Project 5

Due December 4, 2019 at 11:59 PM

This project specification is subject to change at any time for clarification. For this project you will be working with a partner. Navigation is a critical to modern society. Once source and destination coordinates are translated to vertices, the shortest or fastest path can be calculated to route the user to their destination. The goal of your project is to write a program that will be able to parse an OpenStreetMap (OSM) file for an area and then to find shortest/fastest routes as fast as possible. Additionally, the goal is to include bus routes in your search for fastest routes. You will be building a program that can find paths, print and save them. OSM files are XML format (more information about OSM can be found at https://wiki.openstreetmap.org/wiki/OSM_XML), so they are human readable and are easily parsed. The bus stop and routes files are CSV and each have two columns. The stops translate stop numbers to node ID numbers in the OSM file. The bus route file lists the name of the bus route followed by the stop number. The stops are listed in order in the route file. It is highly advised that you use your `CCSVReader`, `CCSVWriter`, `CXMLReader`, and `CXMLWriter` from project 4.

A working example can be found on the CSIF in `/home/cjnitta/ecs34/findroute`. Your program is expected to have the same interface as the working example. The full interface can be listed by typing help after launching the program. A program that can convert saved paths into KML files has also been provided `/home/cjnitta/ecs34/kmlout`. You can run the KML converter without arguments and it will create a KML file for each bus route. Directions of how to view a KML file in google maps can be found here https://www.youtube.com/watch?v=1HqQuHeGa38.

The class you will be developing is the CMapRouter. You must keep the existing public interface to the CMapRouter. You may add any private data members or functions that you choose.

```
// Constructor of the CMapRouter
CMapRouter();

// Destructor of the CMapRouter
~CMapRouter();

// Provided function to calculate the distance in miles
static double HaversineDistance(double lat1, double lon1, double
lat2, double lon2);

// Provided function to calculate the bearing in degrees
static double CalculateBearing(double lat1, double lon1,double
lat2, double lon2);

// Loads the map, stops, and routes given the input streams
bool LoadMapAndRoutes(std::istream &osm, std::istream &stops,
std::istream &routes);

// Returns the number of nodes read in by the osm file
```

```
size_t NodeCount() const;
// Returns the node ID of the node specified by the index of the
// nodes in sorted order
TNodeID GetSortedNodeIDByIndex(size_t index) const;

// Returns the location of the node that is returned by
// GetSortedNodeIDByIndex when passed index, returns
// std::make_pair(180.0, 360.0) on error
TLocation GetSortedNodeLocationByIndex(size_t index) const;

// Returns the location of the node specified by nodeid, returns
// std::make_pair(180.0, 360.0) on error
TLocation GetNodeLocationByID(TNodeID nodeid) const;

// Returns the node ID of the stop ID specified
TNodeID GetNodeIDByStopID(TStopID stopid) const;

// Returns the number of bus routes
size_t RouteCount() const;

// Returns the name of the bus route specified by the index of
// the bus routes in sorted order
std::string GetSortedRouteNameByIndex(size_t index) const;

// Fills the stops vector with the stops of the bus route
// specified by route
bool GetRouteStopsByRouteName(const std::string &route,
std::vector< TStopID > &stops);

// Finds the shortest path from the src node to dest node. The
// list of nodes visited will be filled in path. The return
// value is the distance in miles,
// std::numeric_limits<double>::max() is returned if no path
// exists.
double FindShortestPath(TNodeID src, TNodeID dest, std::vector<
TNodeID > &path);

// Finds the fastest path from the src node to dest node. The
// list of nodes and mode of transit will be filled in path.
// The return value is the time in hours,
// std::numeric_limits<double>::max() is returned if no path
// exists. When a bus can be taken it should be taken for as
// long as possible. If more than one bus can be taken for the
// same length, the one with the earliest route name in the
// sorted route names.
double FindFastestPath(TNodeID src, TNodeID dest, std::vector<
TPathStep > &path);
```

```
// Returns a simplified set of directions given the input path
bool GetPathDescription(const std::vector< TPathStep > &path,
std::vector< std::string > &desc) const;
```

Important assumptions for the project:

- Bus travels at the speed limit of the street. If a speed limit is not specified for the way, it is assumed that it is 25mph.
- People walk at 3mph when not on the bus, and can walk either direction along a street.
- Each segment of the bus route (between each bus stop) takes additional 30s, assume that the bus stops at each bus stop for 30s.

A rough outline for approaching this project might be:

1. Design how you want to hold the vertex/edge information. Once you have this done, you can start on reading in the OSM file. You may want to have some translation from the node IDs provided in the OSM file to your internal node IDs (or indices), this will help to simplify your code.
2. The Haversine formula has been provided to calculate the distance between two points on the globe. This will be useful for determining the edge length in miles. You will probably use this during the loading of the ways from the OSM.
3. Once you have read in the OSM file, you should read in the bus stop data and the bus route data. Consider how you will integrate the bus stop data into the vertex/edge information. You will also need to find the fastest bus path for the bus route. You will probably work on that after you have worked on the next step.
4. Once you can load the data you will want to start on the shortest path search (the fastest path is identical, except your edge weights will be time instead of distance). There are a lot of shortest path algorithms to choose from, it is up to you to implement one that correctly finds the shortest path. You may want to also consider that since the algorithm is identical for shortest or fastest path, you will probably want to parameterize your code so that you don't have multiple functions to maintain when you are optimizing your code.
5. Once you can construct a path, the last step is to be able to convert the list of node IDs and transit method back into steps. Make sure not to list taking the same bus multiple times in a row. Test the working `findroute` program using `print` command to see how the steps should be converted into strings.

Files provided:

- davis.osm – The Open Street Map file of Davis (actually a simplified version from original)
- stops.csv – The bus stop file, has conversions from stop ID to node ID
- routes.csv – The bust stop routes, has the name of the route and the stop ID associated with the route (route stops are in order by rows)
- MapRouter.h – The header for the CMapRouter class
- MapRouter.cpp – The cpp for the CMapRouter class
- testrouter.cpp – A google test that tests the functionality of the CMapRouter
- speedtest.cpp – A cpp file for the main speed test for extra credit

You **must** submit the source file(s), the Makefile, .git files, and README.txt file, in a tgz archive. Do a `make clean` prior to zipping up your files so the size will be smaller. You can tar gzip a directory with the command:
`tar -zcvf archive-name.tgz directory-name`

You should avoid using existing source code as a primer that is currently available on the Internet. You **must** specify in your readme file any sources of code that you have viewed to help you complete this project. All class projects will be submitted to MOSS to determine if students have excessively collaborated. Excessive collaboration, or failure to list external code sources will result in the matter being referred to Student Judicial Affairs.

Extra Credit:
There will be two extra credit parts for this assignment. You must complete the first main assignment before attempting the extra credit parts. The first is an improved interface for find route that allows maintaining a history of the commands as well as allowing editing of the line. The second is a speed test for your `CMapRouter`. A working version that has these features can be found in `/home/cjnitta/ecs34/findroute_extra`.

The improved interface needs to handle arrow keys and allow for editing on the line. The following features need to be implemented:

- Up/down arrows to shift between past commands
- Left/right arrows to allow editing in the middle of the line
- Correct rendering of the current line (no extra characters, etc.)

The speed test will test the speed of your `CMapRouter` against the baseline code. The idea is that your program would be part of a server that would be rebooted daily and then would handle as many queries as possible. The more queries your program can handle in a day, the fewer servers that would need to be in operation to handle the daily load. Your program will have a maximum of 30s to load the map and do any precomputation necessary to start handling the requests.

A `speedtest` program has been provide for the baseline (`speedtest_baseline`) and the optimized version (`speedtest_optimized`). The program will output the amount of queries that could be completed in 24hr, it will also output a brief of the path distances/times. A `speedtest.cpp` source file that will calculate the number of queries has been provided. Do not modify the `speedtest.cpp` when constructing your `speedtest` program. A verbose listing of the paths can be created with the `--verbose` option. Speed comparisons will be done with compiler optimizations disabled.

Helpful Hints

- Read through the guides that are provided on Canvas.
- See http://www.cplusplus.com/reference/, it is a good reference for C++ built in functions and classes.

- You may find the following line helpful for debugging your code:
  `std::cout<<"In "<<__FILE__<<" @ "<<__LINE__<<std::endl;`
  It will output the line string `"In F @ line: X"` where F is the name of the file and X is the line number the code is on. You can copy and paste it in multiple places and it will output that particular line number when it is on it.
- Make sure to use a tab and not spaces with the Makefile commands for the target
- Use a `.gitignore` file to ignore your object files, and output binaries.
- Do not wait to the end to merge with you partner. You should merge your work together on a somewhat regular basis.
- Non-canonical mode will be necessary for the improved interface, see https://www.gnu.org/software/libc/manual/html_node/Noncanon-Example.html for more information
- The improved interface will need to be able to support arrow keys which come in a s VT100 codes. See https://www.csie.ntu.edu.tw/~r92094/c++/VT100.html for more information.