Backtabbing the World Universities Debating Championships

The program files and instructions on how to use it can be found on github.

1 The problem

In British Parliamentary debating, four teams compete in a round, and at the end of it the judges give the team that they think won 3 points, second 2 points, third 1 point, and fourth 0 points.[1] The World Universities Debating Championships ("WUDC") is held over the new year period, with regular rounds being held 29-31 December, three debates per day, and elimination rounds take place in the new year. For the first two days/six rounds, the judges announce and justify the results immediately after each round. However, for the last day/three rounds (the "silent day"/"silent rounds", respecitvely), the results are not announced after the round. The most-cited reason for this is to make the announcements of who goes through to the elimination rounds (usually scheduled for midnight, actually around 2am in the new year) more suspenseful than otherwise.

This is a nuisance. It would be nice to know the results of those rounds before the new year:

1. It could provide useful information in future rounds. For example, often you can choose between higher- and lower-risk strategies: if you are closing, you could run a highly original extension which has a 20% chance of having you come first, and an 80% of having you come fourth, or you could put the boot into the weakest team and have a 90% chance of coming third and a 10% chance of coming fourth. Then if it is round 9, knowing your round 7 and 8 results could be immensely useful: if you need three points to get to the elimination rounds, the strategies give you 20% and 0% probability, respectively, of progressing. But if you need only one point, then they give you 20% and 80% probabilities of progressing. So knowing what happened in rounds 7 and 8 can significantly increase your chances of starting the new year with a smile on your face.

2. πάντες ἄνθρωποι τοῦ εἰδέναι ὀρέγονται φύσει.

Fortunately, the results of rounds 7 and 8 are not entirely secret. This is because the tournament is Swiss-style, wherein teams are matched to teams on approximately the same number of points. So if you started the silent rounds on 14 points and in round 8 are against a team who started the day on 17, you can be reasonably confident that round 7 went well for you and bad for them. You can't be certain that you are both on 17 (i.e., you came first and they came fourth), because there are few enough teams in attendance - usually about 300 or so - that there is a non-insignificant probability that you are on 16 and they are on 17, or that you are on 17 and they are on 18. So things are a little tricky even in this rather easy case: often you are against teams in rounds 8 or 9 who started the day on approximately the same number of points as you. So often you have to look at whom your round 7 opponents are against in round 8, and even further afield, to get an idea of your previous round's performance. This practice, of figuring out round 7 and round 8 performance on the basis of subsequent draws, is known as *backtabbing*.

[1] More detail about the rules can be found here.

2 The draw generation mechanism

The generation of the draw is a complicated matter, set out here in the Constitution of the World Universities Debating Council (section 36). In broad strokes, a draw is made by sorting the teams by how many points they are on. This won't work exactly, because there are four teams per round, and the number of teams on each number of points probably won't always be divisible by 4, meaning some teams will have to debate teams on a different number of points.

The specific mechanism for forming brackets is quite complicated: the code for it is here, specifically the function `_define_rooms_anywhere` starting on line 111. It is probably easiest to understand with an example. Suppose the top end of the tournament goes 18, 17, 17, 16, 16, 16, 15, 15, 14, 14, 14, 14, 14, 14, 13, 13, 13, 13, 13, 13.... To make the top room, start with 18, but that isn't enough to make a room, so you add the 17-point teams. That still isn't enough, so you add the 16-point teams. Now there are six teams on 18, 17, and 16 points, but that is no problem. The top bracket is (18, 17, 16). So delete the first four scores from the list and start again. There are two 16 point teams at the top now, so the 15s need to be pulled up. That makes a room with no left-overs, so the next bracket is (16, 15). And so we delete those ones, and continue. Now there are six teams on 14. One approach is to make a room on 14, and delete four 14s, and continue. However that isn't the method used, somewhat annoyingly for backtabbing because it makes the job much harder. Rather, the pullups are treated like any other team in the bracket, i.e., the teams on 13 selected to debate against teams on 14 won't all be put in the same room, but will more likely be spread across the rooms of teams on 14.[2]. So the procedure is this: you see that the number of teams on 14 isn't divisible by 4. So you need to pull up two teams on 13 to make two rooms. That means the next bracket is [(14, 13), (14, 13)]. That is, this bracket is two rooms, which contain teams on either 14 or 13. And then you continue with the rest of the 13s, and so on. So the list of rooms goes [(18, 17, 16), (16, 15), (14, 13), (14, 13), ...].

In more generality, you start with the top number of points of the teams that haven't been put into a bracket yet. Call that number of points p. There are two options:

- The number of teams on p, which we will call n, is divisible by 4. Then easy! Add (p) to the list of rooms $\frac{n}{4}$ times. Delete all ps from the list, and iterate.
- n is not divisible by 4. We need to make $\lceil \frac{n}{4} \rceil$ rooms to accommodate all the teams on p, which means we need to pull up $4 - (n \bmod 4)$ teams to complete the bracket. If there are at least that many teams on $p-1$, then add $(p, p-1)$ to the list of rooms $\lceil \frac{n}{4} \rceil$ times, and delete all the ps and $4-(n \bmod 4)$ of the $p-1$s from the list, and iterate. If there aren't enough $p-1$s, add the required number of teams on $p-2$ to the bracket, and so on.

Positions also need to be taken into account, because it's unfair if a team keeps getting the same position. This is also a little complicated. Suppose the draw is being decided for round 5, and a team has done OG twice, OO once, CG once, and CO never. For each possible position, a penalty is calculated for if the team draws that position. It is calculated as follows:

1. Calculate how many times the team will have done each position after the round if they were to debate in that position this round. For example, if they were to do OG again, their allocation will be $(3, 1, 1, 0)$.
2. Divide that by the round number, to get $(0.6, 0.2, 0.2, 0)$.
3. Multiply that by log 2 of itself, and flip the sign, which gives approximately $(0.442, 0.464, 0.464, 0)$.

[2] This means that, in general, $(n, n, n, n-1)$ is a significantly more common room composition than $(n, n-1, n-1, n-1)$, although the latter is not impossible.

4. Sum over that, subtract it from 2, multiply that by the round number, and take the fourth power of that, which gives about 98.[3]

This is all implemented by making a large matrix. If there are 300 teams at the competition, one makes a 300×300 matrix, with a row per team, and a column per (room, position) combination. An entry in this matrix represents putting that team in that position, say, putting Columbia A in OO in room 5. The teams are sorted by how many points they have, and the rooms are sorted by how high-ranking they are. Each room is assigned its bracket, e.g., in the example before, the top room would only be allowed to contain teams on 16, 17, or 18 points, while the second room could only have teams on 16 or 15, and so on. This is done by taking all the teams which have points that are not allowed in that room, and putting an ∞ penalty in their cells for that room. And for teams that are allowed in that room, their position penalty is put in the relevant cell. The result will be a matrix that looks like the one on this page, albeit a little larger. Then the Munkres algorithm is run, which chooses 300 cells, to have one cell in each column and one in each row, such that the sum of the penalties of those cells is minimal, which determines the draw.

The aim of backtabbing is this: to know all the inputs of this process, given some of the inputs and the output.

3 Program specifications

The most important one is that it should give accurate results. But there are some others.

First, the backtabber shouldn't just say what is most likely - it should give probabilistic estimates. Considering the first motivation, is important from a decision-theoretic perspective. For example, if it is 51% probable that you need 3 points in the earlier example, you still optimise your chances of making it to the elimination rounds with the safe strategy of trying to get just 1 point. But also, considering the second motivation, if you're doing this for the sake of curiosity (or are trying to make money betting against other backtabbers), it is good to know just how much confidence you should place in each possibility: the backtabber's answer should be appropriately qualified.

Second, the backtabber shouldn't require too much computing power. For me, there was a limit on the amount of compute available (I didn't want to rent a server nor ask friends with better computers to do me a favour) so it would have to run on my computer, a 2021 MacBook Pro. And there are time constraints. It has to give results at least before the results are made public - it's no use having perfect results on January 2. But also, if you are using this for the purposes of, say, giving useful information for round 9 strategy, it has to give some informative answers in the narrow space between the announcement of the round 9 draw and the release of the round 9 motion (usually about 5-10 minutes), after which you can't be seen getting up to mischief on your computer or receiving messages from your friendly backtab buddies.

4 How it works

4.1 Backtabbing round 7

I will start by describing the method for how my program DESCENDER backtabs round 7, because the round 8 case has some additional complications which I will get to later. The key to the backtabber is a loss function, which maps a proposed set of round 7 results to a natural number. The loss function is made up of two kinds of loss:

[3] This process gives approximately 32 for OO and CG, and 0.02 for CO.

1. Out-of-bracket loss ("OOB loss"). The OOB loss for a given round 8 room is calculated as followed: take the maximum post-round 7 score of the teams in the room, and also the minimum. Calculate the number of teams, not in that room, whose post-round 7 score is strictly between those two scores. For example, if a round 8 room contains teams who started day three on 14, 15, 15, and 16, and a given proposed set of round 7 results assigns them 3, 1, 0, and 1 as their round 7 results, the maximum of the room is 17 and the minimum is 15, so the OOB loss for this room is the number of teams not in that room that the proposed set of round 7 results says are on 16. This loss is lower if all the teams in a room are on similar scores post-round 7.

2. Pullup loss. The number of teams on a given score being pulled up (i.e., debating against teams with more points than them) is either 0, 1, 2, or 3. So, for example, the number of teams on 15 debating against teams on 16 or more cannot be larger than 3. For a given number of points, say, 15, the pullup loss is the excess number of teams pulled up beyond what is possible, e.g., if 5 teams are pulled up the pullup loss is 2. More formally for a given score p, it is $\max(\text{num teams on } p \text{ pulled up} - 3, 0)$. The pullup loss for a proposed set of r7 results is that quantity summed over all p.

The total loss is just the sum of those two losses.

DESCENDER runs as follows. A completely random order is chosen for round 7. Then, iterating over the round 7 rooms in a random order, each of the 24 possible results for that room is tried, and the local loss is calculated. The local loss is the sum of the OOB loss for each round 8 room that that room's teams end up in plus the global pullup loss. (If two teams from a round 7 room are in the same room in round 8, that room is double counted.) The result for that room is then set to the order that has the lowest loss. This is repeated a number of times. If the global loss (i.e., sum of the OOB loss for all round 8 rooms plus the pullup loss) reaches 0, then the proposed round 7 result is saved. If the global loss does not reach 0 in 70 iterations, the process is terminated and restarted. And, to weed out options that are plainly going nowhere, if the global loss is above 25 after 20 iterations, the process is terminated and restarted.

4.2 Backtabbing round 8

For backtabbing round 8, there are two key differences.

First, there are less certain foundations. For backtabbing round 7, you know post-round 6 standings with certainty. But even the best backtabber will not be able to answer some questions about round 7. For example, two teams start round 7 on the same number of points and then hit each other again in round 8.[4] One option is to fix a given possible outcome of round 7 and assume that when backtabbing round 8, by the same method as for round 7. But that wastes a lot of time and hence is too computationally expensive, because a lot of those round 7 estimates will not permit the loss function to get anywhere close to 0. Another is not to fix them at all, and backtab two rounds at once. But that is also hopelessly slow. I settled on an intermediate option, which is to import the results of the round 7 backtabber to limit the possible outcomes that can be given to a team in round 7, and in each optimisation cycle vary those again, but only for the first five iterations, which saves on computing power, while making backtabbing round 8 a manageable job.

Second, the loss function very rarely gets to 0. This is simply because the space being searched when backtabbing round 8 is much larger than when backtabbing round 7. On the 2025 dataset, it gets to 0 about once every 500 runs, which is far too long for an impatient fellow like me to wait. To deal with that, I simply make the program print to a file if, when it terminates, the loss function is under 50. Then, when you want to make predictions, you can decide how you want the bias-variance trade-off to go: if

[4] In certain cases, one could decide between them by looking at position history, but there are at least some cases where the two are indistinguishable.

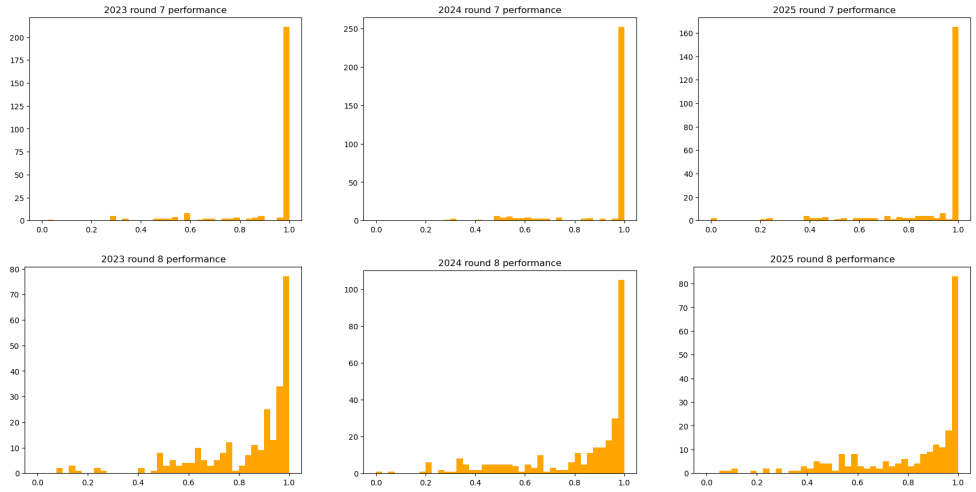
you weight data points closer to 0 more heavily, you have fewer effective data points and hence higher variance, but but with less bias per data point. So if you are running this right before the round, you might assign roughly equal weights, because you have very few data points in general so reducing variance is the number one priority, whereas if it has been running a few hours already then you can afford to be more picky.

5 Implementation and performance

I ran the backtabbing programs (or rather, versions of them that had not been tidied up yet) during WUDC 2025, and posted the receipts on pastebin. A very handy way of speeding up the program, without having to deal with any parallelism,[5] is simply to run four or five instances of the program at the same time, all of which output to the same file.[6] These generated rather robust results within minutes, and generated even more accurate results after about an hour or so. Taking the probabilities it gave to the events that actually occurred, in the last three years (excluding the zeroes it gave to certain events, see next section) the geometric mean of the probabilities it gave to the actual outcomes was as follows:[7]

	2023	2024	2025
Round 7	0.894	0.921	0.900
Round 8	0.764	0.768	0.793

And here are some histograms of the probabilities it gave to the actual outcomes, after about an hour of work:[8]



6 Other ideas

I will put these here because I think they are interesting reading, and also to prevent anyone else from trying them without due warning.

[5] I tried using a ThreadPoolExecutor to speed things up, but performance gains were much larger for the approach I settled on.

[6] The output process involves reading the file, editing it, and writing it, which happens quickly enough that they are unlikely to interrupt each other. And the harm if they do interrupt each other is minimal: you simply lose the result of one search process.

[7] I use the geometric mean because the higher the geometric mean, the higher the probability given to the actual outcome. In calculating this I ignore some cases where it gave probability 0 to an actual outcome (2x in 2025 round 7, 1x in 2024 round 8) for reasons I discuss in the next section

[8] These have different y-axes, but the important thing to notice is the shape of the histogram. Besides, each year has a different number of teams attending.

1. Other methods of backtabbing. There are a few that never had much promise, but I'll mention two interesting ones that could have legs. The first is BOOSTER. The idea here was to write a programme that simulated a few thousand WUDCs, and then train an arbitrarily strong classifier on the output. (I used gradient boosted trees, because I didn't know enough about neural nets.) For each team, the predictors were their post-round 6 score, the post-round 6 scores of their opponents in all the silent rounds, their second-hand opponents (i.e., their opponents' opponents), and in some cases those of their third-hand opponents. This didn't work though, I suspect because it doesn't adequately account for the number of other teams that have been pulled up. But this doesn't rule out the possibility of a strategy in this vein working. The second is CRAWLER. The idea here was to implement a program that backtabs, sort of, like a human: it fills in what one can definitely conclude, and then when it gets stuck it takes an educated guess. The problem here is that even with a lot of inference rules, it has to do a lot of guessing. Worst of all, for every team that has two options you lack the information to decide between, the runtime is approximately doubled. Even with ways to spot these cases and avoid them, it would give quite accurate assessments but only after 13 hours or so, and that was for round 7. With the end of December approaching and little hope of round 8 getting to work in time, I gave up on this, but again I don't think that this approach is entirely hopeless.

2. Fixing the bottom bracket problem. Above I might have seemed a bit blasé about some results happening which DESCENDER gave probability zero to, when really that should be maximally penalised. However, the specific problem applies only to the very bottom bracket.[9] Consider a case where the bottom eight teams have post-round 7 scores of 6, 6, 6, 6, 6, 4, 3, 2. This forms one bracket of two rooms, so having one room with (6, 6, 6, 2) and the other with (6, 6, 4, 3) is perfectly legal. However, this has OOB loss of 2, so the backtabber will say the actual round 7 result can't have happened. I have implemented a fix for this (basically it has a different method of checking OOB loss for the bottom bracket), however decided against it because it significantly slows the program. The point of backtabbing is to predict who will make it to the elimination rounds, and it is already known that teams in the bottom room are simply not going to make it there, so sacrificing bottom room accuracy for efficiency gains is well worth it, in my view.

3. It seemed a bit wasteful, after each iteration, to restart from a completely random point in parameter space. This was especially so for the round 8 backtab: I really wanted it to get to 0 loss, and it seemed plausible that if a previous search process ended at loss 5 you would be better off picking up where that left off. I implemented a second mode that the round 8 backtab could run in, which would start from more promising locations and optionally add some noise to try and shake it into a better basin the loss function. But that didn't deliver any results no matter how I tuned the noise, so I removed that functionality.

4. A lot of possible results for a given room seem like a bad idea. For instance, if the loss function is way lower for $(0, 1, 2, 3)$ and $(0, 1, 3, 2)$ than the other possibilities, you can probably save a lot of computing time if you stop trying, say, whether $(3, 2, 1, 0)$ is a good idea. As obvious as this seemed, implementing something that culled unpromising orders also ended up culling some of the good orders, so although each run was faster than before, it was giving out good solutions at a slower rate.[10] Also, I thought there was a risk of this biasing DESCENDER's outputs against a certain class of possible round 7 outcome, so decided against it.

[9] More precisely, it can only happen when there is a number of teams on p, and few enough teams on $p-1$ that teams on $p-2$ need to be pulled up and debate against teams on p, which only ever happens in the bottom bracket.

[10] I suspect that the problem lies in an asymmetry: if you cut out 1000 bad orders but just 1 of the correct orders, you speed up the program but might prevent it from getting to 0 loss. So extreme specificity is needed to ensure the false positive rate for the identification of bad orders is extremely low.

5. Accommodating swing teams. Sometimes there are "swing" teams in a round, that fill in when teams are missing. This is in one of two cases. First, the number of teams at the competition might not be divisible by 4, and rather than deciding some teams have to sit out a competition they flew internationally for, they instead have some spare judges debate to make up the extra teams. This case is not a problem: you have a swing team which is put in the draw just like any other team. The second case is more problematic. This is when a team, for some reason, does not turn up to the room for the debate, and a substitution has to be made at the last minute, because a debate cannot happen with only three teams. In this case, when the draw is updated, it will show a swing team there, which might be radically out of its bracket. For example, in round 9 of the 2024 WUDC, there is a debate with three teams who ended round 8 on 13 points (Auckland C, Sciences Po B, and Belgrade D) and a swing team that was on 10 points.[11] So if you are backtabbing live, you need to watch out for cases like this, which can be identified in a few ways:

   - One can download the draw immediately on release, which will only show swing teams of the first kind, and any swing teams that appear later are clearly the second sort.
   - One can check if the swing team's being the first kind of swing is mathematically impossible, given the post-round 6 standings and, if necessary, the results of the round 7 backtab.
   - One can message one of the adjudication core or the tabulation team and ask them directly.

   These teams can be handled by manually excluding them from calculations of the loss function, by editing the code in a handful of places.

6. Taking into account information about team positions. DESCENDER currently doesn't use this information. An approach that I plan to implement is to have the round 7 backtabber save to file the details of each individual zero-loss solution it finds. Then the draw algorithm is run on the assumption that that is the actual set of round 7 results, and if the loss of the optimal solution (found by the Munkres algorithm) is the same as the loss of the actual round 8 draw it really is a possible solution. I think it is quite probable that this will rule out a number of currently-accepted solutions, because on some false ones the teams getting pulled up will not be the ones that best fill the open positions in the above bracket. For example, if there are three teams on 16 which have all done their share of OO, the team on 15 getting pulled up will be one that is due to do OO. And if this successfully restricts the set of possible round 7 solutions, then this would significantly increase the accuracy of the round 8 too.

[11] I suspect the missing team was Fordham A.