

Project 6 – OpenStreetMaps

CS 251, Spring 2025

Copyright Notice

© 2025 Ethan Ordentlich, 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 and your user information will be released.

By the end of this project, you will have...

- Implemented a graph
- Parsed input data from an industry-standard format
- Implemented a fundamental graph algorithm
- Have a working maps application that can find paths in real data

Restrictions

- You should not need to allocate memory (with **new**) in this project.
 - You may not edit the signatures of the public member functions, or add public member variables of the **graph** class.
 - You may add private member variables and functions.
 - You may not modify the signatures of other functions. You may add additional helper functions.
-

Logistics

Due:

- Gradescope: 11:59 PM Tuesday April 29
- Submit to Gradescope
 - **graph.h**
 - **application.cpp**
 - Any additional files for the JSON library you choose
- Use grace tokens:

- https://script.google.com/a/macros/uic.edu/s/AKfycbyqq_Kmrrw98LsVOYXt_xtNQ5A9nJoTrq9WEchRomJsTXIiZmZbjSRCgtWob_qnoqT/exec
 - The form will become submittable **once your project autograder closes**.
 - See [\[SP25\] CS 251 Course Info](#) for details.
-

Grading Breakdown

Project grading is based on **milestones**. To complete a milestone, you'll need to both:

- Complete **every** milestone before it, and
- Pass *all* of its tests without any memory errors or leaks.

Milestone	Total Points
graph	70
buildGraph	85
dijkstra	100

By completing the project (all milestones) early, you can get a small number of extra points. Grace tokens cannot be used on early deadlines. The early deadline will not be extended.

Early Deadline	Extra Points
11:59 PM Friday 4/25	10
11:59 PM Monday 4/28	2.5

FAQ

[\[SP25\] OpenStreetMaps FAQ](#)

Running Code

- `make osm_tests`
 - This default target builds the `osm_tests` executable
 - `make test_graph`
 - `make test_build_graph`
 - `make test_dijkstra`
 - `make test_all`
 - `make osm_main`
 - `make run_main`
 - These two targets allow you to run the `main.cpp` file, which provides a text-based interface to find shortest paths.
 - `make osm_server`
 - `make run_server`
 - These two targets allow you to run the `server.cpp` file, which provides a local web server to graphically visualize the shortest paths.
 - If you downloaded the starter code before 11:30 AM on 4/25, you are missing some files. Re-download the starter code to get the `server.cpp` and `httplib.h` files, and the `www` directory.
-

Task: Graph

This project does not have a test-writing component.


In `graph.h`, you'll implement a `graph` class. Make sure to obey the runtime restrictions described in each function's documentation. If you don't, the tests will time out!

The `graph` class represents graphs with all of the following properties:

- Directed – edges have a start and an end.
- Simple – for any two vertices A and B, there is at most one directed edge from A to B.
 - The directed edge from B to A is a separate edge that can also exist.
 - Although in our graph, a vertex can have a self-loop (edge from A to A).

- Weighted – each edge has data associated with it, typically a number.

If you choose the right internal representation and use it appropriately, none of your functions will be over 10 lines. Our `graph.h`, including comments and documentation, is approximately 100 lines in total. If your implementation is significantly longer, you are likely overcomplicating your approach.

 [SP25] OpenStreetMaps FAQ

Implement the functions defined in `graph.h` above the comment according to their documentation and runtime constraints. You **must** use an adjacency list implementation, and should add private members to accomplish this.

Verify that your `graph` implementation works with the provided tests: `make test_graph`. **Do not move on until all the graph tests pass.**

OpenStreetMap Data

[OpenStreetMap](#) (OSM) is a free crowd-sourced map of the world. We'd like to store this data in a graph to be able to run a graph algorithm (pathfinding) on it.

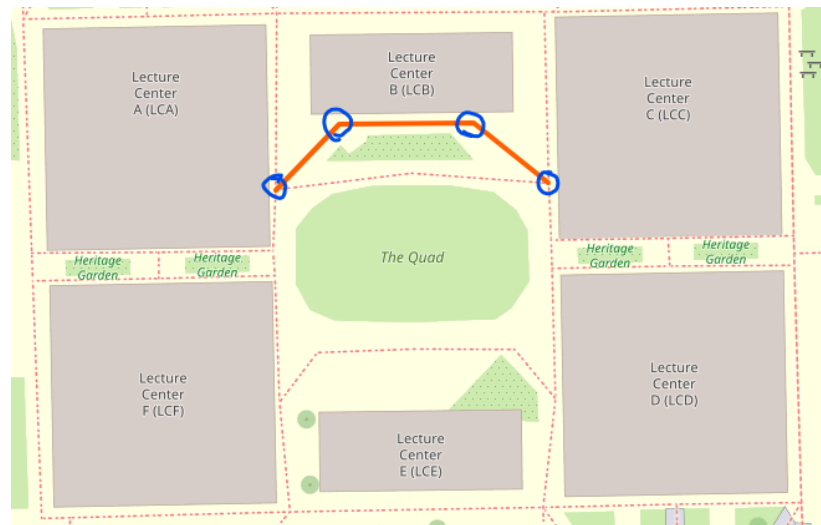
How can a graph be used to represent a map? One interpretation is that vertices represent places of interest; and edges between vertices show connections between these places, such as walking paths or streets.

A **node** in OSM is a single point in the real world, and contains 3 pieces of data that we care about:

- ID (`long long`, unique identifier)
- Latitude (`double`)
- Longitude (`double`)

For example, [Node 462010738](#) identifies an intersection of a bunch of walkways near the quad.

OSM doesn't directly link nodes with edges, instead using "ways". A **way** in OSM is a list of nodes that represents a single "line". For example, [Way 1176484406](#) is the walking path on the north side of the quad, around the benches.



This way consists of 4 nodes, approximately from west (left) to east (right):

1. [10930768586](#)
2. [11757616193](#)
3. [11757616194](#)
4. [10930768587](#)

We can convert a way to graph edges by looking at consecutive pairs of nodes, and the distances between these pairs of nodes. The above way has 3 edges. Even though the list is ordered, the actual footpath is bidirectional, so these are undirected edges.

OSM's data also tells us that the nodes happen to be part of other ways, by using the same node ID in multiple places. This is how it connects footpaths to other footpaths, and builds up the entire graph.

The other kind of way we're interested in is a **building**. OSM defines a building by the nodes in its perimeter. For example, [Student Center East \(Way 151672205\)](#) is made up of 13 nodes, despite seeming rectangular-ish. We won't need the outlines of buildings for

our application, so in the input data file we've given you, **we've converted each building to a single node.**¹

Unfortunately, these new building nodes aren't connected to the graph by OSM ways! To fix this, you will add new edges from building centers to "nearby" way nodes. It won't exactly correspond to reality, but it'll be close enough.²

Task: Loading OSM Data

The format that OpenStreetMaps uses ([osm](#)) is a form of XML. We think it's easier and more useful for you to learn how to work with JSON; so we've pre-processed the OpenStreetMaps data export into a JSON file.³ JSON is one of the most common data formats, and is especially prominent in web development⁴. C++'s standard library doesn't have a JSON parser – rather than having you write one (which can get pretty complicated), we'll use one that someone's written!

For this task (Loading OSM Data) **only**, you may use Large Language Models (LLMs): ChatGPT, Claude, Copilot, etc.

Why now, and why for this task? We've written a lot of C++ code up to this point to practice the fundamentals. Some of it involved code that we gave to you, but this is the first time we'll be using a third-party library. C++ JSON libraries in particular include a lot of code that uses advanced concepts, and support many, *many* different ways of using them. Most C++ libraries (in our experience) don't have good docs.

LLMs are really good at helping write code with **well-known, commonly-used** libraries in a language that we **are familiar with**. Since we (hopefully) have a good understanding of the fundamentals of C++, we have a solid foundation to incorporate LLM output into our own program.

¹ We used a [centroid](#) formula on the building's points to compute the coordinates.

² Unfortunately, OSM doesn't include information about building entrances, or else we'd use that.

³ If you want the script for your own project(s), happy to share it, I guess? It's not a very clean or well-documented script. And it's in Python.

⁴ It even stands for JavaScript Object Notation!

Our recommendation is [nlohmann's json](#), a common industry-standard C++ JSON parser and what we use in the course staff solution. There's many other libraries out there – as long as it's header-only (aka "single-include"), feel free to choose any one of them. Put the necessary files into the root directory of your project (the same directory as all the other source files).

When you submit to the autograder, make sure you submit the file(s) for your chosen JSON library!

One constraint, though: don't make a directory. The autograder isn't written to handle copying around folders, only individual files. Given this constraint, "header-only" or "single-include" libraries are easier to deal with.

Open the file `small_buildings.json`, and scroll through it. A JSON object is similar to a C++ map, where we associate (string) **keys** to **values**. Unlike a C++ map, a JSON object can have values of any type, including other JSON objects. This JSON file has three keys:

- `"buildings"`: a list of (most) UIC buildings as vertices
- `"waypoints"`: a list of non-building vertices
- `"footways"`: a list of OSM ways, as described above

To see the full dataset, check out the `uic-sp25.osm.json` file, which contains the data for the entire UIC campus, plus a little bit extra in the bounding rectangle.

You can read more about JSON in various places online – we like the article [An Introduction to JSON | DigitalOcean](#), since it avoids getting into JavaScript.

Also read `dist.h` to see functions for working with `Coordinates`, and `application.h` to see the `BuildingInfo` struct.

First, add your choice of JSON library to your project by downloading the necessary file(s) and adding them to the project folder.

In `application.cpp`, follow the directions below to implement the `buildGraph` function using your chosen JSON library and an LLM.

A graph that represents the real world should be undirected. Additionally, the JSON data doesn't include edges that connect building vertices to the rest of the graph. To fix this, add an undirected edge between each building and any non-building vertex within **0.036 miles**.⁵

[SP25] OpenStreetMaps FAQ

There are many ways to work with Large Language Models (LLMs), some of which are more effective than others. We strongly encourage you to follow each of the suggestions below so you can see for yourself how different approaches will produce results of varying quality and "correctness", for some definition.

(That said, we're pretty sure that LLMs have vacuumed up a solution to this project or something similar in their training data by now, so who knows—maybe they'll just magically output a completely correct thing no matter what you do. Probably not, but I don't really know how well the premium models work. The LLM landscape could also completely change in the next few days, so... ˘(ツ)˘)

(The last sentence was supposed to be a joke but I wrote it the night of 4/15 and OpenAI released o3 and o4-mini the next morning...)

1. First, add your choice of JSON library to your project by downloading the necessary files and adding them to the project folder.
2. You will have 2-4 separate conversations with an LLM, and compare the "quality" / "correctness" of their outputs.
 - a. Give the large language model **only** the signature and documentation for `buildGraph` from `application.h`, and ask it to use the JSON library you chose above to implement the function.

Does it produce a correct implementation for your needs? If not, can you identify how it's misinterpreted or hallucinated – why is it not compiling, or why are the tests failing?

- b. **In a new conversation**, give the large language model, in one message:

- The signature and documentation for `buildGraph`
- **And** the `BuildingInfo` struct from `application.h`
- **And** the *entirety* of the contents of `graph.h`

⁵ Chosen by just checking the minimum distance from a building to any surrounding node at some previous point, then just sticking with it over multiple semesters.

- **And** the *entirety* of the contents of `dist.h`

For example, something like: "I want to implement this function: <FUNCTION>. I have the following header files: <FILE 1> <FILE 2>".

You can use Shift+Enter to create new lines in most LLM web interfaces without sending the request.

How does this new information change the output the LLM produces?

Make sure that you understand everything that the LLM is writing.

- c. **In a new conversation**, give the large language model, in one message:

- The signature and documentation for `buildGraph`
- **And** the `BuildingInfo` struct from `application.h`
- **And** the *entirety* of the contents of `graph.h`
- **And** the *entirety* of the contents of `dist.h`
- **And** the *entirety* of the contents of `small_buildings.json`
- **And** the relevant information from the yellow box above.

How does this new information change the output the LLM produces? As above, **make sure that you understand everything that the LLM is writing.**

Even the last prompt is unlikely to spit out a perfect solution right away, but that's as far as we'll be able to guide you without knowing your specific situation and prompting.


Due to floating point weirdness, the distance from A to B is not necessarily the same as the distance from B to A. You should call `distBetween2Points` exactly once per pair of vertices, and use that for both edges between A and B.

Verify that your data loading works with the provided tests: `make test_build_graph`. **Do not move on until all the `buildGraph` tests pass.**

Task: Shortest Paths

Starting here, the "no usage of ChatGPT or similar" restrictions are in effect again.

We will use a variant of Dijkstra's: the paths should not include buildings, except for the buildings at the start and end of the path. More generally, we specify a set of vertices that the shortest path should **not** go through.

 [SP25] OpenStreetMaps FAQ

In `application.cpp`, implement `dijkstra`. The following sections contain some tips for implementation.

Miscellaneous

- Note the constant `double INF = numeric_limits<double>::max()` at the top of `application.cpp`. We can use this to represent "infinity".
- The returned vector should be the vertices on the path, in order from `start` to `target`.
 - If the `target` is unreachable from `start`, return an empty path.
 - If `start` and `target` are the same, return a length 1 vector containing `start`.
- The shortest path shouldn't go through vertices that are in `ignoreNodes`. One exception is that `start` and `target` might be in that set, so we ignore that those are ignored.

C++ Priority Queue

To implement Dijkstra's, we'll need a priority queue that can store both the vertices we're considering, and the best known distances to those vertices. We'll use C++'s `priority_queue` in the `<queue>` library, which uses a heap internally.⁶

We can declare a C++ STL `priority_queue` as follows:

⁶ https://cplusplus.com/reference/queue/priority_queue/

```
priority_queue<pair<long long, double>,
               vector<pair<long long, double>>,
               prioritize>
worklist;
```

This has 3 (!!!) template arguments:

- Type of the stored data (`pair<long long, double>`)
 - We need to store pairs because we want to prioritize based on the distance, but we also need to know the node ID that each distance corresponds to.
- Type of the "backing container" for the heap (`vector<pair<long long, double>>`)
- Type of the "comparison function" (`prioritize`)
 - We need to define the comparison function because the C++ priority queue outputs the **largest** element by default, while we want the **smallest**.

The comparison function that we use in the above declaration is a custom class called `prioritize` that needs to be defined somewhere:

```
class prioritize {
public:
    bool operator()(const pair<long long, double>& p1,
                    const pair<long long, double>& p2) const {
        return p1.second > p2.second;
    }
};
```

The details aren't relevant to us, aside from the idea that this is a "callable" that "compares" two elements in a specific way. There are several other ways to specify the comparison function, but they're beyond the scope of what we want to deal with right now.

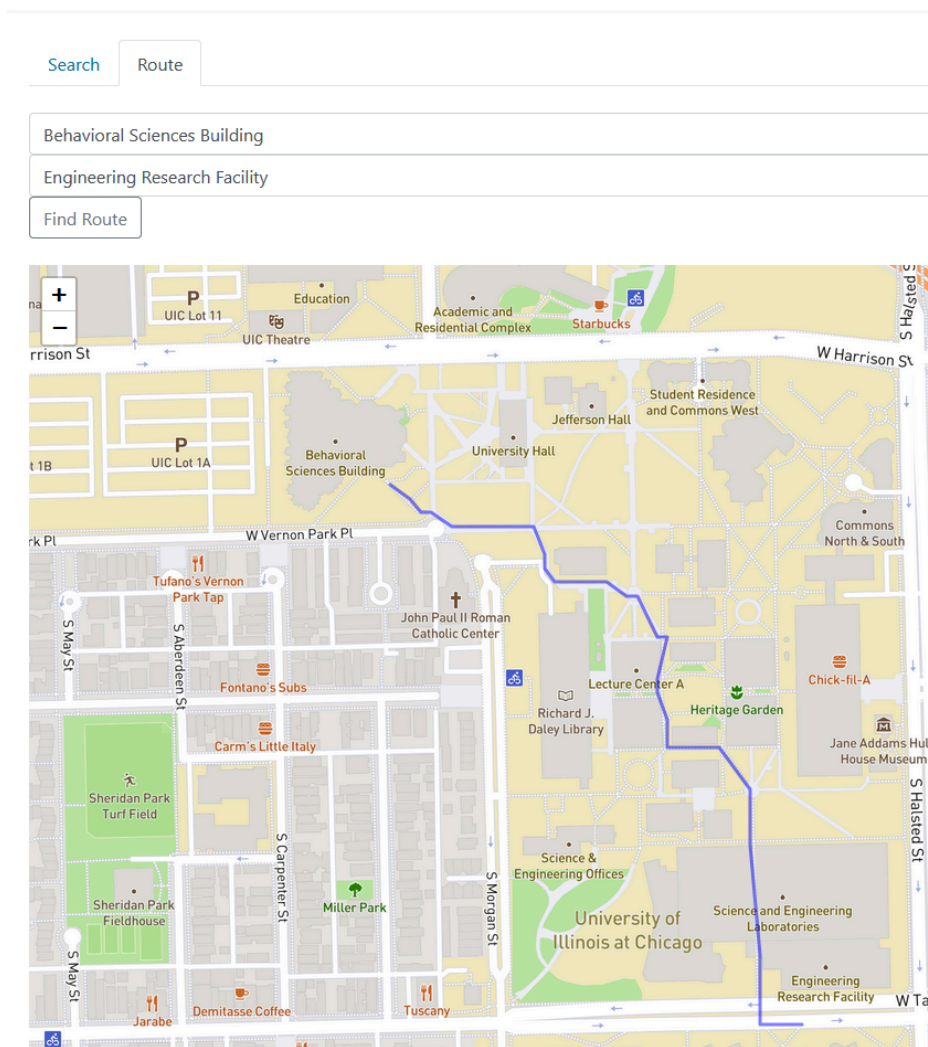
C++'s `priority_queue` does not support updating priorities. Instead, just add the vertex ID again and ignore vertices that are popped multiple times.

Main

Once you're done, you can run `make run_server` to run a **C++ web server** that will display the graphical output of the shortest path between two buildings, using our downloaded OpenStreetMaps data.

If you downloaded the starter code before 11:30 AM on 4/25, you are missing some files. Re-download the starter code to get the `server.cpp` and `httpLib.h` files, and the `www` directory.

Here's an example output.



The more boring way is to `make run_osm` to find pathways between buildings, and output to your terminal. Here's a sample execution, with user input in **red**.

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

```
# of buildings: 58
```

```
# of vertices: 7386
```

```
# of edges: 22444
```

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

```
ARC
```

```
Enter person 2's building (partial name or abbreviation)> SEO
```

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

```
Academic and Residential Complex
```

```
664275388
```

```
(41.874808, -87.650996)
```

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

```
Science Engineering South
```

```
157659429
```

```
(-87.648275, -87.648275)
```

```
Destination Building:
```

```
Lecture Center F
```

```
151672204
```

```
(41.871653, -87.649737)
```

```
Person 1's distance to dest: 0.25556712 miles
```

```
Path:
```

```
664275388->9007520455->2412572929->1645208827->464345369->463814  
052->11174974876->464748194->462010750->462010751->9862302685->9  
870872111->7511858534->9870872110->462010753->9870872084->462010  
766->9870872109->9870872108->462010765->9870872105->462010742->9  
870872103->12108530540->462010738->10930768586->151672204
```

Person 2's distance to dest: 0.23350503 miles

Path:

157659429->5632908806->5632908804->4226840176->1308175623->10612
976164->10612976165->1770663458->4226840184->1647971942->1647971
957->462014176->9870872090->462010748->9862302460->462010745->46
4345426->151672204

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

#

**** Done ****