ProjectB Report

In general

In this assignment, we choose to apply the min-max algorithm to choose the best movement with alpha-beta pruning. However, simply using alpha-beta pruning is not strong enough to have an efficient search algorithm. So, we try to make various optimization and apply strict time control methods for the program. Finally, we get a program that has a good balance between time-cost and 'choose-accuracy'.

The search algorithm

In this game, we choose to use min-max with alpha-beta pruning as a search algorithm.
To apply the abstract search algorithm into the game, we use the current "boardgame" (contains all own tokens and opponent tokens) as root. For every child node, it contains the movement information and the corresponding new "boardgame". As a result, we can evaluate every node at the bottom and choose the best movement according to the min-max algorithm.
The min-max algorithm is powerful in our game because it can forecast our opponent's movement and choose the relative best choice for us. However, the disadvantage is obvious: even we have already used alpha-beta pruning to optimize it, we may still have a huge tree and must spend mins for only one step if we do not make further optimization.

Optimization

The size of the tree depends on three elements: maxdepth, the number of the possible movements, the order of the movements. According to this, we make some modifications to reduce the size of the tree.

1. As we know, the deeper the tree is, the better choice we can make. However, adding one depth may lend to huge time cost growth. In my program, if I choose maxdepth=3, it will end the game in around 10s. But if it turns to 4, it may beyond 1min in some battles. Finally, I choose to use different maxdepth depends on different states. I choose maxdepth from **3 to 7**, which depends on the total number of left tokens and the time our program already used. In short, I will increase the depth when the left tokens' number decreases. And if the time counts over 30s, we start to decrease the depth. When it over 40s, we make maxdepth=3. This method offers a good balance between time cost and choose-accuracy in the real battle. We can end most battles within 30s, very little for 41s with over 50% winning rate.
2. For a father node, the number of child nodes depends on two things: the number of tokens that may be moved and token's possible movement.

To token number: In most cases, we will not move corner tokens if we have tokens that around opponent tokens. So, it is meaningless to consider them as we know they will not be chosen to move by the search function. However, putting them into consideration may highly increase the time-cost. As a result, for every father node, I will consider two states' tokens: the most valuable token to attack and the most dangerous token to escape. By doing this, we will pick around 2-5 tokens into consideration, which may highly decrease the time cost than put all tokens into consideration.

To possible movement: It is hard to decide whether the 'move' action is meaningful before we use min-max. However, we can know whether the 'boom' action is meaningful. My program will only consider putting 'boom' action into consideration if it will remove more opponent tokens than own tokens (except our side have a huge advantage and we want to end the game). What's more, boom adjacent tokens leads to the same result. We also guarantee that for every group of tokens, only one token in them will be chosen for boom action.

However, this optimization is based on our assumption that "this may not be chosen". We cannot guarantee that our chosen tokens are absolutely the best tokens that should be moved. But, as this optimization can a bring huge improvement in time-cost and only hurt us in some rare cases, I still choose to use this to improve our program.

3. According to our evaluation, the boom action tends to have a higher evaluation value than the move action. What's more, I find out that the stack move tends to have higher evaluation value than single token move. So, for every father node, I make its child mode in the order of boom-movestack-movesingle. By doing this, we can maximize the pruning of subtrees.
4. Outside the min-max algorithm, we also make a small improvement. From the beginning, it is meaningless to use min-max to find out the movement. So, We choose to give movement for the beginning steps directly. And if we use min-max, it may cost some time because every token may be valuable.

In summary, we improve the search algorithm from many sides. Some of them may reduce the "intelligence" of the search algorithm. But with this cost, we highly reduce the time-cost. It is more like to try to find a balance between time-cost and intelligence.

Evaluation

The evaluation function costs most of my time for the assignment. I tried over 4 versions of evolution function and want to find out the best one. Finally, I figured out that **my optimization of min-max function limits the performance of my evaluation function. And there is hard to find out a 'best' evaluation function.**

General idea for evaluation

The evaluation function has two parts: One for boom, one for normal movement.

1. Boom part: If the movement is 'BOOM', then we will use this part for evaluation. It is easy as we only need to calculate the token number's change from our side and opponent side. However, we must deal with the special cases that returns a '0' boom evaluation value, which means we will lose the same number of tokens with opponent. In this case, we may change the value depends on the current state. If we have enough advantage and want to end this game by one-for-one, then this boom action is reasonable. So, we will give a new evaluation number for it, which is larger than 0. If the state is equal for both sides, we will leave it for 0, which means not bad or good. And if we are at the position of the weak side, we will give a negative value for this case, which means one-for-one is not a good choice for us.
2. Movement part: this part is for the normal movement. In this part, we will make a combination list in the form of [(goal, token, distance, goal value)]. The distance shows the shortest step for the token to reach the goal point in the present board. Rather than just calculate the distance by Manhattan Distance, we add the stack into calculation, which increases the accuracy of estimation. The goal value is a boom evaluation if we boom at that goal point. Then, we will pick two elements from the list. One is the best combination for attack, one is the most dangerous combination for escape. By dividing their value by distance and add the result, we can have an estimate of the board. As every movement in the board may change the whole combination list, picking the best one from each side is better than the average value for all elements.

Detailed idea for evaluation

1. Move order problem: As our evaluation function is making evaluation for the newest board, which means the Boom-Move and the Move-Boom may have **the same evaluation result** in evaluation (same newest board). And if we choose to move first. For the next step, it may find a new 'best movement' as the tree goes deeper than before. This may bring some problems. As our min-max tree is not unabridged (we only use part of the min-max because of optimization) and our opponent may have a deeper tree than us. We may not get the 'wanted benefit' from moving and lose the boom chance. So, in our evaluation function, I choose to give a higher evaluation value for boom first.
2. Expectation and reality problem: As our min-max has max depth, all of our evaluations are based on the expectation rather than the reality. So, we cannot really know whether we can realize an expectation or not. And it brings a problem. If we find out that we can choose to boom and get a reward of 2 immediately and may can choose to move and get a reward of 6 after 5 rounds, which one should we choose? After trying many times, I find out that it is hard to say which choice is better. It is more like to find a balance between expectation and reality. Finally, in my evaluation function, I choose to pay more attention to immediately boom benefits. As when we meet a powerful opponent, it is hard to get the expectation, getting the immediate reward is safer and practical. However, if

my program finds out that we may boom 3 more in the next movement if we choose to move first, it may choose to make an adventure.

3. In the evaluation function, we also consider the stack element on the board. As stack means higher motility, if we have more stack, we may easier for attacking or escaping in the later game. However, I do not think that stack is a determinate element in the game. So, I only give a small influence for stack in the evaluation function
4. End game boom: When our game is going to end and we have a huge advantage, it may be good for us to choose to remove 1 opponent token by 2 own tokens. However, our normal boom evaluation will never choose it because it is not worth for a single movement (x+1 for 1). So, in this case, we have a new evaluation that cares more about the number of opponent's tokens.

Finding

1. Our min-max search algorithm can only choose a 'best choice' under our own evaluation function. 'Too smart' evaluation function may have a bad performance.
   We predict our opponent's movement by our evaluation and make decision based on it. However, in most cases (in all cases actually), our opponent will not move like our prediction as they have their own evaluation function. And, the performance of our evaluation function will also depend on the opponent's evaluation function. It confuses me for a long time when I try to write a 'smart' evaluation function.
   For example, if I find out that in the next step, my opponent can boom away 3 tokens of mine, then a smart choice is not caring about them anymore if we cannot save them in one step. However, our opponent may not choose to boom in next step. He may choose to boom in many steps later. And for us, we lose the chance to escape. But, if our opponent chooses to boom as we predict, we may save one step from meaningless escape.
   This problem cannot be solved as we can never really know how our opponent makes movements. After I find out this, The direction for evaluation function optimization changes. Before, I try to get an evaluation function that can make every action smart. Later, I find out that the real meaning of evolution function is to find a movement that leads to a game that 'if you can boom three tokens from me in your turn, I will boom at least 3 from you next step'.

A funny play strategy

I have also tried a different movement strategy. In the beginning, I keep following what my opponent did. If I move first, I will move a corner token and then follow my opponent's movement. In this case, our token will meet opponent's token with the same token number.
Then, our opponent may have two choices, he may make an n-for-n boom or keep moving. If he boomed our following tokens, it turns to us move first. Then we will choose to move a not important token again and keep following the opponent. In some states, maybe we find he moved a token which we already moved or both of us already lost most of our tokens, we

start to use min-max to decide our movement.
The benefit of this strategy is that we can force our game in an equal state. When the opponent has 12 tokens, it is dangerous for us if he has a better evaluation. But when he just has 5 tokens, his risk to us may be lower than before. What's more, as follow do not need to use min-max, we cost very little time to calculate while our opponent has already do not have much time. It means we can have a high max depth tree because in this time, we already do not have much tokens on the board. It gives us a huge advantage and increasing the win rate. We may get a dead heat, but hard to lose the game.
However, I do not choose this because it is more like a trick rather than an AI.

Summary

For our program, I make lots of optimization for the min-max search algorithm to try to get a balance between time-cost and choose-accuracy. And I keep improving the evaluation from real battle with others. However, as I mentioned before, the performance of the evaluation function also depends on our opponent. I am not very sure whether my chosen evaluation function is the best one for test opponents. What really surprises me is that my program is stronger than me. When it battles with other's AI, it sometimes makes some movement that I cannot understand. After I spend various time to try to debug, I find out that the code does not go wrong, it just makes the best movement while I have not figure out. Sometimes it will even give the opponent an 'expectation' to let them do not boom immediately and then save own tokens. It is really amazing and interesting. Although sometimes it will lose, it can win me definitely:).