Project 5 – OpenStreetMap

CS 251, Spring 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 use 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.
* You may only modify graph.h and application.cpp.
* As with all other projects, you may not share code.

## Logistics

Due:

* Gradescope: **Friday 4/26**
  + graph.h
  + application.cpp
* This project does not have grace tokens. **By university policy, all work must be completed by the last day of classes (4/26).**

## Task: Building an All-Purpose Graph

In graph.h, you’ll implement a generic 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 nodes 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.
* Weighted – each edge has data associated with it.

As a hint, if you choose the right internal representation and use it appropriately, none of your functions will be over 8 lines. Our graph.h, including comments and documentation, is just under 100 lines in total. If your implementation is significantly longer, you may be overcomplicating your approach.

Implement the functions defined in graph.h according to their documentation. You **must** use an adjacency list implementation, and may add private members.

If you’re getting errors related to “discards const qualifiers”, the problem is that member functions that are declared const cannot modify the class’s data. However, the [] operator might modify a map by creating an entry if the key isn’t present! Make sure that you’re instead using at, and checking that the key exists before doing so.

## 

## OpenStreetMap Data

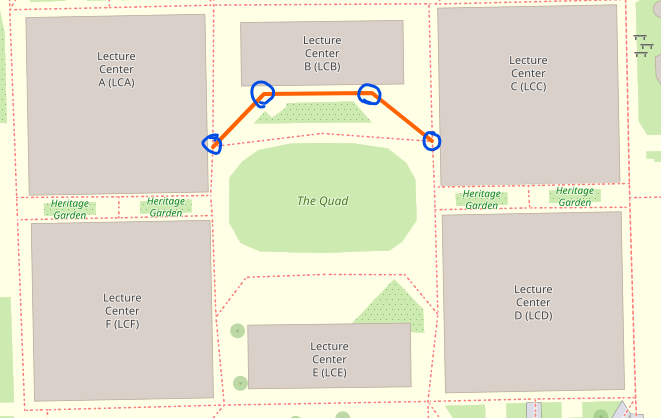
[OpenStreetMap](https://www.openstreetmap.org) (OSM) is a free crowd-sourced map of the world. We’d like to store this data in a graph to be able to run a graph algorithm on it.

How can a graph be used to represent a map? One interpretation is that vertices represent places people can exist, and edges between vertices show how people can move between these places.

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

* ID (long long, unique identifier)
* Latitude (double)
* Longitude (double)

Interestingly, OSM doesn’t directly link nodes with edges, instead using “ways”. A **way** in OSM is a list of nodes that represents a single path or street. For example, [Way 1176484406](https://www.openstreetmap.org/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](https://www.openstreetmap.org/node/10930768586)
2. [11757616193](https://www.openstreetmap.org/node/11757616193)
3. [11757616194](https://www.openstreetmap.org/node/11757616194)
4. [10930768587](https://www.openstreetmap.org/node/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 starting and ending nodes happen to be part of other ways. This is how we connect footpaths to other footpaths, and build 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)](https://www.openstreetmap.org/way/151672205) is made up of 13 nodes, despite seeming rectangular-ish.

To simplify our navigation problem, we convert each building to a single node by averaging the coordinates of the nodes in its perimeter.[[1]](#footnote-0) Unfortunately, these nodes aren’t connected to the graph by footways! To fix this, we will add new edges from building centers to nearby footway nodes. It won’t exactly correspond to reality, but it’ll be close enough.[[2]](#footnote-1)

The starter code already parses the OSM files into C++ data structures. Your task is to convert these data structures into a graph representation.

### Task: Loading OSM Data

First, read osm.h and dist.h to learn about the structs you’ll be using to represent OpenStreetMap data in this program.

In application.cpp, implement the buildGraph function. A graph that represents the real world should be **undirected**.  
  
The uic-2024.osm file should produce a graph with 23505 vertices and 12592 directed edges.

* A FootwayInfo contains a vector<long long> of nodes that make up the footway. Convert these into edges between consecutive pairs of nodes.
* A BuildingInfo contains a Coordinates. Add a vertex for the building to the graph, and create an edge from the building’s node to any other node within **0.041 miles** that is on a footway.**[[3]](#footnote-2)**

You will find the following useful:

* Functions in dist.h for edge weights
  + Due to floating point, 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 nodes, and use that for all edges between A and B.
* The Coordinates struct contains a bool indicating whether it is on a footway.

## Task: Meeting Points and Shortest Paths

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.

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 want to be able to specify a set of nodes that the shortest paths should not go through.

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 for “infinity” in our code.
* The returned vector should be the vertices on the path, in order from start to target.
  + If the target is unreachable from start, just return an empty path.
  + If start and target are the same, return a vector containing just start.
* Remember to not process vertices that are in ignoreNodes, unless the vertex is target.

### 

### C++ Priority Queue

To implement Dijkstra’s, we’ll need a priority queue. While we implemented one in Project 4, we also hardcoded that priorities were ints – and here we need doubles. Instead, we’ll use C++’s priority\_queue in the <queue> library.[[4]](#footnote-3)

We can declare a C++ STL priority\_queue as follows:

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>)
* Type of the “backing container” (vector<pair<long long, double>>)
* Type of the “comparison function” (prioritize)

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 too 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 node’s ID again and ignore nodes that are popped multiple times.

### Main

Once you’re done, you should be able to run make run\_osm\_main to find pathways between buildings. Here’s a sample execution, with user input in **red**.

\*\* Navigating UIC open street map \*\*

# of nodes: 23459

# of footways: 1450

# of buildings: 46

# of vertices: 23505

# of edges: 12592

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:

UIC Academic and Residential Complex (ARC)

664275388

(41.87478, -87.650866)

Person 2's point:

Science & Engineering Offices (SEO)

151960667

(41.870851, -87.650375)

Destination Building:

Stevenson Hall (SH)

151676521

(41.872875, -87.650468)

Person 1's distance to dest: 0.13976157 miles

Path: 664275388->2412572929->1645208827->464345369->463814052->464748194->462010750->462010751->9862302685->9870872111->151676521

Person 2's distance to dest: 0.16078044 miles

Path: 151960667->1647971930->462010746->462010758->11257988853->464345492->462010740->9870872102->462010759->151676521

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

## Scoring

|  | **Points** |
| --- | --- |
| Completely correct graph class | 25 |
| Processing OSM data into graph (buildGraph) | 20 |
| Dijkstra’s algorithm | 50 |
| Documentation + style (manual, see below) | 5 |

**Style**

* 2 points: Code is styled consistently; for example, using the VSCode formatter.
  + (F1, type in “Format Document”)
* 1 point: Code is reasonably styled, but there are consistent significant stylistic issues (e.g. inconsistent indentation, line length > 120, spacing, etc.)
* 0 points: No credit (e.g. entire program is on one line)

**Documentation + Commenting**

* 3 points: Code is well-documented with descriptive variable names and comments, but not overly documented.
* 1.5 points: Code is partially documented, due to a lack of comments and/or poor naming; or code is overly documented with unnecessary comments.
* 0 points: Code has no documentation or appropriate names.

1. An average isn’t quite right, but this is already enough of a digression. [This formula](https://en.wikipedia.org/wiki/Centroid#Of_a_polygon) is more accurate. [↑](#footnote-ref-0)
2. Unfortunately, OSM doesn’t include information about building entrances. [↑](#footnote-ref-1)
3. We promise this number is chosen meaningfully. [↑](#footnote-ref-2)
4. <https://cplusplus.com/reference/queue/priority_queue/> [↑](#footnote-ref-3)