Due Nov 27, 2017, 11.59pm

This project will use IMDB's actor/movie dataset to compute the Bacon Number of an actor, which refers to the shortest path through the movie/actor graph that connects two actors – the so called 'Six Degrees of Separation' problem. You will use the BRIDGES s/w to display the path that leads from the actor to Kevin Bacon (or another chosen actor).

In part 1 (this project), you will build the full actor/movie graph, while in part 2, you will compute the Bacon number using the BFS traversal.

More details of this problem can be found at

`http://introcs.cs.princeton.edu/java/45graph/`

Dataset:

We will use IMDB's actor/movie dataset, which is now directly accessible through the BRIDGES API and illustred in the example at `http://bridgesuncc.github.io/Hello_World_Tutorials/Graph.html`.

The graph representation utilizes the following class in BRIDGES:

- **GraphAdjList<K, E>** This is the main representation of the graph on BRIDGES, mapping vertex values (in our case, actors or movie, names(String)) to any object E (in our case, we will point to edge objects that contain an actor or movie name (String)); Each vertex will point to an adjacency list, which will be used to store the graph edges (using the addEdge() member function). Methods to query for a specific vertex by the key K, setting node and edge attributes are illustrated in the graph example. Refer to the full class description at

  http://bridgesuncc.github.io/doc/java-api/current/html/annotated.html

- **Graph Adjacency List.** The graph representation uses adjacency lists, i.e., each vertex points to a singly linked list. This adjacency list is held in a Java HashMap, specifically, $HashMap < K, SLelement < Edge < K >>>$. Hashmaps are implementations of hash tables in Java. Here we use the key K (representing a vertex) to point its adjacency list (which is of type $SLelement < Edge < K >>$). Thus, indexing into this hashmap with an actor or movie name will point to an adjacency list containing the movies or actors connected to the vertex or movie, respectively. The addEdge() method is used to build the adjacency lists. Again, refer to the graph example on how to access and iterate through the adjacency lists, given a vertex name.

- $Edge < K >$**.** Used by adjacency lists, this is the class that holds the terminating vertex of the edge. The vertex has a name and possibly a edge weight, if applicable.

Tasks. Part 1: Due Nov. 22, 11.59pm

1. **Get the BRIDGES graph example in the above link working,** as per the instructions provided during the class lab exercise. Output the visualization; this should match the visualization at the above link.

2. **Build and visualize the full graph.** The call (refer to the example)

   ```
   ArrayList<ActorMovieIMDB> actor_movie_data =
           (ArrayList<ActorMovieIMDB>) bridges.getActorMovieIMDBData("IMDB", 1800);
   ```

   returns an array of actor-movie pairs (ignore the second parameter and use the size of the array for the number of actor-movie pairs). The ActorMovieIMDB class is described in the Java API documentation (see above link to all classes). Note that actors and movies should only occur once, i.e., you will build a graph that has a unique set of actors and movie nodes. Each actor/movie pair results in two edges, from actor$\Longrightarrow$ movie and from movie$\Longrightarrow$ actor. The graph you are building is effectively an undirected graph.

Thus, prior to adding a vertex (actor or movie) to the graph, you must check if the actor already exists (using the getVertices() method returns the hashmap of all vertices, which you can used to check to see if a particular vertex exists within the hashmap). If it already exists, then the movie or actor is added to its corresponding adjacency list. Else, a new vertex is created and its first edge to its movie or actor is created (again, you must check to see if the movie or actor it connects already exists or not).

At the end of this process, the graph contains a unique list of actors and movies and the edges represent relationships between the movies and actors: an actor's adjacency list contains all the movies corresponding to the actor and a movie's adjacency list corresponds to all the actors in that movie.

3. Once the graph is built, label the nodes of the graph to display the actor name or movie name depending on the node. See the graph example for the appropriate member functions.

Evaluation:

You will do an interactive demo of your implementation. This should be a fun project.

Submission Requirements.

Turn in all of source code to Canvas; ensure it is well documented.