An Analysis of the Codeforces Rating System

Guang Yang

Abstract

Various rating and ranking systems have been prevalent in the world. In particular, the Codeforces rating system is intriguing, due to its unique contest mechanism and meticulously designed rating method. Codeforces is a website that hosts competitive programming contests, and this paper researches its rating system to make competitors clearer about it. The paper proposes a multi-competitor Elo rating method, based on the classical Elo method, to compute the predicted rating change of competitors after the contest. A "predictability index", $AMSE$, is defined to evaluate the accuracy of my prediction, and the value is 0.10639346 when $K = 1.44$ and $K_2 = 0.805$. Assigning different K values corresponding to different sections of contestants would further improve my simulation, which is a rich area for future work.

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 the previous four categories.

There is also a version of Elo for multi-competitor games. "The Elo system was originally invented as an improved chess-rating system over the previously used Harkness system" (*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, https://codeforces.com/, is a website that hosts competitive programming contests. "As of 2018, it has over 600,000 registered users" (*Codeforces,* 2021), and the number of users is increasing at a progressive rate. 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 a predictability indicator 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). 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 500 points and the hardest task, E or F or G, worth 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 $\frac{Initial\ Score}{250}$ 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 an 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 $1000 - 4 \cdot 10 = 960$ 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 $1000 - 4 \cdot 10 - 2 \cdot 50 = 860$ points." (Mirzayanov, 2010). Table 1 is copied from the official blog to help illustrate the rule.

Table 1. 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 n. Assume competitor i has ranking $rank_i$, original rating r_i, predicted change in rating Δr_i, predicted new rating r_i^*, and official new rating r_i'. Let competitor A be the 1st competitor and B be the 2nd competitor.

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

$$Prob(r_1, r_2) = \frac{1}{1 + 10^{\frac{r_1 - r_2}{400}}} \tag{1}$$

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 i is the sum of all expected probability of player i losing the match against every player. Hence,

$$r_i^* = \sum_{j=1}^{n} Prob(r_i, r_j) \tag{2}$$

However, there is a bit of imperfection in equation (2). $Prob(r_i, r_i)$ is always 0.5 because a player has equal ability when competing with oneself. In addition, the expected ranking for the best player, theoretically, is $0 + 0 + 0 + \cdots + 0 = 0.5$, but should be #1 in conventions. So, I add 0.5 to the ranking estimator in equation (2).

$$r_i^* = \sum_{j=1}^{n} Prob(r_i, r_j) + 0.5 \tag{3}$$

Proposition: The sum of all expected rankings should equal $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$.

Proof of proposition:

$$\begin{aligned}
\sum_{i=1}^{n} r_1^* &= \sum_{i=1}^{n} \left(\sum_{j=1}^{n} Prob(r_1, r_2) + 0.5 \right) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{n} Prob(r_1, r_2) + \sum_{i=1}^{n} 0.5 \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \big(Prob(r_1, r_2) + Prob(r_2, r_1) \big) + \sum_{i=1}^{n} Prob(r_i, r_i) + \sum_{i=1}^{n} 0.5 \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 + \sum_{i=1}^{n} 0.5 + \sum_{i=1}^{n} 0.5 \\
&= \frac{n(n-1)}{2} + \frac{n}{2} + \frac{n}{2} \\
&= \frac{n(n+1)}{2}
\end{aligned}$$

Which satisfies the requirement.

Define a function $EloRating(r_1, r_2, tie)$ to calculate the change in the rating for competitor A and B, **given that A beats B**. If tie is true then A and B have the same ranking. Define P_1 be the probability that A beats B, and P_2 be the probability that B beats A.

In $EloRating(r_1, r_2, tie)$:

$$P_2 = Prob(r_1, r_2)$$
$$P_1 = Prob(r_2, r_1)$$
$$\begin{cases} \begin{cases} \Delta r_1 = K(0.5 - P_1) \\ \Delta r_2 = K(0.5 - P_2) \end{cases}, & tie = true \\ \begin{cases} \Delta r_1 = K(1 - P_1) \\ \Delta r_2 = K(0 - P_2) \end{cases}, & tie = false \end{cases}$$

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

$$r_i^* = r_i + \Delta r_i \tag{4}$$

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

$$r_i^* = round(r_i + \Delta r_i) \tag{5}$$

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 K 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 MSE, 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,

$$MSE = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \left(Prob\left(r_j^*, r_i^*\right) - obs_{i,j}\right)^2 \tag{6}$$

Where $obs_{i,j}$ calculates the win-lose relationship between participant i and j.

$$obs_{i,j} = \begin{cases} 1, rank_i < rank_j \\ 0.5, rank_i > rank_j \\ 0, rank_i = rank_j \end{cases} \tag{7}$$

Because there are $\frac{n(n-1)}{2}$ pairs of matches from n competitors, we need to divide equation (6) by $\frac{n(n-1)}{2}$.

$$MSE = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \left(Prob(r_j^*, r_i^*) - obs_{i,j}\right)^2 \Big/ \frac{n(n-1)}{2} \quad (8)$$

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

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

4. Apply it on Data and Check Result

The data I use here 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 MSE value for the 20 matches, and I will call it $AMSE$.

Table 2. The Closest 20 Div.1 matches

Index	Contest Name	Contest ID
1	Codeforces Round #673 (Div. 1)	1416
2	Codeforces Round #680 (Div. 1, based on Moscow Team Olympiad)	1444
3	Codeforces Round #681 (Div. 1, based on VK Cup 2019-2020 – Final)	1442
4	Codeforces Round #683 (Div. 1, by Meet IT)	1446
5	Codeforces Round #684 (Div. 1)	1439
6	Codeforces Round #687 (Div. 1, based on Technocup 2021 Elimination Round 2)	1456
7	Codeforces Round #691 (Div. 1)	1458
8	Codeforces Round #692 (Div. 1, based on Technocup 2021 Elimination Round 3)	1464
9	Codeforces Round #694 (Div. 1)	1470
10	Codeforces Round #698 (Div. 1)	1477
11	Codeforces Round #700 (Div. 1)	1479

12	Codeforces Round #706 (Div. 1)	1495
13	Codeforces Round #707 (Div. 1, based on Moscow Open Olympiad in Informatics)	1500
14	Codeforces Round #709 (Div. 1, based on Technocup 2021 Final Round)	1483
15	Codeforces Round #712 (Div. 1)	1503
16	Codeforces Round #715 (Div. 1)	1508
17	Codeforces Round #722 (Div. 1)	1528
18	Codeforces Round #728 (Div. 1)	1540
19	Codeforces Round #732 (Div. 1)	1545
20	Codeforces Round #736 (Div. 1)	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 $AMSE$ value for each choice of K. For example, when $K = 16$, the MSE value for each match is shown in Table 2 below.

Table 3. 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 MSE for the 20 matches is 0.4064395.

Then I change the value of K to see the pattern of $AMSE$ values.

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

Table 4. 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

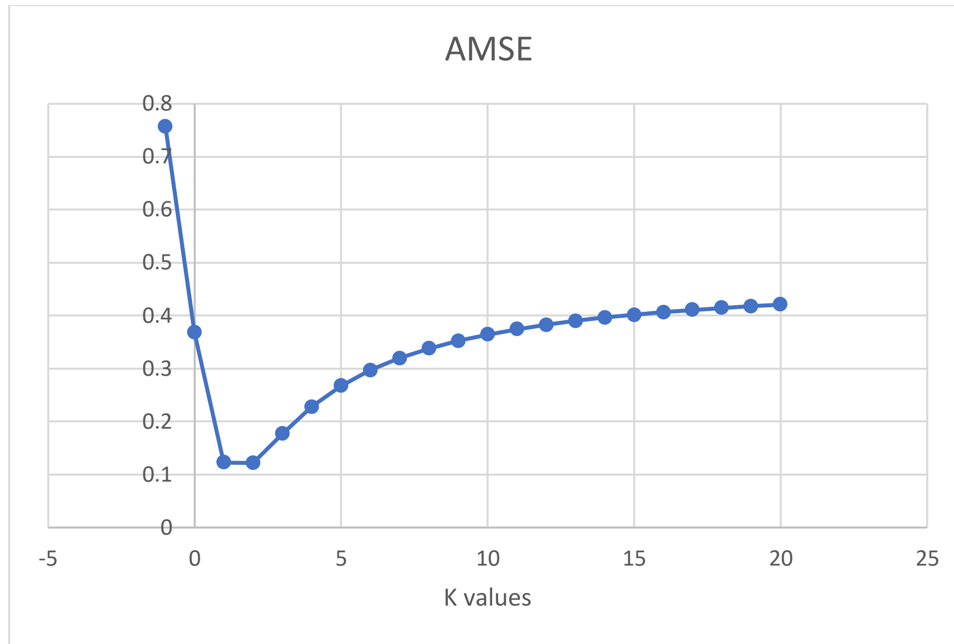

Figure 1. AMSE from different K values with increment 1

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

Table 5. 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

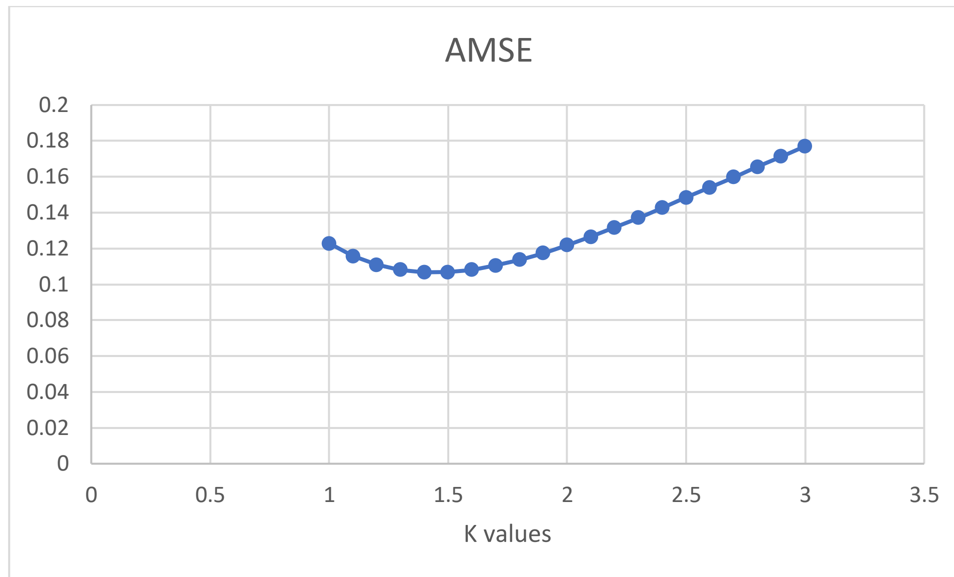

Figure 2. AMSE from different K values with increment 0.1

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

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

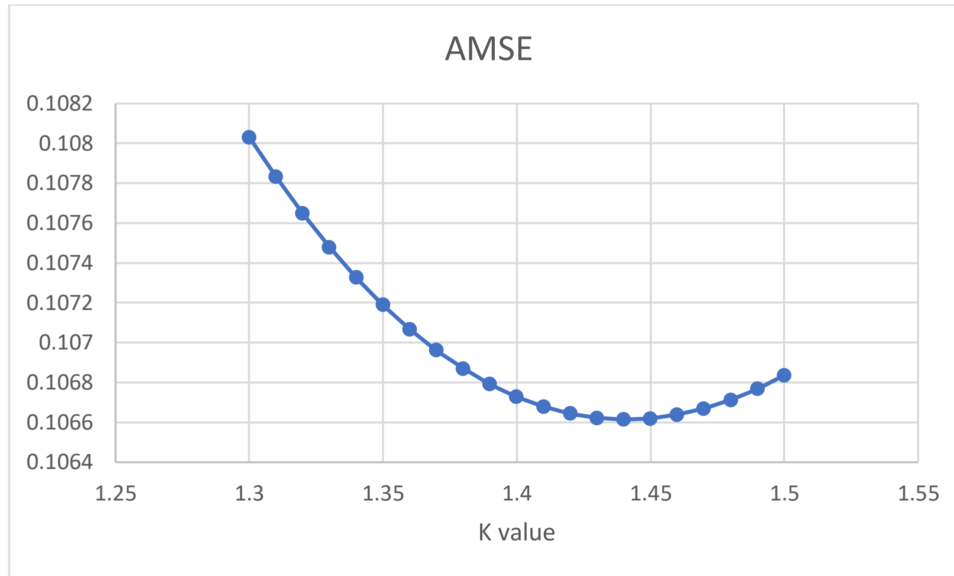

Figure 3. AMSE from different K values with increment 0.01

According to the graph, the K value for the average minimum square error value over all competitions is $K = 1.44$, accurate to two decimal places. The $AMSE$ value corresponding to this choice is 0.10661421.

I also display 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 the 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 K for the top competitors that is less than the normal K value, so the rating of the top will change less.

When $K = 1.44$, 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	Number 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 K value, call it K_2 that is less than K, for the top 7 competitors.

Assume $K = 1.44$ for the following cases. With the similar strategy, the minimum point of $K2$ lies between 0 and 1.5. With increment of 0.1 in region $[0,1.5]$, the respective $AMSE$ value is shown below:

Table 8. AMSE from different K2 values with increment 0.1

K2	AMSE
0	0.10687289
0.1	0.10650347
0.2	0.10643026
0.3	0.10640743
0.4	0.10639879
0.5	0.10639538
0.6	0.10639407
0.7	0.10639359
0.8	0.10639346
0.9	0.10639354
1	0.10639376
1.1	0.10639405
1.2	0.10639439

1.3	0.10639474
1.4	0.10639509
1.5	0.10639545

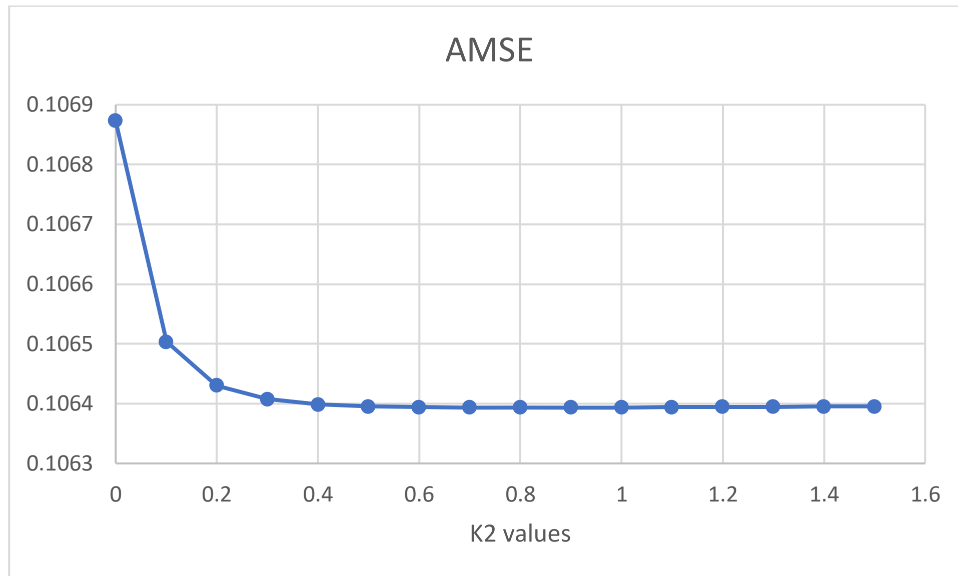

Figure 4. AMSE from different K2 values with increment 0.1

The minimum point of K_2 lies between 0.7 and 0.9. Then I repeat the process again with increments of 0.01 in region [0.7,0.9] to find the desirable K_2.

Table 9. AMSE from different K2 values with increment 0.01

K2	AMSE
0.7	0.10639359
0.71	0.10639356
0.72	0.10639354
0.73	0.10639352
0.74	0.10639351
0.75	0.1063935
0.76	0.10639348
0.77	0.10639348
0.78	0.10639347
0.79	0.10639347
0.8	0.10639346
0.81	0.10639346
0.82	0.10639347
0.83	0.10639347
0.84	0.10639348
0.85	0.10639348

0.86	0.10639349
0.87	0.1063935
0.88	0.10639352
0.89	0.10639353
0.9	0.10639354

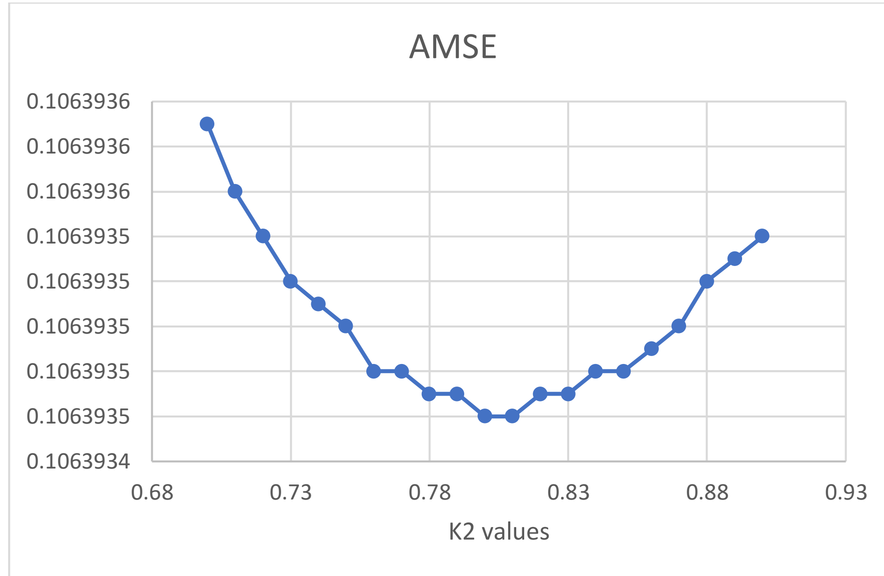

Figure 5. AMSE from different K2 values with increment 0.01

The minimum $AMSE$ value from this new approach is 0.10639346, when $K_2 = 0.8$ or 0.81. I would estimate that when $K_2 = 0.805$, the midpoint of range [0.8,0.81], $AMSE$ will be minimized.

5. Conclusion

The research of this paper developed my novel rating method based on the Elo method designed to produce a multi-competitor Elo method. Data from 20 closest Div.1 matches was used to test the predictability of my method. The multi-competitor Elo method's K value was chosen to minimize the $AMSE$ value. Another K value for the top 7 competitors improved the method's $AMSE$ value, assuring the reliability of the method.

Further improvements and future work are possible. Despite the fact that the first several competitors have a more accurate rating, there are still notable differences between the official rating and my predicted rating. Different K values for different portions of ranking could be developed. Note, such a tuning requires significant computation to choose the K value properly. The inaccuracy of my prediction is a limitation.

Questions beyond algorithm tuning are also possible. For instance, future work could explore allowing Codeforce users to determine how high of a competition rank is required to prevent

their overall rating from falling. Many competitors attend the contest, and even if they do not have a high rise in rating, they hope their overall rating does not fall. Another feature would be to provide a clever strategy of attaining a higher score in Codeforce contests. The essential strategy would integrate time allocation because the more time spent on solving a problem, the more points a contestant loses. This strategy causes some strong competitors to change the order of solving problems, such as solving the last problem at first to earn the most points. Note, the first two problems are usually simple, and there is much less penalty for solving them late. As such, many people can solve them after finishing other valuable problems. Such steps would enhance and enrich research in ranking Codeforce competitions.

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].

10. Mirzayanov, M., 2010. Codeforces contests. *Codeforces*. Available at: https://codeforces.com/blog/entry/456 [Accessed September 7, 2021].

Appendix

1. Python code for scraping the data from the closest 20 Div.1 contests (*QAQrz/Codeforces-Rating-System: Codeforces rating System (third Party implementation),* 2017)

```
1.  # coding=utf-8
2.  import json
3.  import sys
4.  import requests
5.
6.
7.  # Get official rating changes data from Codeforces API
8.  def get_rating(contest_id):
9.      url = 'http://codeforces.com/api/contest.ratingChanges?contestId={}'.format(contest
    _id)
10.     rst = requests.get(url)
11.     json_response = rst.content.decode()
12.     dict_json = json.loads(json_response)
13.     if dict_json['status'] == 'OK':
14.         return dict_json['result']
15.     else:
16.         return None
17.
18.
19. def work(contest_id):
20.     data = get_rating(contest_id)
21.     if data:
22.         with open('tests/cf_rating_official_{}.txt'.format(contest_id), 'w') as f:
23.             for d in data:
24.                 f.write('{} {} {} {}\n'.format(d['rank'], d['handle'], d['oldRating'],
    d['newRating']))
25.     else:
26.         print('error!')
27.
28.
29. if __name__ == '__main__':
30.     if len(sys.argv) < 2:
31.         print('Usage: python3 file_name.py [codeforces_contest_id]')
32.         sys.exit(1)
33.     contest_id = sys.argv[1]
34.     work(contest_id)
```

2. 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;
10.
11. struct user {
12.     double rank;
13.     string handle;
14.     int old_rating;
15.     int official_new_rating;
16.     double new_rating = 0.0;
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;
24.
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. }
38.
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. }
46.
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. }
60.
61. //Probability of player A loses / B wins
```

```
62. double Prob(double rating1, double rating2) {
63.     return 1.0 / (1.0 + pow(10, (rating1 - rating2) / 400.0));
64. }
65.
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. }
84.
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. }
92.
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.    }
115.
116.    void update(double K, double K2) {
117.        for(int i=0;i<7;i++)
118.            users[i].new_rating=users[i].old_rating+users[i].delta*K2/K;
119.        for(int i=7;i<users.size();i++)
120.            users[i].new_rating=users[i].old_rating+users[i].delta;
121.    }
122.
```

```
123.        double validate() {
124.            double tot=0.0;
125.            for(int i=0;i<users.size();i++) {
126.                for(int j=0;j<users.size();j++) {
127.                    if(users[i].rank<users[j].rank) {
128.                        tot+=pow(Prob(users[j].new_rating,users[i].new_rating)-1,2);
129.                    }
130.                    else if(users[i].rank>users[j].rank) {
131.                        tot+=pow(Prob(users[j].new_rating,users[i].new_rating)-0,2);
132.                    }
133.                    else {
134.                        tot+=pow(Prob(users[j].new_rating,users[i].new_rating)-0.5,2);
135.                    }
136.                }
137.            }
138.            char s[20];
139.            for(int i = 0; i < users.size(); i++) {
140.                if(users[i].new_rating != users[i].official_new_rating) {
141.                    sprintf(s, " [diff: %5f]", users[i].new_rating - users[i].official_n
    ew_rating);
142.                    users[i].validation = string(s);
143.                }
144.            }
145.            int amount=users.size()*(users.size()-1)/2;
146.            return tot/amount;
147.        }
148.
149.        int main() {
150.            ofstream outAMSE("tests/AMSE.csv", ios::out | ios::trunc);
151.            outAMSE << "K" << "," << "AMSE" << endl;
152.            ofstream outMSE("tests/MSE.csv", ios::out | ios::trunc);
153.            outMSE << "Index" << "," << "MSE" << endl;
154.
155.            for(double K=-1;K<=20;K+=1) {
156.                aveMSE=0;
157.                for(int i=0;i<20;i++) {
158.                    users.clear();
159.                    string contest_id = contest_ids[i];
160.                    string in_file = "tests/cf_rating_official_" + contest_id + ".txt";

161.                    ifstream in(in_file, ios::in);
162.                    while(in >> u.rank >> u.handle >> u.old_rating >> u.official_new_rat
    ing)
163.                        users.push_back(u);
164.                    in.close();
165.                    reassign_rank();
166.                    calExpRanking(K);
167.                    newWork(K);
168.                    update(K,K2);
169.                    double MinSqErr=validate();
170.
171.                    //printf("In contest id %s, MinSqErr for simulation: %.8f\n",contest
    _id.c_str(),MinSqErr);
172.                    outMSE << i+1 << "," << MinSqErr << endl;
173.
174.                    aveMSE += MinSqErr;
175.                    string out_file = "tests/cf_rating_result_" + contest_id + ".txt";
176.                    ofstream outFile(out_file, ios::out);
177.                    outFile << to_string(users);
178.                    outFile.close();
179.                }
```

```
180.                outAMSE << K << "," << aveMSE/20 << endl;
181.                printf("K=%.2f, AveMinSqErr value: %.8f\n",K,aveMSE/20);
182.            }
183.            outAMSE.close();
184.            return 0;
185.        }
```