Mark Barnes
Zehua Liu
PA 4
3/2/17

Graph Design Analysis

Describe the implementation of your graph structure (what new classes you wrote, what data structures you used, etc)

There are two nodes that exist for the production of this graph: ActorNode and Movie. ActorNodes are designed to include a ‘name’ that is the actors name, a pointer to the previous Movie, and a vector of Movie pointers. The name of the actor, identifies who the actor is. The vector of Movie pointers indicates what Movies the actor has been in (these are the edges). Movies represent the edges and have instance fields ‘title’ for the title of the movie, ‘year’ for the year of release of the movie, ‘preA’ to indicate the previous actor node, and ‘actor’ a vector of all the actors that participate in the movie. All of these values, except the name, are initialized to 0 on construction. When we use the pre-written method “loadFromFile” we read in the actor, movie title, and year of movie, and then load that into an unordered_map. We specifically save the name of the movie as “movie_name#@movie_year.” This allows us to use .compare() to check if two movies are the same (same title and same year). Since there are duplicate actors, and duplicate movies, we only add to the unordered_map new values. As we mentioned previously we add Movie and ActorNodes into an unordered_map. We included in ActorGraph.h, two public unorderedMaps of <string, ActorNode*> and <string, Movie*>, which allows us to check in pathfinder.cpp if the two actors exist, and then if they do the path between them. We then use a BFS to search through the tree for a route from actor A to actor B. During the BFS as we go from one ActorNode to the next, and set the ActorNodes ‘preM’ variable to the edge that connects one node to its parent node. And we set preA nodes to the other ActorNode, this allows us to traverse back up through the tree and determine the best path and also determine if a path does not exist.

After the checkpoint we added a few instance variables to ActorNode and Movie. For the ActorNode we added a boolean ‘done’ for dijkstra weighted path. This way we can determine if a ActorNode has been visited before or not. We also added ‘dist’ which is an int, and whole the sum of the movie_weights from the start actor to this actor. Another is ‘parent’ and this is a pointer, and is used by the disjoint set, and is set to the root node. For Movie we added ‘weight’ and ‘isConnect.’ The ‘weight’ represents the weight of the edge, this is helpful for when calculating a weighted path. It is set to “1 + (2015 - year)” as detailed by the instructions.

Describe *why* you chose to implement your graph structure this way (Is your design clean and easy to understand? what are you optimizing?, etc)

We are chose to implement the graph structure this way so that it focuses on the connections between actors and the movies that they are in. This is clean design. It optimizes connectivity and accessibility of nodes. This design is very Object Oriented and is easy to understand as it models its real world application. When considering what defines a movie, it makes sense to include the title, year of release, and the actors that performed in the film. When viewing an actor it makes sense that it is known the actor’s name and the movies that the actor has acted in. Providing the programmer with this simple information, enables success when determining the degree of separation between one actor and another.

Actor connections running time

Which implementation is better and by how much?

```
[zel035@acs-cseb250-36]:cs100pa4:783$ time ./actorconnections movie_casts.tsv test_pairs2.tsv
out10.tsv bfs

real	0m10.793s
user	0m10.540s
sys	0m0.223s
[zel035@acs-cseb250-36]:cs100pa4:784$ time ./actorconnections movie_casts.tsv test_pairs2.tsv
out10.tsv ufind

real	0m0.449s
user	0m0.429s
sys	0m0.006s
```

The runtime for ufind is ~10 seconds faster than the runtime using BFS. This holds true for the smaller file too. This is seen down below as an argument to support my observation. On the smaller file the time for ufind is ~2x as fast as BFS.

When does the union-find data structure significantly outperform BFS (if at all)?

The union-find data structure significantly outperforms BFS for large files. When the union-find and BFS are used on small files, then the output is around the same though union-find is around twice as fast.

What arguments can you provide to support your observations?

```
[zel035@acs-cseb250-36]:cs100pa4:769$ time ./actorconnections 2-node_simple.tsv
2-node_simple_pair.tsv out10.tsv bfs

real	0m0.035s
user	0m0.004s
sys	0m0.004s
[zel035@acs-cseb250-36]:cs100pa4:770$ time ./actorconnections 2-node_simple.tsv
2-node_simple_pair.tsv out10.tsv ufind

real	0m0.019s
user	0m0.003s
```

```
sys    0m0.003s
```

Regardless of the file size, ufind runs faster than BFS. Using data posted for larger file above, this holds true still.