

Project 5 – OpenStreetMaps

CS 251, Fall 2024

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

- have implemented a graph
- have 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, December 3
- 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/AKfycbw_Pi4Pn-dxBnXAZrSPBY49m-w705uxyf0JIYGDXA0db52p86hby2Ki9vu4-b59vIQJcg/exec
 - This form requires your UIC Google Account. The permissions it requests are to know who you are, and for the script to edit our spreadsheet database in the back end.
 - The form will become active **once the project autograder closes**.
 - See [\[FA24\] CS 251 Course Info](#) for details.

FAQ

[\[FA24\] OpenStreetMaps FAQ](#)

Task: Graph


This project does not involve writing tests.

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.

 [FA24] 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.

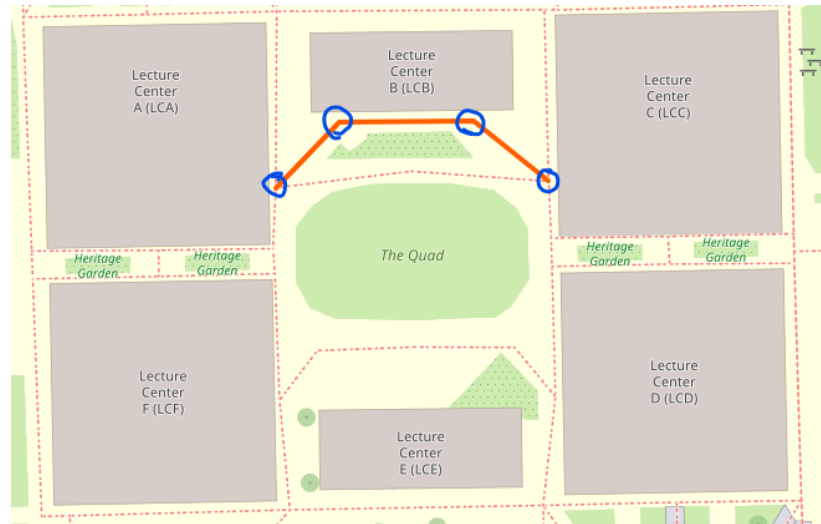
A **node** in OSM is a single point in the real world, and contains 3 pieces of data that we care about:

- ID (`long long`, unique identifier)

- Latitude (**double**)
- Longitude (**double**)

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

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 to east:

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

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

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. JSON is one of the most common serialization formats, and is especially prominent in web development³. 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 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.

ChatGPT is 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 ChatGPT's output into our own program.

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 or can be (quickly) compiled from source, 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.

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

³ It even stands for JavaScript Object Notation!

Open the file `uic-fa24.osm.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

You can read more about JSON in various places online – we like the article [An Introduction to JSON | DigitalOcean](#).

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

[FA24] OpenStreetMaps FAQ

First, add your choice of JSON library to your project by downloading the necessary files and adding them to the project folder.

In `application.cpp`, implement the `buildGraph` function using your chosen JSON library. A graph that represents the real world should be undirected.

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**.⁴

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: Meeting Points and Shortest Paths

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


⁴ Chosen by just checking the minimum distance from a building to any surrounding node.

Rather than shortest paths from A to B, our application will take two starting points, and have them “meet in the middle”. Assume one person starts at building A, and another person starts at building B.

1. Find the building closest to the midpoint of the line between buildings A and B; call this building M.
2. Run Dijkstra’s algorithm to find the shortest path from building A to building M.
3. Run Dijkstra’s algorithm to find the shortest path from building B to building M.

This happens in the `application` function, and **is given in the starter code**. You are not responsible for implementing this.

We will use a slightly altered version 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.

 [FA24] 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 the file. 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 `ignoreVertices`. The 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. While we implemented one in Project 4, we also hardcoded that priorities were `ints`. Here we need `doubles` (and a BST really isn’t a great priority queue...). Instead, we’ll use C++’s `priority_queue` in the `<queue>` library.⁵

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

⁵ 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 want to know the node IDs that each distance corresponds to.
- Type of the "backing container" (`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 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 important, 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 should be able to run `make run_osm` to find pathways between buildings. Here's a sample execution, with user input in **red**.

```
** Navigating UIC open street map **
# of buildings: 58
```

of vertices: 7292

of edges: 22218

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 Offices

151960667

(-87.650506, -87.650506)

Destination Building:

Stevenson Hall

151676521

(41.872744, -87.650112)

Person 1's distance to dest: 0.16098636 miles

Path:

664275388->9007520455->2412572929->1645208827->464345369->463814052->1174974876->464748194->462010750->462010751->9862302685->9870872111->9862302686->9862302687->9862302692->9870872081->9862302654->9862302653->151676521

Person 2's distance to dest: 0.14563479 miles

Path:

151960667->1647971930->462010746->12108530536->1645121274->1645121428->12108530537->1645121533->1647973070->151676521

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

**** Done ****

Scoring

This project is graded 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
<code>graph</code> make test_graph	67
<code>buildGraph</code> make test_build_graph	80
<code>dijkstra</code> make test_dijkstra	100