COMP 322: Assignment 2 - Winter 2014

Due at 11:30pm, Feb 25th 2014

1 Introduction

There are two goals that we would like to attain in this assignment. First, we would like to implement the ideas built up in the first assignment using Standard Library Containers as opposed to pointers. Second, we will use the graph data structure to implement information about a subset of Wikipedia, which we will analyse in later assignments using concepts that we will learn in the second part of the semester.

Specific things that you will learn in this assignment:

- Read from a file using a stream
- Use container classes in the C++ libary: `vector` and `list`. This will provide a brief intro to object oriented programming.
- Use the *algorithms* that come with the C++ library so that you don't need to write everything yourself.

Important reference:

- `http://www.cplusplus.com/reference/stl/vector/`
- `http://www.cplusplus.com/reference/stl/list/`
- `http://www.cplusplus.com/reference/algorithm/`
- `http://www.cplusplus.com/reference/iostream/fstream/`
- The appendix in this document

2 Using graphs to organize Wikis

We started working with and thinking about graph data structures. In this assignment we will use this DS to organize Wikipedia pages. We will analyse a substatial subset of Wikipedia using graph theory.

In particular, we will consider Wiki pages to be vertices in a graph, and there will be an edge between a page and linked pages (i.e. can you jump from one wiki page to another using one mouse-click?). Although computing similarity between pages would require some sort of semantic analysis, we will try to keep it simple and compute a similarity value that is more suited for the purpose of the course: say we want to compute the similarity between "Chicago" and "New York", we will simply count the number of occurences of "Chicago" in the wiki page of "New York", plus the number of occurences of "New York" in the wiki page of "Chicago" (don't be surprised if we will change this in some later assignment!).

To keep data storage and graph analysis at its simplest, each wikipage will be identified with a unique integer ID which will represent the page in a graph where each edge has three pieces of information:

- `origin` : the ID of one of the pages
- `destination` : the ID of the second page
- `weight` : the strength of the connection, as described above.

To access the wiki page itself, we will use a `map<int, Wikipage>`, where `Wikipage` is defined as a `struct` with the following information:

- `ID` : the integer ID, which is also used in the map to get to this item
- `title` : a string representing the title of the page
- `html_location` : the relative location of the file holding the html content of the wiki page
- `txt_location` : the relative location of the file holding the text content of the wiki page

In this assignment you are responsible to implement some of the methods in the provious assignment using STL Containers, as well as the construction and the management of the graph data structure from the raw data on wikipedia pages.

We provide HTML and TXT files for five wikipedia pages. In later assignments we will work with a shorter version of Wikipedia, where all linked pages and pictures will be provided. For the time being, we are only experimenting some preliminary functions with a very small sample.

3 Assignment Requirements

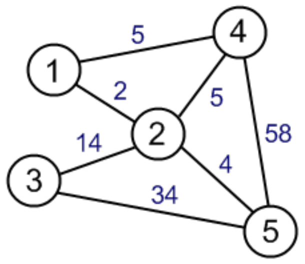
Please add all your code in a file named `wikiGraphStudent.cpp`. Test your code using these and the other files provided: `graphSTL.h` and `simpleWikiTest.cpp`. The TAs will test your code by compiling your submission using the provided `Makefile` and an additional `gradingWikiTest.cpp` (not provided). NOTE: You do not have to use any code from the first homework. `Edge` has been redefined in `graphSTL.h`.

You are responsible for the following (make sure they respect the header definitions in `graphSTL.h`):

1. A method `print` which takes as input a `list<Edge>` and prints all the contents of the linked list. It should print for each element in the linked list the vertices and the weight of the edge.

2. A method `organizeList` which takes as input a `list<Edge>` and an `int` representing the number of vertices in the graph. It goes through the input list of edges and organizes it into separate lists for every vertex. That is, it creates a `vector<list<Edge> >`, where at index i we store a list of all edges that connects vertex i to some other vertex j. Your method should return the vector, after all edges in the initial list has been transfered in the appropriate list of the array mentioned earlier. This method should work exactly like the one in the first homework.

3. A method `printOrganized` which takes as input a graph that has been organized (i.e. output of `organizeList` of type `vector<list<Edge> >`) and prints line by line the neighbours of each vertex in the graph. This time, the information should relate to the Wiki pages that the nodes represent. Also, the method should print the adjacency lists in alphabetical order (Hint: use the STL `sort` algorithm). For the graph given as an example, it should print:



ID association: newyork.ID = 1, chicago.ID = 2, montreal.ID = 3, toronto.ID = 4, miami.ID = 5

```
Page "Chicago" -> New York:2, Toronto:5, Montreal:14, Miami:4
Page "Miami" -> Toronto:58, Chicago:4, Montreal:34
Page "Montreal" -> Chicago:14, Miami:34
Page "New York" -> Toronto:5, Chicago:2
Page "Toronto" -> New York:5, Chicago:5, Miami:58
```

4. A method `buildMap` which takes as input a list of `Wikipage` objects and returns a map that associates every one of the input pages to their corresponding ID. That is `my_map[id]` should be equal to the unique `Wikipage` object `page` that has `page.ID` equal to `id`.

5. A method `countOccurences` which takes as parameters an input file stream and a string, and returns the number of times the given string occurs in the given input stream.

6. A method `createEdge` which takes as parameters two `Wikipage` objects and creates an edge representing their connection. The only tricky piece of information (which requires a bit more work) is the weight that should be given to the `Edge`. To compute this, we count the occurences of each page in the other, and add these counts together (use `countOccurences` on the *text* content of a wiki page).

7. A method `createAllEdges`, which takes as input a list of `Wikipage` objects and returns a list of all possible `Edge` objects that can be created between every pair of pages. Note that for every pair `page_1` and `page_2`, you should create an edge if and only if the weight is greater than 0 (i.e. at least one of them can be found in the content of the other one).

8. A method `saveGraphToFile`, which takes as input an adjacency list representation of a graph, and saves it to a file in the following format: every line should contain three integers separated by white spaces: the first vertex, the second vertex, and the weight of an edge. Note that you should not print an edge twice. For example, you should not have something like this

   ```
   ...
   1 2 2
   ...
   2 1 2
   ...
   ```

   To avoid this, try to print to file only edges with the origin smaler than the destination.

9. A method `readGraphFromFile`, which takes as input a file stream and returns a list of all edges in the file.

4 Appendix

4.1 Using/installing g++

g++ is the GNU C++ compiler. A recent version (4.3.2) is available on most of the Ubuntu lab computers. Our solution was developed using version 4.4.1. For details on how out solution is built, please see the Makefile. Note that to simply use the command `make` you have to add a main method to your `graphStudent.cc` file. If you dont want to write a main method (you should as its the best way to test things!), you can check compile time errors by using the command `make graphAdjList.o`.

If you want to install g++ on your system, you should be able to find instructions around the web. For the Mac, the instructions here look reasonably accurate and up-to-date:

```
http://www.claremontmckenna.edu/pages/faculty/alee/g++/g++_mac.html
```

For PC/Windows systems, there are multiple choices, but the Cygwin environment mimics most of the Linux command line on Windows, so it is a good option:

```
http://www.cygwin.com
```

For Ubuntu (and possibly Debian) users, you can install the free `g++` package with the following command:

```
sudo apt-get install g++
```

If you have trouble, or youre running some other operating system, let us know and well try to help you.

4.2 Container Classes

C++ comes with a standard library. The standard library is used to deal with scenarios that occur quite frequently in practice, such as storing an array of elements or a linkedList of elements. There are several *classes* that are defined in the standard library. The most commonly used of these are `list` (for linked lists), `vector` (for arrays), `set` (you cant add the same element twice), `queue` (provides insertion at one side and deletion from the other side), and `stack` (provided insertion at the same side as deletion). All of these are defined to work on *any* data type and are used in similar ways. For example, a function called `push_back` will always insert an element at the back of any of these data structures (when its defined). The idea is by keeping the function names similar, its easy to program since one isnt constantly trying to remember the various names.

These libraries all use a concept known as *templating* or *generics*. The idea here is that many functions that operate on containers do not depend on the *kind* of data being stored in the container. For example to insert something at the front of an array, one needs to shift every element of the array over and then set the 0th element to be the new one. This is done regardless of the kind of data being stored in the array. So it makes sense to be able to define a *generic* data type that will operate on any kind of data.

Whenever one uses these sorts of *generic* or templated classes, one specifies the kind of data being stored using a bracket notation. For example, the following would create a vector of int and a list of double.

```
vector<int> foo;
list<double> bar;
```

You can also, in some cases, create space at the time of creation for data. For example, writing `vector<int> foo(10)` would create space in memory immediately to store 10 ints. You can add elements to the vector regardless using `push_back` for example, but since the vector is representing an array, it will be necessary to resize the array. In addition, some of the functions defined in the algorithms library will depend on space already being allocated.

Each of these containers are classes. A class in C++ defines a blueprint for what an *object* will look like. A class definition is similar to a struct definition except with the addition that it can do things. One can *do* things on an object using a dot notation and then calling a method defined on an object of that type. For example, to add an element to a vector, one could write:

```
vector<int> foo;
foo.push_back(3); //inserts element 3 at end
//Now foo is a vector containing 3.
foo.push_back(5);
//Now foo is a vector containing 5.
```

These functions are called on data and differ from anything weve seen before in C++, where one would have to pass the variable foo to the method as a parameter. Notice also that these methods return void but are changing foo anyway.

There are many functions defined on all of these data types. The best way to learn about them is by practicing with them and by reading the links above.

4.3 Streams and File I/O

The last thing needed for this assignment is a bit of background on streams. Up until now, we have used the cout << structure in a similar fashion to how one uses the `printf()` function in C or the `System.out.println()` method in Java. However, despite appearing to be the same, cout is actually very different.

`cou` itself is actually an identifier in C++. Its type is an `ostream` (for output stream). A stream is essentially a *channel* from one place to another. For example, there exists a stream that goes from the program you run to *standard output* which is by default directed to the screen. A stream will typically flow in one direction and if you put information in a stream upstream, it will float downstream. For example, if we send information to `cout` it will flow to standard output, which then flows to the screen.

There is a second stream which flows from standard input to your program. Standard input is typically coming from the keyboard, although it can be linked in other ways as well. `cin` is an identifier that can be used to access this stream. The type of `cin` is `istream`.

In C++, streams are handled using the operators `>>` and `<<` . Each of these operators take as input two operands: the stream and the data:

1. `istream >> l-value`
2. `ostream << data`

In the first case, we have `istream >> l-value` which means direct the information from istream into l-value (i.e. a variable).

In the second case we have direct data from *data* to the `ostream`.

So for example, when we write `cout << "Hello world"` we are actually directing to the cout stream the data `Hello world` . If we have a variable `x` and we write `cin >> x` we are directing whatever is currently in the stream `cin` to the variable `x`.

Both of these operators are designed cleverly to *return* the stream on the left side as well. This allows us to put multiple statements together. For example: cout ¡¡ "hello " ¡¡ x; is really (cout ¡¡ "hello") ¡¡ x which means Send to the stream cout the contents hello Evaluate the expression cout ¡¡ "hello" (which will have the type of ostream and evaluate to be cout as well) and send to it the data stored in x To read from a file in C++, you can do the exact same thing as reading from the keyboard, except you must open a stream to a file. In the case of reading from a keyboard, the stream is already created for you. To do this, you should use a data type called ifstream which is defined in the header fstream. This object has the ¿¿ operator defined on it in the same way as cin does. To read from a file, youll need to use the open method, defined on an fstream. Youll also find useful the function fail() which returns a boolean of true or false depending on 5 whether the read was successful. This can be used to detect whether youve reached one past the end of the file. Finally, the close method should be used when you are done with the stream to clean up after yourself. All of these methods are defined on and object of type fstream, so you should use them as one would a container class method. See http://www.cplusplus.com/reference/ iostream/fstream/ and http://www.cplusplus.com/doc/tutorial/files/ for examples of using these.