A non-classical approach to Recommender System for Competitive Programmers

A dissertation submitted to Department of Computer Science and Engineering, BRAC University in partial fulfillment of the requirements for the degree of Undergraduate Program.

Submitted By

Tahanima Chowdhury, 13101108

MD. Maqsud Ul Anwar, 13101033

Ahmed Rafiq Ullah, 12241012

Supervised By

Mr. Moin Mostakim

Department of Computer Science and Engineering

BRAC University

Bangladesh

August, 2017

DECLARATION

We do hereby declare that the thesis titled “A non-classical approach to Recommender System for Sport Programmers” is submitted to the Department of Computer Science and Engineering of BRAC University in partial fulfillment of the completion of Bachelors of Science in Computer Science and Engineering. We hereby declare that this thesis is based on results obtained from our own work. Due acknowledgement has been made in the text to all other materials used. This thesis, neither in whole nor in part has been previously submitted to any University or Institute for the award of any degree or diploma. The materials of work found by other researchers and sources are properly acknowledged and mentioned by reference.

Dated: 14th August, 2017

……
Tahanima Chowdhury (13101108)

……..
(Supervisor)

Mr. Moin Mostakim
Lecturer

……
MD. Maqsud Ul Anwar (13101033)

……
Ahmed Rafiq Ullah (12241012)

ACKNOWLEDGEMENT

First and foremost, we express our solemn to almighty ALLAH, the merciful, who has given us the ability, strength and opportunity to perform this thesis work.

We express our heart full indebtedness and owe a deep sense of gratitude to our honorable and respectable teacher and our supervisor **Mr. Moin Mostakim**, Department of Computer Science and Engineering, BRAC University for his systematic work procedure, scholastic guidance, unwavering support, constructive suggestions, constant encouragement and generosity of time and knowledge throughout the thesis work and in the preparation and successful completion of the manuscript.

We would also like to express our gratitude towards our parents, siblings and close friends for their support, patience, enthusiasm, encouragement and kind co-operation throughout the thesis work.

TABLE OF CONTENT

TABLE OF CONTENT (CONTINUED)

ABSTRACT

To practice solving programming problems from online automated judge is very essential for competitive programmers since this helps develop critical thinking required for solving programming challenges in various programming contests. Competitive programmers use a lot of their time searching for programming challenges in automated online judges whose level of difficulty are within their reach to solve. This is very time consuming and at times the programmers miss out some of the available programming problems. This issue has not been addressed in any previous works so we have come up with a non-classical approach to recommending challenges to competitive programmers.

CHAPTER 1 - INTRODUCTION

1.1 Overview of Programming Contests

Programming contests are mind sports usually arranged over the internet or local network. During the contest, participants write computer programs according the specifications provided in the problem statements. Although there are multiple types of programming contests such as the ones where contestants work on some projects, we are focusing on short-term programming contests consisting of clearly defined tasks. Thus, our work is motivated by programming contests such as: the ACM International Collegiate Programming Contest (ICPC), the International Olympiad in Informatics (IOI), company-branded contests such as the Google CodeJam and the Facebook Hacker Cup, large portals that host regular contests, such as CodeForces, CodeChef and the Algorithm track at TopCoder and so on.

1.2 Importance of Programming Contests

Over the years the popularity of programming contests has increased to a great extent. In fact, Google, one of the most popular technology companies hosts a programming contest each year named Google Code Jam to identify top engineering talent for potential employment at Google. Even Facebook, one of the most popular social network companies hosts a programming contest each year named Facebook Hacker Cup to identify top engineering talent for potential employment at Facebook. In fact sites, such as TopCoder, HackerRank, HackerEarth, Codechef and so on provide employment opportunities through hosting programming contests.

1.3 Introduction to automated online judge

One of the prerequisites to perform well in programming contests is to practice solving programming problems in online automated judge. An online automated judge is an online system to test programs. The system can compile and execute code, and test them with pre-constructed data. Submitted code may be run with restrictions, including time limit, memory limit and so on. The output of the code will

be captured by the system, and compared with the standard output. The system will then return the result. Examples of online automated judges include: UVa Online Judge, TopCoder, Codeforces, CodeChef, SPOJ, HackerRank, HackerEarth, Light OJ, Kattis, URI Online Judge and so on.

1.4 Introduction to Recommender Systems

Recommender systems are systems that help users discover items they will be interested in. They are a key part of almost every modern consumer websites. Recommending useful items not only helps the user to save time but also makes the user aware of items which would otherwise be skipped because of the large quantity of items present in the system.

1.5 Problem Description

There are currently thousands and thousands of programming problems available on various automated online judges over the internet. The programming problems are categorized by various topics such as Ad Hoc, Mathematics, Data Structures, Dynamic Programming, Graph, Geometry and so on. The programming problems are yet again can be further categorized based on varying difficulty levels. For example, some problems from same topic can be of varying difficulty levels. For a competitive programmer to be able to reach a position where he/she has enough critical thinking and expertise to solve challenging programming problems takes years of problem solving and learning new algorithms. Therefore, solving programming problems from various topics and slowly progressing to more difficult and challenging problems is a difficult task to do for a competitive programmer as programming problems in various automated online judges are not well organized at times. So in order to address this problem of searching the right problems to solve next for a competitive, we have proposed a recommendation system for programming problems found in various automated online judges.

CHAPTER 2 - RELATED WORKS

In today's world there is huge quantities of data available. Guiding people towards content of their need or content which would be useful and meaningful to them is a big challenge. In order to address this challenge, several information filtering or recommendation tactics have been applied to data. Two of the most popular and widely used methods of information filtering or recommendation are content-based recommender system and collaborative filtering recommender system.

In a content-based recommender system, the items to be recommended are defined by their associated features. For instance, text recommendation systems like the newsgroup filtering system NewsWeeder (Lang 1995) uses the words of their texts as features. A content-based recommender learns a profile of the user's interests based on the features present in items the user has rated. The recommender system then recommends items similar to previous items rated highly by the user.

A collaborative filtering recommender system uses similarities between users and items to recommend items to users. It builds a set of users whose ratings are similar to the user in question. Afterwards, the recommender estimates the ratings of items (not yet considered) of the user in question based on the ratings of the users in the set.

Tapestry **[5]** is one of the earliest implementations of collaborative filtering-based recommender systems. It is a manual system where users help each other out through recording their reactions about a document which is referred as annotation. Other users of the system can use these annotations for filtering relevant documents. Afterwards, recommender systems which based their prediction by the analysis of ratings were developed. The GroupLens research system **[6, 7]** provides a collaborative filtering recommendation for Usenet news and movies. Ringo **[9]** and Video Recommender **[8]** are email and web-based systems that generate recommendations on music and movies respectively.

Afterwards, other technologies such as Bayesian networks, clustering, and Horting have also been incorporated to recommender systems. A Bayesian network model consists of nodes and corresponding edges. The nodes hold decision trees. The edges represent user information. The model is created based on the training data set. This sort of model is very small, very fast, and essentially as accurate as nearest neighbor methods **[10]**. Bayesian networks perform well in situation where the knowledge of user preferences changes slowly with respect to the time needed to build the model. However, they are not suitable for environments in which user preference models must be updated frequently.

Clustering techniques group users based on preferences that appear to be common or similar. Afterwards when the clusters are created, predictions for an individual can be made by averaging the opinions of the other users in that cluster. Some clustering techniques take into account the partial participation of users in several clusters for predictions. In that case, prediction is generated by taking the average across all the clusters the user is a part of, weighted by degree of participation. Clustering techniques usually generate less personal recommendations than other methods, and in some cases, the clusters have worse accuracy than nearest neighbor algorithms **[10]**. After the clustering is complete, however, performance can be very good, since the size of the group that must be analyzed is much smaller. Clustering techniques can also be applied as an initial step for reducing the candidate set in a nearest neighbor algorithm or for distributing nearest-neighbor computation across several recommender engines. And in fact, this might increase throughput.

Horting is a graph-based technique. Here, the nodes are the users, and the edges between nodes represent the degree of similarity between two users **[11]**. Predictions are generated by navigating the graph using the adjacent nodes at each step and then combining the opinions of the adjacent users. Horting is different from nearest neighbor. This is because in horting the graph may be navigated through other users who have not rated the item in question, thus exploring transitive relationships that nearest neighbor algorithms do not consider. In one study using

synthetic data, Horting produced better predictions than a nearest neighbor algorithm **[11]**.

Schafer et al., **[12]** provides in depth classification and examples of various recommender systems that are being used in E-commerce systems and how these systems give one-to-one personalized recommendations. Although these systems have been successful recommenders for quite some time now, their widespread use have revealed some of their drawbacks such as the problems of sparse data set **[17, 18]**, problems that arise due to high dimensionality **[19]** and so on.

As time passed, recommender systems have developed over time and gave rise to hybrid recommender systems that combine multiple algorithms into complex systems that takes advantage of the strengths of the individual algorithms.

CHAPTER 3 - SOLUTION USING NON-PERSONALIZED APPROACH

3.1 Description

Non-personalized recommender systems are the simplest type of recommender systems. These recommender systems do not consider the unique or specific needs or preferences of the users. The recommendations provided by such systems are same for every other users of the system. In context of an e-commerce site these recommender systems would recommend the top-N products bought by users. As for a movie site, these recommender systems would recommend top-N highly rated movies. Two of the most used algorithms in non-personalized recommender systems are: aggregated opinion approach and product association approach.

3.2 Aggregated Opinion Approach [14]

Aggregated Opinion Approach tries to capture how good of a particular item is as expressed by the average of the users. The opinions are displayed in the form of: average rating of an item, net upvotes of a particular item, net number of likes given on a particular item, percentage of users who have rated an item greater than or equal to 4 stars on a scale of 0 to 5 stars and so on. For example, there are a lot of restaurant websites which implement non-personalized aggregated opinion approach to suggest restaurants to the users. They provide a score to each of the restaurants. This score is the average ratings given to that particular restaurant by the users. The score generally ranges from 0 to 5 since users are allowed to rate the restaurants on a scale of 0 to 5. The score **[20]** of a restaurant is computed using the formula:

$$\boldsymbol{Score} = \boldsymbol{ROUND}(\boldsymbol{MEAN}\ (\boldsymbol{ratings}) \times \boldsymbol{N})\ \ or$$

$$\boldsymbol{Score} = \boldsymbol{MEAN}(\boldsymbol{ratings}) \times \boldsymbol{N},$$

$$where\ N\ denotes\ the\ scale\ ratings\ are\ displayed\ in$$

Travel websites as well as movie websites use aggregated opinion approach. Travel websites provide average ratings and reviews of different places around the world. Movie websites provide a chart on top-N movies having the highest average rating.

3.3 Product Association Approach [15, 16]

A lot of online shopping websites these days use product association approach to recommend items to users through the feature "people who bought *item1* also bought *item2*". This recommendation is made based on the current content of the shopping cart of the user. The recommendations may not be specifically tailored for individual users but are specifically based on what the user is currently doing, i.e viewing or buying. These systems base their recommendations on the idea that "People who did X also did Y". A simple way to express this is percentage of X-buyers who also bought Y. This can be shown in the formula:

$$\frac{(X\ AND\ Y)}{X} \qquad (3.3.1)$$

Although intuitively correct, this formula does not compensate for the overall popularity of Y. This formula can be adjusted by looking at whether X makes Y more likely than not X (!X).

$$\frac{\frac{(X\ AND\ Y)}{X}}{\frac{(!X\ AND\ Y)}{!X}} \qquad (3.3.2)$$

This formula focuses on increase in Y associated with X **[20]**. Another solution to the drawback of formula (3.3.2) is using the Association rule mining which uses the lift metric. It is basically non-directional association **[21, 16]**.

$$\frac{P(X\ AND\ Y)}{(P(X)\times P(Y))} \qquad (3.3.3)$$

More generally association rules look at basket of products, not just individuals.

3.4 Analysis and Pitfalls

Non-personalized recommender systems would be a very good solution to cold start problems, i.e recommending challenges to new users of the system. Our recommender system deals with recommending challenges from an online automated judge to a user. When a new user accesses the system if he/she is recommended top-N highly accepted or solved challenges, it is most likely that those N challenges are the easiest ones. For a newbie or a beginner easiest challenges are most suitable since we are not yet aware of the problem solving level of that user. However, for an existing user who has already solved quite a significant number of challenges this system might not be a very good one to continue making growth as a problem solver. The user would need more personalized recommendation based on the challenges he/she has solved, the kind of problem category the user has been solving challenges from, the level the user is in as a problem solver and so on.

CHAPTER 4 - SOLUTION USING CONTENT-BASED RECOMMENDER SYSTEM

4.1 Description

Content-Based recommender systems assume that the users' preferences do not change significantly over time which is why these systems recommend items to target users similar to previous items rated highly or labeled as relevant by the target users. For example, in case of movies, these systems would recommend movies with same actors, directors and so on. As for websites, blogs, accessing scientific papers and news these systems would recommend articles with similar content.

Plan of Action:

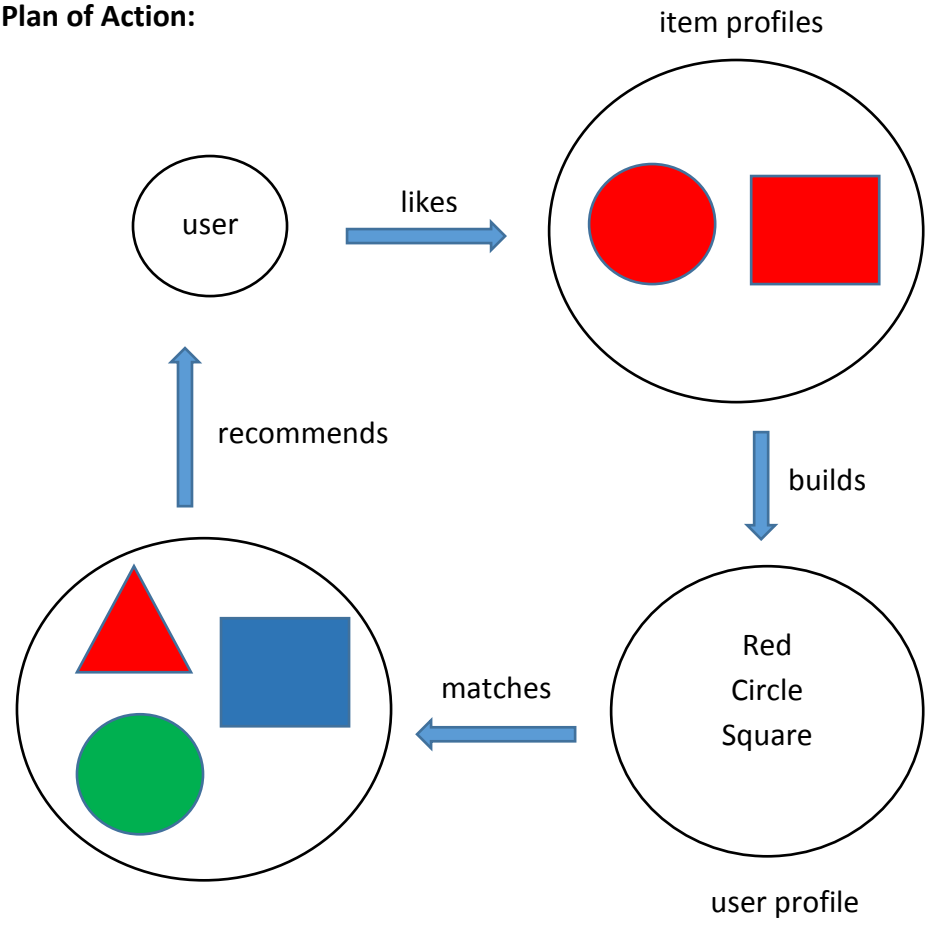


Figure: 4.1.1

4.2 Building Item Profiles

Initially, for each item an item profile needs to be built up. A profile is basically a set of features associated with the item. A content-based recommender systems basically rely on the analysis and exploitation of textual content to build up the item profiles. The set of features that are extracted for building up the item profiles strictly depends on the domain. For example, in a book recommendation scenario the item profile of each items will contain the author's name, genre of the book, plot of the book and so on. In a movie recommender system, the item profiles will consist of the name of actors, name of directors, genre and so on. The value of the features associated with each item can be directly extracted from structured data (for example, the genre of the book or director of a movie) or extrapolated by analyzing unstructured data (for a news article, it is common to exploit only the most significant words in it). One of the most popular techniques to extract relevant keywords from a document is a classical weighting scheme, TF-IDF (term frequency-inverse document frequency) **[22]**. This scheme is built up on the idea that the weight of certain features within a textual content (documents) is related to two factors: its frequency and its commonality. The TF-IDF weight for a keyword *i* in document *j* is computed as the product of these two measures:

$$TF - IDF(i,j) = TF(i,j) \times IDF(i) \qquad (4.2.1)$$

The TF (term frequency) calculates how many times the term (feature) *i* occurs in the document *j*. Since TF is strongly dependent on document length, it is common to use some normalization scheme along with TF. In **[26]** the TF of a term is computed with respect to the maximum frequency of any term within the document.

Formally:

$$TF(i,j) = \frac{freq(i,j)}{maxOthers(i,j)} \qquad (4.2.2)$$

where $freq(i,j)$ is the number of occurrences of term i in document j and $maxOthers(i,j)$ is the maximum frequency among all the terms occurring in document j. In a very similar way, the IDF (inverse document frequency) technique provides a scheme to reduce the weight of very common keywords or terms. The idea behind IDF is that frequent words are not very helpful to distinguish among documents. So a word or term which appears in a fewer number of documents should be given more priority and therefore, should be given more weight. Let N be the number of all the documents and $n(i)$ be the number of documents in which keyword i appears. The inverse document frequency for i is typically calculated as follows:

$$IDF(i) = log\frac{N}{n(i)} \qquad (4.2.3)$$

So, using the TF-IDF technique an item profile would be built using the set of words with highest TF-IDF scores along with their scores. It is convenient to think of an item profile as a vector.

4.3 Building User Profiles

A key part of a content-based recommender system is to build up the user profile. This profile consists of the items the user has interacted with and the associated features of those items. The content-based preferences of user u can be represented as a set named $ContentBasedProfile(u)$, defined as follows:

$$ContentBasedProfile(u) = \{w_{u,f_1}, w_{u,f_2} \dots w_{u,f_n}\} \qquad (4.3.1)$$

Generally, data are obtained from the user in one of the two ways: explicit and implicit. Explicit ways to collect data from users are by asking them to rate or evaluate an item. This is referred as an explicit feedback. There are two main approaches to get explicit feedback: in the simplest case, the user can classify the items as relevant or not relevant by adopting a simple binary rating scale, such as in **[23]**. Alternatively, a discrete numeric scale is usually adopted to judge items, such as in **[24]**. The other way to gather data is referred as implicit feedback. Implicit feedback does not require any active user feedback. Activities of users are

monitored and analyzed to derive relevant information about the users. Approaches for gathering implicit feedback range from simple methods based on relevance scores assigned to specific user actions, such as saving an item, printing it and so on, to more complex ones involving the analysis of free text.

So, basically a content-based recommender system builds up the user profile with a set of features along with their corresponding weights. The weights associated with each features might be a boolean value capturing the fact that the user liked or did not like that feature, navigated or did not navigate that feature and so on. For example, in a movie recommendation scenario each feature may be a movie genre and the value can be set to 1 or 0 meaning whether the user is interested in that genre or not. The weights associated with each features can even be integer or real-valued. For example, in a restaurant recommendation scenario the ratings provided by the user could be in the range 0 - 5 stars. It is convenient to think of a user profile as a vector.

4.4 Making predictions

Recommender systems are built to provide recommendations on items the users will be mostly interested in. In content-based recommender system once the item and user profiles are built the system would like to recommend the items which the users have not navigated yet and the system thinks that the user would be interested in those items. In order to come to a conclusion for recommendation, content based recommender systems need a scoring function. The score of item i for user u, described as $f(u,i)$ denotes how relevant the item i is for the user u by matching item description with user profile.

$$f(u,i) = score\big(ContentBasedProfile(u), Content(i)\big) \qquad (4.4.1)$$

There are various approaches to modeling the score function. One very common approach is the weighted average method. Another approach is the cosine similarity. Cosine similarity is one of the most widely used technique to compute similarity

between two vectors. Given two vectors $d = \{d_1 \dots d_n\}$ and $u = \{u_1 \dots u_n\}$, their cosine similarity $cosSim(d, u)$ is calculated as follows:

$$cosSim(d,u) = \frac{\sum_{i=1}^{n} d_i * u_i}{\sqrt{\sum_{i=1}^{n}(d_i)^2} * \sqrt{\sum_{i=1}^{n}(u)^2}} \qquad (4.4.2)$$

The formula allows to sort or rank the items in a descending order of their relevance values, i.e, according to their similarity scores. Afterwards, the top n items in this ranking (or those overcoming a certain similarity threshold) can be considered as relevant and recommended to the target user. A common alternative for similarity-based models is to implement a content-based recommender system as a kNN classifier. By analyzing generic similarity measures the algorithm looks for the k most similar items among those represented in the vector space and recommends the items whose nearest neighbors (or at least most of them) were already labeled as interesting or relevant by the target user.

4.5 Analysis and Pitfalls

Content based recommender approaches will be of great use for devising part of our recommender model. To recommend challenges to the user, initially, profile of the challenges and profile of the user need to be built. A challenge profile would include category of the challenge, total accepted solutions of that challenge, total number of users attempted that challenge and other statistical information. A user profile would include the type of category challenges the user has been solving challenges from, the level of the challenges the user has been solving problems of and so on. Of course, the user has to have significant number of submissions in order for a user profile to be built for him/her. So, content-based recommender will have difficulties recommending challenges for new users which needs to be tackled using other methods.

CHAPTER 5 - SOLUTION USING COLLABORATIVE FILTERING RECOMMENDER

5.1 Description

A collaborative filtering recommender system recommends items to target users based on the ratings or behavior of other users whose ratings or behaviors are similar to that of the target users. The motivation behind this recommender system is that people usually get the best recommendation from people who have similar tastes. A typical collaborative filtering recommender system will have the following workflow:

1) A user rates items such as books, movies, restaurants, hotels and so on. The system captures the ratings of the user. This rating is considered as an approximate representation of the user's interest in the corresponding domain.
2) The system matches this user's ratings against other users' and finds the people with most "similar" tastes.
3) The system then recommends items that the similar users have rated highly but have not yet been rated by this user.

Plan of Action:

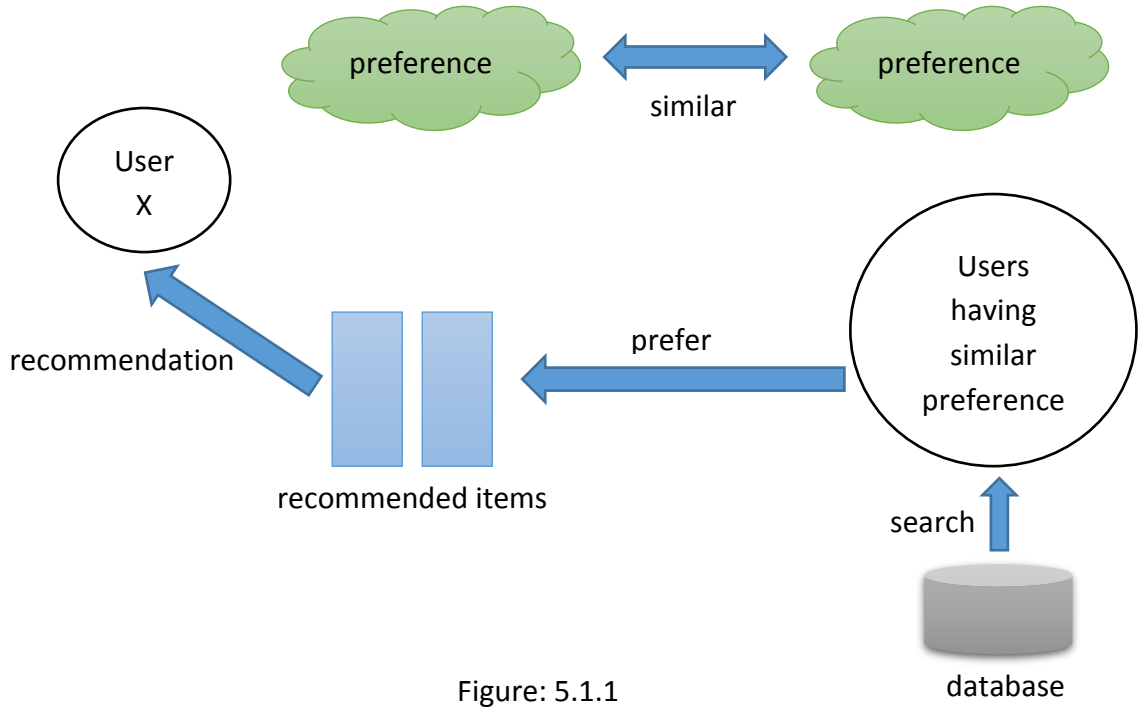


Figure: 5.1.1

5.2 Making Recommendations

One common approach to make recommendations is to analyze the whole ratings space of the system in order to extract the most similar users of the target user and to predict the preferences of the target user according to those of his/her own neighborhood. The simplest heuristic-based technique is the so-called user-based collaborative filtering. We can define $CollaborativeUserProfile(u)$, user profile for user u, as a row vector of the user/item matrix. Formally,

$$CollaborativeUserProfile(u) = \{r_{u,1}, r_{u,2} \dots r_{u,n}\} \qquad (5.2.1)$$

where $r_{u,i}$ is the rating provided by user u on item i. After the profile of a user is built it is required to use some similarity measures to identify users whose preferences are similar to those expressed by the target user u. Then, for every item i that the target user did not navigate yet, the prediction is computed according to the ratings provided on i by the neighbors. Formally, the predicted rating of item i for user u described as $f(u,i)$ can be computed as a score function that takes as input the $CollaborativeUserProfile(u)$ and $Neighborhood(u)$ (that is to say, a set of row vectors extracted from the user/item matrix) and returns a numerical prediction of user interest in item i.

$$f(u,i) = score\big(CollaborativeProfile(u), Neighborhood(u)\big) \qquad (5.2.2)$$

In algorithms for user-based collaborative filtering particular emphasis is put on the formulas for calculating similarity. Beside cosine similarity, already introduced in Formula 4.4.2, it is typical to extract the neighborhood by exploiting the Pearson's coefficient, calculated as follows:

$$sim\big(u,u^{/}\big) = \frac{\sum_{i=1}^{n}\big(r_{u,i} - \bar{r}_u\big) * (r_{u^{/},i} - \overline{r^{/}_u})}{\sqrt{\sum_{i=1}^{n}(r_{u,i} - \bar{r}_u)^2} * \sqrt{\sum_{i=1}^{n}(r_{u^{/},i} - \overline{r^{/}_u})^2}} \qquad (5.2.3)$$

where u and $u^{/}$ is a couple of users.

Once the similarities between the users are determined, it is possible to predict user rating on item i by aggregating neighborhood's ratings on the same item. In

literature many aggregation functions have been exploited: Adomavicius and Tuhzilin **[25]**, for example, presented three different alternatives. In the first case (Formula 5.2.4) the prediction is calculated as the average rating provided by the users in the neighborhood. In the second (Formula 5.2.5) each rating is weighted with the similarity with the neighbors (in order to weigh more the opinion of the most similar users), while in the third one (Formula 5.2.6) the rating is further adjusted by taking into account the deviation from each neighbor's average rating. Formally,

$$f(u,i) = \frac{1}{N} \sum_{u^{\prime} \epsilon N} r_{u^{\prime},i} \qquad (5.2.4)$$

$$f(u,i) = k \frac{1}{N} \sum_{u^{\prime} \epsilon N} r_{u^{\prime},i} \times sim(u,u^{\prime}) \qquad (5.2.5)$$

$$f(u,i) = \bar{r_u} + k \frac{1}{N} \sum_{u^{\prime} \epsilon N} r_{u^{\prime},i} \times sim(u,u^{\prime}) \times (r_{u^{\prime},i} - \bar{r_u}) \qquad (5.2.6)$$

where $r_{u,i}$ is the rating provided by user u on item i, $sim(u,u^{\prime})$ is the similarity between user u and user $u^{\prime}$, $\bar{r_u}$ is the average rating of user u, and k is a normalization factor.

5.3 Analysis and Pitfalls

Collaborative filtering is a good approach to suggest programming problems in an automated online judge. Majority of the users using automated online judges solve programming problems with basically the same intention, i.e, performing well in programming contests. So, for similar users who have similar submission history the system can make use of the fact that they have similar growth rate, preference towards choosing categories and problems and so on. However, this system might not perform well for users who have solved fewer problems. This is because every individual is different. Some of the users might solve problems with steady increase in difficulty levels while some might solve problems of varying difficulty haphazardly. It might happen that based on a user's similar neighbors the system is suggesting

some programming problems of a particular category which the user does not have maturity and understanding to solve. In conclusion, the system should be able to recommend problems for the steady, gradual growth of the users and should not assume that the user has mastered certain topics without proper statistics.

CHAPTER 6 - PROPOSED SOLUTION

6.1 Goals

As we have already reviewed various existing Recommender Systems that are currently being used in many scenarios, we have come to understanding the limitation of the existing systems in recommending programming problems and also a clear understanding of what an ideal Recommender System should be able to do in our problem definition. We have come to several criteria that our recommender system should do:

1) Recommend problems that has not been solved by the user till now.
2) Recommend problems that are most likely to be solvable by the user, i.e, the recommended problems should be in a certain difficulty range based on user's level.
3) Recommend problems in such a way that it ensures steady growth of the user as a problem solver.
4) Recommend problems in a way that ensures all round growth in most topics of the user rather than partial growth in certain topics.

6.2 Structure of an automated online judge

Currently, there are many active automated online judges which provide a wide range of programming problems. Among those automated on-line judges there are some which do not contain significant number of programming problems in terms of quality and difficulty level. A serious competitive programmer usually solves problems from a handful of automated on-line judges that contains wide range of quality problems. Some of the most popular on-line judges include UVa Online Judge, Codeforces, Sphere Online Judge, CodeChef, etc.

1) UVa Online Judge

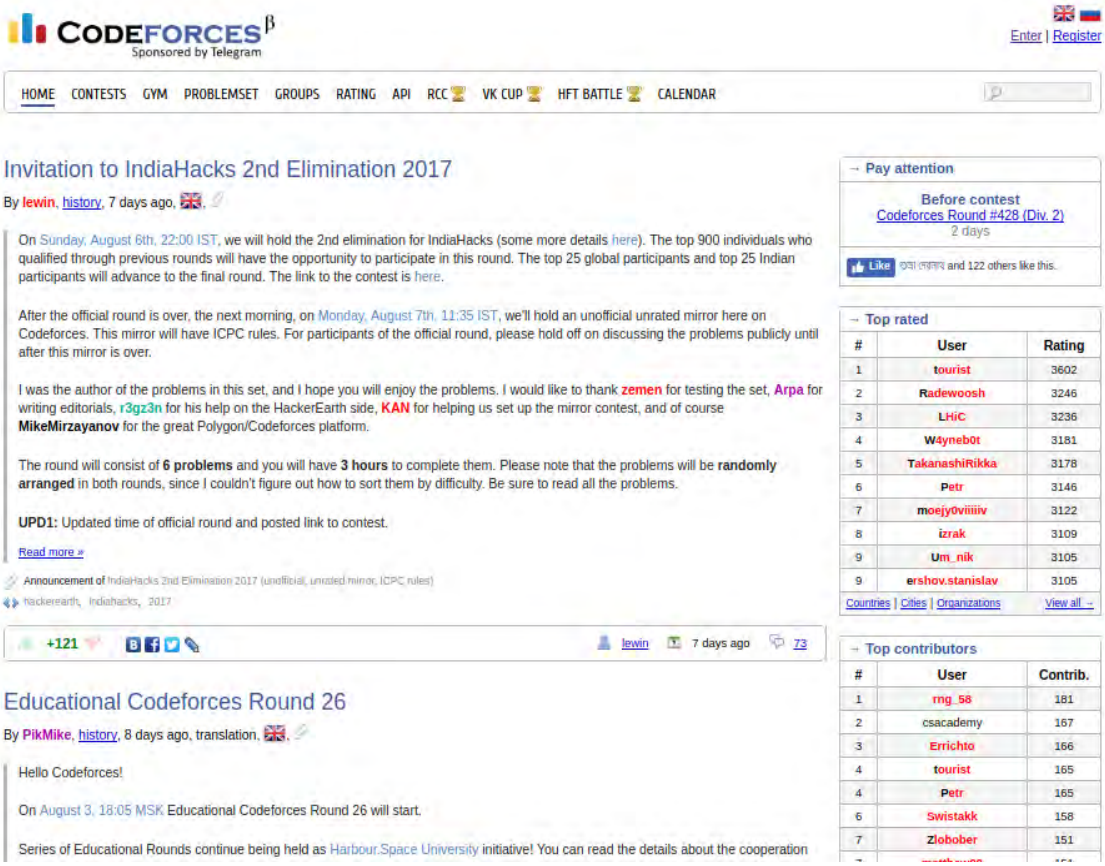
UVa Online Judge is an automated online judge for programming problems hosted by University of Valladolid. Its problem archive has over 4300 problems which consists of programming problems from Programming Challenges by Skiena and

Revilla, ACM-ICPC World Finals, Western and Southwestern European Regionals and so on. User registration is open to everyone and there are currently over 100000 registered users.

2) Codeforces

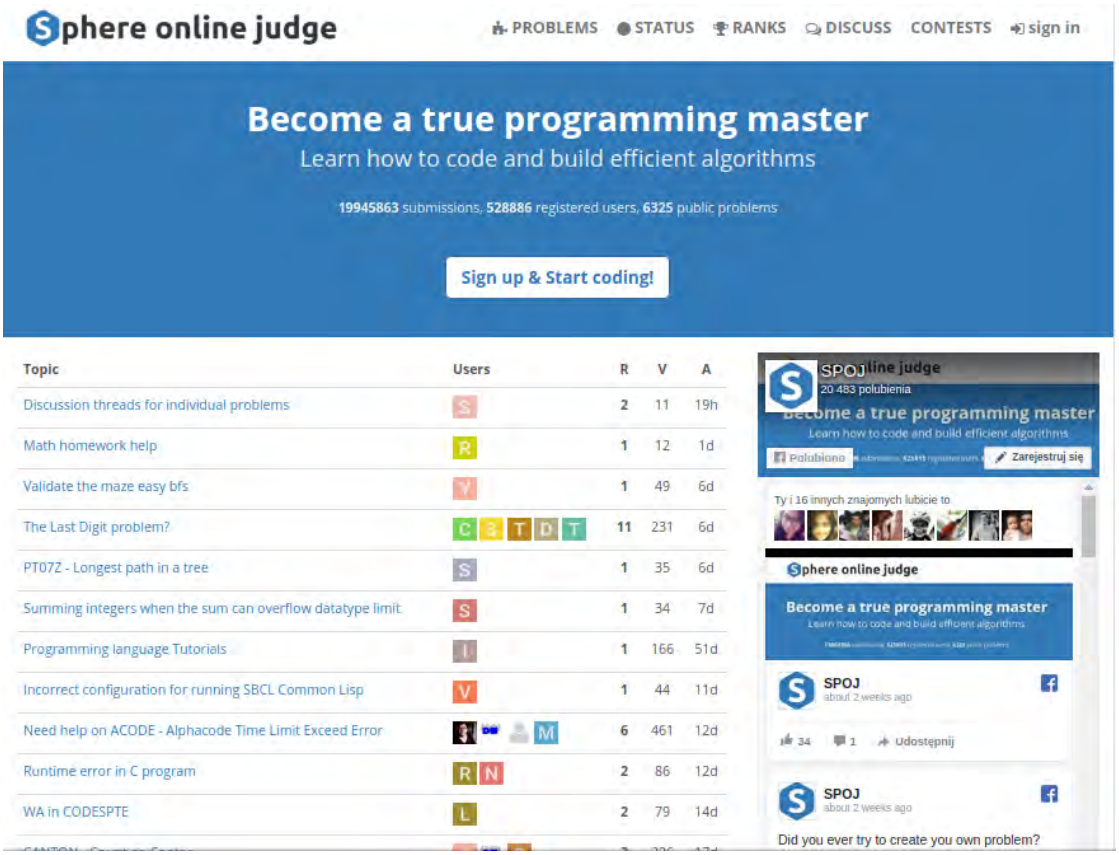
Codeforces is a Russian automated online judge dedicated to competitive programming. It was created and maintained by a group of competitive programmers from Saratov State University led by Mikhail Mirzayanov. Codeforces provides services such as: participation in the short (2-hours) contests, so-called "Codeforces Rounds", held about once a week, participation in educational contests

(2-2.5 hours), held 2-3 times per month, challenge/hack other contestants solutions, social-networking through the use of internal public blogs and so on.



3) Sphere Online Judge

Sphere Online Judge, SPOJ in short is an automated online judge with over 315,000 registered users. Its problem archive consists of over 20,000 problems. The problems in the problem archive are prepared by its community of problem setters or are taken from previous programming contests. SPOJ allows advanced users to organize contests under their own rules and also includes a forum where programmers can discuss how to solve a particular problem.



For the purpose of our proposed solution we would like to use a toolkit for UVa Online Judge named uHunt. This toolkit basically provides problem categories such as: Introduction, Data Structures and Libraries, Problem Solving Paradigms, Graph, Mathematics, String Processing, (Computational) Geometry and so on. It also provides difficulty level of each problems in a problem category and statistics of problems. Besides, it also provides submission data for a user who has a UVa account. Moreover, it exposes a web API that gives access to user and problem statistics.

6.3 Solution

Pre-processing Recommendation phase:

Before processing a recommendation from a user statistics for a user, we have to build a structure that holds information for problems in each categories such as the problem's category, the number of accepted submissions, the number of total submissions and the difficulty level from the uHunt. As for accepted number of submissions and total submissions we were able to acquire them using API provided

in the uHunt website. However, we still have to put problems into various category and gather their difficulty level manually from the website. Using these information we have to build a problem profile.

Some examples of problem profiles are:

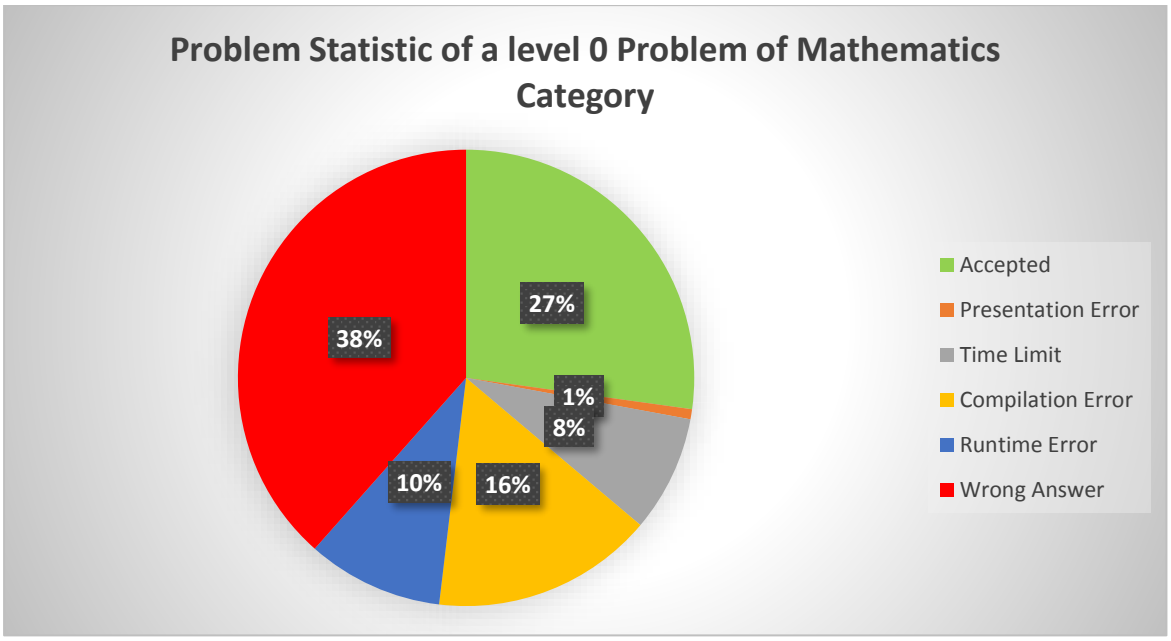


Figure: 6.3.1

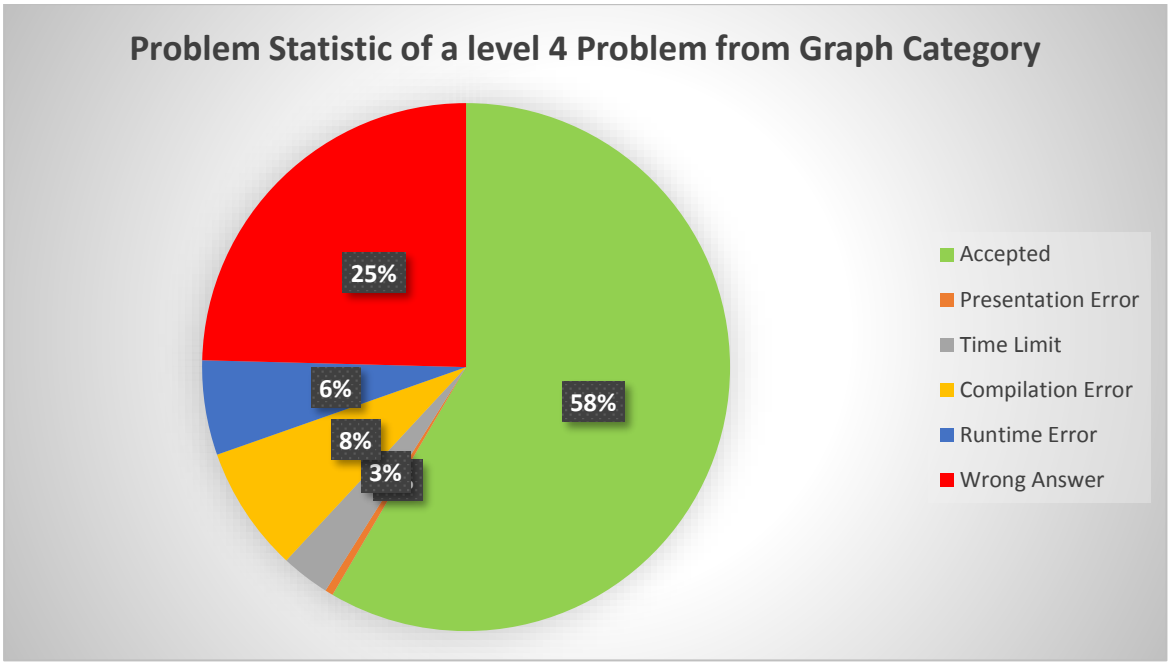


Figure: 6.3.2

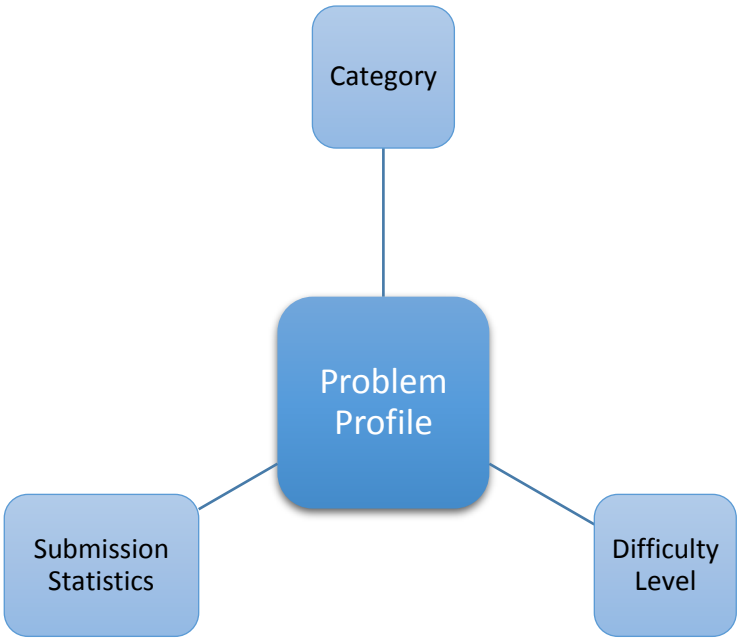


Figure: 6.3.3

There are basically two types of users in an online automated judge: new and existing.

New Users:

As for new users we do not have any statistics or submission history to analyze or evaluate the user's level as a problem solver. So, in this case we'll use a non-personalized approach. We basically will pick *k* problems from each problem category. Those *k* problems will be of the lowest level (easiest problems) in those categories. In uHunt, difficulty level ranges from 0 to 6, with 6 being the most challenging level. In certain categories problem with difficulty level 0 may not exist. In that case the next lowest level is considered. The *k* problems suggested from each category will be randomly picked. One may question why not suggest only easy Ad-hoc/Introduction problems rather than picking certain number of problems from each category? As our goal was to ensure all round growth for a problem solver so we exposed the new user to problems from other categories as well, but those problems are themselves the easiest in their own respective categories.

Pseudocode:

```
for each category ∈ problem category
    count = 0
    for each level starting from lowest level ∈ problem category
        count = count + 1
        suggest this problem
        if count == k
            break
```

Existing Users:

For existing user initially we will fetch user statistics from uHunt toolkit using the user's UVa user name. The data fetched will contain a list of problems solved by the user in UVa Online Judge. Afterwards, we will build a user profile. The user profile will contain the problems solved by that user in each difficulty level for each category of problems. A sample user profile represented using a chart is given below:

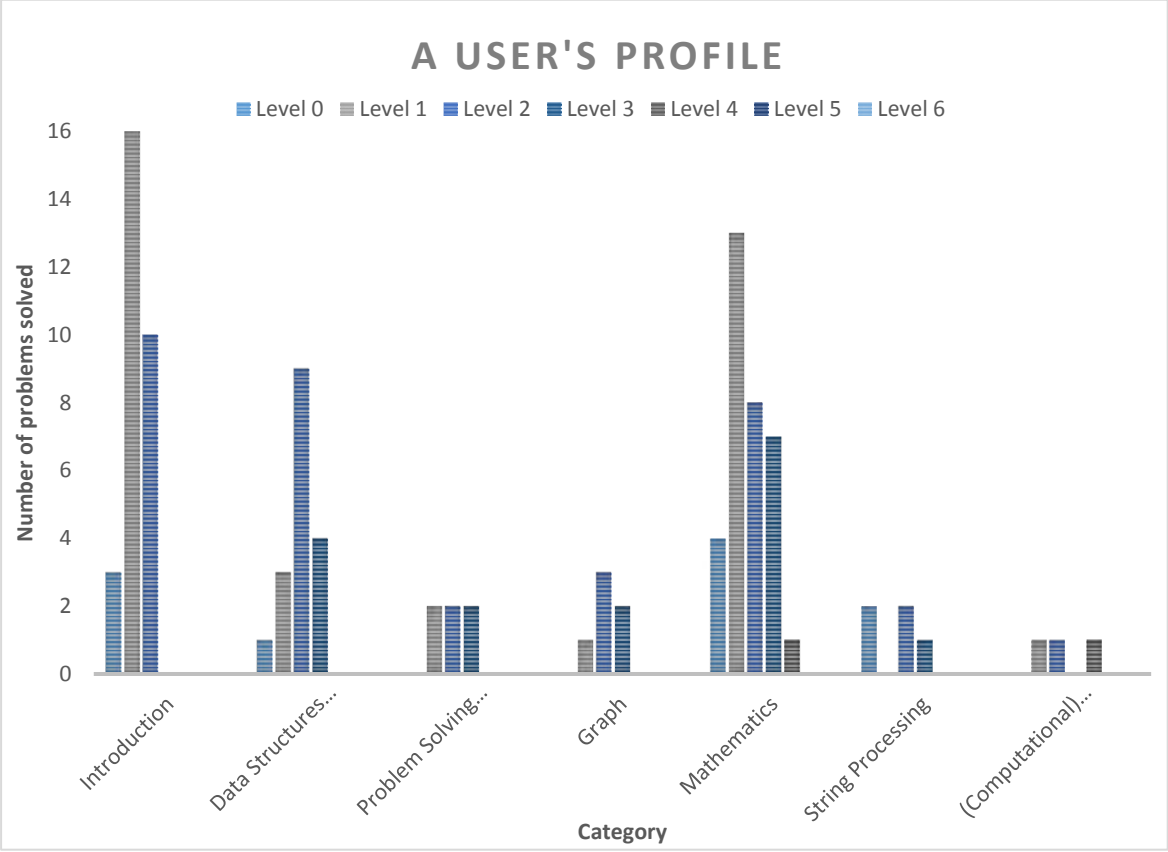


Figure: 6.3.4

Next we will iterate each category one by one and try to determine the user's level in each category separately. It might happen that the user has good strength in some categories while is weak in some other categories. The reason for this is often competitive programmers team up in a group of three when participating in many onsite programming contests which is why as an individual a competitive programmer does not get to read and solve all of the problems in the problem set. This over time builds biasedness towards some categories or topics for that competitive programmer. But to be ranked highly in programming contests all the members of a team have to be well adept in most topics. In onsite programming contests if one coder gets stuck on a problem solution due to some corner cases or any errors, having another team member with significant expertise on same topic will help a lot for identifying the error. So the aim is to ensure growth of the competitive programmers in all categories by providing the right problems in order to level up.

In order to recommend problems that matches user's level we will filter out all the problems solved by that user in a particular category. Next, we will filter the solved problems based on difficulty level. Below is a chart showing problems solved by a user named 'isat1729' in mathematics category which are divided into difficulty levels.

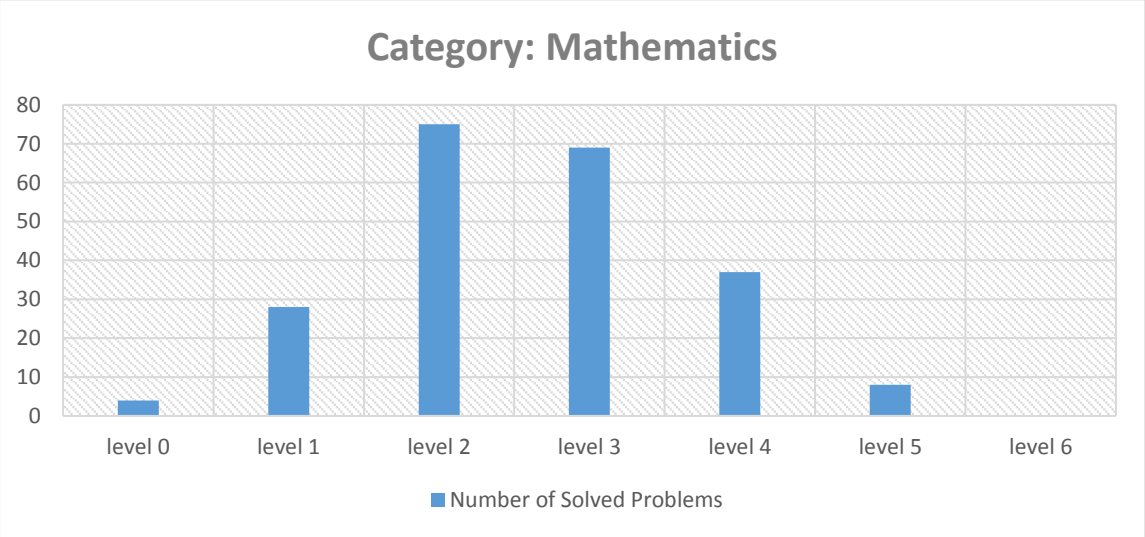


Figure: 6.3.5

Now the problem with this method is that it does not hold adequate information as regards the level the user is currently in. One may simply assume that the user is capable enough to solve level 5 problems so recommended problems should be of level 5 only. But that is not true. Often a competitive programmer can get lucky solving some higher level problems but is not capable enough to solve all other problems of that difficulty level, which basically means there is still a significant amount of programming problems needed to be solved in a lower level in order to gain maturity and consistency.

But now a new question arises; how many problems solved in a particular category ensures mastery of that level? The number of problems in each category in uHunt is not overwhelmingly large. Also the number of problems in each difficulty level are not equal hence, number of problems is not a good way to decide mastery of a level. Thus, we came up with the idea of considering the ratio of problems solved to total number of problems in that particular level in a particular category.

Now we get a data of such format:

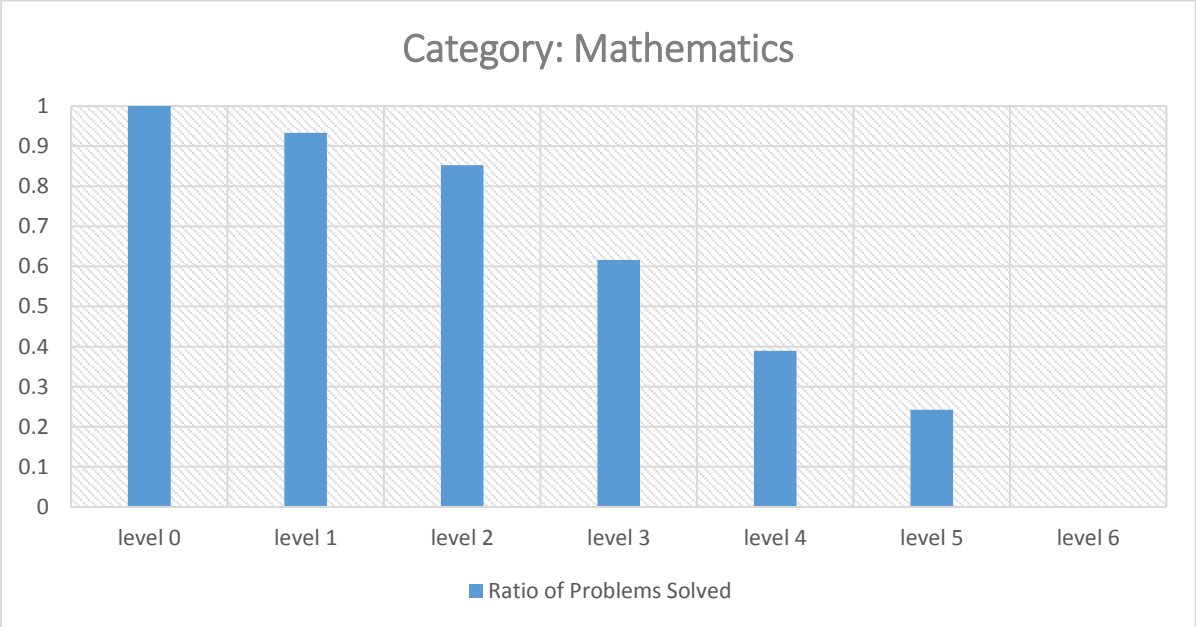


Figure: 6.3.6

From the list of given ratios of all the levels we choose the highest level that has exceeded a certain ratio. If the ratio exceeds 0.7 then we do not recommend

problem from that level but recommend problem from the next level else if ratio exceeds 0.4 but is less than 0.7 then we recommend problem from that level and from the next level. But if ratio is below 0.4 then only problems from current level gets recommended.

This way the recommender system will choose k problems from each category and make a recommendation.

Pseudocode:

For a given user, getting recommended user level for each category

```
Userlevel = get lowest level available in that category
recommendedLevelRatio = 0
Get list of solved problems
Map with key as category and value as recommended level
for each category ∈ problem category
    Make a list of number of solved problems per difficulty level
    Make a list of ratio of solved problems and total problems per difficulty level
    For each level from level ratio list
    If level ratio > recommendedLevelRatio
        Userlevel = level
        recommendedLevelRatio = minimum of (level ratio, 0.7)
        if level ratio > 0.7
            Userlevel = Userlevel  + 1
            recommendedLevelRatio = minimum of (level ratio +1, 0.7)
    add category and Userlevel +1 to Map
    if  recommendedLevelRatio >0.4 and recommendedLevelRatio < 0.7
        add category and Userlevel +1 to Map
return Map
```

For a given user, making recommendation

Get list of all problems

Get list of user solved problems

For each category

    Get userRecommendedLevel

    Filter a list of unsolved problems with given category and user recommended level

    Choose k random problems from filtered list and make recommendation

CHAPTER 7 – IMPLEMENTATION

7.1 Language

We chose Java as the programming language for implementing our proposed solution. The reason for choosing Java is its enriched library and also because it has good build tools; in our case we used Maven.

Java

The Java programming language platform provides a portable, interpreted, high-performance, simple, object-oriented programming language and supporting run-time environment. The development cycle in Java is much faster as Java technology is interpreted, which means it won't require machine code in order to execute program rather an interpreter will run through the a program and execute them which is why programs need to be only compiled and then run. **[3][4]**

Maven

Maven is a tool developed by Apache that is used for building and managing any Java-based project. It allows developer to understand the complete state of a development effort in minimal amount of time. Maven deals with:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features **[2]**

Application Programming Interface (API)

We used web API that were exposed in uHunt tool kit which was developed by Felix Halim for acquiring real-time UVa statistics. The API is free to use for anyone to build

their own statistical tools. The API allows AJAX requests that will return a JSON formatted result. **[1]**

To view specific problem statistics by problem number we used the following URL:

URL: http://uhunt.felix-halim.net/api/p/num/{problem-number} , where {problem-number} is replaced the problem number.

A sample result for a requested problem with problem number 100:


```
{
 "pid": 36,
 "num": 100,
 "title": "The 3n + 1 problem",
 "dacu": 82175,
 "mrun": 0,
 "mmem": 1000000000,
 "nover": 0,
 "sube": 6949,
 "noj": 0,
 "inq": 0,
 "ce": 115236,
 "rf": 0,
 "re": 70686,
 "ole": 326,
 "tle": 60423,
 "mle": 5209,
 "wa": 281744,
 "pe": 5218,
 "ac": 199194,
 "rtl": 3000,
 "status": 1,
 "rej": 0
}
```


To view all the submission of a particular user by user id we used the following URL:

URL: http://uhunt.felix-halim.net /api/subs-user/{user-id}, where {user-id} is id number of a user.

A sample result for requesting user id from given user name parishaullah:

881024

A sample result for user statistics from given user id 881024:

{"name":"Parisha Ullah","uname":"parishaullah","subs":[[18514649,1753,70,0,1482048834,5,-1],[18514770,435,90,0,1482050762,5,6554],[18560316,484,90,160,1483016314,5,4421],[18560317,627,90,0,1483016339,5,652],[18651901,841,90,0,1484903977,5,1981],[19353660,959,90,0,1494525240,5,2143],[19353542,1048,90,670,1494522976,5,6904],[18560293,1311,90,0,1483015942,5,1898],[19353669,1415,90,350,1494525426,5,3324],[18563230,1637,90,360,1483097864,5,8604],[18560297,1724,90,0,1483015993,5,5920],[18563231,1870,90,20,1483097901,5,3107],[19353569,1998,90,10,1494523684,5,395],[18560303,2307,90,0,1483016073,5,3704],[18577213,2412,90,100,1483510458,5,1526],[18560311,2542,90,0,1483016216,5,3781],[19353574,2635,90,0,1494523787,5,388],[18560299,2864,90,0,1483016032,5,2810],[19353523,3087,90,0,1494522767,5,1429],[18533261,3402,90,0,1482416635,5,1743],[18533266,3431,90,0,1482416716,5,1223],[18560310,3710,90,0,1483016160,5,2575],[18514551,3834,90,0,1482047793,5,2314],[18514443,4022,90,0,1482046105,5,2691],[19353571,4952,90,20,1494523727,5,54],[18992434,4952,90,40,1489824543,5,89]]}

In order to implement our solution we have three classes: *Main* class, *Problem* class and the *User* class. The *Main* class contain two methods that is the *main()* method and the *newUserSuggestions()* method. In *main()* method we firstly took a username as input from the console. Next we check whether it is a new user or not.

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    while (input.hasNext()) {
        String user = input.next();

        try {
            List<Problem> recProblems = User.getAccepted(user).isEmpty() ? newUserSuggestions() : User.getRecommendedProblems(user);

            System.out.println(user);
            for (Problem p : recProblems) {
                System.out.println(p);
            }
            System.out.println();
        } catch (IOException ex) {
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Figure: 7.1.1

If the username turned out to be a new user then a list of recommended problems gets generated in *recProblem()* by the *newUserSuggestions()* method.

```
private static List<Problem> newUserSuggestions() {
        ArrayList<Problem> problems = null;
        try {
                problems = Problem.getProblems();
        } catch (FileNotFoundException ex) {
                Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
                System.exit(-1);
        }

        return problems.stream().filter(problem -> problem.level == 0).collect(Collectors.toList());
}
```

Figure: 7.1.2

If username turns out to be an existing user than a list of recommended problems gets generated in *recProblem()* by the *getRecommendedProblems()* method from *User* class.

The *User* class consists of two global String variables that stores URL that will retrieve user ID from given username and user statistics for given user id from uHunt toolkit.

```
private static final String GET_UID_URL = "http://uhunt.felix-halim.net/api/uname2uid/%s";
private static final String GET_USER_SUBMISSIONS_URL = "http://uhunt.felix-halim.net/api/subs-user/%d";
```

Figure: 7.1.3

The *getUID()* method retrieves user ID from uHunt and stores them in global variables

```
public static int getUID(String username) throws IOException {
        URL url = null;
        try {
                url = new URL(String.format(GET_UID_URL, username));
        } catch (MalformedURLException ex) {
                Logger.getLogger(User.class.getName()).log(Level.SEVERE, null, ex);
        }

        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("GET");
        Scanner scanner = new Scanner(connection.getInputStream());
        return scanner.nextInt();
}
```

Figure: 7.1.4

The *getAccepted()* method returns a list of solved problems for a given username

```
public static ArrayList<Problem> getAccepted(String username) throws IOException {
        int uid = getUID(username);

        URL url = null;
        try {
                url = new URL(String.format(GET_USER_SUBMISSIONS_URL, uid));
        } catch (MalformedURLException ex) {
                Logger.getLogger(User.class.getName()).log(Level.SEVERE, null, ex);
        }

        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("GET");
        Scanner responseStream = new Scanner(connection.getInputStream());
        responseStream.useDelimiter("\\A");

        JSONObject response = new JSONObject(responseStream.next());
        JSONArray submissions = response.getJSONArray("subs");

        HashSet<Integer> acceptedPIDs = new HashSet<>();
        for (int i = 0; i < submissions.length(); i++) {
                JSONArray sub = submissions.getJSONArray(i);
                if (sub.getInt(2) == 90) { // verdict is accepted
                        acceptedPIDs.add(sub.getInt(1));
                }
        }

        ArrayList<Problem> problems = Problem.getProblems();
        ArrayList<Problem> acceptedProblems = problems.stream().filter(problem -> acceptedPIDs.contains(problem.problemID)).collect(Collectors.toCol

        return acceptedProblems;
}
```

Figure: 7.1.5

The *getAcceptedRatioPerLevelPerCategory()* returns a Map of key value category and value being another map which contains integer and double which represents difficulty level and ratio of solved problems in that difficulty level for a category

```
public static Map<String, Map<Integer, Double>> getAcceptedRatioPerLevelPerCategory(String username) throws IOException {
        Map<String, Map<Integer, Long>> totalProblems = Problem.getProblemStats(Problem.getProblems());
        Map<String, Map<Integer, Long>> userACProblems = Problem.getProblemStats(User.getAccepted(username));

        Map<String, Map<Integer, Double>> ACRatio = new HashMap<>();
        for (String category : totalProblems.keySet()) {
                Map<Integer, Long> totalcountPerLevel = totalProblems.get(category);
                Map<Integer, Long> userCountPerLevel = userACProblems.getOrDefault(category, Collections.EMPTY_MAP);

                Map<Integer, Double> levelGroup = new HashMap<>();
                for (Integer level : totalcountPerLevel.keySet()) {
                        levelGroup.put(level, userCountPerLevel.getOrDefault(level, Long.valueOf(0)).doubleValue() / totalcountPerLevel.get(level));
                }
                ACRatio.put(category, levelGroup);
        }

        return ACRatio;
}
```

Figure: 7.1.6

The *getRecommendedLevelsPerCategory()* returns Map of key value being the category and value being a list of integers that is recommended levels.

```
public static Map<String, List<Integer>> getRecommendedLevelsPerCategory(String username) throws IOException {
        Map<String, Map<Integer, Double>> ACRatios = getAcceptedRatioPerLevelPerCategory(username);

        Map<String, List<Integer>> recommendation = new HashMap<>();
        for (String category : ACRatios.keySet()) {
                Map<Integer, Double> levelRatios = ACRatios.get(category);

                // get lowest level for category
                int recLevel = levelRatios.keySet().stream().min((i1, i2) -> Integer.compare(i1, i2)).get();
                double recLevelRatio = 0;

                for (Integer l : levelRatios.keySet()) {
                        if (levelRatios.get(l) > recLevelRatio) {
                                recLevel = l;
                                recLevelRatio = Math.min(levelRatios.get(l), 0.70);

                                if (levelRatios.get(l) > 0.70) {
                                        recLevel = l + 1;
                                        recLevelRatio = Math.min(levelRatios.get(l + 1), 0.70);
                                }
                        }
                }

                recommendation.put(category, new ArrayList<>());
                recommendation.get(category).add(recLevel);
                if (levelRatios.get(recLevel) > 0.40 && levelRatios.get(recLevel) < 0.70) {
                        recommendation.get(category).add(recLevel + 1);
                }
        }

        return recommendation;
}
```

Figure: 7.1.7

The *getRecommendedProblems()* method returns a list of recommended problems for a given user.

```
public static List<Problem> getRecommendedProblems(String username) throws IOException {
        List<Problem> allProblems = Problem.getProblems();
        List<Problem> userSolvedProblems = getAccepted(username);
        Map<String, List<Integer>> recommendedLevels = getRecommendedLevelsPerCategory(username);

        // filter out solved problems and problems of other levels
        List<Problem> filteredProblems = allProblems.stream()
                        .filter(problem -> !userSolvedProblems.contains(problem))
                        .filter(problem -> recommendedLevels.get(problem.category).contains(problem.level))
                        .collect(Collectors.toList());
        Collections.shuffle(allProblems);

        List<Problem> recommendedProblems = new ArrayList<>();

        // get 2 problems from each problem
        List<String> categories = allProblems.stream().map(problem -> problem.category).distinct().collect(Collectors.toList());
        for (String cat : categories) {
                recommendedProblems.addAll(filteredProblems.stream().filter(problem -> problem.category.equals(cat)).limit(2).collect(Collectors.to
        }

        return recommendedProblems;
}
```

Figure: 7.1.8

For the *Problem* class there are these are the variables.

```
    private static final String PROBLEM_INFO_FILE = "ProblemInfo.csv";
    private static final String GET_PROBLEM_FROM_PID_URL = "http://uhunt.felix-halim.net/api/p/id/%d";
    private static final String GET_PROBLEM_FROM_PNUM_URL = "http://uhunt.felix-halim.net/api/p/num/%d";

    int problemNum;
    int problemID;
String category;
    int level;
```

Figure: 7.1.9

This is the constructor method of *Problem* Class.

```
public Problem(int problemNum, int problemID, String category, int level) {
        this.problemNum = problemNum;
        this.problemID = problemID;
        this.category = category;
        this.level = level;
}
```

Figure: 7.1.10

Important methods in Problem class are *getProblems()*, *getProblemNum()*, *getProblemID()*, *getProblemStats()* and *getCountPerCategory()*.

```
public static ArrayList<Problem> getProblems() throws FileNotFoundException {
        Scanner fileScanner = new Scanner(new File(PROBLEM_INFO_FILE));
        fileScanner.useDelimiter("\n|,");

        // Ignore first line, it contains file headers
        fileScanner.nextLine();

        ArrayList<Problem> problems = new ArrayList<>();
        while (fileScanner.hasNext()) {
                int problemNum = fileScanner.nextInt();
                int problemID = fileScanner.nextInt();
                String category = fileScanner.next();
                int level = fileScanner.nextInt();

                problems.add(new Problem(problemNum, problemID, category, level));
        }

        return problems;
}
```

Figure: 7.1.11

getProblems() method is responsible for returning a list of problems, where the list consist of object type Problem which contains problem number, problem ID, category and difficulty level of each problems in the list.

```
public static int getProblemNum(int problemID) throws IOException {
        URL url = null;
        try {
                url = new URL(String.format(GET_PROBLEM_FROM_PID_URL, problemID));
        } catch (MalformedURLException ex) {
                Logger.getLogger(User.class.getName()).log(Level.SEVERE, null, ex);
        }

        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("GET");
        Scanner responseStream = new Scanner(connection.getInputStream());
        responseStream.useDelimiter("\\A");

        JSONObject response = new JSONObject(responseStream.next());
        return response.getInt("num");
}
```

Figure: 7.1.12

getProblemNum() take problem ID as parameter and returns a integer value which represents the problem number that by which the numbering is done in UVa online judge.

```
public static int getProblemID(int problemNum) throws IOException {
        URL url = null;
        try {
                url = new URL(String.format(GET_PROBLEM_FROM_PNUM_URL, problemNum));
        } catch (MalformedURLException ex) {
                Logger.getLogger(User.class.getName()).log(Level.SEVERE, null, ex);
        }

        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("GET");
        Scanner responseStream = new Scanner(connection.getInputStream());
        responseStream.useDelimiter("\\A");

        JSONObject response = new JSONObject(responseStream.next());
        return response.getInt("pid");
}
```

Figure: 7.1.13

getProblemID() take problem number as parameter and returns a integer value which represents the problem ID that by which the numbering is done in uHunt toolkit. As gathering any information about problem solved by a user from User statistics require problem ID.

```
public static Map<String, Map<Integer, Long>> getProblemStats(ArrayList<Problem> problems) {
        HashMap<String, Map<Integer, Long>> stats = new HashMap<>();

        List<String> categories = problems.stream().map(problem -> problem.category).collect(Collectors.toList());
        for (String cat : categories) {
                List<Problem> categoryProblems = problems.stream().filter(problem -> problem.category.equals(cat)).collect(Collectors.toList());
                Map<Integer, Long> levelGroup = categoryProblems.stream().collect(Collectors.groupingBy(Problem::getLevel, Collectors.counting()));

                stats.put(cat, levelGroup);
        }

        return stats;
}
```

Figure: 7.1.14

getProblemStats() takes a list of problems of object type Problem and returns a HashMap stats whose key value is a string which represents a category and a the value is another Map whose key value is difficulty level and the value represents the count or number of problems in that category and difficulty level.

```
public static Map<String, Long> getCountPerCategory(ArrayList<Problem> problems) {
        return problems.stream().collect(Collectors.groupingBy(Problem::getCategory, Collectors.counting()));
}
```

Figure: 7.1.15

getCountPerCategory() takes a problem list as parameter and returns a Map whose key is category and value is total number of problems in that category.

CHAPTER 8 - RESULT AND ANALYSIS

We have tested our system taking 12 active users of UVa online judge. Among those 12 users we have picked 3 users (one who is a veteran, one who is mid-level and another who is a beginner) for analyzing the results. The following data are retrieved on the 2nd of August, 2017 and is subject to change afterwards.

1) The Veteran User

Our Recommendation for the user:

User Name: **isat1729**

Problem ID	Problem Number	Category	Level
3402	12250	Introduction	2
3431	12279	Introduction	2
104	168	Graph	3
254	318	Graph	3
166	230	Data Structures and Libraries	4
330	394	Data Structures and Libraries	3
88	152	(Computational) Geometry	3
127	191	(Computational) Geometry	2
181	245	String Processing	4
242	306	String Processing	3
2204	11247	Mathematics	4
2288	11313	Mathematics	3
868	927	Problem Solving Paradigms	3
3678	1237	Problem Solving Paradigms	3

Analysis:

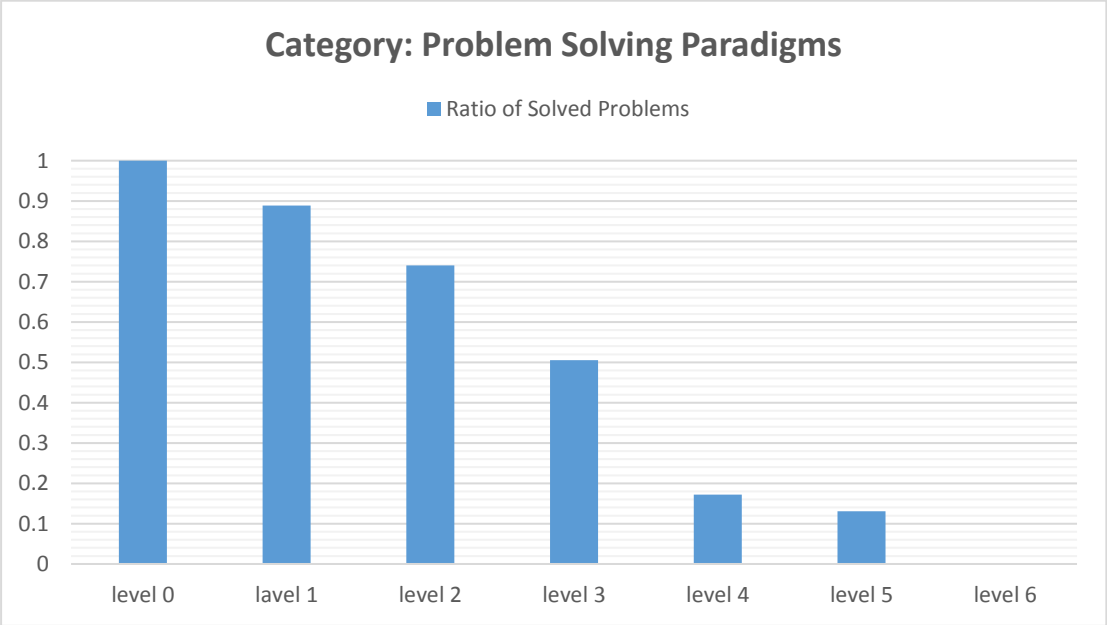


Figure: 8.1

From the chart we can see that in Problem Solving Paradigms user isat1729 has solved more than 50% level 3 problems but has solved less than 20% problems in level 4 and level 5 therefore the user needs to solve a bit more in level 3 in order to level up.

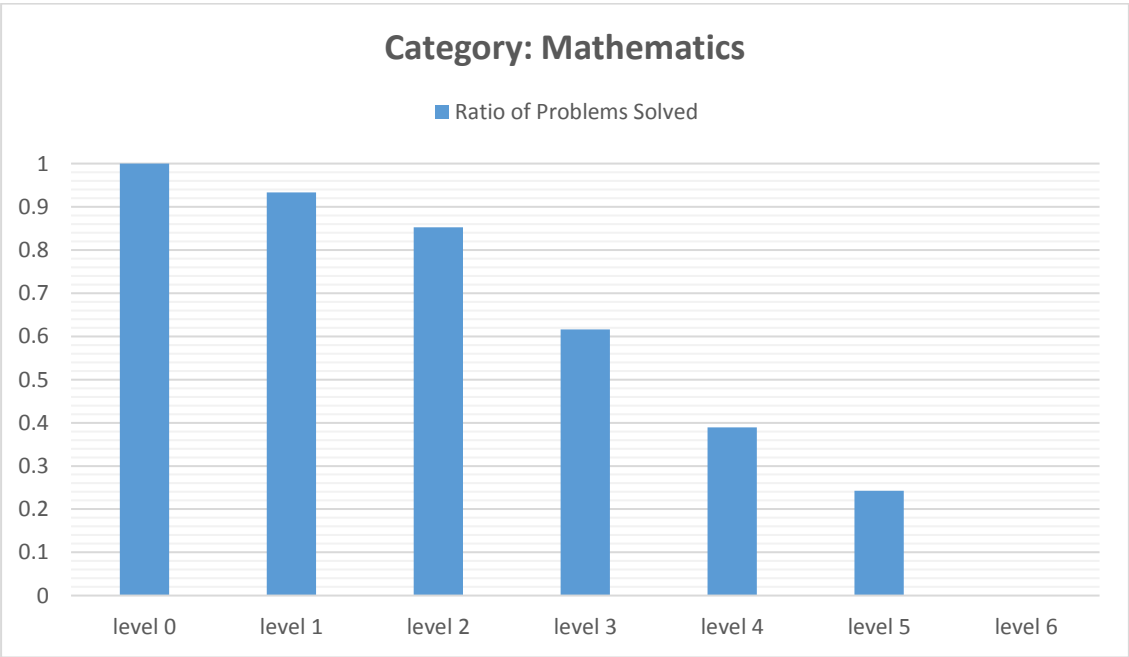


Figure: 8.2

From the chart we can see that in Mathematics user isat1729 has solved more than 60% level 3 problems but has solved nearly 40% problems in level 4 therefore the user needs to solve a bit more level 3 problem and is capable enough to solve level 4 problem.

2) The Mid-Level User

Our Recommendation for the user:

User Name: **holahmeds**

Problem ID	Problem Number	Category	Level
192	256	Problem Solving Paradigms	2
1917	10976	Problem Solving Paradigms	2
127	191	(Computational) Geometry	2
314	378	(Computational) Geometry	2
1222	10281	Mathematics	2
1410	10469	Mathematics	1
54	118	Graph	2
216	280	Graph	2
3402	12250	Introduction	2
3431	12279	Introduction	2
385	444	String Processing	2
424	483	String Processing	1
355	414	Data Structures and Libraries	2
423	482	Data Structures and Libraries	2

Analysis:

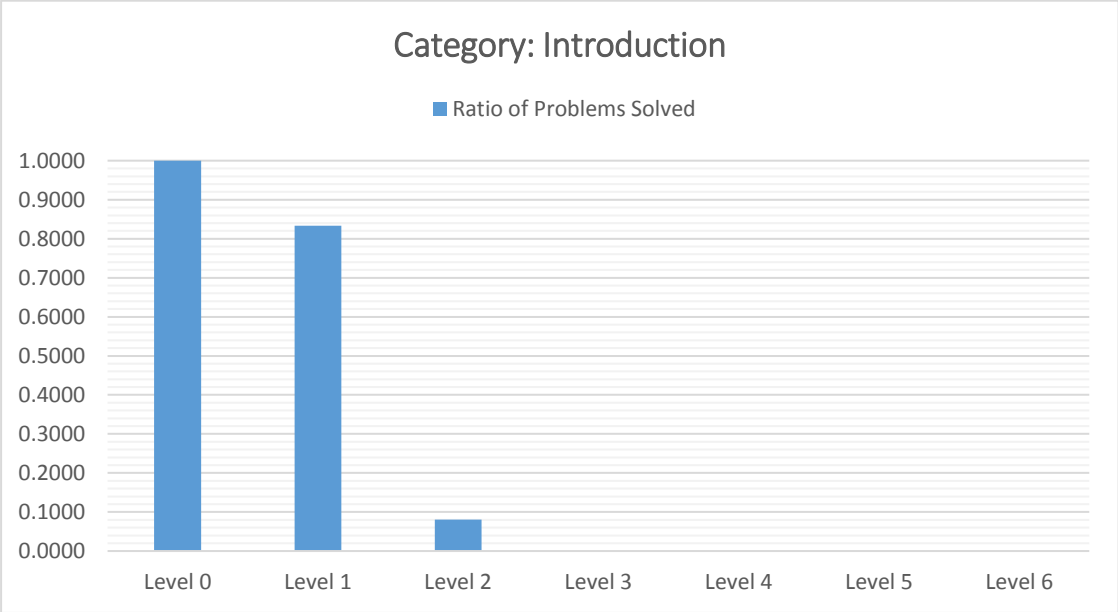


Figure: 8.3

In introduction category holahmeds have solved more 70% of level 1 problem so level 1 problems are no more necessary to be solved and holahmeds should be solving problems from level 2 only as only 8% level 2 problems has been solved so the recommended problems are both level 2 which fits the current skill level of the user as holahmeds needs to level up now.

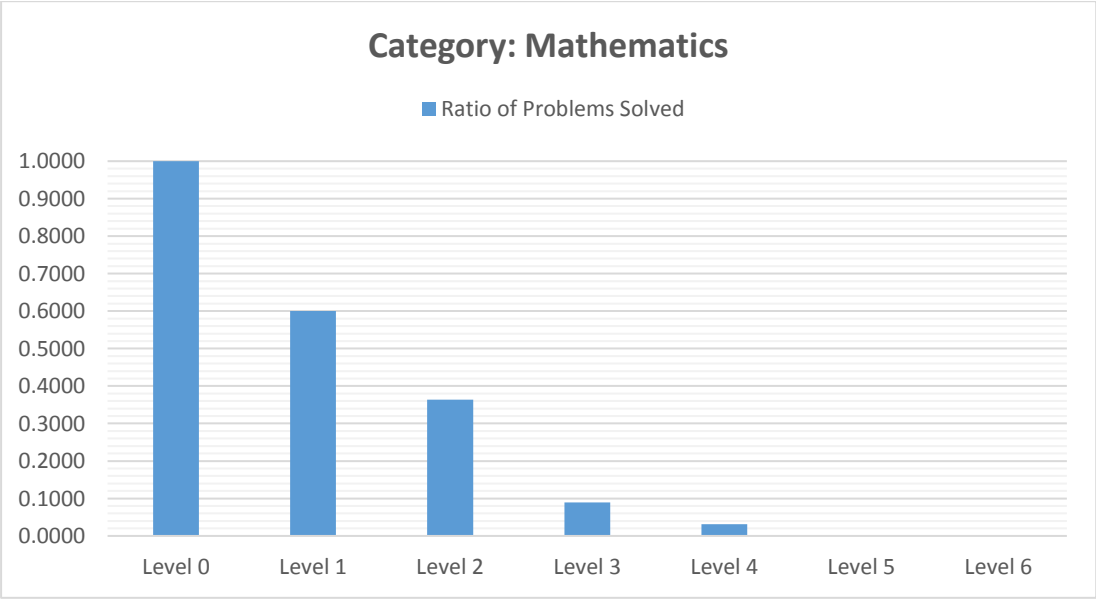


Figure: 8.4

In mathematics category holahmeds has solved less 40% in level 2 and less 70% in level 1 so the recommended problems were of level 2 and level 1 as it makes sense for him spend some more time solving level 1 along with level 2 problems.

3) The Beginner Level User

Our Recommendation for the user:

User Name: **parishaullah**

Problem ID	Problem Number	Category	Level
996	10055	Mathematics	0
1012	10071	Mathematics	0
41	105	Problem Solving Paradigms	1
982	10041	Problem Solving Paradigms	1
399	458	String Processing	0
424	483	String Processing	1
513	572	Graph	1
1246	10305	Graph	1
127	191	(Computational) Geometry	2
314	378	(Computational) Geometry	2
532	591	Data Structures and Libraries	1
991	10050	Data Structures and Libraries	1
3565	1124	Introduction	1
1491	10550	Introduction	1

Analysis:

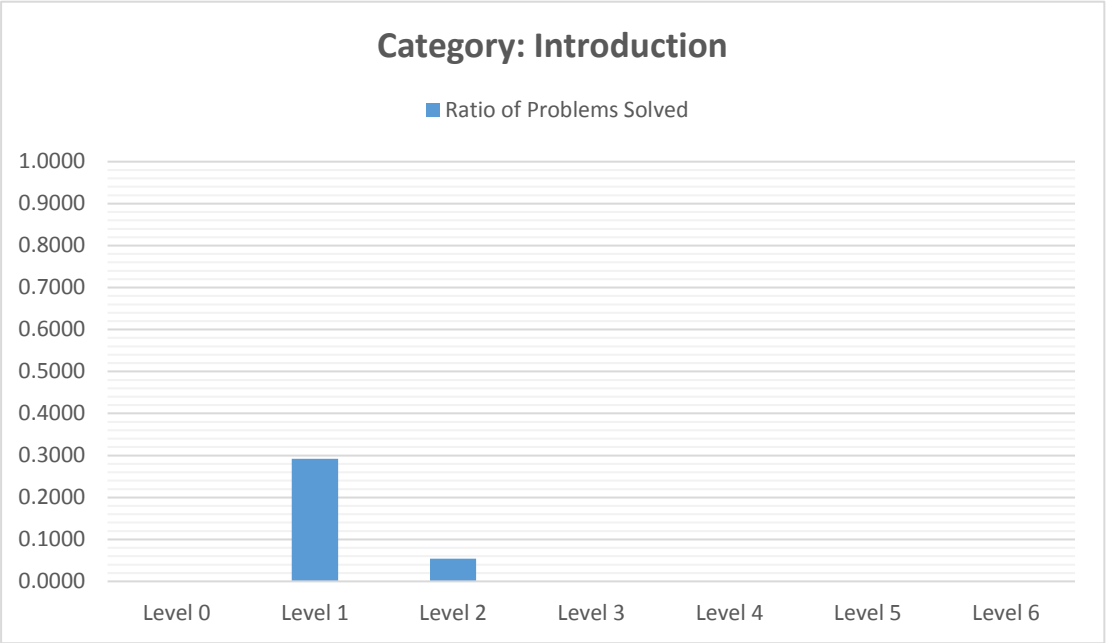


Figure: 8.5

In introduction category user parishaullah has solved very few problems. As she has solved less than 30% problems in level 1 it makes sense recommending her level 1 problems only for now.

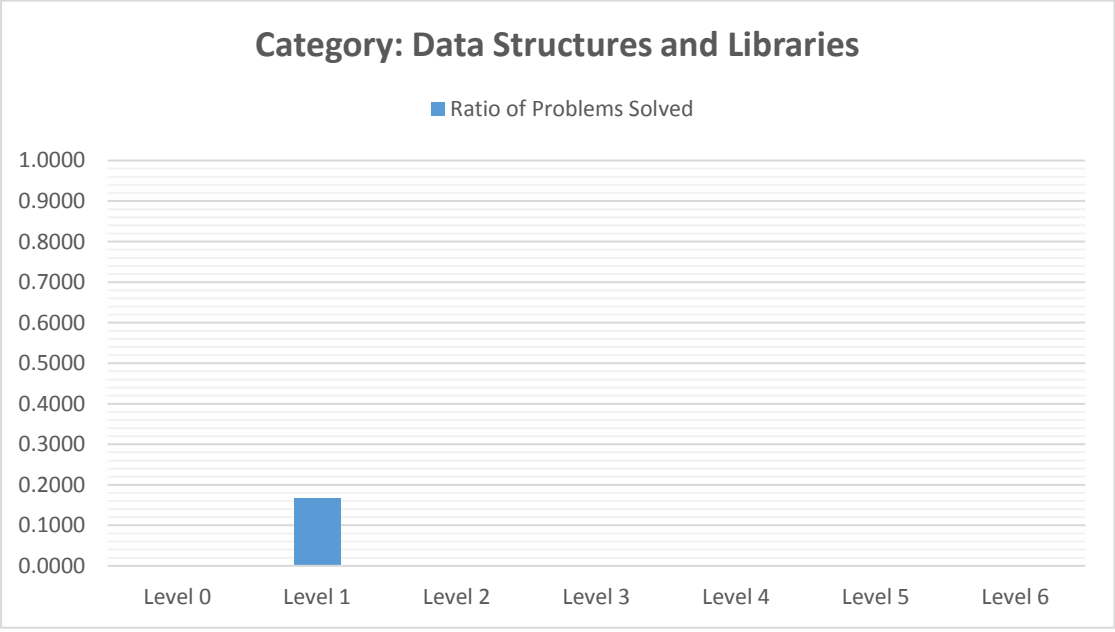


Figure: 8.6

Similarly in Data Structure category she was able to solve skip level 0 and solve level 1 so there is no point in recommending level 0 problems anymore so recommending two level 1 problems is sensible.

From the above results we can say that our recommendation system was able to recommend proper problems to the above users that will most likely to be solvable and will ensure proper growth of the users.

CHAPTER 9 - FURTHER IMPROVEMENTS AND FUTURE WORK

Our proposed solution meets all the goals we have devised for a recommender system recommending programming problems to sport programmers in an automated online judge. However, our solution took advantage of the fact that the toolkit, uHunt already provides the difficulty level of problems and categories of the problems. In a lot of other automated online judges like SPOJ, level and category of a programming problem are not provided beforehand. So, our future work would be to determine the level and category of a problem given these data are not provided. Besides, there are new problems which get added to the problem archive of an online judge. We have to ensure that these problems are not neglected by our recommender system. Moreover, there are certain programming problems which fall under multiple categories. It needs to be kept in mind that suggesting those problems to users would only work when the users are acquainted to those topics and have previous experience of solving problems from those topics separately.

1) New problems:

As time passes new problems get added to online judge's problem bank. And most of these problems comes from various contests that are held online or onsite. These problems are usually uncategorized and un-rated in terms their difficulty. Therefore there is exist the problem of classifying the problem. For finding the category of new problems Machine Learning might be able to play a role if a model can be trained with available datasets.

2) Intelligent system that helps faster growth for a problem solver:

Further improvement that can be implemented in near future is how to ensure a fast level up of particular category at a time. The plan is to recommend problems of particular category more in number so that the user focuses on that particular category for certain period of time until leveling up in that category. After leveling up in that particular category the recommender will focus on another category where

there faster chance of leveling up and hence follow a similar pattern of recommendation.

3) Dealing with problems that falls under multiple categories:

Advanced level problems which are of much higher difficult level often requires knowledge and experience of more than one category. For example there can be a problem that will not only require breadth first search but also dynamic programming to solve it. Hence this requires expertise of graph and problem solving paradigm. Also some problems can be solved by more one approach like a problem that can be solved through dynamic programming can sometimes be solved using mathematics. So a much robust model need to be built that can deal multiple category issues.

CHAPTER 10 - REFERENCES

[1] (n.d.). Last accessed August 10, 2017, from http://uhunt.felix-halim.net/api

[2] (n.d.). Last accessed August 10, 2017, from https://maven.apache.org/what-is-maven.html

[3] (n.d.). Last accessed August 10, 2017, from https://thesocietea.org/2015/07/programming-concepts-compiled-and-interpreted-languages/

[4] (n.d.). Last accessed August 10, 2017, from http://www.oracle.com/technetwork/java/intro-141743.html#318

[5] Goldberg, D., Nichols, D., Oki, B. M., and Terry, D. (1992). Using Collaborative Filtering to Weave an Information Tapestry. Communications of the ACM. December.

[6] Konstan, J., Miller, B., Maltz, D., Herlocker, J., Gordon, L., and Riedl, J. (1997). GroupLens: Applying Collaborative Filtering to Usenet News. Communications of the ACM, 40(3), pp. 77-87.

[7] Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., and Riedl, J. (1994). GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In Proceedings of CSCW '94, Chapel Hill, NC.

[8] Hill, W., Stead, L., Rosenstein, M., and Furnas, G. (1995). Recommending and Evaluating Choices in a Virtual Community of Use. In Proceedings of CHI '95.

[9] Shardanand, U., and Maes, P. (1995). Social Information Filtering: Algorithms for Automating 'Word of Mouth'. In Proceedings of CHI '95. Denver, CO.

[10] Breese, J. S., Heckerman, D., and Kadie, C. (1998). Empirical Analysis ofbPredictive Algorithms for Collaborative Filtering. In Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence, pp. 43-52.

[11] Aggarwal, C. C., Wolf, J. L., Wu K., and Yu, P. S. (1999). Horting Hatches an Egg: A New Graph-theoretic Approach to Collaborative Filtering. In Proceedings of the ACM KDD'99 Conference. San Diego, CA. pp. 201-212.

[12] Schafer, J. B., Konstan, J., and Riedl, J. (1999). Recommender Systems in E-Commerce. In Proceedings of ACM E-Commerce 1999 conference.

[13] Anil Poriya, Tanvi Bhagat, Neev Patel and Rekha Sharma. Article: Non-Personalized Recommender Systems and User-based Collaborative Recommender Systems. *International Journal of Applied Information Systems*6(9):22-27, March 2014.

[14] Gleb Beliakov, Tomasa Calvo and Simon James "Aggregation of preferences in recommender systems".

[15] Ayhan Demiriz "Enhancing Product Recommender Systems on Sparse Binary Data".

[16] Debajyoti Mukhopadhyay, Ruma Dutta, Anirban Kundu and Rana Dattagupta "A Product Recommendation System using Vector Space Model and Association Rule".

[17] Sarwar, B., M., Konstan, J. A., Borchers, A., Herlocker, J., Miller, B., and Riedl, J. (1998). Using Filtering Agents to Improve Prediction Quality in the GroupLens Research Collaborative Filtering System. In Proceedings of CSCW '98, Seattle, WA.

[18] Good, N., Schafer, B., Konstan, J., Borchers, A., Sarwar, B., Herlocker, J., and Riedl, J. (1999). Combining Collaborative Filtering With Personal Agents for Better Recommendations. In Proceedings of the AAAI-'99 conference, pp 439-446.

[19] Billsus, D., and Pazzani, M. J. (1998). Learning Collaborative Information Filters. In Proceedings of ICML '98. pp. 46-53.

[20] "Non-Personalized Recommender Systems with Pandas and Python" by Marcel Caraciolo, Artificial Intelligence in Motion, A blog about scientific python, Data, Machine Learning and Recommender systems(aimotion.blogspot.in).

[21] Manisha Hiralall “Recommender systems for e-shops” Vrije university, Amsterdam.

[22] Gerard M. Salton, Andrew K. C. Wong, and Chung-Shu Yang. A Vector Space Model for Automatic Indexing. Communications of the ACM, 18(11):613–620, November 1975.

[23] Daniel Billsus and Michael J. Pazzani. A hybrid user model for news story classification. In Proceedings of the Seventh International Conference on User Modeling.Banff, Canada, pages 99–108, 1999.

[24] Upendra Shardanand and Pattie Maes. Social information filtering: Algorithms for automating "word of mouth". In Irvin R. Katz, Robert L. Mack, Linn Marks, Mary Beth Rosson, and Jakob Nielsen, editors, Human Factors in Computing Systems, CHI ’95 Conference Proceedings, Denver, Colorado, USA, May 7-11, 1995, pages 210–217. ACM/Addison-Wesley, 1995.

[25] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. IEEE Transactions on Knowledge and Data Engineering, 17:734–749, June 2005.

[26] Soumen Chakrabarti. Mining the Web: Discovering Knowledge from Hypertext Data. Morgan-Kaufmann Publishers, 2002.