In HW0, we developed the initial elements of a Graph class, namely the node and edge objects. In this homework, we will work on extending and using the Graph class.

Our Graph class has a number of limitations. Namely

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

**Setup**

**Problem 1 - Completing Nodes and Edges [15%]**
- Node and Edge operators ................................................. 3
- Modifiable Node Value .................................................. 3

**Problem 2 - Node Iterators [15%]**
- Use the Node Iterators .................................................... 5

**Problem 3 - Edge Iterators [20%]**
- Use the Edge Iterators .................................................... 6

**Problem 4 - Subgraph Viewer [15%]**

**Problem 5 - Incident Iterator and Shortest Path [25%]**
- Incident Iterator ......................................................... 8
- Application: Shortest Path Lengths ................................. 8

**Submission Instructions**
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 cs207

# Apply our changes to your repository
$ git merge cs207/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 cs207/master instead.
Problem 1 - Completing Nodes and Edges [15%]

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.

Node and Edge operators

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

1. `Graph::Node::operator <`
2. `Graph::Node::operator ==`
3. `Graph::Edge::operator <`
4. `Graph::Edge::operator ==`

First, make sure these are correct with any comments provided in your code feedback. Then, we’ve got a cool protip:

**PROTIP:** Inside CS207/Util.hpp we’ve added a bit of magic. By changing the declaration of your classes to:

```cpp
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.

Modifiable Node Value

Second, modify the Graph to become a 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:

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

  Node add_node(const Point&, const node_value_type& = node_value_type());
```
Then, when initializing a Graph we can specify a `node_value_type` as a template parameter which we can then use to store and manipulate extra information about each Node. For example, I could write:

```cpp
// Create an empty graph with nodes that store doubles
Graph<double> graph;
// Add a node with position and value = 3.14
graph.add_node(Point(1,1,1), 3.14);
// Add a node with position and default value.
graph.add_node(Point(0,0,0));
// Work with the values
graph.node(1).value() = graph.node(0).value();
```

where I may interpret `value()` as mass, temperature, distance, weight, importance, etc. The `node_value_type` may take on the default value `node_value_type()` if no such value is passed to `add_node`. 
Problem 2 - Node Iterators [15%]

The index methods node(i) and edge(i) are useful, but also limiting. Your edge(i) function, for example, cannot be implemented quickly (with $O(1)$ 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 abstractions to efficiently loop over objects while hiding the details of how they are stored. These iterators will also let us use STL algorithms on graphs.

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

```cpp
class Graph
public:
    class node_iterator
    {
        Node operator*() const;
        node_iterator& operator++();
        bool operator==(const node_iterator&) const;
        node_iterator node_begin() const;
        node_iterator node_end() const;
    }
```

All these functions must have $O(1)$ complexity. You should then be able to write loops like:

```cpp
for (auto ni = g.node_begin(); ni != g.node_end(); ++ni) {
    auto node = *ni;
    ...
}
```

Use the Node Iterators

The SDLViewer now provides a method add_nodes to add nodes covered by an iterator range to the graph. It also takes a node_map argument, which it uses to map input nodes to internal indexes. Its signature is:

```cpp
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:

```cpp
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.
Problem 3 - Edge Iterators [20%]

Similarly, provide an `edge_iterator` to provide the same functionality over the `Edges`.

```cpp
class Graph
public:
    class edge_iterator
    Edge operator*() const;
    edge_iterator& operator++();
    bool operator==(const edge_iterator& eit) const;
    edge_iterator edge_begin() const;
    edge_iterator edge_end() const;
```

Iterating over the edges between `edge_begin()` and `edge_end()` should visit each edge exactly once. (In particular, it should not visit both `Edge(a, b)` and `Edge(b, a)`: the graph is undirected.)

Use the Edge Iterators

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

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

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

```cpp
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 make sense, and buffer them for display.
Problem 4 - Subgraph Viewer [15%]

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.

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.

Alternatively, if we look at the SDLViewer documentation of add_edges, it states

```cpp
* ... * 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!

Ok, so we can copy all the nodes from our graph to a container, remove the ones we don’t want, and add the rest to the viewer:

```cpp
std::vector<Node> g_nodes(graph.node_begin(), graph.node_end());
// ... remove the nodes we don’t want from g_nodes
viewer.add_nodes(g_nodes.begin(), g_nodes.end(), node_map);
```

This is also kind of annoying and inefficient since we have to make new container, a copy of every single node, manipulate those copies, then send them off to the viewer. Ideally, all we really need is for the node_iterator to skip certain nodes...

We’ve provided a skeleton of a filter_iterator that is constructed on a predicate functor and an iterator range. Functors are classes that are assumed to provide the operator() method for their domain. A predicate functor looks like

```cpp
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. 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.
Problem 5 - Incident Iterator and Shortest Path [25%]

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, for example. Additionally, we would like a Node to be able to carry information other than simply a 3D point. For example, each node could be given a mass or velocity that we can use in computations.

Incident Iterator

An incident_iterator is meant to iterate over all incident edges to the Node. Note that edge_iterator and incident_iterator are similar, but not identical. The edge_iterator needs to be undirected and have the scope of the entire graph. That is, every edge should be iterated over exactly once. The incident_iterator has the scope of only a single Node. That is, the Node that spawns the incident_iterator should always be returned by node1() of each incident Edge and the adjacent Node should be returned by node2().

Add the following public methods to your Graph class.

```cpp
class Graph
public:
    class Node
    
    
    size_type degree() const;
    incident_iterator edge_begin() const;
    incident_iterator edge_end() const;

    class incident_iterator
    
    
    Edge operator*() const;
    incident_iterator& operator++();
    bool operator==(const incident_iterator& iit) const;
```

Recall that the degree() of a node is the number of edges incident to it.

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

Application: Shortest Path Lengths

To show that you have implemented the incident_iterator and node_value_type template correctly, implement a shortest path length function:

```cpp
/** Calculate shortest path lengths in @a g from @a point.
 * @param [in,out] g Input graph
 * @param [in] point Point
```
This function should, for each Node in Graph g, store the shortest path distance to the Node root (determined as the closest node to point) in the node value() (of type int) and return the longest path length in the Graph. You may use breadth-first search to calculate this, for example. The root node should be calculated by computing the shortest distance of each node from the specified point.

HINT: This can be done by using the std::min_element function which is built into the standard library. To use this function you will want to construct your own comparator as outlined below.

```
struct MyComparator {
    Point p_;  
    MyComparator(const Point & p) : p_(p) {}  
};

template <typename NODE>
bool operator()(const NODE& node1, const NODE& node2) const {
    // Return true if node1 is closer to p_ than node2
}
```

When you are confident this is working, use the Node path lengths to generate colors in the plots. The SDLViewer also has the function

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

which can take a node_iterator to gather the coordinates as you have already done, but also applies the functor color_fn to each node in order to determine the node's color. This functor defaults to one which returns white for all nodes.

By creating a custom color functor that takes a Node and returns a CS207::Color, you can use the custom node_value_type (which stores the path length to root after your BFS) to color the nodes any way you like. In Color.hpp, we have provided you with a simple Color class with three static member functions:
Color Color::make_rgb(float, float, float)
Color Color::make_hsv(float, float, float)
Color Color::make_heat(float)

See their comments for documentation. Each of these functions return a CS207::Color object that can be used in the SDLViewer.

To test your work, 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:

![Image of a heat map graph](image.png)

Combine the subgraph work and the shortest_path work to quickly and easily manipulate Graphs! Save an interesting screenshot (or a .cpp file to compile and run) and I’ll show some cool ones in class.
Submission Instructions

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  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
```

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. We can handle versioned tags, such as `hw1_1`, `hw1_2`, etc.

To verify that all of the files were pushed correctly, you can click on your repository to the right of your code.seas account and view the “Source Tree”. There may be a delay between your push and the files showing up in the browser.