CME212, Winter 2020
Advanced Software Development for Scientists and Engineers

Homework 1 – Due Friday January 31st at 4:30 P.M.

---

1 Overview

1.1 Goals

In HW0, you setup the initial elements of a `Graph` class. In this homework, we will work on extending and using the `Graph` class.

Our `Graph` class has a number of limitations. Namely

- Enumerating `Node`s and `Edge`s is possible via the `Graph::node(int)` and `Graph::edge(int)` methods, but this is awkward and potentially slow.
- `Node`s can't be asked about `Edge`s. Traversing the `Graph` is very difficult.
- `Node`s only have a `Point` and an index associated with them. It is difficult to use our `Graph` in more abstract ways.

You will make the `Graph` class more versatile. The object of this homework is to teach you how add new functionality to exiting code while keeping backwards compatibility. We will also introduce the basics of templates and inheritance. You will learn how to implement an iterator that works with **STL**. You will then use these iterator classes to implement shortest path traversal and subgraph traversal.

Concepts Introduced: Ordered collections, private inheritance, templates, iterators, functors.

1.2 Tasks

- Pull new starter code (short).
- Correct any feedback from HW0 (short).
- Add `Node` and `Edge` operators (short).
- Add the use of templates (short).
- Create iterator classes (very time intensive).
- Implement shortest path traversal (time intensive).
- Implement subgraph viewer (time intensive).

1.3 Tips

- Start early!
- Make sure you understand the starter code provided before you write your own code.
- Go over your EX1 to make sure you understand iterators.
- The filter iterator should be build on top of another iterator.
- You can always come to the teaching team with questions.

- Any error you encounter someone else will have had before. Stackoverflow is your best friend.
- Don't forget to pull the starter code.
- We advise committing and pushing frequently as you make progress on the assignment.
- Work together.

1.4 Starter code

- `Color.hpp` provides a `Color` class that allows you to color the nodes in your graph. It has three static member functions:

```
    Color Color::make_rgb(float, float, float)
    Color Color::make_hsv(float, float, float)
    Color Color::make_heat(float)
```

  This is used in the shortest path search.
- `shortest_path.cpp` Where you implement the shortest path. It contains a `main` block that will color your graph if your implementation is correct.
- `subgraph.cpp` Where you implement the filter iterator. It contains a `main` block that will show your subgraph if your implementation is correct.

2 Setup

Our helper code for Homework 1 extends the code from Homework 0. To retrieve it, follow these steps:

```
# Check the status of your git repository
$ git status

# Should say "on branch master"
#   Otherwise, save a HW0 branch and checkout the master branch
#   $ git branch hw0
#   $ git checkout master

# Should also show no changes (all commits should already be made for HW0)
#   Otherwise, commit your changes.
#   $ git commit -am "Stray HW0 changes"

# Load our changes into your repository
$ git fetch CME212-2020

# Apply our changes to your repository
$ git merge CME212-2020/master
```

Then follow git's instructions. For example, if there are conflicts, fix them by choosing between the code versions (deleting the all undesired code) denoted by `<<<<` and `>>>>` and commit the result with `git commit -a`. If the merge produces many conflicts, try `git rebase CME212-2020/master` instead.

Peers Code We will create a repository on Github which will host peers code alongside our solution. You are free to browse this repository and take inspiration. But note that the files have been anonymized so that it's up to you to understand what the code is doing pick out the good apples from the not-as-ripe. github.com/cme212/peercode

3 Completing Nodes and Edges

In the HW0, we designed a basic `Node` class and `Edge` class. We allowed a `Node` object to be added as well as an `Edge` object to connect nodes. In this section, we'll round out some of the operations that we can perform with them.

Objectives

- Extend boolean operators.
- Give Nodes a **templated** value attribute.

3.1 Node and Edge operators

In HW0, you defined the comparison operators for nodes and edges:

```
Graph::Node::operator<
Graph::Node::operator==
Graph::Edge::operator<
GRaph::Edge::operator==
```

There is an easy way get all the comparison operators using **private inheritance**. Inside `CME212/Util.hpp` we've added a class that implements a *totally ordered* collection.

By changing the declaration of your classes to:

```
class MyClass : private totally_ordered<MyClass> { ... }
```

your class will inherit

- The `!=` operator for free when `MyClass` defines the `==` operator.
- The `>`, `<=`, and `>=` operators for free when `MyClass` defines the `<` operator.

The `private` inheritance means that methods and data attributes which are public in the parent class are private in the child class.

Modify your Node and Edge classes so they inherit from totally ordered.

3.2 Modifiable Node Value and Class Templates

Nodes currently only have a position, we want to be able to give the Nodes and attribute such as mass, temperature or color. However we don't want to hard code the implementation for each attribute. That is why we use **Templates** (see lecture 2). Modify the Graph to become a class template: Graph<V>. This template parameter will allow the Nodes to support a user-specified value, of type node_value_type (the V argument of the Graph template). To use this value, provide the following public methods:

```
template <typename V>
class Graph
public:
  using node_value_type = V;
  class Node
    node_value_type& value();
    const node_value_type& value() const;

  Node add_node(const Point&, const node_value_type& = node_value_type());
```

Note You will need to change the creation of graph in viewer.cpp so that the graph is constructed with a template. Otherwise your code won't compile.

Note When you do shortest path traversal you will use this value to store the shortest path to a node from the root node.

4 Node Iterators

Objectives

- Create node_iterator class.
- Test node_iterator iterator.
- Create incident_iterator class.
- Create edge_iterator class.
- Test edge_iterator iterator.

4.1 Iterators

The index methods node(i) and edge(i) are useful, but also limiting. Your edge(i) function, for example, cannot be implemented *quickly* (with $\mathcal{O}(1)$ complexity) unless you store edges contiguously in some vector-like container and upkeep it with any other Edge-related data. We will therefore use **iterator** (see lecture 4) abstractions to loop over objects while hiding the details of how they are stored. These iterators will also let us use STL algorithms on graphs such as `std::min_element`.

First, add a `node_iterator` to the Graph with the following public interface:

Complexity requirement When you implement these functions, we require at most $\mathcal{O}(1)$ computational work. I.e. the amount of 'constant-time' operations used (such as basic arithmetic or dereferencing pointers) should *not* depend on the size of the graph.

4.2 Testing the Node Iterator

The `SFML_Viewer` provides a method to `add_nodes` with an iterator range. It takes a `node_map` argument, which maps input nodes to internal indices. Its Prototype is:

```
template <typename InputIter, typename Map>
void add_nodes(InputIter first, InputIter last, Map& node_map)
```

Change your `viewer.cpp` to add nodes using iterators, like this:

```
auto node_map = viewer.empty_node_map(graph);
viewer.add_nodes(graph.node_begin(), graph.node_end(), node_map);
viewer.center_view();
```

If your `node_iterator` and `Node` are correct, this should plot the points of the input files.

Incident Iterator

A very common graph operation that the `Graph` class still does not support is efficient graph traversal. We can operate on the `Nodes` and `Edges` globally, but cannot efficiently traverse the edges incident to a node.

An `incident_iterator` is meant to iterate over all edges incident to a Node.

Implementation Detail: "Orientation" of Returned Edge The `Node` that spawns the `incident_iterator` should always returned by `node1()` of each incident `Edge` and the adjacent `Node` was returned by `node2()`. Like this:

```
Node n = ...;
for (auto ei = n.edge_begin(); ei != n.edge_end(); ++ei) {
  Edge e = *ei;
  Node n1 = e.node1();
  assert(n1 == n);
}
```

implement the `incident_iterator` class with the following interface. As well as the following methods in the `Node` class.

```
class Node
  // return the number of incident edges.
  size_type degree() const;
  // Start of the incident iterator.
  incident_iterator edge_begin() const;
  // End of incident iterator.
  incident_iterator edge_end() const;
```

Iterating over all the edges incident to a `Node` must have maximum complexity $\mathcal{O}($`n.degree()`$)$. Thus, the iterator functions should have $O(1)$ complexity as well.

4.3 Edge Iterators

An `edge_iterator` is meant to iterate over all edges of a graph. Iterating over the edges should visit each edge exactly once. In particular, it should *not* visit both `Edge(a, b)` and `Edge(b, a)`: the graph is undirected. It does, however, remain unspecified which node is returned as `node1()` and which node is returned as `node2()`.

Provide an `edge_iterator` with the following interface

Hint: You could combine the `Node` and `Incident` iterators to create the `Edge` iterator.

4.4 Testing Edge Iterator

The `SFML_Viewer` also provides the method `add_edges` with the signature

```
template <typename InputIter, typename Map>
void add_edges(InputIter first, InputIter last, const Map& node_map);
```

Edit `viewer.cpp` to call this function after you call `add_nodes`:

```
viewer.add_edges(graph.edge_begin(), graph.edge_end(), node_map);
```

If your `edge_iterator` and `Edges` are correct, this will collect all edges, make sure they refer to previously added nodes, and buffer them for display.

5 Applications of iterators: Shortest path

Objectives

- Use incident iterator to do shortest path traversal.
- Use the shortest distance for each node to color the graph.

5.1 Shortest Path Lengths

To show that you have implemented the `incident_iterator` and `node_value_type` and `node_iterator` correctly, implement a shortest path length function in `short_path.cpp`.

Each traversal starts at a **root** node, which is the closest node to a given point.

The `nearest_node` function should find the `Node` within a `Graph g` that has a position with the smallest Euclidean distance to the `Point point`. The `nearest_node` function can be implemented using the `std::min_element` function which is built into the standard library. You will need to write your own comparison functor to do this (see lecture 2).

Now that we have a **root** we need to traverse the graph starting from the root.

The `shortest_path_lengths` function should, for each `Node` in `Graph g`, update `node.value()` to the path distance to the `root Node` and return the longest path length in the `Graph`. You may use a breadth-first search to calculate this.

5.2 Heat Map

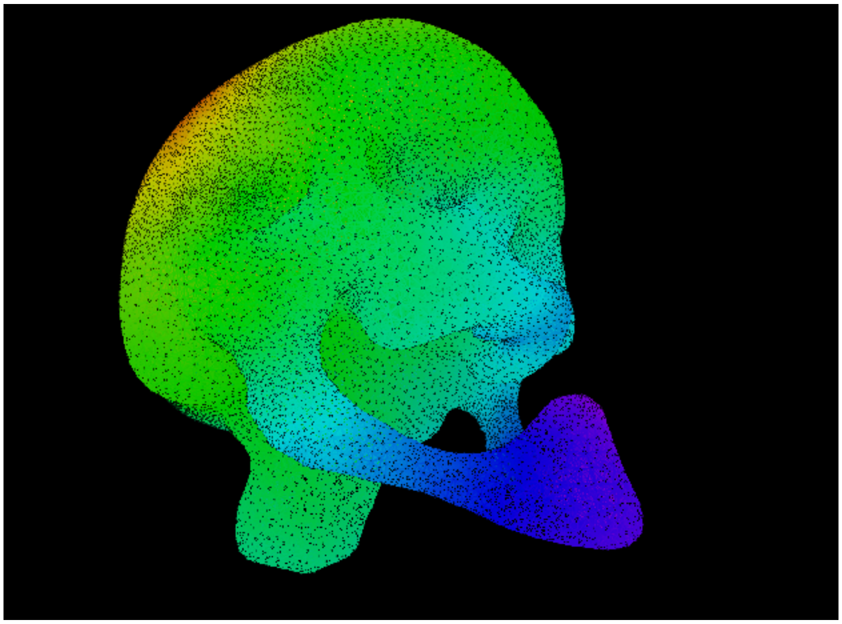
The `add_nodes` function in `SFML_Viewer` can also take in a color functor argument:

```
template <typename InputIter, typename ColorFn, typename Map>
void add_nodes(InputIter first, InputIter last,
               ColorFn color_fn, Map& node_map)
```

The color functor will be applied to each node and return a `Color` object.

You need to create a custom color functor that takes a `Node` and returns a `CME212::Color`.

Write a function that colors a graph's nodes using a heat map, based on their path distance from the closest node to `Point(-1,0,1)`. Below, we plot long paths lengths in blue-purple and short path lengths in red. On `large.{nodes,tets}`, this function should produce results like this:



6 Applications of iterators: Subgraph Viewer

6.1 Objectives

- Implement `filter_iterator` class.
- Create your own predicates.

A subgraph $H = (V^*, E^*)$ of a graph $G = (V, E)$ with $V^* \subseteq V$ is said to be an induced subgraph of G if H has all edges of G that are valid for its vertex set V^*. That is, every edge in G is an edge of H if V^* contains both vertices of that edge.

Naive Thought We want to plot an induced subgraph of our graph. There are multiple ways to do this, but the obvious one would be to construct a new graph object with all the nodes we want and all of the induced edges. This could take some work, memory, and time.

Induced Subgraph Alternatively, examine the `SFML_Viewer` documentation of `add_edges`:

```
* Edges whose endpoints weren't previously added to the node_map by
* add_nodes() are ignored. */
```

so another way would be to simply add only a subset of the nodes to the viewer! Only the edges of the induced subgraph will be added in `add_edges`!

All we really need is for the `node_iterator` to skip certain nodes...

6.2 Filter iterator

To do this we need to define a new iterator class that can iterate over the nodes and based on a given function skip the nodes we do not want. We've provided a skeleton of such a `filter_iterator` that is constructed on a **predicate functor** and an **iterator range**. Again, we emphasize that functors are classes that are assumed to provide the `operator()` method for their domain. This looks like

```
struct MyNodePredicate {
    bool operator()(const Node& node) const {
        // Return true or false based on node
    }
};
```

The `filter_iterator`'s job is to wrap an iterator range and skip any elements that do not satisfy the predicate. Complete the `filter_iterator` and use it to plot induced subgraphs. We supply a simple predicate, `SlicePredicate`, which returns true for nodes with certain positions.

Note The `filter_iterator` takes in another iterator. So most of the methods for the `filter_iterator` are simple wrappers for the underlying iterator methods.

Note Use the `make_filtered` helper function to create your filter iterators.

Hint: You may want to implement a function that checks whether or not the current node in the iteration is valid. This way you can keep skipping nodes until you reach a valid node.

6.3 Predicates

Define and test your own interesting predicate. Write your predicate, including its specification, in `subgraph.cpp`. For example, you might delete half the input graph, delete any isolated nodes (nodes with no edges), delete nodes with some probability, or delete nodes more than a defined distance away from a specified `Point`.

Combine the `subgraph` work and the `shortest_path` work to quickly and easily manipulate `Graph`s! Save an interesting screenshot and we'll show off some cool ones in class.

7 Submission Instructions

Checklist

Submission

Use a Git *tag* to mark the version of code you want to submit. Here's how:

```
$ git status                                    View files git is tracking
```

```
$ git add <files to track>                       Tells git which files to track
$ git commit -am "Describe last few edits"       Commit files
$ git tag -am "My HW1" hw1                       Tag this commit with tag ‘‘hw1’’
$ git push --tags origin master                  Push all commits to git repo
```

It is possible to change your tag later, too, if you discover a bug after submitting:

```
$ git tag -d hw1
$ git tag -am "My HW1" hw1
$ git push --tags origin master
```

We will use Git timestamps on tags and the associated commits to check deadlines. Be careful with overwriting tags – you don't want to lose the submission tag.

View your repository on GitHub (i.e. check the release tab on the UI) to verify that all of the files and tags were pushed correctly.