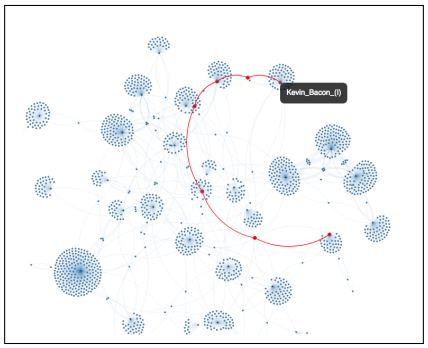
"Bacon Number" by Kalpathi Subramanian & Jamie Payton is licensed under CC BY-SA 2.0. Accessed from www.engage-csedu.org.

ITCS 2214/2215 Bacon Number Computation in Graphs Fall 2017

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 actor-movie graph that connects two actors – the so called 'Six Degrees of Separation' problem. You will use the BRIDGES s/w to display the full actor-movie graph, highlight nodes and movies and also compute the path that leads from a user chosen 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 illustrated in the example at

`http://bridgesuncc.github.io/Hello_World_Tutorials/Graph.html`

and described below.

```
ArrayList<ActorMovieIMDB> actor_movie_data =
        (ArrayList<ActorMovieIMDB>) bridges.getActorMovieIMDBData("IMDB", 1813);
```

The above example illustrates a BRIDGES program and the relevant classes in building a small graph of the IMDB data. The project will use these calls to extract the IMDB data and store the data in list of objects.

Graph Representation in BRIDGES

GraphAdjList<K, E>. This is the main graph class in BRIDGES, and has two components, (1) a set of vertices, accessible via a HashMap, and (2) each vertex points to a an adjacency list, mapping vertex names (in our case, actors or movie names (of type String)) to a singly linked list (of type SLelement¡Edge¡K¿¿) of edge objects containing the terminating vertex of an edge (actor or movie name)). 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

The graph vertices are held in a Java HashMap (for constant time access); this mimics constant time access in traditional graphs (where vertices are typically integers and stored in an array, and each vertex points to a linked list). Access to a specific vertex node in the graph can use the following calling sequence:

```
Element<String> vert = graph.getVertices().getVertex(vertex_name);
```

Thus, the getVertices() accesses the hashmap of the graph's vertices, and the getVertex() method accesses the node corresponding to the actor. This method is useful for setting attributes (color, size, opacity, etc) to the node, for example, vert.setColor(), vert.setSize(), etc.

The graph example also illustrates how to traverse the adjacency list of a vertex and access the terminating vertex of its edges, as well as color the edges. The adjacency list of a particular vertex is accessed using the following call sequence:

```
SLelement<Edge<K> > adj_list = graph.getAdjacencyList(vertex_name);
```

To access the edges, one can iterate through this list and access the edge using the getVaue() member function (of SLelement¡K¿), which will return an Edge object and then access the vertex and weight of that edge. Thus, *adj_list.getValue().getVertex()* will return the actor or movie stored as the termninating vertex of the first edge object.

Tasks.

1. **Get the BRIDGES graph example in the above link working.** Output the visualization; this should match the visualization at the above link.

2. **Build and visualize the full graph.** Once the data is extracted via the BRIDGES API (see above for the calling sequence), the returned list of ActorMovieIMDB objects contains the 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().getVertex()) to check to see if a particular actor exists within the hashmap). If it already exists, then the edge is created between this existing actor node and the movie (a similar check needs to be made to see if the movie node already exists). If the nodes do not exist then a new vertex is added (use the addVertex() method), prior to creating the edges. For a vertex that already exists, use *graph.getVertices().getVertex(vertex_name)* to get the vertex node object. The returned node provides also the handle to apply atributes to it (color, opacity, size, etc).

   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. **Label the Vertices**.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.

4. [**Optional.**] Consider labeling the nodes so that each actor node displays all of the movies that actor is part of, and each movie node displays all the actors in that movie.

5. **Bacon Number Computation.** This requires two steps:

   a. **BFS Traversal.** You will need to use a queue (use any of the Java queue implementations, eg. ArrayQueue, LinkedList, etc., in C++ use STL queue class) to implement a graph BFS traversal.

   b. **Bacon number (path length computaton.** The Bacon number represents the path length from a start node to a destination node. Let $D(i)$ represents the distance of a node i from its start node. During the BFS traversal, we will update the distance as follows:

$$D_i = D_{parent(i)} + 1$$

with the source node $D(source) = 0$. The distance array, mark array (to track visited nodes in the traversal) can be implemented by Java HashMap or STL unordered map, keyed on the actor/movie name. Note that all edges have a weight of 1.

c. **Bacon Number Determination.** The user specifies the start node (by actor name) and the traversal will search for the destination node, for eg., "Kevin_Bacon". As the BFS traversal progresses from the start node, the distances of the visited nodes are updated, until the destination node is reached. The final distance to the destination node is the Bacon number of the source node.

d. **Displaying/Highlighting the Bacon Number Path.** Given that we have a visualization of the actor-movie graph, it makes sense to display the path, once the BFS traversal is completed. This can be done by **keeping track of the parent** of each node that was visited during the graph traversal. Again, a hashmap $Parent < String, String >$ is maintained. When the traversal visits a node N, and N is a child of P, then $Parent(N) = P$. Once the destination actor is found, then we use the hashmap to trace back the path towards the source actor using Parent hashmap (linear in the number of nodes in the path). The parent of the source node is set to NULL or a sentinel to stop the path traversal.

Notes, Extensions to the Project

1. **C++ Version.** Use the C++ version of BRIDGES, replace Java data structures with C++ STL based data structures (unordered_map, queue).

2. **How to use this project.** This project can be overwhelming to beginning students in data structures. In the past, it has worked wonders when it has been broken up into multiple smaller pieces; for instance start with just building the graph example, make sure the students understand the BRIDGES calls to the graph class, then add in the BFS traversal, and finally the Bacon number determination. Its also useful to spend part of a lecture and ensure students can get the graph example working.

3. Additional tasks would include implementation of BFS and DFS algorithms, highight the start node and using opacity gradients to illustrate traversal order (gradually increasing opacity from the start node (radially outwards).

Evaluation:

Typically via an interactive demo. This should be a fun project.