Computer Science II — Homework 6 — Graphs

Overview

This assignment, originally due Thursday, March 6 at 11:59:59pm, must now be submitted by Friday, March 7 at 11:59:59pm. Solutions submitted after this, but before Saturday, March 8 at 11:59:59pm, will be considered one day late. No solutions submitted later than this will be accepted.

This assignment is worth 100 points toward your homework grade.

As we have learned, linked lists are chains of nodes, with each node having a pointer to its next (successor) node and perhaps to its previous (predecessor) node. A *graph* can be viewed as a generalization of a linked list, where each node can have any number of "edges" linking it to other nodes (instead of just two). Moreover, the linked structure need not be regular, as it is in a linked list. Graphs may be used to represent and study a wide variety of structures, including computing networks and the internet, airline routes, and social relationships. This assignment explores some basic properites of graphs and graph algorithms as a way to expand our understanding of dynamic, linked representations. It is good background for the material and probems covered in *Data Structures and Algorithms.*

Your problem is to build a graph and then perform a number of operations on it. These include determining if it is connected, finding paths, and removing one or more nodes from the graph. These operations will be described below.

The problem and the data are structured so that you can earn significant credit even if you are not able to successfully implement all parts of the assignment. This is discussed more at the end of the assignment.

Building the Graph

The graph is specified quite simply. The input will contain pairs of strings. Each string will be the name of a node — think of the nodes as cities — and each pair will indicate a link or an "edge" in the graph. For example, if

```
Albany  Troy
```

is specified, this means there will be a pointer from the node for `Albany` to the node for `Troy` and there will be a pointer from the node for `Troy` to the node for `Albany`. This is similar to the "next" and "prev" pointers in a doubly-linked list. If this is the first time the program has encountered the string `Albany` or the string `Troy` or both then a node (or two) must be created before the pointers can be assigned.

One issue is that you do not know in advance how many other nodes each node is connected to and therefore you do not know how many pointers each node needs. You solve this by storing a *vector of pointers in each node.*

A second issue is that you must create the nodes and store the pointers to them somewhere. With the implementation of a linked list, we just need to store a pointer to the first node. With a graph, there is no notion of a first node, so you must store all of the node pointers explicitly. One possibility is to create a

```
std::map< std::string, Node* >
```

where `Node` is the node class. There will be one entry in this map for each node of the graph. A second possibility is to create a

```
std::vector< Node* >
```

In either case, you should store the name of the node in the `Node` class. That way, when you follow a pointer from one node to the next, you will be able to determine the name of the node.

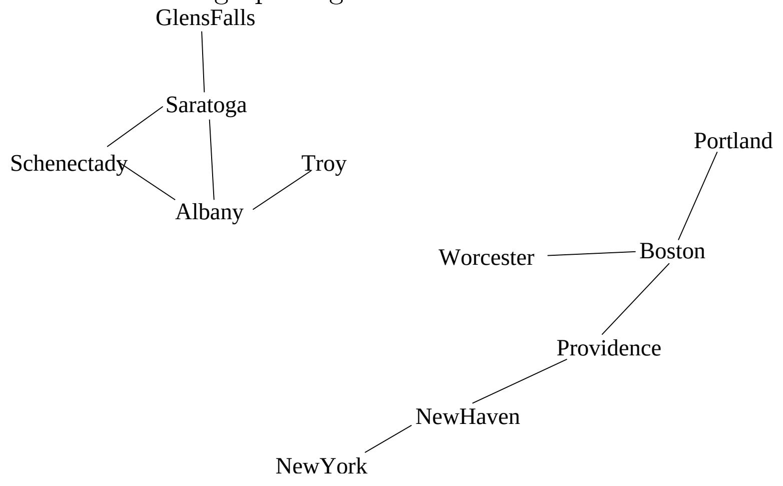
If the input contains a repeated edge, then the repetition should be ignored (although an output error message should be generated).

Example

As an example, if the input file contains

```
Albany Troy
Boston Providence
Worcester Boston
Providence NewHaven
NewHaven NewYork
Schenectady Albany
Albany Saratoga
Boston Portland
Saratoga GlensFalls
Saratoga Schenectady
```

Then a conceptual view of the graph might be



There are several important things to notice about the graph:

- Each node in the figure is represented by just the name of the city. Boxes for pointers are not drawn, and no arrows are indicated on the links. In turning this into a C++ representation, each node must have at least a name and a vector of pointers (3 each for Boston and Albany). Depending on the algorithms that are applied to the graph, each node may need to store additional information.
- The graph, while drawn roughly geographically, really has no geographic information. The only information is in the connections between nodes.
- There are two groups of nodes in the example graph and they have no connections between them. This is an example of a “disconnected” graph.

- The way the input is specified requires that each node in the graph has at least one edge. This is not true in general, but it for the graphs constructed from this input. Once we start making changes to the graph (see below), this may no longer be true.

As an aside, here we are using the term "node" for the basic components of the graph in analogy to the nodes of a linked-list; more often the term "vertex" is used.

Graph Algorithms

Once your program reads and constructs the graph, it should answer several questions about the graph, while allowing the addition and removal of both nodes and edges from the graph. These questions are based on the notion of a "path" in the graph. A path is a sequence of nodes that are linked by edges. For example, the sequence

```
GlenFalls Saratoga Schenectady Albany
```

is a path in the graph. Also,

```
Saratoga Schenectady Albany Saratoga
```

is a path — a special path called a "cycle" because it starts and ends with the same node. For our purposes, repeated nodes (cities) are only allowed at the start and end of a path, and only when looking for a cycle.

One question to be answered is to determine the number of *connected-components* in the graph. A connected component is a maximal set of nodes such that there is a path in the graph between any two nodes in the set — i.e. the set is "connected". The term *maximal* means that no additional nodes can be added to the set while retaining the connected property. For example, the set

```
Boston Portland Providence NewHaven
```

is connected, but not maximal because `Worcester` and `NewYork` can be added.

The second question to be answered is to find shortest paths between nodes in the graph. For example, given the input

```
GlensFalls Albany
```

The cities on the path are

```
GlensFalls Saratoga Albany
```

not

```
GlensFalls Saratoga Schenectady Albany
```

We will not consider the special case that the same city is given as the start and the end. This is the hardest part of the assignment and should be left for last — after you have completed and tested the other parts.

See below for details on the output.

Algorithm: Breadth-First Search

The method for solving these problems is based on a technique called *breadth-first search (BFS).* BFS starts from a single node in the graph, which is immediately marked as "visited" and placed in a list (actually the pointer to the node is placed in the list). The notion of "visited" can be realized by simply storing a `bool` variable with the node. The main loop of BFS removes the first node (actually the pointer to the node) — call it `v` — from the list, and then considers all other nodes `v` is connected to. Any of these nodes that are not "visited" are marked as visited and placed on the *end* of the list. This procedure continues until the list is empty, meaning that all nodes that are connected (by a path) to the initial node have been reached. In some cases — and this is important for path finding — when a node `u` is first reached a special pointer is assigned in that node pointing to the previous node `v` — the one from which `u` is first reached.

Your job will be to implement this algorithm and adapt it to the two search problems described above. This may take some thought, as I have not given all of the details here.

Summary of Input and Output

Your program will be given two input files. The first contains the original graph, with two cities linked by an edge on each input line.

The second, to be read after the graph is initially constructed, specifies one of the following four graph operations on each line of input:

print Print the cities in alphabetical order (which is the order they would naturally be output from the map), with each city name followed by the names of the cities it is connected to via an edge. Here the order should be the order in which the edges were added.

remove-edge city1 city2 Remove the edge linking the two cities, if the edge exists. If the edge does not exist, which includes the possibility that one or both of the cities do not exist, then output the message

```
edge city1 city2 does not exist
```

If the edge does exist then remove it and output

```
edge city1 city2 has been removed
```

This removal could create nodes that have no edges. Do not remove these nodes. They will be their own connected component.

remove-city city-name Remove the node and all of the edges for the given city. Remove the city from the storage of nodes and delete its memory. If the city does not exist, either because it never was part of the graph or because it has already been removed, output a message saying

```
city city-name does not exist
```

otherwise, after your program completes the removal operation output

```
city city-name removed
```

components Output the number of connected components in the graph and the alphabetically-ordered list of cities in each component.

path city1 city2 Find the shortest path between city1 and city2. The output should be

```
path city1 city2:  list-of-cities
```

with the list-of-cities being described above. If one or both of the cities do not exist or if there is no path, the output should be

```
path city1 city2:  <none>
```

In the input, you can assume that the two cities are different.

With all output, follow the example messages and formatting from the course web page as closely as you can.

Suggestions

A file, `hw6_examples.cpp`, is posted on the course web site. It gives an example graph node class and example of code that iterates through a vector of graph node pointers and a list of graph node pointers. Feel free to adapt this code in any way you wish in your solution.

A *substantial fraction of credit* (at least 75 points) may be earned by solutions that correctly create the graph, and correct implement the remove and print functions. Additional credit will be given for solutions that also implement the connected components algorithm. Full credit may be earned only by correctly implementing all functionality.

Submission

Please place all of your source code files and `readme.txt` file in a folder named hw6 (no spaces, no uppercase, not `hw6 submit`, not `H6`). Make sure none of your file names include spaces. Zip your folder prior to submission, and submit it through the course web page.