Graph algorithm

Assignment 13
Due April 30, 2012
CSC 062: Spring, 2012

Introduction

For our final assignment, you will be playing with some graph algorithms. For the first part, you will create a collection of general graph functions in C++. Your functions will apply to any graph, provided it is presented in the correct form. Then, in the second part you will translate movie rating data from Netflix into graphs and analyze the graphs with your implemented functions.

For this assignment you should make liberal use of the STL structures, like `list`, `map`, `set`, and `vector`. In contrast to the previous assignment, all the pointers should be hidden inside the STL structures. It is unlikely that your code will contain any pointers. You *will* use several iterators, however.

Before getting started, read through the *whole* document, so you understand what resources are available and what you are required to do.

Graph algorithms

You are to write three graph functions corresponding to three graph problems discussed in class: shortest paths, connected components and cycles.

Graphs are specified using an adjacency "map": `map<int,list<int> >`. Each vertex is an integer, and the map provides a list of the neighbors of any vertex. The vertices need not be consecutive integers nor in a particular range, for example the vertices could be 10, 100 and 15.

The three functions you need to implement can be found in the header file `graph_operations.h`, which can be found in `/common/cs/cs062/assignments/assignment13`:

```
list<list<int> >
  grop_connected_components(const map<int, list<int> > &adjMap);

list<int>
  grop_one_cycle(const map<int, list<int> > &adjMap);

map<int, list<int> >
  grop_shortest_paths(int source,
                      const map<int, list<int> > &adjMap);
```

Notice that each function name begins with `grop_`, short for "graph operation." Such prefixing is a common approach in C++ to avoid naming conflicts and to identify all the functions that appear in a particular compilation unit.

Each function takes an adjacency map as an argument. The map is passed by `const` reference to avoid copying and to prevent the map from being corrupted.

Implement these three functions in a file `graph_operations.cpp`. You may write auxiliary functions in the same file, but do *not* change the header file.

`grop_connected_components` uses depth-first search to return the list of connected components. Each component is itself a list. The order of the components, and the order of vertices within a component, is not specified.

`grop_one_cycle` uses depth-first search to return a cycle, if there is one. If the graph is acyclic, the result is an empty list. Otherwise, the list specifies a cycle as a list of three or more vertices that starts and ends with the same vertex. If multiple cycles exist than any cycle may be returned.

`grop_shortest_paths` uses Dijkstra's algorithm and returns a map of shortest paths. Notice that in our case the edge weights are intrinsically 1, but we'd still like you to implement it using Dijkstra's. Suppose that `sp` is the map returned. For each vertex `v` which is reachable from `source`, `sp[v]` is the list of vertices on a path from `v` *back to* `source`. A path in our case consists of a list of vertices and should include both `v` and `source`. The path from `source` to `source` is just the single node `source`. There

may be several shortest paths; your function will provide only one. The vertex `v` will *not* appear in the map `sp` if there is no path between `v` and `source`.

You will also need our `priorityqueue62` class from Assignment 10 to implement Dijkstra's algorithm. I have included a working copy in the starter folder, though, you are welcome to use yours.

Testing

Below are seven easy-to-generate graphs that you can use for testing. I *strongly* encourage you to utilized these methods to test your graphs. Pick the appropriate ones for testing your code. A module of C++ functions that produce these graphs can be found in `graph_examples.cpp` in the starter directory for this assignment. The functions also provide examples for using maps and iterators.

i. **An n-cycle**: The vertices are integers from 0 through $n-1$. The vertices u and v are connected by an edge if $u - v = \pm 1$ or $u - v = \pm(n-1)$. There is one connected component, every shortest path has length at most $n/2$, and there is a unique cycle of length n.

ii. **A complete graph on n vertices**: The vertices are integers from 0 through $n-1$. Every pair of distinct vertices forms an edge. There is one connected component, every shortest path has unit length, and there are many cycles.

iii. **An empty graph on n vertices**: The vertices are integers from 0 through $n-1$. There are no edges. There are n connected components, no paths, and no cycles.

iv. **A heap**: The vertices are integers from 0 through $n-1$. The neighbors of a vertex v are $(v-1)/2$, $2v+1$, and $2v+2$, provided those numbers are in the range for vertices. There is one connected component, the paths are short, and there are no cycles.

v. **A truncated heap**: The vertices are integers from m through $n-1$. The edge relationship is the same as for the heap. There are $n-1-2m$ edges, $m+1$ connected components, and no cycles. The paths, when they exist, are short.

vi. **Equivalence mod k**: The vertices are integers from 0 to $n-1$, where $k \leq n$. The vertices u and v are connected by an edge if $u - v$ is evenly divisible by k. There are k components, and each component is a complete graph.

vii. **Empty and full triangles**: The vertices are integers from 0 to 5. There are three edges, joining vertices 3, 4, and 5. This graph can test whether your cycle function searches *all* the connected components for cycles.

Real data

Now that you have some working methods, we will construct graphs that are much larger than the examples above by using data from the Netflix challenge. The file `movie_reviews.txt` also in the starter directory contains information about movie viewers and their ratings of movies. A typical line in the file looks like this:

```
41:30,4;191,3;197,4;357,3;468,4;954,4;
```

The first number, before the colon, identifies a Netflix customer. Following the colon is a sequence of pairs of numbers that identify the movie and the customer's rating or the movie. In this example, the customer gave movie 30 a rating of 4 and movie 191 a rating of 3. Each line of the file corresponds to one customer. The customer in this example rated only six movies. Some of the lines are much longer.

In a file named `graph_movie.cpp`, write a function that reads the movie data file and stores the data in a structure that maps viewers to lists of movie-rating pairs:

```
map<int,list<pair<int,int> > > read_reviews(string filename);
```

Note that this is different than our expected format for our functions written above, since we have a list of `pairs`. We now want to construct a graph where the vertices are the Netflix customers. Write a function that takes the data structure you just created and two integers, assumed to be "viewers" or keys in the map, and returns `true` or `false`, according to whether the two vertices are adjacent:

```
bool adjacent(int v,
              int w,
              const map<int,list<pair<int,int> > > & reviewMap);
```

The criterion for adjacency can change, and you should experiment with several different alternatives (you will need to provide results for three in the end). An easy one with which to begin is that two viewers are adjacent if there is at least one movie that both viewers have rated. Another, stricter criterion is that the two viewers have three movies in common *and* they have given the same rating to each of the three.

Now, write another function that uses the map derived from the movies file and the `adjacent` function to create a graph that can be studied with the graph functions that you wrote:

```
map<int,list<int> >
  create_graph(const map<int,list<pair<int,int> > > & reviewMap);
```

Notice that the `create_graph` method will call your `adjacent` function to determine if two users are adjacent or not.

Finally, write a `main` function that uses the methods above (including some from the first part) to print the number of connected components and size of each component in the graphs that you create. Run your program with three different notions of adjacency, and create a text file named `results.txt` in which you describe your results. The final version of `graph_movie.cpp` should include your definition of `adjacent` that produces the results that you deem most interesting.

Keep in mind that your program may be tested on a file `movie_reviews.txt` that is different from the one supplied. You can assume that the file will have no blank lines, and each viewer number appears at most once, but some viewers may have rated no movies.

Submission

Submit your three files in a folder with you name, assignment number, etc. Make sure your code follows the specifications above *exactly* and that your files are well-documented files in the usual way with Javadoc comments, etc.

`graph_operations.cpp`
`graph_movie.cpp`
`results.txt`

Grading

criterion	points
connected components	3
cycles	3
shortest paths	3
read reviews	2
adjacent	2
create graph	2
results.txt	4
follows specification	2
appropriate comments (including JavaDoc)	2
style and formatting	1
submitted correctly	1

Hints

- Be careful with your parameter passing. Remember if you pass a class (object) using call-by value, it will copy everything. Any modifications, etc. you make to that object locally in the method will *NOT* appear outside of that method. Most of the time, you'll want to pass classes (objects) using call-by reference or call-by constant reference.
- In addition, if you don't use call-by reference, in some situations you will have run-time/memory issues because of the data copying, for example, if you have a recursive method that passes the graph along.
- It may take a minute or two for your `create_graph` method to generate the final graph. If you want to check that it's working, you can print out something in your loop to see the progress.
- The `istringstream` class in `#include sstream` will be useful for parsing the movie data. Review your use of this class from Lab 13.

A note on the data.

The data in the file `movie_reviews.txt` was extracted from the Netflix Prize sample data set. See `http://www.netflixprize.com` for more details about the prize and the data. We are bound by the conditions stated in `/common/cs/cs062/netflix/README`:

> The data set may be used for any research purposes under the following conditions:
>
> - The user may not state or imply any endorsement from Netflix.
> - The user must acknowledge the use of the data set in publications resulting from the use of the data set, and must send us an electronic or paper copy of those publications.
> - The user may not redistribute the data without separate permission.
> - The user may not use this information for any commercial or revenue-bearing purposes without first obtaining permission from Netflix.

The full set contains 480,189 viewers and 17,770 movies. Our data are comprised of the subset containing the 1495 viewers with the lowest serial numbers and the first 1000 movies. We have no direct way of identifying the viewers, but we can identify the movies. The file that associates numbers to movie titles is `/common/cs/cs062/netflix/movie_titles.txt`.