Data Structures Final Project Report

Administrative

Team Members: Sruthika Baviriseaty, Alexis Earley, and Maria Carmona
Link to Video: https://youtu.be/2gKYun6vu5A

Extended and Refined Proposal

Problem:

Our program narrows down possible movies to watch, recommending movies that match with a user's preferences on some criteria. With a combination of these factors and the IMBd ranking, it calculates a match score; the final output returns a user input number of the best-matched movies.

Motivation:

There are 86,000 movies on IMDb, and so it is unsurprising that people often spend long periods of time scrutinizing them and deciding which to watch. This can take away from the precious time that people have to relax and spend time together. As such, our algorithm eases this burden and allows people to just enjoy themselves.

Features implemented:

- From the start, our algorithm took in user preferences for language, genre, duration, and actors, returning top movie recommendations. However, as time went on, we decided to add other criteria, including countries, directors, writers, production company, and minimum/maximum year. Multiple inputs can be added and separated by commas.
- Our program recommends top movies based on a match score derived from the inputted preferences of the user. The match score is found by first calculating an individual match score for each preference (if specified). Later on, it combines these to return a total match score in a range from 0 to 1 (0 being worst match and 1 being best match).
- In both the unordered map and priority queue implementation, the user can set how many movie recommendations they want initially. With the priority queue implementation, the user is also able to request more movies beyond what they originally asked for, either until the user is done or all 86,000 movies have been displayed.
- The user can request more information on any one of their top recommendations; this information includes but is not limited to all actors/actresses, number of votes, average rating, and description.

Description of data:

Both the movies.tsv and ratings.tsv files have around 86,000 rows. (This gives a total of over 100,000 rows, as per the requirement.) The data in the first file contains general information about all of the movies, while the second file contains detailed ratings information for each movie. This spans across different age and gender demographics.While they include data that we used, like language and genre,

they also included irrelevant information that we had to skip through, such as the budget and reviews from critics. (Many of these fields had a number of empty fields, which is why we chose not to use them).

Tools/Languages/APIs/Libraries used:

We worked in C++ (especially making use of fstream to parse the data). Because we wanted to focus our 75 hours on making an efficient and useful program, we had a menu driven program implemented via command line. We also accessed the Console API to change the text color printed to our IDE's terminal.

Data Structures/Algorithms implemented:

The two data structures we implemented ourselves are the **unordered map** and **priority queue** (min and max heap).

- The **unordered map** is based on a hash table. The keys are movie IDs and the values are Movie pointers. It is implemented with chaining using a vector of vectors, which in turn contain pairs of strings (the hash code) and Movie pointers. It is resized after it is 75% full. It is used to store all of the movies in the first implementation. The top values are selected using a min heap, and then this is converted to a max heap. However, since the user would often want only 5-20 recommendations, the time to create these heaps is negligible and this method is quite fast.
    - This data structure inserts and searches in O(1) time. Because of this, with file reading included, inserting 86,000 values into the unordered map took 5.6 seconds. Also with file reading included, inserting 86,000 values into the heap took 20.6 seconds.
- The **priority queues** we made include both a minimum and maximum heap. While small ones are also used with the unordered map option, the priority queue option creates one large heap holding all of the Movie pointers. Popping off the maximum heap enables recommended movies to be printing in descending order for virtually as long as the user likes.
    - This data structure inserts in O(1) time and searches in O(n) time. Because of this, with file reading included, inserting 86,000 values into the unordered map took 15.7 seconds. This is similar to the unordered map. Also with file reading included, inserting 86,000 values into the heap took 992.0 seconds. This is much longer than the unordered map.

Data Structures/Algorithms used:

- Unordered set: this structure is used to store genres, countries, languages, writers, and directors for both Movie and Priority Queue class variables. This structure was used because of its O(1) search time.
- Vectors: this structure is used to store minimum and maximum heaps in Priority Queue class and actors in both the Movie and Priority Queue class variables. A 2D vector with pairs is used for the hash table. These structures were used either for their O(1) element access time or because O(1) search time was not important.
- Arrays: this structure is used to store ratings based on age and gender in a 2D array with 5 rows and 3 columns. This was used for the O(1) element access time and because there was a fixed size.

Distribution of Responsibility and Roles:

- Alexis read in the file data, created the movie class, and implemented the unordered map class.

- Maria determined the match score based on selected user preferences.
- Sruthika implemented the priority queue and created the menu options.

Analysis

Any changes the group made after the proposal:

At the beginning of the project, we were implementing all of our data structures using the C++ STL library because we wanted to focus on creating various functionalities that the user can benefit from. Later on, after talking to the professor, we switched from the STL library to manual implementation of our main data structures.

We also decided to omit the ordered map, which we initially wanted to implement, and instead compare the unordered map to the priority queue. This was done because the ordered map does not add a benefit to our program (it would simply take longer), while the priority queue does. Although it does take longer as well, it allows the user to decide whether they want to see more recommendations or not, which can keep being popped off.

Additionally, we were also going to use two more actor spreadsheets, but later on decided not. This was because it makes our program unnecessarily busy. Ultimately, actor's biographies (such as place of birth) will not help users pick a movie to watch.

Complexity analysis of the major functions/features you implemented in terms of Big O for the worst case:

Parsing through each file in the functions buildMinQueue() and MovieNaviGator() takes O(r) time in the best, average, adn worst cases. Here r is the number of rows in the file. This is because it iterates through each row individually.

The matchscore function utilizes two helper functions. The first one (matchesifelse) takes O(n * m) time in the worst case, where n is the number of the characteristics of the current movie, and m is the number of preferences of the user. In this function we have to visit each characteristic of the current movie to check whether it matches any of the preferences of the user. We then generate a match score for each preference that matches.

The other helper function (totalscorecalc) is O(n) time, as it has to count the number of items in each characteristic of the current movie (i.e. how many languages are in the vector of languages, etc.). Due to these two nested helper functions, the time complexity of the overall function, matchscore, is O(n^2 * m) where n is the characteristics of the current movie and m is the preferences of the user.

Inserting (insert()) and searching (find()) within the unordered map takes O(1) time because of the hash map implementation. Although it is based on a vector of vectors, and each item in the nested vector must be examined within the find() function, each nested vector should have about one object. Thus it maintains this time complexity.

Heapifying a heap takes O(log n), so subsequently both the functions with pushing and popping from the heap take O(log n) time. The function that shows all the elements in the heap takes O(n log n) time. This

is because accessing the top takes O(1) time, but popping takes O(log n) for each element. The function that searches in a heap takes O(n), as iterating through a vector takes linear time.

Reflection

As a group, how was the overall experience for the project?:

It was enjoyable, as we got a lot of insight on how commonly used data structures are implemented. Both our unordered map and priority queue implementations are specific to Movie* objects, so we can imagine how much more complicated it would be to consider all data types. In addition, all of us are avid movie-watchers, so it is interesting to see the basics of a recommendation engine. Our program does only half of the work by searching for movies that match a user's preferences, but a more advanced algorithm would be able to infer from a user's watch history.

Did you have any challenges? If so, describe.:

The project was challenging at first because we misunderstood the assignment. Instead of implementing the data structures ourselves, we thought we could just use C++ STL libraries. As such, it was challenging to suddenly pivot and consider how we could design these data structures ourselves. As I (Alexis) struggled with understanding hash maps at first, implementing one was difficult. However, I definitely learned a lot, as will be described.

Furthermore, as we had a number of people working on the project, it was a little challenging to make sure we were all on the same page. Although we did well, especially in the midst of exams, there were a few miscommunications about implementations that had to be corrected. For instance, the unordered map was mistakenly integrated into the code to be used for both implementations. As the goal was to only use it for one and to use exclusively a large heap for the other, this had to be corrected. Also, a few pieces of code had to be rewritten, as people were working on them at the same time and features were lost during git pushes.

At first, it was hard to understand how a minimum heap could be used to store the largest K elements. But, I (Sruthika) later understood that either as long there is capacity or an element is larger than the smallest element in a min heap, then it accumulates the top some percentile of all elements. Additionally, it was hard to adapt a traditional heap with simple data types to one that can hold objects; I had to overwrite comparing simply the integer magnitudes to comparing class variable magnitudes.

If you were to start once again as a group, any changes you would make to the project and/or workflow?:

If we had to start over, we could start with the Movie class first, then each data structure, then the MovieNaviGator class, then the match score, and finally the menu. Trying to do the menu simultaneously with other parts of the project meant that it kept having to be changed. Also, creating the data structures near the end meant that some syntax had to be rewritten based on what we called the functions of these data structures.

We would also make some changes to the project if we had to restart. It would have been nice to account for case-sensitive input in the menu. Also, although we didn't have time, it would have been great to implement a better UI with drag-down options.

Comment on what each of the members learned through this process:

- **Maria:** Through this process I specifically expanded my knowledge on the usefulness of the heap and the map data structures. Additionally, this project taught me how to think more efficiently to make my implementations less complex and expensive.
- **Sruthika:** I got a better understanding of the pros and cons of various data structures. The unordered map is far better in terms of fast retrieval, but the priority queue is good for designating importance to different elements. Working on this project also got me thinking about how many applications I see daily using a similar framework. The YouTube recommended section uses a user's watch history, but it ultimately aligns in terms of the topic, view count, and watch time. These are similar to our genre, votes, and duration metrics that we considered in our match score algorithm.
- **Alexis:** Before this project, I truly did not understand hash tables. Building one myself allowed me to really understand how they work, especially the O(1) time complexity. I also learned a lot about other data structures and their benefits. Because our data file was so large, things had to be as efficient as possible. As such, we really had to understand these structures and weigh their benefits. I also definitely developed a greater ability for working on coding with a team. Communicating and pushing to GitHub is essential!

References

https://www.techiedelight.com/min-heap-max-heap-implementation-c/