Binary Decider Ranking Algorithm

Aidan Klobuchar

1 Introduction

One day, while watching some One Piece Youtuber, the website [http://enako.jp/cs/sort/] was mentioned, where one could rank your favorite (or who you thought were the strongest) characters through the deciding on a number of randomly presented one vs. one matchups. After the algorithm determines that the results are converged for the Top N (such as the top 25), it presents your determined ranking. This intrigued me, and I became curious to understand how this algorithm worked. Unfortunately, I couldn't find the source code or source code for anything similar online, so I decided to understand the algorithm be recreating it myself. And so I did. The result is the script `binary_decider.tex`. This script works in the same way as the aforementioned website, where a series of one vs. one matchups are presented to the user, with the choices somewhat intelligently selected, until the algorithm has converged, and the set of rankings are presented. In this little writeup, I will describe how the algorithm works in more detail and then present some speed and accuracy testing for a few different test cases.

2 Algorithm Description

The algorithm itself has a few key method and components. The most crucial, probably, is a matchup matrix where each entry represents the matchup between A and B, with a score of -1 meaning that A loses to B with 100 % certainty, a score of 0 means that there is no real information, so the matchup is a true 50/50, and a score of 1 means that A defeats B with 100 % certainty. Using this matrix, we first check to see if convergence has been achieved. If not, then two participants need to be chosen, leaning towards participants and matchups that have less certain scores. Then the matchup is presented to the user and the results need to be recorded. In order to speed things up, these results need to propagate through the matrix, so if we already know that B $>$ C and then find out that A$>$B, we should put some points in A$>$C. That is, we want some degree of transitivity. Finally, once convergence has been achieved, we need some method for ranking the participants based on the nature of the matrix.

2.1 Choosing Match Participants

Selecting the match participants is actually a pretty important aspect of the algorithm. If pure randomness is used, many repeat matchups my be presented, as could matchups with nearly certain outcomes. Thus we need to bias the participant/matchup selection. For one,

we definitely want to bias things towards matchups that are uncertain. We also want to especially weight those matchups that have no information associated with them. Finally, if we only want to rank the top N participants, we want to weight matchups/participants that are in that top N more heavily, in order to speed up convergence.

For in order to handle the primary weighting, I use a Gaussian weighting based on the values within the matchup matrix. More specifically, the first selection is made using weights based on the row sums of the absolute value of the matchup matrix, with lower values being preferred, as they represent participants who have less known about them. Mathematically, the weights are just given by $\overline{w_A} = \exp(-2 \times \sum_j m_{ij})$, where the entries of the weight vector are indexed by i and m_{ij} represents the entry in the i'th row and j'th column of the matchup matrix. The second participant is just selected using the same Gaussian weightings using the absolute values of the entries in row A; $\overline{w_B} = \exp(-2 \times m_{Aj})$.

While this gets us most of the way there, we do need to modify these weight vectors a bit. Given that we are concerned with considering the top N extra carefully, we'll consider the weighting variable $\lambda := \text{length}(\overline{w_A})/N$, that is, the inverse of the percentage of the entrants who need preferential treatment. Using this, every member of the top N is given a multiplicative boost of 4λ (this could be rejiggered to favor higher ranks more than lower ranks within the top N, but I haven't tried that out yet), first within $\overline{w_A}$ and then, once participant A has been selected, to the resulting $\overline{w_B}$. Finally, within $\overline{w_B}$, those matchups that have a weight of zero are given an extra weight of $2 * \lambda^2$, so that no information entries can still be selected from outside the top N, especially if N is less than half of the total entrants.

2.2 Recording and 'Propagating' Results

Anytime the user decides the result of a matchup, the results need to be recorded in the matchup matrix, M. Now, we don't want one matchup to be 'certain' (as errors can occur), so we will associate the results with some lower probability, in my case I use 0.8. So if entry i defeats entry j, and the matrix is empty, than m_{ij} will be set to 0.8 and m_{ji} will be set to -0.8. Now, if the matchup occurs a second time with the same result, we can't just add 0.8 to the result again; the entries need to lie within [-1.0, 1.0]! Thus we will update entries using the formula

$$m_{ij} = m_{ij} + \text{sign(res)} * ((1 - (\text{sign(res)} * m_{ij})) * p), \tag{1}$$

where p is our set probability value, 0.8 in our case, and res is the user determined result of the matchup. Now, a necessary condition of this updating scheme is that $m_{ij} = -m_{ji}$, so that we only really need to consider a triangular subset of the matrix, in order to speed up the execution of the algorithm, though the mechanics of getting that to work (which was a pain) are not important here. The idea behind this formula is to reduce the distance between the current value and the value of certainty, reducing this by the value p. So if the value was already 0.8, the gap to certainty is 0.2. Reducing this by 0.8 (0.8*0.2) gives us a value of 0.96 for a gap of 0.04. However, we want more effect going the 'wrong' way. So if the value is 0.8, but A just *lost* to B, we want the difference to be 1.8 instead (the gap

towards -1). This gives us 0.8-(1.8*0.8) = -0.64. This might be a bit unstable of a jump. but this presented no problems in my testing (described below).

Now, as mentioned in previous sections, we want to be able to propagate results to common opponents. To do this, we need to look for everyone who is predicted to defeat the winner of the current matchup and for everyone who is predicted to lose to the loser of the current matchup. The scores between the winner and the losers to the loser, as well as the scores between the loser and the winners over the winner need to be updated in some fashion. Furthermore, this process can be continued down the chain, looking at loser's to the loser's losers and so on. Of course, this can be time-consuming to compute for diminishing returns, so it's best to set a propagation limit. Now, for the actual updating formula, we will use a modified form of the general updating formula above, with modifications taking into consideration the value of the score to be updated, the value of the linked matchup, and how far along the chain we have propagated. Taking these into consideration, we end up with the formula

$$m_{ij} = m_{ij} + (1 - (m_{ij} * \text{sign}(\text{res}))) * m_{ik} * e^{-0.5*n} * p', \tag{2}$$

where m_{ij} is the matchup to be updated, m_{ik} is the linked matchup (so if j beats k who beats i, we now have information that says that j should beat i), res is the matchup result (so who wins or loses in the i vs. k matchup), n is the current propagation link (first jump, second jump, etc.) and p' are the accumulated percentages, so in this case it *should* be the absolute value of m_{jk}, as that is the parent matchup and m_{jk}*m_{ij} for any third link, though I didn't include the first part in my code. Thus the overall idea is basically the same as the base case formula, except that an exponential decay is built (with no real basis other than it seeming like a good idea) and the use of the intermediate value m_{ik}, reducing the effect of the propagation further, as we don't want it to be too noisy. Although this may seem to be too diminished, testing shows that building in even just one level of propagation drastically reduces the number of matchups needed to reach convergence.

2.3 Determining Convergence

Speaking of convergence, we need to determine hos to know when it has been achieved. The most obvious way is if all of the matchups have passed a certain accuracy/certainty threshold. Similarly, we can aim for a high average certainty, which may be more useful when dealing with a very large number of matchups, where one or two stragglers may be costing the algorithm too much time. To be precise, I considered the matrix to be converged when the absolute value of all of the matchups within in the top N are have crossed a preset goal value (0.8, 0.85, and 0.9 are used during testing). Further, if the mean of the top N matchup values are greater than 1 + the goal over two, that will count as converged. Using M_N as the top N submatrix of M and g as the goal value, satisfying either of

$$\min(|M_N|) \geq g \tag{3}$$

$$\text{or} \tag{4}$$

$$\text{mean}(|M_N|) \geq \frac{1+g}{2}, \tag{5}$$

will be considered a converged matrix.

Now, there may also be cases where the algorithm gets stuck and cannot converge for some reason. In such a case, we want to have safeguards in place to prevent an infinite presentation of matchups. My idea here is to stop the algorithm when the matrix has not changed much, in some sense, for a given length of time (I use 10 matchups). In order to prevent very premature termination, this only kicks in when no matchups or left that have zero information about them. Let $\overline{v}$ be the collection of mean table values (mean($|M_N|$)) for the previous 10 matchups and t be some preset tolerance value (0.01). Then the two possible emergency convergence conditions are

$$\max(\overline{v}) - \min(\overline{v}) < (t/N) \tag{6}$$

$$|\max(v_1) - \max(v_2)| < \left(\frac{t}{2 * N}\right), \tag{7}$$

where N are the number of entries being considered (the top N), while v_1 & v_2 are the first and second halves of the $\overline{v}$ list. The idea of the second condition is to still being able to terminate the algorithm if there is a cyclic 'bouncing' of the table values.

2.4 Ranking Entries

The final task that needs to be addressed is the ranking of the results. The easiest and most obvious way to do this is just sort by the sums of the matchups for each entrant. This would just be taking the sums for every row in M, with more positive sums indicating strength and more negative ones indicating weakness, with scores near zero being the expected average. One could consider other ranking methods, such as finding the total number of favorable, relatively neutral, and unfavorable matchups and then even weighting those matchups a bit by the row sums. Howerver, I have found that these alternative methods perform clearly worse than the more intuitive summation method, so we'll just be going with that. Finally, as a fighting videogame fan, I am always fascinated by the idea of 'tiers', where entrants that perform similarly to each other but not to other entrants would be grouped with each other. These can be generated automatically through the use of clustering methods. Here, I use a density based clustering method (HDBSCAN) to determine the number of tiers that exist within the results and then use the more standard k-means clustering to create this number of tiers (though the option to just use a user submitted number of tiers straight to the k-means method is available in the code). This makes the results a bit nicer, in my opinion.

3 Speed and Accuracy Tests

With the algorithm created and described, we now need to test it to see if it even works, and if so, if it works well. To this end, I created an 'auto-run' method, where the entrants to the 'competition' are just a range of numbers abd the larger of the two numbers wins. Thus the algorithm can be run a number of times with no external input needed. For each run, the number of steps is recorded, as well as the percentage entrants whose position was

correctly determined. However, this on its own doesn't fully capture the accuracy picture. For example, if every pair of entrants were swapped (instead of 1,2,3,4,5,6 it was 2,1,4,3,6,5) there would be zero correct predictions but the answer wouldn't be as wrong as it could be (e.g. 4,6,5,1,3,2). Thus I also computed the root mean squared error (RMSE) for each run of the algorithm, based on the differences between the expected rank and the determined rank (so 1.0 for the first example and 3.1 for the second). Running this test method 100 times each for a variety of entrant lengths, propagation limit values, and convergence goals, gives the results contained in Table 1 below. We'll first look at the number of required matchups, noting that there are 105, 595, 1225, and 2415 unique matchups for 15, 35, 50, and 70 entrants, respectively. As previously mentioned, changing the propagation limit from off (a value of 1, for only the initial matchup) to anything higher results in a marked decrease in the number of required matchups, with a speed of anywhere from 3 to 7 times. Increasing the limit further to 3 results in a diminished reduction and increasing it to 4 (from preliminary tests I did that are not shown here) has virtually no positive effect. We also see that the accuracy has a clear drop off the number of entrants and the amount of propagation allowed increase, while increasing as the convergence goal increases, with the most noticeable jump being between 0.85 and 0.90. While turning propagation off does result in a higher level of accuracy, I do not believe that it is at all worth the massive time increase. Further, we see that the RMSE rarely exceeds 1.0 and only then in the worst cases and never with a convergence goal of 0.9.

Table 1: Basic Speed and Accuracy Results

Participants	Prop. Limit	Convergence Goal	Mean Steps	Percent Right	RMSE Score
15	1	0.80	158.74	0.999	0.004
15	1	0.85	183.86	1.000	0.000
15	1	0.90	186.69	1.000	0.000
15	2	0.80	67.12	0.932	0.180
15	2	0.85	74.02	0.979	0.058
15	2	0.90	89.29	0.993	0.018
15	3	0.80	61.51	0.925	0.185
15	3	0.85	67.24	0.960	0.101
15	3	0.90	83.07	0.993	0.018
35	1	0.80	1043.66	0.990	0.039
35	1	0.85	1069.46	0.996	0.017
35	1	0.90	1073.20	0.994	0.026
35	2	0.80	234.06	0.642	0.651
35	2	0.85	248.52	0.729	0.544
35	2	0.90	320.53	0.866	0.347
35	3	0.80	208.12	0.599	0.716
35	3	0.85	224.25	0.683	0.588
35	3	0.90	289.18	0.838	0.385
50	1	0.80	2216.34	0.976	0.105
50	1	0.85	2349.17	0.980	0.081
50	1	0.90	2344.56	0.983	0.073
50	2	0.80	383.48	0.501	0.916
50	2	0.85	403.43	0.566	0.782
50	2	0.90	536.36	0.755	0.512
50	3	0.80	392.60	0.449	1.02
50	3	0.85	424.73	0.554	0.805
50	3	0.90	473.47	0.718	0.561
70	1	0.80	4386.04	0.935	0.238
70	1	0.85	4904.93	0.969	0.148
70	1	0.90	4991.43	0.972	0.127
70	2	0.80	705.74	0.375	1.227
70	2	0.85	771.63	0.461	0.965
70	2	0.90	867.85	0.621	0.695
70	3	0.80	621.84	0.334	1.323
70	3	0.85	673.56	0.442	1.021
70	3	0.90	759.55	0.602	0.725

This automated method assumes that the correct victory is chosen by the user to win the matchup 100 % of the time. In reality, however, we would expect this to never be the case. So we can test what happens to the algorithm when the expected winner of a matchup only wins some percentage of the time, in this case using 75 %, 80 %, 85 %, and 90 %. The

results of a quick test of these effects is collected in Table 2. We see no significant changes in any of the metrics (speed or accuracy) as the probabilities change across these values, indicating the robustness of this method, which is nice to see.

Table 2: Probabilistic Victory Speed & Accuracy Test, with Prop. Limit = 3 and Goal = 0.8

Participants	Probability	Mean Steps	Percent Right	RMSE Score
35	0.75	234.94	0.580	0.768
35	0.80	237.08	0.562	0.771
35	0.85	234.98	0.602	0.744
35	0.90	232.58	0.590	0.757
35	1.00	237.59	0.575	0.760

Another possible occurrence is the case of a *triangle* where A>B>C>A holds. I also decided to test this by including a triangle specification within the auto run method. Let's decide to set 15 as the triangle. Then 15 is removed from the list and replaced with 15_i, 15_j, and 15_k. When facing other members of the groups, these three are just treated as 15. However, when facing each other, i>j>k>i holds. This may cause issues not only with accuracy but with convergence. However, looking at the test results gathered in Table 3, we see that the convergence and accuracy are handled fine, with no changes in the 35 case as when compared with Table 1. The only oddity is that the no propagation case no longer has high levels of accuracy. Other than that case though, there is almost no change in the algorithms behavior, even with the addition of probabilistic victors. So far, so go.

Table 3: Speed & Accuracy Test with One Triangle Added

Participants	Prop. Lim.	Goal	Probability	Mean Steps	Percent Right	RMSE Score
25	1	0.80	0.8	429.17	0.947	0.150
25	1	0.80	1.0	431.69	0.941	0.159
25	1	0.85	0.8	431.84	0.953	0.139
25	1	0.85	1.0	428.21	0.924	0.196
25	2	0.80	0.8	160.03	0.781	0.473
25	2	0.80	1.0	159.96	0.797	0.448
25	2	0.85	0.8	173.23	0.877	0.317
25	2	0.85	1.0	171.27	0.867	0.330
25	3	0.80	0.8	145.97	0.720	0.551
25	3	0.80	1.0	144.12	0.720	0.546
25	3	0.85	0.8	153.97	0.827	0.404
25	3	0.85	1.0	154.35	0.806	0.432
35	1	0.80	0.8	708.05	0.680	0.657
35	1	0.80	1.0	704.06	0.676	0.657
35	1	0.85	0.8	701.66	0.635	0.712
35	1	0.85	1.0	702.37	0.671	0.686
35	2	0.80	0.8	254.46	0.645	0.671
35	2	0.80	1.0	252.23	0.647	0.674
35	2	0.85	0.8	267.24	0.746	0.538
35	2	0.85	1.0	270.02	0.739	0.543
35	3	0.80	0.8	224.72	0.600	0.743
35	3	0.80	1.0	222.83	0.585	0.772
35	3	0.85	0.8	238.42	0.674	0.637
35	3	0.85	1.0	241.66	0.691	0.601

Finally, we need to test how only looking for a subset of the total entrants, looking for the Top N, effects the convergence time. And I did just this and the results are collected in Table 4. The accuracy results are not reported, as they are always at perfect accuracy or very close to this mark. While doing so does result in some reduction in the number of steps needed, the result is not as large as one might expect. However, things are working properly, as I found by looking at the non-propagation results. As these take a long time to converge, I only tested this briefly. However, this resulted in the largest decrease due to only looking for the top subset of the entrants. Thus it looks like the diffusion caused by adding in propagation more or less cancels the effect of trying to narrow the scope of the search.

Table 4: Top N Speed Test with Goal = 0.9

Participants	Top	Prop. Lim.	Probability	Mean Steps
35	5	2	0.8	228.79
35	15	2	0.8	307.65
35	5	2	1.0	235.76
35	15	2	1.0	302.20
35	35	2	1.0	320.53
35	5	3	0.8	209.19
35	15	3	0.8	266.19
35	5	3	1.0	207.02
35	15	3	1.0	263.96
35	35	3	1.0	289.18
50	5	2	0.8	389.32
50	15	2	0.8	507.52
50	5	2	1.0	388.68
50	15	2	1.0	502.68
50	50	2	1.0	536.36
50	5	3	0.8	337.70
50	15	3	0.8	424.60
50	5	3	1.0	337.41
50	15	3	1.0	419.53
50	50	3	1.0	473.47
70	5	2	0.8	659.44
70	15	2	0.8	837.13
70	5	2	1.0	648.32
70	15	2	1.0	840.90
70	70	2	1.0	867.85
70	5	3	0.8	544.30
70	15	3	0.8	645.43
70	5	3	1.0	538.78
70	15	3	1.0	645.52
70	70	3	1.0	759.55

4 Conclusion

All in all, I think that algorithm that I developed and discussed here has been successful, though there is no easy or obvious way to compare it to the one used by the aforementioned website. In general, I recommend the use of a propagation limit of 3 and a convergence goal of 0.9. Of course, there are still some issues and considerations to take into account. For one, the overall accuracy, especially when considering fully ranking a large number of entrants, can be a little suspect, though such a case is not really what this method is designed for (who want to sit through hundreds of matchups?). Further, the top N speed still seems

too low when given a large number of entrants, e.g. finding the top 5 out of 70 entrants still takes over 500 matchups! An overall matchup limit could be built in to prevent this or the use of much less stringent conversion criteria (0.7 or even 0.5) could be used. Another avenue of improving the speed would be further adjustment of the matchup selections, using either different distributions or weighting novel/top N matchups even harder. Different propagation formulas or strengths are yet another avenue of possible improvement, as this is not something I have looked inot very much; this is the first formula I have tried. On the accuracy front, further adjustment of the 'worst-case' convergence conditions would likely help in the large entrant cases, as those seems to be the conditions invoked most often as the data size increases. For a completely new feature, the consideration of the number of measurements for each matchup could be used to effect how variable values are, as would be done in something like ELO or just general Bayesian statistics (countered with stronger result propagation maybe). The only real negative to this would be the need to store an entire new matrix of values, increasing the memory cost, though that will not be an issue given the number of entrants this algorithm is good for. If I do anything further with this, this Bayesian framework will be my chief port of call.