```
/*
Implement depth-first search on undirected graphs using the adjacency-list
representation
and use it to determine the number of connected components in an undirected graph.
Programed by Joseph Crandall
last modifed 10/26/15
*/
import edu.gwu.algtest.*;
import edu.gwu.util.*;
import edu.gwu.geometry.*;
import java.util.*;
//import java.lang.*;

public class UndirectedDepthFirstAdjList implements UndirectedGraphSearchAlgorithm{

        public static final String ANSI_RESET = "\u001B[0m";
        public static final String ANSI_YELLOW = "\u001B[33m";
        public static final String ANSI_RED = "\u001B[31m";
        public static final String ANSI_GREEN = "\u001B[32m";

        public static void printYellow(String args){
                System.out.println(ANSI_YELLOW+args+ANSI_RESET); // note: "\n" in
front of args
        }
        public static void printRed(String args){
        System.out.println(ANSI_RED+"\n"+args+ANSI_RESET); // note: "\n" in front of
args
        }
        public static void printGreen(String args){
        System.out.print(ANSI_GREEN+" "+args+ANSI_RESET); // note: "\n" in front of
args
        }


        int numVertices;              // Number of vertices, given as input.
        int numEdges;                 // We keep track of the number of edges.
        boolean isWeighted;             // Is this a weighted graph?
        LinkedList<GraphEdge>[] adjList;
        int[] visitOrder;
        int complCount;
        int visitCount;
        int[] complOrder;
        int numConnectedComponents;
        int[] compLabels;

        // The list (of lists): one list for each vertex.
    // We are using java.util.LinkedList as our linked list.
```

```
    // This will store instances of GraphEdge.

        // We will test new insertions to see if they exist. Because LinkedList performs
        // ".equals()" testing for containment, we need an instance (and only one) of
GraphEdge
        // for such testing.

        //GraphEdge testEdge = new GraphEdge (-1, -1, 0);


        public void initialize (int numVertices, boolean isWeighted){


            // Adjacency-list representation
            adjList = new LinkedList [numVertices];
            for (int i=0; i < numVertices; i++){
                  adjList[i] = new LinkedList ();
            }
            // Initlaize all the global variables
            this.numVertices = numVertices;
            this.isWeighted = isWeighted;
          visitOrder = new int[numVertices];
          complOrder = new int[numVertices];
          compLabels = new int[numVertices];
          numEdges = 0;
          complCount = -1;
          visitCount = -1;
          numConnectedComponents = 0;
          numEdges = 0;
          for(int i = 0; i < visitOrder.length; i++){
                        visitOrder[i] = -1;
                    complOrder[i] = -1;
                    compLabels[i] = -1;
          }
          }

          // Insert a given input edge.
          public void insertUndirectedEdge (int startVertex, int endVertex, double weight){
            // if (! isWeighted) {
            //   weight = 1.0;
          // }

            // Adj-list representation: see if the edge is already there.
            //testEdge.startVertex = startVertex;
            //testEdge.endVertex = endVertex;

            // Exploit the methods in java.util.LinkedList.
```

```
    // if (adjList[startVertex].contains (testEdge)) {
            //  // ... report error ...
            //  return;
    // }

    // It's undirected, so add to both vertex lists.
    GraphEdge e = new GraphEdge (startVertex, endVertex, 1.0);
    adjList[startVertex].addLast (e);

    // We wouldn't have this for directed graphs:
    GraphEdge e2 = new GraphEdge (endVertex, startVertex, 1.0);
    adjList[endVertex].addLast (e2);

    numEdges ++;

    //printArrayOfLinkedLists();
}

//how to go about printing
public void printArrayOfLinkedLists(){
        for(int i=0;i<adjList.length; i++){
                ListIterator<GraphEdge> itr = adjList[i].listIterator();
                while(itr.hasNext()){
                        printGreen(itr.next().toString());
                }
        }
}



        // for(int i=0;i<adjList.size(); i++){

 //          System.out.print("index "+i+" in adjacency list");
 //          List l = adjList[i].adjList;
 //          for(int q=0;q<adjList.get(g).size(); q++){
 //              LinkedList ls = new LinkedList();
 //              ls = adjList.get(g);
 //              System.out.println(ls.get(q));
 //          }
        // }

public int[] depthFirstVisitOrder(){
        //all other methods use this method
        int touch = 0;
        for (int i = 0; i < visitOrder.length; i++){
                if (visitOrder[i] < 0){
                        this.numConnectedComponents++;
```

```
                        this.depthFirstRecursive(i, touch);
                        touch++;
                    }
                }
                return this.visitOrder;
        }

        public void depthFirstRecursive(int index, int touch){
                //visit count is incremented first
                this.visitCount++;
                //update the visit order array with the count number
                this.visitOrder[index] = this.visitCount;
                //update the community number
                this.compLabels[index] = touch;
                if(adjList[index]!=null){
                        ListIterator<GraphEdge> itr = adjList[index].listIterator();
                        while(itr.hasNext()){
                                GraphEdge e = itr.next();
                                if(this.visitOrder[e.endVertex] < 0){
                                        depthFirstRecursive(e.endVertex, touch);
                                }
                        }
                }
                //update the completion order at the end of the recursive call
                complCount++;
                complOrder[index] = complCount;
        }

        public int[] depthFirstCompletionOrder(){
                this.depthFirstVisitOrder();
        return this.complOrder;
}
        public int[] componentLabels(){
                this.depthFirstVisitOrder();
        return this.compLabels;
        }
        public int numConnectedComponents(){
                this.depthFirstVisitOrder();
        return numConnectedComponents;
        }
```

```
        //empty implementation of unused UndirectedGraphSearchAlgorithm methods
        public GraphEdge[] articulationEdges(){
                GraphEdge edge = new GraphEdge(1,0,1);
                GraphEdge edgeArray[] = new GraphEdge[]{edge};
                return(edgeArray);
        }
        public int[] articulationVertices(){
                int[] myIntArray = new int[]{1,2,3};
                return (myIntArray);
        }
        public int[] breadthFirstVisitOrder(){
                int[] myIntArray = new int[]{1,2,3};
                return (myIntArray);
        }
        public boolean existsCycle(){
                return(false);
        }
        public boolean existsOddCycle(){
                return(false);
        }


        //algorithm interface
        //edu.gwu.algtest.Algorithm
        //get name
        public java.lang.String getName(){
                return "Joseph Crandall's implementation of UndirectedDepthFirstAdjList";
        }
        //get property extractor
        public void setPropertyExtractor(int algID,edu.gwu.util.PropertyExtractor prop){
        //empty implementations, method definition empty body
        }


        public static boolean randomCoinFlip (double p){
  if (UniformRandom.uniform() < p){
        return true;
  }
  else{
        return false;
  }
}
```

```
public static void main(String []args){
        System.out.println("main is running");

        UndirectedDepthFirstAdjList DFS = new UndirectedDepthFirstAdjList();
        //intialize and add edges
        DFS.initialize(4,false);
        DFS.insertUndirectedEdge(0,1,1);
        DFS.insertUndirectedEdge(2,3,1);
        DFS.insertUndirectedEdge(0,3,1);
        DFS.printArrayOfLinkedLists();
        int[] a = DFS.depthFirstVisitOrder();
System.out.print("Visit Order: ");
for(int i = 0; i < a.length; i++){
System.out.print(a[i] + " ");
}
System.out.println();
a = DFS.depthFirstCompletionOrder();
System.out.print("Completion Order: ");
   for(int i = 0; i < a.length; i++){
        System.out.print(a[i] + " ");
   }
System.out.println();
int x = DFS.numConnectedComponents();
System.out.println("Connected components: " + x);

//PART 3 OF EXERCISE4

//V is the number of vertices in a graph, for this probelm run from V = 10 and V =
20

//turn on the number of vertices for the test
final int V = 10;
//final int V = 20;

/*N is number of trials used to determine the overal average number of
connected components in a graph created with a given probability P
        For this test, number of trials used 1000,
        and probability is {0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6,0.7, 0.8, 0.9, 1}
*/
final int N = 1000;
final double P = 0.05;
double pmax = 1;
//while loop through different p values

double p = 0.01;
```

```
        while (p < 0.095){
                //initalize the sum of all averages to be accumulated over N trials
                double total_sum_average = 0;

                for(int l = 0; l < N; l ++){
                        //initialize the sum of componets accumulated over a single trial,
and the average for that trial
                        int sum = 0;
                        double average = 0;
                        for(int k = 0; k < V; k++){
                                DFS.initialize(V,false);

                                //flip the coin for every pair of vertices, if true, add edge
                                for(int i = 0 ; i < V - 1; i ++){
                                        for(int j = i+1; j < V; j++){
                                                if(randomCoinFlip(p)){
                                                        DFS.insertUndirectedEdge(i,j,0);
                                                }
                                        }
                                }
                                //update number of connected componets and the sum
                                x = DFS.numConnectedComponents();
                                sum = sum + x;
                        }
                        //calclate the average for the given trial and add it to the sum of all
averages
                        average = (double)sum/V;
                        //printYellow("Average for " + V + " is " + average);
                        total_sum_average = total_sum_average + average;
                }
                //print average number of componets for probability P
                printRed("For " + N + " trials l "+ V +" verticies l probability p = " + p + ", l
overall average number of componets is " + (double)total_sum_average/N);

           p = p + 0.01;
                }
        }
}
```