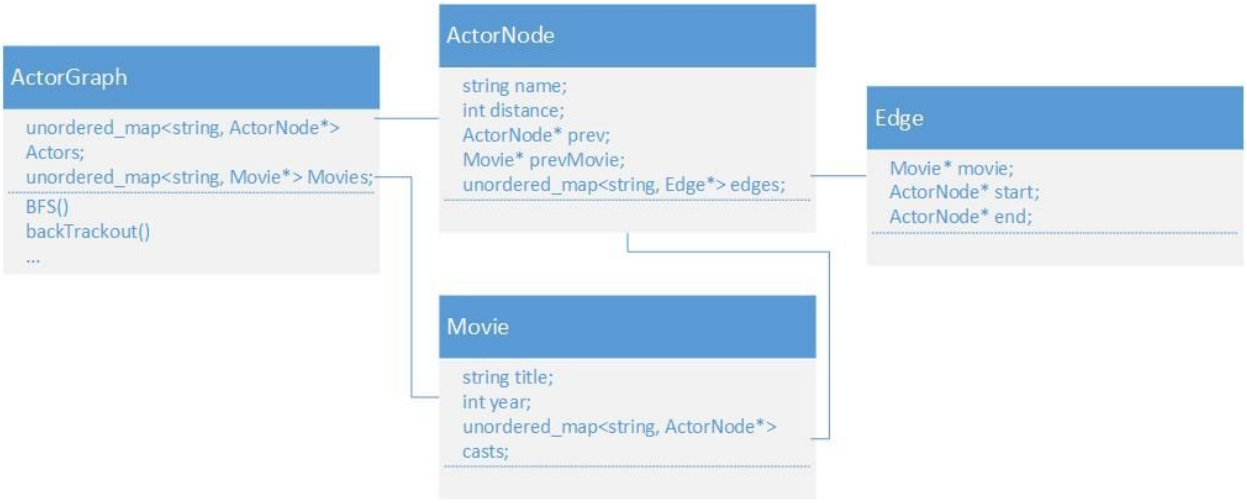
CSE100 PA4 Report
Cheng Qian A53209561

Graph Design Analysis:



I design and implement the following class:
ActorGraph:
The Graph class. It has two table to store the pointers to all the actors and movies, separately.
For the actors table, the key is actor's name and the value is the pointer to its entity.
For the movies table, the key is movie's title + movie's year, because the there might be different movies with same name. So we need this combination as the unique identifier of the movie.

ActorNode:
The node class. It has the basic actor attributes and a table to store the pointers to all its adjacent edges.
The pointers to previous node and to previous movies are used in search algorithm. I will explained later.
For the edges table, the key is movie's title+ movie's year + end node name. The reason why we use this ID is that there might be different movies that both actors(vertices of this edges) cast in and different movies with same name.

Edge:
The edge class has two pointers to its two vertices and a pointer to the movie.

Movie:
The Movie class contain the title and year of the movie. It also has a "cast" table to store all the pointers to casts.
For the casts table, the key is the actor's name and value is the pointer to the entity.

The cast table is used to make connection based on the information of movie_cast.tsv.The main idea is centered at the movie in the info, and make connections between current actor with all the actors in the cast list of the movie.

Explanation:
First of all, as we treat this problem a graph problem, we need to have the graph class and node class definitely. In the graph class, I implement two table of actors and movies. This help me fastly get the address of each entity through their name, instead of finding it through iteratively traversing the whole set. However, we cannot easily just use adjacency list as the topological structure of our graph, because two actors might be commonly casted in several movies and those connections are different. So the implementation of an “Edge” class is necessary. In this way, I can traverse through all the edges of current node in search algorithm instead of traversing the adjacent nodes, which might lose some information.

One question is, using directed edge or undirected edge? My choice is directed edge. Although this graph is said to be an undirected graph(edge represent the common movie that both actors has been casted in), it is still more reasonable to represent every undirected edge by two directed edge. In my design, every node has a table of adjacent edges pointer. When traverse through all the edges in the search algorithm, I can easily get the pointer to the neighbour node by calling the “end” pointer in my edge class. Otherwise if I only have an undirected edge, I need to choose the node that is not the current node itself, which may do harm to the efficiency of program.

Another question is, why we need previous node pointer and previous movie pointer in the ActorNode class? Here we first call BFS function to set up all the distance/prev/prevMovie of the nodes in the path. Then we only need to backtrack from the end node, all the way to the start node, using a stack to record all the nodes on the path. While the stack is not empty, pop the nodes out and print its previous movie first and its name secondly. In this way, I optimize the running time to print the path because I already attached the intermediate movie information with the nodes.

runtime on 5 query pairs:
BFS 0.011
Dijkastra 0.429

Actor connections running time:

same_pair.tsv:
BFS:
time:1242147832(nanosecond) = 1.24215(second)
ref:The duration in milliseconds is : 5438.35

Unionfind:
time:12628532(nanosecond) = 0.0126285(second)
ref:The duration in milliseconds is : 1658.47

pair.tsv:
BFS:
time:20435735480(nanosecond) = 20.4357(second)
ref:The duration in milliseconds is : 2823.27

Unionfind:
time:184977773(nanosecond) = 0.184978(second)
ref:The duration in milliseconds is : 1639.84

Question:

- **Which implementation is better and by how much?**

Union find with the use of disjoint set is better. It can reduce the runtime to 0.9% of the time the BFS way costs.

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

I did not observe the case where the union-find data structure significantly outperform BFS than another case. Perhaps because my programme logic is as below:
I first read all the record in the pairs.tsv and store all the **unique** pairs(start_actor_name + end_actor_name) into a set, and then write a function that read this set as input and output a map whose key is the set's element and value is the begin year we want. Finally, for each record in the original pair file, I search it in my map and return the value. So if the pair file is full of the same pair, then I only got a set with only one pair and run my algorithm for once, to implement the map with only one pair. In this case, if we assume there are no two same pairs in the pair.tsv, the runtime to find only one pair will dropped to 1% of the time to find for 100 pairs, which also matched with my experimental result.

- **What arguments can you provide to support your observations?**

By analyzing the principle of union find, we know that after a successful find operation, the current node's prev pointer will be pointed to the root of the up tree, as well as all the nodes on the path, which is path compression. So union find is a self-optimizing process, comparing with the BFS way. Thus union find with disjoint set will have a better algorithm efficiency.
And the argument of why I did not observe some special case that union-find is much more outperform BFS is as above in the second question.