Lab 10: The Oracle of Bacon

Due: May 3rd at 11:59pm (wink, wink)

Overview

The goal of this lab is to implement a graph using an adjaceny list and use your implementation to build an *Oracle of Bacon*. **This is a group lab, groups of two, but you must work with someone in your lab section.**

Deliverables

Your submission should include requisite code to run your program, but the following files will be the focus of grading:

- `Makefile`
- `README`
- `graph.[h|inl]`
- `edge.[h|inl`
- `graph-algorithm.[h|inl]`
- `test_graph.cpp`
- `oracle.cpp`
- `library/`

All starter code is provided via `update35`.

Submission

You will submit using `handin35`. Be sure to place all relevant files in `cs35/labs/09`.

The Oracle of Bacon

The *Oracle of Bacon* searches a movie database to find a path between Kevin Bacon and other actors using a graph defined by the co-star relationship: two actors are connected by an edge if those two actors co-starred in a movie together. An actor's Bacon number is the length of the shortest path between that actor and Kevin Bacon. For example, the Oracle of Bacon might report that Kevin Bacon is connected to Kurt Russell with a length-2 path:

```
Kurt Russell was in Mean Season, The (1985) with Andy Garcia Andy
Garcia was in Air I Breathe, The (2007) with Kevin Bacon

  Kurt Russell has a Kevin Bacon number of 2
```

Are you a movie buff? Try guess the Bacon Number yourself and test it out. The number of movies you need to report is the link number. Can you find two movie stars with a link number greater than 3? Greater than 5? It is harder than it sounds.

This game extends to other actors. For example, we could ask what the Giant number is for Andre the Giant and any other actor, say Justin Bieber.

```
Justin Bieber was in School Gyrls (2009) with Fred Willard
Fred Willard was in I Could Never Be Your Woman (2007) with Wallace Shawn
Wallace Shawn was in Princess Bride, The (1987) with Andre the Giant

   Justin Bieber has a Andre the Giant number of 3
```

In this lab you will first implement several graph algorithms and then use the Graph and other data structures from this course to respond to queries about movies and actors. Particularly, your oracle should be able to answer the following questions:

- What were all the movies an actor was in?
- What were all the actors in a movie?
- What is the shortest distance between two actors?
- What is the most recent path between two actors?
- What is the average distance from one actor to all other connected actors?

The design and implementation of your oracle will be mostly up to you, and determining the appropriate data structure to solve a problem is the goal.

Starters Abstract As usual, this is a large and involved lab. You should consider the outline below as a guideline for completing the lab, and note that a description of all provided code can be found at the end of this document.

1. Complete the Graph implementation in `graph.inl`
2. Test your graph implementation in `test_graph.cpp`
3. Compete the graph algorithms in `graph-algorithms.inl`
4. Add tests for your graph algorithms in `test_graph.cpp`
5. Complete the Oracle program in `oracle.cpp`
6. (**Challenge**) Build an Oracle of Baseball

Implementing the Graph

You will complete the graph implementation in `graph.inl`. Note that we will be using an Adjacency List implementation of a graph, which is based on a hash table. To start, **copy your hash table implementation** into the library and fix the include statements so that it can be compiled. Here is the template type for the hash table implementation:

```
    HashTable<V,vector<Edge<V,E,W> >* > outgoingEdges;
```

Note that, the table maps a vertex label to a pointer to a vector storing Edge's. Additionally, your graph will maintain a vertex cache structure:

```
   HashTable<V,V*> vertexCache;
```

The vertex cache maps a vertex label (e.g., a string) to the pointer to the pointer to the vertex label as used in the edges. Note that the edge class is defined as follows:

```
template <typename V, typename E, typename W>
class Edge {
  protected:
    E label;   // The name of this edge.
    V* src;    // The source, or head, vertex for this edge.
    V* dest;   // The destination, or tail, vertex for this edge.
    W weight;  // The weight of this edge.
\\..
```

This means you do not need to maintain multiple copies of a vertex label and instead consider just comparing pointers; however, when a user provides some input, say a string "Kevin Bacon", they are provided a duplicate copy of a vertex label that does not have the same pointer value as the one you are using in the graph. You need a way to translate that string to the current pointer value being used in the graph. That is where the `verexCache` comes in, providing an easy mechanism to translate a value to the current pointer in the graph.

When using your graph in the oracle, you should think of actors as the vertices, and edges representing movies that link them together. Thus the label on a edge is the movie title. The weight on the label is the age of the film; for example, a film released in 1995 will have a weight 17 (2013-1995). This will be important for finding the recent path between actors.

Implementing the Graph Algorithms

You are required to complete the following algorithms over the graph. The purpose of each of the functions is self explanatory, and each maps to the questions your oracle must be able to answer. You should feel comfortable expanding these functions with additional helper functions where needed.

```
/**
 * Returns the path found from the vertex src to the vertex dest,
 * using an unweighted breadth-first search of the graph.
 */
template <typename V, typename E, typename W>
vector<V> bfs(V src, V dest, Graph<V,E,W>* g);


/**
 *Return the path found from the vertex src to the vertex dest using
 *Dijsktra's algorithm
 **/
template <typename V, typename E, typename W>
vector<V> dijkstra(V src, V dest, Graph<V,E,W>* g);


/**
 * Returns the number of edges in the shortest path from src to dest.
 */
template <typename V, typename E, typename W>
int getUnweightedPathLength(V src, V dest, Graph<V,E,W>* g);

/**
 * Returns the cost of the shortest path from src to dest.
 */
template <typename V, typename E, typename W>
int getWeightedPathLength(V src, V dest, Graph<V,E,W>* g);


/**
 * Given a vertex src, returns the average distance between src and
 * all other vertices reachable from src.
 */
template <typename V, typename E, typename W>
double averageDistance(V src, Graph<V,E,W>* g);
```

Hints and Tips

You might find the following hints more or less helpful:

- Reading large data files can be slow. Make sure you only read the data file once to build your data structures, and then pass pointers to your data structures if you need to use them in other functions.
- To compute the path between two vertices in the graph, you should use auxiliary data much like computing the maze path in the Lab 5.

- An intuitive but bad way to compute the average distance between a vertex `v` and each other vertex in `v`'s component is:

```
totalDistance = 0.0
numVertices = 0.0
component = Use BFS to find all vertices in v's component
For each vertex u in component:
    totalDistance += distance(v, u)
    numVertices += 1
return totalDistance / numVertices
```

  This is needlessly inefficient because many computations of `distance(v,u)` might each search $O(n)$ other vertices. (Given that, what is the total running time of this pseudocode?) Instead, you should directly compute the distance between `v` and each other vertex during a single breadth-first search of the graph.

- Templates can be very particular about using c-strings versus strings. For example, in your test file you will get an error if you try:

```
Graph <string, string, int>* graph = new Graph <string, string, int>;
graph->insertVertex("Parrish");
//do some more inserts, code is looking good!

//now let's do something interest
bfs("Parrish","Ville", graph); //Compiler error
```

  C++ interprets `"Parrish"`as a c-string and doesn't like that you are trying to send that in for a string. Simply modify your call by casting it as a string explicitly.

Input Files

The data files for this lab are in the `/usr/local/doc` directory, accessible from the CS lab computers. Some of these files are moderately large. **It is strongly recommend that you test your program using the smaller files**, and only use the larger files when you are confident that your program works. The files are:

- `Bacon0Links.txt`: All records for only the movies that Kevin Bacon appeared in (68 records, 2.1KB)
- `Bacon1Links.txt`: All records for Kevin Bacon and his co-stars (38307 records, 1.3MB)
- `BaconModernPopLinks.txt`: All records for actors that have appeared in at least 10 movies since 1970 (995894 records, 36MB)
- `Bacon2Links.txt` : All records for Kevin Bacon, his co-stars, and his co-stars' co-stars (1693497 records, 62MB)
- `BaconAllLinks.txt`: All records except for TV show appearances or direct release to video (4074807 records, 145MB)

A typical line in the file looks like:

```
Kevin   Bacon   A Few Good Men  1992
```

where each field (first name, last name, title, year), is separated by a tab (`"\t"`).

Testing Your Graph and Graph Algorithm

As before, you are required to test your graph and graph algorithms in `test_graph.cpp`. I have provide two sample graphs in `simple_test()` and `swat_test()` for you to advance and use in further testing.

Handling User Input

Your program should handle user input, both from the command line and within the program appropriately. That is, when a user provides a bad input, say he/she forgets to include the database file on the command line, an error should be reported and useful information provided. Similarly, within the program, if a user provides a bad choice or selection in the menu, you should report an error and allow the user to select again. **Your program should never crash due to bad input, and if it cannot proceed, useful information should be reported**.

Error Handling

You should carefully consider how errors should be handled in your program, particularly for your hash table. Throwing an exception is the logical choice; identifying where and when to throw an exception is an important part of this lab.

Grading and Extra Credit

Rubric

This lab will be graded on a 80 point scale:

- (15 Points) Graph Implementation
- (20 Points) Graph Algorithms
- (15 Points) Graph Tests
- (20 Points) Oracle Program
- (10 Points) Coding Style and Memory

Note that there are points dedicated to aspects of your lab that are not part of the solution. Do pay attention to coding style and managing memory, design and lay out your code effectively so that it is readable and easily debugged. Run your program through `valgrind` and plug any memory leaks.

Challenge and Extra Credit

There is no extra credit on this lab. There is one challenge problem.

- (**Challenge**) Extend your code so that it can take in baseball data for the Oracle of Baseball. Similar to the Oracle of Bacon, the Oracle of Baseball will describe the relationships between baseball players and the teams they play for. You can see an example of the Oracle of Baseball here `http://www.baseball-reference.com/oracle/` and dowload baseball data here `http://www.baseball-databank.org/`.

Provided Code

- `Makefile` : The makefile. You may need to edit this file if you add new code to your project.
- `README` : The README file. You will need to edit this file.
- `graph.[h|inl]` : The graph implementation files. You will need to edit the inline file.
- `edge.[h|inl` : The edge implementation files. You will not need to edit these files, but you will need to review them.
- `graph-algorithm.[h|inl]` : The graph implementation files. You will need to edit these files.
- `test_graph.cpp` : The graph testing file. You will need to edit these files.
- `oracle.cpp` : The oracle program. You will need to edit this file.
- `library/` : The library of previously implemented data structures. You do not need to edit this direcrory, but you may copy any previously implemented data structures here if you want access to them.

Compilation

Compilation will occur using `make`. To compile your program

```
bash> make
```

If you add extra functionality contained in other files, you should edit the `Makefile`, but it is not necessary. To compile the test code:

```
bash> make test
```

Which will compile an executable `test_graph` at first, you can add more tests as you develop.

Execution

```
> ./oracle
Usage: ./oracle <movie-file>
Required options:
  <movie-file>:     a movie file to generate the database
```