

Assignment

COURSE: DATA STRUCTURES AND ALGORITHMS

CODE: 504008

TOPIC: Analyze the movie ratings of viewers

Version: 1.2 – Date: 23/11/2023

(Students carefully read all instructions before conducting)

I. Problem statement

This assignment aims to analyze the dataset of the viewer (user) ratings of movies. We use a graph model to capture the relationships between the films and the viewers. Then, we execute several analytical queries to obtain meaningful user insights from the graph.

Required functionalities for the system are listed below:

- Read data from existing files and build a graph based on this data.
- Develop functions that can extract information from graph structures based on the specific requirements of the problem.

II. Resources

The attached source code includes:

- Data files and expected output files:
 - o The *data* folder consists of 03 files, including:
 - *user.csv*: information of viewers.
 - *movie.csv*: information on movies.
 - *rating.csv*: information on viewers' ratings.
 - o The *expected_output* folder consists of 07 outputs from *Req1.txt* to *Req7.txt*, which are expected results corresponding to 07 requirements.
- Source code files:
 - o *Main.java*: containing the **main** method that calls necessary methods to test.
 - o *Movie.java* and *User.java*: Movie and User classes respectively. **STUDENTS DO NOT MODIFY THESE FILES.**
 - o *RatingManagement.java*: The **RatingManagement** class has an attribute of *rating* for storing the list of viewers' ratings, *movies* storing the list of movies, and *users* storing the list of viewers in the system. This class provides the initialization method and the

getter method. The students are required to complete the blank methods in this file, without modifying the existing methods.

III. Assignment Procedure

- Students download the source code.
- Students carefully read and understand the descriptions and the provided files, then implement the **Rating** class and integrate it with the **RatingManagement** class, following the class diagram and the specifications in the subsequent sections.
- After finishing the above classes, students compile and execute the program using the **main** method in the *Main.java* file to test the outputs of requirements.
 - To execute each requirement, students must first compile the code and then run the Main file with the proper parameter for the desired task. For instance, to run Requirement 1:

java Main 1

- To execute multiple requirements, the student must first compile the code and then run the Main file with the appropriate parameters for the desired requirements. For instance, to run Requirement 1, 2, and 3:

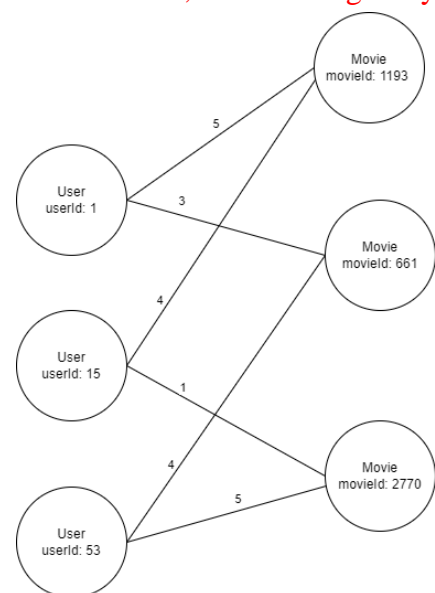
java Main 1 2 3

Students complete the following tasks and compare your work with the results provided in the *expected_output* folder.

- For requirements that students are not able to complete, do not delete related methods and ensure the program can be executed successfully given the main method in *Main.java*.
- The Google Drive folder of the assignment also consists of *version.txt*, students regularly check for updates via this file. If there are any updates, students carefully read the descriptions and download the newest resources. This file provides updated contents and dates in case the assignment has any changes.

IV. Descriptions of the problem

The data on the ratings of movies given by viewers will be represented as a bipartite graph, which has two sets of nodes: one for the users and one for the movies. In this assignment, students will construct a graph and save it in an Edge List format. The graph has approximately 10,000 nodes and over 1,000,000 edges (the exact numbers will be determined after completing Requirement 1).



V. Descriptions of Classes

- Descriptions of class attributes and methods are below:
 - The class attributes are always declared with the private access modifier.

User
- id: int - gender: String - age: int - occupation: String - zipCode: String
+ User(id: int, gender: String, age: int, occupation: String, zipCode: String) + getId(): int + setId(id: int): void + getGender(): String + setGender(gender: String): void + getAge(): int + setAge(age: int): void + getOccupation(): String + setOccupation(occupation: String): void + getZipCode(): String + setZipCode(zipCode: String): void + toString(): String

- **User class (STUDENTS DO NOT MODIFY THIS CLASS):**
 - Including 05 attributes:
 - id: int – identity number of the viewer
 - gender: String – gender of the viewer
 - age: int – range of age
 - occupation: String – viewer' job
 - zipCode: String – zip code
 - Methods:
 - The method fully initializes parameters in the order of the above 05 attributes.
 - The getter/setter methods are fully implemented for all attributes.
 - **toString()** method: according to format that is **User[identity number of viewer, gender, age group, occupation, zip code]**

Movie
- id: int - name: String - genres: ArrayList<String>
+ Movie(id: int, name: String, genres: ArrayList<String>) + getId(): int + getName(): String + getGenres(): ArrayList<String> + setId(id: int): void + setName(name: String): void + toString(): String

- **Movie class (STUDENTS DO NOT MODIFY THIS CLASS):**
 - Including 03 attributes:
 - id: int – identity number of movie
 - name: String – movie’s title
 - genres: ArrayList<String> – list of movie’s genres
 - Methods:
 - The method fully initializes parameters in the order of the above 03 attributes.
 - The getter/setter methods are fully implemented for all attributes.
 - **toString()** method: according to format that is **Movie[identity number of movie, movie’s title, list of movie’s genres]**

Rating
- attr1: int - attr2: int - attr3: int - attr4: long
+ Rating(userId: int, movieId: int, rating: int, timestamps: long) + toString(): String

- **Rating class – viewer’s rating for the movie (Students implement this class)**
 - Including 04 attributes:
 - Attribute 1: int – identity number of viewer
 - Attribute 2: int – identity number of movie
 - Attribute 3: int – rating star

- Attribute 4: long – timestamp of the movie’s rating with a larger value corresponding to a more recent submission date.

Students name the attributes by yourself.

- Methods:
 - The method fully initializes parameters in the order of the above 04 attributes in order (identity number of viewer, identity number of movie, rating star, timestamp).
 - The getter/setter methods are fully implemented for all properties.
 - **toString()** method: according to format that is **Rating[identity number of viewer, identity number of movie, rating star, timestamp]**

RatingManagement
- ratings: ArrayList<Rating> - movies: ArrayList<Movie> - users: ArrayList<User>
+ MovieManagement(moviePath: String, ratingPath: String, userPath: String) - loadEdgeList(ratingPath: String): ArrayList<Rating> - loadMovies(moviePath: String): ArrayList<Movie> - loadUsers(userPath: String): ArrayList<User> + getMovies(): ArrayList<Movie> + getUsers(): ArrayList<User> + getRating(): ArrayList<Rating> + findMoviesByNameAndMatchRating(userId: int, rating: int): ArrayList<Movie> + findUsersHavingSameRatingWithUser(userId: int, movieId: int): ArrayList<User> + findMoviesNameHavingSameReputation(): ArrayList<String> + findMoviesMatchOccupationAndGender(occupation: String, gender: String, k: int, rating: int): ArrayList<String> + findMoviesByOccupationAndLessThanRating(occupation: String, k: int, rating: int): ArrayList<String> + findMoviesMatchLatestMovieOf(userId: int, rating: int, k: int): ArrayList<String>

- **RatingManagement** class – Analyze the cinematic preferences and behaviors of viewers:
 - Including 03 attributes:
 - ratings: ArrayList<Rating> – The edge list of graphs
 - movies: ArrayList<String> – The list of movies
 - users: ArrayList<String> – The list of viewers
 - Methods:
 - The initialization method accepts three parameters that specify the location of the data files in comma-separated values (CSV) format.
 - The getter/setter methods are fully implemented for all attributes.
 - 09 corresponding methods address the requirements ranging from 1 to 7.

VI. Descriptions of input and output files

- The *movies.csv* file stores information about various movies. Each line in the file represents a movie and its attributes which are separated by commas (“,”) and based on format:

identity number of movie, movie's title, genres

Notice: The movie genres can have multiple values separated by hyphens (“-”). The order of the attributes follows the header row at the beginning of the file.

```
MovieID, Title, Genres
1, Toy Story (1995), Animation-Children's-Comedy
2, Jumanji (1995), Adventure-Children's-Fantasy
3, Grumpier Old Men (1995), Comedy-Romance
4, Waiting to Exhale (1995), Comedy-Drama
5, Father of the Bride Part II (1995), Comedy
```

For example, the first data line is the movie having the identity number of the movie of 1, the movie's title of *Toy Story (1995)*, and genres: *Animation, Children's, and Comedy*.

- The *users.csv* file stores information about various viewers in the system. Each line in the file represents a single viewer and its attributes which are separated by commas (“,”) and based on format:

identity number of viewer, gender, range of age, occupation, zip code

Notice: The symbols “M” and “F” are used as abbreviations for male and female, respectively. The order of the attributes follows the header row at the beginning of the file.

```
UserID, Gender, Age, Occupation, Zip-code
1, F, 1, K-12 student, 48067
2, M, 56, Self-employed, 70072
3, M, 25, Scientist, 55117
4, M, 45, Executive/managerial, 02460
5, M, 25, Writer, 55455
```

For example, the first data line is the viewer having the identity number of the viewer of 1, the gender of *female*, the age group of 1, the occupation of *K-12 student*, and the zip code of 48067.

- The *ratings.csv* file stores information about various viewers' movie ratings. Each line in the file represents a rating and its attributes which are separated by commas (“,”) and based on format:

identity number of viewer, identity number of movie, rating star, timestamp

Notice: The ratings are made on a 5-star scale (whole-star ratings only, such as 1 star or 2 stars). The order of the attributes follows the header row at the beginning of the file.

```
UserID,MovieID,Rating,TimeStamp  
1,1193,5,978300760  
1,661,3,978302109  
1,914,3,978301968  
1,3408,4,978300275  
1,2355,5,978824291
```

For example, the first data line is the rating of the viewer having the identity number of *1*, for the movie having the identifying number of *1193*, with a score of *5 stars*, and at a specific time that the viewer rating is *978300760* (converted to 01/01/2001, 5:12:40 AM, GMT+7).

- The *expected_output* folder consists of 07 results files corresponding to 07 tasks:
 - o For Task 2 and 3: Each line in each Req2.txt and Req3.txt corresponding to an object is formatted according to the **toString()** method of the **respective** class.
 - o For the remaining tasks: Each line in each file .txt is the title of the movie corresponding to the requirements of the problem.

Notice:

- Students carefully read the **main** method to find out approaches to complete tasks in the designated order.
- Students may add extra actions in the **main** method to verify completed methods; however, you must ensure **the program can be executed successfully with the original main method**.
- Students may add extra methods to support your work; however, new methods must be declared and implemented in files for submission and the submission can be executed successfully with the given *Main.java*.
- Students avoid using absolute paths when defining methods that involve reading files. Since absolute paths are specific to a particular system, they may prevent the marking process from accessing the file correctly, resulting in a compilation error.
- **DO NOT MODIFY CLASS AND METHOD NAMES, STRICTLY FOLLOW THE GIVEN CLASS DIAGRAM.**

VII. Requirements

Besides the libraries provided in the files received, students have permission to utilize the classes in **java.util** package and the necessary classes for reading file tasks in the **java.io** package. However, students are not allowed to import any other libraries that are not explicitly mentioned above.

Students conduct the assignment in Java 11 or Java 8. Students are not allowed to use var data type. Submissions are judged in Java 11 and students take responsibility for errors caused by Java version differences.

To facilitate the work, students may define additional methods within the designated files. However, **students must not modify the names and input parameters of the existing methods and the methods specified in the paper, as this would compromise the evaluation process.**

In the **Description of Classes** section of the lesson, some attributes of the class are assigned a number. When students implement this class, they should choose an appropriate name and data type that matches the description. Additionally, they should ensure that the *access modifier* for all attributes is set to *private*.

The large volume of data poses a challenge for students who need to process requests efficiently. Students should monitor the program's running time and apply algorithmic optimization techniques when necessary to reduce it. The marking program has a strict time limit for each student's work, and any work that exceeds this limit will receive 0.0 points.

1. REQUIREMENT 1 (1 point)

Students define the methods in the **RatingManagement** class and use the **Main** class to evaluate the results of the work.

Students implement the method:

private ArrayList<Movie> loadMovies(String moviePath)

private ArrayList<User> loadUsers(String userPath)

public ArrayList<Rating> loadEdgeList(String ratingPath)

The method reads data from files located in a specific directory. The method takes three parameters: *ratingPath*, *moviePath*, and *userPath*, which specify the paths of the files containing the rating, movie, and viewer data, respectively. The method stores the data in the existing attributes in the class. The method also uses a function called **loadEdgeList** to add the rating data to an Edge List.

Students implement the proposed method, compile the *Main.java* file, and execute the command *java Main 1* to write the result to the *Req1.txt* file in the *output* directory. The output file contains the number of vertices and edges in the graph, as computed by the implemented methods. Students should verify the correctness of their output by comparing it with the sample output in the *Req1.txt* file in the *expected_output* folder.

Notice: This is the task that students must complete correctly to gain scores from later tasks (from Requirement 2 to Task 7). If the reading file process does not convert data into the edge list of the graph, the requirements below are not scored.

The following tasks apply transformations to the information extracted from the file.

2. REQUIREMENT 2 (2 points)

Students implement the method:

```
public ArrayList<Movie> findMoviesByNameAndMatchRating(int userId, int rating)
```

to return a list of movies rated by a viewer whose *userId* is greater than or equal to the *rating* score. As a result, the movies must be named alphabetically.

Students implement the proposed method, compile the *Main.java* file, and execute the command *java Main 2* to write the result to the *Req2.txt* file in the *output* directory. Students should verify the correctness of their output by comparing it with the sample output in the *Req2.txt* file in the *expected_output* folder.

3. REQUIREMENT 3 (2 points)

Students implement the method:

```
public ArrayList<User> findUsersHavingSameRatingWithUser(int userId, int movieId)
```

to return a list of viewers provided that these viewers rate a movie with a *movieId* code that has the same number of stars as the viewer with the code *userId* rated for that *movieId*.

Students implement the proposed method, compile the *Main.java* file, and execute the command *java Main 3* to write the result to the *Req3.txt* file in the *output* directory. Students should verify the correctness of their output by comparing it with the sample output in the *Req3.txt* file in the *expected_output* folder.

4. REQUIREMENT 4 (2 points)

Students implement the method:

```
public ArrayList<String> findMoviesNameHavingSameReputation()
```

to return a list of movie titles favored by at least any two viewers (the definition of favorite is rated greater than 3 stars). As a result, the movies must be ordered alphabetically.

Students implement the proposed method, compile the *Main.java* file, and execute the command *java Main 4* to write the result to the *Req4.txt* file in the *output* directory. Students should verify the correctness of their output by comparing it with the sample output in the *Req4.txt* file in the *expected_output* folder.

5. REQUIREMENT 5 (1 point)

Students implement the method:

```
public ArrayList<String> findMoviesMatchOccupationAndGender(String occupation, String  
gender, int k, int rating)
```

to return a list containing up to the *k* of movie names of users with the same *occupation* and the same *gender* with the same *rating* score. The results must be arranged alphabetically.

Students implement the proposed method, compile the *Main.java* file, and execute the command *java Main 5* to write the result to the *Req5.txt* file in the *output* directory. Students should verify the correctness of their output by comparing it with the sample output in the *Req5.txt* file in the *expected_output* folder.

6. REQUIREMENT 6 (1 point)

Students implement the method:

```
public ArrayList<String> findMoviesByOccupationAndLessThanRating(String occupation,  
int k, int rating)
```

to return a list containing up to the *k* of movie titles rated less than the *rating* by viewers with the same *occupation*. The results must be arranged alphabetically.

Students implement the proposed method, compile the *Main.java* file, and execute the command *java Main 6* to write the result to the *Req6.txt* file in the *output* directory. Students should verify the correctness of their output by comparing it with the sample output in the *Req6.txt* file in the *expected_output* folder.

7. REQUIREMENT 7 (1 point)

Students implement the method:














```
public ArrayList<String> findMoviesMatchLatestMovieOf(int userId, int rating, int k)
```

to return a list containing up to *k* movie titles that are rated greater than or equal to the *rating* by viewers of the same *gender* with viewer having *userId* and these movies share at least 01 genre with the latest movie reviewed by the user with the *userId* (the “latest” time is considered by the *timestamp* attribute) whose review score is greater than or equal to the *rating*.

Students implement the proposed method, compile the *Main.java* file, and execute the command *java Main 7* to write the result to the *Req7.txt* file in the *output* directory. Students should verify the correctness of their output by comparing it with the sample output in the *Req7.txt* file in the *expected_output* folder.

VIII. Notices for submission

- If students are not able to accomplish a task, then remain in the original state of the method. DO NOT DELETE THE METHOD OF THE TASK because it causes compile errors for the **main** method. Students must test whether your implementation can be compiled and executed successfully with the original **main** method before submitting.
- Do not use absolute paths in your work.
- All output files, outputX.txt for $X = \{1,2,3,4,5,6,7\}$, are written in the *output* folder which is in the same folder as the given file source code files.
- For students that use IDEs (Eclipse, NetBeans, etc.), you must ensure the program can be compiled and executed command prompt, do not use packages, and outputX.txt must be in the *output* folder. **If the work is placed in the package, students will get 0.0 points for the whole assignment.**
- An example structure of the assignment organized correctly:

Name	Type	Size
 data	File folder	
 expected_output	File folder	
 output	File folder	
 Main.class	CLASS File	4 KB
 Main.java	JAVA File	4 KB
 Movie.class	CLASS File	2 KB
 Movie.java	JAVA File	1 KB
 Rating.class	CLASS File	2 KB
 Rating.java	JAVA File	2 KB
 RatingManagement.class	CLASS File	10 KB
 RatingManagement.java	JAVA File	14 KB
 User.class	CLASS File	2 KB
 User.java	JAVA File	2 KB

- Inside *output* folder:

Name	Type	Size
Req1.txt	Text Document	1 KB
Req2.txt	Text Document	2 KB
Req3.txt	Text Document	39 KB
Req4.txt	Text Document	86 KB
Req5.txt	Text Document	1 KB
Req6.txt	Text Document	1 KB
Req7.txt	Text Document	1 KB

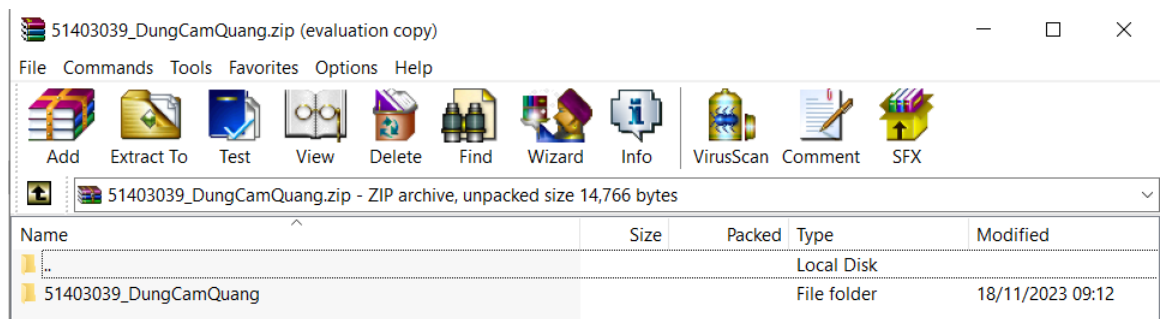
IX. Submission instructions

- Students submit *Rating.java*, and *RatingManagement.java*, **DO NOT RENAME THE TWO DESIGNATED FILES AND DO NOT SUBMIT ANY OTHER FILES**.
- **Students copy the two designated files to a folder named StudentID_FullName** (FullName: no separators, no Vietnamese accents), then compress them as a **zip** file and submit them by the deadline.
- For any careless mistakes, including but not limited to wrong folder names, not placing files under the designated folder, redundant files, etc. then students get **0.0** points.
- A correct submission is as follows:

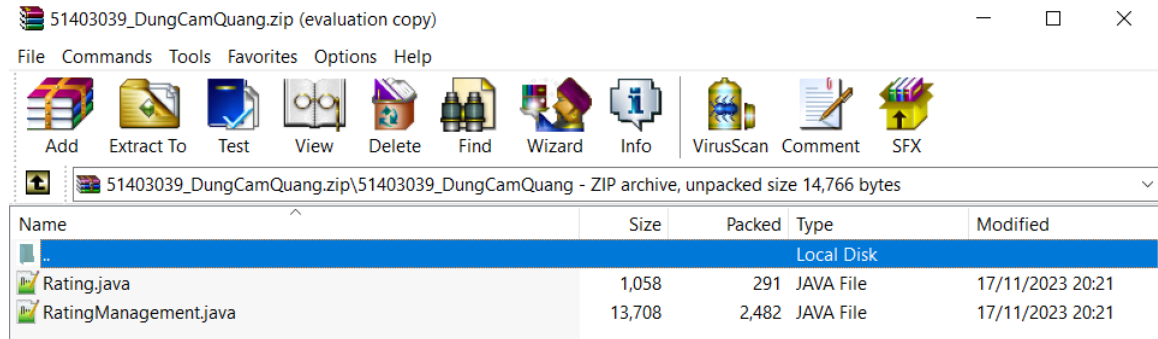
- o Submission file:

 51403039_DungCamQuang.zip 18/11/2023 09:13 WinRAR ZIP archive 4 KB

- o Inside the compressed folder:



- Inside the folder:



X. Evaluation and policy

- Since submissions are judged automatically using test cases (input and output file structures are the same as the descriptions in the assignment), students take responsibility if not follow the instructions or cause compile errors.
- *All the cases in which students hard assign the link during the process of reading the file will get 0.0 points for the whole assignment.*
- Test cases for evaluation are in the same format as the ones in the assignment but have different contents. Students gain scores for each task if and only if the program passes all test cases of that task.
- Compile errors cause **0.0 points for the whole assignment.**
- **All submissions are carefully checked for plagiarism. Illegal actions including but not limited to copying source code on the internet, copying from other students, allowing other students to copy yours, etc. cause 0.0 points for Progress II and being banned for the final examination.**
- **Students are expected to maintain the confidentiality of their source code. If the source code of a student becomes accessible to others, that student will receive 0.0 points for the assignment.**
- **If students use chat GPT in this big assignment, students will be addressed similarly to plagiarism behavior.**
- If the submission is suspected of plagiarism, additional interviews may be conducted to verify the confidentiality of students' work.
- **Deadline: 23h59, Dec 10th, 2023.**
- **Students submit correctly to the Assignment section on the ELIT system.**

-- THE END --