# Project 4

Due September 9, 2018 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 via GPS is a critical to modern society. In order to navigate an individual to their location, first GPS coordinates must be translated to a vertex in the map. After the 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. 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.

OSM files are XML format, so they are human readable and are easily parsed. In order to aid in reading in the OSM files, an XMLParser class has been provided to simplify the parsing of the XML.

A working example can be found on the CSIF in /home/cjnitta/ecs60/proj4. Your program is expected to handle approximately the same number of requests as the baseline example when executed on at least 10,000 samples. Extra credit may be available for solutions that significantly outperform the provided baseline solution, or that outperform the optimized version provided. You may use any C++ STL container you want for this project. Your program may not have memory leaks and will be tested using valgrind, be concerned with the definitely lost and indirectly lost lines as there will may be errors with the external libraries like expat.

You can run the example program with the command:
```
./proj4_baseline file.osm [num_samples]
```

If it is taking a significant amount of time to execute you may want to lower the number of samples from the default 1000 to something smaller like 100 or 10:
```
./proj4_baseline file.osm 100
```

You solution must match the output of the baseline. It is best to output the results to a file and run diff on the files. Using bash you can output the results to a file and still see the run time stages with the command:
```
./proj4 file.osm 1> outfile.txt
```

You can diff two files with the command:
```
diff file1.txt file2.txt
```

If your two files match, the only difference should be run times, you should see something like:
```
3001,3003c3001,3003
< Duration (load): 114
< Duration (proc): 23394
< Queries per day: 3693249 (+-116790), 3576459 min
---
> Duration (load): 29753
> Duration (proc): 13035
> Queries per day: 6626025 (+-209533), 6416492 min
```

A map of Davis has been provided. It can be found in the data directory of the given tgz as well as the /home/cjnitta/ecs60/proj4 directory on the CSIF.

You **must** submit the source file(s), a Makefile, README.txt file and interactive grading timeslot CSV file, in a tgz archive. 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.

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.
2. The CXMLParser class provided uses the expat C library to parse XML and provides the beginning and end information for each XML element. You will want to subclass the CXMLParser and implement the StartElement and EndElement functions. When an istream that holds XML data is passed to the Parse function of your subclass, the StartElement and EndElement functions will be called. It will be up to you to build your graph when the functions are called. You may want to just have your CImplementation be the subclass so that the graph can be built inside of it, but that design choice is entirely up to you.
3. After the data can be read in, you will want to be able to find the closest vertex given a lat/lon location. The Haversine formula has been provided to calculate the distance between two points on the globe. You could exhaustively search the vertices to find the closest one, but this will obviously be time consuming. An exhaustive search may not be a bad idea for an initial solution if you wish to work on later parts and come back to this part for later optimization.
4. Once you can load the data and find the closest vertex, 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. You may want to have some translation from the node IDs provided in the OSM file to your internal node IDs, this will help to simplify your code.

5. Once you can construct a path, the last step is to be able to convert the list of node IDs back into street names. You may want to have a mapping of edges (source node, destination node) to street names. Make sure not to list the street name twice in a row, and if the edge doesn't have a street name, assume that it is either part of the current or subsequent street.