

Project 5 – Open Street Maps

CS 251, Fall 2023

Collaboration Policy

By submitting this assignment, you are acknowledging you have read the course collaboration policy. This project should be done individually. You may not receive any assistance outside of the CS 251 instructional team. **Do not post your work for this class publicly, now or after class.**

Late Policy

You are given a grace period of 24-hours. You do not need to let anyone know you are using this grace period, you simply continue to work and submit during the time period. Beyond this period, no late submissions will be accepted. Please note that if you are continually using the grace period, then you are off track from our anticipated pace that facilitates spacing of assessments.

What & Where to Submit

graph.h and application.cpp **to both zyBooks & Gradescope**

Submit to both Gradescope & zyBooks. Gradescope will be used for autograding your graph class implementation and to manually grade valgrind, style, and verify use of adjacency list (0 points on assignment if you do not use an adjacency list). zyBooks verifies implementation and execution output against the solution (the application portion of the program).

Table of Contents

[Project Overview](#)

[Write the Graph Class](#)

[Overview of Application](#)

[Getting Started](#)

[5. Add nodes as vertices.](#)

[6. Add edges based on footways.](#)

[7. Input start and destination buildings, locate them on map.](#)

[8. Find the center of the two buildings, locate the building closest to that center.](#)

[9. Search the footways for the nearest start and dest nodes.](#)

[10. Run Dijkstra's algorithm from each starting node.](#)

[11. Output the two paths and respective distances to the destination.](#)

[Abbreviated Requirements & Reminders](#)

Copyright Notice

Copyright 2023 Adam T Koehler, PhD - University of Illinois Chicago

This assignment description is protected by [U.S. copyright law](#). Reproduction and distribution of this work, including posting or sharing through any medium, such as to websites like [chegg.com](#) is explicitly prohibited by law and also violates [UIC's Student Disciplinary Policy](#) (A2-c. Unauthorized Collaboration; and A2-e3. Participation in Academically Dishonest Activities: Material Distribution).

Material posted on [any third party](#) sites in violation of this copyright and the website terms will be removed. Your user information will be released to the author.

Citations/Resources

Shanon Reckinger, PhD - University of Illinois Chicago; Joe Hummel, PhD - University of Illinois Chicago; Kai Bonsol.

Project 5 – Open Street Maps

CS 251, Fall 2023

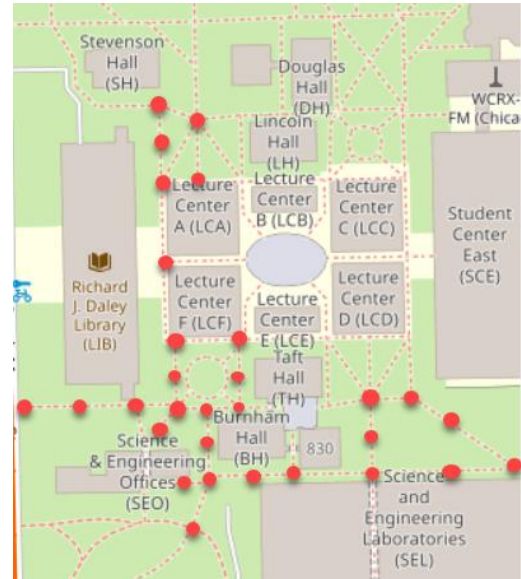
Project Primary Topics

- Graphs
- Dijkstra's Algorithm
- Priority Queue
- Dynamic Memory

Project Overview [\(Starter Code\)](#)

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

The current 100 vertex limitation is due to the underlying implementation of an adjacency matrix. The first step of the project is to remove the 100 vertex limitation by rewriting the graph class to use an adjacency list representation instead of an adjacency matrix (**graph.h**). 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.

Project 5 – Open Street Maps

CS 251, Fall 2023

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 Catch or Google Tests framework (but since testing 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 buildtest*, *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 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 since we have no size limitation anymore.
7. **No linear search** --- Any instances of linear search will result in failure of all graph.h tests. Example: a call to **addVertex(v)** has to check if v exists, and this must be done in $O(\lg N)$ 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 previous projects that students are submitting files without the proper include statements. This leads to submissions that work differently on different IDEs (some may not compile). You must include all appropriate header files in each file you write. Even if it works in your IDE, that doesn't mean it is right!

Project 5 – Open Street Maps

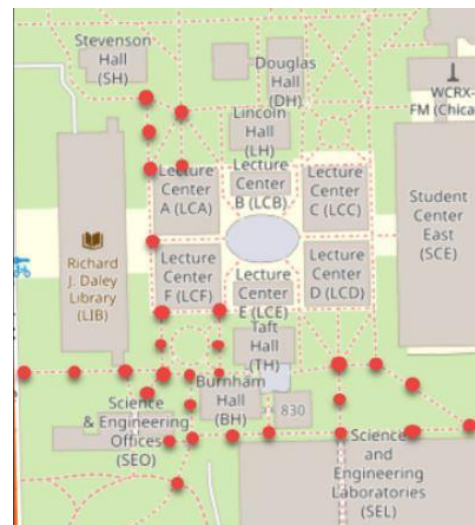
CS 251, Fall 2023

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 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 as 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".



Project 5 – Open Street Maps

CS 251, Fall 2023

```
** 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 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->2518698
060->2518698056->2518698058->2518698054->2518698055->2518698053->2518698
051->4226840021->4226840022->4226840023->4226840063->4226840089->4226840
092->5913782408->4226840111->4226840124->5913782407->4226840141->4226840
135->4226840132->4226840131->5632908808->5632908810->1699207006->4715379
81
```

```
Person 2's distance to dest: 0.20880144 miles
```

```
Path: 1645121521->464345546->462010768->462010748->462014176->1647971957
->1647971942->4226840184->1770663458->1308175623->4226840176->471537981
```

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


Project 5 – Open Street Maps

CS 251, Fall 2023

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 & 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.

Project 5 – Open Street Maps

CS 251, Fall 2023

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;
};
```

Project 5 – Open Street Maps

CS 251, Fall 2023

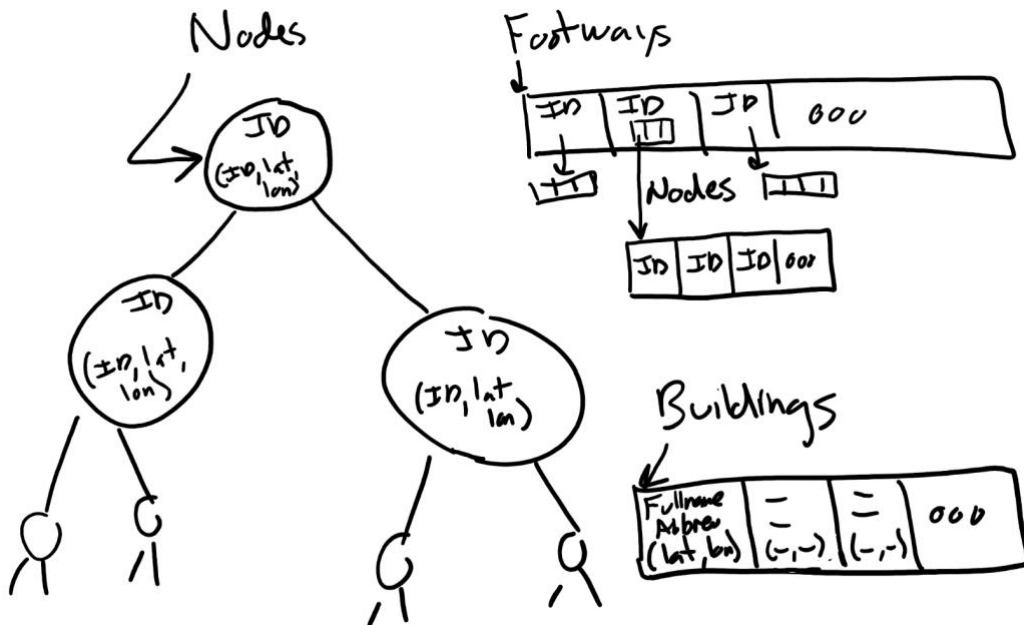
A **node** in the map is stored as a `Coordinate`, a **way** as a `FootwayInfo`, and a **building** as a `BuildingInfo`. Here are the functions that load the XML and store the data in a set of data structures:

```
//  
// 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.



Project 5 – Open Street Maps

CS 251, Fall 2023

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

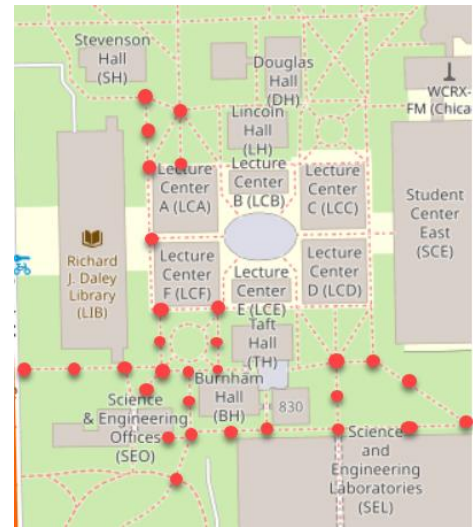
# of nodes: 18297
# of footways: 382
# of buildings: 34

Enter 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 in the graphs lectures. For this problem the graph type is now

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

Project 5 – Open Street Maps

CS 251, Fall 2023

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 them 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. Search by abbreviation first
2. If not found, then search the full name 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.]

Project 5 – Open Street Maps

CS 251, Fall 2023

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, 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.

The following examples use uic.osm

```
Enter person 1's building (partial name or abbreviation), or #> LCA
Enter person 2's building (partial name or abbreviation)> Lecture Center B

Person 1's point:
  Lecture Center A (LCA)
  (41.872116, -87.649889)
Person 2's point:
  Lecture Center B (LCB)
  (41.872207, -87.649302)
Destination Building:
  Lecture Center A (LCA)
  (41.872116, -87.649889)

Nearest P1 node:
  462010739
  (41.87228, -87.649841)
Nearest P2 node:
  465776439
  (41.872125, -87.649536)
Nearest destination node:
  462010739
  (41.87228, -87.649841)

Person 1's distance to dest: 0 miles
Path: 462010739

Person 2's distance to dest: 0.026315778 miles
Path: 465776439->462010738->462010739

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

Project 5 – Open Street Maps

CS 251, Fall 2023

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

```
Person 1's point:
```

```
Science Engineering South (SES)
(41.868949, -87.648478)
```

```
Person 2's point:
```

```
Behavioral Sciences Building (BSB)
(41.87375, -87.652899)
```

```
Destination Building:
```

```
Richard J. Daley Library (LIB)
(41.871873, -87.650448)
```

```
Nearest P1 node:
```

```
471537981
(41.869081, -87.648615)
```

```
Nearest P2 node:
```

```
5632839991
(41.874027, -87.652499)
```

```
Nearest destination node:
```

```
464345492
(41.871881, -87.650075)
```

```
Sorry, destination unreachable.
```

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

Project 5 – Open Street Maps

CS 251, Fall 2023

12. Repeat with another pair of buildings.

Here's some sample outputs (again using uic.osm).

```
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->251869806
0->2518698056->2518698058->2518698054->2518698055->2518698053->2518698051-
>4226840021->4226840022->4226840023->4226840063->4226840089->4226840092->5
913782408->4226840111->4226840124->5913782407->4226840141->4226840135->422
6840132->4226840131->5632908808->5632908810->1699207006->471537981

Person 2's distance to dest: 0.20880144 miles
Path: 1645121521->464345546->462010768->462010748->462014176->1647971957->
1647971942->4226840184->1770663458->1308175623->4226840176->471537981

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 #> █
```

Project 5 – Open Street Maps

CS 251, Fall 2023

```
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 #> █
```

Abbreviated Requirements & Reminders

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. **There is a significant penalty for not having a clean valgrind report!**
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. Either manually or simply by your implementation failing to finish a test case in a timely manner.
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.