concerning the while loop, the condition of continuity implies having one node at least in the priority queue. This loop looks at first for the current node, which is the first element stored in the priority queue. If this current node is actually the end node, then the algorithm finishes. Otherwise, the loop continues.

The next step in the loop is to search for the successors of the current node and store them on the priority queue. The nodes are stored in the priority queue taking into account their score. This score is given as the sum of the current distance and a certain weight, computed by the heuristic function that will be discussed later.

When discovering nodes, if a node was previously visited, then the algorithm does not store it in the priority queue. If the node was already in the priority queue and the score is now smaller than before, it gets requeued with the updated score.

III. CODING DECISIONS AND COMMENTS

While coding this program, some decisions had to be made in order to optimize the performance of the code. In the following sections, the decisions and procedures followed in the development of our code will be discussed.

A. Structures

1. *Reading csv file and writing binary file*

The first step is to read the data necessary to execute the algorithm. This information is stored on a CSV(Comma Separated Values) file which contains the different nodes and ways of the map. In order to do it, a structure that holds the necessary information is needed.

In our code, the data has been stored in a structure called Node, which contains the identifier of the node, the latitude and longitude coordinates, the number of successors each node has and finally a pointer to the successors.

```
typedef struct {
    unsigned long id;
    double latitude, longitude;
    unsigned short nSuccessors;
    unsigned long *successors;
} Node;
```

A binary file will be written, so the CSV file has to be read only once. This will save time since reading a binary file is much faster than processing a text file. By doing this step and saving the number of neighbours each node has, there is no need to use a dynamically allocated structure like a linked list to save space. We just need extra space to create the binary file, since we cannot know the number of neighbours beforehand.

Notice, this structure does not save the name of each node. This is a way to optimize our code as it is not necessary for the computations required since we can look up for the ids of the different nodes before executing the code. If needed the only change required would be to add it to the struct.

2. *Reading binary file and* ***A**** *structures*

When reading the binary file the data is read and stored in the same structure as before. This is mandatory, as the binary file data has to be reinterpreted and stored in ram.

The additional structures used in the algorithm are AStarPath, QueueElement and AStarControlData. The first of this structures saves the g value and its parent, it is used to know the path followed by the algorithm and be able to reconstruct it. The next structure defines each element of the priority queue. Finally, AStarControlData is a struct that identifies if a node has already been visited or not.

B. Binary file

A hugely recommended procedure is to save the nodes data as a binary file, as the access to the information is optimal in comparison with other procedures. It requires the understanding of how has the data been stored, meaning that the way it is written must be the same way it is read. Generally, this is the best way to proceed when working with huge amounts of data.

The binary file stores all nodes in the previously explained Node structure, in order of appearance in the CSV file, and an array with all the successors of each node in the same order as before.

In order to read the binary file, the data is reorganized taking into account the size of the stored information.

Let's consider the first node. All the related information to this node is saved in bits, so that the node number one has $sizeof(unsignedlong) + sizeof(double) +$

$sizeof(double) + sizeof(unsignedshort) + sizeof(unsignedlong)$. Since the size of each data type is known we can reconstruct the structure by reading in succession the adequate amount of bytes.

The data of the neighbour nodes can be accessed by reading in succession the amount of bytes that these take up. This can be known since the number of successors of each node is already available.

C. A^* algorithm

The code for the algorithm follows the previously mentioned structure in the pseudo-code. The code used in this section is actually the same as the one proportioned in class, with some non-relevant changes and some different variable names to make it more clear.

We have chosen this implementation since it is very optimized an easy to follow.

D. Distance between nodes: Haversine formula

In our code, the distance between nodes is computed using the Haversine formula. This formula takes the latitude and longitude of two nodes, and approximating the Earth as an sphere of radius $R \approx 6.371.000$ meters, computes the geodesic between them. This formula takes the following form:

$$a = \sin^2(\frac{\Delta\phi}{2}) + \cos\phi_1 \cdot \cos\phi_2 \cdot \sin^2(\frac{\Delta\lambda}{2}) \qquad (1)$$

$$\text{Distance} = 2 \cdot \text{R} \cdot \arctan\,(\frac{\sqrt{\text{a}}}{\sqrt{1-\text{a}}}) \qquad (2)$$

where the subindex 1 and 2 refer to each node, ϕ is the latitude, λ is the longitude and $\Delta\phi$ and $\Delta\lambda$ are the difference between latitude and longitude of both nodes.

This equation has been chosen as it is very precise, even thought it considers the Earth as a perfect sphere.

The main issues with this formula are the following. Considering the Earth as a perfect sphere induces a significant error at the poles and the equator, where the Earth radius is smallest an biggest respectively in comparison with the approximated radius. But as Spain is in the tropic, the effects of this approximation are less important. Another approximation made, is to consider a flat surface, as the topography is relevant in measuring the actual distance.

Notice that because of this approximations, the real distance can differ about a $\approx 0.3\%$ [2] from the actual value, depending on the location. As shown in the figure, in Spain the relative error is almost negligible.

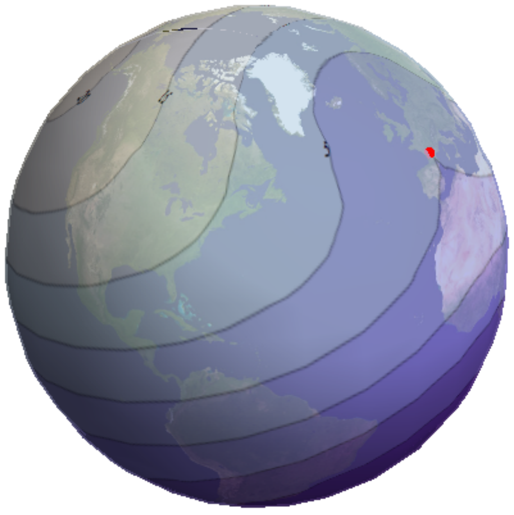

FIG. 1: This figure shows the relative error using Haversine formula as a function of the latitude and longitude. Where the line that begins at the red point, shows the 0% error value.[3]

E. Heuristic function

As previously mentioned, the heuristic function is problem-specific, meaning that it will return the minimal path from start to goal taking into account the criterion implemented in the heuristic function, when it is admissible. It determines the weight we associate to each node to minimize the computational time while finding the optimal path.

In our code, the travelled distance is minimized. To do it, the heuristic function computes the distance between the current node and the ending node, using the Haversine formula, which is the same distance used to compute the distance between nodes. This heuristic has been chosen as it is admissible, meaning that it is zero at the end node, and never overestimates the real cost.

Furthermore, Euclidean distance was tested but performed up to 5 times slower than the Haversine Distance. Also, it could have been considered the Spherical Law of Cosines, however the Haversine as it was slightly faster and historically been more used in this context.

IV. ALGORITHM PERFORMANCE

As previously explained, our code consists in two parts. The first part, which is reading the file and saving the information in a binary file, which takes the longest. This is caused by the slowness of reading the CSV file, which constrain the performance as mentioned earlier. The second part, computes the optimal path from the starting node to the end node minimizing the path length using the ***A**** algorithm.

The advantage of having the code separated in this two parts is that only one execution is required to convert the CSV data into binary. Then the other program only computes the desired path between two points using the binary data, which is much faster.

The computation time of the first code takes in our computer around 52.35 seconds to execute. As this code is executed only one time, it has not been fully optimized. In the possible improvements sections this topic will be discussed.

A. Example: Basílica de Santa Maria del Mar in Barcelona, to the Giralda in Sevilla

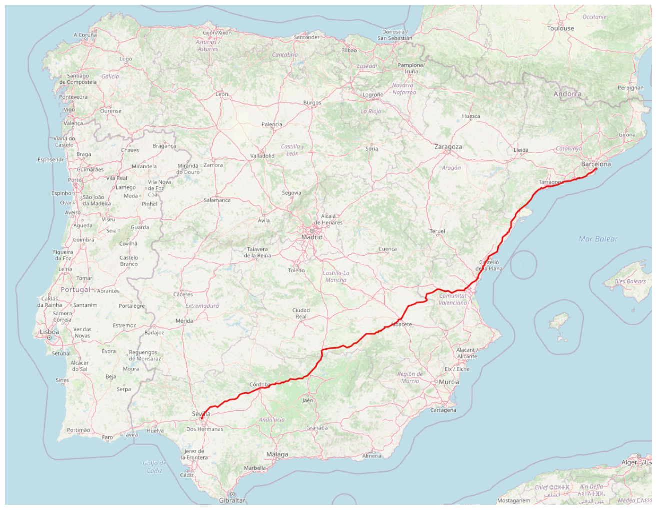

FIG. 2: Path from Basílica de Santa Maria del Mar in Barcelona, to la Giralda in Sevilla.

The second algorithm takes around 4.637 seconds to compute the optimal route from "Basílica de Santa Maria del Mar (Plaça de Santa Maria)" in Barcelona to the "Giralda (Calle Mateos Gago)" in Sevilla. With a path distance of 958815 m.

V. POSSIBLE IMPROVEMENTS

The most part of this algorithm is already optimized, but some important parts can be changed in order to obtain better results.

An example of these possible improves will be to improve the distance function, and therefore the heuristic in order to obtain a better performance. The distance has been computed using the Haversine formula, which approximates the Earth as a perfect sphere. A possible way to improve this idealization will be to parameterize for each node, the distance above the sea-level at each point being able to compute the actual real distance the track follows.

Another possible improvement is to make the code more user-friendly, as how it is actually written, if a different starting node and end node are wanted, these must be hard coded directly in the A^* main code.

It has also been attempted to precompute the distance between the nodes and their successors, and writing it into the binary file, in the same fashion as the successors are saved. This way, the execution of the algorithm after writing the binary file will be even faster. The main problem with this procedure is that it takes too much memory and some values end up being overflowed, causing some values of the node structure to take seemingly random values.

VI. INSTRUCTIONS FOR COMPILING

A. Code structure

- A-star algorithm
  - output
    - binaryDataSpain.bin
    - path.csv
  - res
    - spain.csv
  - src
    - Astar_algorithm.c
    - Astar_algorithm.exe
    - binarySearch.h
    - distanceFunction.h
    - Filevalues.h
    - csvtobinary.c
    - csvtobinary.exe
    - node.h
    - readBinary.h
    - strsep.h
    - structsAstar.h
    - writeBinary.h
  - README

The definition of each file can be found in the README file.

B. Executing process

In order to execute the code, the following steps must be followed:

- First, the csvtobinary.c needs to be executed. This will read the spain.csv file, save the values on the node structure and write the binary file
- Secondly, the Astar_algorithm.c needs to be executed. This will read the binary file, execute the A^* algorithm and will write the latitude and longitude of the final nodes in path.csv file, inside the output folder.

If the starting and ending nodes needs to be changed. In the Astar_algorithm.c we must change the nodes ids for ids of these other nodes.

```
line 40: unsigned start = binarySearch(nodes, 240949599UL, NUM_NODES_ESP);
line 41: unsigned end = binarySearch(nodes, 195977239UL, NUM_NODES_ESP);
```

for

```
line 40: unsigned start = binarySearch(nodes, (start node id)UL, NUM_NODES_ESP);
line 41: unsigned end = binarySearch(nodes, (end node id)UL, NUM_NODES_ESP);
```

VII. CONCLUSIONS

In sum, ***A**** algorithm finds the shortest path distance between two nodes minimizing the total path distance. In this case 958815 m, which is less than the one computed by Google Maps, as we are minimizing distance and not travelling time. Because of the followed procedure, with the binary file already written, our code takes only 4.637 seconds to compute the minimum distance path.

The major drawback of this algorithm is the need to reserve the memory with size as big as the graph. If for example we had a graph containing all the nodes in the world, a new approach would be needed.

[1] P. E. Hart, N. J. Nilsson, and B. Raphael, IEEE Transactions on Systems Science and Cybernetics **4**, 100 (1968).
[2] *Calculate distance, bearing and more between latitude/longitude points*, https://www.movable-type.co.uk/scripts/latlong.html, accessed: 2021-12-12.
[3] *How accurate is approximating the earth as a sphere?*, https://gis.stackexchange.com/questions/25494/how-accurate-is-approximating-the-earth-as-a-sphere#25580, accessed: 2021-12-12.