Lab walk-through #10-1

Topic **Graph Algorithms**

1. Introduction

Like so many of the other problem domains that we have studied, the algorithmic investigation of graphs is relatively recent. Pairwise connections between items play a critical role in a vast array of computational applications. You can answer various questions about pathing and relationships between the various items using graphs.

2. Lab Objectives

By the end of this lab the student will learn how to implement and study various graphing algorithms.

3. Lab Setup

Before beginning this lab, you should have:

1. The Eclipse and Java Runtime Environment on your computer
2. Completed the previous searching algorithms lab and its practice questions
3. Read Chapter 4 in your textbook

4. Lab Exercise

Graph algorithms aim to solve problems in a wide variety of important applications whose solution we could not even contemplate without good algorithmic technology. Their application to complex real world problems make graph algorithms very popular but challenging.

Graph algorithms consist of two kinds of information: items(vertices) and connections(edges). Vertices are connected together using edges and one can traverse the graph by moving from one vertex to another only using edges. Here are examples of items and connections that are used in various applications:

application	item	connection
map	intersection	road
web content	page	link
circuit	device	wire
schedule	job	constraint
commerce	customer	transaction
matching	student	application
computer network	site	connection
software	method	call
social network	person	friendship

Typical graph applications

A **graph** is a set of **vertices** and a collection of **edges** that each connects a pair of vertices. A **path** in a graph is a sequence of vertices connected by edges. A **simple path** is one with no repeated vertices. A **cycle** is a path with at least one edge whose first and last vertices are the same.

When there is an edge connecting two vertices, we say that the vertices are **adjacent** to one another and that the edge is **incident** to both vertices. The **degree** of a vertex is the number of edges incident to it. A **sub-graph** is a subset of a graph's edges (and associated vertices) that constitutes a graph.

A graph is **connected** if there is a path from every vertex to every other vertex in the graph. A graph that is **not connected** consists of a set of **connected components**, which are maximal connected sub-graphs.

An **acyclic graph** is a graph with no cycles. A **tree** is an acyclic connected graph. A disjoint set of trees is called a **forest**. A **spanning tree** of a connected graph is a sub-graph that contains all of that graph's vertices and is a single tree.

The **density** of a graph is the proportion of possible pairs of vertices that are connected by edges. A **sparse** graph has relatively few of the possible edges present; a **dense** graph has relatively few of the possible edges missing. A **bipartite graph** is a graph whose vertices we can divide into two sets such that all edges connect a vertex in one set with a vertex in the other set.

Section 1: APIs for undirected graph

Our starting point in developing graph-processing algorithms is an API that defines the fundamental graph operations. This API contains two constructors, methods to return the number of vertices and edges, and a method to add an edge.

public class	Graph	
	Graph(int V)	*create a V-vertex graph with no edges*
	Graph(In in)	*read a graph from input stream* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge* v-w *to this graph*
Iterable<Integer>	adj(int v)	*vertices adjacent to* v
String	toString()	*string representation*

API for an undirected graph

Task 1: Constructors for Graph

You will now implement these methods to create your own Graph implementation. Create an eclipse project called cas2xb3_lab10 and a class called Graph.java with the above API. For a start, implement the two constructors first.

There are 2 types of constructors shown above. The first one takes a single Vertex V (in the form of an integer) and creates a one-vertex graph with no edges. The second constructor takes input from an input stream. In your textbook they use their library called In.java. You can use this library or you can use a file input stream or System.in.

Next we need a graph implementation that will allow us to iterate through the vertices and edges of the graph appropriately. We would need a data structure that has the following requirements:

1. We must have the space to accommodate the types of graphs that we are likely to encounter in applications.
2. We want to develop time-efficient implementations of Graph instance methods—the basic methods that we need to develop graph-processing clients.

It turns out that the most efficient data structure for a variety of applications as well as simplicity, is an **Adjacency List** data structure, where we keep track of all the vertices adjacent to each vertex on a linked list that is associated with that vertex.

We maintain an array of lists so that, given a vertex, we can immediately access its list. To implement lists, we use our Bag ADT (from section 1.3 in your textbook) with a linked-list implementation, so that we can add new edges in constant time and iterate through adjacent vertices in constant time per adjacent vertex.

To add an edge connecting v and w, we add w to v's adjacency list and v to w's adjacency list. Thus, each edge appears **twice** in the data structure. This Graph implementation achieves the following performance characteristics:

1. Space usage proportional to V + E
2. Constant time to add an edge
3. Time proportional to the degree of v to iterate through vertices adjacent to v (constant time per adjacent vertex processed)

Note: It is important to realize that the order in which edges are added to the graph determines the order in which vertices appear in the array of adjacency lists built by Graph. Many different arrays of adjacency lists can represent the same graph. These differences do not matter for many applications.

However to accommodate this representation, we would add an extra attribute variable called adj:

```
private Bag<Integer>[] adj;
```

Then, we have to add extra code in our constructors to maintain this adjacency matrix, as shown below:

```
public Graph(int V){
        this.V = V; this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V]; // Create array of lists.

        for (int v = 0; v < V; v++) // Initialize all lists
            adj[v] = new Bag<Integer>(); // to empty.
    }
```

```
public Graph(In in){
        this(in.readInt()); // Read V and construct this graph.
        int E = in.readInt(); // Read E.

        for (int i = 0; i < E; i++)
        { // Add an edge.
            int v = in.readInt(); // Read a vertex,
            int w = in.readInt(); // read another vertex,
            addEdge(v, w); // and add edge connecting them.
        }
    }
```

Now we implement the rest of the API as follows:

```
    public void addEdge(int v, int w){
        adj[v].add(w); // Add w to v's list.
        adj[w].add(v); // Add v to w's list.
        E++;
    }

    public Iterable<Integer> adj(int v)
    { return adj[v]; }
```

This is the basic implementation of the Graph API and will allow you to construct a simple graph but not do anything with it. This is quite pointless as far as algorithms go.

Section 2: Example of Graph client code

To use the graph to perform some operations or get some details about its structure, we will need more functions that can process the information in the data structure. Let's see what types of tasks we can accomplish with the graph implementation.

Task 2: Compute the degree of V

How do we calculate the number of edges connected to a particular vertex V? Add the following code to your implementation.

```
    public static int degree(Graph G, int v){
        int degree = 0;
        for (int w : G.adj(v)) degree++;
        return degree;
    }
```

Task3: *Compute the maximum degree*

How do we calculate the maximum degree of a graph G? That is, what is the degree of a vertex in the graph with the most edges connected to it? Add the following code to your implementation.

```
public static int maxDegree(Graph G){
      int max = 0;
      for (int v = 0; v < G.V(); v++)
      if (degree(G, v) > max)
      max = degree(G, v);
      return max;
}
```

Can you think of another way to implement this method?

Task 4: Compute the average degree

Perhaps we would like to calculate the average degree of a graph. Add the following code:

```
public static int avgDegree(Graph G)
{ return 2 * G.E() / G.V(); }
```

Task 5: Count self-loops

This is a very popular problem for many graph algorithms in some applications. A self-loop is an edge that is connected to the same vertex on both ends. Add the following code to your implementation to count these loops.

```
public static int numberOfSelfLoops(Graph G){
      int count = 0;
      for (int v = 0; v < G.V(); v++)
      for (int w : G.adj(v))
      if (v == w) count++;
      return count/2; // each edge counted twice
}
```

Task 6: A string representation of the graph's adjacency lists (instance method in Graph)

Naturally, we would want to get an idea of what the graph looks like, and one way of doing this is printing out the adjacency list. Add this method to your code to include this feature.

```
public String toString(){
      String s = V + " vertices, " + E + " edges\n";
      for (int v = 0; v < V; v++)
      {
            s += v + ": ";
            for (int w : this.adj(v))
            s += w + " ";
            s += "\n";
      }
      return s;
}
```

You can add more methods that might be necessary according to the application you are designing for. Some examples could be:

- Adding a vertex

- Deleting a vertex
- Deleting an edge
- Checking whether the graph contains the edge v-w

To implement these methods, a Bag ADT would not be ideal. It would be better to use a ST (Symbol table) or SET to accomplish this. However, we will not use these implementations. They complicate the implementation and have a performance penalty of log V in some methods. It does not add anything to your understanding of Graphs, so we will skip it for the moment. Adding some other crucial methods such as removing self-loops or a parallel edge is not a difficult task in our implementation.

Section 3: Design pattern for graph processing

Ideally, it is better to separate the client code from the implementation of the graph. We want to decouple our implementation from Graph representation for this reason.

To do so, we develop, for each given task, a **task-specific class** so that clients can create objects to perform the task. Generally, the constructor does some pre-processing to build data structures so as to efficiently respond to client queries. A typical client program builds a graph, passes that graph to an algorithm implementation class (as argument to a constructor), and then calls client query methods to learn various properties of the graph.

As an example, let us look at the following search graph processing API:

public class	Search	
	Search(Graph G, int s)	*find vertices connected to a source vertex s*
boolean	marked(int v)	*is v connected to s?*
int	count()	*how many vertices are connected to s?*

Graph-processing API (warmup)

Source is the vertex provided as an argument to the constructor from the other vertices in the graph. In this API, the job of the constructor is to find the vertices in the graph that are connected to the source. Then client code calls the instance methods marked() and count() to learn characteristics of the graph.

And to test our Search class, we can use a TestSearch client that looks like this:

```
public static void main(String[] args)
{
            Graph G = new Graph(new In(args[0]));
            int s = Integer.parseInt(args[1]);
            Search search = new Search(G, s);

            for (int v = 0; v < G.V(); v++)
                  if (search.marked(v))
                        StdOut.print(v + " ");
            StdOut.println();

            if (search.count() != G.V())
                  StdOut.print("NOT ");
```

```
            StdOut.println("connected");
      }
}
```

When carrying out tasks on your Graph data structures, remember to always decouple your client code from your implementation. This makes your testing as well as your implementation clearer.

Section 4: Depth First Search

Although much of the above methods are useful in telling us about the graphs we construct, the most common interest we have in the data structure is traversing the graph. This tells us some important information about the structure of the graph itself, and makes it most useful to the problems being solved.

One way of doing this might sound familiar to you. Depth First Search allows us to systematically examine each edge and vertex in a graph to learn some of its properties. One trick would be to think of the graph like a maze, this analogy is dealt with in your textbook on page 530. But here are the basics to traversing a graph using DFS:
To search a graph, invoke a recursive method that visits vertices.
To visit a vertex:

1. Mark it as having been visited.
2. Visit (recursively) all the vertices that are adjacent to it and that have not yet been marked.

Task 7: Implementing DFS on a graph

Study the implementation of DFS below and add it to your code. It maintains an array of boolean values to mark all of the vertices that are connected to the source. The recursive method marks the given vertex and calls itself for any unmarked vertices on its adjacency list. If the graph is connected, every adjacency-list entry is checked.

```
public class DepthFirstSearch
{
      private boolean[] marked;
      private int count;

      public DepthFirstSearch(Graph G, int s){
            marked = new boolean[G.V()];
            dfs(G, s);
      }

      private void dfs(Graph G, int v){
            marked[v] = true;
            count++;
            for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
      }

      public boolean marked(int w) { return marked[w]; }
```

```
    public int count() { return count; }
}
```

What is the order of growth of the traversal?

Task 8: Finding paths

One of the most classical problems in graph algorithms is the following. Given a graph and a source vertex s, support queries of the form:

Is there a path from s to a given target vertex v? If so, find such a path.

Use the following API to implement this. Remember to create a new class called Paths first and extend your `DepthFirstSearch` class from the previous task. Similar code has been highlighted in gray.

public class Paths		
	Paths(Graph G, int s)	*find paths in G from source s*
boolean	hasPathTo(int v)	*is there a path from s to v?*
Iterable<Integer>	pathTo(int v)	*path from s to v; null if no such path*

API for paths implementations

The constructor takes a source vertex s as argument and computes paths from s to each vertex connected to s. After creating a Paths object for a source s, the client can use the instance method `pathTo()` to iterate through the vertices on a path from s to any vertex connected to s. For the moment, we accept any path; later, we shall develop implementations that find paths having certain properties.

Add the following code to your newly made class.

```
public class Paths{
    private boolean[] marked;//Has dfs() been called for this vertex?
    private int[] edgeTo; //last vertex on known path to this vertex
    private final int s; //source

    public DepthFirstPaths(Graph G, int s){
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        this.s = s;
        dfs(G, s);
    }

    private void dfs(Graph G, int v){
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]){
                edgeTo[w] = v;
                dfs(G, w);
            }
    }
```

```
	public boolean hasPathTo(int v)
	{ return marked[v]; }

	public Iterable<Integer> pathTo(int v){
		if (!hasPathTo(v)) return null;

		Stack<Integer> path = new Stack<Integer>();
		for (int x = v; x != s; x = edgeTo[x])
			path.push(x);

		path.push(s);
		return path;
	}
}
```

To save known paths to each vertex, this code maintains a vertex-indexed array edgeTo[] such that edgeTo[w] = v means that v-w was the edge used to access w for the first time. The edgeTo[] array is a parent-link representation of a tree rooted at s that contains all the vertices connected to s.

You can test your implementation using the test client on page 535 of your textbook.

Section 5: Breadth First Search

Of course finding if two vertices are connected or what path we could take is good information. But what if we wanted to get the shortest possible path to a vertex v? Here we would need a different sort of search – breadth first search. Convince yourself why.

To find a shortest path from s to v, we start at s and check for v among all the vertices that we can reach by following one edge, and then we check for v among all the vertices that we can reach from s by following two edges, and so forth. In BFS, we want to explore the vertices in order of their distance from the source. It turns out that this order is easily arranged: use a (FIFO) queue instead of a (LIFO) stack (as was used for DFS).

Task 9: Implement BFS

Create a class called BreadthFirstPaths and add the following code to your class:

```
public class BreadthFirstPaths
{
	private boolean[] marked;//Is a shortest path to vertex known?
	private int[] edgeTo; // last vertex on known path to this vertex
	private final int s; // source

	public BreadthFirstPaths(Graph G, int s){
		marked = new boolean[G.V()];
		edgeTo = new int[G.V()];
		this.s = s;
		bfs(G, s);
	}
```

```
    private void bfs(Graph G, int s){
        Queue<Integer> queue = new Queue<Integer>();
        marked[s] = true; // Mark the source
        queue.enqueue(s); // and put it on the queue.
        while (!q.isEmpty()){
            int v = queue.dequeue();//Remove next vertex frm queue
            for (int w : G.adj(v))
            if (!marked[w]){ //For every unmarked adjacent vertex,
                edgeTo[w] = v;//save last edge on shortest path,
                marked[w] = true;//mark it because path is known,
                queue.enqueue(w);//and add it to the queue.
            }
        }
    }
    public boolean hasPathTo(int v){ return marked[v]; }
    public Iterable<Integer> pathTo(int v){}// Same as for DFS
}
```

The bfs() method marks all vertices connected to s, so clients can use hasPathTo() to determine whether a given vertex v is connected to s and pathTo() to get a path from s to v with the property that no other such path from s to v has fewer edges. Notice that it is not a recursive algorithm.

BFS is based on maintaining a queue of all vertices that have been marked but whose adjacency lists have not been checked. We put the source vertex on the queue, and then perform the following steps until the queue is empty:

1. Take the next vertex v from the queue and mark it.
2. Put onto the queue all unmarked vertices that are adjacent to v.

What would be the order of growth of BFS? Look at page 541 for hints.

Section 6: Finding connected components in a graph using DFS

Connectivity. Given a graph, support queries of the form:

Are two given vertices connected? And how many connected components does the graph have?

Our next direct application of depth-first search is to find the connected components of a graph.

public class	CC	
	CC(Graph G)	*preprocessing constructor*
boolean	connected(int v, int w)	*are* v *and* w *connected?*
int	count()	*number of connected components*
int	id(int v)	*component identifier for* v (*between* 0 *and* count()-1)

API for connected components

The id() method is for client use in indexing an array by component, as in the test client below, which reads a graph and then prints its number of connected components and then the vertices in each component, one component per line. To do so, it builds an array of Bag objects, and then uses each vertex's component identifier as an index into this array, to add the vertex to the appropriate Bag.

Task 10: Implement the CC API

Do this by extending the DepthFirstSearch code you implemented earlier. Create a class called CC.java and add the following code in it.

```
public class CC{
	private boolean[] marked;
	private int[] id;
	private int count;

	public CC(Graph G){
		marked = new boolean[G.V()];
		id = new int[G.V()];
		for (int s = 0; s < G.V(); s++)
		if (!marked[s])
		{
		dfs(G, s);
		count++;
		}
	}
	private void dfs(Graph G, int v){
		marked[v] = true;
		id[v] = count;
		for (int w : G.adj(v))
		if (!marked[w])
		dfs(G, w);
	}
	public boolean connected(int v, int w){ return id[v] == id[w]; }
	public int id(int v) { return id[v]; }
	public int count() { return count; }
}
```

The implementation CC uses our marked[] array to find a vertex to serve as the starting point for a depthfirst search in each component. The first call to the recursive DFS is for vertex 0— it marks all vertices connected to 0. Then the for loop in the constructor looks for an unmarked vertex and calls the recursive dfs() to mark all vertices connected to that vertex. Moreover, it maintains a vertex-indexed array id[] that associates the same int value to every vertex in each component.

There are more problems we can solve with the use of DFS:

1. Cycle detection. Support this query: Is a given graph acyclic?
2. Two-colourability. Support this query: Can the vertices of a given graph be assigned one of two colours in such a way that no edge connects vertices of the same colour? Which is equivalent to this question: Is the graph bipartite?

6. Further Practice Problems

For further practice, here are some suggested questions from your textbook:
4.1.3, 4.1.4, 4.1.5, 4.1.8, 4.1.13, 4.1.16, 4.1.17, 4.1.18