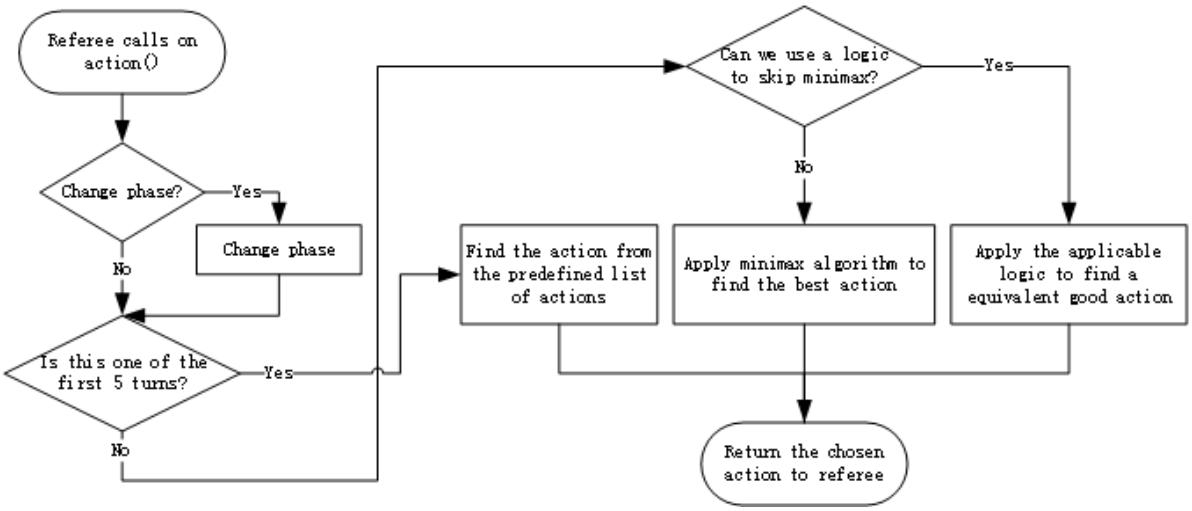
RoPaSci360 part b report

By Yiyang Huang and Siyang Qiu (team FiveYang)

Approach description

Overview

Our team implemented a RoPaSci360 AI agent based on a modified version of minimax algorithm. The algorithm would calculate the expected utility value for each of the possible actions it can take and choose the best action for the corresponding player. The efficiency of the algorithm is improved using alpha-beta pruning and depth limitation techniques. After reaching the depth limitation, the current state of the board is evaluated using an evaluation function, described later in this report. To further decrease the total time cost and perform more intuitive smart moves (to humans) we added 5 critical state logics. The algorithm will check every turn if the current state meets the enter condition to any of those critical state logics. If any of the conditions were met, the algorithm will perform the critical state logic and skip running minimax algorithm for that turn.



Minimax algorithm

Minimax algorithm is optimal for deterministic, perfect-information games. But RoPaSci360 is a simultaneous-play board game, which means although both players share perfect information of the game, but non-deterministic comes into play since each player isn't sure the state of the board after performing certain action.

In our minimax algorithm, we assumed that we make the move first and the opponent will react to our move. To be more specific, when performing a minimax search, we first find out all actions we can perform, and pass it to the min_value function which will perform all potential opponent actions and make an update. When the depth limit has reached, evaluation in the max node will be called and the lowest evaluation is returned. Since our

agent assumes that the opponent will pick an action to counter our action which cause a lower evaluation. Hence our agent will make a move such that will cause the least harm.

In order to reduce the time complexity of minimax algorithm, we applied alpha-beta pruning to reduce the number of nodes visited while preserving the “optimality” of our minimax algorithm. In practice, the first max function has actions which we will be performing next turn, so when an action which has higher min_value than the current alpha will be used as the next move.

Our depth limit of the search is dynamically adjusted based on the number of nodes visited already. According to our measurement in online battleground, about 8,000 nodes can be visited per second. Our algorithm would reduce the depth limit from 2 to 1 when exceeding 200,000 nodes visited in total or more than 4 tokens on board to ensure that the runtime of our agent does not exceed 60 seconds.

Start-game sequence

A fixed start-game sequence provides the first 5 actions we perform, including throwing 3 different types of tokens then moving them forward for 2 turns. This saves us the time to search in the early stage of the game since at that time, no tokens can be killed and so search isn’t necessary.

Critical state logic

On each turn before we perform the minimax algorithm, entering condition for a set of critical state logics are checked. In case the current state satisfies the entering condition for a logic, the action for the current turn would be determined by the logic instead of the minimax algorithm. There are currently five critical state logics implemented in our AI agent.

1. Take down an opponent token, if possible, when all player tokens are safe. Take down action in this logic is only done through slide/swing action of a player token.
2. Throw on an opponent token, in attempt to kill it, if there are more than or equal to 3 opponent tokens in our throw zone. This guarantees at least 2/3 chance of killing an opponent token. If we missed, the newly thrown player token will be right next to the targeted token and can kill that token again next turn through slide/swing actions of logic 1.
3. Hide on the same tile with an opponent token of the same type, if possible, when the token is in danger of being eaten via slide / swing action of an opponent token. This allows the token to be safe unless the opponent wants to lose one of their tokens of the same type as ours.
4. Run away to a random tile that will not get the token killed if it is trapped at a corner, given it is closely chased by an opponent token that can kill it, and the token is the only one we currently have on the board.
5. Trade with an opponent token, if possible, when both players can take down a token of the other side through a slide/swing action. This way, we are making sure that if we are to lose a token, the other side is losing one as well at the same time.

These five logics are what we believe more like what humans would perform given the critical

states that satisfy the logics' entering conditions. Although the logics have a small chance to lead us to a slightly worse state than what minimax algorithm will return but have a great chance to bring us advantage in the long run by acting more like a human than a computer that looks ahead only a few turns.

Repeated state checking

We implemented the repeated state checking functionality to prevent our game drawing after a particular state occurring 3 times. Since we do not know what action the opponent agent will make, and we only control our part of the board, we need to be more conservative when checking the repeated state.

We used a dictionary to store the information about states and count. In this dictionary, the key is a tuple representation of our part of the board (player configuration) and the value correspond to a tuple with 2 elements:

1. The maximum count number an opponent configuration has occurred with current player configuration.
2. A dictionary mapping each opponent configuration to its count.

With this dictionary, before an action is returned, we can perform the following check:

- Use the player configuration (the part of the board we control) after making this action as the key.
- Find the maximum count for which an opponent configuration has occurred with this particular player configuration. If the count is 2, meaning that there is a risk of it increase to 3 if we choose to do that action, so we shall pick a different action.

After a throw action is performed by either us or the opponent, we reset the dictionary.

Board data structure

We implemented a Board class to hold the information on a game board, a board object is not aware of the color our agent is and this increased the reusability of functions in board. Also, by implementing the Board data structure, we can clone a board and perform some "unreal updates" from minimax algorithm and get the evaluation of this modified board without having to change the real game board.

Successor

The successor function in Board requires an argument which specify the color of which is making the action. Then the successor will return a list of valid actions for the given color. We use it to find our potential actions in the max_value function and the opponent's potential actions in the min_value function. In our successor function we only have included a fraction of all the possible actions, to be more specific, we have included the following:

1. All slide, swing actions possible
2. For each opponent token inside the area reachable by throw, a throw action with a type wining the opponent token will be added. (Opponent here is relative to the color we give in the successor function)
3. Another 3 throw actions with different type but to a specific location relate to current number of throws left. This is to prevent the situation which we do not have any other

actions and both agents stuck in an infinite loop when opponent refuse to get into our throw zone. (Throw zone meaning the area reachable by our throw)

Evaluation function

The evaluation function is called by the minimax algorithm after expanding all necessary branches in the first two turns to return the estimated utility value of the state after performing each set of actions. To best estimate the state to the respectively real utility value, four evaluation features are used to quantify the given state, listed in order of their relative priority.

1. **Number of our tokens defeated.** This is the total number of our tokens dead.
2. **Number of opponent tokens defeated.** This is the total number of opponent tokens dead.
3. **Difference in number of throws left for each player.** This is the number of tokens throws left for us minus the number of throws left for the opponent.
4. **Difference in number of tokens in respective opponent's throw zone.** This is the number of opponent tokens in our throw zone minus the number of our tokens in the opponent throw zone.
5. **Balance of distance to meat and distance to danger for all player tokens.** This is the total distance from a token of ours to all the opponent tokens it can eat, minus the total distance from that token to all opponent tokens that can eat that token.

There are also 2 penalty features that affect the evaluation result in a non-linear way. They are Boolean features that will not affect the evaluation normally but will greatly penalize the utility value of a state if that state matches an undesirable condition as described below. These penalty features exist to avoid entering specific types of paths that will cause us to lose.

1. If our current tokens cannot eat any opponent tokens on the board, give a small penalty. In the normal phase, the penalty is set to the same as the cost for using a throw action. Hence, a good solution to get rid of this penalty would be to throw a useful token of another type, with no additional cost for the throw, but increased utility value because the new token can start to threaten opponent tokens via feature 4.
2. If the opponent has an invincible token, apply a very large penalty. This is a state that we wish to avoid until the last resort, because entering this state would mean that we will not be able to win any more. The best outcome for us will be a draw.

But before calculating the estimated utility value for each state, terminate state conditions are checked to make sure this state is not a terminate state. If the state is a terminate state, and we won the game, the evaluation function would simply return a positive infinity (100 million is used). If we lost the game, the evaluation function would return a negative infinity (negative 100 million is used). If the game is draw, 0 is returned by the evaluation function.

We have three sets of weight combinations that copes with three different phases during the game.

1. The first set is for the normal phase. This set is used throughout most of the game

process, in attempt to bring us consistent advantage with each move performed, by careful balancing between defense and attack. This phase can be thought of as a conservative attack phase. The specific weights are described below.

a. Number of our tokens defeated: 10000.
b. Number of opponent tokens defeated: 10000.
c. Difference in number of throws left: 5000.
d. Difference in number of tokens in opponent throw zone: 990.
e. Balance between distance to meat and distance to danger: 1.
f. Can't defeat any opponent penalty: 5000.
g. Opponent invincible token penalty: 1000000.

2. The second set is the survival phase. This set is used when the opponent has an invincible token. In this case, the best outcome we would get will be a draw. Therefore, we would play very conservatively, by trying our best to escape until the 360th turn. The specific weights are described below.
    a. Number of our tokens defeated: 1000000.
    b. Number of opponent tokens defeated: 1000.
    c. Difference in number of throws left: 5000.
    d. Difference in number of tokens in opponent throw zone: 990.
    e. Balance between distance to meat and distance to danger: 1.
    f. Can't defeat any opponent penalty: 0.
    g. Opponent invincible token penalty: 1000000.
3. The third set is the chase down phase. This set is used when we have an invincible token. In this case, we want to win the game and avoid going into a draw. Therefore, we will want to eat all opponent tokens as fast as possible, as well as avoiding them also having an invincible token. The specific weights are described below.
    a. Number of our tokens defeated: 1000.
    b. Number of opponent tokens defeated: 1000000.
    c. Difference in number of throws left: 0.
    d. Difference in number of tokens in opponent throw zone: 990.
    e. Balance between distance to meat and distance to danger: 1.
    f. Can't defeat any opponent penalty: 50000.
    g. Opponent invincible token penalty: 1000000.

Performance evaluation

The efficiency of our minimax algorithm is **O(b^2d)**. It depends on various factors:

→ the searching depth of our minimax algorithm (d)
- For every depth of our minimax algorithm, one action from each of the two players are simulated.

→ the branching factor of search (b)
- ***b = number of friendly tokens * number of possible moves for each token + number of opponent tokens in our throw zone + a constant of 3 throws***

- The number of the possible moves for each token depends on density of friendly tokens. If they are closely packed, each token can have a max of 18 slide or swing actions. If they are not packed together, each token will only have 6 slide actions.
- If the strategy of the opponent agent is to throw all their tokens onto the board, our branching factor would be increased by a linear amount. This will have a great impact on our minimax algorithm performance since our agent searches a depth of 2 at the beginning of the game.

Consider a simple case, each player has 3 tokens on board and assuming they sit far away from each other, and no swing action is possible. Each player will have a branching factor of 3 * 6 + 3 = 21, with a depth limit of 2, the number of leaf nodes visited by performing a minimax algorithm would be 21 ^ 4 = 194481 which in theory takes about 24 seconds to run. But when we reduce the depth limit to 1, only 21 ^ 2 = 441 leaf nodes need to be visited and this only cost approximately 0.05 seconds (using 8000 nodes/sec from observation).

Space complexity:
The minimax algorithm makes a copy of all information about the current board every turn after it simulates a possible pair of actions via deepcopy(). But when the algorithm finished calculating the evaluation function for that branch, the space is released. So, the maximum space complexity we need is the space to store a single branch, which is the depth (d) of the tree multiplied by the space of one board data structure.