Marie Luu-Pierce, Farangiz Akhadova, Ashley Yang

CS230 Final Project: Bechdel Analysis

Through this project, we were able to analyze the issue within a lot of movies. The truth is that there is little to no female representation in the film industry, or if there is, shallow characterization of the women, if any at all. Through the Bechdel Test, and other tests, we are able to analyze 50 films from Hollywood, the super big daddy of the film industry, and see if they pass a series of quantitative tests to deem them "adequately" representative of women.

For our HollywoodGraph implementation, our constructor had two separate LinkedLists for movies and actors, so that in our readFromFile, on top of having our Vector of vertices, we could also separate each vertex into the correct linked list. We chose LinkedLists as they could be any length, and this way, when we needed an actor or movie specifically, we could check if the object was in the actor list or in the movie list using contains(), depending on what we needed. We attempted to use inheritance from the "AdjListsGraph", but it didn't work, so we wrote in the methods separately.

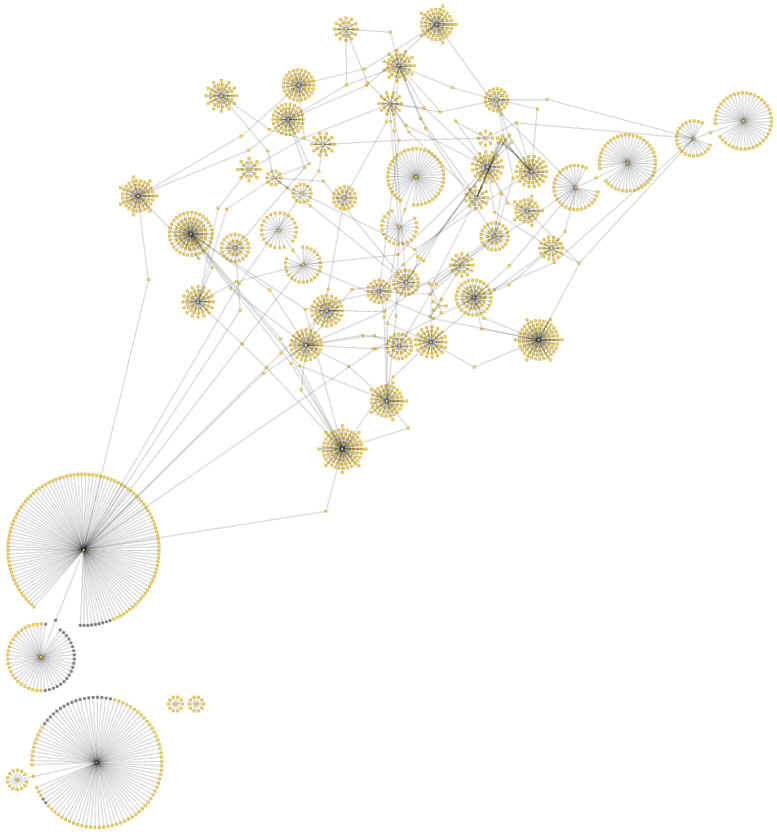
For Task 1.2, these were our thoughts on the graph: The graph displays the vertices (actors and movies) as yellow boxes, each with edges pointing to each other, signifying that a specific actor starred in that particular movie. The vertices had two edges pointing to one another, indicating that it is undirected. With the small txt file, we could see the vertices and the edges very clearly; seemingly random distribution of lines and shapes. However, with the larger txt file, it was very geometric, beautiful, constellation-esque. The movies were encompassed by their actors, circular and there were actors that were farther from the movie vertices (circles) but were still connected, signifying that they starred in other movies.

To determine the amount of movies separating two actors, we used BFS to find the shortest path between the actors. We used a queue to keep track of all the actors, as it would be easy to add and remove items, and added the first actor to the queue. Then, for each of the movies of the current actor, we would get the actors in those movies (adjacent actors) and add them to the back of the queue (which is why we needed to use a queue, so that we could add more elements to the back and remove processed elements from the front, and that way we could continue iterating through and adding more at the same time). This would repeat (for each adjacent actor, we'd get the movies, then get the adjacent actors for each movie) until actor 2 was found. We also kept an array to hold the distances. Each index would hold the distance from the corresponding actor to actor 1. We used an array to make it easier to access previous elements, as we would add the distance for each adjacent actor by taking the current actor's distance and adding one. So distance[0] is 0, because the corresponding actor is actor1, and then for each adjacent actor of actor 1, distance would be 1, then for each adjacent actor of those actors, distance would be 2, and so on. Once actor 2 was found, the distance at that index would be returned.

The test that we added was the FAME test (for our initials), which tested whether at least half of the lead actors in a given movie were women. We thought that this was a meaningful test to include as it is important for women to have leading roles. We did this by keeping a Vector of the total number of actors for each movie, then we just used a for loop to get the number of women and divided by the total to get the result.

The testing in which we ensured our methods and classes were working effectively were by observing the nextBechdel_cast txt file, and our generated TGF file to check our work. For instance, for the reading in the file method, we tested it on the small_bechdelCast txt file, and then printed out the vertices vector, and also movies and actors LinkedLists. We then moved onto

the larger file, the nextBechdel_cast txt, and printed out the same things (vertices, movies, and actors), and made sure they were the same. For the saveAsTGF file, we were able to test our small movies txt file, and check our results in the TGF file; similarly to the nextBechdel_cast txt file, upon inspection, we were able to see that everything was correct. We also would print our toString to see if our vertices were populating correctly, as well as printing out our movies and actors vectors. As for the methods such as moviesFromActor and actorsFromMovie, we tested this extensively based on our large and small cast file, and tested large actors such as Tyler Perry as well as actors who had only starred in 1 movie like Stephen Lang. We also tested if the actor or movie did not exist. For our moviesSeparatingActors method, we tested extensively using our TGF file created from the large Bechdel testing file, as well as the small txt file. We tested many cases such as ones where the actors were in the same movie, where the actors were one movie apart, as well as where they do not have any movies in common (returning -1).



Throughout this project our group gained valuable insights into object-oriented programming paradigms and problem-solving techniques in order to analyze real-life data concerning gender imbalance in Hollywood movies. By working together collaboratively as a group we were able to design and implement methods that explored and addressed this issue.

We applied OOP paradigms by designing a class structure to represent Hollywood movies and actors, encapsulating their attributes and behaviors. For instance, the “HollywoodGraph” class serves as the main component of our program and utilizes inheritance from the “AdjListsGraph” class to implement a graph-based representation of relationships between the movies and the actors.

The methods that we wrote demonstrate problem-solving techniques and allowed us to work with real-life data. For example, our readFromFile() method parses through a CSV file containing information about movies and actors, and effectively extracts the relevant data to populate our graph-based data structure.

The use of different data structures, such as vectors and linked lists, illustrates our understanding of the importance of selecting appropriate data structures to accomplish complex tasks. In our code, we use vectors to store lists of edges connecting movies and actors, and linked lists to maintain the vertices. We used our ability to leverage different data structures to efficiently manage and manipulate the data. Additionally, the moviesbyActor() method iterates through the list of vertices and edges in the graph and efficiently locates the actors and retrieves their associated movies.

Throughout the project, we made various implementation decisions, such as optimizing runtime efficiency while maintaining code reliability and adhering to style guidelines. FOr example, in the toString() method, we carefully crafted the output format to provide a clear and concise representation of the graph's vertices and edges. In addition, our method FAMEcalc() demonstrates consideration for runtime efficiency by effectively processing data to determine gender balance in leading roles.

Our team collaboration on this project was organized and efficient. For tasks 1.1 and 1.2 everyone contributed equally by brainstorming ideas and programming together. We all worked together to implement the class, the constructor, the initialization and declaration of the instance variables. We worked together to implement an efficient readFromFile(), the methods that had to do with adding edges and vertices, as well as the toString() method. Then, Erin took charge of Task 2.1 where she implemented the moviesByActor() method, which retrieves movies by the actors. Farangiz implemented Task 2.2, which is the actorsFromMovie() method. Farangiz and Marie worked together after discussing various approaches with the entire group to implement 2.3 which is the moviesSeparatingActors() method. Ashley and Erin worked together to implement Task 2.4 which is the FAMEcalc() method, and involved the writingFAMEtoFile() method to output the FAMEcalc() results.

Throughout the week we scheduled regular meetings to discuss the progress, share code snippets, and provide feedback on each other's work. We communicated frequently to coordinate tasks and edit each other's code, and ensured that everyone remained on track and accountable for their assignment responsibilities.