

Project #07 (v1.2)

Assignment: OSM navigation using Dijkstra's algorithm
Submission: via Gradescope (unlimited submissions)
Policy: individual work only, late work is accepted
Complete By: Friday March 03 @ 11:59pm CST for full credit

Late submissions: 02% penalty for each 12-hour period past the due date (max penalty 08%)
NO submissions accepted 48 hours after the due date

Pre-requisites: HW 09

Overview

Here in project 07 we're going to continue working with Open Street Maps, and finally put them into action. In particular, we'll build a graph of the NU campus from the nodes and footways, and use Dijkstra's algorithm to navigate between buildings.

Recall that a **node** is a point on the map, consisting of 3 values: id, latitude, and longitude. Nodes are shown as red dots in the screenshot, and become the **vertices** in our graph. The **footways**, shown as the dashed lines, become the edges in our graph. **Example:** if footway F consists of nodes N1, N2, N3 and N4, then we add the following edges: N1 -> N2, N2 -> N3, N3 -> N4, and back in the other direction N4 -> N3, N3 -> N2, and N2 -> N1. The weight on each edge is the distance (in miles) from the START vertex to the END vertex.



Getting Started

We will continue working on the EECS computers. The first step is to setup your project07 folder and copy the necessary files:

1. If you are working on a Mac or Linux, open a terminal. If you are working on Windows, open Git Bash.
2. Login to moore: `ssh moore` or `ssh YOUR_NETID@moore.wot.eecs.northwestern.edu`

- | | |
|--|--|
| 3. Make a directory for project 07 | mkdir project07 |
| 4. Make this directory private | chmod 700 project07 |
| 5. Move ("change") into this directory | cd project07 |
| 6. Copy the files needed for project 07: | |
| a. To use your solution: | cp -r ../project06/release . |
| b. To use our solution: | cp -r /home/cs211/w2023/project07/release . |
| 7. List the contents of the directory | ls |
| 8. Move ("change") into release dir | cd release |
| 9. List the contents of the directory | ls |
| 10. Build the program | make build |
| 11. Run the program | ./a.out |

IF YOU ARE WORKING FROM YOUR SOLUTION, please complete the following additional steps:

- | | |
|---|---|
| 12. Copy graph files | cp /home/cs211/w2023/project07/release/graph.* . |
| 13. Copy distance files | cp /home/cs211/w2023/project07/release/dist.* . |
| 14. Edit your makefile | micro makefile |
| a. <u>Build</u> : add graph.o and dist.cpp to the list of files compiled with g++ | |
| b. <u>Submit</u> : change submission command to project07 | |
| 15. Build the program | make build |
| 16. Run the program | ./a.out |

AT THIS POINT you should have a working solution to project 06:

```
hummel@moore$ ./a.out
** NU open street map **

Enter map filename>
nu.osm
# of nodes: 15070
# of buildings: 103
# of footways: 686

Enter building name (partial or complete), or * to list, or $ to end>
$

** Done **
# of calls to getID(): 0
# of Nodes created: 15070
# of Nodes copied: 16383
```

```
Enter building name (partial or complete), or * to list, or $ to end>
Mudd
Seeley G. Mudd Science and Engineering Library
Address: 2233 Tech Drive
Building ID: 42703541
Nodes:
533996670: (42.0586, -87.6747)
533996671: (42.0585, -87.6741)
533996672: (42.0583, -87.6741)
533996673: (42.0582, -87.6739)
533996674: (42.0581, -87.6738)
4838815124: (42.0581, -87.6737)
9119071427: (42.058, -87.6738)
9119071426: (42.0579, -87.6738)
2240260053: (42.0579, -87.6738)
2240260054: (42.0579, -87.6739)
533996668: (42.0579, -87.6739)
533996675: (42.0579, -87.6741)
533996669: (42.0579, -87.6747)
533996670: (42.0586, -87.6747)
Footways that intersect:
Footway 376278372
Footway 986532630
```

Part 01: Building the Graph

As noted earlier, the graph is built from the nodes and footways. The nodes become the vertices, in particular each node id is added to the graph as a vertex. Since there are 15,070 nodes, this means the graph will contain 15,070 vertices.

The footways are the pathways through the map, and we'll add edges based on the footways. Recall that a footway consists of a set of nodes --- we'll add edges between each node so that we can navigate the graph as if we were walking along the footway. Footways have nodes in common as they intersect with one another, allowing navigation across campus (assuming you stay on the pathway :-). You should have 5,422 edges when the graph is correctly built.



As a debugging check, we're going to add the "!" command to the main() program to output the edges associated with footway ID 986532630, which in the OSM file contains 4 nodes:

```
<way id="986532630" visible="true" version="1" changeset="111627186"
  <nd ref="9119071425"/>
  <nd ref="533996671"/>
  <nd ref="533996672"/>
  <nd ref="2240260064"/>
  <tag k="highway" v="footway"/>
</way>
```

Here's how your program should behave once part 01 is completed (note that the user prompt has changed in main() to prepare for project 07, "Enter building name, * to list, @ to navigate, or \$ to end>"):

```
** NU open street map **
Enter map filename>
nu.osm
# of nodes: 15070
# of buildings: 103
# of footways: 686
# of graph vertices: 15070
# of graph edges: 5422
Enter building name, * to list, @ to navigate, or $ to end>
!
Graph check of Footway ID 986532630
Edge: (9119071425, 533996671, 0.00189528164215)
Edge: (533996671, 9119071425, 0.00189528164215)
Edge: (533996671, 533996672, 0.0147792802238)
Edge: (533996672, 533996671, 0.0147795161921)
Edge: (533996672, 2240260064, 0.019349796194)
Edge: (2240260064, 533996672, 0.019349796194)
Enter building name, * to list, @ to navigate, or $ to end>
```

Notice the output of the edges matches the XML node references, with edges in both directions. Since it's week nine of the quarter, we are confident you no longer need step-by-step directions on how to complete

part 01. You can reason this out based on the preceding discussion and screenshot. Think about the steps you need, and in what order these steps need to happen. And complete them ONE STEP AT A TIME, don't try to solve part 01 all at once. Some guidelines / hints:

- a. We are using the same graph class that was used in HW 09. See "graph.h" for details. However, since the vertices are node ids, the graph is defined to have vertices of type **long long**, with edge weights of type **double**.
- b. In main(), you need to change the # of digits that are output by default. To do this, first `#include <iomanip>` at the top of "main.cpp", and then at the beginning of main() do **`cout << setprecision(12);`**
- c. To build the graph, you'll need to traverse the nodes and the footways. DO NOT write this code in the main() function; write a helper function in either "main.cpp" or another .cpp file to build the graph. Define the graph variable G in main(), but build it in a helper function.
- d. Recall the Nodes are defined in a map; you can use a "foreach" loop to traverse the map container. However, the map is currently declared as private, so you cannot access the map from your helper function. You can take any of the following options:
 1. Make the map public. This is bad design, but allowed.
 2. Add a public method to the class, and pass the graph to this method. Since this method is part of the class, it can foreach through the map and add the node ids as vertices to the graph. Much better approach.
 3. To get a better feel for how C++ works, what you can do is add the necessary support to the Nodes class that allows outsiders to iterate through the map, while leaving the map private. In short, the "foreach" loop requires access to the begin() and end() of the container. Add the following method declarations to "nodes.h":

```
//  
// allow foreach through the map:  
//  
std::map<long long, Node>::iterator begin();  
std::map<long long, Node>::iterator end();
```

Now implement these methods in "nodes.cpp":

```
//  
// allow foreach through the map:  
//  
std::map<long long, Node>::iterator Nodes::begin()  
{  
    return this->MapNodes.begin();  
}  
  
std::map<long long, Node>::iterator Nodes::end()  
{  
    return this->MapNodes.end();  
}
```

Now the helper function can iterate through the nodes using **for** (auto& pair : nodes) ...

- e. To add the edges you have to traverse the footways. For each footway, you traverse the node ids and add edges in both directions. If the footway contains 4 nodes N1, N2, N3 and N4, you add edges N1 -> N2, N2 -> N3, N3 -> N4, and back in the other direction N4 -> N3, N3 -> N2, and N2 -> N1. The weight on each edge is the distance (in miles) from the START vertex to the END vertex, e.g. the distance from N1 to N2. Note that a footway is guaranteed to have at least 2 nodes.

To compute the distance between two points, a function is provided. See “dist.h”:

```
//  
// DistBetween2Points  
//  
// Returns the distance in miles between 2 points (lat1, long1) and  
// (lat2, long2). Latitudes are positive above the equator and  
// negative below; longitudes are positive heading east of Greenwich  
// and negative heading west. Example: Chicago is (41.88, -87.63).  
//  
// NOTE: you may get slightly different results depending on which  
// (lat, long) pair is passed as the first parameter.  
//  
double distBetween2Points(double lat1, double long1,  
                           double lat2, double long2);
```

To compute the distance from vertex V to vertex W, pass V’s coordinates as lat1 and long1, and W’s coordinates as lat2 and long2. As noted in the comments, it’s possible to get a slightly different result if you pass the coordinates in a different order, so be consistent: when computing the weight for an edge, the START vertex should always be (lat1, long1) and the END vertex should always be (lat2, long2).

- f. The code to implement the “!” command should also be a helper function; DO NOT write this code in main() --- call the helper function. Note that the “!” command is what we call a “sanity check”; we need some assurance we have built the graph correctly before continuing.

Part 02: Building locations

Suppose we are trying to navigate from building S to building D. In a perfect world, we would first take S’s entrance information into account; in the presence of multiple entrances we might select the one that is closest to D. If entrance information is not available, then we could consider the footways that intersect with S, and in the case of multiple intersections, select the one that is closest to D. If neither entrance or intersection information is available, then we might select S’s perimeter node that is closest to D. It’s a non-trivial computation to decide where to start, and likewise where to end.

A real navigation app would need to take the above into account. Since time is limited for this project, we’re going to take a simpler approach. To navigate from S to D, we’ll compute the center of S and D, find the footway nodes FS and FD that are closest to these centers, and navigate from FS to FD. This has obvious flaws (e.g. we might start from the wrong side of the building), but it’s a reasonable algorithm to start with.

The first step is to compute the location of each building. Recall that each building contains a vector of

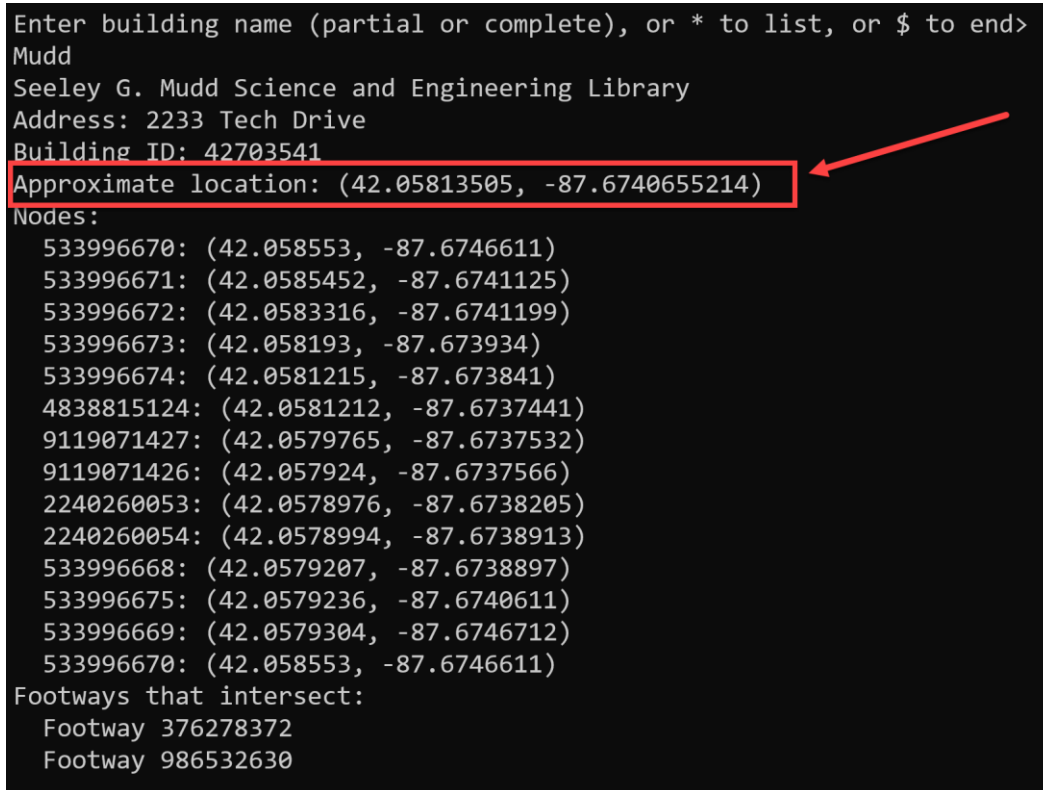
node ids, which represent the nodes that form the perimeter of the building. We'll define the location of a building as the average latitude and longitude of the building's perimeter nodes:

1. Add a method **getLocation()** to your Building class. This method will need to be passed the nodes (much like Building::print()) so it can lookup each node id and retrieve the node's latitude and longitude. To do the lookup, use the Nodes::find() function --- do not write your own search loop. The method must return (not output) the average latitude and longitude. One option is to return via two reference parameters. Another option is to have the function return a pair<double, double> that contains the values. To return a pair from a function, you'll need to #include <utility> in "building.h", and then code the function like this:

```
double avgLat, avgLon;  
.  
.  
.  
return make_pair(avgLat, avgLon);
```

You may assume that a building has at least one node id; use an assert(...) in your code to check the vector size if you want to confirm this.

2. Modify Building::print() to call getLocation() and output the building's approximate location. The screenshot below shows an example.



```
Enter building name (partial or complete), or * to list, or $ to end>  
Mudd  
Seeley G. Mudd Science and Engineering Library  
Address: 2233 Tech Drive  
Building ID: 42703541  
Approximate location: (42.05813505, -87.6740655214)  
Nodes:  
533996670: (42.058553, -87.6746611)  
533996671: (42.0585452, -87.6741125)  
533996672: (42.0583316, -87.6741199)  
533996673: (42.058193, -87.673934)  
533996674: (42.0581215, -87.673841)  
4838815124: (42.0581212, -87.6737441)  
9119071427: (42.0579765, -87.6737532)  
9119071426: (42.057924, -87.6737566)  
2240260053: (42.0578976, -87.6738205)  
2240260054: (42.0578994, -87.6738913)  
533996668: (42.0579207, -87.6738897)  
533996675: (42.0579236, -87.6740611)  
533996669: (42.0579304, -87.6746712)  
533996670: (42.058553, -87.6746611)  
Footways that intersect:  
Footway 376278372  
Footway 986532630
```


Part 03: Start and Destination buildings

Let's figure out which buildings the user wants to navigate between. Add a helper function in "main.cpp" called **navigate()**. For now it's a void function with no parameters. Modify **main()** so that when the user enters the "@" command, you call this function.

Focus on the **navigate()** function. The first step is to see from which building the user wants to start. Input the partial or full name of a building and search for this building --- much like we are already doing in the method **Buildings::findAndPrint()**. If your function needs data from **main()**, pass this data as a parameter... If no building is found, output "***Start building not found" and return from the function. Otherwise, assume the start building is the first building found, and output the building's name and approximate location. Repeat for the destination building... If no building is found, output "***Destination building not found", otherwise select the first building found as the destination, and output the building's name and approximate location.

Example:

```
Enter building name (partial or complete), or * to list, or $ to end>
@
Enter start building name (partial or complete)>
Mudd
  Name: Seeley G. Mudd Science and Engineering Library
  Approximate location: (42.05813505, -87.6740655214)
Enter destination building name (partial or complete)>
Swift
  Name: Annie May Swift Hall
  Approximate location: (42.0523951091, -87.6750774727)
```

Hint: how to keep track of the building when you find it? Create an empty Building object --- e.g. named **startB** --- and construct this building with invalid values such as 0 and "". Then if you find the start building, copy the building B that you found into **startB**:

```
startB = B;
```

After the search loop, you can check **startB**'s ID and see if it's 0 or an actual building ID. Here's an example where the start building is not found:

```
Enter building name (partial or complete), or * to list, or $ to end>
@
Enter start building name (partial or complete)>
Fred
***Start building not found
```

Okay, at this point we have the code in place to find the start and destination buildings. As discussed earlier, the plan is to start our navigation from the footway node FS that is closest to the Start building. In the

navigate() function, after you output the start building's name and location, search the footways for the nearest Footway node. How? Call the **distBetween2Points()** function (see "dist.h") and keep track of the footway node id with the smallest distance; when you call the function, use the building's (lat, lon) as the parameters **lat1** and **long1**. If two nodes have the same distance, keep the first one you encounter as your minimum --- in other words, only update your "min node" if you find one with a distance that is strictly smaller. After you find the minimum, output the footway ID, node ID, and the distance. Feel free to initialize your "min" starting distance to **INF**, which is properly defined in C++ as

```
constexpr double INF = numeric_limits<double>::max();
```

You'll need to `#include <limits>`. Repeat for the destination node (don't copy-paste the code, write a helper function that finds the minimum and call it twice, once for the start building and again for the destination building). **Example:**

```
Enter building name, * to list, @ to navigate, or $ to end>
@
Enter start building name (partial or complete)>
Mudd
  Name: Seeley G. Mudd Science and Engineering Library
  Approximate location: (42.05813505, -87.6740655214)
  Closest footway ID 376278372, node ID 2240260064, distance 0.0102273965997
Enter destination building name (partial or complete)>
University Library
  Name: Northwestern University Library
  Approximate location: (42.0530985325, -87.6742155567)
  Closest footway ID 490417332, node ID 4826014902, distance 0.00455449913142
```

BTW, it's okay to have nested loops inside your helper functions here in project 07, just not inside `main()`.


Part 04: Dijkstra's algorithm to find Shortest Weighted Path

Everything is now in place to run Dijkstra's algorithm and find the shortest path from FS to FD, where FS is the footway node closest to the start building and FD is the footway node closest to the destination building.

If you completed HW 09, you have an implementation of Dijkstra's algorithm, which you can copy-paste here into "main.cpp" or a separate .cpp file. You'll need to change the algorithm code from using string-based vertices to those of type `long long`. The one difference is that you need to incorporate the concept of Dijkstra's "predecessors array" so that the path from FS to FD can be output. But let's start with the version from HW 09 (and class Thursday Feb 23, lecture notes [here](#)) that implements the algorithm without the predecessors array. In HW 09, Dijkstra's algorithm computes two values: a vector of "visited" vertices, and the distance to each vertex from the starting vertex. Modify your **navigate()** function to call your **dijkstra()** function, and upon return output (1) the size of the visited vector, and (2) the distance to the destination vertex. If the

distance to the destination == INF, output “**Sorry, destination unreachable.” Here’s an example of navigating from “Mudd” to “Ford”:

```
Enter building name, * to list, @ to navigate, or $ to end>
@
Enter start building name (partial or complete)>
Mudd
  Name: Seeley G. Mudd Science and Engineering Library
  Approximate location: (42.05813505, -87.6740655214)
  Closest footway ID 376278372, node ID 2240260064, distance 0.0102273965997
Enter destination building name (partial or complete)>
Ford
  Name: Ford Design Center
  Approximate location: (42.0569162733, -87.6766148467)
  Closest footway ID 491941796, node ID 4840241543, distance 0.011164948377
Shortest weighted path:
  # nodes visited: 2305
  Distance: 0.373053310489 miles
```



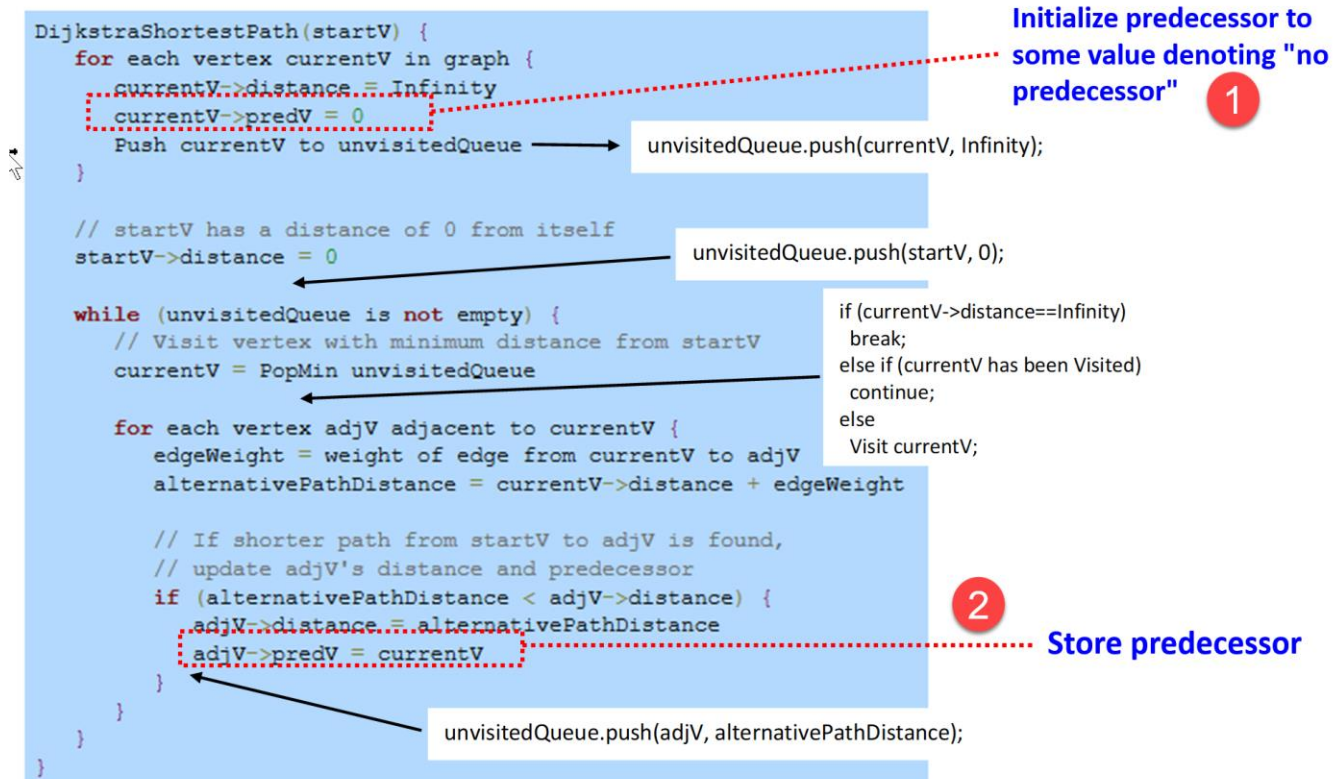
Dijkstra’s algorithm visits 2305 vertices, and finds a shortest distance of 0.3730577268 miles. [NOTE: Dijkstra’s algorithm is computing the distance to all reachable vertices, which is why the # of visited vertices is so high. We could reduce this # if we stop when the destination vertex is visited, but ignore this optimization.] Here’s another example from Mudd to Segal Visitors Center:

```
Enter building name, * to list, @ to navigate, or $ to end>
@
Enter start building name (partial or complete)>
Mudd
  Name: Seeley G. Mudd Science and Engineering Library
  Approximate location: (42.05813505, -87.6740655214)
  Closest footway ID 376278372, node ID 2240260064, distance 0.0102273965997
Enter destination building name (partial or complete)>
Segal
  Name: Segal Visitors Center/Parking Garage
  Approximate location: (42.0505184647, -87.6734721824)
  Closest footway ID 484580547, node ID 4774678138, distance 0.00508488850273
Shortest weighted path:
  # nodes visited: 2305
  Distance: 0.804291086921 miles
```

Two steps left... First, if you haven’t already, fix the program prompt to include the “@” command we’ve added here in project 07. Here’s the new prompt that main() should display:

```
Enter building name, * to list, @ to navigate, or $ to end>
```

Second (and final step!) is to complete Dijkstra's algorithm so it computes and returns the "predecessors array". This will allow the path from start to destination to be output. In pseudo-code terms, there are only two additions needed to Dijkstra's algorithm:



The classic representation is an array, but you are free to use a vector, or map, or any C++ container available in standard C++ 17. Modify your `dijkstra()` function to return a predecessors array of some kind, and have your `navigate()` function (or another helper function) output the shortest weighted path from the starting node to the destination node. [*Hint: use a stack, or recursion.*] Here are a couple examples:


```

Enter building name, * to list, @ to navigate, or $ to end>
@
Enter start building name (partial or complete)>
Mudd
Name: Seeley G. Mudd Science and Engineering Library
Approximate location: (42.05813505, -87.6740655214)
Closest footway ID 376278372, node ID 2240260064, distance 0.0102273965997
Enter destination building name (partial or complete)>
Ford
Name: Ford Design Center
Approximate location: (42.0569162733, -87.6766148467)
Closest footway ID 491941796, node ID 4840241543, distance 0.011164948377
Shortest weighted path:
# nodes visited: 2305
Distance: 0.373053310489 miles
Path: 2240260064->533996672->533996671->9119071425->9119071420->9119071421->9119071422->9119058008->3
775974464->2240260143->9119057998->2240260144->2240260146->2240260151->2240260149->2240260148->22402601
47->2240260150->4838625898->4840282003->4840282004->4840282005->4840282006->6685320452->4840282007->484
0282008->10285735118->2239483512->10285735117->4838625891->4840280244->9826982553->4840280243->48386258
90->4838625880->4838625879->10285735116->9312105609->4838831922->2239483461->4838831921->4840241542->48
40241543
  
```

```

Enter building name, * to list, @ to navigate, or $ to end>
@
Enter start building name (partial or complete)>
Mudd
  Name: Seeley G. Mudd Science and Engineering Library
  Approximate location: (42.05813505, -87.6740655214)
  Closest footway ID 376278372, node ID 2240260064, distance 0.0102273965997
Enter destination building name (partial or complete)>
Segal
  Name: Segal Visitors Center/Parking Garage
  Approximate location: (42.0505184647, -87.6734721824)
  Closest footway ID 484580547, node ID 4774678138, distance 0.00508488850273
Shortest weighted path:
# nodes visited: 2305
Distance: 0.804291086921 miles
Path: 2240260064->533996674->4838815124->4804966822->3797194109->4804966825->4804966824->4804966827->
4804966826->3797194110->375479406->4804966832->4804966831->4804966830->4804966833->3797194107->37547940
5->4837654457->375479404->2648980402->1765697265->375477623->1641389963->375477620->375480070->17656975
56->375477618->1641389803->1765697433->375477617->1765697518->1765697353->375477615->1765697424->375477
614->375477612->4771951638->1765697607->4771951637->375477611->4771951636->1765697302->375477610->47719
51635->375477609->4771951634->4771951633->4771951631->4771951632->375477608->3797201521->375477607->477
1951629->1765697215->4771951630->375477606->1765697326->375477605->4771951626->1765697347->4771951621->
375477604->4771951622->375477601->4771951623->1765697296->1765697290->1765697376->375477599->4771951624
->4771951625->1765697523->375477598->1765697501->4776045311->4776045310->375477597->375477596->47760453
09->1765697525->375477595->1765697312->375477594->1765697288->1765697286->4776045307->4776045308->37547
7593->1765697548->375477592->1765697223->1765697387->375477591->375477590->375477588->1765697284->17656
97524->1933980306->1933980317->1765697336->4806695298->4838561251->1765697602->1765697383->1765697393->
4838561254->4838561250->4838561253->4817039708->4817039709->9312129135->3673954438->3673947021->9312129
138->3673954441->4838552027->3673954463->4774714348->3673954428->4775444233->5701174817->8726174176->47
75444235->5701174818->4775444236->4775444253->4775444254->4775444255->4775444251->5701178693->570118207
7->4775444252->4775444256->5701182078->5671906921->4774678142->5701212386->4774678140->4774678139->4774
678138

```



Grading and Electronic Submission

You will submit all your .cpp and .h files to Gradescope for evaluation. To submit from the EECS computers, run the following command (you must run this command from inside your **project07/release** directory):

```
/home/cs211/w2023/tools/project07 submit *.cpp *.h
```

The “Project07” submission will open a few days before the due date. Once available, you have unlimited submissions and can submit to Gradescope at any time.

By default, Gradescope records the score of your **LAST SUBMISSION** (not your highest submission score). If you want us to grade a different submission, activate a different submission via your “Submission History”. This must be done before the due date.

The autograding portion of the project will be worth 80/100 points. The remaining 20 points will be determined manually by confirming the following requirements and reviewing your overall approach. However, you may lose correctness points if you have a feature correctly implemented, but manual review

reveals that the proper steps were not taken (“no nested loops in main()”):

1. *No nested loops inside main()*
2. *Using graph as specified to store nodes and footways, coded as separate helper function*
3. *Implementation of Building::getLocation() as specified*
4. *Implementation of navigate() helper function (additional helper functions are encouraged!)*
5. *Implementation of Dijkstra’s algorithm as a separate function*
6. *Header comments at top of each .h and .c file, and above each function*
7. *Meaningful comments describing blocks of code (e.g. loops)*
8. *No memory errors AND no memory leaks. You can run valgrind on the EECS computers as we did on replit: **valgrind ./a.out***

Academic Conduct Policy

Northwestern publishes a basic guide to academic integrity, which can be found [here](#). In summary, here are NU’s eight cardinal rules of academic integrity:

1. *Know your rights*
2. *Acknowledge your sources*
3. *Protect your work*
4. *Avoid suspicion*
5. *Do you own work*
6. *Never falsify a record or permit another person to do so*
7. *Never fabricate data, citations, or experimental results*
8. *Always tell the truth when discussing your work with your instructor*

School policies and more information can be found on NU’s academic integrity [website](#). With regards to CS 217, unless stated otherwise, all work submitted for grading **must** be done individually. While we encourage you to talk and learn from the course staff, peers, and others, 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.

Examples of what is not allowed? Downloading work from a github repository and submitting it as your own, whether partial or complete. Downloading answers from StackOverflow and submitting them as your own. Emailing your work to another student, or receiving work from another student. Sharing your screen with another student so they can see your work (and potentially submit it as their own). Participating in a screen share and taking screenshots or photos of someone else’s work, and then using that as a guide to submit your own work. Copying answers posted to Piazza, making a few simple changes, and then submitting as your own work. Allowing someone else to write / type the answer for you.

Okay, so what is allowed? Talking to the instructor or course staff, and getting insights --- but not direct answers --- in how to solve the problem. Talking to other students about the problem, using diagrams / natural language / pseudo-code to convey ideas. Searching the internet for guidance, and using that guidance to help you form your own solution. If you do receive help / guidance, it’s always best to cite your source by name or URL.