Project: Map Path Finder

Embedded C++

by

Philipp AHRENDT,

and

Sivadinesh PONRAJAN

Autonomous Systems M.Sc.

EIT Digital

Polytech Nice-Sophia

Université Cote d'Azur

Supervisor: Bernard PLESSIER

Submitted: March 30, 2023

Contents

1 Introduction

Graph theory is a fundamental theory in modern real-world applications and the field of computer science. Graphs are vital for applications such as transportation systems, social and computer networks, and epidemiology (modeling the spread of diseases). Combined with different search algorithms that allow traveling through the graph and its vertices and edges it is used to find the shortest or simply valid path from a start to an end node. During the course of this project, we will implement a graph library with C++, as well as three search algorithms, namely: Bread First Search, Dijkstra, and A-star. First, the graph and the algorithms will be implemented in a console application, that allows specifying of two nodes, one of the three algorithms, and a file of map data. This file is then read and prints out the path and other information.

Finally, we will design a GUI that allows users to load a graph file, visualize the graph, and perform searches using various algorithms, with the final results displayed in the user interface.

This project aims to give an understanding of the basics of search algorithms, showing the basic implementation of a graph library and its various functionality, while at the same time being a useful tool to visualize an example real-world application, eg. Finding the shortest path from a map with real-world coordinates. Both the console application and the GUI help understand the concepts of graphs and search algorithms, while the GUI especially helps to visualize the benefit of using A-star compared to the other two algorithms.

2 Graph Library

In the first part, we started with the Lemon Graph library, but since this library did not have all the functionality we needed and wanted and to have more control and understanding of our graph library we decided to write our own.

It consists of the three main components of any graph library: a graph class, a vertex (or node) class and an edge (or arc) class.

Although

2.1 Vertex

Attribute	**Type**	**Description**
uid_	int	Unique identifier for the vertex.
long_	double	Longitude of the vertex's location.
lat_	double	Latitude of the vertex's location.
x_	double	X-coordinate of the vertex's location.
y_	double	Y-coordinate of the vertex's location.
weight_	double	Weight of the vertex, used in the search algorithms (Dijkstra and A-star).
estimate_	double	Estimated cost from the current vertex to the target vertex, used in A* search.
adjacencyList_	std::set<int>	A set of integers representing the unique identifiers of the neighboring vertices.

Table 1: Attributes of the `Vertex` struct.

2.2 Edge

Name	**Type**	**Description**
fromID_	integer	The ID of the vertex at the start of the edge.
toID_	integer	The ID of the vertex at the end of the edge.
length_	double	The weight/ length of the edge.

Table 2: Attributes of the `Edge` Struct

2.3 Graph

The Graph class represents a graph data structure and provides methods for reading in graph data from a file in CSV format, adding vertices and edges to the graph, converting latitude and longitude coordinates to Cartesian coordinates, computing edge weights, and performing a breadth-first search (BFS), Dijkstra or A-star algorithm to find the (shortest) path between two vertices.

The constructor takes a file name as an argument and calls the read_file method to populate the graph with vertices and edges from the file. The read_file method reads in each line of the file, which is assumed to be in CSV format with the first element indicating whether the line represents a vertex or an edge. If the line represents a vertex, the method adds the vertex to the graph and updates the sum of the latitude and longitude values for the mercator projection. If the line represents an edge, the method adds the edge to the graph.

The convertToCartesian method takes the average latitude and longitude of all vertices and converts each vertex's latitude and longitude to Cartesian coordinates using the mercator projection. The method then computes the weight of each edge as the Euclidean distance between its two vertices in Cartesian coordinates.

The addVertex method takes a row of CSV data for a vertex and adds the vertex to the graph with its ID, latitude, and longitude as its attributes.

The addEdge method takes a row of CSV data for an edge and adds the edge to the graph with its source and destination vertices and length as its attributes.

The get_edge_weight method takes two vertex IDs and returns the length of the edge between them.

The implemented search algorithm functions will be explained in the following Section.

Attribute	Type	Description
vertices	std::map<int, Vertex>	A map of all vertices in the graph, where each vertex is represented by an integer ID and a Vertex struct containing information about that vertex.
edges	std::vector<Edge>	A vector of all edges in the graph, where each edge is represented by an Edge struct containing information about that edge.
edgeLookUp	std::multimap<int, Edge>	A multimap used for quickly looking up all edges incident to a given vertex. The multimap maps each vertex ID to a vector of Edge structs representing all edges incident to that vertex.

Table 3: Attributes and methods of the Graph class

3 Search Algorithms

3.1 Breadth First Search

BFS is an uninformed search algorithm that explores the vertices of a graph in layers, starting from a given source vertex and moving towards its adjacent vertices before moving on to the next layer. BFS guarantees that the shortest path between the source vertex and any other vertex in the graph will be found, provided that all edges have the same weight. The time complexity of BFS is O(|V| + |E|), where |V| is the number of vertices and |E| is the number of edges in the graph.

The `Graph::bfs(int start, int goal)` function implements the breadth-first search algorithm on a graph represented by an adjacency list. The function returns a vector of pairs containing the IDs of the vertices in the shortest path from the start vertex to the goal vertex, along with the length of that path.

Arguments

- **start:** The ID of the start vertex.
- **goal:** The ID of the goal vertex.

Return Value

The function returns a vector of pairs, where each pair consists of an integer (the ID of a vertex in the shortest path) and a double (the length of the path from the start vertex to that vertex). If no path exists between the start and goal vertices, an empty vector is returned.

Algorithm

The function uses a deque to implement the active queue and a set to keep track of the closed set. The parent map is used to keep track of the parent of each visited vertex. The function begins by initializing the active queue with the start vertex ID and an empty path vector. It then enters a loop that continues until the active queue is empty.

At each iteration of the loop, the function dequeues the front vertex vcurrent from the active queue and checks if it is the goal vertex. If vcurrent is the goal vertex, the function returns the shortest path from the start vertex to the goal vertex by calling the backtrace function with the parent map, start vertex ID, and goal vertex ID as arguments.

If vcurrent is not the goal vertex, the function adds vcurrent to the closed set, increments the numberOfVertices counter, and iterates over the adjacency list of vcurrent. For each vertex vnext in the adjacency list, the function checks if vnext is already in the closed set. If vnext is in the closed set, it is skipped. If vnext is not in the active queue, it is added to the back of the active queue, and its parent is set to vcurrent in the parent map.

Once the loop completes, the function returns the path vector. If the goal vertex was not reached, the path vector will be empty.

The `Graph::backtrace` is used to return this path vector. This function takes in a map of parent nodes and their children, as well as a start node and a goal node. It then performs a backtrace from the goal node to the start node, by repeatedly adding the current node and the weight of the edge from the previous node to the current node to a vector. It then gets the next node by getting its parent from the map, which was set in the search algorithm function. Once the backtrace has reached the start node, it adds the start node with weight 0 to the vector and then reverses the vector to get the path in the correct order. The function then returns the resulting path vector.

Example Usage

To find the shortest path between vertices 1 and 5 in a graph g, the following code can be used:

```
1 std::vector<std::pair<int, double>> path = g.bfs(1, 5);
```

This will return a vector of pairs representing the shortest path from vertex 1 to vertex 5, along with the length of that path.

For the information about the path concerning vertex 73964 to vertex 272851 see Table 5 and for a full trace of all the paths see Appendix 6.

Length of the path	13080.25
Number of vertices on the path	69

Table 4: Path length and visited vertices count from 19791 to 50179

```
❯ ./graph_traversal --start 42159 --end 287 --algorithm bfs --file dataset/graph_dc_area.2022-03-11.txt
Total visited vertex: 201
Total vertex on path from start to end = 10
Vertex[   1] =    42159, length =         0.00
Vertex[   2] =    42154, length =       118.55
Vertex[   3] =    59974, length =       236.97
Vertex[   4] =    59970, length =       286.47
Vertex[   5] =   117100, length =       443.75
Vertex[   6] =    11069, length =       499.07
Vertex[   7] =    11065, length =       548.53
Vertex[   8] =   122982, length =       649.72
Vertex[   9] =    70438, length =       755.28
Vertex[  10] =      287, length =       785.66
Info: path calculated in 202us
```

Figure 1: BFS trace for (42159, 287)

3.2 Dijkstra Algorithm

Dijkstra's algorithm is a weighted graph search algorithm that finds the shortest path between a given source vertex and all other vertices in the graph. It works by assigning a tentative distance to every vertex and iteratively choosing the vertex with the smallest tentative distance as the current vertex until the destination vertex is reached. The time complexity of Dijkstra's algorithm is O(|E| + |V| log |V|), where |V| is the number of vertices and |E| is the number of edges in the graph.

The parameters and return values of the `std::vector<std::pair<int, double» Graph::dijkstra(in start, int goal)` are the same as the BFS described above.

Algorithm:

1. Create an empty deque called `active_queue` to keep track of the vertices to visit, a set called `closed_set` to keep track of the visited vertices, a map called `parent` to keep track of the parent of each visited vertex, and an integer called `numberOfVertices` to keep track of the number of vertices visited.

2. Create a lambda function called `compare` to compare the weights of two vertices based on their IDs.

3. Set the weight of all vertices to a maximum value using the member function `set_all_vertex_weight_to_max_value()`.

4. Set the weight of the start vertex to 0 using the member function `set_weight(0)`.

5. Add the start vertex to the `active_queue`.

6. While the `active_queue` is not empty, do the following:
   (a) Get the first vertex in the `active_queue`, which has the minimum weight according to the lambda function `compare`.
   (b) If the current vertex is the goal vertex, return the shortest path from start to goal using the function `backtrace(parent, start, goal)`.
   (c) Remove the current vertex from the `active_queue` and add it to the `closed_set`.
   (d) Increment the `numberOfVertices` counter.
   (e) For each vertex adjacent to the current vertex, do the following:
      i. If the adjacent vertex is already in the `closed_set`, skip to the next adjacent vertex.
      ii. Calculate the weight of the adjacent vertex as the weight of the current vertex plus the weight of the edge between them, using the member function `get_edge_weight(vcurrent, vnext)`.
      iii. If the adjacent vertex is not in the `active_queue`, add it to the `active_queue`, set its weight to the calculated weight, set its parent to the current vertex in the `parent` map, and increment the `newVerticesCount`
      iv. If the adjacent vertex is already in the `active_queue`, but the calculated weight from the current node to this adjacent node is smaller than the stored weight then set the new weight of this node and update its parent to the current node.
   (f) Partially sort the active queue from the beginning with the newly added nodes so that the node with the smallest weight is at the top of the queue.

Note: In a sorted queue, the vertices are stored in a sorted or partially sorted vector, where each vertex is in its correct position in the queue. While the insertion and deletion of a vertex at the top or bottom are constant O(1), insertion or deletion, in general, is linear O(n). So, when a new vertex is inserted, it is inserted at the bottom of the queue, but since we need a (partially) sorted queue to get the vertex with the least weight, which requires finding its correct position in the queue, this takes in effect O(n) time for each insertion operation.

On the other hand, in a priority heap, the vertices are stored in a binary tree where the parent node has a lower priority than its children. When a new vertex is inserted, it is placed in the next available position in the heap and then the heap is restructured until it is in the correct position according to its priority. This takes O(log n) time for each insertion operation.

We have used both versions in our implementation (`Graph::dijkstra` and `Graph::dijkstra_priority`).

You can run the console application with the following command after doing `Make` in the folder with the files:

```
> ./graph_traversal --start 73964 --end 272851 --algorithm astar
    --file dataset/graph_dc_area.2022-03-11.txt
```

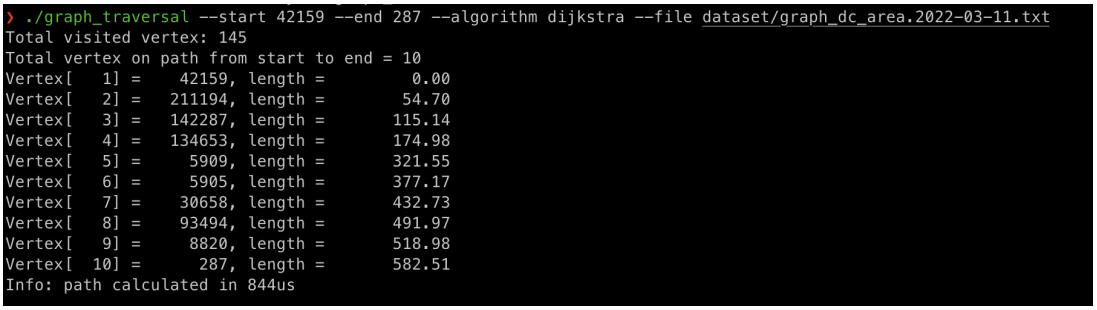

Figure 2: Dijkstra trace for (42159, 287)

3.3 A-star (A*) Shortest Path

A* search algorithm is another weighted graph search algorithm that is often used in pathfinding and graph traversal. It works by searching the graph while evaluating the estimated distance between the current node and the goal using a heuristic function. The algorithm chooses the next node with the lowest sum of the estimated distance and the distance from the start node. A* search algorithm can be seen as an extension of Dijkstra's algorithm that uses heuristics to guide the search. The time complexity of A* search algorithm is similar to Dijkstra's algorithm, i.e., O(|E| + |V| log |V|).

The `Graph::astar` function takes the same parameters and returns the same vector of pairs as the other two search algorithms. Until step 6.e the algorithm is identical to the dijkstra algorithm, but then uses the heuristic function:

As before as long as the goal vertex is not found the function adds vcurrent to the closed_set, increments the numberOfVertices counter, and initializes a counter newVerticesCount for the number of vertices that are pushed into the active_queue for the partial sort.

The function then loops through the adjacency list of vcurrent to explore the neighbors of vcurrent. For each neighbor vnext, the function calculates the cost g of the path from the start vertex to vnext through vcurrent, and the estimated cost f of the path from the start vertex to the goal vertex through vnext.

If vnext is not already in the active_queue, the function sets the weight and estimate of vnext to g and f, respectively, adds vnext to the active_queue, sets the parent of vnext to vcurrent, and increments the newVerticesCount counter.

Otherwise, if the estimated cost f of the path from the start vertex to the goal vertex through vnext is less than the current estimate of vnext, the function updates the weight and estimate of vnext, sets the parent of vnext to vcurrent, and since it is already part of the active_queue it does not add it again.

After exploring all the neighbors of vcurrent, the function performs a partial sort on the active_queue using a lambda function compare that compares the estimated costs of the vertices. The newVerticesCount parameter is used to sort only the vertices that were added in the current iteration.

```
❯ ./graph_traversal --start 42159 --end 287 --algorithm astar --file dataset/graph_dc_area.2022-03-11.txt
Total visited vertex: 23
Total vertex on path from start to end = 10
Vertex[   1] =    42159, length =         0.00
Vertex[   2] =   211194, length =        54.70
Vertex[   3] =   142287, length =       115.14
Vertex[   4] =   134653, length =       174.98
Vertex[   5] =     5909, length =       321.55
Vertex[   6] =     5905, length =       377.17
Vertex[   7] =    30658, length =       432.73
Vertex[   8] =    93494, length =       491.97
Vertex[   9] =     8820, length =       518.98
Vertex[  10] =      287, length =       582.51
Info: path calculated in 114us
```

Figure 3: A-star trace for (42159, 287)

3.4 Comparison

We can see a side-by-side comparison of the relevant information of the three implemented search algorithms in Table 5. The comparison is done using the same start and end vertex for all algorithms, start: 73964 and end: 272851.

BFS is the fastest algorithm out of the three, because it does not do any sorting or weight/ cost calculation. It assumes same cost for each vertex. It finds the shortest path **if** every edge has the same length. This can be seen by the lower number of vertices on the final path. At the same time it can also be noted that the number of vertices on the final path is the same for dijkstra and astar. This is because they both find the actual shortest path when taking the real edge length into account. At the same time it is obvious that A-star visits a significantly lower number of vertices to find the shortest path compared to dijkstra. As a consequence it is also faster than dijkstra. This is all thanks to the heuristic estimation function. The full trace for all of the algorithms can be found in the Appendix 6.

In summary, BFS is an efficient algorithm for finding the shortest path in unweighted graphs, while Dijkstra's algorithm and A* search algorithm are more suitable for weighted graphs, where edge weights represent the cost of moving from one vertex to another. A* search algorithm is often faster than Dijkstra's algorithm since it uses heuristics to guide the search.

	BFS	Dijkstra	Dijkstra (Prio heap)	A-star
Time [us]	**11,480**	348,163	27,480	24,931
Number of vertices on final path	**66**	76	76	76
Total number of vertices visited	14468	14938	14938	**2253**
Total path length	11595.40	**7151.78**	**7151.78**	**7151.78**

Table 5: Comparison of the 3 search algorithms

4 Graphical User Interface

As we have successfully validated our path algorithms, we proceeded with the second part of the project i.e. displaying the map and the resulting shortest path.

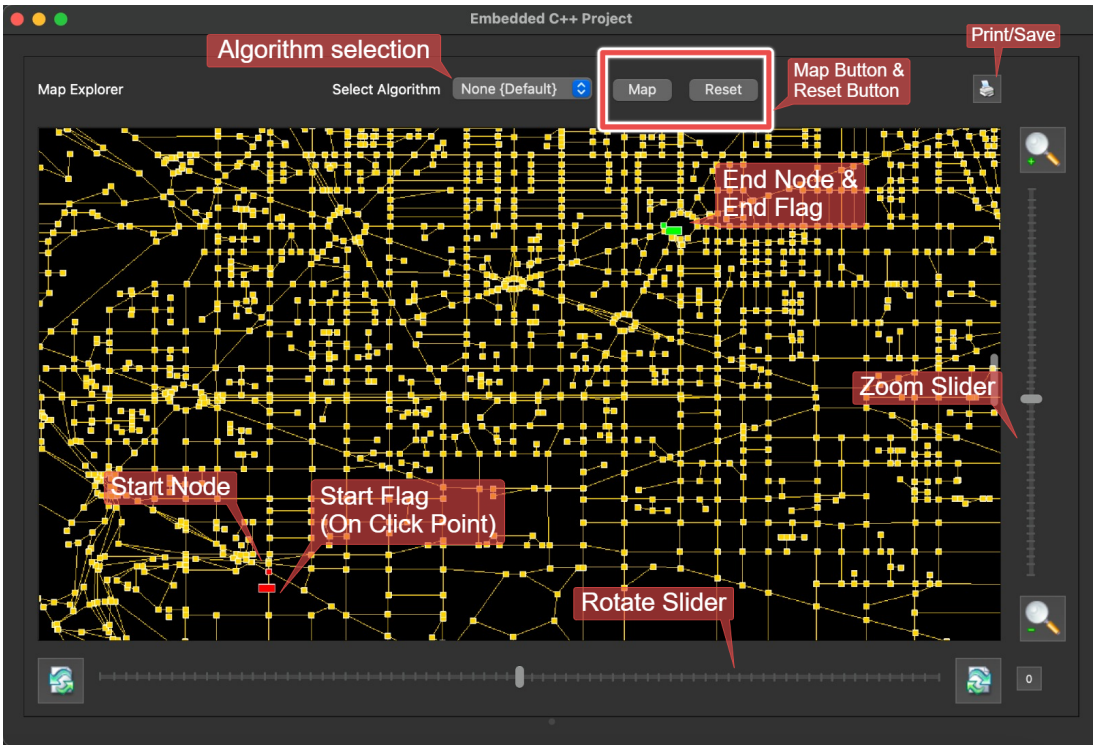

Figure 4: GUI of the application

4.1 Structure of the code

Since we are recreating the same algorithms with visualization, we imported our graph library so that we could use the BFS, Dijkstra, and the A-star algorithms under the class Graph.

This graph class can this be imported and reused in the QT's code.

4.2 Description of GUI items

Basically, our application has a map area that is used to display the map, a combobox/-dropdown to select the algorithm, a map button to plot the path on the map, and a reset button to reset the plotted map back to the empty map.

All the UI interactive functions like button click, zoom in / zoom out, and rotate are handled in a class called "View". While the rest are handled by the MainWindow class.

Map Area

For the map area, we used "QGraphicsView" which is part of the Graphics View Framework of QT. "QGraphicsScene" is instantiated and was assigned to the QGraphicsView. So when we plot the map, we can just use the QPainter and QPen to draw the lines or structures.

Initially, the graph class is instantiated and when the application starts, it opens a prompt to select the input file. When the input file is selected, it automatically reads the vertices and edges and populates the data into the maps in the same structure that was explained earlier. Since the input file has only the data of latitude and longitude, the cartesian conversion function is called and the Mercator projection is made. All these things will happen in the constructor of the graph class.

In the MainWindow, the populateScene function is responsible for the plotting of the map. When reading the input data, the min and max of the latitude and the longitude were estimated which is now used to compute the width and the height of the image to be created.

Initially, the vertices were plotted, and then the edges were drawn. We created a separate class to create the QGraphicsItem called "house" to plot and represent the vertex in the map. While for creating the edges, we used the inbuilt QT function of "drawLine" from the QPainter is used.

QGraphicsItem - vertex

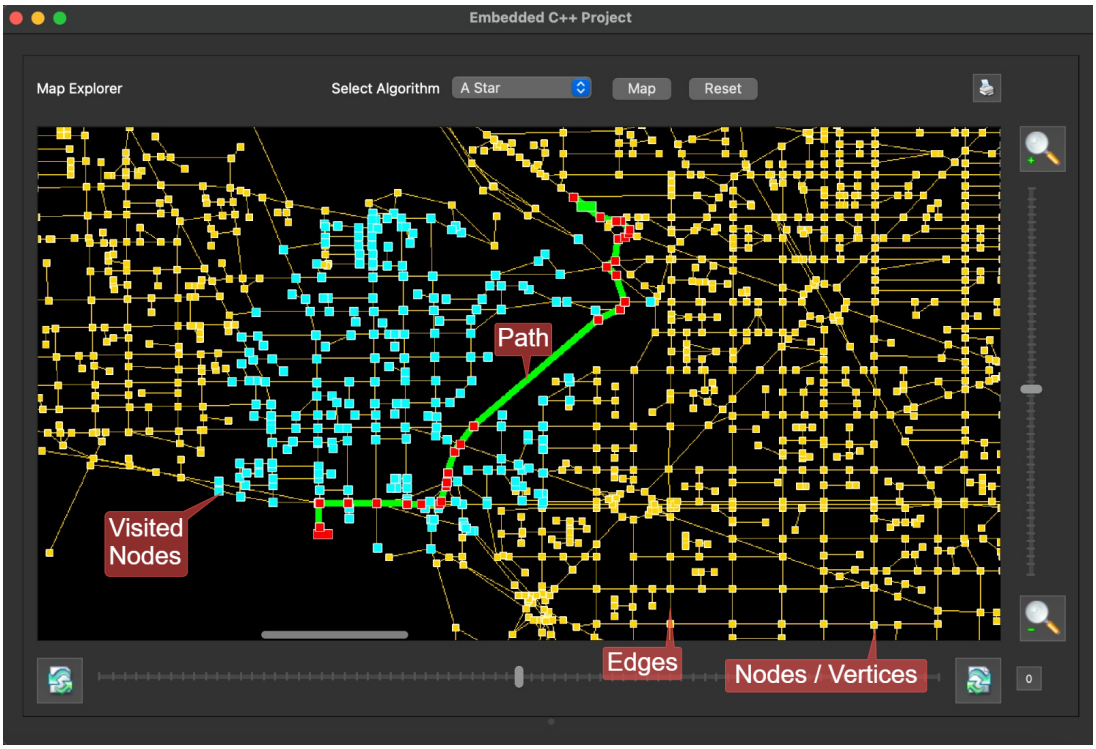

Figure 5: The vertex, edges, path, visited nodes.

As mentioned earlier, we created a new class called "house" to plot and represent the vertex in the map. The vertices map is iterated so that the x and y coordinates of every vertex are plotted by passing as an argument for the instantiated object. The color in which the item should be created and the individual id is also sent as arguments. With the x and y coordinates, a bounding rectangle is plotted in that particular position and it is represented as a square-shaped block.

Then in the populateScene function, edges are created by iterating the "plotEdgeLookUp" multimap. where the x and y coordinates of the nodes were obtained.

Map Button

By default, the dropdown is selected to none i.e. no algorithms are selected and the index of the dropdown is passed as an argument for the populateScene function. So when we select an algorithm in the dropdown and the map button is clicked, the populateScene function is called again with the new argument, and the whole scene is painted again.

Irrespective of the choices of the algorithms selected, when the map button is clicked, the default map will be populated at first and based on the selection of the algorithms,

the path will be highlighted.

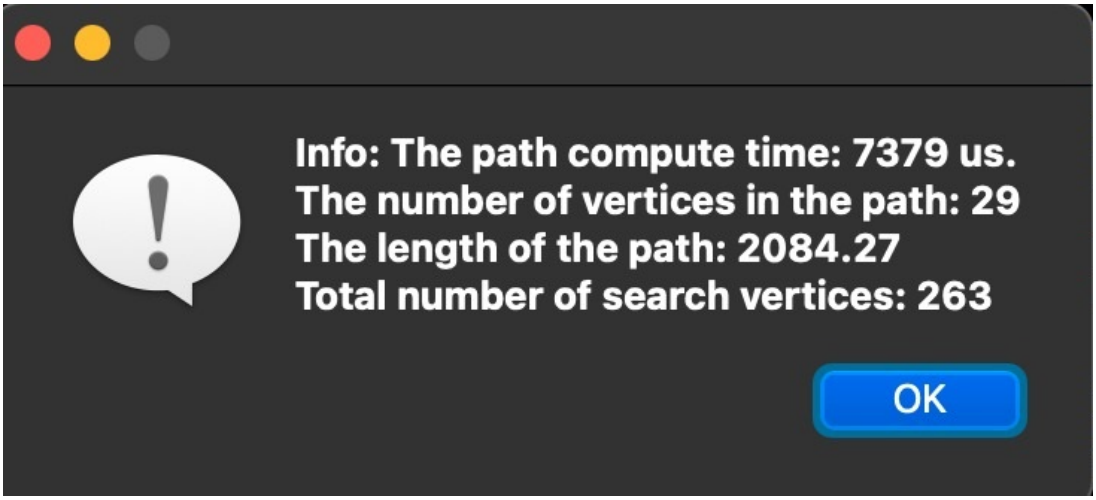

Figure 6: The selected start node and the end node at the bottom of the map

On every map function call, the path computes time, the number of vertices in the path, the length of the path, and the total number of vertices visited are invoked from the graph class and displayed.

4.3 Graph Interface

We have also created an override function to get the x and y coordinates where the user is clicking. So basically when the user clicks somewhere on the map, the click is handled and a new flag is created in the map. In the override function, we also calculate the nearest node and create a start/end node and create a new item on that same position over the existing position. When the start or end node is selected, the IDs of selected nodes were also displayed at the bottom of the map.

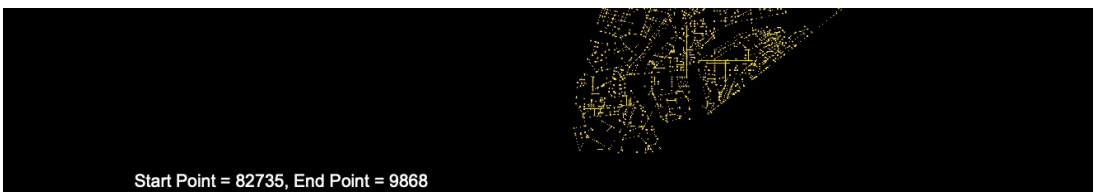

Figure 7: The selected start node and the end node at the bottom of the map

Based on the selected algorithm, the functions are called from the graph class so that the path is returned as a vector of Vertex. We can also access the visited nodes through the member function. Based on the visited nodes and the path, the new items were recreated

with a different color to distinguish between them, and the edges of the path from the start node to the end node were also highlighted in a different color.

Reset Button

When the reset button is triggered, the start flag, end flag, and the rest of the items are removed, the dropdown is reset and the populateScene function is called with default arguments so that only the map is populated and all the other highlights are discarded.

Zoom/Rotate

From the specified example "QT 40000 chips", we also adapted some of the features like zoom in/zoom out and rotate functions similarly. We have individual sliders for both of the functionalities. Whenever there was a change in the slider's value, it will be reflected in the map interface directly.

4.4 Additional Feature

We also have added the feature of saving and printing the map image as a pdf.

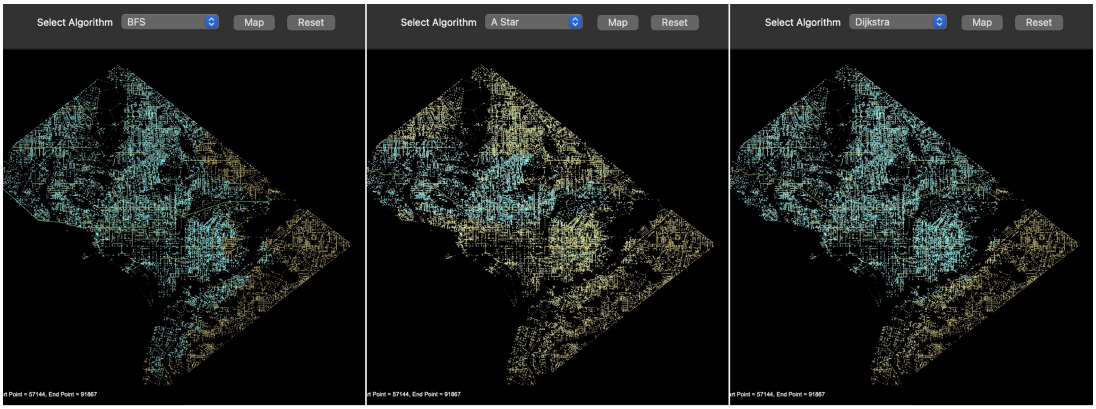

Figure 8: The comparison of BFS, A-Star and Dijkstra plot

5 Conclusion

During the course of this project, we were able to implement our own basic graph library to construct a graph from a given map file. We then succeeded in implementing three different search algorithms and were able to compare them. We could conclude that while the BFS algorithm was the fastest and had the least number of vertices on the path, the Dijkstra and A-star algorithm found the shortest path, with the A-star algorithm being

much faster than the Dijkstra and visiting far fewer nodes.

In the next step, we implemented a graphical User Interface using QT to display the whole map through its nodes and edges. We also managed to allow a user the possibility to select the search algorithm of their choice, finding the graph similar to the console application and then finally displaying the found path on top of the map, highlighting all the visited nodes and the nodes and edges of the final path.

6 Appendix

The full code can be found at https://github.com/phiahr/Map_Path_Finder

6.1 BFS Traces

```
>./graph_traversal --start 19791 --end 50179 --algorithm bfs --file
   dataset/graph_dc_area.2022-03-11.txt

Total visited vertex: 19871
Total vertex on path from start to end = 69
Vertex[   1] =    19791, length =         0.00
Vertex[   2] =    19785, length =        89.16
Vertex[   3] =    61395, length =       237.35
Vertex[   4] =     6684, length =       415.05
Vertex[   5] =     6679, length =       579.33
Vertex[   6] =    72093, length =       716.14
Vertex[   7] =    57943, length =       762.64
Vertex[   8] =    36611, length =       809.37
Vertex[   9] =    40589, length =       946.43
Vertex[  10] =    40582, length =      1064.10
Vertex[  11] =   130043, length =      1216.25
Vertex[  12] =    77738, length =      1346.57
Vertex[  13] =   181119, length =      1473.91
Vertex[  14] =   181111, length =      1622.78
Vertex[  15] =    61305, length =      1724.94
Vertex[  16] =    61299, length =      1789.75
Vertex[  17] =    39379, length =      1925.27
Vertex[  18] =    39372, length =      2096.69
Vertex[  19] =   116730, length =      2297.12
Vertex[  20] =    41124, length =      2427.91
Vertex[  21] =   179258, length =      2566.02
Vertex[  22] =    71551, length =      2647.63
Vertex[  23] =    11308, length =      2731.30
Vertex[  24] =      771, length =      2909.40
```

```
Vertex[  25] =    38465, length =      3082.89
Vertex[  26] =    45963, length =      3286.17
Vertex[  27] =    45966, length =      3364.49
Vertex[  28] =    45967, length =      3385.35
Vertex[  29] =   131161, length =      3691.82
Vertex[  30] =   131181, length =      3818.99
Vertex[  31] =    87279, length =      4218.05
Vertex[  32] =     1566, length =      4336.77
Vertex[  33] =     1567, length =      4434.04
Vertex[  34] =   119626, length =      4650.32
Vertex[  35] =   103487, length =      4853.96
Vertex[  36] =   246807, length =      5039.22
Vertex[  37] =   246803, length =      5144.31
Vertex[  38] =   246837, length =      5283.08
Vertex[  39] =    29893, length =      5879.79
Vertex[  40] =    53176, length =      6132.61
Vertex[  41] =    26180, length =      6201.40
Vertex[  42] =     5357, length =      6223.74
Vertex[  43] =    50931, length =      6693.23
Vertex[  44] =    66420, length =      7110.93
Vertex[  45] =   268141, length =      7849.32
Vertex[  46] =   268144, length =      7893.54
Vertex[  47] =   265180, length =      8524.91
Vertex[  48] =   105361, length =      8864.58
Vertex[  49] =    44790, length =      9234.87
Vertex[  50] =   176878, length =      9417.15
Vertex[  51] =   264760, length =      9732.12
Vertex[  52] =   224866, length =     10292.43
Vertex[  53] =    23560, length =     10333.49
Vertex[  54] =    23567, length =     10476.95
Vertex[  55] =    18535, length =     10675.62
Vertex[  56] =    13400, length =     10717.97
Vertex[  57] =     5508, length =     10819.60
Vertex[  58] =     5502, length =     10911.26
Vertex[  59] =   167442, length =     11044.71
Vertex[  60] =    72572, length =     11200.65
Vertex[  61] =    72586, length =     11326.01
Vertex[  62] =   179917, length =     11704.96
Vertex[  63] =     3044, length =     12068.29
Vertex[  64] =    48616, length =     12190.38
Vertex[  65] =   111550, length =     12575.75
Vertex[  66] =   136094, length =     12721.24
Vertex[  67] =   111185, length =     12792.68
Vertex[  68] =   115261, length =     12846.56
Vertex[  69] =    50179, length =     13080.25
Info: path calculated in 16,481us
```

Trace from Vertex 19791 to 50179

```
> ./graph_traversal --start 73964 --end 272851 --algorithm bfs --file
   dataset/graph_dc_area.2022-03-11.txt

Total visited vertex: 14468
Total vertex on path from start to end = 66
Vertex[   1] =    73964, length =         0.00
Vertex[   2] =    39648, length =       312.36
Vertex[   3] =    83978, length =       343.28
Vertex[   4] =   277329, length =       564.27
Vertex[   5] =   277861, length =       709.69
Vertex[   6] =    85872, length =       897.99
Vertex[   7] =    89941, length =      1013.32
Vertex[   8] =    89946, length =      1269.44
Vertex[   9] =   260698, length =      1395.41
Vertex[  10] =   260104, length =      1481.44
Vertex[  11] =   142337, length =      1616.11
Vertex[  12] =    39852, length =      1813.19
Vertex[  13] =    64491, length =      1883.68
Vertex[  14] =    55853, length =      1964.58
Vertex[  15] =    95706, length =      2010.75
Vertex[  16] =    99865, length =      2274.47
Vertex[  17] =    99872, length =      2537.27
Vertex[  18] =     8973, length =      2632.37
Vertex[  19] =    61833, length =      2732.64
Vertex[  20] =   137770, length =      2802.59
Vertex[  21] =    83810, length =      2911.24
Vertex[  22] =    12367, length =      3085.61
Vertex[  23] =    12390, length =      3162.47
Vertex[  24] =   119033, length =      3203.47
Vertex[  25] =      431, length =      3256.72
Vertex[  26] =     3981, length =      3391.28
Vertex[  27] =    64185, length =      3773.92
Vertex[  28] =    63921, length =      4355.18
Vertex[  29] =   104839, length =      4872.78
Vertex[  30] =    54663, length =      5038.52
Vertex[  31] =    53819, length =      5156.58
Vertex[  32] =    53810, length =      5309.19
Vertex[  33] =    22847, length =      5458.55
Vertex[  34] =    22843, length =      5540.05
Vertex[  35] =    15043, length =      5648.41
Vertex[  36] =    15002, length =      6228.87
Vertex[  37] =   104325, length =      6513.75
Vertex[  38] =    54059, length =      6594.86
Vertex[  39] =    39677, length =      6739.37
Vertex[  40] =    39678, length =      6787.43
Vertex[  41] =   102584, length =      7632.83
Vertex[  42] =    56149, length =      7761.19
```

```
Vertex[  43] =    56156, length =      8202.73
Vertex[  44] =   164168, length =      8247.97
Vertex[  45] =   164170, length =      8299.08
Vertex[  46] =   236064, length =      8472.83
Vertex[  47] =   213936, length =      8598.51
Vertex[  48] =   171983, length =      8612.52
Vertex[  49] =   164856, length =      8803.30
Vertex[  50] =    86775, length =      9030.09
Vertex[  51] =    73218, length =      9051.72
Vertex[  52] =    86778, length =      9150.01
Vertex[  53] =   121137, length =      9469.62
Vertex[  54] =    87122, length =      9558.07
Vertex[  55] =    87129, length =      9790.40
Vertex[  56] =   153499, length =      9959.88
Vertex[  57] =   153503, length =     10110.49
Vertex[  58] =    23996, length =     10184.69
Vertex[  59] =    23980, length =     10431.32
Vertex[  60] =    27136, length =     10459.51
Vertex[  61] =   173005, length =     10580.33
Vertex[  62] =   127517, length =     10961.15
Vertex[  63] =    60949, length =     11027.77
Vertex[  64] =   272881, length =     11041.82
Vertex[  65] =   272803, length =     11361.07
Vertex[  66] =   272851, length =     11595.40
Info: path calculated in 11,403us
```

Trace from Vertex 73964 to 272851

6.2 Dijkstra Trace

```
> ./graph_traversal --start 73964 --end 272851 --algorithm dijkstra --
    file dataset/graph_dc_area.2022-03-11.txt

Total visited vertex: 14938
Total vertex on path from start to end = 76
Vertex[   1] =    73964, length =         0.00
Vertex[   2] =    73963, length =        34.91
Vertex[   3] =   121925, length =        67.20
Vertex[   4] =    32152, length =       254.00
Vertex[   5] =   244143, length =       377.27
Vertex[   6] =   243413, length =       467.35
Vertex[   7] =   243414, length =       606.69
Vertex[   8] =   243424, length =       723.74
Vertex[   9] =    79573, length =       786.41
Vertex[  10] =     1420, length =       824.72
Vertex[  11] =     1423, length =       898.92
```

```
Vertex[  12] =    30494, length =       960.87
Vertex[  13] =    30498, length =      1010.05
Vertex[  14] =    24965, length =      1119.95
Vertex[  15] =    24967, length =      1141.30
Vertex[  16] =       77, length =      1191.43
Vertex[  17] =    39258, length =      1252.64
Vertex[  18] =   109973, length =      1300.11
Vertex[  19] =   109988, length =      1409.27
Vertex[  20] =   188268, length =      1439.17
Vertex[  21] =    17602, length =      1525.97
Vertex[  22] =    17607, length =      1584.71
Vertex[  23] =    11354, length =      1599.75
Vertex[  24] =   142342, length =      1672.44
Vertex[  25] =    30800, length =      1698.52
Vertex[  26] =    78764, length =      1745.17
Vertex[  27] =   272040, length =      1846.43
Vertex[  28] =   171290, length =      1908.86
Vertex[  29] =    10885, length =      2082.62
Vertex[  30] =   108480, length =      2220.21
Vertex[  31] =    12752, length =      2277.19
Vertex[  32] =    12757, length =      2434.97
Vertex[  33] =     1134, length =      2529.65
Vertex[  34] =    13714, length =      2547.21
Vertex[  35] =    46477, length =      2576.57
Vertex[  36] =    16463, length =      2588.89
Vertex[  37] =    28306, length =      2616.99
Vertex[  38] =    42173, length =      2732.36
Vertex[  39] =    88768, length =      2830.31
Vertex[  40] =    54163, length =      2910.48
Vertex[  41] =    54167, length =      2991.84
Vertex[  42] =   104325, length =      3066.79
Vertex[  43] =    54059, length =      3147.90
Vertex[  44] =    39677, length =      3292.40
Vertex[  45] =    70616, length =      3952.39
Vertex[  46] =    37578, length =      4195.97
Vertex[  47] =   116180, length =      4434.38
Vertex[  48] =    55707, length =      4666.01
Vertex[  49] =   142252, length =      4683.07
Vertex[  50] =   142253, length =      4686.96
Vertex[  51] =   142255, length =      4701.04
Vertex[  52] =    63320, length =      4935.83
Vertex[  53] =    63322, length =      4993.11
Vertex[  54] =     7805, length =      5050.18
Vertex[  55] =   173622, length =      5067.41
Vertex[  56] =    81098, length =      5186.96
Vertex[  57] =   130056, length =      5380.96
Vertex[  58] =    48903, length =      5467.00
```

```
Vertex[  59] =   123493, length =      5492.56
Vertex[  60] =    20502, length =      5557.41
Vertex[  61] =    18279, length =      5601.18
Vertex[  62] =    18283, length =      5717.01
Vertex[  63] =    31427, length =      5788.74
Vertex[  64] =   109443, length =      5891.80
Vertex[  65] =   109464, length =      6031.84
Vertex[  66] =   118675, length =      6103.88
Vertex[  67] =    22474, length =      6139.27
Vertex[  68] =    37272, length =      6265.73
Vertex[  69] =   197200, length =      6298.39
Vertex[  70] =    18544, length =      6319.86
Vertex[  71] =    20331, length =      6396.91
Vertex[  72] =   127517, length =      6517.53
Vertex[  73] =    60949, length =      6584.15
Vertex[  74] =   272881, length =      6598.20
Vertex[  75] =   272803, length =      6917.45
Vertex[  76] =   272851, length =      7151.78
Info: path calculated in 319,470us
```

Trace from Vertex 73964 to 272851

6.3 A-star Trace

```
> ./graph_traversal --start 73964 --end 272851 --algorithm astar --file
   dataset/graph_dc_area.2022-03-11.txt
Total visited vertex: 2253
Total vertex on path from start to end = 76
Vertex[   1] =    73964, length =         0.00
Vertex[   2] =    73963, length =        34.91
Vertex[   3] =   121925, length =        67.20
Vertex[   4] =    32152, length =       254.00
Vertex[   5] =   244143, length =       377.27
Vertex[   6] =   243413, length =       467.35
Vertex[   7] =   243414, length =       606.69
Vertex[   8] =   243424, length =       723.74
Vertex[   9] =    79573, length =       786.41
Vertex[  10] =     1420, length =       824.72
Vertex[  11] =     1423, length =       898.92
Vertex[  12] =    30494, length =       960.87
Vertex[  13] =    30498, length =      1010.05
Vertex[  14] =    24965, length =      1119.95
Vertex[  15] =    24967, length =      1141.30
Vertex[  16] =       77, length =      1191.43
Vertex[  17] =    39258, length =      1252.64
Vertex[  18] =   109973, length =      1300.11
```

```
Vertex[  19] =   109988, length =      1409.27
Vertex[  20] =   188268, length =      1439.17
Vertex[  21] =    17602, length =      1525.97
Vertex[  22] =    17607, length =      1584.71
Vertex[  23] =    11354, length =      1599.75
Vertex[  24] =   142342, length =      1672.44
Vertex[  25] =    30800, length =      1698.52
Vertex[  26] =    78764, length =      1745.17
Vertex[  27] =   272040, length =      1846.43
Vertex[  28] =   171290, length =      1908.86
Vertex[  29] =    10885, length =      2082.62
Vertex[  30] =   108480, length =      2220.21
Vertex[  31] =    12752, length =      2277.19
Vertex[  32] =    12757, length =      2434.97
Vertex[  33] =     1134, length =      2529.65
Vertex[  34] =    13714, length =      2547.21
Vertex[  35] =    46477, length =      2576.57
Vertex[  36] =    16463, length =      2588.89
Vertex[  37] =    28306, length =      2616.99
Vertex[  38] =    42173, length =      2732.36
Vertex[  39] =    88768, length =      2830.31
Vertex[  40] =    54163, length =      2910.48
Vertex[  41] =    54167, length =      2991.84
Vertex[  42] =   104325, length =      3066.79
Vertex[  43] =    54059, length =      3147.90
Vertex[  44] =    39677, length =      3292.40
Vertex[  45] =    70616, length =      3952.39
Vertex[  46] =    37578, length =      4195.97
Vertex[  47] =   116180, length =      4434.38
Vertex[  48] =    55707, length =      4666.01
Vertex[  49] =   142252, length =      4683.07
Vertex[  50] =   142253, length =      4686.96
Vertex[  51] =   142255, length =      4701.04
Vertex[  52] =    63320, length =      4935.83
Vertex[  53] =    63322, length =      4993.11
Vertex[  54] =     7805, length =      5050.18
Vertex[  55] =   173622, length =      5067.41
Vertex[  56] =    81098, length =      5186.96
Vertex[  57] =   130056, length =      5380.96
Vertex[  58] =    48903, length =      5467.00
Vertex[  59] =   123493, length =      5492.56
Vertex[  60] =    20502, length =      5557.41
Vertex[  61] =    18279, length =      5601.18
Vertex[  62] =    18283, length =      5717.01
Vertex[  63] =    31427, length =      5788.74
Vertex[  64] =   109443, length =      5891.80
Vertex[  65] =   109464, length =      6031.84
```

```
Vertex[  66] =   118675, length =      6103.88
Vertex[  67] =    22474, length =      6139.27
Vertex[  68] =    37272, length =      6265.73
Vertex[  69] =   197200, length =      6298.39
Vertex[  70] =    18544, length =      6319.86
Vertex[  71] =    20331, length =      6396.91
Vertex[  72] =   127517, length =      6517.53
Vertex[  73] =    60949, length =      6584.15
Vertex[  74] =   272881, length =      6598.20
Vertex[  75] =   272803, length =      6917.45
Vertex[  76] =   272851, length =      7151.78
Info: path calculated in 25,753us
```

Trace from Vertex 73964 to 272851