CMPUT 275 — Tangible Computing **Winter 2020**
Assignment 2 — Directions

Driving Route Finder

You will be implementing a driving route finder (like Google Maps) for the Edmonton area. The driving route finder is a client/server application where Arduino is the client and the C++ application running on a desktop computer is the server. The user will be able to scroll around with a joystick on a zoomable map of Edmonton displayed by the Arduino and select start and end points for their route. The Arduino client will communicate these points to the C++ server, which has all of the street information for Edmonton. The server application will find the shortest path (by distance along the path) and return the waypoints of this path to the Arduino client. The Arduino will then display the path as lines overlaid on the original map. The user can then repeatedly query new points via the Arduino to receive new routes.

We will provide the Arduino code for scrollable/zoomable maps, which will just display the latitude and longitude of a selected point, and a text file containing information about the Edmonton road network, similar to Weekly Exercise 5. You will use the the weighted directed graph class and Dijkstra's algorithm discussed in class for efficient route finding. You are responsible for implementing the following (more details on each task will follow):

1. A function for parsing the text file describing the road network. This function builds the weighted directed graph and stores coordinates (i.e., latitude and longitude) of every vertex.
2. A function for computing the Manhattan distance between every two vertices of this graph. This function is called by the function that builds the graph to compute the weight of an edge before it is added to the graph.
3. A function that efficiently computes least cost (or shortest) paths starting from a given vertex.
4. A C++ route finding server that accepts route finding requests and returns the shortest paths through the specified protocol. For Part I, your application will communicate via stdin and stdout to receive and answer the route finding requests. For part 2, it will communicate with your Arduino.
5. Augment the provided Arduino client program so that it sends a request to the route finding server and then receives the route data from the server.
6. Display the route on the Arduino (i.e., overlay it on the map).

The assignment must be submitted in two parts. **The first part is due 11:55pm Friday, March 13. The second part is due 11:55pm Friday, March 20**.

Part I: Server

For Part I, you will implement the server side of the required functionality (items 1–3 in the list above).

Graph Building

Upon starting up, your server will need to load the Edmonton map data into a *WDigraph* object, and store the ancillary information about vertex locations in an appropriate data structure. You will be given the definition of the WDigraph class in `wdigraph.h`; it is derived from the Digraph class.

Similar to Weekly Exercise 5, the road network data is stored in CSV format in a file. An excerpt of the file looks as follows:

```
V,29577354,53.430996,-113.491331
V,1503281720,53.434340,-113.490152
V,36396914,53.429491,-113.491863
E,36396914,29577354,Queen Elizabeth II Highway
E,29577354,1503281720,Queen Elizabeth II Highway
```

There are two types of lines: those starting with `V` (standing for "vertex") and those starting with `E` (standing for "edge"). In a line that starts with `V` you will find a unique vertex identifier followed by two coordinates, giving the latitude and longitude of the vertex (in degrees). In a line that starts with `E` you will find the identifiers of the two vertices connected by the edge, followed by the street name along the edge. We guarantee that no street name has a comma and no two vertex lines have the same identifier. The endpoints of an edge will be specified on previous lines before the line that describes the edge.

Your code must satisfy the following requirements:

1. You must use the identifiers read from the file and converted to integers as the vertex identifiers in the graph.
2. You must store the coordinates in 100,000-ths of a degree as integers (the reason being that the client side will use this convention, too). If you read a coordinate, such as 53.430996, into a double variable $coord$, you must convert it to a *long long* variable by using $static\_cast < long\ long > (coord * 100000)$. In this example you would get 5343099. The latitude and longitude will then be stored in the following data structure, called Point:

```
struct Point {
    long long lat;  // latitude of the point
    long long lon; // longitude of the point
};
```

You will need to implement the following function:

```
void readGraph(string filename, WDigraph& graph, unordered_map<int, Point>& points) {
  /*
  Read the Edmonton map data from the provided file
  and load it into the given WDigraph object.
  Store vertex coordinates in Point struct and map
  each vertex to its corresponding Point struct.

  PARAMETERS:
  filename: name of the file describing a road network
  graph: an instance of the weighted directed graph (WDigraph) class
  points: a mapping between vertex identifiers and their coordinates

  */
}
```

The server calls this function with filename being `"edmonton-roads-2.0.1.txt"`. Note we are not using the street names in this assignment; thus, they can be ignored. The `readGraph` function is essentially the same as the function you wrote for Weekly Exercise 5 with three main differences: (a) you will instantiate an object from *WDigraph* class enabling you to store and later retrieve the cost associated with each edge, (b) you will store vertex coordinates in a hash table so that you can display the least cost path on the map in the second part of this assignment, (c) it should be the *directed* graph so do not add both directions of each edge to make it undirected (unless both directions exist in the file already).

Cost Function

You will also need to write a function that computes the cost of an edge read from the road network file. The function will take two variables of type *Point* and return the *Manhattan distance* between these points, i.e., the sum of the horizontal and vertical distances between them. The Manhattan distance between two points (x_1, y_1) and (x_2, y_2) is defined as

$$|x_1 - x_2| + |y_1 - y_2|$$

where $|z|$ means the absolute value of z.

Your task is to implement a function with the following declaration:

```
long long manhattan(const Point& pt1, const Point& pt2) {
  // Return the Manhattan distance between the two given points
}
```

Dijkstra's Algorithm

Your task is to write a function to find the search tree of least-cost paths from a given node. The function has the following declaration.

```
void dijkstra(const WDigraph& graph, int startVertex, unordered_map<int, PIL>& tree) {
  /*
  Compute least cost paths that start from a given vertex
  Use a binary heap to efficiently retrieve an unexplored
  vertex that has the minimum distance from the start vertex
  at every iteration

  PIL is an alias for "pair<int, long long>" type as discussed in class

  PARAMETERS:
  WDigraph: an instance of the weighted directed graph (WDigraph) class
  startVertex: a vertex in this graph which serves as the root of the search tree
  tree: a search tree used to construct the least cost path to some vertex

  */
}
```

The declaration of this function is provided in `dijkstra.h`.

The running time of `dijstrak()` should be $O(m \log m)$ where m is the number of edges in the graph represented by `graph` (under the usual assumption that lookups and updates with `unordered_set` take constant time).

Server

After loading the Edmonton map data, your server needs to provide routes based on requests from clients. For Part I, your server will be receiving and processing requests by reading from and writing to `stdin` and `stdout`, respectively. For Part 2, your server will be communicating with your Arduino. In both cases we will use the same protocol described below, the only change will be that the server will be reading from and writing to the serial port connected to the Arduino (how to do this will be described later). Thus, while the description below talks about the server communicating with the Arduino, for Part I the Arduino will not be there and the server will be reading from and printing to `stdin` and `stdout`.

All requests will be made by simply providing a latitude and longitude (in 100,000-ths of degrees) of the start and end points in ASCII, separated by single spaces and terminated by a newline. The line should start with the character `R`. For example,

```
R 5365486 -11333915 5364728 -11335891<\n>
```

is a valid request sent to the server (the newline character is shown with `<\n>` here). The server will then process the request by first finding the closest vertices in the roadmap of Edmonton to the start and end points according to the Manhattan distance (breaking ties arbitrarily), next computing a shortest path along Edmonton streets between the two vertices found, and finally communicating the found waypoints from the first vertex to the last back to the Arduino.

The communication of the waypoints to the Arduino is done by a series of prints, each of which consists of the latitude and longitude of a waypoint along the path. However, before communicating the first waypoint, the server tells the client the number of waypoints. After each print, the Arduino must acknowledge the receipt of data (preventing unwanted buffer overflows) by responding with the character `A`. As an example, assume that the number of waypoints is `8`. Then, the server starts with

```
N 8<\n>
```

and the Arduino responds with

```
A<\n>
```

Next, the server sends the coordinates of the first waypoint (corresponding to the location of the first vertex):

```
W 5365488 -11333914<\n>
```

The client responds again with

```
A<\n>
```

Upon receiving this acknowledgement, the server sends the next waypoint, which the client acknowledges again. This continues until there are no more waypoints, when the server sends the character 'E' followed by a newline:

```
E<\n>
```

This ends the session between the client and the server.

- Part I: At this point your program should end. It will process only one request.
- Part II: The server's next state is to wait for the next request from the client, and the client (to be implemented in Part II) will show the route and allow the user to select new start and end points.

By showing the data sent by the server in black and the data sent by the client in blue, the above exchange of messages looks as follows:

```
R 5365486 -11333915 5364728 -11335891<\n>
N 8<\n>
A<\n>
W 5365488 -11333914<\n>
A<\n>
W 5365238 -11334423<\n>
A<\n>
W 5365157 -11334634<\n>
A<\n>
W 5365035 -11335026<\n>
A<\n>
W 5364789 -11335776<\n>
A<\n>
W 5364774 -11335815<\n>
A<\n>
W 5364756 -11335849<\n>
A<\n>
W 5364727 -11335890<\n>
A<\n>
E<\n>
```

The number of spaces between the letters and numbers in all cases is one.

When there is no path from the start to the end vertex nearest to the start and end points sent to the server, the server should return an empty path to the Arduino by sending the "no path" message, that is:

```
N 0<\n>
```

Upon receiving this message the Arduino should notify the user that there is no path on the road network from the start to the end and should allow the user the select a new pair of start and end points. The Arduino does not need to acknowledge by sending `A<\n>` upon the receipt of the "no path" message. Accordingly, upon sending the "no path" message, the server should consider the answer to the Arduino's request complete and move to the wait state again.

Part I Submission

You should submit all source files compressed as `a2part1.tar.gz` or `a2part1.zip`. Your submission must include the following files.

- `server.cpp`: the server program. Its main function should call `readGraph` and `dijkstra` functions described earlier in addition to input processing and output formatting.
- `heap.h`: declaration and implementation of the binary heap class
- `dijkstra.cpp`: implementation of Dijkstra's algorithm using binary heap
- All source code files related to the graph and weighted graph class.
- `dijkstra.h`: header file for Dijkstra's algorithm
- a custom Makefile with at least these targets: (a) the main target `server` which simply links all object files, (b) the target `dijkstra.o` which compiles the object, (c) the target `server.o` which compiles the object, (d) the target `digraph.o` which compiles the object, and (e) the target `clean` which removes the objects and the executables.
- your `README`, following the Code Submission Guidelines

If you want to add more files to the project, then create approprite `Makefile` targets and clearly indicate in the `README` the purpose of these new files.

You can test your entire server program using the files provided with this assignment on eClass. These test files are in the `tests.tar.gz` file. For example:

```
make server
./server < test00-input.txt > mysol.txt
```

This will load the graph of Edmonton from `"edmonton-roads-2.0.1.txt"`, read a request from `test00-output.txt` instead of the keyboard, and print the output to `mysol.txt` instead of the terminal. You can examine the output by looking at `mysol.txt`. You can quickly determine if the output agrees with with the provided expected output `test00-output.txt` by running

```
diff mysol.txt test00-output.txt
```

Note that we will also test your implementation of Dijkstra's algorithm separately.

Part II: Client

For part II, you will implement the Arduino side of the assignment (items 4–6 of the list in Page 1) and complete the communication protocol. A video demonstrating this part will be uploaded sometime before the deadline for Part I.

We will provide you with code for moving around with a joystick on a scrollable/zoomable map of Edmonton. You will have to implement the interface for selecting two points with the joystick, communicating these points to the server over the serial port, receiving the resulting path and displaying the path overlaid on the map. While the route is displayed, the user should be able to continue to scroll/zoom on the map. If they select new start and destination points, a new path should be retrieved from the server. You may decide whether you want to still display the route when the user has already selected a start point but has not yet selected a destination point.

For the communication protocol, the server must switch to communicating with the Arduino through a serial port via the help the `serialport.h` and `serialport.cpp` files. These will be provided on eClass and an in-class demonstration of how to use these files will be given.

In addition, both the client and the server must implement timeouts when waiting for a reply from the other party. In particular, the timeout should be effective when the server is waiting for acknowledgement of data receipt from the Arduino, and also when the Arduino is waiting for either the number of waypoints, the next waypoint, or the final 'E' character. The length of the timeout is by default one second, except when the Arduino waits for the number of waypoints to be received from the server, in which case the timeout should be 10 seconds. When a timeout expires, the server should reset its state to waiting for a client to start communicating with it. Similarly, when a timeout expires, the Arduino should restart the communication attempt. Both the client and the server should similarly reset their states upon receiving a message which does not make sense in their current state.

Part II Submission

You should submit a single file `a2part2.tar.gz` or `a2part2.zip` containing your solution. The client and server code should be in two separate subdirectories of the submitted archive, called `client/` and `server/`, respectively. We should be able to compile and upload the client code simply be typing `make upload` from within the directory `client/` when working in the VM. We should also be able to compile and run the server by typing `make` followed by `./server` from within the directory `server/` when working in the VM.

Your `README` file should be outside of both directories. That is, after we extract the files from the submitted archive we should see the following three entries:

- The directory `client/`
- The directory `server/`
- The file `README`

The file `README` should be clearly broken into two sections, one for the client and one for the server.

Additional Details

You can view your three main tasks as follows:

- Modify the desktop server so it communicates over the serial port. An example of how to do this will be covered in the lecture on Tuesday, March 3 (see the corresponding entry on eClass for the code used in class).

  Note, in particular, that the `readline()` method will read up to and including the `<\n>` character and include this character in the returned string. However, the `writestring()` will not write an additional `<\n>` character so you will have to do that yourself.

  You must use a timeout of 1 second anytime you want to read from the serial port in the desktop server. After a timeout or anytime the server reads an unexpected message, it should immediately return to the state where it is waiting for the initial request without sending any more messages.

As stated above, if there is no path in the Edmonton road graph, simply communicate a message of `N 0<\n>` and return to the state where it is waiting for a request. Do not wait for an acknowledgement in this case.

- Have the Arduino communicate with the server to get the waypoints. The Arduino should use exactly the protocol described above.

  You should use a timeout of 10 seconds when waiting for the first reply of the form
  `N <num_waypoints><\n>`
  as the server may take a few seconds to compute this path using Dijkstra's algorithm. You should use a timeout of 1 second for every part of the communication (i.e. when waiting for lines starting with `W` or `E`).

  After a timeout or anytime the client reads an unexpected message, it should restart the request from the beginning (i.e. starting with the `R` line).

  The code provided allows up to 500 waypoints to be stored. This is large enough to store almost all possible routes but small enough to give you a few hundered extra bytes of memory to work with. If the path being communicated has over 500 waypoints, your client will treat this as no path. You can handle this however you want: read all waypoints but don't store them, or just quit communication for this request and let the server timeout waiting for the acknowledgement, or anything else that works.

- Drawing the waypoints. If there is a path, you should draw it anytime you update the map (and after the path is first received). Use the drawLine method of the tft object. See the video for some small details you may wonder about.

Misc. Notes

- You may notice the scrolling slows down a lot, especially in the highest zoom level near the lower-right corner of the map. This is ok and it is not your job to fix. This is because redrawing the patch of the map after moving the cursor has to seek to near the end of a really big .lcd file.

  A way around this could be to write the images to known blocks of memory (like with the restaurant data) so we can index into the correct position directly, but this is much more complicated so this simpler solution suffices.

- It is ok if your cursor erases part of the route you drew. Of course, there are ways to avoid this but it is not a required part of the assignment.

- To make the search run faster for queries near each other, you may terminate it as soon as the endpoint is added to the search tree in `dijstrak()`. It won't be a dictionary of all reachable vertices, but it has the endpoint vertex of interest. **This is optional**.

- Often, students are interested in the "A*" heuristic improvement to Dijkstra's algorithm (not covered in class) which works very well with geographic graphs like road networks. Feel free to use this if you want, but clearly document in your README that you have done so. This is the only acceptable excuse to modify the parameters of the `dijstrak()` function. Again, **this is optional** and you will lose marks for not properly implementing this search if you choose to incorporate this into your submitted solution.

- Read and understand the provided client code! It is possible to finish the assignment by only working in a few places of the code, but it certainly helps to understand everything that is going on.

The comments in the code suggesting you should make changes at those locations are just that: suggestions. Feel free to ignore.

You may add additional files if it helps.

- Indicate in the `README` every place where the client code was modified apart from the two `.cpp` files we indicated you should modify! This may be slightly tedious, but it is required. We don't need an exhaustive description of what was changed (it should be clear from your comments), we just need to know where to look.

  If you forget where you edited, you can use diff with your solution against the original template code.

- If you spot a bug in the provided code, you are not required to fix it but the instructors would still like to know about it! Yes, we know that the provided code supports 5 zoom levels but seems to contain data for a 6th zoom level. We decided against using the 6th zoom level because it was far too slow to work with.