Project #07: Part 2 of 2: navigating with Dijkstra's alg

Complete By: **On-time:** Friday 5/1 @ 11:59pm
Late: Friday 5/8 @ 11:59pm (-10%)

Assignment: C++ program to navigate openstreetmap

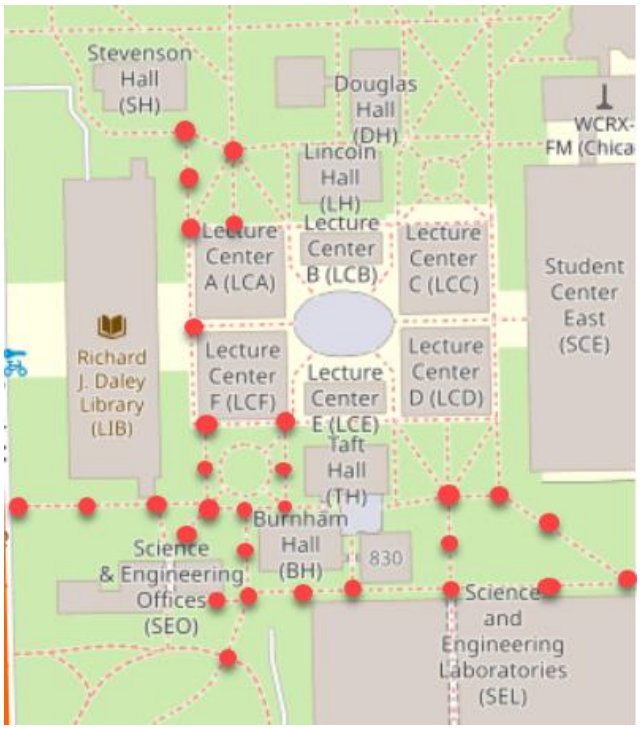
Policy: Individual work only, late work *is* accepted

Submission: all program files via Gradescope; the first 12 submissions are free, each additional submission 1 pt

Background

We're all familiar with navigation apps. While we don't have the ability to display the results graphically, we can at least perform the back-end operations of loading the map, building the graph, and computing the shortest weighted path between two points. In our case we're going to navigate between UIC buildings on the East campus, using the footpaths. But the foundation is there to extend the program to do more general navigation between any two points.

We are working with open-source maps from https://www.openstreetmap.org/. Browse to the site and type "UIC" into the search field, and then click on the first search result. You'll see the East campus highlighted. Notice the "export" button --- we used this button to download the map file (map.osm) we'll be working with.



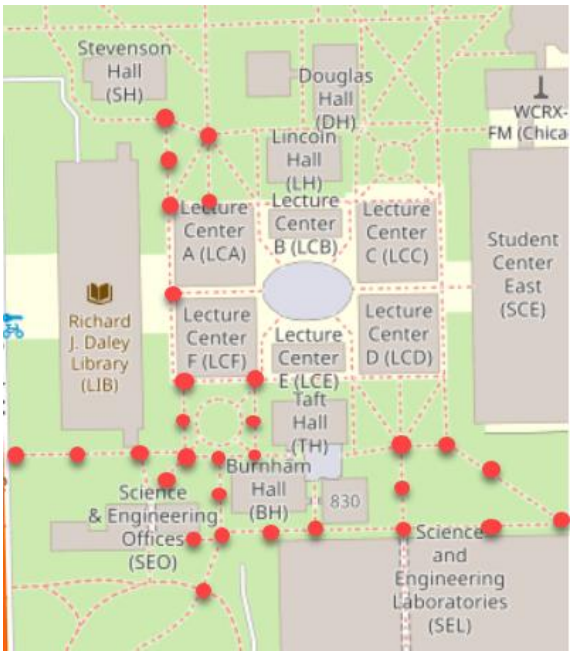
Zoom in. We're going to focus on two features of a map: "Nodes" and "Ways". A **node** is a point on the map, consisting of 3 values: id, latitude, and longitude. These are shown as red dots (there are thousands more). A **way** is a series of nodes that define something. The two most important examples in our case are **buildings** and **footways**. In the screenshot to the right, the buildings are labeled and the footways are the dashed lines. For a building, the nodes define the building's perimeter. For a footway, the nodes define the endpoints of the footway, but might also include intermediate points along the way (especially if the footway is not a straight line). More details of openstreetmap are available on Wikipedia.

Assignment

The assignment is to write a console-based C++ program to input a campus map (e.g. UIC's East campus) and navigate between buildings via footways. The program should be general enough to work with any college map, though we don't plan to extensively test this. Given the time constraints, we're going to provide helper functions to read the map for you, which are available in XML format. Your job is to build the underlying graph, input two buildings from the user, and then use Dijkstra's algorithm to find the shortest weighted path. This is repeated until the user enters # for the start building. Here are the main program steps:

```
** Navigating UIC open street map **

Enter map filename> map.osm

# of nodes: 18297
# of footways: 382
# of buildings: 34
# of vertices: 18297
# of edges: 3596

Enter start (partial name or abbreviation), or #> SEO
Enter destination (partial name or abbreviation)> SCE
Starting point:
 Science & Engineering Offices (SEO)
 (41.870827, -87.650253)
Destination point:
 Student Center East (SCE)
 (41.872051, -87.648041)

Nearest start node:
 6291902791
 (41.870796, -87.650207)
Nearest destination node:
 1645121521
 (41.871909, -87.648247)

Navigating with Dijkstra...
Distance to dest: 0.19574903 miles
Path: 6291902791->5632908778->1647971987->463814283->5
632908773->462014177->1587477497->482724372->462014176
->462010748->462010768->464345546->1645121521

Enter start (partial name or abbreviation), or #> #
** Done **
```

1. Load map into xmldoc.
2. Read nodes.
3. Read footways.
4. Read buildings.
5. Add nodes as vertices.
6. Add edges based on footways.
7. Input start and destination buildings, locate on map.
8. Search the footways and find the nearest nodes to the start and destination buildings; these become the "start" and "dest" nodes.
9. Run Dijkstra's algorithm from the start node.
10. Output the distance and path from start node to destination node. If no path exists, output "`Sorry, destination unreachable`".
11. Repeat with another pair of buildings.



The footways don't actually intersect with the buildings, which is the reason for step #8: we have to find the nearest node on a footway. Then navigation is performed by moving from node to node (red dots) along one or more footways. The footways (dashed lines) intersect with one another, yielding a graph. The graph is built by adding the nodes as vertices, and then adding edges between the nodes based on the footways. Since our graph class created directed graphs, you'll want to add edges in both directions. Nodes are identified by unique 64-bit integers; use the C++ datatype "long long". The edges weights are distances in miles; use "double".

From XML to data structures

An openstreetmap is represented as an XML document. Very briefly, an XML document is a text-based representation of a tree, with a concept of "parent" and "children". An openstreetmap starts with an <osm> node, and contains <node>, <way>, and other child nodes:

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" … >
   <node id="25779197" lat="41.8737233" lon="-87.6456365" … />
   .
   .
   .
   <way id="32815712" … >
     <nd ref="1645121457"/>
     .
     .
     .
     <nd ref="462010732"/>
     <tag k="foot" v="yes"/>
     <tag k="highway" v="footway"/>
   </way>
   .
   .
   .
   <way id="151960667" … >
     <nd ref="1647971990"/>
     <nd ref="1647971996"/>
     .
     .
     .
     <nd ref="1647971990"/>
     <tag k="name" v="Science &amp;amp; Engineering Offices (SEO)"/>
   </way>
   .
   .
   .
</osm>
```

Looks very similar to HTML, right? HTML is a special case of XML. We are using tinyxml2 to parse the XML.

Functions are provided in "osm.cpp" to read the XML and build a set of data structures. First, here are the structure definitions (defined in "osm.h"):

```
//
// Coordinates:
//
// the triple (ID, lat, lon)
//
struct Coordinates
{
```

```
  long long ID;
  double Lat;
  double Lon;
};


//
// FootwayInfo
//
// Stores info about one footway in the map.  The ID uniquely identifies
// the footway.  The vector defines points (Nodes) along the footway; the
// vector always contains at least two points.
//
// Example: think of a footway as a sidewalk, with points n1, n2, ...,
// nx, ny.  n1 and ny denote the endpoints of the sidewalk, and the points
// n2, ..., nx are intermediate points along the sidewalk.
//
struct FootwayInfo
{
  long long         ID;
  vector<long long> Nodes;
};


//
// BuildingInfo
//
// Defines a campus building with a fullname, an abbreviation (e.g. SEO),
// and the coordinates of the building (id, lat, lon).
//
struct BuildingInfo
{
  string      Fullname;
  string      Abbrev;
  Coordinates Coords;
};
```

A **node** is the map is stored as a Coordinate, a **way** as a FootwayInfo, and a **building** as a BuildingInfo. Here are the functions that load the XML and store the data in a set of data structures:

```
//
// Functions:
//
bool LoadOpenStreetMap(string filename, XMLDocument& xmldoc);
int  ReadMapNodes(XMLDocument& xmldoc, map<long long, Coordinates>& Nodes);
int  ReadFootways(XMLDocument& xmldoc, vector<FootwayInfo>& Footways);
int  ReadUniversityBuildings(XMLDocument& xmldoc,
       map<long long, Coordinates>& Nodes,
       vector<BuildingInfo>& Buildings);
```

These functions build three data structures: **Nodes**, **Footways**, and **Buildings**. A drawing is provided in the

Appendix on the last page, and here's the C++ declarations:

```
int main()
{
  map<long long, Coordinates>  Nodes;     // maps a Node ID to it's coordinates (lat, lon)
  vector<FootwayInfo>          Footways;  // info about each footway, in no particular order
  vector<BuildingInfo>         Buildings; // info about each building, in no particular order
  XMLDocument                  xmldoc;
```

The nodes are stored in a map since you'll need to do frequent lookups by ID. The footways are stored in a vector because there is no particular order to them; linear searches will be necessary. The buildings are also stored in a vector because it will be searched by partial name and abbreviation (and some buildings have no abbreviation), so an ordering is not clear; linear searches will be necessary.

Getting started

If you are working in Codio, some of the files were already provided in a sub-directory called "part02". Save your work from part01, e.g. in the sub-directory "part01" or locally on your machine. Then open a terminal window and copy the files from part02 up one level (..) to your home directory:

```
cd part02
cp * ..
cd ..
```

Next, browse to the course dropbox, projects, project07-part02-files, and upload the files from the Codio-updates folder: main.cpp, osm.cpp, osm.h. Finally, in order to build the program, you'll need to update the makefile to compile all four C++ files into one program.exe:

```
build:
   rm -f program.exe
   g++ -O2 -std=c++11 -Wall main.cpp dist.cpp osm.cpp tinyxml2.cpp -o program.exe
```

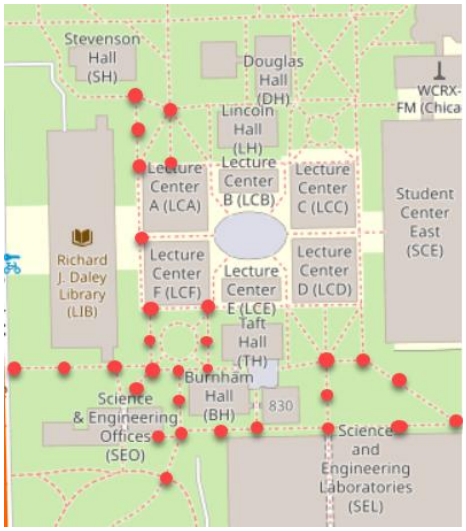
And this point you should be able to build and run the program, load the map, and output some stats about the UIC East campus map:

```
** Navigating UIC open street map **

Enter map filename> map.osm

# of nodes: 18297
# of footways: 382
# of buildings: 34

Enter start (partial name or abbreviation), or #> #
** Done **
```

If you are working outside of Codio, all files necessary for part02 can be found in the course dropbox, projects, project07-part02-files folder.

Assignment details

As discussed earlier, here are the main program steps:



1. *Load map into xmldoc.*
2. *Read nodes.*
3. *Read footways.*
4. *Read buildings.*
5. Add nodes as vertices.
6. Add edges based on footways.
7. Input start and destination buildings, locate on map.
8. Search the footways and find the nearest nodes to the start and destination buildings; these become the "start" and "dest" nodes.
9. Run Dijkstra's algorithm from the start node.
10. Output the distance and path from start node to destination node. If no path exists, output "`Sorry, destination unreachable`".
11. Repeat with another pair of buildings.

A main program is provided in "main.cpp", and the provided code implements steps 1 – 4. Your assignment are steps 5 – 11, which can be done completely in main.cpp. You'll need your implementation of Dijkstra's algorithm from HW #17, which you'll need to modify to compute the predecessors. The topic of predecessors was discussed in class on Friday 4/24 (day 38). For this problem the graph type is now

```
graph<long long, double>  G;  // vertices are nodes, weights are distances
```

Here are more details about each step:

5. **Add nodes as vertices.** Self-explanatory, add each node to the graph.
6. **Add edges based on footways.** A footway is a vector of nodes, defining points along the footway. Let's suppose the footway is {N1, N2, N3, N4}. Then you add edges in both directions between N1-N2, N2-N3, and N3-N4. A footway contains at least 2 nodes.
7. **Input start and destination buildings, locate on map.** Code is provided to do the input, your job is to find the buildings in the Buildings vector. Note that the user can enter multiple words (e.g. "Thomas Beckham" or "Henry Hall"), and the input can denote a full building name, some part of the name, or an abbreviation (e.g. "SEO", "LCA", or "SCE"). Unfortunately, some abbreviations overlap, e.g. "BH" and "TBH", so if you only search for partial matches, you might find "TBH" instead of "BH". The simplest solution is the following:
   a. *Search by abbreviation first*
   b. *If not found, then search the fullname for a partial match (use .find?).*

   If the start building is not found, output "Start building not found", skip steps 8-10, and get another pair of inputs. Likewise if the dest building is not found, output "Destination building not found".
8. **Search the footways for the nearest start and dest nodes.** Assuming the start and destination buildings were found, you have the start and destination coordinates. The problem is that the buildings are *not* on

the footways, so there's no path between buildings. The solution we're going to take is to search through the Footways, and find the nearest footway node to the start building. Likewise search and find the nearest footway node to the destination building. How? Call the **distBetween2Points()** function (provided in "dist.cpp"), and remember the node with the smallest distance; when you call the function, use the building's (lat, lon) as the first parameter. If two nodes have the same distance, use the first one you encounter as you search through the Footways. [This is basically a "find the min" algorithm.]

9. **Run Dijkstra's algorithm.** Don't forget to redefine **double INF** = numeric_limits<**double**>::max();
10. **Output distance and path to destination.** Dijkstra's algorithm returns the distances (as a map), and the predecessors (however you want). If the destination is unreachable (which can happen, e.g. "SEO" to "SSB" is unreachable), output "Sorry, destination unreachable". Otherwise output the distance and path as shown in the screenshots.
11. **Repeat with another pair of buildings.**

```
Enter start (partial name or abbreviation), or #> Thomas Beckham
Enter destination (partial name or abbreviation)> SES
Starting point:
 Thomas Beckham Hall (TBH)
 (41.865794, -87.647374)
Destination point:
 Science Engineering South (SES)
 (41.868949, -87.648478)

Nearest start node:
 1643970101
 (41.865897, -87.647908)
Nearest destination node:
 471537981
 (41.869081, -87.648615)

Navigating with Dijkstra...
Distance to dest: 0.6761249 miles
Path: 1643970101->2899430147->261209364->2899430150->2518698048->2518698060->2518698056->2518698058->2518698
054->2518698055->2518698053->2518698051->4226840021->4226840022->4226840023->4226840063->4226840089->4226840
092->5913782408->4226840111->4226840124->5913782407->4226840141->4226840135->4226840132->4226840131->5632908
808->5632908810->1699207006->471537981
```

```
Enter start (partial name or abbreviation), or #> SEO
Enter destination (partial name or abbreviation)> SSB
Starting point:
 Science & Engineering Offices (SEO)
 (41.870827, -87.650253)
Destination point:
 Student Services Building (SSB)
 (41.874815, -87.658162)

Nearest start node:
 6291902791
 (41.870796, -87.650207)
Nearest destination node:
 2051982643
 (41.87541, -87.657126)

Navigating with Dijkstra...
Sorry, destination unreachable

Enter start (partial name or abbreviation), or #> Piazza
Enter destination (partial name or abbreviation)> Venice Italy
Start building not found
```

```
Enter start (partial name or abbreviation), or #> UH
Enter destination (partial name or abbreviation)> Henry Hall
Starting point:
 University Hall (UH)
 (41.873779, -87.651011)
Destination point:
 Henry Hall
 (41.874091, -87.650555)

Nearest start node:
 464748192
 (41.873812, -87.651134)
Nearest destination node:
 5632839973
 (41.873986, -87.650486)

Navigating with Dijkstra...
Distance to dest: 0.096034424 miles
Path: 464748192->5632839978->5632839988->5632839977->463814003->46
4345369->463814052->464748194->462010750->5632839974->5632839973
```

Requirements

1. You must solve the problem as intended, i.e. build a graph from the map data, and use Dijkstra's algorithm to find the shortest weighted path.
2. The graph class must be the same graph class submitted for part 1. Do not modify the graph class in order to create a custom class for the purpose of solving this particular assignment.
3. Dijkstra's algorithm should be your solution from HW #17, modified as needed for this assignment (i.e. predecessors, different graph type).
4. No global variables.

Grading, electronic submission, and Gradescope

Submission: all .cpp and .h files required to compile your program

Your score on this project is based on two factors: (1) correctness as determined by Gradescope, and (2) manual review of program files for commenting, style, and approach (e.g. adherence to requirements). Since part 2 is longer than part 1, it is worth 100 points for correctness, and 40 points for manual review of commenting, style and overall approach / efficiency. In this class we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness. Example: if you submit to Gradescope and your score is a reported as a 0, then that's your correctness score. The only way to raise your correctness score is to re-submit.

In this project, your program will be allowed **12 free submissions**. After 12 submissions, each additional submission will cost 1 point. Example: suppose you score 100 after 15 submissions, and you activate this submission for grading. Your autograder score will be 97: 100 – 3 extra submissions. Note that you cannot use another student's account to test your work; this is considered academic misconduct because you have given your code to another student for submission on their account.

By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it will be considered academic misconduct.

Policy

Late work *is* accepted. You may submit as late as 1 week after the deadline for a penalty of 10%. After late period has expired, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you

cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .

Appendix: data structures