Description of our program

Core algorithm: minimax and alpha-beta pruning.

Our program will generate all the possible states for the next move, and choose the overall best way by running the minimax algorithm, The alpha-beta pruning will prune a branch if one of its sub-states has a lower utility score than the current overall best.

There are two reasons why our group chose minimax.1. We have access to state information, therefore we can write an evaluation function easily. 2. Since there are lots of non-meaningful movements, we can apply a-b pruning or cut-off to reduce the computational complexity.

Handling simultaneous-play

Our program updates both actions from lower and upper then evaluate the utility score for each of them in the same state, rather than evaluating on the separation movement, because the MAX will never lose a token because it's 1 step forward when the evaluation process happens.

The main idea is that we want to achieve a higher as possible utility score than the opponent. That is (MAX-utility-score) - (MIN-utility-score).

Feature selection:

1.Weight of each kind of token

The weight is calculated as (no opponent Beat-what token + (1/3) * opponent remaining throw)/ (opponent remaining throw + opponent tokens on board)

-We want to have a evaluation that how important a type token is(eg: the value of "p" is zero if the opponent has neither throw option or rock on board)

2.Remaining throw

3.Closest distance from a token to its opponent, calculating as max(summation of 1/(distance to beat-what)

4.Reaming tokens = tokens on board + remaining throw

5.Number of invincible tokens = token that has no opponent

-as a tiebreaker when two possible beats can happen, prioritise to beat the only left What-beat,if there is one.

Strategy & motivation

Strategy1: Keep the ratio of each kind of token as close as its weight(as described in feature 1).
Motivation1: if the opponent throws more specific kinds of tokens on board, we need more what-beat tokens against him.

Strategy2: Keep the heaviest weight type of token alive as possible
Motivation2: Having the heaviest weight means having more beat-what. Applying this strategy may avoid having invincible tokens.

Strategy3: The closer beat-what distance and loss in opponent tokens have a higher utility score.
Motivation3: We want to force our token to move to its Beat-What and eventually kill it.

Strategy4: Define the distance function as described in feature3.
Motivation4: If a token is moving toward a single What-Beat, it has a higher chance to run away. Whereas a token has more chances to beat an opponent token while moving towards a group of What-beat.

Strategy5: Killing rather than escaping.
Motivation5: If the escaping coefficient is too high, a token might run away even if it can kill the other token in the next move. However, in most cases, it's no emergence for a token to beat a token first then run away.(*this is achieved by setting a higher weight for distance to BEAT-WHAT than WHAT-BEAT)

Strategy6: When having the less remaining throw, moving the tokens has a higher priority.
Motivation6: Less the remaining throws there are, more valuable they will be. Since any kind of token can be projected onto more and more places, the opponent will get threatened more than ever when you have less to throw, this can influence their decision making.Furthermore, a strong projection is required to handle some edge cases(like you are running out a specific kind of token, or the next possible move of your opponent is highly deadly) and these projections are not expected to be wasted.

Machine learning(source code: state.py, at the bottom)
Learning set generation
Around 1000 of states were created and manually assigned utility scores, say based on our intuition. Those information were stored in multiple json files.. Then we write a tiny function to read in those states & their utility scores from the json files and fit the date into a linear regression model imported from sklearn. Once the linear model is fitted, a coefficient matrix can be simply gained from just calling a function (the gradient descent can be applied automatically). **The coefficient matrix was recorded for the prediction process.**

Prediction process(evaluation)
Once we need to evaluate a state, the utility score of an arbitrary state can be obtained from the product of the coefficient matrix and the features of this particular state.

However; this technique is eventually aborted
There are two reasons that a linear model may not be sufficient enough. 1. an assigned utility score, mainly based on human intuition, might not inflect how good a state is precisely. For example an ordinary person might evaluate a chess state as 100 percent winnable whereas a chess professor may figure this state is almost lost in just several moves. 2. It's so hard to modify the coefficient of each feature(we need to re-create 1000 training sets). We assigned a weight to each of features instead and adjust them according to its winning rate against the others(manual gradient descent).

Performance evaluation

For the convenience, We define *OT = Time complexity , OS = Space complexity, O(constant) is in O(1), constance*O(n) is in O(n)
Evaluation

1. Iterate through the entire board, to find out what tokens do each player has and storing them in a dict.OT(size of board) ,OS(total tokens)
2. Iterate through all the (player tokens) x (opponent tokens) to calculates features
    A. Calculate how many invincible tokens for each side OS(1)
    B. Calculating the distance among a token and its beat-what OS(1)
    C. calculate the distance among a token and its what-beat OS(1)

    Complexity above is OT (3*(nb upper tokens * nb lower tokens))
3. Check the remaining throws of each side from a dict. OS(2),OT(2)

Updating states and reverting states

Updating

1. Applying two actions.OT(2),OS(2)
2. Check the battle result, wrose case is 16(same type token in a grid)+1(upper movesto there) + 1(lower moves to there) = OT(18),OS(18)

Revert

1. Undo 2 actions = OT(2),OS(2)
2. Putting all the betean tokens back to the list, worst case, as described as updating2, = O(18),OS(18)

Minimax algorithm

Expansion

1.Finding all possible movement for upper

a) traverse the entire board once, and find return all the grids that are next to an existing token. OT(Size of Board + 6 * nb of tokens),OS(6*nb of tokens)

b) traverse the entire board to find the throw zone.OS(Size of board) =OT(Size of board)

2.Finding all possible movement for lower = same as the upper

3.Do the cross product of upper and lower movement OS(nb of possible state for upper *nb of possible state of lower) OT = OS(self)

3.Update, evaluate, and then revert(All the complexities are discussed above)

Alpha-beta pruning:OS(2),OT(1) and it reduces the overall OS and OT significantly with respect to this particular game. Even its

Overall performance

Minimax algorithm

If we consider the size of board is a constant term, the complexity of our minimax algorithm will be nominated by the cross product of upper and lower movement and its depth, which is exactly the same as the theoretical time complexity OT(possible movement^depth we look at). The space complexity will be dominated by the number of movement and depth, and eventually converge to the theoretical OS(possible movement * depth we look at).

Pruning

The best case is OT(2b-1) where the first MAX movement generated is the best and all the first utility scores of every other MIN movements are lower than MAX-best.

The worst case is examining all the childrens and leads to OT(b^2)
The average case is OT(b^(d/2)) and this value is very close to our a-b pruning performance

performance (done by a automatic- testing pipeline)

```
** game over!
* winner: upper
win rate=0.96  lose rate =0.0  out of100
```

Other aspects

1.Efficiency optimisation.

Our group found that the deepcopy version(creating a snapshot for each possible state and storing them in the memory for further analysis) provided in the Game utility takes a lot of time. Specifically, deepcopy takes 90% of the excursion time since it needs to iterate through the entire state, including board and throw, to make a copy, and lots of memory space is required to store those snapshots.

Our group invented a “backtracking” technique to cope with the issue above. An on-the-fly state is actually generated and all the movements and beatens of a turn are stored. As long as the on-the-fly state is evaluated, movements information is used to revert the board back to the previous state.This algorithm reduces the time complexity from O(entire board + throw) back to the O(2), which is the two movement of each colour, plus O(defeated token), wrose case at O(17), which happens when a side used all of its throw to put a same type token into the same place and the other side kill all of them once. The space complexity is reduced from (branching factor)*O(entire game) to O(actions + throw) which is the same as the time complexity.

In terms of actual practices, the deepcopy version takes 10 seconds to beat a random player whereas the backtracking version takes 0.3s.(Without changing the evaluation function and any other stuff)

2.Enhanced algorithms : Weaken the opponent

The minimax is imperfect in the parosci360 circumstances. The classic minimax algorithm assumes the sense that the opponent can always reach a state that minimizes the players’ utility; However, this property ignores the existence of throw action, which provides a branch of variety and uncertainty. In the other words, the ordinary minimax assumes our opponent can look at the future and can steadily use that highly uncertain information to beat back.

For example, if a "P" is just adjacent to a "r", which is in the throw zone of the lower.
The logic of the ordinary Minimax(Max):
If MAX moves "P" to the "r" , the best way of Min to beat back is throwing a "s" directly onto Max's "P", and MAX will earn no point from this movement since both of them lose a token.

The sensible logic of an actual player(Min):

Since Max also has lots of token on board with lots of meaningful moves, the chance of MAX to move the "P" to my "r" directly is low; Therefore, Min will never throw a "s" onto its "r" because its no different from committing suicide.

The weaken-opponent technique
we weaken the power of the Min to "predict the feature". The implementation is just simply using a lower feature coefficient for Min than Max.

Performance outcome
This weakening algorithm indeed improves the performance when playing with the dummy greedy, we found our move is somehow more aggressive than the ordinary MINIMAX and the winning rate is indeed improved.

3.Enhanced algorithms:"heuristic deepening minimax" (source code: state.py, at the bottom)

We create a "heuristic deepening minimax" algorithm to improve the performance for our minimax. In the very beginning of the game, with lots of remaining throws, the branching factor is quite high and eventually leads to minions of possible state were generated, even only with 2-ply minimax. In contrast, at the few states before reaching the termination state, the branching factor reduced significantly, and even if we look at more steps forward, say 5-ply, the nodes generated is still far more less than 2-ply evaluation at the start.

Our heuristic deepening minimax algorithm can look at the branching factor first then decide how many states it looks forward to. This algorithm may enhance the winning rate a lot since we can flexibly focus on the last few steps, which may determinate the result significantly, without wasting our computational power.

The heuristic depends on the remaining throw and remaining token of each side, fewer (remaining throw + remaining token) will advise our algorithm to look deeper.

*However, this algorithm was commented out and not being put into practice since the 100MB limitation of a game.

Supporting work

1. **We implemented a local random player (highly random) to be a dummy (randy.py)**
2. **We Imported OS to play with a dummy random player automatically, and it will print the result (win rate, lose rate, draw rate, nb of games.**
   (in the outtest folder, named as autotesting.py)
   !!!!! The Game.py should be modified as the version that can print the results to a log. Autotesting will use the information from that log)