Artificial Intelligence

RoPaSci360 agent

Gilbert Harlim Putra 1006990 and Justin Shi 1081278

Simultaneous Game challenge

The game was modeled as 2-player adversarial search simultaneous play zero-sum game. Though we originally aimed to train our agent through Reinforcement Learning and self-play and develop our own environment to do so, we eventually settled on an implementation of Simultaneous Alpha-Beta Pruning proposed by Saffidine (2012) with Game-Theory to find the optimal moves.

Simultaneous Alpha-Beta Pruning

For a certain state of the game, generate both players possible moves (A and B to be the set of legal moves for row and column player respectively). Let $p_{i,j} = minval = Alpha \ for \ i \in A, \ j \in B$and let $o_{i,j} = maxval = Beta \ for \ i \in A, \ j \in B$. By initializing Pessimistic Bound[1] ($p_{i,j}$) and Optimistic Bound[2] ($o_{i,j}$) for every iteration of legal moves A and B, matrix P and O will be formed to store each respective moves in an orderly manner indicating that row and column are for maximising player and minimizing player respectively.

```
1  SMAB (state q, lower bound α, upper bound β)
2  if q is a terminal state then
3      return payoff for q
4  else
5      let A = the set of legal moves for the row player;
6      let B = the set of legal moves for the col player;
7      let p_{i,j} = minval for i ∈ A, j ∈ B;
8      let o_{i,j} = maxval for i ∈ A, j ∈ B;
9      let P denote the matrix formed by all p_{i,j};
10     let O denote the matrix formed by all o_{i,j};
11     for each (a, b) ∈ A × B do
12         if row a and column b not dominated then
13             let α_{a,b} as defined in Fig. 4 restricted to
               non-dominated actions;
14             let β_{a,b} as defined in Fig. 5 restricted to
               non-dominated actions;
15             let q_{a,b} = the state reached after applying
               (a, b) to q;
16             if α_{a,b} ≥ β_{a,b} then
17                 let v_{a,b} = SMAB (q_{a,b}, α_{a,b}, α_{a,b} + ε) ;
18                 if v_{a,b} ≤ α_{a,b} then a is dominated;
19                 else b is dominated;
20             else
21                 let v_{a,b} = SMAB (q_{a,b}, α_{a,b}, β_{a,b}) ;
22                 if v_{a,b} ≤ α_{a,b} then a is dominated;
23                 else if v_{a,b} ≥ β_{a,b} then b is dominated;
24                 else let p_{a,b} = o_{a,b} = v_{a,b};
25             end
26         end
27     end
28     return Nash(P restricted to non-dominated actions)
29 end
```

Throughout the development of the suggested Simultaneous Alpha-Beta Pruning, there were three different iterations of the algorithms.

1. **The First iteration** **of the algorithm was a trial of the algorithm to see how and study how it actually worked. This version avoids calculating Pessimistic Bound ($\alpha_{a,b}$ is the bound after applying legal move a and b to the current state) and Optimistic Bound ($\beta_{a,b}$ is the bound after applying legal move a and b to the current state) for each pair of movement and instead uses Lower Bound α and Upper Bound β, skipping Linear Programming calculations.**

2. **The Second iteration** **of the algorithm was the appropriately implemented algorithm, without skipping calculations of $\alpha_{a,b}$ and $\beta_{a,b}$. The calculation of each respective bound will be shown later.**

[1] **Pessimistic Bound: The minimal payoff of a stateafter applying action a and b from legal moves set of A and B.**

[2] **Optimistic Bound: The maximal payoff of a state afterapplying action a and b from legal moves set of Aand B.**

3. **The Final iteration** of the algorithm was the combination of the second iteration and the suggested method to improve the time constraint by Saffidine, (2012). The suggested method was to explore the set of legal moves A and B in a lexicographical ordering over tuples ($min\{a, b\}$, a, b), called L-shaped cell ordering in the paper, with Skipping Parameters (S)[3] to save time with the Linear Programming. This ordering has been proven to actually accelerate the algorithm in finding an optimal solution. As it has been set that the maximizing player is the row player, the ordering explores maximizing player actions first, rather than minimizing player actions. For example, if in cell order 8 it finds the optimal solution, it will skip the whole matrix as it has already explored most of the maximising player (row) actions.

1	2	3	4	5
6	10	11	12	13
7	14	17	18	19
8	15	20	22	23
9	16	21	24	25

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_{a-1} \\ x_{a+1} \\ \vdots \\ x_m \\ x_{m+1} \end{pmatrix}, P = \begin{pmatrix} p_{1,1} & \cdots & p_{1,b-1} & p_{1,b+1} & \cdots & p_{1,n} \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{a-1,1} & \cdots & p_{a-1,b-1} & p_{a-1,b+1} & \cdots & p_{a-1,n} \\ p_{a+1,1} & \cdots & p_{a+1,b-1} & p_{a+1,b+1} & \cdots & p_{a+1,n} \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{m,1} & \cdots & p_{m,b-1} & p_{m,b+1} & \cdots & p_{m,n} \\ \alpha & \cdots & \alpha & \alpha & \cdots & \alpha \end{pmatrix}, e = \begin{pmatrix} p_{1,b} \\ \vdots \\ p_{a-1,b} \\ p_{a+1,b} \\ \vdots \\ p_{m,b} \\ \alpha \end{pmatrix}$$

$$f = (o_{a,1} \quad \cdots \quad o_{a,b-1} \quad o_{a,b+1} \quad \cdots \quad o_{a,n})$$

$$\alpha_{a,b} = \max x^t e, \text{subject to } x^t P \geq f, 0 \leq x \leq 1, \sum_i x_i = 1, \text{or minval} - 1 \text{ if the LP is infeasible}$$

$$x = (x_1 \quad \cdots \quad x_{b-1} \quad x_{b+1} \quad \cdots \quad x_n \quad x_{n+1})$$

$$O = \begin{pmatrix} o_{1,1} & \cdots & o_{1,b-1} & o_{1,b+1} & \cdots & o_{1,n} & \beta \\ \vdots & & \vdots & \vdots & & \vdots & \beta \\ o_{a-1,1} & \cdots & o_{a-1,b-1} & o_{a-1,b+1} & \cdots & o_{a-1,n} & \beta \\ o_{a+1,1} & \cdots & o_{a+1,b-1} & o_{a+1,b+1} & \cdots & o_{a+1,n} & \beta \\ \vdots & & \vdots & \vdots & & \vdots & \beta \\ o_{m,1} & \cdots & o_{m,b-1} & o_{m,b+1} & \cdots & o_{m,n} & \beta \end{pmatrix}, f = \begin{pmatrix} p_{1,b} \\ \vdots \\ p_{a-1,b} \\ p_{a+1,b} \\ \vdots \\ p_{m,b} \end{pmatrix}$$

$$e = (o_{a,1} \quad \cdots \quad o_{a,b-1} \quad o_{a,b+1} \quad \cdots \quad o_{a,n} \quad \beta)$$

$$\beta_{a,b} = \min ex^t, \text{subject to } Ox^t \leq f, 0 \leq x^t \leq 1, \sum_i x_i = 1, \text{or maxval} + 1 \text{ if the LP is infeasible}$$

*X*is a set of actions and its probability for each respective player ($\alpha_{a,b}$ **X is for row player and** $\beta_{a,b}$**X is for column player) other than the current action we are exploring. To compensate for the deviation of current action, other probabilities** X_{m+1}**and** X_{n+1}**are added to set the total probability to 1. Matrix P has also been added with Lower Bound Alpha to compensate for the deviation with the exception of the current action, it is also the same for Matrix O but it has been added in the columns with Upper Bound Beta.** $\alpha_{a,b}$**and** $\beta_{a,b}$**can only be calculated if the set conditions are true, else it will be the default set value for it.**

[3] **Skipping Parameters, S: This parameter has been set to skip every Linear Programming calculations of Alpha a,b and Beta a,b until a certain set S number of columnshas been evaluated in the Pessimistic and Optimistic Bound Matrices.**

Modules and Classes

The directory can be broken down into:

state.py: Contains RoPaSci360 game logic, globals and functions that define the game. It also contains a legal moves generator as well as the RoPaSci360 class.

adversarial.py: Contains the adversarial search algorithms including the three iterations of Simultaneous Alpha-Beta Pruning and Double Oracle method suggested by Bosansky, (2013).
heuristic.py: Contains all state features and weighted evaluations in the game.

gametheory.py: To compute the Nash Equilibrium expected payoff and strategy. Taken from COMP30024 Artificial Intelligence.

player.py: Contains the player logic and some preset actions for early game (6 turns into the game).

Heuristics

There are seven main features and two weighted evaluations used to inform the behaviour of the agent. They are all calculated and represented as percentages so none of the features and evaluations has an unintentional overbearing effect on the behaviour of the agent. The percentages are then multiplied by their weights to influence this behaviour.

The seven following features are:

Cost to enemy:
This feature computes the average distance between our tokens and enemies tokens that can be captured. The difference between this average and the maximum distance of nine is then taken as a percentage of the maximum distance. This value informs the program of the capturing potential of its tokens; giving a lower value if tokens are further away from the opposing tokens that they may capture, and a higher value if tokens are closer to their capturable opponents. This encourages the program to move tokens closer to capturable tokens, in order to maximise this value.

Total tokens:
The feature is simply the fraction of how many remaining tokens the program has available, out of how many tokens it is given at the start. In order to keep this value high, the program is incentivized to keep this value high, and not kill its own tokens to achieve the goals of the other heuristics.

Cost from enemy:
This feature calculates the average distance that tokens are from all of the opposing tokens that may capture them, before turning it into a percentage of the maximum. Unlike the first heuristic, this value does not need to be adjusted as it already gives higher values if tokens are further away from the opponents that may capture them. In order to maximise this value, the program is encouraged to move tokens away from dangerous opponents.

Save throws:
This feature is the percentage of throws that the program has remaining. This feature incentives the program not to quickly use up all of its throws, after it has executed its opening playbook or set moves. This is in order to avoid a situation where the opponent may gain invincible tokens if all tokens are preemptively thrown, while allowing very aggressive throws later on in the game, when the opponent has multiple tokens sitting in the program's throw range.

Enemy captured:
This feature is the percentage of enemy tokens that have been captured. Obviously, the program is required to capture the enemy tokens

in order to defeat its opponent, and in order to encourage this behaviour, this heuristic essentially rewards the program for capturing enemy tokens.

Cost to allies:
This feature is the average distance between our token and its closest allies. This creates an incentive for the program to have a closer set of tokens, so that in case of enemy attacks it can defend itself quicker. (This has been found to be ineffective and results in more losses)
Balanced token:
This feature is to maintain the balance of the number of rock, paper, scissor tokens with the enemies. If the enemy has more than the player, the evaluation falls.

The two weighted evaluation functions are as follows:

Conservative:
Just like its name, this evaluation function has been set so that the player behaves neither too aggressively or passively. (Measured in percentages)

- Cost to enemy × 12
- Total tokens × 20
- Cost from enemy × 15
- Save throws × 2
- Enemy captured × 20
- Balanced token × 5

The sum of all these will be divided again by the total of the multipliers (12+20+15+2+20+5).

Greedy:
This evaluation function highly encourages capturing enemies tokens and maintaining a high percentage of total tokens. (Measured in percentages)

- Cost to enemy × 15
- Total tokens × 15
- Cost from enemy × 12
- Save throws × 1
- Enemy captured × 15

The sum of all these will be divided again by the total of the multipliers (15+15+12+1+15).

Optimizations Techniques

The set of legal moves for each player can be a total of 264 moves at worst.

For throw actions, after every throw the legal set of throwing increases (5, 11, 18, 26, 35, 43, 50, 56, 61 per throw), and as there are three symbol possibilities, this causes this to be multiplied by three (at worst 183).
For slide actions, if all nine tokens are on the board it will be a total of at worst, 54 possible outcomes.
For swing actions, if all nine tokens are on the board it will be a total of at worst, 27 possible outcomes.

Our optimization strategy is that for the first six turns of the game, our program follows a preprogrammed set of six actions. With this we essentially eliminate the time constraint for the first six turns. For the following self-generated sets of legal moves, we cut down the throw actions to just where we can stomp the enemy tokens with ours if it is within our throw range. This reduces the total legal moves significantly to less than 90 legal moves, which improves the time that the Simultaneous Alpha-Beta Pruning takes. Although this method may miss optimal moves, the time gained through this implementation, as well as the high likelihood that one of these throws will eliminate an opponent's token, when multiple tokens are in range, was determined to be worth the trade-off.

Despite the optimization steps taken, the Simultaneous Alpha-Beta with depth 2 still takes more than 30 seconds on our machines,

which makes it non-ideal. With this in mind, the algorithm is only set to depth of 1 which conducts only one-step lookaheads. With just one-step lookahead and the optimization, it can compute optimal moves within 0-5 seconds. This is a significant improvement as before optimization, it would take 20-30 seconds each turn.

Furthermore, with the Skipping Parameters and Cell-ordering method suggested by Saffidine, et al (2012). The algorithm can compute much faster, approximately 1-2 seconds per turn.

In conclusion, this optimization reduces the time constraint significantly and introduces a capturing strategy while sacrificing possible optimal legal moves.

Other Techniques

Double-Oracle method and Serialized Alpha-Beta pruning (Bosansky, 2013) result in slower computation per turn despite the optimization than the Simultaneous Alpha-Beta pruning (Saffidine, 2012).

Reinforcement Learning by using Custom Environment utilizing OpenAI gym with Proximal Policy Optimization algorithm and Actor Critic Policy. At first, this method with its alluring self-play can create maybe one of the best RoPaSci360 agents, but after proceeding and building the environment. We also had to modify Actor Critic Policy to be fitted into our environment, which we determined that we simply did not have the time to understand and finish within the scope of the project. In the future, we may finish it as our own personal project because this is such a marvelous approach to AI.

Another method that was considered, was forgoing a search algorithm entirely and simply using heuristics to guide the actions of the player. In this implementation, the program had simple actions that could be issued to it, attack, run away, throw defensively, or throw offensively. These actions would then be issued according to the state of board.

As the player constantly keeps track of all the tokens on board, it would not be necessary to search around each token to find optimal moves. Instead each token was issued an attack and defend value, calculated according to the positions of the opponent's tokens and the token with the highest attack or defend value would then take the corresponding action, either attacking or running away. These actions could also be influenced by the overall state of the board.

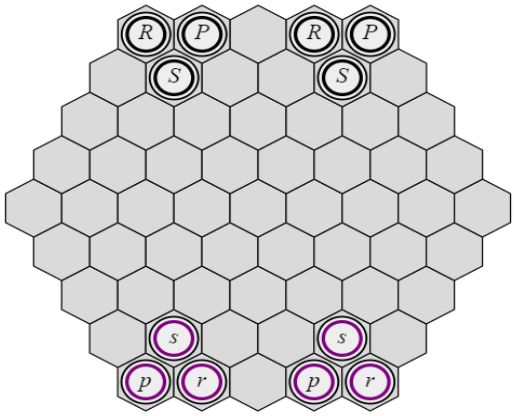
The program would always aim to deny the opponent any invincible tokens and would prioritize attacking if it had any invincible tokens of its own, additionally it would also additionally favour either attacking or defending depending on whether it held a favourable current board position or not. Additionally once a certain turn was reached, it could be set to an extremely aggressive playstyle, in order to close out the game before the turn limit was reached.

Ultimately this implementation's advantages were it's incredibly fast computational times and overall understanding of the board state; as it did not need to perform searches on the board and thus its knowledge was also not limited to its search depth. However, it could miss optimal moves and as our other more conventional implementations sped up, its lightning fast speed became more and more irrelevant.

Ultimately, this agent was retired and its heuristic calculations were cannibalised to feed and improve the evaluation capabilities of our other agents.

Agent approach

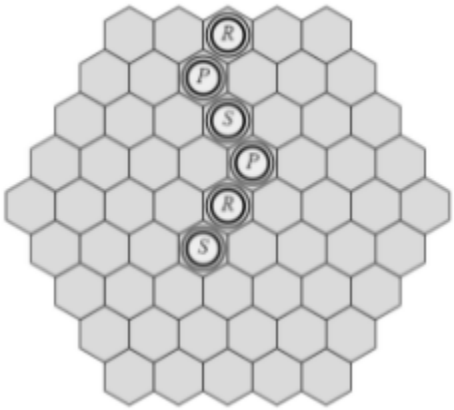
Preset Action 1:



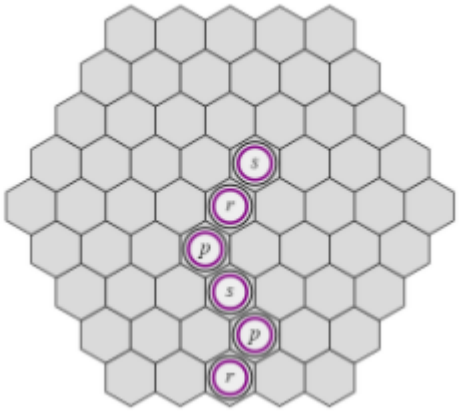
These are the preset actions for the agent if it is playing as Upper or Lower

Preset Action 2:

If playing as Upper



If player as Lower



After running some trials with Preset actions 1 and 2, we found out that preset action 1 consistently wins over preset action 2. Both presets have been tested with Conservative and Greedy weighted evaluation functions. Comparing Conservative and Greedy evaluations, we observed that conservative with preset action 1 results in consistent wins relative to the other combinations.

Future Considerations

The following are things that still could be improved:

- **Research into RoPaSci360-appropriate heuristics**
- **Use the memory close to maximum limit**
- **Combine a few adversarial searches to define the player logic**

References and Acknowledgements

**Saffidine, A. (2012). Alpha-Beta Pruning for Games with Simultaneous Moves.
Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (p. 556-562)**

**Bošanský, B. (2013). Using Double-Oracle Method and Serialized
Alpha-Beta Search for Pruning in Simultaneous Move Games.
Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (p. 48-54)**