Graph Design Analysis

Describe the implementation of your graph structure:

We create four new classes: ***Node***, ***Edge***, ***NodeHash***, and ***EdgeHash***.

1) class ***Node***: each actor is defined as an instance of ***Node***. An instance of ***Node*** contains the name of an actor and a list (more specifically, a vector) of all the movies he played in.

2) class ***Edge***: instead of being a single edge connecting two actors via a movie, an instance of ***Edge*** is a set of edges which includes all actors who played in a movie. In other words, an instance of ***Edge*** contains the title of a movie and a list (more specifically, a vector) of its cast.

3) class ***NodeHash***: an instance of ***NodeHash*** is a hash table (implemented using an array) which stores instances of ***Node*** (or actors) using indices acquired by hashing actor names.

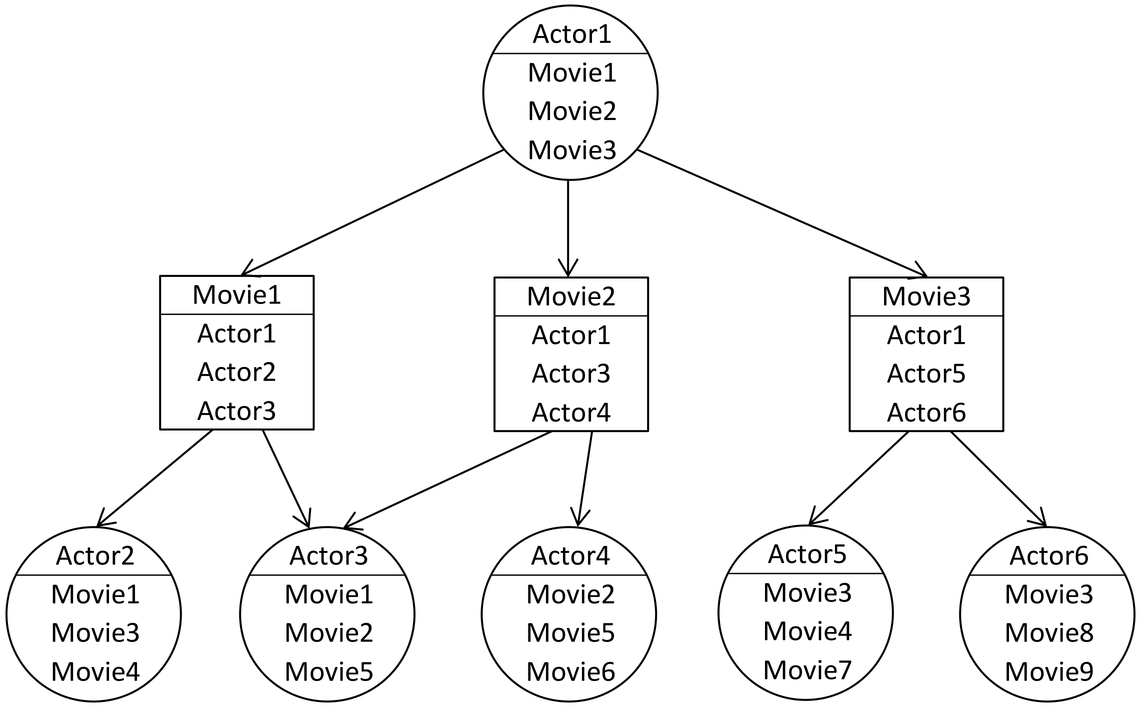
4) class ***EdgeHash***: an instance of ***EdgeHash*** is a hash table (implemented using an array) which stores instances of ***Edge*** (or movies) using indices acquired by hashing movie titles.

We modified class ActorGraph by adding two member variables to it: ***nodes***, and ***edges***.

1) ***nodes***: an instance of NodeHash. 2) ***edges***: an instance of EdgeHash.

(In the following text, neighbor/actor/cast/node will refer to instances of ***Node***, and movie/edge will refer to instances of ***Edge***.)

To visit all neighbors of an actor, we access each movie he played in and visit all its cast. This procedure is illustrated by the graph shown below.



In the graph shown above, although Actor3 can be accessed through both Movie1 and Movie2, it will be pushed into the queue only once because Actor3.dist is no longer INFINITY after the first visit.

Therefore, by our implementation, a graph is an instance of ActorGraph whose member variables are a hash table used to store actors and a hash table used to store movies.

Describe why you chose to implement your graph structure this way:

We can consider instances of ***Node*** and ***Edge*** as pools. Every time we read an actor name X and a movie title Y from file, we hash them ($O(1)$) to get their indices in ***nodes*** and ***edges***, and then add X into Y's pool ($O(1)$) and add Y into X's pool ($O(1)$). Let $|L|$ be the number of lines in the file, then the whole process of build and update graph is $O(|L|)\times\big(O(1)+O(1)+O(1)\big)=O(|L|)$.

This design is efficient because every time we read an actor X and a movie Y from file, we only need to update two things: X's pool (node X) and Y's pool (edge Y). In other words, we don't need to search the whole ***nodes*** to update all actors who played in Y.

Actor Connections Running Time

	100 same pairs (from (LUV, FREEZ) to (LYMAN, WILL))	100 different pairs (pair.tsv)
bfs running time (average)	0.718078s	0.903417s
ufind running time (average)	0.861956s	1.39615s

Which implementation is better and by how much: BFS is better in both finding same pairs and finding same pairs. In finding same pairs, BFS only needs 83% of the time needed by union-find. In finding different pairs, BFS only needs 64% of the time needed by union-find.

When does the union-find data structure significantly outperform BFS: as shown in our result, BFS always runs faster than union-find.

Argument: in BFS, we only need to construct the graph once, then we only need to limit our search to those edges (movies) which were made on or before a certain year each time. In union-find, however, we need to update the data structure every time a group of movies which were made on a certain year are added. Therefore, BFS is faster compared with union-find.