**An analysis of the Codeforces rating system**

Guang Yang

**Abstract (TBD)**

1 research problem and objectives

2 methods

3 key results or arguments

4 conclusions

**1. Introduction**

Multi-competitor ranking is a present and on-going research area, especially given the advent of massive online gaming. “It depends on a skill rating system to infer accurate player skills from historical data” (Minka et al., 2020) in order to match players with opponents similar to their levels. Microsoft Research proposed the TrueSkill method and TrueSkill 2, with TrueSkill 2 being their multi-competitor ranking method.

A related paper written by Minka et al. (2020), “TrueSkill 2: An improved Bayesian skill rating system”, presents “TrueSkill2, a collection of model changes to TrueSkill as well as a new system for estimating model parameters”. The improved method gives significantly more accurate skill ratings than the original TrueSkill method, reflected by a variety of indicators to a game studio.

Minka’s paper begins by illustrating a set of top priority of qualities needed by a modern game studio, then continues with what TrueSkill model has satisfied and what has not. Following this is the detail of the classic TrueSkill model. TrueSkill 2 is modified in certain ways to meet the requirement omitted by the classic TrueSkill model. The various requirements, or in other words, assumptions of the model, is vital for the theory of the paper to hold. Rigorous explanations on the validity of assumptions are vital as well.

For the parameter estimation section, proper values were assigned to different parameters. One purpose is to reduce ambiguity, such as fixing β to 1. The other purposes are explained in the paper to fit the design of the game itself. For different game applications, the parameters tend to differ, so game developers should adjust the model to suit better to their games.

The paper also includes the classification of confounding variables. The essential and basic part of the model is developed by disregarding those confounding variables. Then the paper classifies those variables to four categories, with elaboration in section 6, 7, 8 and 9. The algorithm is tested and shortcomings are found in each category, then the model is improved to yield a more accurate estimation. Some features not added are explained in section 10, and the main reason of not adding them is that they are overlapping with previous four categories.

There is also a version of Elo for multi-competitor games. (*Elo rating system,* 2021).One such version is the Elo-MMR rating system, elaborated by “An Elo-like System for Massive Multiplayer Competitions” (Ebtekar & Liu, 2021). The base case is the Bayesian model for multiple competitors, similar to Minka et al.’s paper but more complicated with more variables. Then the author proposes the two-phase algorithm for skill estimation in detail, and the elaboration has many advanced formulas and mathematical terms. After that is the discussion on skill evolution over time, and a term “pseudodiffusion” is put forward. A set of pseudocode helps illustrate the idea. Then the paper evaluates the theoretical effectiveness of the algorithm, with calculations of time complexity and optimizations. Finally, data from past contests of different competitive programming sites such as Codeforces and TopCoder is put into the algorithm to determine the effectiveness of prediction. In the appendix part, there is also proof of theorems used in the paper.

Another version is a multi-competitor Elo method applied on Formula One matches. The article “Who’s The Best Formula One Driver Of All Time?” (Moore, 2018), describes this rating method adjusted to rate the competitions with multiple competitors.

Similar to the Elo rating method, competitors are assigned an initial rating of 1300. The largest difference is that “each session or race is treated as if it were a round-robin 1-on-1 tournament. A driver who finishes second out of 15 cars is viewed as having gone 13-1 in this tournament, losing to the first-place finisher and defeating the rest” (Moore, 2018). In this version, only competitors’ ranking will determine its rating change, but the actual scores are not taken into account. My intended research topic included the effect of actual scores on rating change, so I would only learn the idea of this method. This paper will change the simple win-lose score into a weighted version of competitor’s points.

The article also points out that artificial adjustment on rating changes is necessary in order to prevent rating inflation or deflation. Without the adjustment, the initial uniform standard for determining competitor’s ability would fluctuate over time, certainly unfair for different competitors that stay active in different time.

Other variations also exist such as the Massey method. Greene et al. (2014) used several ranking methods to evaluate the strength of US Men’s Ice Hockey team. It concludes with the global ranking of the team, the chance to win medal in the 2014 Olympics, and the improvement of the team.

For head-to-head sports, Greene et al.’s paper uses the Massey method, the Elo method, and the TrueSkill method to analyze the US Men’s Ice Hockey team’s placement over time. After this is the comparison of the methods. In the comparison section, the Elo rating method is classified as straight Elo (holding k value constant), simple weighted and heavy weighted. I may adapt this classification process in my paper as well. Straight Elo rating method predicts better result than the other two variations of the Elo method. Then the passage evaluates the predictions of the three rating methods quantitively over time.

The major content of this paper is head-to-head sports. Rating systems on multi-competitor sports are mentioned, but unfortunately, they cannot be analyzed in the same way as head-to-head sports.

In this paper, I will design and analyze the rating system for Codeforces, the most famous website that hosts international competitive programming competitions. (*Codeforces,* 2021), If competitors participate in rated contests, their rating will change. I will research the way of rating change after the contest based on competitor’s performance.

The effectiveness of the rating system will be evaluated based on data of rating change of participants from the past contests. I will compare the rating change predicted by my method and program to the official rating change, and use predictability indicators to measure the extent of accuracy.

**2. Description of Codeforces Contest Mechanism**

This part is particularly useful for readers desired to investigate the Codeforces contest mechanism.

A newly registered user has default rating 1500. There are four divisions in Codeforces contests: Div.1 requires a rating greater than 1900, Div.2 requires a rating less than 2100, Div.3 requires a rating less than 1600, and Div.4 requires a rating less than 1400. Although Div.4 existed in the Codeforces history, it was only held once in [Codeforces Round 640 (Div. 4)](https://codeforces.com/contest/1561/standings). Contests in other three divisions are held regularly.

There are two set of data for a problem: one is pretest and the other one is system test. Contestants submit their code to see if it passes every test point in the pretest. If it was successful, then the contestant can wait till the end of the contest and wait for the system test. Only codes that pass system test earn the scores for a problem.

For the problems in a contest, every problem has an initial score, with the convention of the easiest task, A, worth of 500 points and the hardest task, E or F or G, worth of 3500 points. Other problems have different points but the point increases with difficulty. Also, there is a mechanism of problems devaluating with time. For a regular 2-hour contest, the value of a problem decreases at a rate of per minute. If the problem is successfully accomplished, then it will stop devaluating. There is penalty for submitting the problems as well. For each unsuccessful attempt that fails the pretest, the contestant loses 50 marks to the problem.

Another interesting concept, also the unique and symbolistic feature of Codeforces contests, is “hacking.” Dozens of contestants are allocated to the same room, and they can view each other’s code after successfully passing the pretest of a problem. Then contestants in the room can meticulously design some data to kill other people’s code. If it was successful, i.e., other people’s code fails the special data, then it is a “successful hack,” and this set of data will be used at the system test. A successful hack brings extra 100 points to this problem; however, an “unsuccessful hack” result in a 50 points reduction. Although hacking other’s codes is fun, the risk is noticeable, and most hackings are not easy since the pretest usually consists of tens or hundreds of data points.

Despite all those rules, no matter how many attempts or unsuccessful hacks the contestant did, when he or she solves the problem, the score for it cannot drop below 30% of its original points.

“For example, if the problem B was solved after 10 minutes of contest, then it costs points. For each attempt there is penalty of 50 points. So, if the problem B was solved after 10 minutes from the beginning with the third attempt, the score for it is points.” (https://codeforces.com/blog/entry/456) Table 1 is copied from official blog to help illustrate the rule.

Table . Codeforces contest mechanism sample

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Problem | Max. score | Min. score (30%) | Loss (points per minute) | Score at the contest end |
| A | 500 | 150 | 2 points | 260 |
| B | 1000 | 300 | 4 points | 520 |
| C | 1500 | 450 | 6 points | 780 |
| D | 2000 | 600 | 8 points | 1040 |
| E | 2500 | 750 | 10 points | 1300 |

**3. My Rating System – Multi-Competitor Elo Method**

I decided to change the classical Elo rating method a little to produce a multi-competitor variant.

Assume the number of competitors in a given match is . Assume competitor A has ranking , original rating , predicted change in rating predicted new rating , and official new rating . Competitor B has ranking , original rating , predicted change in rating predicted new rating , and official new rating .

Define the function that returns the probability of A losing to B.

Then, with the idea of “each session or race is treated as if it were a round-robin 1-on-1 tournament” (Moore, 2018), every competitor in this Codeforces competition should play a round-robin as well. Therefore, I assume that every two competitors had a competition based on their rating, and the expected rank of competitor is the sum of all expected probability of player losing the match against every player. Hence,

However, there is a bit of imperfection here. is always because the player has equal ability when competing with oneself. In addition, the expected ranking for the best player, theoretically, is , but should be in conventions. So, I add to this ranking estimator.

Proposition: The sum of all expected rankings should equal .

Proof of proposition:

Which satisfies the requirement.

Define a function to calculate the change in the rating for competitor A and B, **given that A beats B**. If is true then A and B have the same ranking. Define be the probability that A beats B, and be the probability that B beats A.

In

Note: this function returns the pair .

I would go through every pair of competitors to calculate their change in rating. The final change in rating is the sum of all s in function. And the expected rating is:

The rating numbers in Codeforces are usually integers; hence, I use the function to convert the new rating to an integer.

Additionally, in order to prevent certain strong competitors from getting an extremely high rating from finishing #1 in several matches in a row, I decide to change the value for top competitors. This idea will be illustrated later.

For analyzing the effectiveness of my prediction, I use a “predictability index”, which is the “Mean Square Error” or , of the scorings of all round-robin matches. In a competition I record the expected new ratings and the real win-lose relationships between every pair of competitors. Then we have,

Where calculates the win-lose relationship between participant and .

Because there are pairs of matches from competitors, we need to divide this formula for by .

The idea of this formula is adapted from “http://opisthokonta.net/?p=1387” (Opisthokonta et al., 2016).

A smaller value indicates that my method works for the win-lose relationship for more pairs of competitors. For example, if competitor loses to competitor , and the expected rating of competitor is indeed higher than competitor , then it is a correct prediction for this pair of competitors.

**4. Apply it on Data and Check Result**

The data I use here mainly comes from the Codeforces website. The ranks and scores for each competitor in every competition is accessible on the website. According to the Codeforces competition rules, only users with a rating greater than 1900 are eligible to participate the Div.1 contests, which is the hardest among all divisions and the type with least participants, approximately 1000. This data size is large enough to analyze the rating system but not excessively large to waste a long time in program; therefore, I decide to collect the data of contestants’ rankings, handles (means ID in Codeforces), official old ratings and new ratings on the closest 20 Div.1 matches. The effectiveness of prediction is reflected by the Average value for the 20 matches, and I will call it .

Table . The Closest 20 Div.1 matches

|  |  |  |
| --- | --- | --- |
| Index | Contest Name | Contest ID |
| 1 | [Codeforces Round #673 (Div. 1)](https://codeforces.com/contest/1416) | 1416 |
| 2 | [Codeforces Round #680 (Div. 1, based on Moscow Team Olympiad)](https://codeforces.com/contest/1444) | 1444 |
| 3 | [Codeforces Round #681 (Div. 1, based on VK Cup 2019-2020 – Final)](https://codeforces.com/contest/1442) | 1442 |
| 4 | [Codeforces Round #683 (Div. 1, by Meet IT)](https://codeforces.com/contest/1446) | 1446 |
| 5 | [Codeforces Round #684 (Div. 1)](https://codeforces.com/contest/1439) | 1439 |
| 6 | [Codeforces Round #687 (Div. 1, based on Technocup 2021 Elimination Round 2)](https://codeforces.com/contest/1456) | 1456 |
| 7 | [Codeforces Round #691 (Div. 1)](https://codeforces.com/contest/1458) | 1458 |
| 8 | [Codeforces Round #692 (Div. 1, based on Technocup 2021 Elimination Round 3)](https://codeforces.com/contest/1464) | 1464 |
| 9 | [Codeforces Round #694 (Div. 1)](https://codeforces.com/contest/1470) | 1470 |
| 10 | [Codeforces Round #698 (Div. 1)](https://codeforces.com/contest/1477) | 1477 |
| 11 | [Codeforces Round #700 (Div. 1)](https://codeforces.com/contest/1479) | 1479 |
| 12 | [Codeforces Round #706 (Div. 1)](https://codeforces.com/contest/1495) | 1495 |
| 13 | [Codeforces Round #707 (Div. 1, based on Moscow Open Olympiad in Informatics)](https://codeforces.com/contest/1500) | 1500 |
| 14 | [Codeforces Round #709 (Div. 1, based on Technocup 2021 Final Round)](https://codeforces.com/contest/1483) | 1483 |
| 15 | [Codeforces Round #712 (Div. 1)](https://codeforces.com/contest/1503) | 1503 |
| 16 | [Codeforces Round #715 (Div. 1)](https://codeforces.com/contest/1508) | 1508 |
| 17 | [Codeforces Round #722 (Div. 1)](https://codeforces.com/contest/1528) | 1528 |
| 18 | [Codeforces Round #728 (Div. 1)](https://codeforces.com/contest/1540) | 1540 |
| 19 | [Codeforces Round #732 (Div. 1)](https://codeforces.com/contest/1545) | 1545 |
| 20 | [Codeforces Round #736 (Div. 1)](https://codeforces.com/contest/1548) | 1548 |

With the Python code from “<https://github.com/QAQrz/Codeforces-Rating-System/blob/master/spider_txt.py>” (*QAQrz/Codeforces-Rating-System: Codeforces rating System (third Party implementation),* 2017), I scrape the data from all the 20 contests in Table 1 to get the result of the closest 20 Div.1 matches.

My target is to minimize the value for each choice of . For example, when , the value for each match is shown in Table 2 below.

Table . MSE for each contest given K=16

|  |  |  |
| --- | --- | --- |
| Index | MSE | |
| 1 | 0.39443 | |
| 2 | 0.402534 | |
| 3 | 0.396674 | |
| 4 | 0.412647 | |
| 5 | 0.413538 | |
| 6 | 0.395988 | |
| 7 | 0.396692 | |
| 8 | 0.400353 | |
| 9 | 0.411328 | |
| 10 | 0.400943 | |
| 11 | 0.414437 | |
| 12 | 0.407261 | |
| 13 | 0.390727 | |
| 14 | 0.403505 | |
| 15 | 0.422167 | |
| 16 | 0.406531 | |
| 17 | 0.419262 | |
| 18 | 0.387294 | |
| 19 | 0.421552 | |
| 20 | 0.430927 | |

In this example, the average for the 20 matches is 0.4064395.

Then I change the value of to see the pattern of values.

First, with an increment of 1, the data is shown below at Table 3.

Table . AMSE from different K values with increment 1

|  |  |
| --- | --- |
| K | AMSE |
| -1 | 0.757543 |
| 0 | 0.368619 |
| 1 | 0.122816 |
| 2 | 0.1218 |
| 3 | 0.177011 |
| 4 | 0.227971 |
| 5 | 0.267187 |
| 6 | 0.296857 |
| 7 | 0.319711 |
| 8 | 0.337735 |
| 9 | 0.352267 |
| 10 | 0.364209 |
| 11 | 0.374183 |
| 12 | 0.382629 |
| 13 | 0.389866 |
| 14 | 0.396133 |
| 15 | 0.401611 |
| 16 | 0.406439 |
| 17 | 0.410727 |
| 18 | 0.41456 |
| ­­­­­­­­­­­­­­­­19 | 0.418006 |
| 20 | 0.421122 |

Figure . AMSE from different K values with increment 1

The minimum point of lies between 1 and 3. Then I repeat the process again but with increments of 0.1 in region to find the desirable .

Table . AMSE from different K values with increment 0.1

|  |  |
| --- | --- |
| K | AMSE |
| 1 | 0.122816 |
| 1.1 | 0.115815 |
| 1.2 | 0.110977 |
| 1.3 | 0.108031 |
| 1.4 | 0.106728 |
| 1.5 | 0.106836 |
| 1.6 | 0.108143 |
| 1.7 | 0.110455 |
| 1.8 | 0.113601 |
| 1.9 | 0.117427 |
| 2 | 0.1218 |
| 2.1 | 0.126604 |
| 2.2 | 0.13174 |
| 2.3 | 0.137123 |
| 2.4 | 0.142682 |
| 2.5 | 0.148358 |
| 2.6 | 0.154099 |
| 2.7 | 0.159866 |
| 2.8 | 0.165625 |
| 2.9 | 0.171347 |
| 3 | 0.177011 |

Figure . AMSE from different K values with increment 0.1

The minimum point of lies between 1.3 and 1.5. Then I repeat the process the third time but with increments of 0.01 in region to find the desirable .

Table 6. AMSE from different K values with increment 0.01

|  |  |
| --- | --- |
| K | AMSE |
| 1.3 | 0.108031 |
| 1.31 | 0.107831 |
| 1.32 | 0.107647 |
| 1.33 | 0.107479 |
| 1.34 | 0.107326 |
| 1.35 | 0.107189 |
| 1.36 | 0.107067 |
| 1.37 | 0.106961 |
| 1.38 | 0.106869 |
| 1.39 | 0.106791 |
| 1.4 | 0.106728 |
| 1.41 | 0.106679 |
| 1.42 | 0.106644 |
| 1.43 | 0.106622 |
| 1.44 | 0.106614 |
| 1.45 | 0.106619 |
| 1.46 | 0.106638 |
| 1.47 | 0.106668 |
| 1.48 | 0.106712 |
| 1.49 | 0.106768 |
| 1.5 | 0.106836 |

Figure 3. AMSE from different K values with increment 0.01

According to the graph, the value for the average minimum square error value over all competitions is , accurate to two decimal places. The value corresponding to this choice is 0.106614.

I also displayed the difference between my predicted new rating and official new rating for every competitor. To my surprise, the difference is very large for the several competitors at the top. In order to prevent their ratings from getting too large, in other words, prevent inflation of rating, the official rating system implements a way to control the rating change for those top competitors. This stimulates me to use a separate for the top competitors that is less than the normal value, so the rating of the top will change less.

When , I record the number of competitors at top rankings with the absolute value of the difference between my predicted new rating and official new rating more than 30. I would call these competitors having a large difference.

Table 7. Number of competitors with a large difference

|  |  |
| --- | --- |
| Index | # of top competitors |
| 1 | 6 |
| 2 | 6 |
| 3 | 4 |
| 4 | 11 |
| 5 | 2 |
| 6 | 8 |
| 7 | 6 |
| 8 | 3 |
| 9 | 4 |
| 10 | 5 |
| 11 | 13 |
| 12 | 8 |
| 13 | 5 |
| 14 | 8 |
| 15 | 10 |
| 16 | 9 |
| 17 | 9 |
| 18 | 1 |
| 19 | 2 |
| 20 | 12 |

The average value is 6.6, and to the nearest integer is 7. Therefore, I decide to apply the new value, call it that is less than , for the top 7 competitors.

Assume for the following cases. With the similar strategy, the minimum point of lies between -1 and 1.44 (negative sign included for consistency). With increment of 0.1 in region , the respective value is shown below:

Table 8. AMSE from different K2 values with increment 0.1

|  |  |
| --- | --- |
| K2 | AMSE |
| -1 | 0.107143 |
| -0.9 | 0.107114 |
| -0.8 | 0.107085 |
| -0.7 | 0.107056 |
| -0.6 | 0.107028 |
| -0.5 | 0.107001 |
| -0.4 | 0.106974 |
| -0.3 | 0.106948 |
| -0.2 | 0.106922 |
| -0.1 | 0.106897 |
| 0 | 0.106873 |
| 0.1 | 0.106849 |
| 0.2 | 0.106827 |
| 0.3 | 0.106805 |
| 0.4 | 0.106784 |
| 0.5 | 0.106763 |
| 0.6 | 0.106744 |
| 0.7 | 0.106725 |
| 0.8 | 0.106707 |
| 0.9 | 0.10669 |
| 1 | 0.106674 |
| 1.1 | 0.106659 |
| 1.2 | 0.106645 |
| 1.3 | 0.106632 |
| 1.4 | 0.106610 |

Figure 4. AMSE from different K2 values with increment 0.1

The minimum value from this new approach is 0.106610, when .

**The shape could tell that another indicator is better? Future work**

**5. Conclusion**

In this essay, I first introduce some literature work of other literature works on the ranking method. Then I give an introduction of the Codeforces rating system. I develop my own rating method based on the Elo method and change a little to produce a multi-competitor Elo method. After this, I collect the data of 20 closest Div.1 matches to test the predictability of my method. I adjust my value based to minimize the value, then add another special for the top 7 competitors to improve my method. The final value is 0.106610, small enough to assure that my rating method is reliable.

However, my method is undoubtedly not enough. Despite the fact that the first several competitors have a more accurate rating, there is still much difference between the official rating and predicted rating. I thought a way to set different values for different part of ranking instead of using a single for every match, but it requires a significant amount of calculation and time to tune the value properly. The inaccurateness of my prediction is a limitation, and I hope in the future I can develop a better method. Hopefully I can reduce my value to 0.05 in that method.

: Include future work, 5/6 parts stronger

**Bibliography**

1. Minka, T., Cleven, R. & Zaykov, Y., 2020. TrueSkill 2: An Improved BAYESIAN skill rating system. *Microsoft Research*. Available at: https://www.microsoft.com/en-us/research/publication/trueskill-2-improved-bayesian-skill-rating-system/ [Accessed August 17, 2021].

2. Justin Moore, R.D.and N.P., 2018. Who's the best Formula one driver of all time? *FiveThirtyEight*. Available at: https://fivethirtyeight.com/features/formula-one-racing/ [Accessed August 17, 2021].

3. GREENE, P.E.T.E.R. et al., 2014. *RANKING METHODS FOR OLYMPIC SPORTS: A CASE STUDY BY THE U.S. OLYMPIC COMMITTEE AND THE COLLEGE OF CHARLESTON*. Available at: https://blogs.cofc.edu/math/files/2014/10/P140617006-2iy2vvx.pdf [Accessed August 17, 2021].

4. Ebtekar, A. & Liu, P., 2021. Aram Ebtekar Vancouver, BC, Canada - arxiv. *arxiv.org e-Print archive*. Available at: https://arxiv.org/pdf/2101.00400 [Accessed August 17, 2021].

5. Anon, 2021. Codeforces. *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Codeforces [Accessed September 7, 2021].

6. Anon, 2021. Elo rating system. *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Elo\_rating\_system [Accessed September 7, 2021].

7. Mirzayanov, M., 2015. Open Codeforces rating SYSTEM [UPDATED on October 2015]. *Codeforces*. Available at: http://codeforces.com/blog/entry/20762 [Accessed September 7, 2021].

8. QAQrz, 2017. QAQrz/Codeforces-Rating-System: Codeforces rating System (third Party implementation) dreamerblue, ed. *GitHub*. Available at: https://github.com/QAQrz/Codeforces-Rating-System [Accessed September 7, 2021].

9. Opisthokonta et al., 2016. opisthokonta.net. *opisthokontanet*. Available at: http://opisthokonta.net/?p=1387 [Accessed September 7, 2021].

**Appendix**

C++ code for calculating the expected rating change:

1. #include <cstdio>
2. #include <iostream>
3. #include <fstream>
4. #include <cmath>
5. #include <algorithm>
6. #include <string>
7. #include <vector>
8. #define DEBUG printf("passing [%s] in line %d\n",\_\_FUNCTION\_\_,\_\_LINE\_\_);
9. **using** **namespace** std;
11. **struct** user {
12. **double** rank;
13. string handle;
14. **int** old\_rating;
15. **double** new\_rating = 0.0;
16. **int** official\_new\_rating;
17. **double** exp\_ranking = 0.5;
18. **double** delta = 0.0;
19. string validation;
20. } u;
21. vector<user> users;
22. string contest\_ids[21] = {"1416","1444","1442","1446","1439","1456","1458","1464","1470","1477","1479","1495","1500","1483","1503","1508","1528","1540","1545","1548"};
23. **double** aveMSE;
25. string to\_string(user &u) {
26. **char** s[200];
27. sprintf(s, "%5.0f  %-24s seed: %12.6f  rating: %4d -> %4f vs %4d%s\n",
28. round(u.rank),
29. u.handle.c\_str(),
30. u.exp\_ranking,
31. u.old\_rating,
32. u.new\_rating,
33. u.official\_new\_rating,
34. u.validation.c\_str()
35. );
36. **return** string(s);
37. }
39. string to\_string(vector<user> &users) {
40. string s;
41. **for**(**int** i = 0; i < users.size(); i++) {
42. s += to\_string(users[i]);
43. }
44. **return** s;
45. }
47. **void** reassign\_rank() {
48. **int** last\_idx = 0, last\_rank = 1;
49. **for**(**int** i = 0; i < users.size(); i++) {
50. **if**(users[i].rank > last\_rank) {
51. **for**(**int** j = last\_idx; j < i; ++j)
52. users[j].rank = i;
53. last\_idx = i;
54. last\_rank = users[i].rank;
55. }
56. }
57. **for**(**int** i = last\_idx; i < users.size(); i++)
58. users[i].rank = users.size();
59. }
61. //第一个人输掉的概率，也是第二个人赢的概率
62. **double** Prob(**double** rating1, **double** rating2) {
63. **return** 1.0 / (1.0 + pow(10, (rating1 - rating2) / 400.0));
64. }
66. //Function to calculate Elo rating
67. //K is a constant.
68. //Player A wins over Player B.
69. //tie = true if tie, false otherwise
70. pair<**double**,**double**> EloRating(**double** Ra, **double** Rb, **double** k, **bool** tie) {
71. **double** Pa,Pb,deltA,deltB;
72. Pb=Prob(Ra,Rb);//prob of b winning
73. Pa=Prob(Rb,Ra);//prob of a winning
74. **if**(tie) {
75. deltA=k\*(0.5-Pa);
76. deltB=k\*(0.5-Pb);
77. }
78. **else** {
79. deltA=k\*(1-Pa);
80. deltB=k\*(0-Pb);
81. }
82. **return** make\_pair(deltA,deltB);
83. }
85. **void** calExpRanking(**double** k) {
86. **for**(**int** i=0;i<users.size();i++) {
87. **for**(**int** j=0;j<users.size();j++) {
88. users[i].exp\_ranking+=Prob(users[i].old\_rating,users[j].old\_rating);
89. }
90. }
91. }
93. **void** newWork(**double** k) {
94. **for**(**int** i=0;i<users.size();i++) {
95. **for**(**int** j=i+1;j<users.size();j++) {
96. pair<**double**,**double**> pr;
97. **if**(users[i].rank<users[j].rank) {
98. pr=EloRating(users[i].old\_rating,users[j].old\_rating,k,**false**);
99. users[i].delta+=pr.first;
100. users[j].delta+=pr.second;
101. }
102. **else** **if**(users[i].rank>users[j].rank) {
103. pr=EloRating(users[j].old\_rating,users[i].old\_rating,k,**false**);
104. users[j].delta+=pr.first;
105. users[i].delta+=pr.second;
106. }
107. **else** {
108. pr=EloRating(users[i].old\_rating,users[j].old\_rating,k,**true**);
109. users[i].delta+=pr.first;
110. users[j].delta+=pr.second;
111. }
112. }
113. }
114. }
116. **void** update() {
117. **for**(**int** i=0;i<users.size();i++) {
118. users[i].new\_rating=users[i].old\_rating+users[i].delta;
119. }
120. }
122. **double** validate() {
123. **double** tot=0.0;
124. **for**(**int** i=0;i<users.size();i++) {
125. **for**(**int** j=0;j<users.size();j++) {
126. **if**(users[i].rank<users[j].rank) {
127. tot+=pow(Prob(users[j].new\_rating,users[i].new\_rating)-1,2);
128. }
129. **else** **if**(users[i].rank>users[j].rank) {
130. tot+=pow(Prob(users[j].new\_rating,users[i].new\_rating)-0,2);
131. }
132. **else** {
133. tot+=pow(Prob(users[j].new\_rating,users[i].new\_rating)-0.5,2);
134. }
135. }
136. }
137. **char** s[20];
138. **for**(**int** i = 0; i < users.size(); i++) {
139. **if**(users[i].new\_rating != users[i].official\_new\_rating) {
140. sprintf(s, " [diff: %5f]", users[i].new\_rating - users[i].official\_new\_rating);
141. users[i].validation = string(s);
142. }
143. }
144. **int** amount=users.size()\*(users.size()-1)/2;
145. **return** tot/amount;
146. }
148. **int** main() {
149. ofstream outAMSE("tests/AMSE.csv", ios::out | ios::trunc);
150. outAMSE << "K" << "," << "AMSE" << endl;
151. ofstream outMSE("tests/MSE.csv", ios::out | ios::trunc);
152. outMSE << "Index" << "," << "MSE" << endl;
154. **for**(**double** K=1.3;K<=1.5001;K+=0.01) {
155. aveMSE=0;
156. **for**(**int** i=0;i<20;i++) {
157. users.clear();
158. string contest\_id = contest\_ids[i];
159. string in\_file = "tests/cf\_rating\_official\_" + contest\_id + ".txt";
160. ifstream in(in\_file, ios::in);
161. **while**(in >> u.rank >> u.handle >> u.old\_rating >> u.official\_new\_rating)
162. users.push\_back(u);
163. in.close();
164. reassign\_rank();
165. calExpRanking(K);
166. newWork(K);
167. update();
168. **double** MinSqErr=validate();
170. //printf("In contest id %s, MinSqErr for simulation: %.8f\n",contest\_id.c\_str(),MinSqErr);
171. outMSE << i+1 << "," << MinSqErr << endl;
173. aveMSE += MinSqErr;
174. string out\_file = "tests/cf\_rating\_result\_" + contest\_id + ".txt";
175. ofstream outFile(out\_file, ios::out);
176. outFile << to\_string(users);
177. outFile.close();
178. }
179. outAMSE << K << "," << aveMSE/20 << endl;
180. printf("K=%.2f, AveMinSqErr value: %.8f\n",K,aveMSE/20);
181. }
182. outAMSE.close();
183. **return** 0;
184. }