
CSC172 LAB

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 his own work, but every student must list the name of the 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 (`directed`, `V`, `E`, and `edge`).

```
public class Graph{
    private int Vcnt, Ecnt; // number of vertices and edges
```

```

    boolean digraph ; // false for undirected graphs
    private boolean adj[][];

    public Graph(int numVertices, boolean isDirgraph)
    public boolean directed()
    public int V() // return number of vertices
    public int E() // return number of edges
    public void insert(Edge e)
    public void delete(Edge e)
    public boolean edge(int node1, int node2) //are they connected?
    public AdjList getAdjList(int vertex)
}

//of course, we need an edge class
class Edge {
    int v, w; // and edge from v to w
    public Edge(int v, int w)
}

```

2. Implement the `insert(Edge e)` method. This would be nearly trivial if it were not for the digraph boolean. You need different settings based on the state of this variable. 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. This would be nearly trivial if it were not for the digraph boolean. You need different settings based on the state of this variable. You also need to keep track of the edge count in this method. The method should have no effect if the edge does not exist.
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 start us up, to end us, and to provide us with a “getnext” capability. Let's use the basic skeleton:

```

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 an index “i” to scan past false entries in row v of the adjacency matrix. Implement this private helper class as follows:

```

private class AdjArray implements AdjList {
    private int v; // what vertex we are interested in
    private int i; // so we can keep track of where
    AdjArray(int v ) {
        // write the code for the constructors
        // save the value of the vertex passed in
        // 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 < V();i++)”
        // and search the appropriate row return the index
        // of the next true value found
        // “if (edge(v,i) == true) return i;”
        // if the loop completes without finding anything return -1
    }
    public int beg() {
        // reset “i” back to negative one
        // return a the value of a call to “next”
    }
    public boolean 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 static method and have it work to print out the representation of the graph:

```

static void show (Graph G) {
    for (int s = 0; s < G.V() ; s++) {
        System.out.print(s + “: “);
        AdjList A = G.getAdjList(s) ;
    }
}

```

```
        for (int t = A.beg(); !A.end(); t = A.next()) {
            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 (fix any bugs due to typos, of course). 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 compressed/archived (i.e. “zipped”) file that contains the following.) You will need to include the images of your plotted results.

1. A 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.).
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. It is expected that you will have a file for the test program class.
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

172/grading.html

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

(README file counts for 10%)