Optimization - A*

Jorge Medina (1618633), Marcell Veiner (1619878)

December 2021

1 Introduction

This report is written to discuss a solution to the well-known *Routing Problem*, which is about finding the optimal route from a starting position to a goal position. To be more precise, we will consider paths (Definition 1.2) on a weighted graph with directed edges (Definition 1.1). Then the optimal route (Definition 1.4) is a path from source to goal such that the distance (Definition 1.3) of the path is minimal.

Definition 1.1 (Weighted and Directed Graph) *A weighted and directed graph is an ordered triple* $G = (V, E, \phi)$ *such that*

- V *is the set of vertices or nodes.*
- E *is the set of edges, and* $E \subseteq \{(x, y) \mid x, y \in V \text{ and } x \neq y\}$.
- ϕ *is a map* $\phi : E \to \mathbb{R}$ *such that* $\forall (x, y) \in E \quad \phi((x, y)) = w \in \mathbb{R}$ *is the weight of the edge* (x, y).

Definition 1.2 (Path) *Let* $G = (V, E, \phi)$ *be a weighted and directed graph. A path* p *is a sequence of vertices* $p = (v_1, v_2, ..., v_n)$*, such that for all* $i = 1, ..., n-1$ *and* $j = 1, ..., n-1$*:*

- $(v_i, v_{i+1}) \in E$ *(consequently* $v_i \in V$*).*
- $v_i \neq v_j$ *whenever* $i \neq j$.

Definition 1.3 (Distance) *Let* $G = (V, E, \phi)$ *be a weighted and directed graph and* $p = (v_1, v_2, ..., v_n) \in V^n$ *a path. Then the distance of* p *is* $d(p) = \sum_{i=1}^{n-1} \phi((v_i, v_{i+1}))$.

Definition 1.4 (Optimal Route) *Let* $G = (V, E, \phi)$ *be a weighted and directed graph, then* $p = (v_1, v_2, ..., v_n) \in V^n$ *is the optimal route from* v_1 *to* v_n *if* $\forall p' = (v_1, v'_2, ..., v'_{n-1}, v_n)$ *it holds that* $d(p) \leq d(p')$.

A commonly used algorithm for finding the shortest path is *Dijkstra Algorithm*, which is a greedy search algorithm that computes a minimal spanning tree from a source vertex, therefore it **always obtains the optimal route** once the goal vertex is found. Although Dijkstra's is good for exploring unknown search spaces, it does not consider the location of the goal node, which is often known. Thus the algorithm spends relevant time exploring directions opposite to the goal node before exhausting more promising paths.

AStar (A*) is a variation of Dijkstra's Algorithm that makes use of the location of the goal node, thus it is called an informed search algorithm. A pseudocode of the algorithm can be found at [1]. It works by penalising paths based on a heuristic function (Section 2.3), which assigns a cost based on how far the current node is from the goal. This extra cost leads to more exploitation (that is exhausting promising paths) before resorting to exploration (trying new directions).

1.1 Heuristic functions

The A* algorithm will always find the optimal route whenever the heuristic function does not overestimate the optimal route distance. Such heuristics are known as admissible.

The difference between two admissible heuristics lies on its ability to predict the actual distance. Therefore, the performance of the A* algorithm depends heavily on the heuristic function, which is also problem specific. Finding the right heuristic function to use is critical for the application.

For graphs embedded in a metric space, the metric is always an admissible heuristic distance since it cannot overestimate the route distance. In this case, the metric is the geodesic distance over the surface of the Earth.

1.2 Problem Statement

With our definitions the Routing Problem can be stated as: *Given an graph G, source vertex v, and goal vertex w, find the optimal route p*. In our case, we will be working with the road network of Spain (G), obtained from OpenStreetMap [3], to compute optimal route (p) from Basílica de Santa Maria del Mar (v) in Barcelona to the Giralda (w) in Sevilla. The map contains longitudinal and latitudinal information about vertices, which are connected by roads (both one-way and two-way).

In the following sections we discuss:

1. How to build the graph from the downloaded map file.
2. How it is stored so that it optimises execution time

3. How to obtain efficiently the node with least cost
4. What heuristic function is more efficient
5. What is the effect of underestimating and overestimating the heuristic function

2 Programming Strategy

The algorithm was implemented in C, and is separated into two chunks. The first part is responsible for reading the map file and storing it in data structures that the A* algorithm can then use (the second part). However, reading the .csv file is considerably more time consuming than the actual algorithm. Moreover, most of the time such an algorithm would be called on the same map, but with different source and goal nodes, as compared to changing the map file between calls. Considering that reading binary files is orders of magnitude quicker than text files, it is worth writing the created data structures to binary file, which allows us to minimise the time the program spends with IO. We will start this section by discussing this section of the code.

2.1 Reading the Map

The given map file is a .csv file with "|" as a separator. The first useful information is a list of the nodes (one per line) from which we can obtain information about their id, name, latitude, and longitude. This information is read by the `map.h` script, by iterating through each line of the map file, and tokenizing that line with `strsep`. We make use of some specific information about the file, such as the fact that all nodes come before the roads between them, and relations, which are unrelated to our problem come last. Given this, we can stop reading when this section of the file is reached (see Listing 1).

```
...
while ((read = getline(&line, &len, fp)) != -1) {
  if (starts_with(line, "node")) {
    process_node(line, nodes, no_nodes);
    no_nodes++;
  }
  else if (starts_with(line, "way")) {
    process_way(line, nodes, no_nodes, no_ways);
    no_ways++;
  }
  else if (starts_with(line, "#")) continue;  //skip lines starting with #
  else break;  // relations are last, we can stop reading
}
...
```

Listing 1: Map Main Loop

Moreover, we know that there 23,895,681 nodes with a maximum degree (valence) of 9. Both of these could be obtained from an additional first pass through the map, but since they do not change between calls, they have been defined in macros. Moreover we used information about the other fields to determine their data types, which lead to node data structure (Listing 2). We allocate a fixed vector of the node data structure, and set the values for each node we read. The name of each node is allocated dynamically when reading, especially because most nodes do not have names defined, making it somewhat obsolete, but given another map with node names, this information could be used easily. The successors of a node are set when reading the ways (edges).

However, in order to set a node v as a successor of a node w, we first need to find w in the array of nodes. Although the node ids are too large to be used as indices, they appear in increasing order, and thus successive nodes in the array have successive node ids. This can be then used to speed up finding w, by using binary search instead of a sequential search, reducing the runtime from $\mathcal{O}(n^2)$ to $\mathcal{O}(log(n))$.

```
typedef struct{
  unsigned long id;  // Node identification
  char *name;
  double lat, lon;  // Node position
  unsigned short nsucc;  // Number of node successors; i. e. length of successors
  unsigned short nsucc_all;  // Number of successors allocated
  unsigned long *successors;  // Vector of successor indeces not ids
}node;
```

Listing 2: Node Data Structure

Ways only contain a chain of node ids making a way or road, and a field indicating if they are one-way, or two-way, resulting in two edges instead of one. This part is somewhat more complicated, as the file contains corrupt data i.e., ways with invalid node ids, and ways with only one node. The latter should be ignored completely, but in the first we should only discard the invalid node from the edge but assume a connection between the one before and after the invalid node.

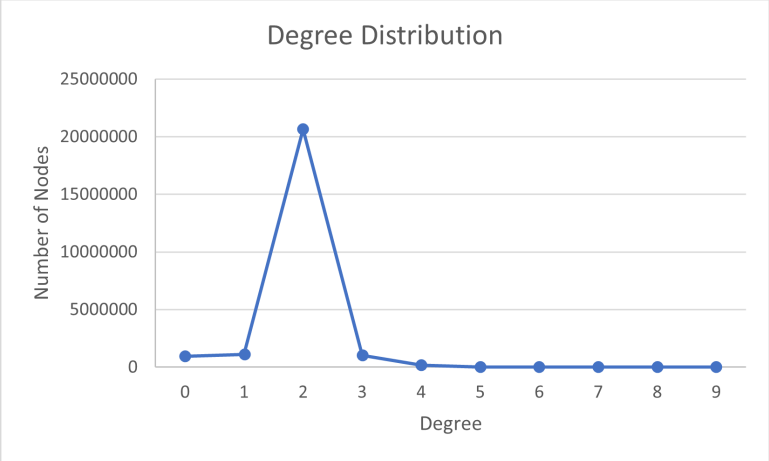

Figure 1: Degree Distribution

One simple strategy to set the successors would be to allocate enough memory to support the maximal degree for each node. However, given the degree distribution (Figure 1) this would be wasteful. A better strategy is to allocate memory dynamically, but this results either in realloc-s every time a node gets another successor, or an additional pass of the file. We have instead opted to reduce the number of realloc-s by making use of the `nsucc_all` field of the data structure, which contains information on how many successors we have space allocated. This together with a `ALL_STEP` parameter defined in a macro, reduce the realloc calls, by allocating memory for `ALL_STEP` successors at once (see Listing 3). Since the map is then written to binary, and re-read, the unused memory is reclaimed.

```
void allocate_succ(node * nodes, unsigned long pos){
  if (nodes[pos].nsucc_all == 0){
    // No memory allocated yet
    if ((nodes[pos].successors = (unsigned long *) malloc(ALL_STEP * sizeof(unsigned long))) == NULL)
      printf("Cannot allocate memory for successors \n");
    nodes[pos].nsucc_all = ALL_STEP;
  } else if (nodes[pos].nsucc_all == nodes[pos].nsucc){
    // Used all allocated memory
    if ((nodes[pos].successors = (unsigned long *) realloc(nodes[pos].successors, (nodes[pos].nsucc_all + ALL_STEP) * sizeof(unsigned long))) == NULL)
      printf("Cannot REallocate memory for successors \n");
    nodes[pos].nsucc_all = nodes[pos].nsucc_all + ALL_STEP;
  }
}
```

Listing 3: Allocate Successors

With the nodes and successors allocated, we can write the data to a binary file, the code for this is, and for reading it back, can be found in the `binary_IO.h` script. As these have not been modified, only wrapped in functions, they will not be discussed. We now turn our attention to the main algorithm.

2.2 The A* algorithm

The A* algorithm requires another data structure `(AStar_status)` that stores some information of the nodes:

- `g`: distance of the optimal known route from the source v_1 to a node v_i.
- `h`: heuristic cost of the node v_i.
- `parent`: previous node v_{i-1} to the node v_i in the optimal known route.
- `expanded`: whether the node has been expanded, i.e. removed from the priority queue.

Instead of another queue, as in some implementations of the AStar, we used the data structure in Listing 4, which kept track of the costs of the nodes, their parent nodes, and their visited status.

```
typedef struct {
    float g, h;
    unsigned long parent;
    bool expanded;
} AStarStatus;
```

Listing 4: AStar Status

With these procedures and data structures, the code could be compartmentalised and we obtained an easy to read and clear main loop for AStar (Listing 5). The `update_neighbours_distance` function explores the current node (if not already visited), calculates the distances of neighbouring nodes, and manages the queue by calling the relevant functions. More particularly, if a neighbour has not yet been visited, it is enqueued with the calculated distance, but in case the node has already been seen, but the new path is shorter its priority needs to be increased, and the queue reordered.

```
void astar(unsigned long source_index, unsigned long goal_index, node *nodes,
    unsigned long n_nodes, char *     path_route, char * stats_route, double
    heuristic_param, int save){
    AStarStatus *status = init_astarstatus(n_nodes);
    status[source_index].g = 0;

    PqElem *pq = init_pq(source_index, 0);
    unsigned long index;
    do{
        index = extract_min(&pq);
        status[index].visited = true;
        pq = update_neighbours_distance(index, goal_index, status, pq, nodes,
    heuristic_param);
    } while(index != goal_index);

    if (save != 0) save_results(source_index, goal_index, status, nodes, pq,
    path_route, stats_route);
    return;
}
```

Listing 5: AStar Function

2.2.1 Priority queue

To store the nodes to be visited, the algorithm uses a priority queue, ordered by the cost of the nodes, which is the sum of the cost from the source to the node, and the heuristic cost from the node to the goal. From the data structure of a queue element (Listing 6) we can see that under the hood the priority queue is a linked list. To keep a correct version of the queue, wee need procedures to initialise the queue, enqueue, dequeue, and to reorder it all of which can be found in the `priority_queue.h` script.

```
typedef struct PqElem{
    unsigned long index;
    double cost;
    struct PqElem *next;
}PqElem;
```

Listing 6: Prioity Queue Elements

The time complexity of the search is $O(n)$, being n the number of elements in the queue. In order to increase the priority of existent nodes, we should be able of finding the right position and the updated node without traversing the queue twice. This is achieved by **saving the position of the parent element** in the queue, then finding the desired element, and then rewiring it as the successor of the previously found parent.

2.3 The Heuristic Function

The heuristic functions have been separated from the rest of the code, and can be found in the `heuristics.c` script. We have implemented three formulas for calculating distances on a spherical earth based on [2]. We can switch between these formulas in the wrapper function for the heuristic calculations using a parameter supplied when running the program.

```
double heuristic(node current_node, node goal, int heuristic_func, double
    heuristic_param){
  double heu_dist;
  if (heuristic_param==0){ //Dijsktra case
    return 0;
  }
  switch (heuristic_func){
    case 3:
      heu_dist = equirectangular(current_node, goal);
      break;
    case 2:
      heu_dist = spherical_law(current_node, goal);
      break;
    case 1:
      heu_dist = haversine_dist(current_node, goal);
      break;
  }
  if(heuristic_param != 1.0){
    heu_dist = pow(heu_dist, heuristic_param);
  }

  return heu_dist;

}
```

Listing 7: Heuristic Main

The equirectangular heuristic calculates the distance between coordinates using Pythagoras' theorem and an equirectangular projection. This formula is only suggested because of its low computational needs, as it is the less accurate of the three. A somewhat more complicated but also more accurate method is the one called the spherical law of cosines, which uses trigonometric calculations on a spherical triangle to obtain the distance. Lastly, the most recommended option is the haversine formula which is well-conditioned for numerical computations, and calculates the distance using Formula 1.

$$\begin{aligned} a &= sin^2\left(\frac{\Delta\phi}{2}\right) + cos(\phi_1)\cdot cos(\phi_2)\cdot sin^2\left(\frac{\Delta\lambda}{2}\right) \\ c &= 2\cdot atan2\left(\sqrt{a}, \sqrt{1-a}\right) \\ d &= R\cdot c \end{aligned} \tag{1}$$

where ϕ is latitude, λ is longitude, R is earth's radius.

Additionally, we introduced a heuristic parameter α, with which we can test the effects of underestimation ($\alpha < 1$) and overestimating ($\alpha > 1$) the heuristic distance.

$$h = d^{\alpha} \tag{2}$$

There are different behaviours of the A* algorithm depending on the heuristic parameter α. For the case $\alpha = 0$, the **Dijkstra's algorithm** is retained, for $0 < \alpha \leq 1$, the heuristic function is **admissible** and for $\alpha > 1$ the heuristic function is **inadmissible**.

Regarding the code implementation, in order to avoid unnecessary computations, if $\alpha = 0$, then it will return directly `0` as heuristic cost.

3 Results

3.1 Optimal route obtained

The route obtained with our implementation of A* is shown in fig. 2, and has a path distance of $958.81km$. This route is compared to the one found by Google Maps, which has a path distance of $977km$. The difference is mainly due to the fact that our implementation minimises distance, Google Maps minimises travel time.The path is rather similar, and our implementation provides a sensible solution. Since the heuristic function of Google Maps is unknown, some differences are present (e.g. our implementation passed though Albacete, while Google maps avoided it)

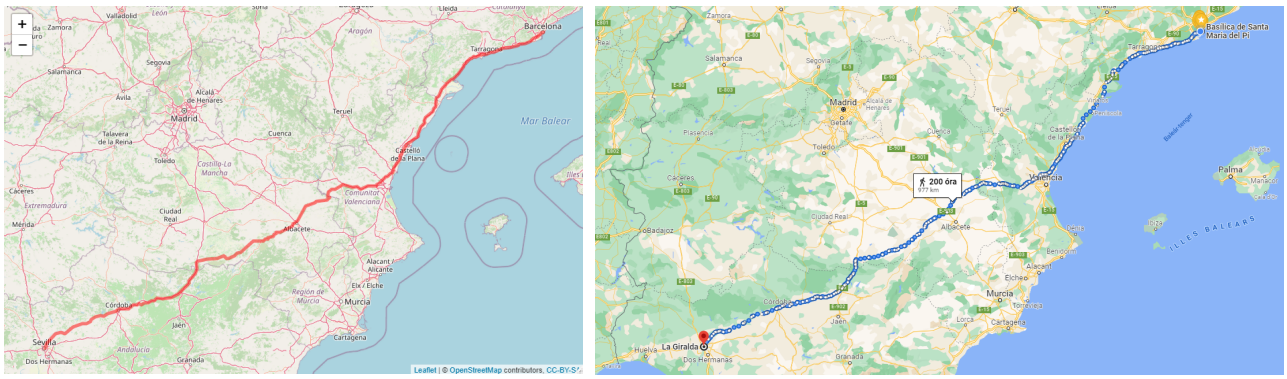

Figure 2: Route found by AStar (left) and Google (right)

3.2 Performance analysis

The following results have been obtained on a 64-bit Ubuntu machine running with 16 GB of RAM. As we have discussed, we get a significant speedup if we have the binary map file already created. More precisely, if no binary file has been found, the program **took a total of 25.75s to create the binary and execute the algorithm**, in contrast to the 3.60s if the file has already been created. Additionally, we have compiled the program with the compiler flags `O3` and `-Ofast`, to see if the compiler can automatically optimise the code further. This option disregards strict standards compliance, and enables optimizations. For the haversine distance, the compiler optimised the code to run under 3.19s. The rest of the runtimes can be seen on Figure 3.

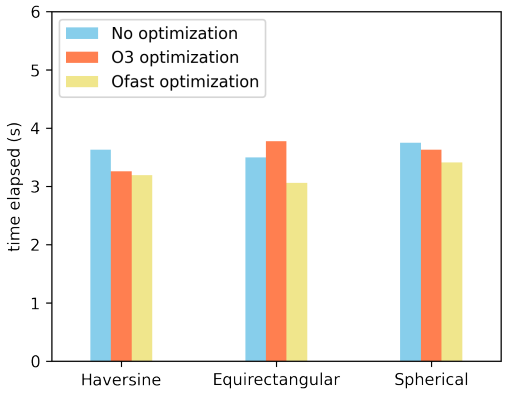

Figure 3: Time elapsed (in seconds) with Different Heuristics

We can see that each distance formula produced relatively the same results in runtime, and that they could each be optimalized to run under 5 seconds. We have compared the routes, distances, and number of nodes visited and explored for the three heuristic functions as well, but have found no difference. Meaning they have found the same route, and the approximate calculations were not significant enough to alter the results in anyway (See Table 1).

	Haversine	Spherical	Equirectangular
No optimization	3.626	3.498	3.745
O3 optimization	3.256	3.772	3.629
Ofast optimization	3.191	3.060	3.410

Table 1: Performance of different geodesic distance formulas

3.3 Exploitation vs. exploration

The A* algorithm has been run using values of the heuristic parameter α (eq. 2) in the range $[0,2]$ in order to assess the exploration and the efficiency for each case.

3.3.1 Optimal distances

As it is shown in fig. 4, for $\alpha < 1$, the algorithm is as exploratory as Dijkstra's, and it returns the same route distance (958.8km, table 2). Dijkstra algorithm always returns the optimal path, so for $0 < \alpha < 1$, **the obtained path is optimal**.

Parameter	0	0.25	0.5	1	1.25	1.5	2
Distance	958816.33	958816.33	958816.33	958816.33	1367358.70	1367358.70	1374972.05
No. nodes in route	6648	6648	6648	1705	12118	12118	12297
No. expanded nodes	12286269	12285953	12279521	2315856	13615	13615	14041
Elapsed time	14.894	16.620	16.789	3.627	0.916	0.902	0.915

Table 2: Performance of different heuristic parameters for Haversine formula

For $\alpha = 1$, the found route is also the optimal one. However, it requires less exploration: the number of expanded nodes decreased by 81.15%. This is because $\alpha = 1$ is the largest value for admissible heuristic functions.

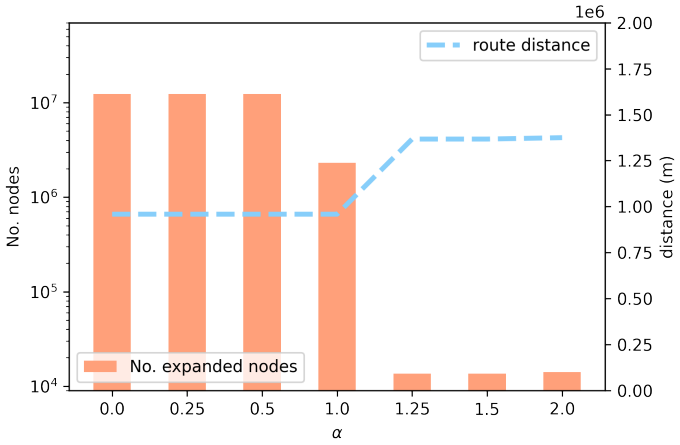

Figure 4: No. of expanded nodes and route distance for different values of α (eq. 2). Note that the node scale is logarithmic but the distance scale is linear.

For $\alpha > 1$, the number of expanded nodes dropped several orders of magnitude, resulting in **highly exploitative** behaviour with a negligible exploration compared to the previous cases. The found routes are not optimal since the heuristic function overestimates the distance, and thus, is inadmissible. In fig.5 is shown a comparison between the optimal route and the route obtained with inadmissible heuristics.

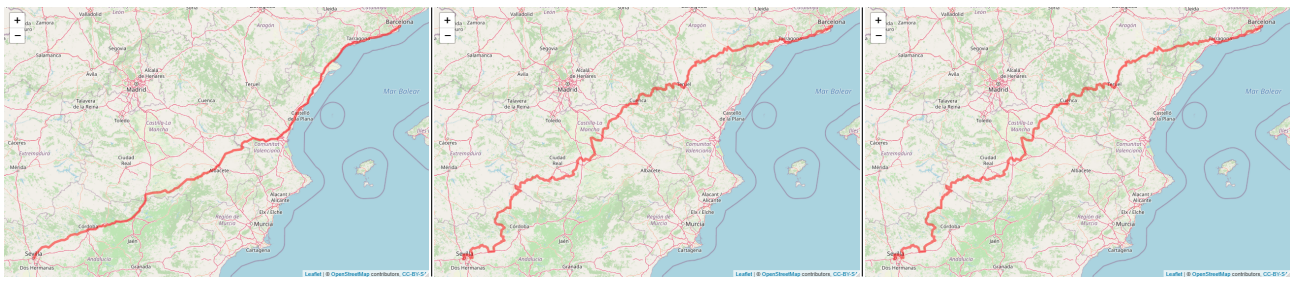

Figure 5: Route found using the Haversine distance for both admissible and inadmissible heuristic parameters α ($\alpha = 1$ and $\alpha = 1.5, 2$ respectively).

3.3.2 Exploration vs. efficiency

The elapsed time of the A* algorithm is also dependent of the parameter α: less exploration means less computation. In this section, the effect of α on the elapsed time will be analysed for different values of α.

The elapsed time is decreased by 75.6% from Dijkstra to A* with an admissible heuristic, while returning the same route distance (fig.6). A* algorithm is more efficient therefore. The efficiency is also increased when using an inadmissible heuristic, but the obtained route distance is a 44.6% larger than the optimal one.

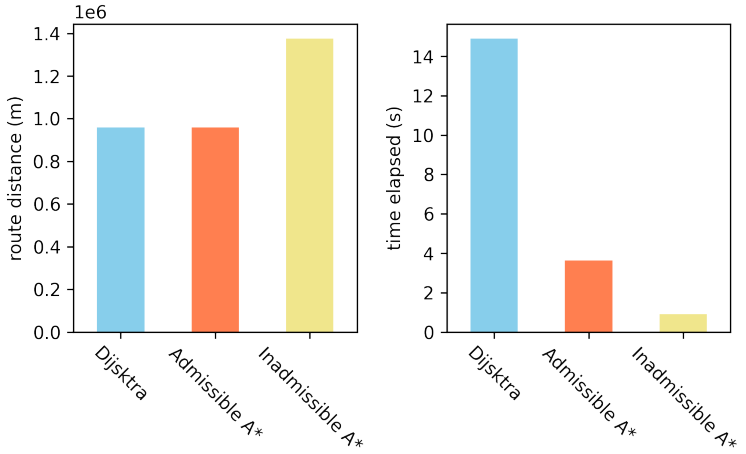

Figure 6: Route distances and elapsed times for the cases $\alpha = 0$ (Dijkstra), $\alpha = 1$ (optimal admissible heuristic), and $\alpha = 2$ (inadmissible heuristic)

4 Conclusions

The routes that our implementation of A* returns are very similar to those of Google Maps. Plus, it is appropriate alternative when the exact distance between nodes in unknown.

Using an underestimation of the geodesic distance ($\alpha < 1$ in eq. 2) will lead to a more exploratory behaviour. A small decrease in α would result in almost identical number of expanded nodes to that of Dijkstra ($\alpha = 0$). However, more exploration entails worse efficiency, and, since the optimal route is also found without underestimation ($\alpha = 1$), there is no benefit in using an underestimation.

On the other hand, using an overestimation ($\alpha > 1$) yields non-optimal routes but with faster computation times. The decrease in computation time (2.68s) does not overweight the effects of increasing the path distance (424km). Therefore, the most sensible choice is using the exact geodesic information ($\alpha = 1$), as it is the most time efficient admissible heuristic.

5 Further improvements

The most time consuming task in our current implementation is locating the position of a node in the priority queue. This is required every time a node is inserted or updated in the queue, and it has a linear time complexity $O(n)$, where n is the number of nodes in the queue. An alternative approach is using a priority tree, since its time complexity is $O(log(n))$. Substituting the priority queue for a priority tree is not a major challenge due to the modularity of the implementation (i.e. the priority queue does not depend on the rest of the algorithm), and could easily be done as future work.

Further research could also be done with the admissibility criterion, and experimenting with bounded relaxation, and dynamic weights [4]. Alternatively, the algorithm could be extended to handle optimising travel time as well, given that the data supports this.

References

[1] A* algoritm pseudocode. `https://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf`. Accessed: 10/12/2021.

[2] Calculate distance, bearing and more between latitude/longitude points. `http://www.movable-type.co.uk/scripts/latlong.html`. Accessed: 08/12/2021.

[3] Openstreetmap. `https://www.openstreetmap.org/#map=6/40.007/-2.488`. Accessed: 06/12/2021.

[4] Ira Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 12–17, 1973.