Navigation System

You will implement a navigation system (like Google Maps) that allows the user to scroll around on a map, e.g., the map of Edmonton, to select start and end points of a trip. The coordinates of these points are then sent to a route-finding server. This server computes the shortest path between the two selected points (or nearest points to them in the road network), and returns the route information (coordinates of the waypoints along the shortest path) to a *plotter* program. The plotter displays the route as line segments (e.g., red-orange line) overlaid on the original map by connecting the waypoints as shown below.

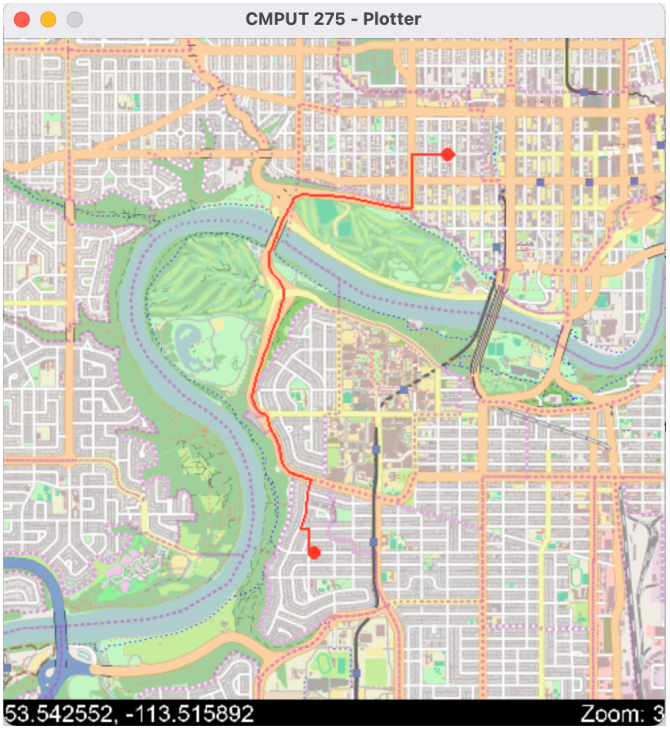

Figure 1: Plotter: A route displayed on the map of Edmonton.

The navigation system can be viewed as a client/server application where client and server communicate with each other using an *interprocess communication* (IPC) method to solve a complex problem. The client is the plotter program written in Python. It displays the map and allows the user to use the mouse or keyboard to zoom and scroll around on a map of Edmonton to select the start and end points of a trip. This *plotter* program will be provided to you as part of the starter code for Part II. The server is a C++ program that runs the graph search algorithm on a road network (i.e., a directed graph) upon receiving a route finding request from the client that includes coordinates of the start and end points of a trip. It then sends coordinates of the waypoints along the route, which are nodes on the shortest path, from the trip's start point to its end point, to the client. The user can repeatedly query new points via the client to receive new routes. In this assignment, the client and server both run on the same machine, i.e., your VM.

We will provide a text file containing information about the Edmonton road network. You will use the the weighted directed graph class and Dijkstra's algorithm (to be discussed in class) for efficient route finding.

The assignment must be submitted in two parts. **The first part is due 9:00pm Monday, March 21. The second part is due 9:00pm Monday, April 4**. Just like weekly exercises and per the Course Outline, submissions will be accepted penalty-free until 11:55pm. There will also be an option to submit before **9:00pm on the respective Tuesday, for a 20% late penalty**. In Part I, your task is to implement the following functions on the server side (more details on each task will follow):

1. A function for parsing the text file that describes 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 two vertices of this graph. It 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. This function must run Dijkstra's algorithm using a C++ `priority queue`, which is a binary heap.

Instead of communicating with the client, your server uses `stdin` and `stdout` to receive and respond to a route finding request. Thus, the plotter will not be used in this part to visualize the routes.

In Part II, you are responsible for completing the server code written in Part I so that it accepts route finding requests from the client (instead of reading from `stdin`), serves each request by computing the shortest path using the functions implemented in Part I, and sends the corresponding route information, i.e., coordinates of the waypoints along the route, back to the client (rather than writing to `stdout`). More specifically, your task is to add functions to the C++ program so that it can use a pair of *named pipes* to communicate with the plotter program that is provided as part of the starter code for Part II and displays the route on the map. The client and server adopt a specific communication protocol which will be discussed later.

Part I: Server

You will implement a standalone C++ program that emulates the route-finding server. It reads the coordinates of the trip start and end points from `stdin` and prints the route information to `stdout`.

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.

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, no two vertex lines have the same identifier, and the identifier fits in an integer (32 bits). 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 using an integer type `long long` because the `float` type does not have enough significant digits. 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. Note the very last digit gets chopped off during the conversion, which is acceptable as the coordinates are accurate enough even when the last digit is lost. 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 Edmonton map data from the provided file and
  load it into the WDigraph object passed to this function.
  Store vertex coordinates in Point struct and map each
  vertex identifier to its corresponding Point struct variable.

  PARAMETERS:
  filename: the name of the file that describes 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, hence they can be ignored. The `readGraph()` function is essentially the same as the function you wrote for Weekly Exercise 6 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 a *directed* graph so do not add both directions of each edge to make it undirected (unless both directions exist in the file already). This is to represent one-way streets in Edmonton.

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 $|z|$ represents the absolute value of z.

Your task is to implement a function with this signature:

```
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.

  NOTE: 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 to construct the least cost path from startVertex to some vertex

  */
}
```

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

For full credit, the running time of `dijkstra()` must 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).

Min Heap

You will use the built-in heaps in C++ to implement Dijkstra's algorithm. They are called *priority queues*: `https://www.cplusplus.com/reference/queue/priority_queue/`

Note, by default in C++, a priority queue is built on a *max heap* meaning the item that is greatest with respect to the < operator will be found at the top of the heap (e.g., via the `top()` method) and this is the item that will be removed when you call the `pop()` method. To make the C++ priority queues act like a

min heap (as required by Dijkstra's algorithm), you can declare a priority queue for a type `T` (that can be compared via a relational operator) as follows:

```
priority_queue< T, std::vector<T>, std::greater<T>>
```

Any priority queue of this type will have the minimum item appearing at the top.

Putting It All Together

After loading the Edmonton map data, your server must provide routes based on the client's requests. For Part I, your server will be receiving and processing requests by reading from `stdin` and writing to `stdout`. Thus, the user interacts with the route-finding server via the standard input/output rather than the plotter running on the client side. In Part II, your server will directly communicate with the client using operation on named pipes (supported by the operating system), following a specific protocol that will be described in the next section. Another difference between the two parts is that the server processes only one route request in Part I, while in Part II, it awaits the next request after processing the current request.

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. In Part I, the line should start with the character `'R'`. For example

```
R 5365486 -11333915 5364728 -11335891
```

is a valid route request. The server will serve the request by first finding the closest vertices in the Edmonton's road network to the start and end points according to the Manhattan distance (breaking ties arbitrarily), computing a shortest path along Edmonton streets between the two vertices found, and finally printing the found waypoints from the first vertex to the last back. You may write an auxiliary function for finding the closest vertex in the graph to a given point.

Before printing the latitude and longitude of the first waypoint, the server prints the number of waypoints it is going to print. After each print, the user 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
```

and the user responds with

```
A
```

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

```
W 5365488 -11333914
```

The user responds again with

```
A
```

Upon receiving this acknowledgement, the server sends the next waypoint, which the user acknowledges again. This continues until there are no more waypoints; the server sends the character `'E'`:

```
E
```

This ends the session between the user and the server. At this point your program should end as it is supposed to process only one request in Part I. Note that the user does not acknowledge the character `'E'` and that each line above is terminated by a newline character `'\n'`.

If we print the data sent by the server in black and the data sent by the client in blue, the above exchange of messages can be displayed as follows:

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

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 user by sending the "no path" message, that is:

```
N 0
```

The user will acknowledge the receipt of the "no path" message by sending `'A'`. The server will send the character `'E'` to end the session.

Part I Submission

Submit all of the required files according to the instructions for GitHub Classroom. Do **not** use compressed files or archives with GitHub Classroom.

When you clone the repository, there should be *a special directory* called `soln/` (this exact name) with the following files in that directory:

1. `edmonton-roads-2.0.1.txt`: the description of Edmonton's road network.
2. `server.cpp`: the C++ program that is our route-finding server. Its main function should call the `readGraph()` and `dijkstra()` functions described earlier in addition to input processing and output formatting.
3. `dijkstra.cpp`: the C++ program that contains the implementation of Dijkstra's algorithm. You should use a min heap in this implementation.
4. `dijkstra.h`: the header file for Dijkstra's algorithm.
5. all source code and header files related to the graph and weighted graph class.
6. a custom `Makefile` (use this exact name; add it to the git repo) with at least these targets:
   (a) the main (and default) target `server` which simply links all object files,
   (b) the target `dijkstra.o` which compiles the respective object,
   (c) the target `server.o` which compiles the respective object,
   (d) the target `digraph.o` which compiles the respective object,
   (e) the target `clean` which deletes the executable and object files.

   As long as they do not interfere with the targets described above, you can add other targets.

   The repo has its own Makefile (in the parent directory) that should not be modified.
7. your `README` (use this exact name) conforms with the Code Submission Guidelines. Create the file yourself and add it to the git repo. Clearly indicate every file that was modified.

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

Note that your files and functions must be named **exactly** as specified above. In your code, do **not** have any extraneous calls to input, output, or other I/O functions that might interfere with automated testing.

A tool has been developed by the TAs to help check and validate the format of your submission, *prior* to your final commit to git.

Run it from the root directory via:

```
python3 submission_validator.py --help
```

to see abbreviated instructions printed to the terminal. You may also run

```
make validate
```

from the main directory to validate the format of your submission.

If your submission passes this validation process, and all validation instructions have been followed properly, you will not lose any marks related to the format of your submission. (Of course, marks can still be deducted for correctness, design, and style reasons, but not for submission correctness.)

When your marked assignment is returned to you, there is a 7-day window to request the reconsideration of any aspect of the mark. After the window, we will only change a mark if there is a clear mistake on our part (e.g., incorrect arithmetic, incorrect recording of the mark). At any time during the term, you can request additional feedback on your submission.

Testing Your Server

We will not provide a testcenter for the assignment. You can manually test your server program using the files provided in a special directory called `tests/`. For example, you can run the following commands from the `soln/` directory:

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

This will load the graph of Edmonton from `"edmonton-roads-2.0.1.txt"`, read a request from `test00-input.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 the following command from the `soln` directory:

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

Please Note: There is not always a unique minimum-cost path between two locations (e.g., between opposite corners of a perfectly-rectangular block). So the output might not agree with the provided output. But you can check if the total driving length of your output agrees with the driving length of the provided output. Also, feel free to discuss these tests and other examples on the eClass discussion forums.

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

Part II: Client/Server Application

You will implement the communication protocol between the client and server. A video demonstrating the expected behaviour of both programs will be uploaded before the starter code for Part II is released.

Plotter Program

We will provide a Python plotter program, `client.py`, which loads the map of Edmonton and makes it possible for the user to scroll around, zoom in/out, and select start and end points using the mouse and keyboard. In particular, the user can:

- press W/S/A/D to move up/down/left/right the patch that is displayed in the window, respectively;
- press R to remove all routes and selected points on the map;
- press Q and E to respectively zoom in and out on the map, keeping the mouse cursor at its previous position;
- click left mouse button to select current point as the start or end point of a trip;
- drag left mouse button to scroll around on the map.

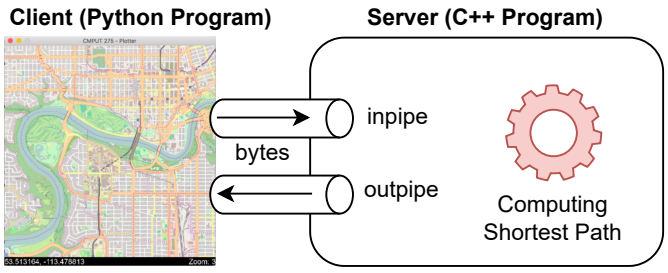

Figure 2: Communication between client and server.

Communication via a Pair of Pipes

When the user selects both start and end points of a trip, the plotter program writes the respective coordinates at once to a *named pipe*, called `inpipe`, as shown below:

```
53.530870 -113.532972
53.530870 -113.514991
```

The route-finding server will read the coordinates from this pipe to compute the shortest path. Subsequently, it will write the waypoints along the shortest path to another pipe, called `outpipe`, so that the client can read the coordinates of the waypoints and display the route by drawing a line between every two waypoints:

```
53.53087 -113.53297
53.51620 -113.53297
53.51620 -113.51499
```

```
53.53087 -113.51499
E
```

The character `'E'` in the last line signals the plotter that there are no more waypoint coordinates to read from the pipe for this route. Note, no trailing spaces are allowed at the end of each line, and there is a single space character between the latitude and longitude of every waypoint. Each line above must end with the newline character (`'\n'`). Make sure you do not print a null character at the end of the line!

The function `create_and_open_fifo()` in the starter code creates the two named pipes, `inpipe` and `outpipe`, and opens them for read and write operations, respectively. You will use `read` and `write` functions (blocking operations) with these pipes in the server to communicate with the client.

We adopt a different communication protocol in Part II. As you can see in the above example, the client reads/writes the coordinates in degrees, whereas the server your wrote for Part I reads/writes coordinates in 100,000-ths of degrees. The client sends the route request in two lines without adding the `'R'` character. Additionally, the server does not send the number of waypoints and does not expect an acknowledgement after sending the coordinates of each waypoint. You must edit the server program to implement this communication protocol. The server only sends the `'E'` character after writing all waypoints for a route request to `outpipe`. Note that in the conversion to 100,000-ths of degrees, the last digit of coordinates will be chopped off as explained before.

After the coordinates of start and end points are sent to the server, the client goes to a state where it waits on receiving waypoints' coordinates from the server. When the route is displayed completely, the user can continue scrolling around on the map. If they select new start and end points, a new path should be retrieved from the server. Therefore, the server must wait for the next request from the client as soon as it answers a route request query. Finally, when the user closes the plotter window, the client writes a single character, `'Q'`, to the pipe. The server must clean up and quit, if it reads this character from the pipe.

This protocol must be implemented completely for full credit.

Testing Your Client/Server Application

To test the full navigation system, you will need to open two terminals; this can be done by running `gnome-terminal` as shown in class. You will then run the client from one terminal and the server from the other terminal (after making the executable). Specifically, in the first terminal, you can run the following command from the `soln` directory to run the server:

```
./server/server
```

Similarly, in the second terminal, you can run the following command from the `soln` directory to run the client:

```
python3 client/client.py
```

Note that the server **must** run before the client. Otherwise, you will get an "Unable to make a fifo" error. You may also get this error if the pipes are not deleted in the previous run, e.g., because the client/server program is terminated unexpectedly. Delete `inpipe` and `outpipe` manually to resolve this error.

Part II Submission

Submit all of the required files according to the instructions for GitHub Classroom. Do **not** use compressed files or archives with GitHub Classroom.

When you clone the repository, there should be *a special directory* called `soln/` (this exact name) that contains the following files and subdirectories:

1. a custom `Makefile` (use this exact name; add it to the git repo) with at least these targets:
   (a) `run`: the main (and default) target which runs the `client` and `server` programs (assuming the executable exists in the `soln/server/` subdirectory) in two different terminals. A new terminal can be opened to run a program by running
   `gnome-terminal -- ./program`
   on the VM. Note that `gnome-terminal` is not installed on lab machines. Thus, you do not need to add this target if you have a Mac with Apple M1 Chip and use a machine in CSC 1-21 instead of the VM. Note that this target is **optional**, but helpful to test your program on the VM using a single make command.
   (b) `clean`: removes all executable, objects, and named pipes.

   You may add other targets as long as they do not interfere with the targets described above. The repo has its own Makefile (in the parent directory) that should not be modified.
2. your `README` (use this exact name) conforms with the Code Submission Guidelines. Create the file yourself and add it to the git repo.
3. `soln/map/` subdirectory: contains PNG files used by the plotter. These files must remain unchanged.
4. `soln/client/` subdirectory: contains the Python program `client.py`. This is the plotter program that we provide and must remain unchanged. Do not add any other files to this directory.
5. `soln/server/` subdirectory: contains the following files
   (a) all source code and header files from Part I
   (b) `edmonton-roads-2.0.1.txt`: the description of Edmonton's road network.
   (c) `server.cpp`: the C++ program that is our route-finding server. This is the only C++ program that you will edit in Part II.
   (d) a custom `Makefile` (use this exact name; add it to the git repo) with at least these targets:
      i. the main (and default) target `server` which simply links all object files,
      ii. the target `dijkstra.o` which compiles the object,
      iii. the target `server.o` which compiles the object,
      iv. the target `digraph.o` which compiles the object,
      v. the target `clean` which deletes the executable and object files.

      As long as they do not interfere with the targets described above, you can add other targets.

A tool has been developed by the TAs to help check and validate the format of your submission, *prior* to your final commit to git. Run it from the root directory via:

```
python3 submission_validator.py --help
```

to see abbreviated instructions printed to the terminal. You may also run

```
make validate
```

from the main directory to validate the format of your submission.

If your submission passes this validation process, and all validation instructions have been followed properly, you will not lose any marks related to the format of your submission. (Of course, marks can still be deducted for correctness, design, and style reasons, but not for submission correctness.)

When your marked assignment is returned to you, there is a 7-day window to request the reconsideration of any aspect of the mark. After the window, we will only change a mark if there is a clear mistake on our part (e.g., incorrect arithmetic, incorrect recording of the mark). At any time during the term, you can request additional feedback on your submission.

Misc. Notes (for Parts I and II)

- The assignment should be completed on the virtual machine that we provided for this course **or** on CSC 1-21 machines. Instructions for connecting to the lab machines remotely will be posted on eClass. To test your program on a lab machine, you can `ssh` into the machine two times, and run the client and server programs in separate terminals. Note that pipe operations are handled differently on Windows and macOS, so you may get an error or unexpected results if you do not use the VM or lab machines.
- If your client or server terminates unexpectedly, the named pipes will not close and you will get an error that says `mkfifo()` failed in the next run. In that case, delete `inpipe` and `outpipe` before running your code again. The `clean` target in your custom Makefile (in the `soln/` directory) can be used to delete these pipes.
- You may notice that it takes longer for the plotter to load the map in the highest zoom level than the lowest zoom level. This is because it has to load and render an image that is larger in size.
- 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 `dijkstra()`. It will not be a dictionary of all reachable vertices, but it has the endpoint vertex of interest. **This is optional**.
- You may have heard about 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 `dijkstra()` 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.
- Carefully read the starter code we provide in Part II, and play with the plotter program to understand how it works! 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.
- If you spot a bug in the starter code, you are not required to fix it but the instructors would still like to know about it!