**ENGINEERING METHOD INTEGRATIVE TASK 2 CYED1**

**JUAN FELIPE JOJOA CRESPO A00382042**

**FELIPE ROJAS PRADO A00393918**

**WILLIAM JOSEPH VERDESOTO**

**Problem statement:**

The well-known company MEGA-MOVIES needs you and your group of programmers to develop a movie recommendation system to offer personalized suggestions to users.

The system should allow users to enter their movie preferences and, using algorithms, find similar movies and recommend them based on other users' preferences and interactions.

The system should allow users to search for movies by genre, director, actors, etc. Also, you should provide an automatic recommendation feature that suggests movies based on user preferences and similarities between movies.

The system should also offer an intuitive graphical user interface that allows users to browse and discover movies, view details of each recommended movie, and add movies to their favorites list.

In addition, the company wants to know what is the scope and effectiveness of the software to recommend a movie to a person taking into account the filters and the user's selections. (That is, if a movie can be related to another through filters).

**Phase 1: Problem Understanding**

**Problem Description:**

The goal is to develop a movie recommendation system that provides personalized suggestions to users. The system should allow users to input their movie preferences and,, find similar movies and recommend them based on user preferences and interactions with other users.

The system should also allow users to search for movies by genre, director, actors, etc. Additionally, an automatic recommendation feature should suggest movies based on user preferences and similarities between movies. The system should provide an intuitive graphical user interface that enables users to explore and discover movies, view details of recommended movies, and add movies to their favorites list. In addition, the company wants to know what is the scope and effectiveness of the software to recommend a movie to a person taking into account the filters and the user's selections. (That is, if a movie can be related to another through filters).

**Objectives:**

* Develop a movie recommendation system.
* Provide personalized movie suggestions based on user preferences and similarities between movies.
* Enable users to search for movies based on various criteria such as genre, director, actors, etc.
* Include an automatic recommendation feature that suggests movies based on user preferences and movie similarities.
* Create a user-friendly graphical user interface for easy movie exploration, detailed movie viewing, and adding movies to favorites.

**Scope:**

The movie recommendation system will focus on providing personalized movie suggestions to users based on their preferences. It will allow users to search for movies based on different criteria and include an automatic recommendation feature. The system will also provide a user-friendly graphical interface for seamless movie exploration and interaction.

**Stakeholders:**

* **Users:** Individuals interested in discovering and receiving movie recommendations.
* **System Administrators:** Responsible for system maintenance, updates, and ensuring smooth operation.
* **Movie Database Providers:** The system may require access to a comprehensive movie database for accurate recommendations.

**Constraints:**

* **Performance:** The system should handle a large number of users and movies efficiently to provide real-time recommendations.
* **Privacy:** User preferences and personal information should be securely stored and protected.
* **Availability:** The system should be available and accessible to users without significant downtime.
* **Compatibility:** The graphical user interface should be compatible with various platforms and devices.
* **Maintainability:** The system code should be well-structured, documented, and easily maintainable for future updates and enhancements.

**Functional Requirements:**

**RF1: User registration:** Allow users to create an account in the system.

**RF2 Login:** Allow users to access the system with their credentials.

**RF3 Movie Preferences Entry:** Allow users to enter their movie preferences, such as genre, director, actors, etc.

**RF4 Movie Search:** Allow users to search for movies by genre, director, actors, etc.

**RF5 Personalized Recommendations:** Find similar movies and generate recommendations based on user preferences and similarities between movies.

**RF6 Auto recommendation feature:** Provide a feature that automatically suggests movies based on user preferences and similarities between movies.

**RF7 View movie details:** Allow users to view detailed information for each recommended movie, such as synopsis, cast, duration, etc.

**RF8 Add movies to favorites:** Allow users to add movies to their favorites list for easy access in the future.

**RF9 Graphical User Interface (Optional):** Provide an intuitive graphical interface that allows users to browse and discover movies, interact with system functions, and perform actions such as search, view details, and add to favorites.

**Non-functional Requirements:**

**RNF1 Usability:** The system must be easy to use and understandable for users.

**RNF2 Efficiency:** The algorithms used to find similarities and generate recommendations must be efficient at runtime.

**RNF3 Scalability:** The system must be able to handle a large number of users and movies without degrading its performance.

**RNF4 Security:** The system must guarantee the confidentiality of user information and protect against possible vulnerabilities.

**RNF5 Availability:** The system must be available and accessible to users at all times, with minimal downtime.

**RNF6 Adaptability:** The system must be adaptable to different platforms and devices, such as computers, tablets, and mobile devices.

**RNF7 Maintainability:** System code must be well structured, documented, and easy to maintain and update in the future.

**REQUIREMENTS TABLE INTEGRATIVE TASK 2 CYED 1**

| Client | MEGA-MOVIES |
| --- | --- |
| User | Administrator, Movie watcher |
| Functional requirements | **RF1 User registration:** Allow users to create an account in the system.  **RF2 Login:** Allow users to access the system with their credentials.  **RF3 Movie Preferences Entry:** Allow users to enter their movie preferences, such as genre, director, actors, etc.  **RF4 Movie Search:** Allow users to search for movies by genre, director, actors, etc.  **RF5 Personalized Recommendations:** Find similar movies and generate recommendations based on user preferences and similarities between movies.  **RF6 Auto recommendation feature:** Provide a feature that automatically suggests movies based on user preferences and similarities between movies.  **RF7 View movie details:** Allow users to view detailed information for each recommended movie, such as synopsis, cast, duration, etc.  **RF8 Add movies to favorites:** Allow users to add movies to their favorites list for easy access in the future.  **RF9 Graphical User Interface (Optional):** Provide an intuitive graphical interface that allows users to browse and discover movies, interact with system functions, and perform actions such as search, view details, and add to favorites. |
| Problem context | The MEGA-MOVIES company needs you to develop a movie recommendation system to offer personalized suggestions to users, with the aim of facilitating the search and discovery of movies based on their preferences and similarities with other movies. |
| Non-functional requirements | **RNF1 Usability:** The system must be easy to use and understandable for users.  **RNF2 Efficiency:** The algorithms used to find similarities and generate recommendations must be efficient at runtime.  **RNF3 Scalability:** The system must be able to handle a large number of users and movies without degrading its performance.  **RNF4 Security:** The system must guarantee the confidentiality of user information and protect against possible vulnerabilities.  **RNF5 Availability:** The system must be available and accessible to users at all times, with minimal downtime.  **RNF6 Adaptability:** The system must be adaptable to different platforms and devices, such as computers, tablets, and mobile devices.  **RNF7 Maintainability:** System code must be well structured, documented, and easy to maintain and update in the future. |

| Name or identifier | **RF1 User Registration** | | |
| --- | --- | --- | --- |
| Abstract | Allow users to create an account in the system. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| name | String | Can’t be null |
| email | String | Can’t be null |
| password | String | Can’t be null |
| General activities needed to obtain the results | * Collect user information * Validate and verify user data * Create a new user account | | |
| Result or postcondition | Successful user registration. | | |
| Outputs | msgConfirmation | String | Can’t be null |

| Name or identifier | **RF2 Login** | | |
| --- | --- | --- | --- |
| Abstract | Authenticate and log in a user. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| email | String | Can’t be null |
| password | String | Can’t be null |
| General activities needed to obtain the results | * Validate user credentials * Authenticate user * Grant access to the system | | |
| Result or postcondition | Successful user login. | | |
| Outputs | msgAccess | String | Can’t be null |

| Name or identifier | **RF3 Movie Preferences Entry** | | |
| --- | --- | --- | --- |
| Abstract | Allow users to enter their movie preferences. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| genre | Int | Can’t be null or less than 0 |
| director | String | Can’t be null |
| actor | String | Can’t be null |
| General activities needed to obtain the results | * Validate and store user preferences. * Analyze preferences to generate personalized recommendations. | | |
| Result or postcondition | User preferences stored and ready for recommendation generation. | | |
| Outputs | msgConfirmation | String | Can’t be null |

| Name or identifier | **RF4 Movie Search** | | |
| --- | --- | --- | --- |
| Abstract | Allow users to search for movies based on specified criteria. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| genre | Int | Can’t be null or less than 0 |
| director | String | Can’t be null |
| actor | String | Can’t be null |
| General activities needed to obtain the results | * Analyze the search criteria. * Retrieve movies matching the criteria. | | |
| Result or postcondition | Movies matching the search criteria are retrieved. | | |
| Outputs | listMovies | String | Can’t be null |

| Name or identifier | **RF5 Personalized Recommendations** | | |
| --- | --- | --- | --- |
| Abstract | Allow users to interact with the recommended movies and perform various actions. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| None | | |
| General activities needed to obtain the results | * Display recommended movies to the user. * Capture user actions on recommended movies. * Update user preferences and movie ratings based on user actions. | | |
| Result or postcondition | User interactions with recommended movies are captured and processed. | | |
| Outputs | updatePreferences | String | Can’t be null |

| Name or identifier | **RF6 Auto recommendation feature** | | |
| --- | --- | --- | --- |
| Abstract | Provide automatic movie recommendations to users based on their preferences and movie interactions. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| userPreferences | String | Can’t be null |
| movieRatings | String | Can’t be null |
| interactions | String | Can’t be null |
| General activities needed to obtain the results | * Analyze user preferences and movie interactions. * Apply recommendation algorithms to generate personalized movie suggestions. * Consider user similarity and movie similarities in the recommendation process. | | |
| Result or postcondition | Automatic movie recommendations are generated based on user data. | | |
| Outputs | listMovies | String | Can’t be null |

| Name or identifier | **RF7 View movie details** | | |
| --- | --- | --- | --- |
| Abstract | Allow users to view detailed information about a selected movie. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| movieID | String | Can’t be null |
| General activities needed to obtain the results | * Retrieve movie details from the database or external source. * Display movie title, genre, director, actors, synopsis, and other relevant information. * Provide options to view movie trailers, ratings, and reviews if available. | | |
| Result or postcondition | Users can access and view detailed information about the selected movie. | | |
| Outputs | movieDetail | String | Can’t be null |

| Name or identifier | **RF8 Add movies to favorites** | | |
| --- | --- | --- | --- |
| Abstract | Allow users to add selected movies to their favorites list. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| movieID | String | Can’t be null |
| General activities needed to obtain the results | * Identify the user and validate their authentication. * Retrieve the selected movie details. * Add the movie to the user's favorites list. * Update the user's favorites list in the database or storage. | | |
| Result or postcondition | Selected movie is added to the user's favorites list. | | |
| Outputs | movieFavList | String | Can’t be null |

| Name or identifier | **RF9 Graphical User Interface (Optional)** | | |
| --- | --- | --- | --- |
| Abstract | Provide a graphical user interface for the movie recommendation system. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| None | | |
| General activities needed to obtain the results | * Design and develop user interface components, including menus, search bars, movie lists, and recommendation displays * Implement user interactions and event handling * Integrate with the backend system to fetch movie data and recommendations | | |
| Result or postcondition | Graphical user interface is successfully displayed and functional. | | |
| Outputs | movieFavList | String | Can’t be null |

**Phase 2:**

* **The most efficient way to know what each person's taste is in movies:** The most efficient way to recommend movies to a person based on general filters may vary depending on that person's specific preferences and requirements. However, here are some general steps you can take:

Get insights into the person's likes and dislikes: Before recommending movies, it's important to understand the genres, styles, and themes the person is interested in. You can ask direct questions or look up previous conversations to get an idea of ​​their preferences.

Consider general filters: When recommending movies, you should keep in mind some general filters that can affect the choices. These filters can include preferred language, movie length, age rating, release year, and more. Be sure to keep these filters in mind when searching for recommendations.

Use reliable sources of information: There are several reliable sources where you can get information about movies. Some of them include:

* IMDb ([www.imdb.com](http://www.imdb.com) ): IMDb is an online database that provides detailed information about movies, including ratings, reviews, and recommendations.
* Rotten Tomatoes ([www.rottentomatoes.com](http://www.rottentomatoes.com) ): Rotten Tomatoes collects reviews from critics and audiences to provide aggregate scores and movie recommendations.
* Metacritic ([www.metacritic.com](http://www.metacritic.com) ): Metacritic assigns scores to movies based on critical reviews and provides recommendations based on those scores.

Use recommendation algorithms: You can take advantage of recommendation algorithms based on the preferences of the person to suggest movies. Streaming platforms such as Netflix, Amazon Prime Video or Spotify use algorithms to recommend content based on each user's viewing and listening history.

Personalized recommendations: Try to make personalized and specific recommendations instead of giving a generic list of movies. It uses the information you've collected about the person's preferences and suggests movies you think they'll like based on that.

**Bibliographic reference:** [**https://www.makeuseof.com/recommendation-sites-for-movies-you-would-hate-to-miss/**](https://www.makeuseof.com/recommendation-sites-for-movies-you-would-hate-to-miss/)

* **Load txt File and GSON Save:** Data load txt and save Gson are two commonly used methods for storing and retrieving data in software applications.

Data load txt refers to the process of loading data from a text file into an application. The text file typically contains data in a structured format, such as comma-separated values or JSON, which can be easily parsed and loaded into the application.

On the other hand, save Gson refers to the process of converting an object or collection of objects into JSON format and then saving it to a file or database. Gson is a popular Java library that provides easy-to-use methods for converting objects to JSON and vice versa.



**Why would we use it?:** Gson can be used for serialization and deserialization of JSON data. It can be utilized to convert the movie data, preferences, and other relevant information into JSON format for storage or communication purposes. Gson provides a convenient way to convert Java objects to their JSON representation and vice versa, making it easier to handle data exchange with external systems or databases.

For example, Gson can be used to serialize user preferences for movies into JSON format before storing them in a database. It can also be used to deserialize JSON responses from external APIs or data sources, allowing the system to parse and utilize the retrieved movie details or recommendations.

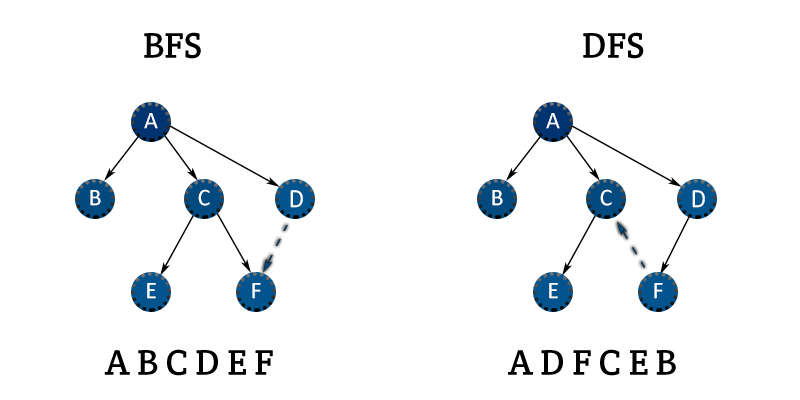
By using Gson, the system can seamlessly convert Java objects to JSON and vice versa, facilitating data transfer and integration within the movie recommendation system.

**Bibliographic reference:** [**https://www.baeldung.com/gson-save-file**](https://www.baeldung.com/gson-save-file)

* **Tours over Graphs (BFS, DFS):** Tours over Graphs, specifically Breadth-First Search (BFS) and Depth-First Search (DFS), are graph traversal algorithms used to explore and navigate through a graph's vertices and edges.

BFS (Breadth-First Search) is an algorithm that starts at a given vertex and explores all its neighboring vertices before moving to the next level of neighbors. It explores the vertices in breadth-first order, meaning it visits all the vertices at distance 1 from the starting vertex, then all the vertices at distance 2, and so on. BFS is often used to find the shortest path between two vertices or to visit all the vertices in a connected component.

DFS (Depth-First Search) is an algorithm that starts at a given vertex and explores as far as possible along each branch before backtracking. It explores the vertices in depth-first order, meaning it goes as deep as possible before exploring other branches. DFS is often used to traverse the entire graph or to search for a specific vertex or path.



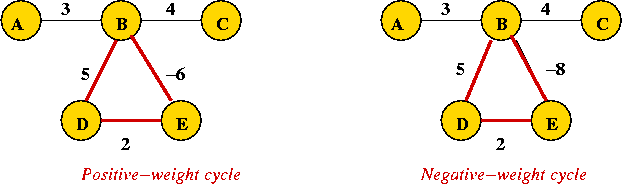
**Why would we use it?:** BFS and DFS can be used to navigate the graph representation of movies and their relationships. These algorithms can help find similar movies, generate recommendations based on user preferences and interactions, and explore the graph to provide relevant movie suggestions.

**Bibliographic reference:** Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

* **Paths of Minimum Weight (Dijkstra, Floyd-Warshall):** Paths of Minimum Weight refer to algorithms used to find the shortest or minimum-weight paths in a graph. Two commonly used algorithms for this purpose are Dijkstra's algorithm and Floyd-Warshall algorithm.

Dijkstra's algorithm is a single-source shortest path algorithm that finds the shortest path from a given source vertex to all other vertices in a weighted graph. It uses a greedy approach, iteratively selecting the vertex with the minimum distance and updating the distances of its neighboring vertices. Dijkstra's algorithm can be used to optimize the delivery routes in the movie recommendation system, finding the most efficient path for delivering movies to users based on factors such as distance or transportation costs.

The Floyd-Warshall algorithm, on the other hand, is an all-pairs shortest path algorithm that finds the shortest paths between all pairs of vertices in a weighted graph. It works by considering all possible intermediate vertices in the paths and updating the shortest distances accordingly. The Floyd-Warshall algorithm can be used to analyze the overall connectivity and distances between movies in the recommendation system, allowing for more comprehensive recommendations based on the overall graph structure.

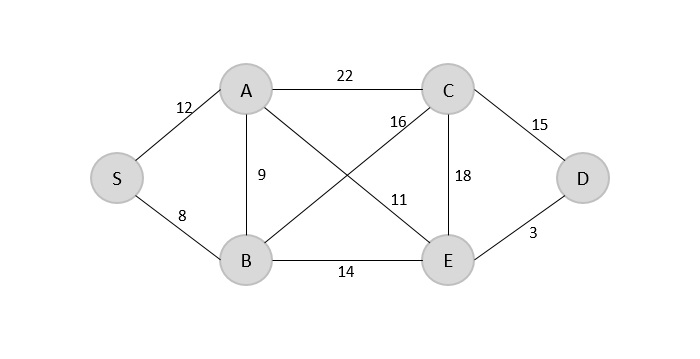


**Bibliographic reference:** Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

* **Minimum Coverage Tree -MST- (Prim, Kruskal):** A Minimum Coverage Tree (MST) refers to a tree that spans all the vertices of a connected, weighted graph with the minimum possible total weight. Two commonly used algorithms for finding the MST are Prim's algorithm and Kruskal's algorithm.

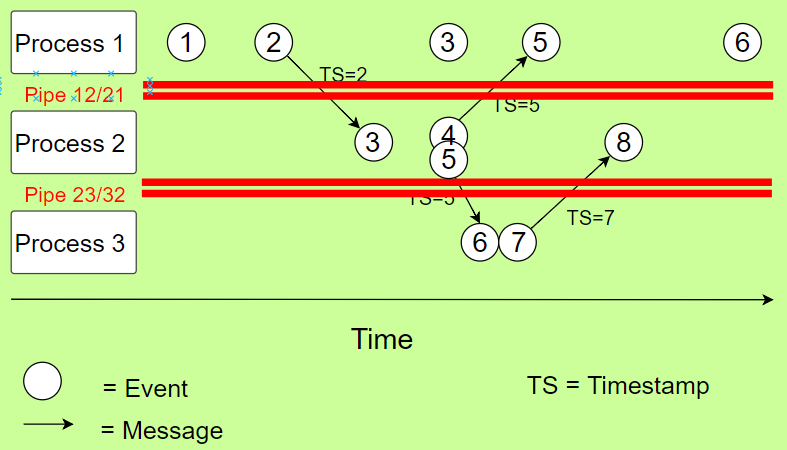
Prim's algorithm is a greedy algorithm that starts with an arbitrary vertex and repeatedly adds the minimum-weight edge that connects a visited vertex to an unvisited vertex. It gradually builds the MST until all vertices are included. Prim's algorithm can be used in the movie recommendation system to construct a minimum coverage tree that represents the relationships between movies based on their similarities or interactions.

Kruskal's algorithm is another greedy algorithm that starts with the edges sorted by their weights and progressively adds the edges with the lowest weight that do not form cycles until all vertices are connected. Kruskal's algorithm can also be employed in the movie recommendation system to construct an MST that captures the minimum-weight relationships between movies.

These MST algorithms can assist in identifying the most influential or central movies in the recommendation system, determining key connections between movies, and optimizing the recommendation process based on the graph structure.

**Bibliographic reference:** Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

* **Data and Time functions:** In computer science, a timestamp is a sequence of characters or encoded information that indicates when an event occurred, usually representing a specific date and time. It's commonly used for logging events or keeping track of when particular operations happened.



**Why would we use it?:** To incorporate date and time functions into the movie recommendation program described in your previous question, you can use them in various ways. Here are some possible use cases:

**Movie Release Dates:** You can store the release dates of movies in your dataset and use date functions to filter and sort movies based on their release dates. For example, you can recommend recently released movies or allow users to search for movies released within a specific time frame.

**User Interaction Timestamps:** If you track user interactions, such as movie ratings or views, you can store timestamps for these interactions. Date and time functions can help you analyze user behavior patterns over time. For instance, you can identify the most popular movies in a specific month or recommend movies that were trending during a particular period.

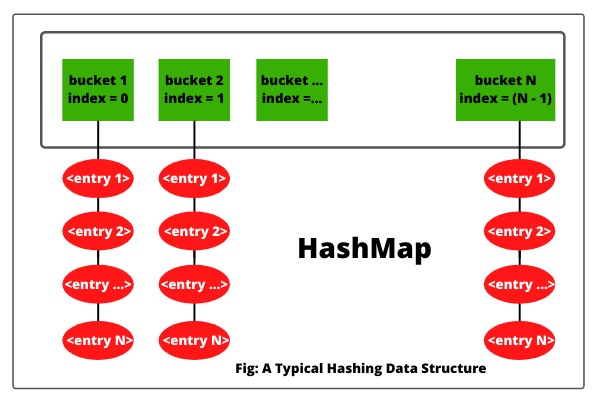
**Personalized Recommendations:** Date and time functions can be used to tailor recommendations based on specific user preferences. For example, if a user prefers classic movies, you can prioritize recommending movies from a certain era or filter out recent releases.

**Scheduled Updates:** You can use date and time functions to schedule regular updates for your movie database. For instance, you can automatically fetch and add new movie releases on a daily or weekly basis, ensuring that your recommendations stay up-to-date.

**Time-based Filtering:** You can allow users to apply time-based filters when searching for movies. For instance, they can specify a specific year range, or filter movies based on seasons, such as holiday-themed movies for Christmas or Halloween.

**Bibliographic reference:** <https://ieeexplore.ieee.org/document/805196>

* **Hashmap:** A HashMap is a data structure used to efficiently store and access data. It works by using keys and values, where each key is assigned a specific value. This allows data to be searched and accessed quickly using the corresponding key.



**Why would we use it?:**

**Efficient Data Retrieval:** HashMaps provide fast and efficient data retrieval. By using a HashMap, you can store movie-related information (such as genre, director, actors, etc.) as key-value pairs. This allows for quick access to specific movie attributes based on the key, resulting in efficient search and retrieval operations.

**Flexibility in Storing Additional Information:** HashMaps allow you to store additional information associated with each movie. Apart from the basic attributes, you can include additional data such as ratings, reviews, or user preferences. This flexibility enables you to incorporate a wide range of information relevant to movie recommendations.

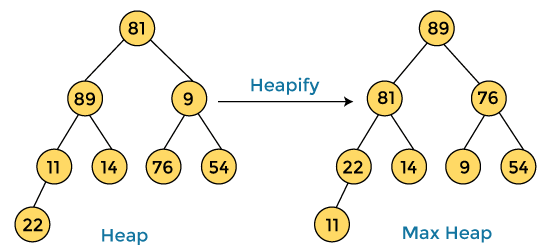
**Simplified Data Structure:** HashMaps offer a simple and intuitive data structure. Each movie can be represented as an entry in the HashMap, with its attributes stored as key-value pairs. This structure allows for easy maintenance and manipulation of movie data within the recommendation system.

**Customization of Movie Attributes:** With a HashMap, you can customize the movie attributes and associate them with specific keys. This enables you to define and utilize attributes that are relevant to your recommendation algorithm and specific user preferences. You can include attributes like release year, language, runtime, or any other criteria that affect the movie recommendation process.

**Efficient Recommendation Generation:** When generating movie recommendations, a HashMap can help optimize the process. By indexing movies based on their attributes, you can efficiently identify similar movies or filter recommendations based on specific user preferences. This can enhance the speed and accuracy of the recommendation algorithm.

**Bibliographic reference:** [**https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)

* **Heapsort:** Heapsort is a sorting algorithm that uses a binary heap data structure to sort elements. The basic idea is to convert the input array into a binary heap, which has the property that the parent node is always greater than its children nodes (max heap). Then, the maximum element is repeatedly removed from the heap and placed at the end of the array, which results in a sorted array in ascending order.



**Bibliographic reference:** <https://www.geeksforgeeks.org/heap-sort/>

**Phase 3: Proposed Solutions:**We have to know that proposing solution options for problems serves to identify and address the obstacles or challenges that arise in a certain context. This can help improve efficiency, reduce costs, increase productivity, improve quality, and ultimately achieve desired goals.

In addition, the solution proposal can also encourage creativity and critical thinking in the people involved, which can lead to significant improvements in the process and greater customer or user satisfaction.

In summary, proposing solution options is an important part of the problem solving process and can lead to significant improvements in the efficiency and effectiveness of any process or situation.

The solutions for improving the movie recommendation system are divided into different types, including Load and Save, Database, Sort, and Date. Depending on specific situations that may arise during the procedure, there are multiple solutions available for each type.

1. **Load and Save**

* Simple text File
* Excel file .csv
* Json in text file

1. **Database**

* Graphs
* HashMap
* HashTable
* Arrays
* ArrayList

1. **Sort**

* HeapSort
* BubbleSort
* CollectionsSort
* DFS And BFS

1. **Date**

* Actual Time Function
* Calendar Function
* Date format Function

**Phase 4: Discarded Solutions**

1. *Load and save:*

* **Excel file:** We assume that this type of format is not valid according to the requirements obtained, we are interested in the simple text file that the initial statement asks us for.

1. *Database:*

* **ArrayList and Array:** We discarded the idea of ​​using ArrayList and Arrays for the following reasons:   
    
  **Inefficiency in searching:** If you need to search for specific elements in an ArrayList, the search time can be inefficient. Linear search in an ArrayList has a time complexity of O(n), which means that the larger the list, the more time it will take to find a specific element.

**Costly insertion and deletion:** If you need to perform frequent insertion or deletion operations on the ArrayList, it can be inefficient. These operations may require reorganization of elements, which can be costly in terms of time and resources.

**Fixed capacity:** Unlike dynamic data structures like ArrayList, graphs can have a variable number of connections and relationships between movies. Using an ArrayList to store the connections may require a fixed size, which can be inconvenient and not scalable.

**Difficulty representing complex relationships:** Graphs can contain complex and multidirectional relationships between movies. Using an ArrayList can complicate the representation of these relationships and make it difficult to track and manipulate the connections between movies.

1. *Sort:*

* **Heapsort, Collection Sort and BubbleSort:** We discarded the idea of ​​using these sorting algorithms for the following reasons:

**HeapSort:** HeapSort has a time complexity of O(n log n) and requires additional memory for building and maintaining the heap. Since you are primarily working with graph-based movie recommendations, where sorting may not be the primary focus, HeapSort may not provide significant advantages in terms of efficiency or functionality.

**BubbleSort:** BubbleSort has a time complexity of O(n^2), making it inefficient for large datasets. It involves swapping adjacent elements repeatedly, which may not be ideal for sorting large lists of movies or for the purpose of movie recommendations.

**CollectionsSort:** Collections.sort() is a general-purpose sorting algorithm provided by the Java Collections framework. While it offers convenience and flexibility, it may not be optimized specifically for the movie recommendation system. Using a more specialized sorting algorithm or considering other graph-based techniques for recommendation may yield better results.

1. *Date:*

* **Date format function:** According to the requirements of the statement that we were able to obtain, it is not appropriate to use this type of format in the implementation of the problem, since we need complete information on the movies.

**Phase 5: Evaluation Criteria:**

In order to determine the optimal solution for implementation, we will develop an evaluation system using String values to simplify the assessment process.

**Evaluation System:**

A = Excellent

B = Good

C = Average

D = Below average

F = Failure or insufficient

* U: Usability
* M: Maintainable
* S: Scalability
* E: Efficiency

*Load and save*

* Text Simple file:

U) B

M) B

S) C

E) F

* Json Text File:

U) A

M) B

S) C

E) B

*Database*

* HashMap:

U) B

M) B

S) C

E) A

* Array:

U) D

M) A

S) F

E) F

*Sort:*

* Heapsort:

U) B

M) B

S) A

E) A

* Collections Sort:

U) D

M) B

S) B

E) D

*Date*

* Current Time:

U) A

M) B

S) B

E) C

* Calendar Time:

U) A

M) B

S) B

E) C

After analyzing the evaluation criteria, we have concluded that the following functions are the most suitable:

HashMaps for their efficiency and scalability, as well as their ability to easily access and store information. This is in contrast to fixed arrays, queues, and stacks which lack direct access to stored data.

Json for loading and saving data due to its user-friendly and familiar nature when compared to txt files, which would require learning new functions.

Heapsort for its efficiency and maintainability compared to collection sort, which would require a comparator function and sorting each time.

Calendar for its familiarity, even though it is as good as the current time function.

**Phase 6: Preparation of reports and specifications:**

class MovieRecommendationSystem:

graph: Graph

function \_\_init\_\_():

graph = Graph() **// Create an empty graph**

function addMovie(movie):

graph.addVertex(movie) **// Add a movie as a vertex to the graph**

function addConnection(movie1, movie2):

graph.addEdge(movie1, movie2) **// Add a connection between two movies as an edge in the graph**

function findSimilarMovies(movie):

similarMovies = graph.getAdjacentVertices(movie)  **// Get movies similar to the given movie**

return similarMovies

function generateRecommendations(userPreferences):

recommendations = []

for preference in userPreferences:

similarMovies = findSimilarMovies(preference)

recommendations += similarMovies

recommendations = removeDuplicates(recommendations) **// Remove duplicate movie recommendations**

return recommendations

function searchMoviesByFilter(filter):

movies = graph.getVerticesByFilter(filter)  **// Get movies that match the given filter**

return movies

function addToFavorites(movie, user):

user.addToFavorites(movie) **// Add the movie to the user's favorites list**

class Graph:

adjacencyMatrix: Matrix

adjacencyList: List

function \_\_init\_\_():

adjacencyMatrix = createEmptyMatrix()  **// Initialize an empty adjacency matrix**

adjacencyList = createEmptyList()  **// Initialize an empty adjacency list**

function addVertex(movie):

vertex = createVertex(movie)

adjacencyList.add(vertex) **// Add the movie as a vertex to the adjacency list**

function addEdge(movie1, movie2):

edge = createEdge(movie1, movie2)

adjacencyMatrix.addEdge(edge) /**/ Add the connection between two movies as an edge in the adjacency matrix**

adjacencyList.addEdge(edge) **// Add the connection between two movies as an edge in the adjacency list**

function getAdjacentVertices(movie):

adjacentVertices = adjacencyList.getAdjacentVertices(movie)  **// Get the movies connected to the given movie**

return adjacentVertices

function getVerticesByFilter(filter):

matchingVertices = adjacencyList.getVerticesByFilter(filter)  **// Get movies that match the given filter**

return matchingVertices

class User:

preferences: List

favorites: List

function \_\_init\_\_():

preferences = createEmptyList() **// Initialize an empty list of user preferences**

favorites = createEmptyList() **// Initialize an empty list of user favorites**

function addToPreferences(movie):

preferences.add(movie)  **// Add the movie to the user's preferences list**

function addToFavorites(movie):

favorites.add(movie)  **// Add the movie to the user's favorites list**