Durand Hugo
Mulac Nino
Vermot-Desroches Matthias

ROBO4
2023-2024

Report

I. Questions

5) The start and end points have been placed on the exact same locations on the 2 maps. Google reports 22.6km for the fastest path, my implementation of A-Star reports 18.49km. What is yours? Explain the difference vs Google.

The difference is that Google is also probably trying to predict the traffic and the speed at which we will traverse the path, our algorithm does not do this.

II. Demonstration

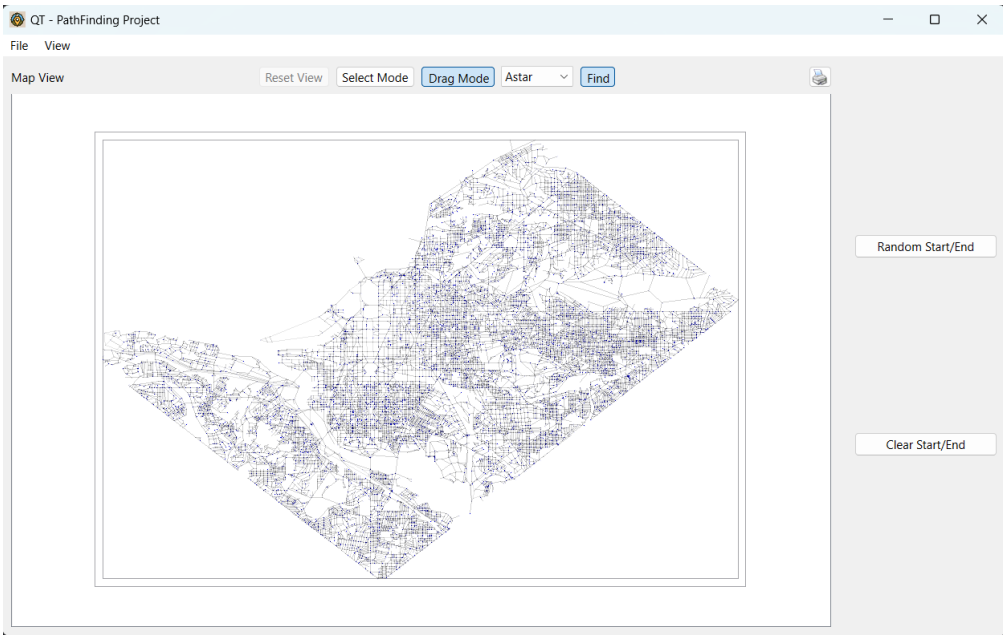
Unfortunately, we can't manage to get our path finding functions called correctly, and can't check if they work correctly.
But here is a little walkthrough of our application :

First, when we launch the project, this window open:

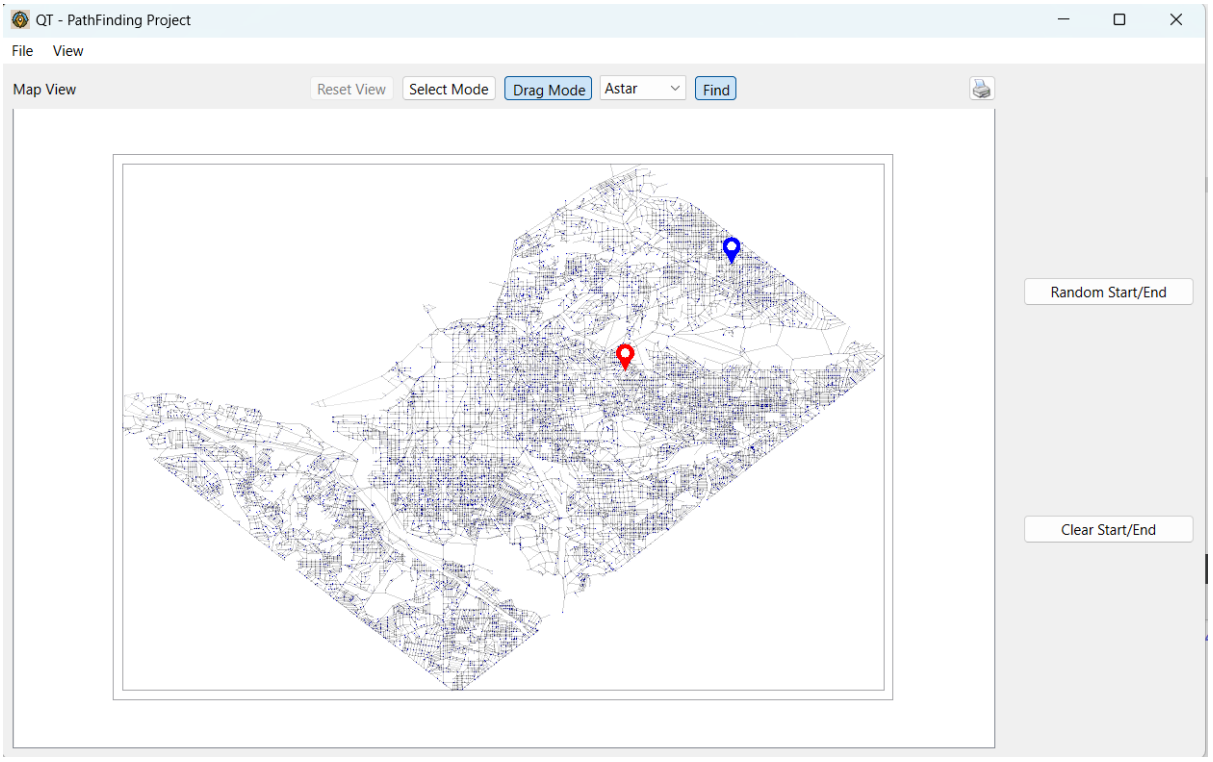
Then, we can load the map by hovering over the "file" at the left hand corner and then left clicking on the "load map" case.

Durand Hugo
Mulac Nino
Vermot-Desroches Matthias

When the map is loaded, we obtain this result:



Now, we can place Start and End Vertex either at random using the “Random Start/End” button or manually by choosing the select mode on the top of the window.



After that, we could in theory use the method we programmed to find the shortest path. However, no matter what we do, we always return 0.

Durand Hugo
Mulac Nino
Vermot-Desroches Matthias

III. Details on the graph search algorithm

1. Algorithms

We use a double-ended queue to store the opened vertices.
We needed a FIFO container, with a way to find an item in it to ensure we don't loop and go backwards in the search.

Since the queue doesn't support it, we chose to use the deque.

To store the closed vertices, we use a set for its capability of verifying the presence of an element in itself.

2. Data structures

We wanted to make the data access as fast as possible, so we chose to work on vertex IDs, and to assume the vertex IDs are as contiguous as possible since we store the vertices in a vector with the vertex ID matching the position in the vector.
This way of storing data is really fast, since accessing data from a vector at an arbitrary position is really simple, but all unused IDs make "holes" in our vector, thus wasting memory.

```
void Graph::addVertexToGraph(const Vertex &vertex)
{
   vertices_[vertex.getID()] = vertex;
}
```

To store distances, we chose to combine the IDs(uint32_t) of the vectors of the edge in an unordered map by shifting the lowest ID by 32 and adding it to the greatest to get a unique uint64_t edge ID. This way, we have a simple and symmetric pseudo-hash function, even simpler than a sorted pair<uint32_t, uint32_t> for example.

```
double Graph::distanceBetweenVertices(const uint32_t vertex1_id, const
uint32_t vertex2_id)
{
   return edges_[(((uint64_t)std::min(vertex1_id, vertex2_id)) << 32) +
std::max(vertex1_id, vertex2_id)];
}
```

The Vertex class is also as simple as possible, with only the fields for storing the location, the ID, and a vector of neighbors for simpler path finding.

Loading our map in these conditions is really fast, and only takes around 10 seconds.