HW4: graphs, depth first search, and breadth first search
Due Saturday the 13th of December, 2014, at 11:55 pm
50 points: depth first search for path from city 1 to city 2
50 points: breadth first search for path from city 1 to city 2

OVERVIEW

Graphs are the last and most interesting structure that we'll cover in this class. Here, we're making a graph of routes between various world cities. Graphs consist of vertices (aka nodes) connected by edges. We need a data structure that can keep track of both. Imagine that we are smugglers and we need to find ways to get contraband in and out of all the great cities in the world. (You are free to imagine something nicer, but I prefer to envision a life of crime.) We have a little plane, though, so maybe we can't make long trips in one flight; we have to stop at cities along the way and refuel. This assignment will tell us whether we can make it from one city to another in our wobbly, overstuffed plane.

We add edges to this graph if they fall within the limit of the distance we can travel. With that done, you will be able to search the graph for a path between two vertices. We don't promise that this path exists for all limits. But if you increase the distance limit to a high enough number, all cities will become reachable from each other.

DETAILS

vertex.h (does not change)

The first file you need is vertex.h, which looks like a header but actually has all of the vertex implementation code as well. Treat it like a header, though. You will not change anything in this file. vertex.h defines our vertices, which are the cities. Each vertex knows its name (Birmingham, England), its latitude (in degrees and minutes), and its longitude (in degrees and minutes). It can also tell us all these things via methods like get_city_name(), get_latitude_degrees(), etc. Our vertex coordinates are not very accurate, since there are no seconds. If we were working in missile defense, we'd need better accuracy. But this is close enough for us.

test_graph.cxx (does not change unless you add more tests)

Testgraph.cxx just poses a few depth-first and breadth-first searches. You should add more tests as you get confident in the ones in there. But we'll use our own test code to grade.

graph.h and graph.cxx (graph.cxx will change)

You'll also need graph.h and graph.cxx. Don't change graph.h. The code you write will all be in graph.cxx. If you look at graph.h for the private member variables of the graph, you will see that we use a data structure from the C++ Standard Template Library (STL) called a vector (http://www.cplusplus.com/reference/stl/vector/) to store the vertices. A vector is an array-like class that can hold any data type. It comes with its own constructors and destructors, assignment operator, copy constructor, etc.; you don't need to write any of that. The vector class also has functions for adding and removing items, grabbing the i^{th} item, etc. Here's the line declaring the vertex vector in the file. We have to #include <vector> for this to work.

```
vector<vertex> vertices;  // vector of cities
```

Remember, our Vertices got defined in vertex.h. So this line promises that we'll keep Vertices in the vector called vertices (and we promise not to shove ints or statisticians or Bags or other data types in there).

We store the edges in a second STL data structure called a map (http://www.cplusplus.com/reference/stl/map/). A map is actually a way to store a piece of data by key; the key, value combination is a pair.

You'll find maps useful for lookup functions like this later on if you keep programming.
The edge map in our graph is a little more complicated than the sqrtTable map. Here is its declaration:

```
map<vertex *, vector<vertex *> > edges;
```

This map uses a vertex pointer (vertex*) as the key. For each vertex* key, it stores a vector (aha, an array) of other vertex pointers (vertex*). Squint and you can see that this is a nested data type: the map contains a vector. In theory, you can keep nesting as deep as you care to, but you have to put spaces between the > or < brackets in older C++ code. Else the compiler thinks you're talking about input/output operators (>>, <<) and gets all bent out of sorts.

We also have another map in a variable called path; this connects vertex* to other vertex*. Adding this variable lets us not just find the path, but also display it. Recall how these searches work; we pop a vertex* off the stack or the queue, and then explore its neighbors. Path remembers the relationship between a vertex* and its neighboring vertex* pointers in the search.

This is probably a bit alarming. But take heart. Those awful vertex* are just pointers to the vertex items we defined in the vertices vector. They let us take a vertex, get its address (using &), and then use the address (that's a pointer, you remember) to look up a list of that vertex's neighbors in the graph. If we know all the vertices, and we keep a list of each vertex's neighbors, then that's all that the graph needs to know about how it's connected. This map of vectors is called the *adjacency list* representation of a graph. It differs from the example in lecture, which uses an adjacency matrix, with 0s for missing edges.

Part 1, covered last week in lab:

When we run the code, we'll open up a data file of cities and their latitude and longitude coordinates. There are 119 cities in the file. The code I gave you does a lot of things already. It opens the file. Then it reads in what it assumes is a city name till it sees a colon (this marks the boundary between the name and the latitude/longitude data). Then it reads in the latitude degrees, the latitude minutes, and the compass direction (N, S). If the direction is S, then the latitude numbers are flipped to negative numbers. It then reads in the longitude degrees, the longitude minutes, and the compass direction (W, E). If the direction is E, it also flips the longitude numbers to negative ones. This lets us tell the difference between places that are, say, 10 degrees north of the equator and places that are 10 degrees south of it.

Finally, when we have all that information in hand, we call our little vertex constructor, make a vertex, and stick it on the end of the vertices vector. Here is that code from the init_graph_from_file() function in graph.cxx:

```
vertex city(city_name, lat1, lat2, long1, long2);  // constructor call
g.add_vertex(city); // this code just puts this vertex at the end of our vertices array
```

and now our graph has one more city in it. We read this info in for all the cities. That part of the code is done.

But we still need our edges in here. After the nodes are defined, we do this by writing two for loops, each running over all the vertices with an iterator. This is the code for the outer loop

```
vector<vertex>::iterator it1, it2;
for (it1 = g.vertices.begin(); it1 != g.vertices.end(); ++it1)
{
        for (it2 = g.vertices.begin(); it2 != g.vertices.end(); ++it2)
        {
                if (it1 != it2) // no self edges
                {
                        // *it1 dereferences the iterator to get a vertex.
                        // &*it dereferences the iterator to get a vertex
                        // and then passes in its address….
                        g.add_edge(&*it1, &*it2, limit); // add edge if it's <= limit
                }
        }
}
```

The iterator here is like a pointer; you get the thing inside by dereferencing the iterator with * or ->. So inside the loop, you can examine the vertex's city by saying:

```
cout << it->get_city_name() << endl;
```

You can also compare the iterators, which is handy for not making an edge to yourself. Self edges would be silly for our graph problem. See where that gets handled in the for loops above?

We need to do something odd to treat the iterators it1 and it2 like vertex*, even though they look like that should be easy. Subtle point: iterator pointers aren't directly usable this way. Undaunted, we dereference it1 and it2 with * and then pass add_edge their addresses using &:

```
vertex* itvertex = &*it1;
```

so that they once again are proper vertex*.

add_edge() calls great_circle_distance(), which I wrote already, check if the distance is within the limit we can travel in one hop, and add that edge to the graph's edge map if it is. We do this by adding the edge to a vector of neighbor vertex* in your edges map:

```
// push u, a vertex*, on the back of the edge vector for v, another vertex*
edges[v].push_back(u);
```

Remember that v here is the key for the map entry, and the vector of neighboring vertex* for v is the thing the map returns when you index into it with the key.

All clear so far? *hysterical laughter*

Part 2, this week's work:

Most of this code exists already; all you need to write here is the depth-first and breadth-first search codes. does_dfs_path_exist()builds a map of visited vertices and initializes them all to false, builds a map of vertex* predecessors (initialized to nullptr)and also matches your city1 and city2 to vertex* v and u in your graph. We start the search by marking your starting vertex* v as visited, and sticking it on your yet_to_explore deque. Then does_dfs_path_exist() calls depth_first_search().

In depth_first_search, u is the destination city we want to find (we started from city v) and we have our faithful yet_to_explore deque and visited map with us, as well as a map for the path. You need to find the next node to process in the yet_to_explore (which end matters here), then you need to find its neighbors, and put each unvisited neighbor vertex* on the deque, marking it visited as you do. For depth_first_search, you treat the yet_to_explore like a stack (push and pop the same end). You have front() and back(), which tell you the thing at that position; you also have push_front(), pop_back(), pop_front(), and push_back() functions that change the front or the back of the deque. See http://www.cplusplus.com/reference/stl/deque/ for a summary.

For breadth first search, it's the same idea, only the ends of the deque are processed differently.

You will write the code for the breadth first and depth first searches. The comments outline the steps for this; right now you have dumb but happy stubs that always tell you that they have found a path. Both searches manipulate a double ended queue (deque); when it should be a stack, depth-first-search pushes and pops from one end of the queue, and when it should be a queue, breadth-first search pushes on one end and pops from the other. Both these searches are covered in the notes with the slides; you just have to use our yet_to_explore instead of the queue or stack, and our visited map, and our path queue.

Here's the specification for depth-first search:

```
// LOGIC: if there's still stuff on the stack,
// we pop one vertex*, then check it to see if it's the destination
// if yes, we are done (return true)
// if no, we push all its unvisited neighbor vertex* on the stack
// for each neighbor, we store a pointer to the vertex* we came from
// and then return the result of depth first search again.
// If the stack is empty, we give up (return false)
bool graph::depth_first_search(vertex* u, map<vertex*, bool>& visited,
        deque<vertex*>& yet_to_explore, map<vertex*, vertex*>& path)
```

Some hints: given a limit of 1500 km, a path exists from Veracruz, Mexico to Sao Paulo, Brazil. My breadth first search code prints the path here (backwards, from destination to source, but that's ok) as:

Sao Paulo, Brazil
Asuncion, Paraguay
Iquique, Chile
Lima, Peru
Guayaquil, Ecuador
Panama City, Panama
Guatemala City, Guatemala
Veracruz, Mexico

Likewise, at a limit of 4000 km, there's a path from Singapore to Odessa:
Odessa, Ukraine
Teheran, Iran
Bombay, India
Singapore, Singapore

You should not be able to find a shorter path (in terms of hops) at this distance limit using breadth-first search.

Good luck. You all know my email and office hours by now.