About Lab 9

In Lab 9 we will play the "Kevin Bacon game". The Internet Movie Database makes available files with a very large number of lines with the format

<actor>|<movie>

1. We will start by reading one of these files and forming a graph showing the connections between actors and movies.
2. We will choose one particular actor (possibly Kevin Bacon, but any actor or actress would do) to be the "source" node. We will then run the Shortest Path algorithm to find the shortest path from the source to every other node in the graph.
3. Given any actor's name, we can then display the path from the source to that actor.

For example, if A is the source actor and we are seeking a path to actor D, we might find that A and B were both in movie X, B and C were both in movie Y, and C and D were both in movie Z. This will print as

A => X => B => Y => C => Z => D

To get started, consider the file "Simple.txt", which contains the following data:

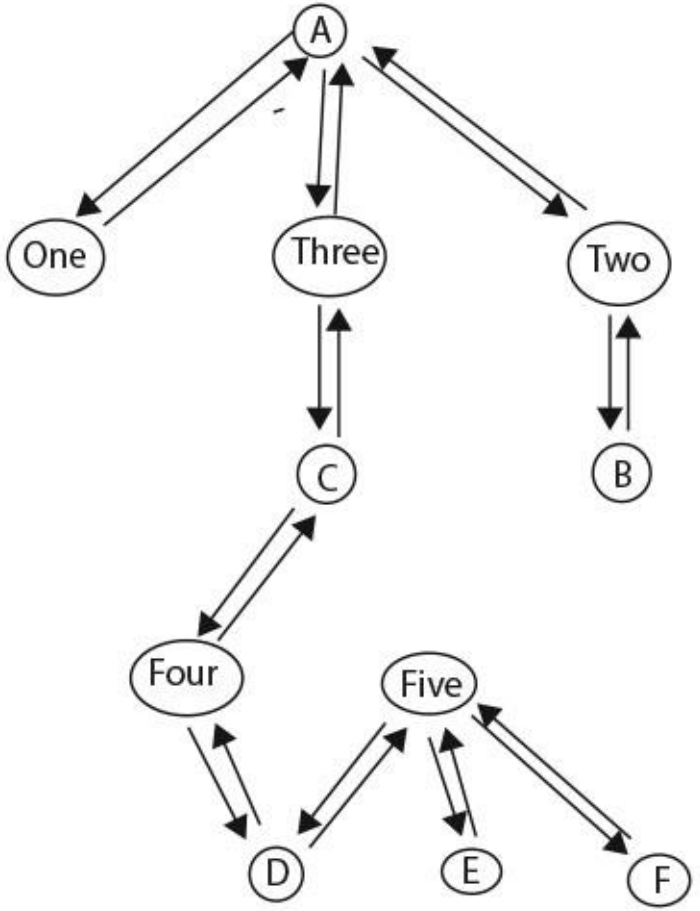
```
A|One
A|Two
A|Three
B|Two
C|Three
C|Four
D|Four
D|Five
E|Five
F|Five
```

Here A,B, C, D, E, and F are all actors

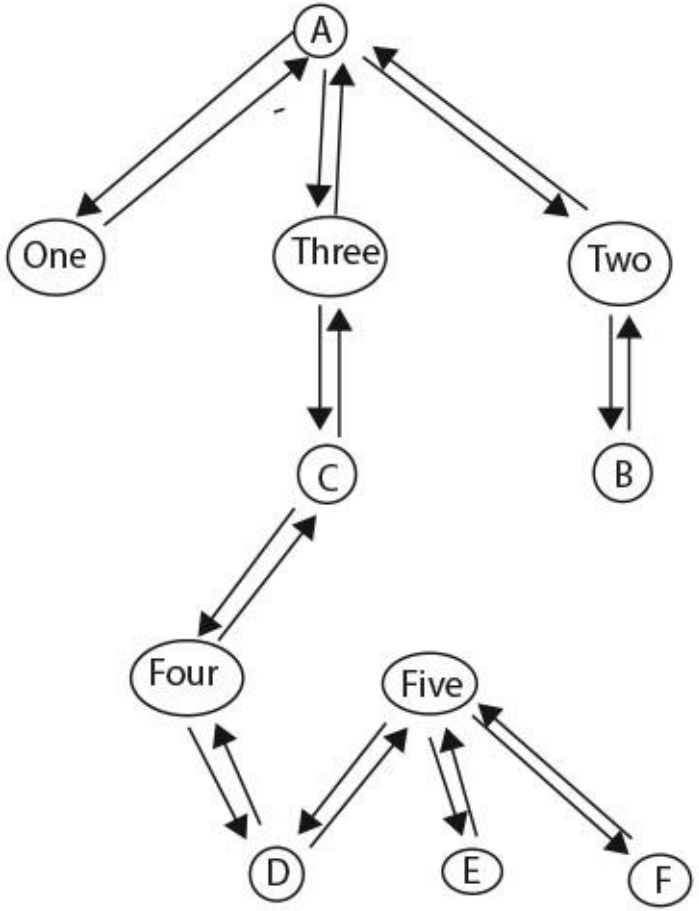
One, Two, Three, Four, and Five are all movies

We want to build a graph like this:

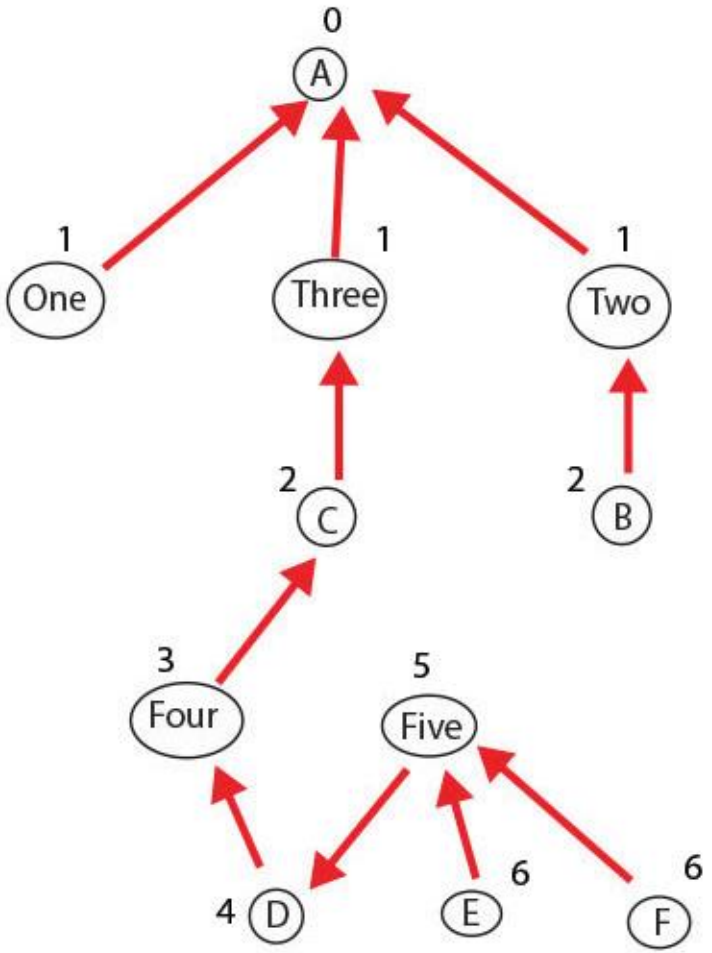
A|One
A|Two
A|Three
B|Two
C|Three
C|Four
D|Four
D|Five
E|Five
F|Five

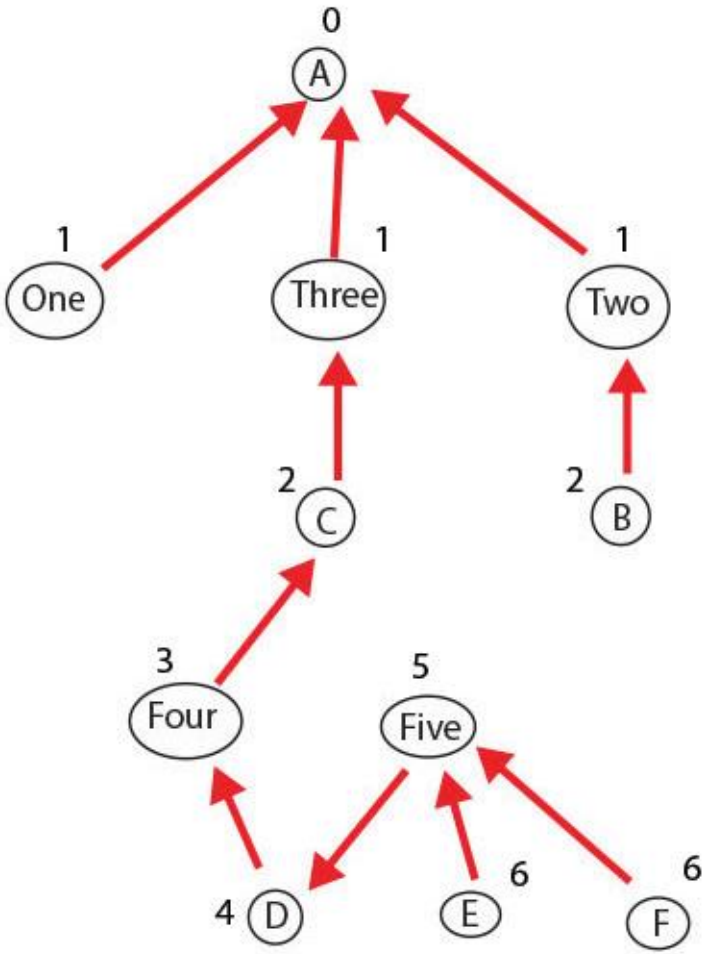


Note that every edge is duplicated so it runs in both directions. We are essentially making an undirected graph here. It is important that we are able to go from an actor to her movie OR from the movie to its actor.

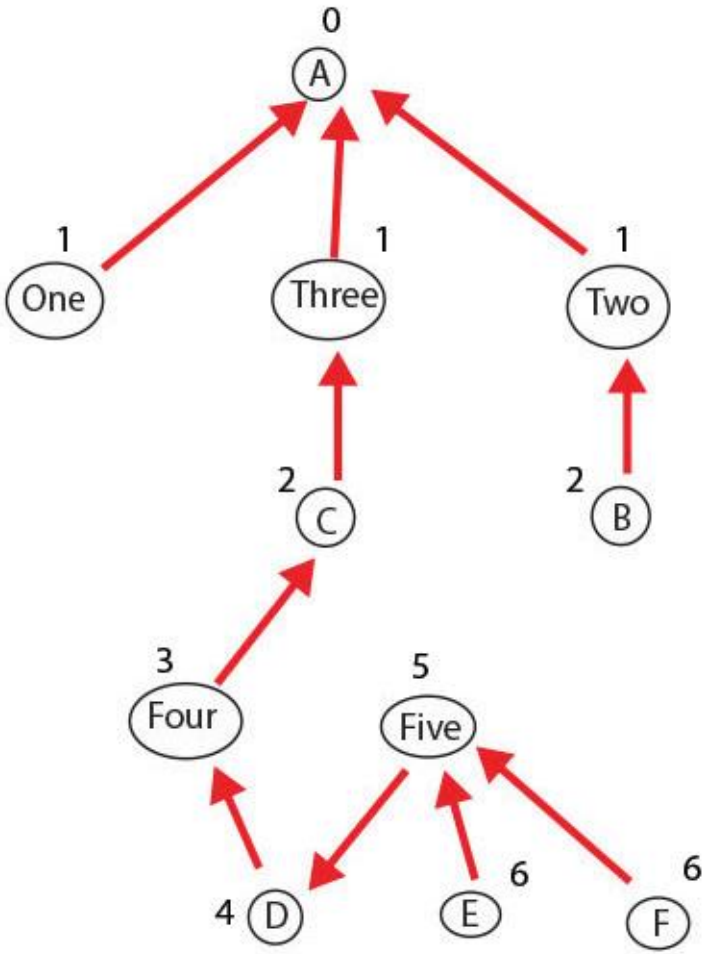


The Single Source ShortestPath algorithm for unweighted graphs adds the following "predecessor" links. We are using node A as the "source". The small numbers outside the nodes indicate the node's distance from the source:





Now, if we want the path from A to any node, we follow these links back from that node to the source, pushing each node onto a stack. When we get to the source, where there is no predecessor link, we reverse course and start popping the stack, adding the name of each popped node to a string. The result is the path from the source to the node.



Note that all actors have even distances and all movies have odd distances.

There is starter code for this lab – our usual framework for graphs, plus an application program. You need to modify the graph framework to implement a shortest path algorithm.

Here are the four files:

Edge.java This is complete; you shouldn't need to do anything with it.

```
public class Edge {
        public Vertex destination;

        public Edge(Vertex d) {
                destination = d;
        }

        public Vertex destination() {
                return destination;
        }
}
```

See: you are already a quarter of the way through the lab.

Vertex.java Your work on this comes in two parts. The first part is really easy. Our shortest path algorithm needs every vertex to have a value or distance variable, and also a predecessor on the shortest path back to the source. You need to add those variables and have them initialized by the Vertex constructor

```
public class Vertex {
    String name;
    List<Edge> adjacentList;
    // ADD DISTANCE AND PRED VARIABLES HERE

    public Vertex(String name) {
            this.name = name;
            adjacentList = new LinkedList<Edge>();
            // INITIALIZE DISTANCE AND PRED VARIABLES HERE
    }

    public int distance() {
            return -1;
            // ACTUALLY RETURN THE DISTANCE VARIABLE
    }
```

Graph.java Now the real work starts. The Graph class has the vertexMap – a HashMap<String, Vertex>. This much is done. You need to write **loadFile(String filename).** This reads a data file and builds the corresponding graph. Each line of the data file is formatted as

actor|movie

such as

Bob Geitz|Dumb Or Even Dumber??

First you need to separate out the actor and movie fields. If i is the index of the '|' character in a line then the actor field is line.substring(0, i) and the movie field is line.substring(i+1)

You can then call addEdge(actor, movie) to create the edge that needs to be added to the graph.

Remember that we are adding edges from actors to movies and edges from movies to actors. Every actor|movie line that you read from the file should result in two edges being added to the graph.

The next step is findAllPaths(String s). This implements the shortest path algorithm for unweighted graphs. You will keep a queue of Vertices. Argument s is the String name of the source node. You can get the actual source vertex as vertexMap.get(s). The queue starts with this node. Over and over you remove the head of the queue (call this h), look at its outgoing edges (h.adjacentList) and if the destination of any of those edges is a vertex with the default distance then you assign its predecessor to be h and its distance to be h.distance+1, and add it to the queue. It sounds complicated but it is very straightforward and you have tools to give you everything you need.

You will need a queue of vertices. Java has several classes you can use for a queue, though none of them are actually called Queue. The easiest to use is probably LinkedList<Vetex>. This has an offer( ) method to enqueue objects and a poll( ) method to dequeue them.

In the findAllPaths( ) method, when you assign a distance to a node you know that the node is reachable from the source. The Graph class has two lists: one of reachable actors and one of reachable movies. If the vertex has an even distance it should be an actor so add it to the actors list; otherwise add it to the movies list. This assumes the source vertex is an actor; if it is a movie they will be flipped.

There is one final step. In the Vertex class you need to add a method String getPath( ). For any vertex v, v.getPath( ) should return a string that says how to go from the source vertex to v, such as

Kevin Bacon (I) => Mystic River => Paul Newman => Cool Hand Luke => Bob

You do this just like you did in the Maze lab (Lab 3): follow the predecessor links until you get to a node with a null predecessor. This should be the source node. Push each of those nodes on a stack. Then repeatedly pop the stack until it is empty, adding what was on the stack to the string. I'll leave it to you to work out how to get the arrows into the string.

MakeGraph.java There is a fourth file in the starter code. This uses the graph methods you have implemented to play the Kevin Bacon Game. You should not need to make any changes in it. The program expects args[0] to be the name of a data file. If you want to change data files you need to re-run the program.

Here are the command-line options of the program. The commands are not case-sensitive, but the names of actors are.

Connect: On a separate line the system will prompt you for the actor to be the Source, and will run the Shortest Path algorithm with this source.

Path: On a separate line the system will prompt you for the name of an actor and will print the shortest path from the source to that actor.

Actors: The system will print the list of actors reachable from the source

Movies: The system will print the list of reachable movies

Refresh: This erases all of the effects of the shortest path algorithm so you can re-run Connect with a different source.

Quit: This exits from the system.

Note that you need to know how IMDB refers to various actors. Kevin Bacon is “Kevin Bacon (I)”. Paul Newman is “Paul Newman (I)”. Daniel Radcliffe is “Daniel Radcliffe”. Emma Watson is “Emma Watson (II)”. You can look them up on imdb.com to see what names it uses.

The full IMDB files are very large – millions of lines. I put one of them on the class website and put code in the starter Graph.loadFile() method for reading data files from the Net. After you are sure your code is working try using data file

http://cs.oberlin.edu/~bob/cs151/Labs/Lab9/imdb.no-tv.txt

This has 5 million lines, listing 1.6 million actors and almost half a million movies.