A.Graph Design Analysis:

1. I use actors as my node class and in the Actor class it has fields : dist, prev, name, done , and edges. dist and name are pretty self-explanatory- dist keep track of the actor's distance from the starting node when running BFS as well as Dijkstra's algorithm. prev utilizes std::pair to store a Movie*, Actor* pair, in order to keep track of BOTH the last movie and the last actor after running either one of the two searching methods.

   done is a boolean that only used in Dijkstra's algorithm to mark whether an Actor node has been visited or not.

   edges is a pointer to a "vector" that holds pairs, which are again consist of Movie* and Actor*. These pairs represent the connection between actors and the second of the pair (Actor*) is the another actor that an actor is connected to. So to be precise, it is a directed edge.

The second class I wrote is the Movie class. It holds name, year, and cast of a particular movie in the read in file. Both name and year are strings and cast is a pointer to a "vector" that holds Actor*. The cast field keeps track of all the actors that had appeared in a movie. With the cast field, I manage to loop through all the actors in a same movie in interest and then connect them all with each other with edges in my buildGraph() method.

My actorGraph class holds two pointers of unordered maps (hash maps); one for all actors in the file and the other for all movies in the file, respectively.

The actorList maps the name(key) of the actor to the actor pointer(value), and the movieList maps the name(key) of the movie to movie pointer(value).

2.
I use unordered_map as much as I can for storing all actors and movies from the cast file, since I know beforehand that they can be large in size. Unordered_map provides me fast(constant) access from the key, which is string in my case, to the value, which is either a pointer to a movie object or an actor object.

In terms of optimization, I used pointers almost everywhere as long as I can. You can see that I always pass in pointers (passed by reference) to my methods to avoid copying huge amount of data.

The containers, actorList and movieList, are also initialized as pointers to unordered maps.

As the readers would notice in my later file, I stick to maximize my use of hash map and pointers and steer away from linear searching, except for where I saved the name of actor pairs in temporary vectors to ensure the order of printing output.

Aside from the temporary containers to help order output, the only place I use vector is when I am loading the input file of actor pairs and parsing every line in it.

Lastly, I have to mention that I did somehow sacrifice readability of my code when using the above approach. The use of pointers generally makes my lines of code longer when calling methods. Also, using data structures such as hash map from C++ STL means iterator is unavoidable if I try to loop through any container, which again could hurt the readability a bit.

I have tried my best to refactor by renaming some long references/values with meaningful variable names. But I would say there's still quite some room for improvement of code style due to the time constraint of this PA.

B.Actor connections running time:

In my runtime analysis of actorconnections, I use the timer class from PA2 and I convert the unit from nanosecond to second by dividing it with 1 billion, since I figure for this large amount of input queries, accuracy to seconds would suffice.

The following lines are the actual output from my terminal on local machine, where I copied "BACON, KEVIN (I) HOUNSOU, DJIMON" from test_pairs.tsv and pasted it 100 times.

```
DanieltekiMacBook-Pro:pa4 xxxxxxxx$ ./actorconnections movie_casts.tsv
100-same-pairs.tsv runtime_bfs.tsv bfs

It takes 11 seconds to finish BFS!
Actor connections finished!!

DanieltekiMacBook-Pro:pa4 xxxxxxxx$ ./actorconnections movie_casts.tsv
100-same-pairs.tsv runtime_ufind.tsv ufind

It takes 30 seconds to finish ufind!
Actor connections finished!!
```

It turned out that for the same 100 pairs , my BFS takes roughly a bit more than 10 seconds and union-find takes around 28-30 seconds to complete. For BFS, the timer is called at the beginning of building all edges for each movie year. For ufind, it's called right before initializing the disjoint set object. The timer is stopped after printing out all lines to output file for both algorithms.

Now, the following is the result of running 100 different queries with the two algorithms.

```
./actorconnections movie_casts.tsv pair.tsv runtime_bfs2.tsv bfs
It takes 12 seconds to finish BFS!
Actor connections finished!!

./actorconnections movie_casts.tsv pair.tsv runtime_ufind2.tsv ufind
It takes 21 seconds to finish ufind!
Actor connections finished!!
```

1. My BFS is overall two to three times faster than ufind.

2.&3.

The good news is that it seems both of my algorithms are very fast. However, I assume, by the hint in the PA4 writeup, that normally union-find should run faster than BFS.

One reason I can think of that might cause this result is that I write more setters and getters in the union-find class compared to my BFS. BFS is called directly from ActorGraph class and I realize afterward that I was trying harder in my UnionFind.hpp to stick to the OOP practice and try to encapsulate as much as possible, while I tend to commit getters and setters in my ActorGraph class because I felt like it was already a huge class and didn't want to further complicate the code(Note: I was stuck at a bug for 3 whole days and it greatly affected my overall progress of this PA , so I was trying to get everything working first and kinda slacked on the style)

Besides, I suspect that I can further speed up my union-find algorithm by getting rid of the disjointNode class and use string (the actor's names) instead, since the disjointNode has to store Actor pointer and other fields in it. It was my intention at the beginning, but I gave up using strings and hash map to represent the up-tree structure at one point where I found out it's too hard to debug and once again I made compromises in order to simply get my program work.

However, there is still a pattern that can be seen. We see my union-find takes roughly three times longer than BFS in the case of 100 same pairs (30 sec vs 11 sec). While for the input of different pairs, the BFS is still blindingly fast but the union-find also performs significantly better. Union-find only takes two times longer than BFS in this round (21 sec vs 12 sec).

So it can be deduced that union-find becomes relatively efficient when dealing with different pairs.