AI PART B REPORT

Evaluation

We decided to represent the value of each state as a dictionary in our code, with each element corresponding to the value of the state for each player. For the sake of this report, we will refer to a state as a tuple, (Rvalue, Gvalue, Bvalue), where each entry refers to the value for the corresponding player. Our overall strategy was to mainly focus on taking and preserving pieces, and consider the distance to the end goal as a somewhat secondary goal. We also added a parameter (num_protects) to incentivise our pieces to remain close together when possible. We also considered how many of our pieces were threatened to be taken, and how many opponent pieces we were threatening. We ensured that our distance to goal was taken to be the average, else taking an opponent's piece would significantly increase the total distance to goal and hence be avoided.
If we somehow managed to take every piece on the board, we completely changed the strategy of our agent to simply make any exit move if possible, and otherwise make any move that reduces the total distance to goal. This way we would always win from this state, and had no risk of timing out since the move can be made instantly.

For each player we simply did a direct calculation on the weighted sum of each of these parameters (see *weights.py* for weighting). However theses parameters only considered how good the arrangement of one's own pieces were, and did not consider the value of the opponent's position. It is clear that in order to have a well considered evaluation function, it is a player's *relative* position that needs to be considered (i.e. how good one's position is compared to how good the opponents' position is). It is no use selecting moves that will bring you closer to your goal, if in doing so your opponents end up even closer to their goal than you are. By only considering the raw values generated just by the parameters, it is possible to have a 3 tuple (10,10,10), which would be chosen by player 1 over (2,1,1) (since 10 > 2) when clearly, the latter gives player 1 the lead. In order to consider each player's relative position, we used the formula EVALx = RAWx - 0.5(RAWy + RAWz) to calculate the value of a position for player x, where RAWa represents the value given by the sum of the weighted parameters for player a. The result of applying this formula means the 3 tuple will now sum to zero. This removes the issue above, as it is the relative value of a player's position which is calculated. This was a good basis upon which to build our MaxN algorithm, where each player needs to only select their choice based on the value in the corresponding position.

After a bit more thought, we noticed that for a given EVAL for a given player, the actual value of the state can still vary, depending on the EVAL for the other players. For example, if there were two options to choose from (2, 5, -7) and (2, -1, -1) (noting that both tuples sum to 1, since relative position is now considered), player 1 is far better off choosing the latter state, since they are clearly winning, whereas in the first state, although they are a long way ahead of player 3, they are losing significantly to player 2. We wanted to ensure our AI didn't get caught up maximising our value just by destroying one player and letting the other win. We never wanted to play for second place, so we wanted our evaluation state to weight the leading opponent more heavily than the trailing opponent. So, we updated the formula to EVALx = RAWx - 0.45*min(RAWy, RAWz) - 0.65*max(RAWy, RAWz). This does mean that

the evaluation state will no longer sum to zero, however we deemed this appropriate for the following reason. Consider the two states with RAW values (60, 60, -90) and (60, 60, -180). From the perspective of player 1 and 2, the the second state is only ever so slightly better for them, because they are still just as much of a threat to each other, and player 3 is still very unlikely to beat either of them. However, for player 3, the second state is much worse, since they need to make up a lot more ground to have any chance of winning. This means the the sum of the value of the state will decrease since the change in player 3 is significant, but the change in the first 2 players is insignificant.

Strategic weighting

We decided that at different points of the game, certain features became more or less important, depending on the state. For example, if we only have 1 piece left (and less than 3 pieces exited), the average distance to the goal is no longer as important as capturing new pieces, since we cannot win before we have done so. Likewise, if we have far more than enough pieces to win, we have changed the weighting so that our AI prioritises getting to the end goal slightly more, and taking more pieces slightly less.

Different Implementations

We considered two major techniques for implementing our AI's strategy. We initially went for maxN, which works just like minimax, where we assume each player wishes to maximise their own value of the state (which consequently minimises their opponents). We kept track of who's turn it was and slightly changed the weighting of each players evaluation function accordingly. If it was your turn, then we considered being in a position where you were able to take pieces as more important than avoiding positions you were able to be taken (since if its our turn, we get to do the taking!), and prioritised these factors in the opposite way for player's whose turn it wasn't.

maxN seemed to work pretty well, however when we compared our maxN algorithm (which we named Dizzy) to a "greedy" algorithm (essentially maxN with depth 1, which we named Kanye), the greedy algorithm consistently performed better. We suspected that we were unable to get Dizzy to consider a reasonable depth without being far too slow. It's likely that only considering a depth of a 2 or 3 meant that its pessimism when considering opponents moves made it choose safer options, which lacked payoff. We tried many techniques to improve the efficiency of Dizzy, with a lot of effort going into making transposition tables and memoization, but with no success.

State class

We used a state class to keep track of the state, and also the dictionary which contains the evaluation of that state for each player. When a move was made, instead of recalculating the entire state, we simply calculated the *change* in the evaluation, so we could simply add or subtract the corresponding amount, increasing efficiency.

Overall effectiveness

In the end, we were unable to reach the optimisation of our AI, it is likely that we overlooked something important when implementing maxN, which meant we were unable to utilise anything much better than a greedy algorithm.