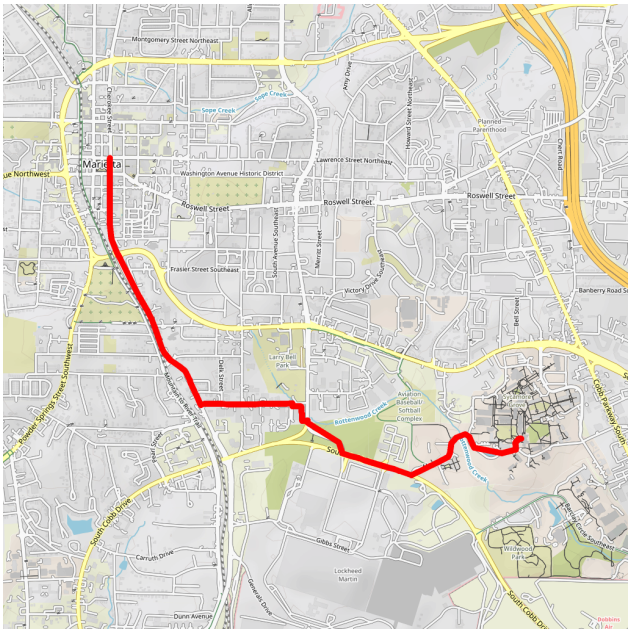
Homework 1 (Due 01/21)

CS7253: Graph Algorithms
Kennesaw State University
Spring 2022

In this project, we will implement Google Maps (sort of). In particular, we will implement Dijkstra's algorithm, and run it on a map of Marietta. Our goal is to find the shortest path (in meters) between two given places in Marietta, for example, from the J Building at KSU to Sweetreats at Marietta Square:



Skeleton code is provided to help you get started. It includes implementations of the following classes:

- an `UndirectedGraph` class which implements an undirected graph, which maintains a list of `Node` objects, where each `Node` object maintains a list of `Edge` objects that are incident to the node.
- a `Map` class which is derived from the `UndirectedGraph` class, where we have `Intersection` objects derived from `Node` objects, and `Street` objects derived from `Edge` objects. Each `Intersection` has a latitude and longitude, and each `Street` object has a length and a (possibly empty) name.

In this project, you are to implement the following function:

```
Map::Path find_shortest_path(Map::Intersection* s, Map::Intersection* t, Map::Map* map);
```

which returns the shortest path from node/intersection `s` to node/intersection `t` in the given `map`.
I am suggesting, but not requiring, that you use the following additional data structures:

- `std::unordered_map<Map::Intersection*,int> distances` can be used to keep track of the shortest (known) distance to a given node.
- `std::unordered_map<Map::Intersection*,bool> open` can be used to keep track of any node whose shortest path we are looking for
- `std::unordered_map<Map::Intersection*,bool> closed` can be used to keep track of any node whose shortest path has already been found

- `std::unordered_map<Map::Intersection*,Map::Intersection*> back_pointer` is a data structure that can be used to keep track of, for a given node n, the node m that we last visited on the shortest path to reach n. This data structure is used to recover the nodes on the shortest path. (One could alternatively/additionally keep track of the last edge, if you want to print out each street used).

The file `src/dijkstra.cpp` contains the skeleten of the function you are to implement, with some helper functions included, for your convenience.

- `Intersection* find_closest_open_node(unordered_map &distances, unordered_map &open)` is a function that returns the closest open node
- `LinkedList extract_path(Intersection* t, unordered_map back_pointer)` returns the shortest path to node `t`, when given a populated `back_pointer` data structure.

The file `src/main.cpp` contains the main function, and two tests. The first test corresponds to the example we covered in class. The second test corresponds to a map of Marietta, and requests a path from the J Building at KSU to Sweetreats at Marietta Square. The expected shortest path has a length of 4,337 meters (or 2.69488 miles). (Compare this to the route suggested by google maps).

Visualization: For the Marietta route test, the GPS coordinates of each node will be printed out. These GPS coordinates can be visualized using "my google maps" (`https://www.google.com/maps/d/`). You can create a new map, and create a new layer by importing a csv (comma-separated values) file containing the GPS coordinates. The map and path that we showed at the beginning of the document was drawn by a python script that is included with this project, and can also be used to visualize routes.
Turn in: your modified version of `dijkstra.cpp` onto the course website under "Assignments" and "Homework 01." Projects are due Friday, January 21 by 11:59pm. Please start early in case you encounter any unexpected difficulties.

Included files:

- `homework01.pdf`: this document
- `Makefile`: an optional sample Makefile if you use the command line. Type "make" on the command line to build the project. The executable is "build/main".
- `src`: the directory containing all the source files for this project.
- `scripts`: the directory containing all scripts and data used to generate the map, and for visualizing paths.

Hint: Make sure your code compiles, passes the given tests, and does not crash or SEGFAULT.

Hint: Use the debugger.

For fun: I have included all of my scripts for creating the Marietta example. To create such an example, one needs a map (which you can download from `openstreetmap.org`), you need to convert the map to a data structure you can use in your code, and ideally you need a way to visualize the map as well as any path that you generate. All of these scripts have been included, which are mixture of `python` and `jython` scripts (note that you need to also install the appropriate modules). Note that again, these scripts are included for fun. They are not required to do the project.

The file `src/marietta_map.cpp` is a C++ file that contains a hard-coded version of a map of Marietta. This map was originally downloaded from the following address:

```
https://www.openstreetmap.org/export#map=15/33.9470/-84.5259
```

The original map downloaded from `openstreetmap` is an .xml file, which was processed by the `jython` script `scripts/01-test.py`. `jython` is a python interpreter implemented in `Java`, which is used to interface with the Java library `graphhopper` for reading maps and for projecting GPS coordinates onto a map (the version of `graphhopper` that was used to implement the script has been included).

Plotting: The project also contains scripts for visualizing the paths that you found. For an example image, see `scripts/maps/path.png` for a static map (image) and `scripts/maps/example.html` for a dynamic map (webpage). Use your web browser to open either of these. See the script `scripts/02-example.py`, which requires a number of additional python modules.

Converting a map to C++: The script `scripts/03-create_header.py` was used to convert the map of Marietta to C++ code. Note that the included `marietta_map.cpp` file is large, and may take some time to compile (the compiler should only need to compile it once however).

Finding a source and destination node: The script `scripts/04-get_source_sink.py` can be given a pair of GPS coordinates, and return you a pair of indices corresponding to their closest nodes on the map.