POLYTECH
NICE-SOPHIA

Project: Map Path Finder

The goal of this project is to learn more about graph algorithms, with special focus on shortest path algorithms which are widely used by all map applications (car navigation system, smartphone app, google map, ...) to find the best route between two given points.
This project has two parts: non-graphic from Section 1 to Section 5, and graphic with Section 6. Students are encouraged to complete the non-graphic Sections first.

1 - Introduction to Graph

To read: https://en.wikipedia.org/wiki/Graph_theory.

A road map is abstracted by a directed graph, vertices represent road intersections, edges represent a road segment between two intersections.
A single edge between two vertices indicates a one-way street

Figure 1: Graph Abstraction of a one-way and a two-way Street.

The example below shows a more complex sub graph and the road map equivalent

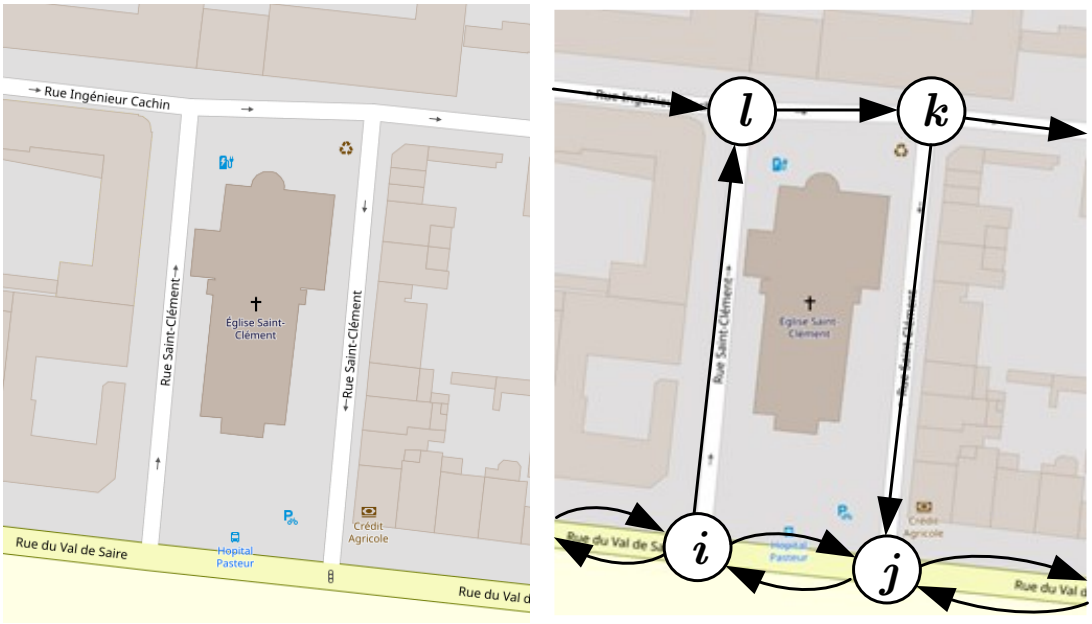
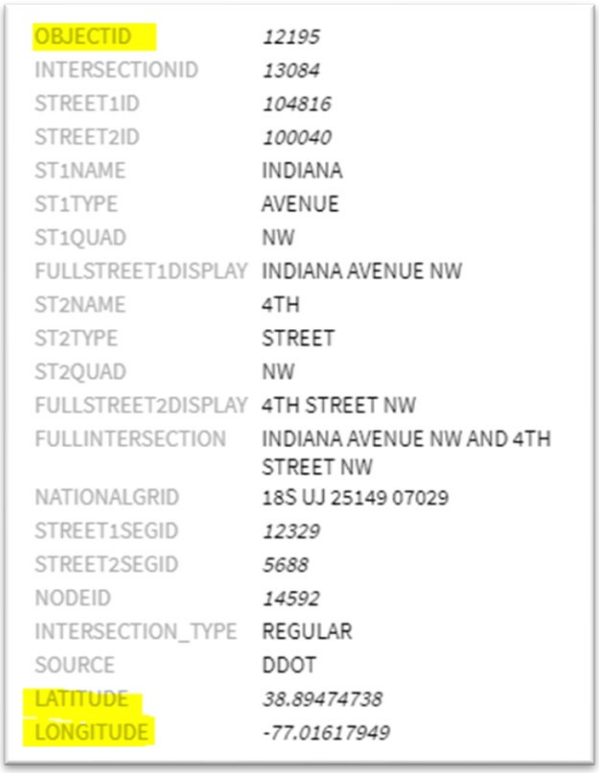


Figure 2: Equivalence between a Graph and Road Map.

Terminology:

The *adjacency list* of a vertex is the list of all vertices directly connected to that vertex through one edge. For example, vertex i is connected to vertices l and j, so the adjacency list of i is $\{l, j\}$.

In “professional curated” map data, a vertex has multiple properties. For this lab, we shall use the LATITUDE and LONGITUDE properties. Example below from https://mapshaper.org/



OBJECTID	*12195*
INTERSECTIONID	*13084*
STREET1ID	*104816*
STREET2ID	*100040*
ST1NAME	INDIANA
ST1TYPE	AVENUE
ST1QUAD	NW
FULLSTREET1DISPLAY	INDIANA AVENUE NW
ST2NAME	4TH
ST2TYPE	STREET
ST2QUAD	NW
FULLSTREET2DISPLAY	4TH STREET NW
FULLINTERSECTION	INDIANA AVENUE NW AND 4TH STREET NW
NATIONALGRID	18S UJ 25149 07029
STREET1SEGID	*12329*
STREET2SEGID	*5688*
NODEID	*14592*
INTERSECTION_TYPE	REGULAR
SOURCE	DDOT
LATITUDE	*38.89474738*
LONGITUDE	*-77.01617949*

Figure 3: Real Life Vertex Properties.

The other interesting property is OBJECTID, a unique 32-bit unsigned number assigned to each vertex.

For edges, the list of properties two to three times larger, the 3 properties interesting for this lab are: FROMID, TOID and SHAPELEN. The first two properties are the start and end vertices for the edge, the last one is the length of the path.

POLYTECH NICE-SOPHIA

2 - Creating Graph Classes

2.1 Graph Classes

You will find many C++ graph packages of various quality and complexity: Boost Graph and Graph Blas are very complex with a steep learning curve; CXX Graph [here] or Graph Lemon [here] are less complex. You can also write you own package to match your need inspired from Graph Snippet [here]. I strongly encourage you to do so.
Some updates will be needed on these packages to be able to read the graph description file provided with this lab.

2.2 Reading a Graph Map File

The graph map is stored in a comma separated value (CSV) file which describes the vertices and the edges. Each vertex line, starting with the character V, is associated with a number, the vertex identifier, followed by the longitude and latitude coordinates, 2 others optional numbers may be given for the planar coordinates. An edge, starting with the character E, is associated with 2 vertex identifiers: the edge start point and the edge end point, additional numbers and string are irrelevant for the present time.
Comments can be found in the file and starts with the character '#'.

An extract of such graph map file is shown below.

```
# Map Graph CVS file
# Vertex List
# V,vertexid,longitude,latitude,x*,y*
V,76,-77.0260635553,38.9686399657,,
V,77,-76.9908002122,38.9030180358,,
V,86,-76.9915263921,38.9029300881,,
# Edge List
# E,source_vid,dest_vid,length,name,extra0,extra1
E,193,194,36.95411263398699,Kennedy St,,
E,195,196,60.53805931418018,North Capitol St,,
E,479,480,33.332227646394806,Unicorn Ln,,
E,663,664,44.053407969870804,15th St,,
```

Figure 4 Example of Graph Map CVS file

You have to load the graph map file into your graph objects, your code could be like:

```
// Code Snipet
#include "graph.h"
//
main(int argc, char *argv[]) {
  // file name is argv[2]
  // create a graph object from given file
  Graph graph(argv[2]);
```

Figure 5: Example of Code Snippet.

Up to you to implement all the required details of a graph library to read the file I have given to you.

In the following I assume that you have a graph library with `Graph`, `Edge` and `Vertex` classes and that you have loaded a graph map file in memory. You are now able to develop algorithms which will operate on the loaded graph.

POLYTECH
NICE-SOPHIA

3 - Breadth First Search

3.1 Introduction

BFS stands for “breath first search” and is a common way to explore a graph to answer questions such as

- Are all vertices reachable for a given vertex?
- How many edges must be traversed to go from vertex V_a to vertex V_b?

The pseudo code to visit all vertices from a given vertex is shown below.

```
// Basic BFS algorithm in pseudo C+ code

Graph::bfs(uint32_t vstart) {

  container<uint32_t> active_queue;
  set<uint32_t> closed_set;

  // ID of the start vertex
  active_queue.push_end(vstart);

do {
    // from the current vertex in the front of the queue
    // compute all vertices reachable in 1 step
    auto vcurrent = active_queue.pop_front();
    closed_set.add(vcurrent);
    for(vnext in adjacency_list of vcurrent) {
      if (vnext is in closed_set) {
        continue;
      }
      if (vnext is not already in active_queue) {
        active_queue.push_end(vnext);
      }
    }
  } while (active_queue.size() != 0)
}
```

Figure 6: Pseudo Code for BFS.

Implement a variant of the above BFS algorithm with a start and end vertices. The do loop must stop when the end vertex is found. You have to find the best STL container for the active queue.

What is the length of the path from vertex 19791 to vertex 50179? How many vertices along the path? Same question from 73964 to 272851.

To answer these questions, you certainly need to update the algorithm to use the parent vertex member of your `Vertex` class.

POLYTECH NICE-SOPHIA

4 - Dijkstra Shortest Path

4.1 Data Structure Update

We assume now that all edges of the graph are associated with a number, called a weight, so for example W_{ij} is the weight of the edge connecting vertex V_i to vertex V_j.

You will have to refactor your Graph library code and assign a weight of each edge as the cartesian distance between the two connecting vertices.
The steps to compute the weight are:

1) When processing the graph CVS file, read the longitude and latitude numbers associated with each vertex.
2) Convert longitude and latitude using a *local* Mercator projection described in the appendix. You now have values (x, y) in a Cartesian space associated with each vertex.
3) Compute the edge weight using the standard Euclidian distance:

$$W_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

4) Note that the edge weight could already be given as the 3rd parameter, highlighted in yellow. When present, as shown in yellow below, it must be used for W_{ij} instead of the computed value from above

```
E,3532,4631,86.78,50,LINCOLN MEMORIAL CIR SW,
```

4.2 Shortest Path Problem

The shortest path problem from V_a to vertex V_b is about finding a path connecting these two vertices which minimizes the sum of the weights of the edges.

The well-known Dijkstra algorithm find a shortest path assuming that the weights of the edges are non-negative, which is the case here.

The pseudo code is given next page, you notice that main differences versus the BFS algorithm are:

1) A partial sort of the queue done at each iteration to ensure that the vertex with the lowest cost is pop-up out of the queue first
2) An association of a weight to a current vertex, representing the sum of the weight of the edges from start vertex to current vertex.

I have left out from the pseudo code the parent each management which must be added to easily build the result path.

```
// Dijkstra algorithm in pseudo C++ code

Graph::dijkstra(uint32_t vstart, uint32_t vend) {

  container<uint32_t> active_queue;
  set<uint32_t> closed_set;

  set_all_vertex_weight_to_max_value();
  // ID of the start vertex
  active_queue.push_end(vstart);

do {
    // from the current vertex in the front of the queue
    // compute all vertices reachable in 1 step
    auto vcurrent = active_queue.pop_front();
    if (vcurrent == vend) break;
    closed_set.add(vcurrent);
    for(vnext in adjacency list of vcurrent) {
      if (vnext is in closed_set) {
        continue;
      }
      auto w = vcurrent.get_weight() + get_edge_w(vcurrent, vnext);
      if (vnext is not already in active_queue) {
        vnext.set_weight(w);
        active_queue.push_end(vnext);
      } else if (w < vnext.get_weight()) {
        vnext.set_weight(w);
      }
    }
    // the partial sort ensure that the vertex with the smallest w
    // is the first on the active_queue
    active_queue.partial_sort();
  } while (active_queue.size() != 0)
}
```

Figure 7: Pseudo Code for Dijkstra Algorithm.

You have to find the best STL container for the active queue.

You will find implementations on the web using a “priority heap”. If you can explain in detail what is a priority heap and why it is better and *you provide you own implementation*, please use a priority heap.

Implement the Dijkstra algorithm and answer the question: total length and number of vertices of the shortest path from vertex 73964 and 272851?

4.3 Additional Remarks

Note that the terminating condition is not when the end vertex is found but when the end vertex is pop-up of the queue. It’s done to avoid pathological cases like the one shown below:

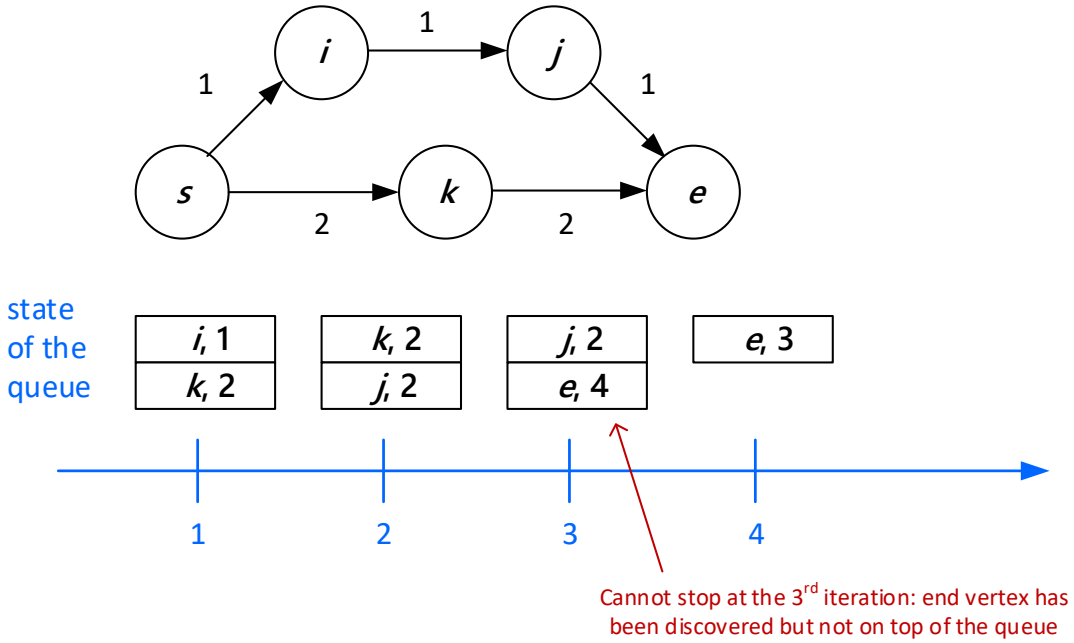


Figure 8: Corner Case for Dijkstra Algorithm.

Note: if the queue at step 2 is sorted in order (j,2),(k,2), the problem does not appear. But these two different orderings are equivalent as the weight is 2 for both.

5 - A-Star (A*) Shortest Path

5.1 Introduction

Dijkstra Shortest Path suffers from a major problem: it explores most of the vertices of the graph, which is not efficient when the said vertices are linked to geographical data, such as the intersections on a map. The A-Star algorithm and variants, make use of this extra information to order the vertices of the queue. The screen capture below has been taken from a YouTube video [link] and compares both algorithms. The visited vertices are purple colored, and clearly shows the advantage of A-Star to quickly find the shortest path.

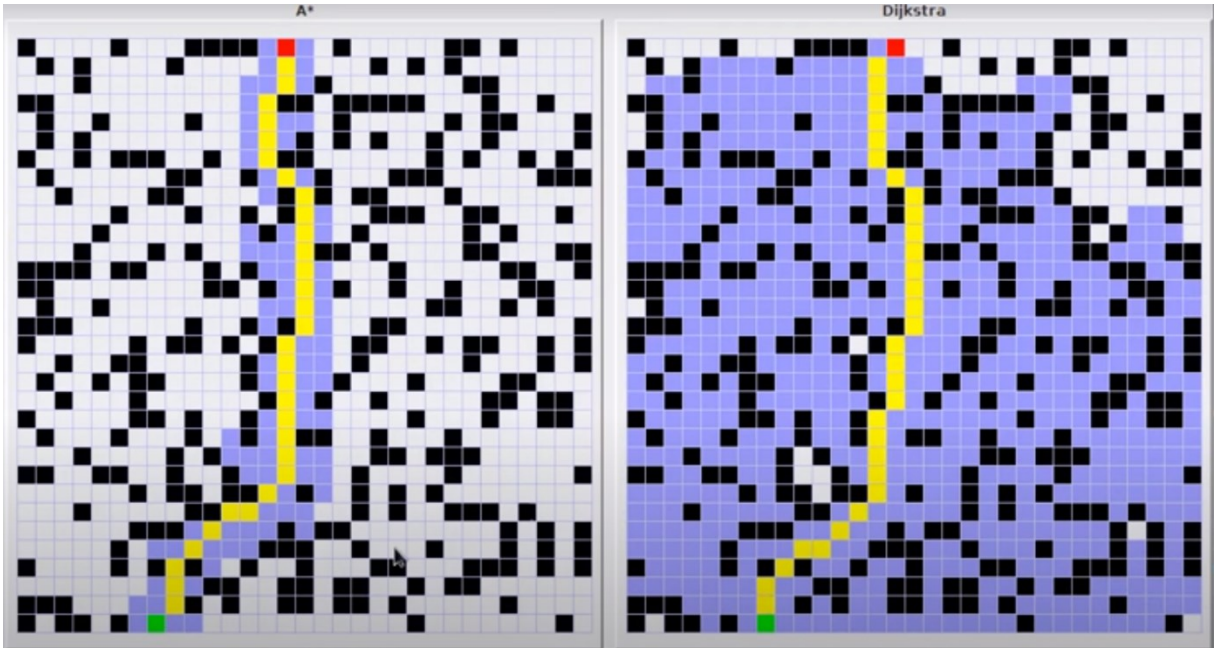


Figure 9: A-Star vs. Dijkstra Algorithms.

5.2 A Star Heuristic

To order the queue, A-Star computes for each vertex, V_n, the sum of two values: the length of the path to that vertex, as found in the Dijkstra algorithm and an estimation of the length of the path from the vertex to the target vertex. The estimated distance from the start vertex, V_s, to the end vertex, V_e , passing though vertex V_n is given by:

$$f(n) = g(n) + h(n)$$

For A-Star to work correctly, the heuristic function, $h(n)$, must never overestimates the real distance, so the length of the line segment from n to e is the best estimation.
I used the same notation as the beautifully done Wikipedia page on A-Star [here], which I invite you to read.

```
// A Start algorithm in pseudo C++ code

Graph::astar(uint32_t vstart, uint32_t vend) {

  container<uint32_t> active_queue;
  set<uint32_t> closed_set;

  set_all_vertex_weight_to_max_value();
  // ID of the start vertex
  active_queue.push_end(vstart);

do {
    // from the current vertex in the front of the queue
    // compute all vertices reachable in 1 step
    auto vcurrent = active_queue.pop_front();
    if (vcurrent == vend) break;
    closed_set.add(vcurrent);
    for(vnext in adjacency_list of vcurrent) {
      if (vnext is in closed_set) {
        continue;
      }
      auto g = vcurrent.get_weight() + get_edge_w(vcurrent, vnext);
      auto f = g + heuristic_distance_estimator(vnext, vend);
      if (vnext is not already in active_queue) {
        vnext.set_weight(g);
        vnext.set_estimate(f);
        active_queue.push_end(vnext);
      } else if (f < vnext.get_estimate()) {
        vnext.set_weight(g);
        vnext.set_estimate(f);
      }
    }
    // the partial sort ensure that the vertex with the smallest estimate
    // is the first on the active_queue
    active_queue.partial_sort_on_estimate();
  } while (active_queue.size() != 0)
}
```

Figure 10: Pseudo Code for A-Star Algorithm.

Implement A-Star algorithm and answer the question: length and total weight of the shortest path from vertex 73964 and 272851?

6 - Displaying Map and Path

Once your path algorithms are validated, you can move on to the second part of this project: displaying the map and the resulting shortest path.

6.1 Map Viewer

The objective of this section is to display on the screen the map graph loaded from a file. As you have already converted longitude and latitude information into cartesian coordinates, it's quite easy to render the graph on the screen.
Don't reinvent the wheel, reuse some Qt code such as the "40000 chips" which is a good starting point.

My result from the Washington DC graph map:

Figure 11: Washington DC Graph Map Rendering.

Render each edge as a straight line on the Q Graphic Scene using the cartesian coordinates.
Render each vertex as a small circle.

Note that the provided data is quite simplified to facilitate the rendering on a screen. In particular, the curved roads are approximated with line segments, though the length of the edge is correct. Therefore, when the edge length is given in the CSV file, it must be used.

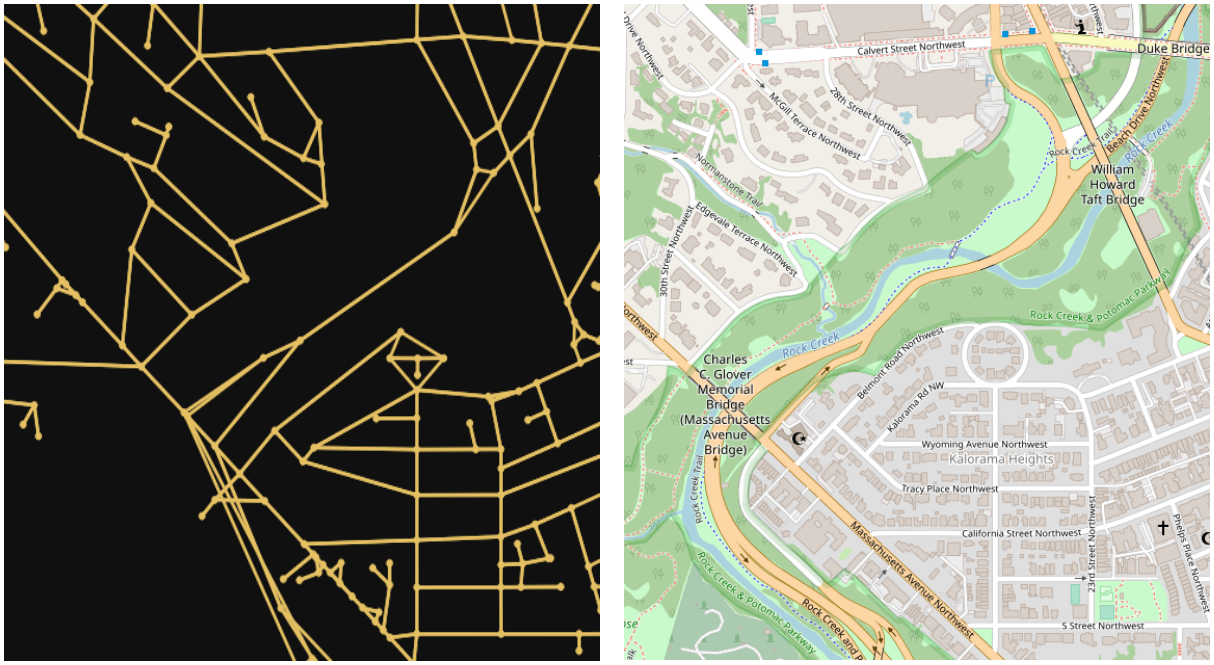


Figure 12: Comparing Details on Curved Roads.

To prepare for path rendering, make sure your code supports the following features:

- Selection of a start point and end point
- Selection of the graph algorithms (BFS, Dijkstra, A-Star)

For example, on my implementation, the start point is with the blue flag, the end point is the red flag. I also display the vertex index on the low left corner of the screen.

Figure 13: Start Point, End Point Details.

6.2 Displaying the Result Path

Once the algorithm, the start point and the end point have been selected, you can compute the path and render it on the screen at shown in my example. I have also rendered the open set and closed set.

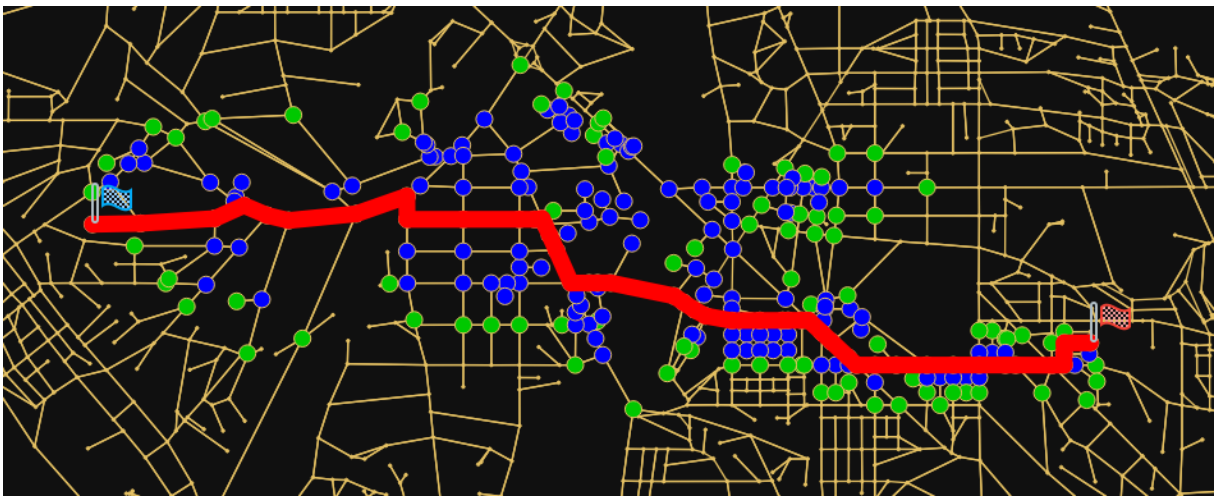


Figure 14: A-Star Result.

Important: the full path must also be displayed on the console as shown in the appendix, with the total time (Section below).

6.3 Measuring Time

You need to measure the time taken by your program to compute the path. You will have to use the STL library which offers several classes and methods.
Read:
http://en.cppreference.com/w/cpp/chrono

You can use the example given in the link and make the necessary updates to report a time duration in microseconds instead of seconds, so don't use `std::chrono::system_clock`.

As an example, I obtained the following result to compute a path from vertex 34384 to 8055 using A-Star

```
INFO: path calculed in 95871us
```

6.4 Friend functions

To facilitate reading of the duration, we propose to add coma separators for thousands, the displayed result looks like:

```
INFO: path calculed in 95,871us
```

You have many possible ways to solve this little problem.
The following line is found in my code:

```
std::cout << "Info: path calculated in " << ELEC4::Commify(my_value) << "us";
```

Here `Commify()` is not a function but a constructor which returns a `Commify` object. This object is then printed using the `operator<<`.

To help you further, I have given you a few lines of my `Commify` class:

```
class Commify {
private:
  ...
public:
  explicit Commify(int64_t value) {
    ...
  }
  friend std::ostream& operator<<(std::ostream &os, const Commify &c) {
    ...
    return os;
  }
};
```

You must complete this class.
Hint: http://en.cppreference.com/w/cpp/io/basic_ostringstream/str

For each of the 3 algorithms, reports

- The path compute time
- The number of vertices in the path
- The length of the path.
- The total number of search vertices.

6.5 Comparing with Google Map

The results is not as fancy as Google but it is worth comparing.

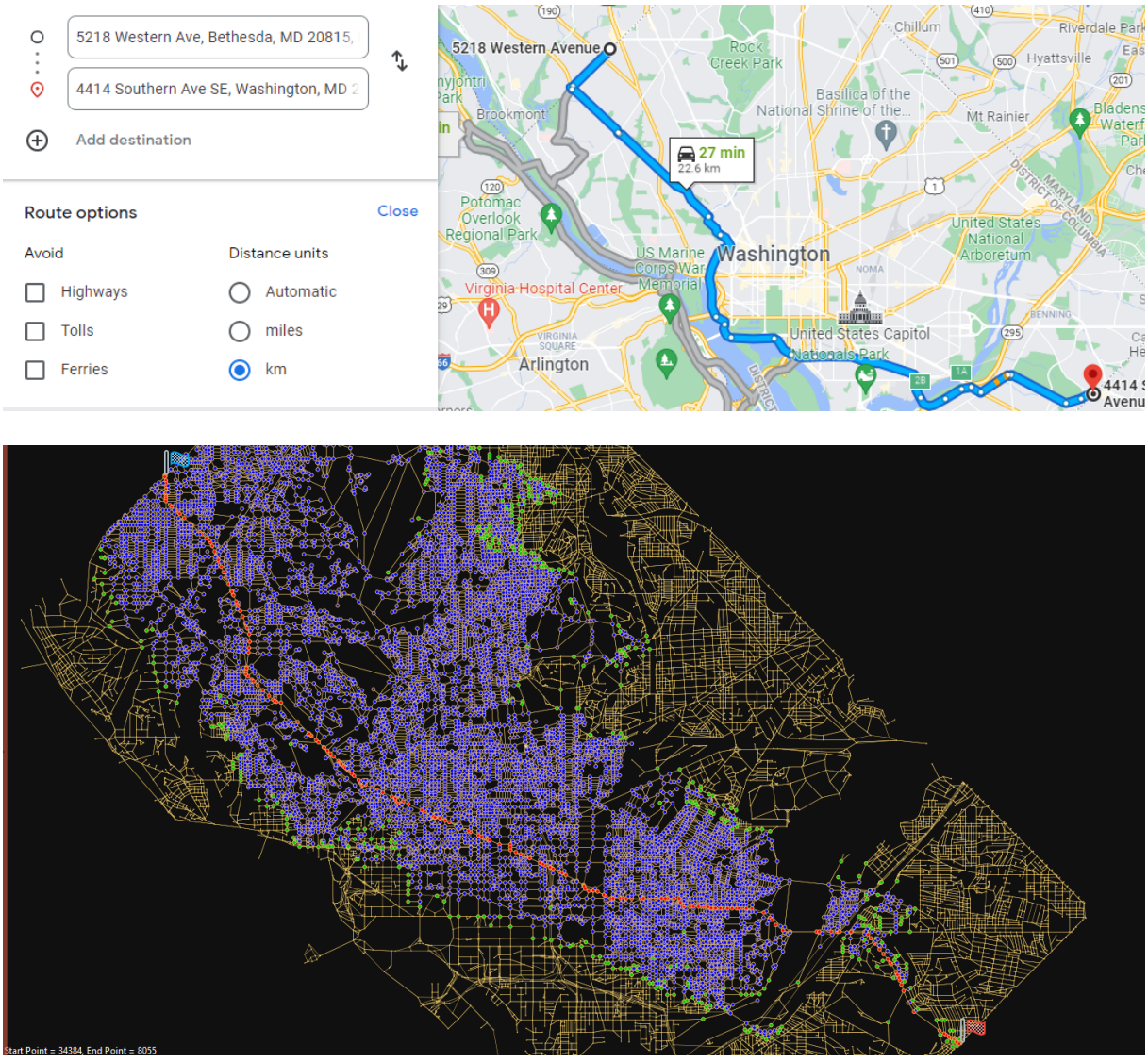


Figure 15: Google Map vs Our Implementation.

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.

Q What is your? Explain the difference vs Google.

6.6 Map UI Do and Don’t

6.6.1 Map Rendering

When you render the map, ensure that the aspect ratio and orientation match the original map. I have captured side by side (1) the original map from Google map, (2) a correct implementation (3) an implementation with incorrect aspect ratio, (4) flipped map implementation

Figure 16: Correct and Incorrect Map Rendering

6.6.2 Display of Path

Once you have computed a path with any of the 3 algorithms, make sure you display the trace as shown in the appendix.

6.6.3 UI Interface Suggestion.

To select the start and end vertices as well as one of the 3 algorithms, consider an UI interface such as the one below

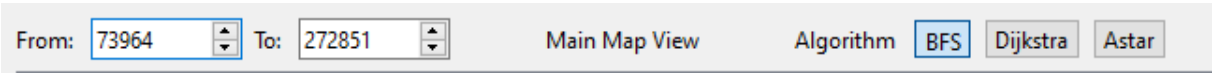


Figure 17:Possible implementation for UI

6.6.4 Display of the path

Be careful in selecting the color, the path must be highly visible.

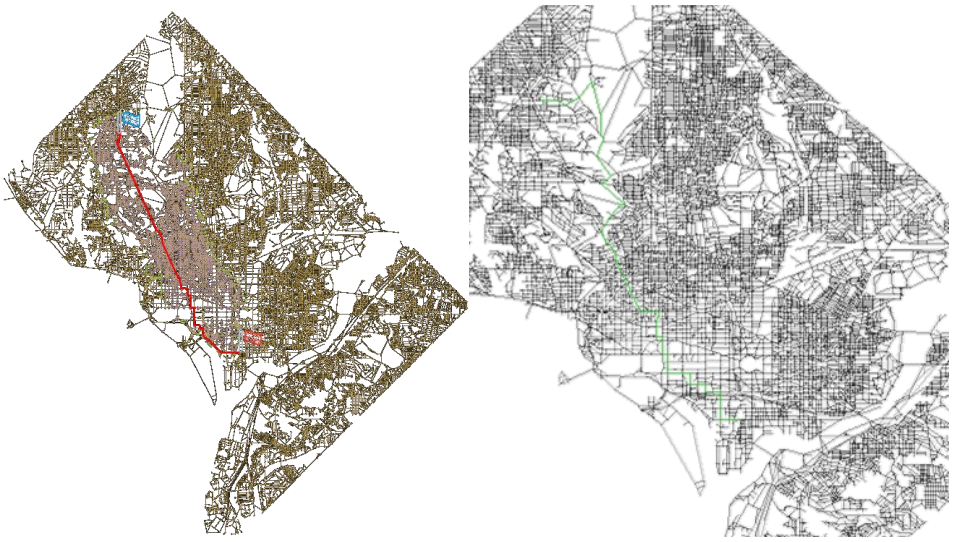


Figure 18: Correct and Incorrect Path Display

6.6.5 Fluid Display

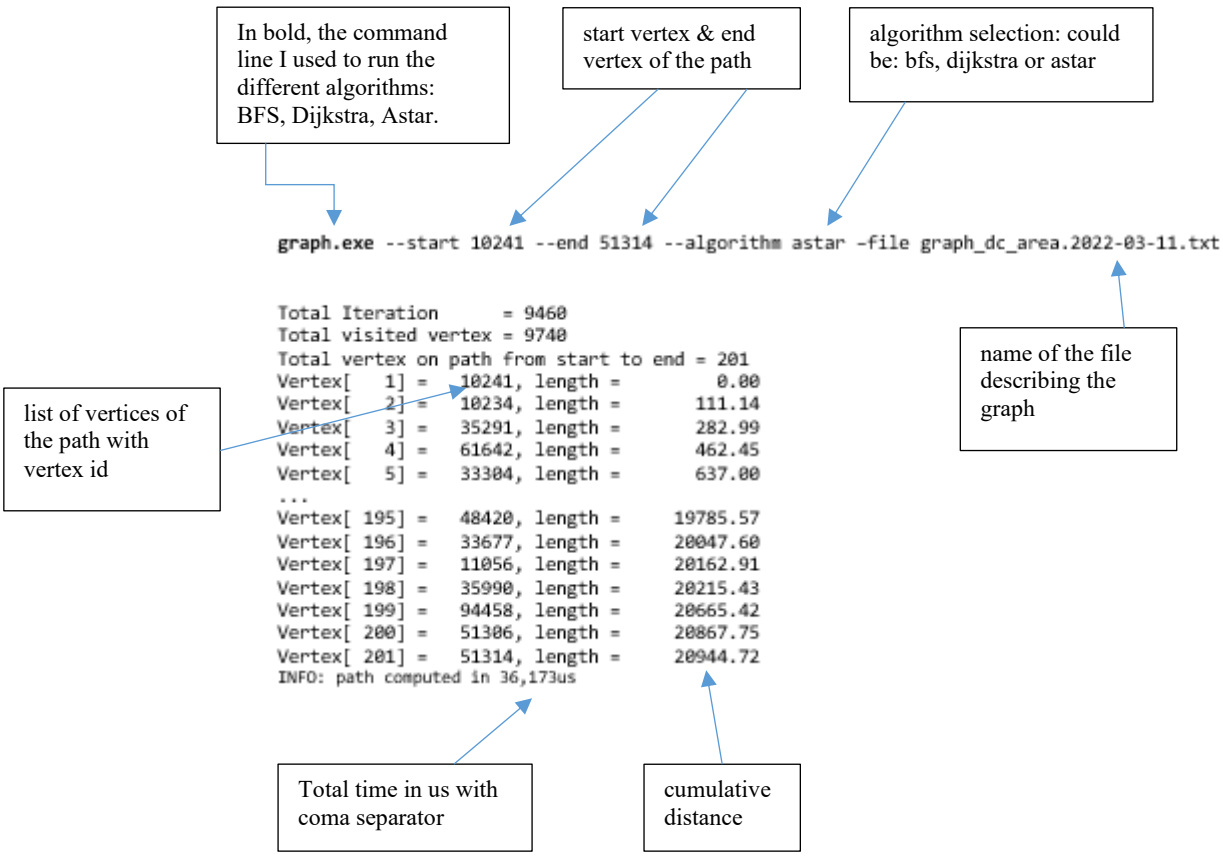
Try to zoom in and out, the response time of your UI must be fast and the screen must not freeze.

6.6.6 Computation Time <1s

The run time of the 3 algorithms must not exceed 1 second including from the worst case a start vertex at top of the map, the end vertex at the bottom of the map.

6.6.7 Console Version

You may want to investigate first a console-based version of the program which display the results on the terminal



POLYTECH
NICE-SOPHIA

7 - Deliverables

This lab will be graded, so prepare the following

An archive created with `tar` linux or `zip` utility containing the following:

Your Qt project

Your report in pdf format

The archive name must be `graph_NAME1_NAME2_NAME3.tar.gz`
Name1, Name2 and Name3 are the names of the people in the group.
To create the archive, use the following command
(replace the names in italic by the folder and file actual names).

```
tar czf graph_NAME1_NAME2_NAME3.tar.gz your_qt_folder your_report.pdf
```

The *your_report.pdf* document must:

- Demonstrate execution of your program, with a few screen captures.
- Provide high level details on how you have coded the graph search algorithm (container used).
- Don't copy paste your code, but only relevant code fragments.
- Don't repeat in your report what is said in this document.
- You will not be graded on the length on your report.
- Answer all the questions found in this lab document

Important: clean-up your Qt folder before creating the archive, object files and executable are not needed, original graph text file is not needed. A `project.pro` file or a `CMakeLists.txt` is needed however. If you provide a console based version, include a `Makefile`.

No credit will be given if the archive is not created with tar or zip
No credit will be given if the report is not a pdf file
No credit will be given if your files cannot be compiled by qt Creator

8 - Appendix: Mercator's Projection

The Wikipedia page is a good introduction (here).

To convert longitude and latitude to cartesian coordinates, you can use the standard Mercator's formula, but the x and y numbers that you will get must be re-scaled corrected because the Mercator projection scales all geometries which are not near the equator.

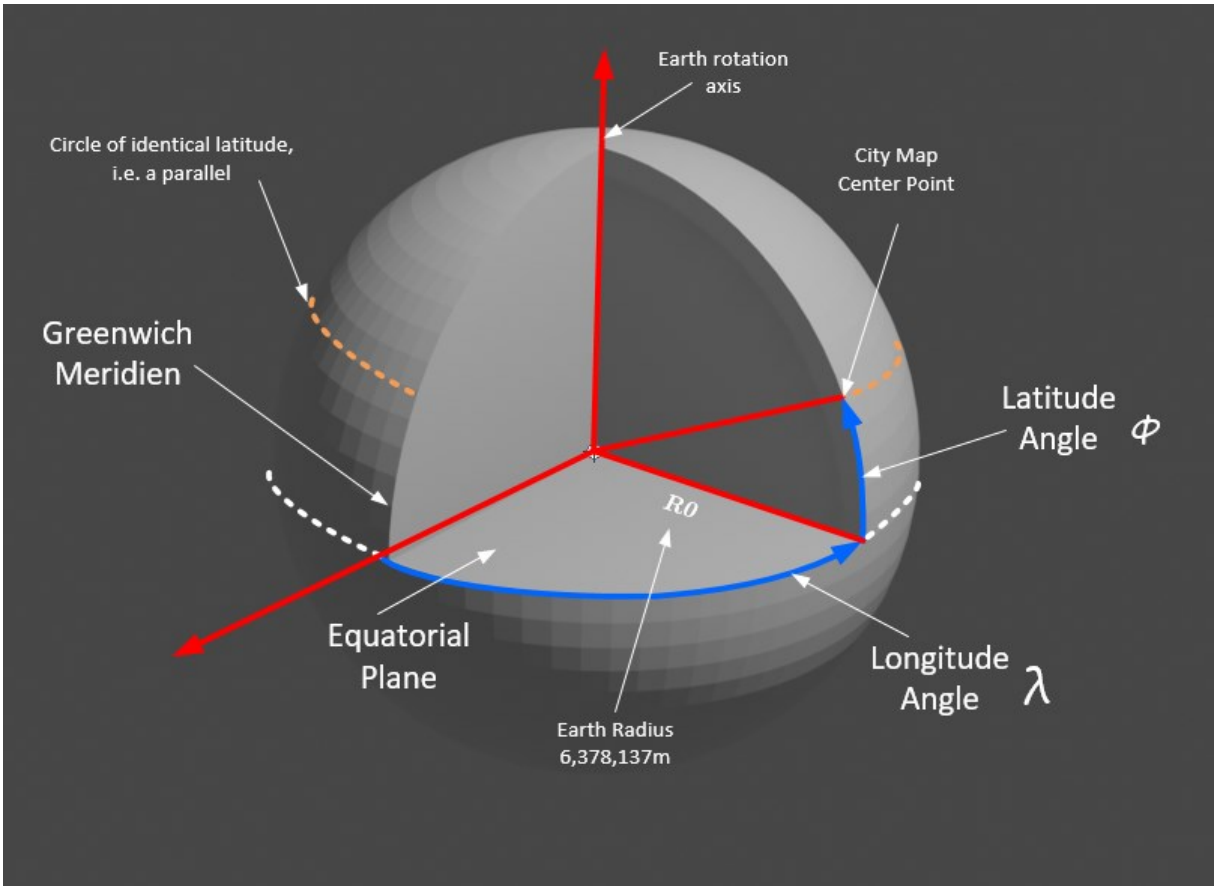


Figure 19: Longitude and Latitude Overview.

Standard Mercator's Formula (all angles are in radian):

$$\begin{cases} x = R_0\ (\lambda - \lambda_0) \\ y = R_0 \ln(\ tan\left(\frac{\phi}{2} + \frac{\pi}{4}\right)) \end{cases}$$

For a local map, like the one we are using, it is possible to use a local Mercator's projection with λ_c and ϕ_c as the coordinates of the center of the local map

$$\begin{cases} x = R_0 \cdot \cos(\phi_c) \cdot (\lambda - \lambda_c) \\ y = R_0 \cdot \ln(\, tan\left(\frac{(\phi - \phi_c)}{2} + \frac{\pi}{4}\right)) \end{cases}$$

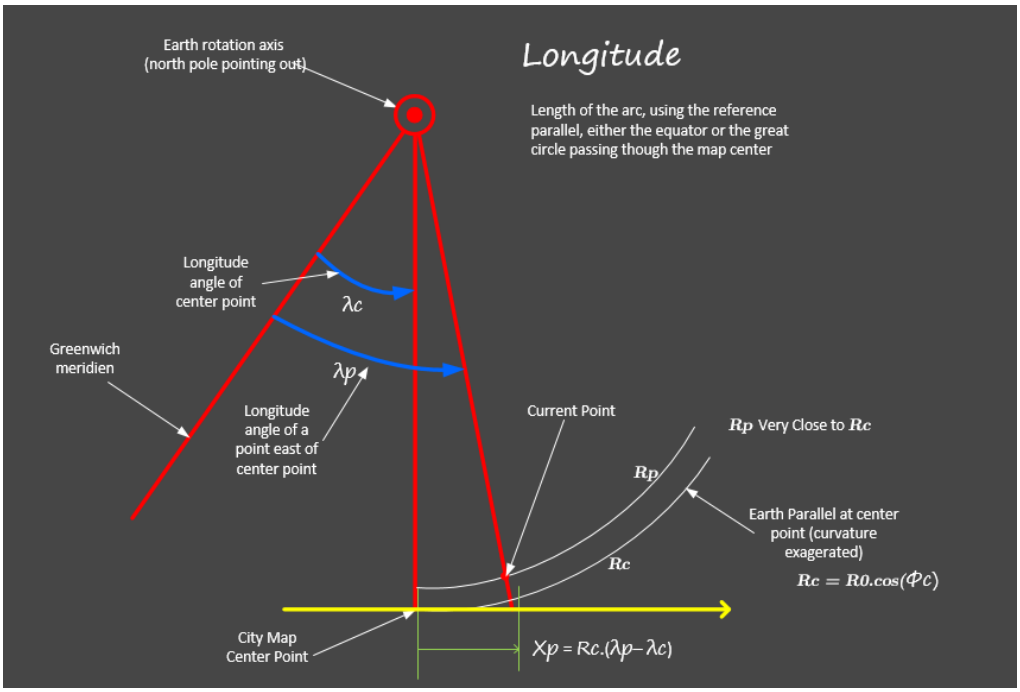


Figure 20: X Axis Formula.

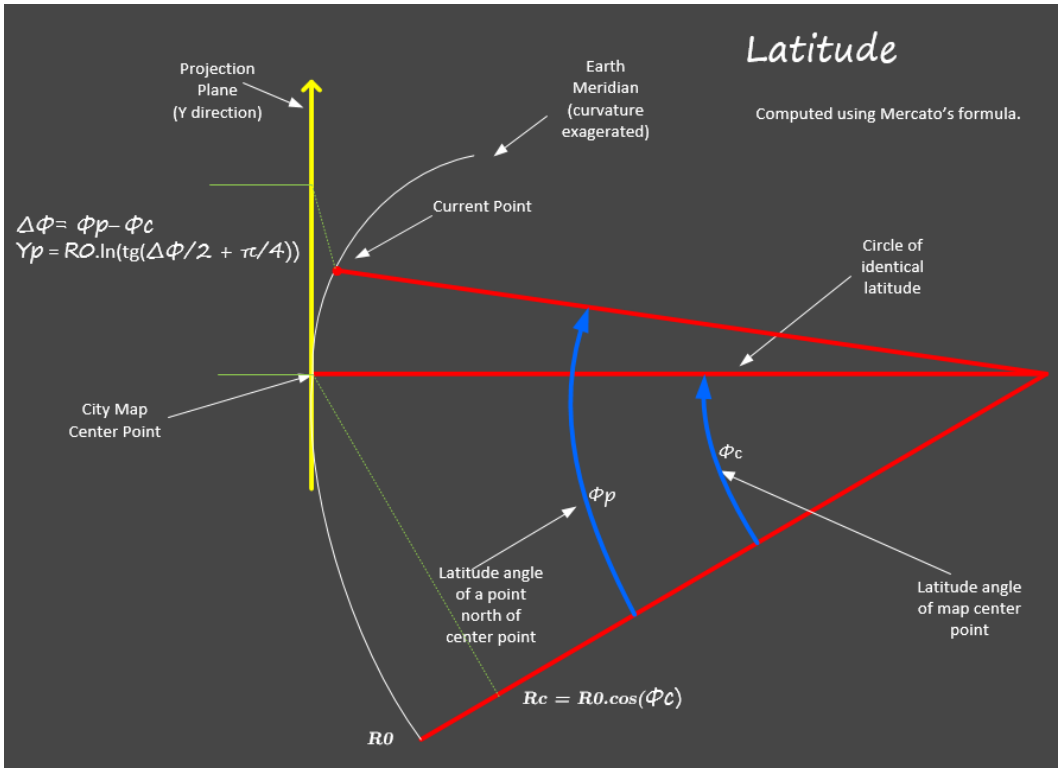


Figure 21: Y Axis Formula.

9 - Appendix: Example of Trace

Example of trace which must be display on the console either from a console program as shown below or from the UI implementation.
Make sure you align all the fields to facilitate the reading.

```
graph.exe --start 10241 --end 51314 --algorithm astar –file graph_dc_area.2022-03-11.txt


Total Iterations       = 9460
Total visited vertices = 9740
Total vertex on path from start to end = 201
Vertex[   1] =   10241, length =         0.00
Vertex[   2] =   10234, length =       111.14
Vertex[   3] =   35291, length =       282.99
Vertex[   4] =   61642, length =       462.45
Vertex[   5] =   33304, length =       637.00
...
Vertex[ 195] =   48420, length =     19785.57
Vertex[ 196] =   33677, length =     20047.60
Vertex[ 197] =   11056, length =     20162.91
Vertex[ 198] =   35990, length =     20215.43
Vertex[ 199] =   94458, length =     20665.42
Vertex[ 200] =   51306, length =     20867.75
Vertex[ 201] =   51314, length =     20944.72
INFO: path computed in 36,173us
```

POLYTECH
NICE-SOPHIA

10 - Appendix: Table of Figures