CelebriGator Project Report

# Administrative

* Team Name: Aman’s A-Listers
* Team Members: Pablo Hernandez-Perretti, Joseph Fleming, Samuel Falzone
* GitHub URL: <https://github.com/bizononlee/Project-3>
* Video Link: INSERT LINK HERE

# Extended and Refined Proposal

## **2A. Problem**

The problem our project is designed to solve is that of understanding the entertainment industry. Specifically, our project enables users to determine the strength of relationships between actors based on overlaps in relevant work history. In doing so, our project also enables users to make more educated decisions regarding which new titles/genres to explore based on the interconnectedness of new actors with those they know and love. Moreover, our project will be solving the question of which SSSP algorithm, between Dijkstra’s Algorithm and the Bellman-Ford Algorithm, performs best in a social-network type of application such as this one. Additionally, our project includes a comparison of DFS and BFS in detecting whether any connection exists between two actors (based on mutual connections in work history); providing simpler and computationally cheaper output to those who would like a binary characterization of the relationship between two actors. We have utilized parallel data structures and all other programmatic constructs to isolate algorithmic performance for our comparison.

## **2B. Motivation**

## Cinema fanatics who appreciate the brilliant performances in their favorite films are often searching for new content to consume that is in line with their interests. Doing so typically entails the costly (time and money) trial and error of the various new films or performances that circulate at the forefront of social media platforms. Our project enables fans to swiftly verify the probability that any given movie will align with their interests based on confirmation of the connectedness of leading cast members. Moreover, countless cinema fanatics keep themselves up to date on the latest celebrity gossip; our project enables them to better understand the professional underlay that likely drove many popular celebrity partnerships and feuds.

## From a technical perspective, our project enables future software engineers to directly compare the performance of two popular SSSP algorithms (Dijkstra’s & Bellman-Ford) in a simplified social network graph that utilizes real-world data. Our project’s stopwatch functionality will enable a direct comparison of the performance of two parallel implementations of these algorithms. This knowledge is a valuable context for those deciding which SSSP algorithm is optimal for their prospective application.

## **2C. Features Implemented**

## Our project implements two primary features: binary characterization of the relationship between two actors, and calculation of the exact strength of the relationship between two actors. Binary characterization of the relationship between two actors entails utilizing both BFS and DFS to determine whether there is any relationship between the two passed-in actors. Results from both the BFS and DFS algorithms are timed, and the efficiency comparison will be outputted to users. Calculating the strength of the relationship between two actors entails running both the Dijkstra’s and Bellman-Ford SSSP algorithms to procure the shortest distance between two actors. Across both features, we have implemented stopwatch functionality that will assess the efficiency of all four algorithms for every unique query. Our project does not have a graphical user interface (GUI), users can only interface with our application through the CLI by typing simple commands or the names of actors they’d like to compare.

## **2D. Data Description**

## Our database of choice for this project can be found at <https://datasets.imdbws.com>, and is downloadable via the hyperlink “name.basics.tsv.gz”. Our database of choice is one of several public databases maintained by my International Movie Database (IMDB). Moreover, our database can be visualized as a table where rows would represent a given actor and columns would represent various attributes of each actor. Specifically, there are over ten million actors included in this database, each of which has six listed attributes in the database. To better align with the restrictions and purpose of this assignment, we altered the database for our own personal dataset to include only the names and most notable previous roles held by roughly ninety-four thousand actors (93,968). We reduced the number of entries in our dataset by sorting actors out who: had passed away, were born before 1950, or had less than 4 notable film appearances throughout their career.

## **2E. Tools, Languages, APIs, & Libraries**

## The entirety of the programming for our implemented algorithms and graph API relies on nothing other than the C++ STL. On the other hand, the source code for manipulating the public IMDB database into our usable dataset is written in Python and utilizes various public libraries and tools to manipulate data more efficiently at such a large scale. Specifically, the source code for generating a usable dataset implements both pandas [1] and NumPy [2], which are extremely popular public libraries containing useful mathematical and data manipulation operations. Additionally, the database alteration source code takes advantage of useful tools such as Jupyter Notebooks [3] and Conda [4], which are common editor and file management systems, respectively.

## **2F. Data Structures & Algorithms**

## The algorithms at the center of our project’s functionality are Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra’s Algorithm, and the Bellman-Ford Algorithm. Each of the previously mentioned algorithms is implemented exclusively utilizing data structures within the C++ STL. The previously mentioned algorithms rely on several common containers in the C++ STL, including unordered\_map, vector, priority\_queue, stack, queue, and set. Each of the previously mentioned has unique performance metrics for varying tasks, when and where a data structure was implemented into the various algorithms will be reflected in the complexity analysis component of this report. Moreover, the graph API itself takes advantage of the unordered\_map data structure to store an adjacency list, as well as to support rapid access of an actor ID based on a given name and vice versa.

## **2G. Distribution of Responsibilities**

## The distribution of responsibilities for the completion of this project was as follows: Pablo Hernandez-Perretti was responsible for generating the dataset, implementing the graph API, implementing DFS, and writing sections 1 and 2 of this report; Joseph Fleming was responsible for implementing Dijkstra’s algorithm, implementing BFS and implementing the application’s CLI UI; Samuel Falzone was responsible for implementing the Bellman-Ford algorithm and writing sections 3 and 4 of this report.

# Analysis

## **3A. Alterations Since Proposal**

## All things considered, there were relatively few alterations from the proposal that were permanent. One of which was the exclusion of extra visual aspects of the UI. Originally it was planned to make a far more visually appealing UI however it was determined that it adds no functionality to the program and as such was not a priority. While the two options were kept the same, it was deemed reasonable to increase the number of closest relationships shown to 5 to account for the average number of connections for each actor/actress. There were also extra limitations put on the data set to remove actors with too few accreditations as well as actors above a certain age. More importantly, any movie with only 1 actor, or more than 20 actors/actresses was excluded from the data set. These changes were made to limit the size of the data set to optimize the program while maintaining a significant dataset. An additional feature was also added which first checks the validity of any entered name before running any graph algorithms to save the user wasted time. Time also allowed us to implement both BFS and DFS as a method to determine if individuals are connected, and Dijkstra and Bellman-Ford were not changed as the method for determining the weight of the paths.

## **3B. Complexity Analysis**

## In this project, there are two distinct sets of algorithms with two different purposes. The first group consists of BFS and DFS which were both used as a method of determining there is a path between two actors/actresses. Both algorithms have a worst-case time complexity of O(E+V) where E is the number of edges in the graph and V is the number of vertices. This is because if the two vertices do not share a path, the algorithms must access all vertices and edges within that connected graph portion. Each algorithm has a specific instance in which they outperform the other. BFS will perform better if the two vertices in question are close to each other, which occurs more often, whereas DFS outperforms when there is a substantial separation between the two vertices. The second group of algorithms consists of Dijkstra’s and Bellman-Ford, which were used to calculate the shortest weight path from a single source vertex to all other vertices. With full implementation, Bellman-Ford has the potential to have a worst-case time complexity of O(V3). However, in this implementation, there cannot be a complete graph, there are no negative weight edges, and a break condition was implemented to stop needless iterations. As such the worst-case time complexity for Bellman-Ford was reduced to O(V\*E) where V and E are the same as in previous time complexities. On the other hand, Dijkstra’s time complexity has the potential to have a worst-case of O(V2), however, for this implementation, Dijkstra’s made use of an adjacency list as well a priority queue. In doing so all edges can be traversed in O(E+V) and the priority queue operations maintain a complexity of O(Log(V)). As such the final worst-case time complexity of this implementation of Dijkstra’s algorithm is O(E\*Log(V)). This was reflected in testing made with the group’s code which showed Dijkstra’s consistently outperforming Bellman-Ford by a significant increase. The large difference in time spent on these two algorithms can also be attributed to the fact that in this graph the number of edges far outweigh the number of vertices which is less than favorable for Bellman-Ford.

# Reflection

## **4A. Group Experience**

## Overall, the group’s experience with Project 3 was an incredibly positive one. The final source code was entirely functional, all deadlines were met on time, and there were no major conflicts within the group. All members were constantly searching for ways to improve the implementation and prevent future issues from occurring. In doing so the group was able to make necessary changes along the way while ensuring the highest quality of work from all members. All issues and conflicts of opinions that were encountered were resolved through constructive collaboration and feedback. The only negative aspects of our experience were some troubles with consistent communication and slight issues with the actual content of the project.

## **4B. Challenges**

## In terms of communication, the problems were solely due to consistency. Occasionally a question would go unanswered in the group chat for a short time, slightly disrupting the group’s workflow. There was also some trouble scheduling group meetings which lead to some impromptu meetings. However, these issues were minimal and were expected due to having 3 independent schedules. In terms of the actual content of the project, the biggest issue encountered was with the size of our selected data set. There was a constant struggle to determine how much data could be left out while still maintaining a significant data set as well as high-level optimization. For example, while it was easy to decide that movies with only 1 actor/actress would be left out of the data set, it is much harder to determine if movies with over 50 actors/actresses should be included despite their exponential increase to the data set’s complexity. Pablo’s main challenge was that for each decision to drop a data group, an addition or adjustment needed to be made to the API to properly create the data set. Due to the consistency of these decisions, the changes to the API required an exorbitant amount of time. Samuel and Joseph struggled with the development and testing of the algorithms due to the sheer size of the data. Joseph’s main challenge was caused by communication issues which made combining the various pieces of code into one cohesive program harder as consistent formatting was not established beforehand. Samuel’s biggest challenge was encountered while optimizing an algorithm which led to a substantial amount of time spent trying to fix it individually rather than reaching out to the group for advice.

## **4C. Lessons**

## If the process was restarted, several changes could have been made to improve our experience and the workflow of the project. For example, the filtering conditions used for creating the data set should have been completely established before any code was written to avoid unnecessary changes to preexisting code. Communication could have been greatly improved by creating an agreed-upon schedule for meetings and collaboration at the start of the project. Also, as a rule, if an issue is encountered while working first reach out to teammates for advice in case they already have the answer, which would minimize time wasted. In terms of programming techniques, Pablo learned how to use Python libraries in tandem with C++’s file IO, Joseph learned how to cohesively adapt and merge code while maintaining functionality and quality, and Samuel learned how to optimize algorithms for specific implementations with varying aspects.

# References

[1] pandas, Data Manipulation. <https://pandas.pydata.org/>

[2] NumPy, Mathematical Functions. <https://numpy.org/>

[3] Jupyter Notebooks, Text Editing. <https://jupyter.org/>

[4] Conda, File Management System. <https://conda.io/>