CSC172 LAB 18

GRAPH PRACTICE

1 Introduction

The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming .

Every student must hand in their own work, but every student must list the name of their lab partner if any on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

2 Graph Practice (as opposed to “Graph Theory”)

In this lab, you will implement a basic graph ADT using the adjacency matrix data structure. We will use integers to identify the nodes in a graph. In order to distinguish between directed and undirected graphs, we will use a boolean value set at construction time.

1. Begin this lab by defining the basic skeleton of the graph ADT. Implement the constructors for the Graph and Edge classes and the basic access methods (isDirected(), vertices(), edges(), and connected()).

```
public class Graph {
    private int vertexCount, edgeCount;
    boolean directed; // false for undirected graphs, true for directed
    private boolean adj[][];

    public Graph(int numVerticies, boolean isDirected) { // your code here }
    public boolean isDirected() { // your code here }
    public int vertices() { // return the number of vertices }
    public int edges() { // return number of edges }
    public void insert(Edge e) { // your code here }
    public void delete(Edge e) { // your code here }
    public boolean connected(int node1, int node2) { //are they connected? }
    public AdjList getAdjList(int vertex) { // your code here }
}

// Of course, we need an edge class
public class Edge {
      public final int v, w; // an edge from v to w
      public Edge(int v, int w) { // your code here }
}
```

2. Implement the `insert(Edge e)` method. The behavior of this method will change based on whether the graph is directed or undirected. You also need to keep track of the edge count in this method. The method should have no effect if the edge already exists.
3. Implement the `delete(Edge e)` method. As with insert, take care to handle directed and undirected graphs appropriately. You should decrement the edge count in this method after deleting the edge. The method should have no effect if the edge does not exists.
4. The trickiest part is the `getAdjList()` method which returns an iterator over the connections for any given vertex. As an exercise we will “roll our own” iterator. The key functionality of any iterator is to pick a starting location, get the next item, and know when we've reached the end. Start by typing out the following interface:

```
interface AdjList {
   int begin()
   int next()
   boolean end()
}
```

What we want is the ability to implement this interface in a class so that we can write the Graph method as:

```
public AdjList getAdjList(int vertex) {
   return new AdjArray(vertex);
}
```

So, we need to make a new private class inside the Graph class that gives us access to the "rows" of the adjacency matrix and processes them properly. The trickiest part is the "next" method. Use and index "i" to scan past false entries in row v of the adjacency matrix. Implement this private helper class according to the skeleton class provided below:

```
private class AdjArray implements AdjList {
   private int v; // what vertex we are interested in
   private int i; // so we can keep track of where we are

   public AdjArray(int v) {
      // write the code for the constructors
      // save the value of the vertex passed in
      // (that will be where the iterator starts)
      // start the "i" counter at negative one
   }
   public int next() { // perhaps the trickiest method
      // use a for loop to advance the value of "i"
      // "for (++i; i < vertices(); i++)"
      // and search the appropriate row return the index
      // of the next true value found
      // "if (connected(v,i) == true) return i;"
      // if the loop completes without finding anything return -1
   }
   public int begin() {
      // reset "i" back to negative one
      // return the value of a call to "next"
    }
    public boolen end() {
      // if "i" is less than the number of vertices return false
    }
}
```

5. If you implemented the iterator properly, then you should be able to include the following method to print out the representation of the graph:

```
public void show () {
    for (int s = 0; s < vertices(); s++) {
        System.out.print(s + ": ");
        AdjList A = getAdjList(s);
        for (int t = A.begin(); !A.end(); t = A.next()) // use of iterator
            System.out.print(t + " ");
        System.out.println();
    }
}
```

6. Write a main text method to build two graphs and print them out using the show method above. Yes, you have to use the method above, this lab is about graphs, but it is also about iterators. Show your code working for the graphs found in figure 9.10 (directed) and figure 9.62 (undirected) of the Weiss text book.

3 Hand In

Hand in the source code from this lab at the appropriate location on the BlackBoard system at my.rochester.edu. You should hand in a single zip file (compressed archive) containing your source code, README, and OUTPUT files, as described below.

1. A plain text file named README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.), and one sentence explaining the contents of all the other files you hand in.
2. Several Java source code files representing the work accomplished for this lab. All source code files should contain author and partner identification in the comments at the top of the file.
3. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line.

4 Grading

Each section (1-6) accounts for 15% of the lab grade (total 90%)

README file counts for the remaining 10%