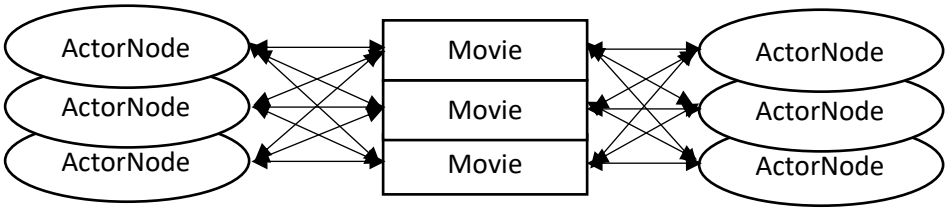
Graph Design Analysis

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

For the implementation of the graph structure, 4 classes were written: ActorNode, ActorEdge, Movie, and ActorGraph. The data structures used are described below in bullet points, but can be summed up into vectors, queue, priority queue, and unordered_map.

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

The main reason for writing 4 classes for the graph is to simplify the problem of visualizing a graph. ActorGraph contains the methods that acts on the graph and the components of the graph, such as the node, the edge, and the movie. The graph is implemented with undirected edges because a connection between two actors connects the two actors as they both acted in the same movie. It makes logical sense. The structure of the graph is as such: ActorNode is connected to a Movie which connects to other ActorNodes. An ActorNode can connect to multiple movies, and as such, the movies connect to its respective actors. The Movie class essentially acts as an incoming edge between the ActorNodes as it contains a vector of all actors in the movie. The ActorNode class contains a vector of movies which acts as an outgoing edge. A simplified model is as follows:



The ActorEdge class is used as a form of parent pointer for pathfinder as it represents a connection between two ActorNodes. Whenever a search path determines that the two actors are connected, an ActorEdge is created between the two actors. In order to print out the path, the end node traverses through the ActorEdge to the previous node, shifting the pointer to the previous node while passing the name of the movie that connects the two actors.

Bullet Point Description of each class:

- ActorNode class contains:
    1. data fields :
        - A vector of Movie class pointers called mList
            - mList : for use in all programs
                - to keep track of what movies the actor has played a role in
        - 4 integer data type fields
            - dist : for use in pathfinder
                - both weighted and unweighted edges
            - prevE : for use in pathfinder

        - to keep track of the path to print out
    - index : for use in actorconnections
        - to keep track of where the node is in a vector
    - parent : for use in actorconnections' Union Find
        - to keep track of the index of the sentinel (unless the node itself is a sentinel which would point to -1)
- String
    - name : for use in all programs
        - stores the actor's name
- Boolean
    - done : for use in pathfinder with weighted edges
        - if node has been visited, then then done will be true
    - sentinel : for use in actorconnections' Union Find
        - to designate if a node is a sentinel

2. 2 Constructors
    - ActorNode(string n) : for use in all programs
        - Only sets the name field and initializes the vector
    - ActorNode(string n, int i) : for use in actorconnections
        - Sets the name field and the index field and initializes the vector
3. Destructor
    - Frees the memory allocated to the vector of movie pointers (mList)
4. Setters and getters for each data field
5. Other functions:
    - reset() : for use with BFS
        - resets prevE, done and dist fields to 0
    - getNeighbors()
        - go through all the movies in mList and get all the actors from the movies
            - returns a vector of pairs containing actors and the names of the movie
    - getWeightedNeighbors()
        - go through all the movies in mList and get all the actors from the movies
            - returns a vector of pairs containing actors and pointers to the movies

- Movie class contains:
    1. data fields :
        - string
            - name : for use in all programs
                - stores name of the movie
        - int
            - weight : for use in weighted edges in pathfinder
                - stores the weight of the movie (which would be 2015 – (year produced) + 1)
        - vectors
            - ActorEdge pointers : for use in pathfinder

        - Keeps track of all edges
    - ActorNode pointers : for use in all programs
        - Keeps track of all actors played in this movie
2. 2 Constructors
    - Movie(string n) : for use in all programs
        - Sets the name and initializes the vectors
    - Movie(string n, int w) : for use in weighted edges in pathfinder
        - Sets the name and weight of the movie, and initializes the vectors
3. getter methods
    - getter methods for all data fields
4. Destructor
    - Free the memory for the two vectors

- ActorEdge class contains:
    1. data fields :
        - ActorNode pointers : for use in pathfinder
            - a1 : one actor in the movie
            - a2 : another actor in the movie
        - string
            - name : for use in all programs
                - Keeps track of the name of the connecting movie
        - Movie pointer
            - movie : points to the movie that connects the two actors
    2. 2 Constructors
        - ActorEdge(ActorNode * one, ActorNode * two, string n) : for use in all programs
            - Connects the two ActorNodes and stores the name of the movie
        - ActorEdge(ActorNode * one, ActorNode * two, string n) : for use in weighted pathfinder
            - Connects the two ActorNodes and stores the pointer to the movie
    3. getters
        - getter methods for ActorNode pointers and name
- ActorGraph class contains:
    1. data fields :
        - 2 unordered_map : for use in all programs
            - <string, ActorNode*> : stores the names of the actors and the actor nodes
            - <string, Movie *> : stores the names of the movies and the movie nodes
        - vector : for use to free up edges created between actors
            - <ActorEdge *> : stores all edges created between actors in order to free the memory allocated to the edges
        - 3 int fields :
            - aCount : keeps count of number of actors
            - mCount : keeps count of number of movies
            - eCount: keeps count of number of edges

2. Constructor
    - Default constructor to initialize vector and unordered_maps
3. Destructor
    - Calls removeVecs() to iterate through each vector and unordered_map and delete individual elements
4. Loading methods
    - loadFromFile : loads the data directly from the inputfile and structures the graph with weight or not
    - loadFromTuple: loads the data from a tuple<string, string, int> : for use in actorconnections
5. Finding methods
    - findPaths : finds the shortest path between the source and end nodes (unweighted)
    - findWeightedPaths : find the lowest weighted path between the source and end nodes
    - findYears : find whether or not two nodes are connected (for use in actorconnections

Actor Connections Running Time

Trial Number	Finding 100 Pairs		Finding 5 Pairs	
	BFS	Union-find	BFS	Union-find
	Time (s)	*Time (s)*	*Time(s)*	*Time (s)*
1	5.58491	0.512799	0.5268	0.2758
2	5.46404	0.562309	0.562703	0.265581
3	5.63912	0.50755	0.53311	0.27129
4	5.33986	0.513246	0.556543	0.258686
5	5.35766	0.522071	0.519955	0.263613
6	5.34295	0.514919	0.533287	0.257932
7	5.44595	0.516829	0.537881	0.26565
8	5.5263	0.500568	0.562102	0.269064
9	5.67146	0.502448	0.527026	0.262187
10	5.37761	0.517798	0.544588	0.279033
Average	**5.474986**	**0.5170537**	**0.5403995**	**0.2668836**

Which implementation is better and by how much?

Union-find implementation is definitively better by almost an order of magnitude when searching for 100 pairs, and only slightly faster when searching for 5 pairs.

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

According to the data above, union-find significantly outperforms BFS when finding a large number of pairs. Given a smaller number of pairs to search, the two algorithms perform similarly.

What arguments can you provide to support your observations?

With the implementation of path compression, the worst-case time complexity of the Up-tree used in Union-find is reduced from O(log *n*) to O(*N* + *M* log**N*) across all operations where *N* is the number of “union” operations and *M* is the number of “find” operations (see Stepik 4.7.13). log**N* is never more than 5 which results in an almost constant O(1) worst-case time complexity for each find or union operation. BFS, on the other hand, has the tightest time complexity of O(|V| + |E|) where E is the number of edges and V is the number of vertices. The faster search times in the table above reflects the respective time complexities.