Report

Part 1: Approaching

1.1 Problem analysis

RoPaSci 360 is a two-player game, with both players acting as the opponents of each other and aim at maximizing their benefit while minimizing the opponent's benefit. As the two players take action simultaneously, they can get the information about the current state but without knowing the action chosen by another player. Also, the symbol of the tokens which haven't been thrown are unknown, it is an imperfect information game and involves chance moves. The initial state has no token on the board and the terminal test needs to test three results which are win, lose and draw. The action involves throw, slide and swing.

1.2 Algorithm choosing

Adversarial search is used in searching in which two or more players with conflicting goals are trying to explore the same search space for solution and examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

We use the Minimax algorithm to complete the search, as this algorithm supports two players playing the game and satisfies our problem that the opponent gets the minimum benefit while the player gets the maximum benefit.

1.3 Mini-Max tree implementation

We consider two turns further for deciding the best move, and we construct a four-layer minimax tree. To implement this, we first need all the available moves for player and opponent, thus we use possible_move to find all the possible position for tokens currently on the state to SLIDE and SWING (stored in possible_move), together, we use possible_throw functions to generate all the possible THROW actions (stored in possible_throw). We add them together to form a player_total (all available actions for player) and an opp_total (all available actions for opponent).

The first layer in the minimax tree is the player action layer that aims to return the maximum evaluation score among the branches. For each action of the player_total, loop all the actions in opp_total which is the second layer. The second layer focus on the minimum evaluation score. Under the second loop, we use the update function involving the two actions (one for player and one for opponent) which are currently looping to generate a new state. Based on this new state, we apply the possible_move and possible_throw functions again to find the new player_total and new opp_total. At this stage, we start our third layer with looping the new player_total to find the maximum and then the fourth layer is to loop the new opp_total to find the minimum. Then, use the update function again involving the player action from the third loop and opponent action from the fourth loop and produce a new state. The states generated under the fourth loop are the possible states in two turns further. We use the evaluation function to give each possible state an evaluation score that represents how ideal the state is.

Each time when we update the new state, we need to return to the previous state before the next update in the same layer.

1.4 Mini-Max and alpha-beta pruning

First, we consider the second min layer which is the fourth layer. We would like to find the minimum evaluation score for each action in new_player_total among its association with each action in new_opp_total. We record the minimum evaluation score for the first action in new_player_total as min_max (beta). Then for the following actions in new_player_total, if the state generated with an opponent action has a score less than min_max, we prune the following branches and start looping the next action in new_player_total. The min_max will be renewed when all the states generated from a new_player_total action have larger scores than the current min_max, and the new min_max equals to the minimum score among these states. After iterating all the new_player_total actions, the min_max remained is the maximum min_max, that is the alpha, and all alphas compose the second max layer.

Next, we consider the first min layer which consists of the minimum of the min_maxs (beta). Similar to the above, we use min_max1 to record the minimum, and renew the min_max1 to prune and find the maximum min_max1. At the time when we renew the min_max1, we also record and renew the corresponding action in player_total as the final_player_action. After finishing the iterating, the final_player_action will return the best move for the player in this turn.

1.5 Handle simultaneously

Update function in the Player class is mainly used to handle simultaneous updates of both players' actions on the player's board(game state).

Firstly, the update function will record the symbol(s) of both player and opponent and their previous coordinate(s) if the actions are 'slide' or 'swing'. Then their previous coordinate(s) will be deleted from the game state if the actions are 'slide' or 'swing'.

Secondly, there are 2 situations being considered:

ONE. Player and opponent tokens are both going to the same coordinate.

TWO. Player and opponent tokens are going to different coordinates.

ONE. If player and opponent tokens are going to the same coordinate, 3 sub-situations are considered:

a. After arriving at the same coordinate, there are 3 different symbols in the hex.
   According to the rule, all tokens on that hex are being removed from the game state.
b. After arriving at the same coordinate, there are 2 different symbols in the hex.
   Comparing that 2 symbols and decide which symbol wins. All tokens with that winning symbol remain in that hex/coordinate in the game state.
c. After arriving at the same coordinate, there is 1 symbol in the hex.
   The Player's token, opponent's token (and possibly other tokens on that hex before updating) will all remain in that hex/coordinate in the game state.

TWO. If player and opponent tokens are going to the different coordinates, 2 sub-situations are considered:

a. No other tokens occupying the destination coordinate.
   Add the coordinate and token information(symbol and player/opponent) in the game state.

b. Have other tokens occupying the destination coordinate.
   (2 different symbols)If other tokens could be defeated, then substitute the coordinate with the winning token.
   (2 different symbols)If it is 'lose', then do nothing in the game state.
   (1 symbol)If it is 'draw', then append (either player or opponent's) token at the coordinate in the game state.

Init and action functions in the Player class are both called by Referee at the same time so the simultaneous nature is handled by Referee.
Init function in the Player class for both players is called at the same time at the beginning of the game to initialize both players in the game.
Action function in the Player class is also called at the same time in each turn by Referee so that all players have the same game state and make decisions on their actions based on the same game state.

Part 2: Optimizing

2.1 Use evaluation function to reorder

To make the alpha-beta pruning more effective, we use update_player_action function to update a state used only one action from either player. Then, we evaluate the score of each state generated and use insertion sort to order the actions from the highest evaluation score to the lowest score. Ordering the player_total can make the player actions with higher evaluation score appears earlier in the max layer that will improve the alpha-beta pruning. We also apply it to the opponent side, that ordering the opp_total can push the state which minimize the player benefit to the front in the min layer, that will also improve the alpha-beta pruning. It is also applied to the new_player_total and new_opp_total used in the third and fourth layer.

2.2 make space limit/time limit

To make the searching faster, we make limit on the possible actions of each turn for both player and opponent. As we reordered the actions based on their utilities, most of the actions listed behind always get pruned and will not affect the result, thus after testing the program with different numbers of actions keeping for each layer, we found keeping the first six actions is preferred that the others considering both time used and the game result.

To improving the gaming finishing time that keeps each game finished in one minute, we use a strategy that after 55 seconds, we directly return the action that listed on the first place among the reordered player_total list. This method also follows the greedy algorithm as it returns the action with the highest utility with only considering a single move. Also, based on our previous results of battle with greedy algorithm using the same evaluation function, we found greedy also returned comparative good moves and much quicker.

2.3 set throw condition:

To better control the number and symbols of tokens on the state, as well as avoid throwing too quickly or too slowly, we set a condition in possible_throw based on difference in number of symbols between player tokens and opponent tokens. If the difference exceeds one, then we put the corresponding throw action in the possible_throw list and add to all possible actions (e.g., player_total) to help defeat the opponent tokens. For example, if player has one paper on the board and opponent has three scissors, then for the player side, it will put a throw action with rock in the possible_throw list. As we give a high weight for the feature of whether the symbols are balanced on the board, if there is a unbalanced state, the throw action (e.g., a rock) will have a relative high mark and has very large possibility to be the best move returned.

Part3 Evaluation

3.1 evaluation feature

The evaluation function evaluates each possible state based on:

1. Number of opponent tokens being defeated
2. The closeness of opponent tokens that can be defeated by player tokens
3. The closeness of opponent tokens that can't be defeated by player tokens
4. The balances of symbols between player and opponent
5. The closeness of player's token towards player's side

3.2 strategic motivations /How does your game-playing program select actions throughout the game?

1. For the first 3 turns, 3 tokens with different symbols will be throw on the closest 3 layers with the appointed position. E.g.if player is 'upper' then action will be ['THROW', 'r', (4, -1)], ['THROW', 's', (3, -1)], ['THROW', 'p', (2, -1)]
   Why:

a. For the first 3 turns, there is no possibility for the opponent tokens to go to the player's side and defeat the player tokens. (If the player is 'upper' and the opponent is 'lower', the opponent could reach maximum row -2.)

b. Choose 3 different symbols to make sure no matter what the opponent throw, the player has a chance to defeat it.

c. 3 tokens connect with each other so that they could swing to further coordinate in the future turns. Also, different symbols mean they can protect each other by swing if there is an opponent token coming.

d. Save game total time as there is a time limit of 60 secs per game.

2. For the following turns, one action will be selected among all possible actions based on their evaluation score using minimax and alpha-beta pruning.

   Each possible state generated during minimax and alpha-beta pruning will be evaluated by the evaluation function:

   Firstly, the evaluation function checks the number of opponent tokens defeated by player tokens. Evaluation_score += number * 10000

   This weight makes sure that the state with defeats will always be chosen by the algorithm.

   Secondly, it checks the closeness of player tokens with opponent tokens that can be defeated. For each player token A, it finds the distance between A and

opponent token Bi. if the distance is 1, then evaluation_score += 80. If the distance is 2, then evaluation_score += 70…..if distance is 8, then += 10.
This makes sure that states with player tokens closer to defeatable opponent tokens will have higher evaluation scores but not higher than throwing actions if symbols are unbalanced.
Thirdly, it checks the closeness of player token with opponent tokens that can't be defeated. The closer it is, the lower score it has. E.g. distance is 1, evaluation_score -= 80…..distance is 1, evluation_score -= 10. (reverse of second point)
Fourthly, it checks the balances of symbols in the state for player and opponent. If the amount of 's' tokens for the player - amount of 'p' for the opponent > 2, then there are too many symbols 's' in the state. Evaluation_score will be reduced by 1000 to discourage this kind of state. Same with 'r' tokens and 'p' tokens for players.

3.3 effectiveness of the game-playing program and program performance by comparing with multiple programs based on different approaches and selection of the most effective ones.

Compared with:

1. Random_action module we wrote

In all possible actions, there are many more possible throws than possible moves. Hence when choosing a random action among all possible actions, there is a higher probability to throw than to slide for the random_action module.
To deal with this, we set the score for feature 4 balancing symbols higher. By having more balances for symbols in the state, we can prevent random_action from having some invincible tokens.
Also, number of throws will be used up by random_action quickly. So to avoid accidently defeated by their throw, we could have a feature to add more score for tokens closer to our side. So it is outside random_action's throw range.
Despite the threat of throws and invincible tokens from random_action, there are no other worries as their slide and swing action has no strategy at all.

2. Greedy module we wrote

As the evaluation function of the greedy algorithm is pretty similar to ours, the behavior of it will be similar as it returns the most immediately promising action. We found that greedy also favors 'throw' when player's tokens are within throw range. This is mostly because of its evaluation function which has higher score on throw if it can defeat other tokens.
Also, score will be higher when greedy's token is closer to player tokens that can be defeated. Hence even if player is not within range, greedy will still throw a token that can defeat many player symbols and be as close to the player as possible. And if it's within throw range it will choose throw to defeat rather than slowly move to goal.

To deal with it, the score of the closeness of player to its side has changed higher. So that player's token would move to its own side. And then wait for greedy to come. Once it comes closer, player's token will defeat it. At the same time, more restricted balance has been introduced in evaluation functions to control symbols within 1 difference.

Part 4: Additional program for testing

4.1 Random action program

We developed a program called random_action that will randomly return an action from player_total (the list of all possible actions) by using the random module.

By testing with the random action program, the winning rate of our program get to 100% and will win around 17s-18s.

4.2 Greedy algorithm program

We used the same evaluation function to develop the greedy program that will return the move with the highest evaluation score among all the available moves including all possible slide, swing and throw. It only consider the current turn and does not put the opponent actions into consideration. Battling with the greedy program, we use longer time to defeat, this is because the action greedy returned each time would be more optimal. Also, as greedy used the same evaluation function as our program, the strategy involves similarity to some extent that extends the gaming time.

Part 5: Complexity analysis

Time complexity of Minimax algorithm is with b is the number of maximum branches and m is maximum depth. Under this implementation, the b should be 6 and m should be 4. As we use alpha-beta pruning, and we order the action lists, the time complexity becomes .

As the Minimax algorithm is depth-first exploration, the space complexity is .

The Mini-Max tree is finite, and the opponent is optimal. Thus, the algorithm is complete and optimal.