Project 7 – Openstreet Maps
CS 251, Fall 2021

Collaboration Policy: By submitting this assignment, you are acknowledging you have read the collaboration policy in the syllabus for projects. This project should be done individually. You may not receive any assistance from anyone outside of the CS 251 teaching staff. Also, please see copyright at the bottom of this description. Do not post your work for this class publicly, even after the course is over.

Late Policy: You are given a total of 5 late days to use at your discretion (for all projects, not each project). You may use the late days in 24-hour chunks (either 1 day at a time or all five at once). You do not need to alert the instructor to ask permission to use late days. You can manage your use of late days on Mimir directly.

Early Bonus: If you submit a finished project early (by Wednesday, November 24th at 11:59 pm), you can receive 10% extra credit. In order to receive the early bonus: (1) your submission needs to pass 100% of the tests cases; (2) you may not have any submissions after the early bonus deadline.

Test cases/Submission Limit: Unlimited submissions.

IDE: You will need to use Codio to develop your code, as we have set up all the Linux tools needed (gdb, GoogleTests, valgrind). To find the starter code, just login at codio.com and you will see Project 5. If you haven't set up your account yet, use these instructions.

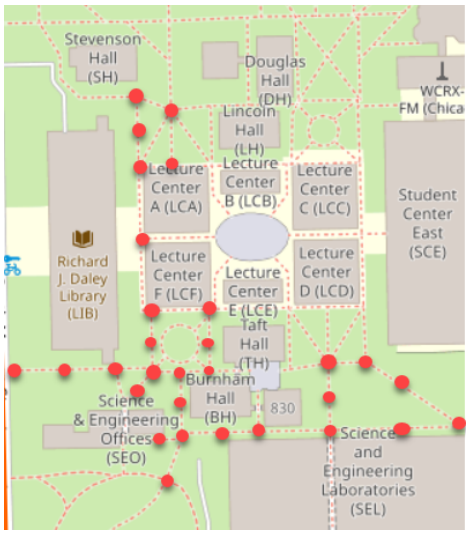
What to submit: (1) graph.h; (2) application.cpp

.pdf starter code

Project Overview

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 above, 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.

Write the Graph Class

We've been working with a **graph** class limited to at most 100 vertices. This limitation is due to the underlying implementation based on an adjacency matrix. The first step of the project is to remove this limitation by rewriting the graph class to use an adjacency list representation. The "list" does not have to be a linked-list, you are free to use whatever data structure you want to represent a "list of edges". And you are free to use any of the built-in C++ data structures: map, set, list, deque, etc.

A current version of the "graph.h" file is available in the starter code. A testing program is also provided that reads a graph from an input file, and then outputs the graph to see if it was built correctly (i.e. "testing.cpp"). Use this as an initial testing platform if you wish. A sample input file is available in "graph.txt" and you can visually see this graph is "graph.pdf". You are encouraged to test the graph using the Google Tests framework (but since it will not be graded, it is not required). Otherwise, you can use the testing.cpp file to get your graph.h implementation functional (see makefile for make builtest, make runtest).

You are going to completely rewrite the class, which means e.g. that you should delete the **EdgeData** structure, the **AdjMatrix** data member, and the **MatrixSize** constant. Your new class will have no size limit. You can even delete and replace the **Vertices** vector, it's up to you. The class must remain templated with VertexT and WeightT, and you must retain all public functions

as currently defined. Your job is to replace how they are implemented, but you must keep all existing public functions. In particular, here are the major steps (and requirements):

1. Delete all aspects of the adjacency matrix.
2. Replace with an implementation based on an adjacency list; see zybooks section 10.13 or lecture slides/video.
3. Delete the constructor graph(int n).
4. Add a default constructor graph().
5. If you decide to dynamically-allocate memory, add a destructor. You'll also need to add a copy constructor and operator= to properly make deep copies.
6. Re-implement all other functions. For **addVertex**, delete the "if matrix is full" check.
7. **No linear search** --- Any instances of linear search will result in failure of all graph.h autograder tests (which will be manually applied during grading). Example: a call to **addVertex(v)** has to check if v exists, and this must be done in O(lgN) worst-case time (where N is the # of vertices). You may assume the graph is sparse, which implies a vertex has a small number E of edges and thus E is significantly less than the total # of edges M. This allows you to use a linked-list for your adjacency list, and it's legal to search this list in "linear" O(E) time since E is very small. What you cannot do is have a single list of *all* the graph edges, since this would require an expensive linear search of O(M) time.
8. Finally, update the **dump()** function to properly output the vertices and edges, based on your final implementation. When you output the edges, output in a readable format such as:

```
A: (A,B,80) (A,C,100), …
B: (B,A,100) (B,F,123), …
```

WARNING on include statements: We have seen in this project and the last that students are submitting files without the proper include statements. This leads to submissions that work differently on the IDE than in the autograder. You must include all approrpriate header files in each file you write. Even if it works in your IDE, that doesn't mean it is right!

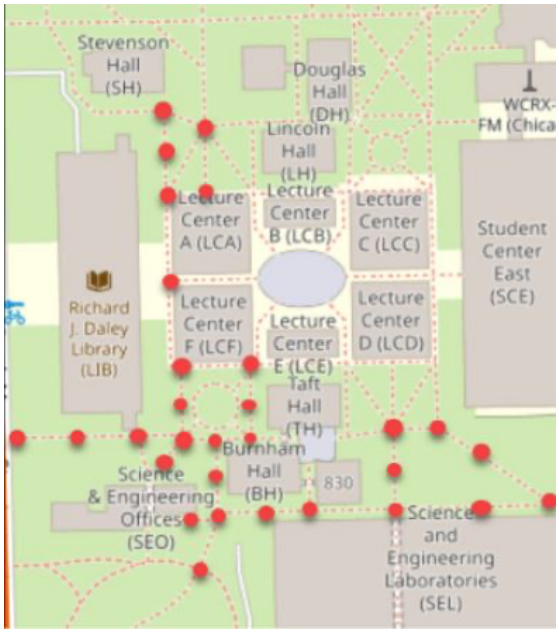
Overview of Application

The application portion of this project is to write a console-based C++ program to input a campus map (e.g. UIC's East campus) and navigate two people at two different buildings to a "good" building to meet 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 person 1's building and person 2's building from the user, find a "good" building for the two people to meet, and then use Dijkstra's algorithm twice to find the shortest weighted path for both people to the destination

building. This is repeated until the user enters # for the person 1's building. Here are the application 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 person 1 and person 2's buildings, locate on map.
8. Find the center of the two buildings, locate the building closest to that center. Output the building name.
9. Search the footways and find the nearest nodes to person 1 and person 2's building, these become the two "start" nodes. Then find the nearest node to the destination building, this becomes the "dest" node.
10. Run Dijkstra's algorithm for the two starting nodes.
11. Output the two paths to the destination node, along with their respective distances. If no path exists from either starting node, **repeat steps 8-11 on the second closest building to the center.**
12. 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".



```
** 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 "a" for the standard application or "c" for the creative component application> a

Enter person 1's building (partial name or abbreviation), or #> TBH
Enter person 2's building (partial name or abbreviation)> SCE

Person 1's point:
 Thomas Beckham Hall (TBH)
 (41.865794, -87.647374)
Person 2's point:
 Student Center East (SCE)
 (41.872051, -87.648041)
Destination Building:
 Science Engineering South (SES)
 (41.868949, -87.648478)

Nearest P1 node:
 1643970101
 (41.865897, -87.647908)
Nearest P2 node:
 1645121521
 (41.871909, -87.648247)
Nearest destination node:
 471537981
 (41.869081, -87.648615)


Person 1's distance to dest: 0.6761249 miles
Path: 1643970101->2899430147->261209364->2899430150->2518698048->2518698060->2518698056->251
840063->4226840089->4226840092->5913782408->4226840111->4226840124->5913782407->4226840141->

Person 2's distance to dest: 0.20880144 miles
Path: 1645121521->464345546->462010768->462010748->462014176->1647971957->1647971942->422684
```

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;
};
```

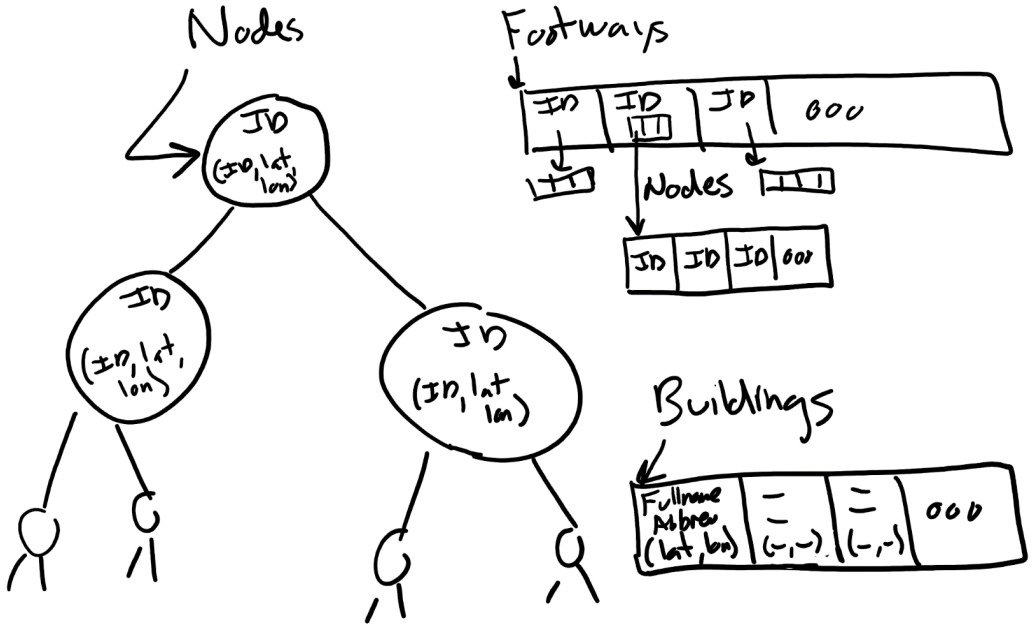
A **node** in the map is stored as a Coordinate, a **way** as a FootwayInfo, and a **building** as a BuildingInfo. Here are the functions that load the XML and store the data in a set of data structures:

```
//
// 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 below, 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.



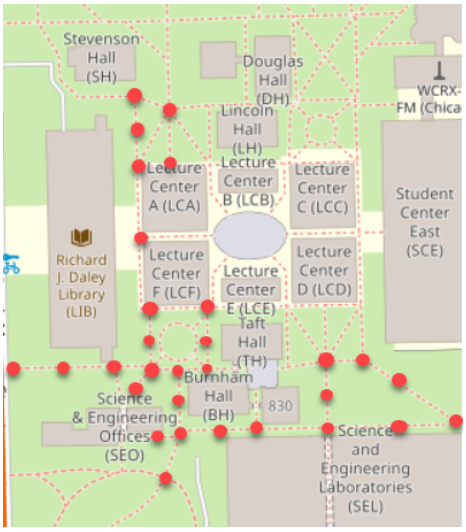
Getting Started

Using the makefile, you can now move to running the application portion of this project. To do so, you will type make build and make run. At this point you should be able 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 "a" for the standard application or "c" for the creative component application> a

Enter person 1's building (partial name or abbreviation), or #> #
** Done **
```

Assignment Details

As discussed earlier, here are the application 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 person 1 and person 2's buildings, locate on map.
8. Find the center of the two buildings, locate the building closest to that center. Output the building name.
9. Search the footways and find the nearest nodes to person 1 and person 2's building, these become the two "start" nodes. Then find the nearest node to the destination building, this becomes the "dest" node.
10. Run Dijkstra's algorithm for the two starting nodes.
11. Output the two paths to the destination node, along with their respective distances. If no path exists from either starting node, **repeat steps 8-11 on the next closest building to the center.**
12. Repeat with another pair of buildings.

An application program is provided in “application.cpp”, and the provided code implements steps 1 – 4. Your job is to do steps 5 – 12, which can be done completely in application.cpp. You’ll need your implementation of Dijkstra’s algorithm including computing the predecessors. The topic of predecessors was discussed during Week 13 lecture. 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. Call the **distBetween2Points()** function (provided in “dist.cpp”) for the coordinates of each pair of nodes to determine the edge weight. You can assume 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:

1.
    1. *Search by abbreviation first*
    2. *If not found, then search the fullname for a partial match (use .find?).*

If person 1's building is not found, output “Person 1's building not found”, skip steps 8-11, and get another pair of inputs. Likewise if person 2's building is not found, output “Person 2's building not found”.

8. Find the center of the two buildings, locate the building closest to that center. Assuming person 1 and person 2's buildings were found, you have the coordinates of two buildings. These two points will form a line, and your job is to search the Buildings vector for the building with coordinates closest to the center of that line. Once you find the destination building, output it. Note that this step may need to be repeated with the next closest building if there happens to not be a path from either starting node.

To find the center of two coordinates, you can utilize the **centerBetween2Points()** function (provided in "dist.cpp") for the coordinates of the two buildings. Note that this function returns a Coordinates struct.

9. Search the footways for the nearest start and dest nodes. You now have the coordinates of person 1's building, person 2's building, and their destination building. 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 person 1's building. Likewise search and find the nearest footway node to the person 2's building and 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.]

10. Run Dijkstra's algorithm from each starting node. Don't forget to redefine **double INF** = numeric_limits<**double**>::max();

11. Output the two paths and respective distances to the destination. Dijkstra's algorithm returns the distances (as a map), and the predecessors (however you want). If there is no path from person 1's building to person 2's building (which can happen, e.g. "SEO" to "SSB" is unreachable), then there is no building they can meet at. In this case, output "Sorry, destination unreachable." If the destination is unreachable from either starting node , repeat steps 8-11 with the second closest building to the center, if the second closest building is also unreachable, continue with the third closest, etc. Output the new destination building and new destination node for each next closest building. Otherwise output the distance and path as shown in the screenshots.

```
Enter person 1's building (partial name or abbreviation), or #> Davenport
Enter person 2's building (partial name or abbreviation)> Wohlers

Person 1's point:
 Davenport Hall
 (40.107214, -88.226266)
Person 2's point:
 Wohlers Hall
 (40.10361, -88.229854)
Destination Building:
 Gregory Hall
 (40.10569, -88.228109)

Nearest P1 node:
 1508108935
 (40.10742, -88.22628)
Nearest P2 node:
 5902424424
 (40.103537, -88.229757)
Nearest destination node:
 5421642570
 (40.105784, -88.228205)

?t least one person was unable to reach the destination building. Finding next closest building...

New destination building:
 Smith Memorial Hall
 (40.105723, -88.226171)
Nearest destination node:
 6055331272
 (40.105754, -88.226535)

Person 1's distance to dest: 0.18438685 miles
Path: 1508108935->6025219804->1508108770->1508108793->6025211781->1508108592->6025211782->6025211783->1508108879->150
7->1508108706->5184308442->8494309390->1507899474->5947620710->6055331267->5409108889->5409108888->6055331270->605533

Person 2's distance to dest: 0.28732376 miles
Path: 5902424424->5902424423->5397345765->2175868870->6114371055->5397392395->2175868863->2175868867->2175878093->246
5->1997964662->5727723966->1997964673->5727723967->1507899874->626693481->5407919674->1507899876->1507900053->5184308
699377795->5184308414->6055331270->6055331271->6055331272

Enter person 1's building (partial name or abbreviation), or #>
```

```
Enter person 1's building (partial name or abbreviation), or #> Levis
Enter person 2's building (partial name or abbreviation)> Mumford

Person 1's point:
 Levis Faculty Center
 (40.108647, -88.221128)
Person 2's point:
 Mumford Hall
 (40.103698, -88.225965)
Destination Building:
 Evans Hall
 (40.105608, -88.223324)

Nearest P1 node:
 5727744969
 (40.108493, -88.221478)
Nearest P2 node:
 6058881855
 (40.1037, -88.225926)
Nearest destination node:
 5438365833
 (40.105641, -88.223345)

At least one person was unable to reach the destination building. Finding next closest building...

New destination building:
 Busey Hall
 (40.105604, -88.222671)
Nearest destination node:
 6050226424
 (40.105601, -88.222702)

At least one person was unable to reach the destination building. Finding next closest building...


New destination building:
 Freer Hall
 (40.104974, -88.222755)
Nearest destination node:
 5411139442
 (40.104856, -88.222583)

Person 1's distance to dest: 0.5578211 miles
Path: 5727744969->5418851689->5424856594->5727744966->5415129117->5727744964->5727744963->5411155453->5411155454->
7->6933320510->5423334848->5727934075->5404433858->6050226622->5727735601->1997964684->5398858554->1997964675->199
1512373207->6061173281->1997964659->5411139441->5411139460->5411139453->5411139442

Person 2's distance to dest: 0.2995131 miles
Path: 6058881855->5423647614->6058881860->6058881858->5423647613->5727944590->626693578->1510730895->5381124178->5
->2176657630->626693566->5830737954->5407778529->5412533343->5412533360->1512372816->6058399657->5412533326->54125
61173269->6061173270->6061173271->6061173272->5398858521->5727880448->5398856418->5398856419->6061173266->60611732
73207->6061173281->1997964659->5411139441->5411139460->5411139453->5411139442

Enter person 1's building (partial name or abbreviation), or #>
```

12. Repeat with another pair of buildings.

Here's some sample output.

```
Enter person 1's building (partial name or abbreviation), or #> TBH
Enter person 2's building (partial name or abbreviation)> SCE

Person 1's point:
 Thomas Beckham Hall (TBH)
 (41.865794, -87.647374)
Person 2's point:
 Student Center East (SCE)
 (41.872051, -87.648041)
Destination Building:
 Science Engineering South (SES)
 (41.868949, -87.648478)

Nearest P1 node:
 1643970101
 (41.865897, -87.647908)
Nearest P2 node:
 1645121521
 (41.871909, -87.648247)
Nearest destination node:
 471537981
 (41.869081, -87.648615)

Person 1's distance to dest: 0.6761249 miles
Path: 1643970101->2899430147->261209364->2899430150->2518698048->2518698060->2518698056->2518698058->251869
->4226840021->4226840022->4226840023->4226840063->4226840089->4226840092->5913782408->4226840111->422684012
226840132->4226840131->5632908808->5632908810->1699207006->471537981

Person 2's distance to dest: 0.20880144 miles
Path: 1645121521->464345546->462010768->462010748->462014176->1647971957->1647971942->4226840184->177066345

Enter person 1's building (partial name or abbreviation), or #>
```

```
Enter person 1's building (partial name or abbreviation), or #> SEO
Enter person 2's building (partial name or abbreviation)> SSB

Person 1's point:
 Science & Engineering Offices (SEO)
 (41.870827, -87.650253)
Person 2's point:
 Student Services Building (SSB)
 (41.874815, -87.658162)
Destination Building:
 Behavioral Sciences Building (BSB)
 (41.87375, -87.652899)

Nearest P1 node:
 6291902791
 (41.870796, -87.650207)
Nearest P2 node:
 2051982643
 (41.87541, -87.657126)
Nearest destination node:
 5632839991
 (41.874027, -87.652499)

Sorry, destination unreachable.

Enter person 1's building (partial name or abbreviation), or #>
```

```
Enter person 1's building (partial name or abbreviation), or #> Piazza
Enter person 2's building (partial name or abbreviation)> TBH
Person 1's building not found

Enter person 1's building (partial name or abbreviation), or #> TBH
Enter person 2's building (partial name or abbreviation)> Piazza
Person 2's building not found

Enter person 1's building (partial name or abbreviation), or #>
```

Creative Component

After you finish the application, now it's your turn to make something interesting! The creative() function just needs parameters added and you can implement your own extensions/ideas. Here are some ideas, but don't feel limited to just these:

- Maybe person 1 has had a long day and does not feel like walking as much as person 2 does. Implement a weighted "best place to meet" for the original navigation app.
- Set up a navigation application which takes a list of people and a potential meeting building, then checks to see if each person can reach the destination.
- Find a more sophisticated algorithm for determining a "good meeting building". Perhaps use Dijkstra's results to compare pairs of distances to different buildings?
- Take a schedule of classes, their location (and time?), and calculate the path the student should take from building to building.

Requirements for your creative component:

- You must have a comment at the top of application.cpp that describes what your creative component does and how to run it. If this is missing, you will get no credit and no regrade requests are accepted. The TAs grading must be able to run your creative component to test it. So all the information must be included that allow them to do this. If other commands must be called first in order for your command to work, make sure to include that information.
- It must be a similar difficulty and amount of work/code as would be appropriate for 10 points of the project.

Requirements

1. You must have a clean valgrind report when your code is run. Check out the makefile to run valgrind on your code. All memory allocated (call new) must be freed (call delete). The test cases run valgrind on your code.
2. No global or static variables.
3. You may not change the API provided for graph.h. You must keep all public member functions and variables as they are given to you. If you make changes to these, your code will not pass the test cases. You may add private member variables and private member helper functions. You may not add public member variables or public member functions.
4. Code must be written efficiently. Complexity should be minimized. Deductions to final submission will be applied for solutions that are inefficient, in space or in time.

5. 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.
6. Dijkstra's algorithm should be applied as described in zybooks, lecture, and lab. No other versions of the algorithm are allowed.

Citations/Resources
Joe Hummel, University of Illinois at Chicago and Kai Bonsol.