COMP30024 - Project 2 Report
SID: 1086144 1081894

Approach

Game Theory

We implemented game theory for the program to select actions throughout the game. We considered implementing minimax algorithm but due to the simultaneous-play nature of RoPaSci360, we decided to use game theory as it provides mixed strategy which introduces uncertainty in opponent's program when selecting actions.

The pseudocode for selecting an action is as follow:

```
Matrix = []
For each action in player_action():
        Row = []
        For each action in opponent_action():
                New_state = state.update(player_action, opponent_action)
                row.append(eval(New_state))
        matrix.append(row)
Return matrix

p, v = solve_game(matrix)

action = np.random.choice(len(actions), 1, p = p)
Return action
```

The pay-off matrix is represented by player_action as row, and opponent_action as column, with the value being the evaluation score of the state formed by respective player action and opponent action.

From the pseudocode we can see that the search depth is only 1-ply, only one new state is created for each combination of player and opponent action, and the program only evaluated these new states to produce a single pay-off matrix.

Evaluation

Evaluation function is one of the most important aspect in the program as it determines the strategy of the game-playing program, we chose the following features for evaluation:

1. Number of upper tokens - number of lower tokens
2. Number of player throws remaining - number of opponent throws remaining
3. Summation of all player tokens and capturable opponent tokens distance
4. Closest player token and capturable opponent tokens distance
5. Number of opponent token in immediate danger (closest dist == 1)
6. Summation of all opponent tokens and capturable player tokens distance
7. Closest opponent token and capturable player tokens distance
8. Number of player token in immediate danger (closest dist == 1)
9. Number of invincible player token
10. Number of invincible opponent token

Distance

Since evaluation function returns a larger value if the state is more advantageous, the smaller the distance of player tokens and capturable opponent tokens, the larger the return value should be and vice versa.
To account for this inverse relation, we do the following operations to get the score:

```
max_dist = Hex.dist(Hex(-4, 4), Hex(4, -4))
value = max_dist - closest_capture_distance
```

Max_dist is the largest distance between any point in the board, this way we can get higher value if the tokens are close to each other.

Capturing

Since capturing opponent tokens means that there will be a new closest capturing distance that is more likely of lesser value than the closest capturing distance before capturing opponent tokens. The weighting of #1 feature should account for this, and the program should not avoid capturing opponent tokens to preserve the value of closest capturing distance.

Invincible tokens

We decided that invincible tokens are the key to victory in RoPaSci360, so the weighting of this feature should be the highest.

Weighting

Each feature carries an individual weighting such that different strategies can be employed and customised. For example, a greedy opponent will only select at the most immediately promising action available, according to the formula and the features stated above:

$$\begin{aligned} \text{Eval}(s) &= w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \\ &= \sum_{i=1}^{n} w_i f_i(s) \\ &= \boldsymbol{w}.\boldsymbol{f}(s) \end{aligned}$$

We can create a simple greedy opponent that only attacks the closest capturable token and escapes from the closest dangerous token by using

```
w = [20, 0, 0, 10, 10, 0, 10, 10, 20, 20]
```

We will also be using genetic algorithms to improve the quality of weighting for the evaluation function.

Genetic algorithms (GA)

To obtain an evaluation function with better quality, we decided to improve the weighting of the evaluation function with a genetic algorithm.
The core mechanic of genetic algorithms consists of:

1. Initialize population
2. Calculate population fitness
3. Selecting parents
4. Crossover
5. Mutation
6. Create next generation

The methodology is to calculate the fitness of the population, then select the population with highest fitness as parents and apply crossover and mutation to them to create a new generation, this process is then repeated until we are satisfied with it.

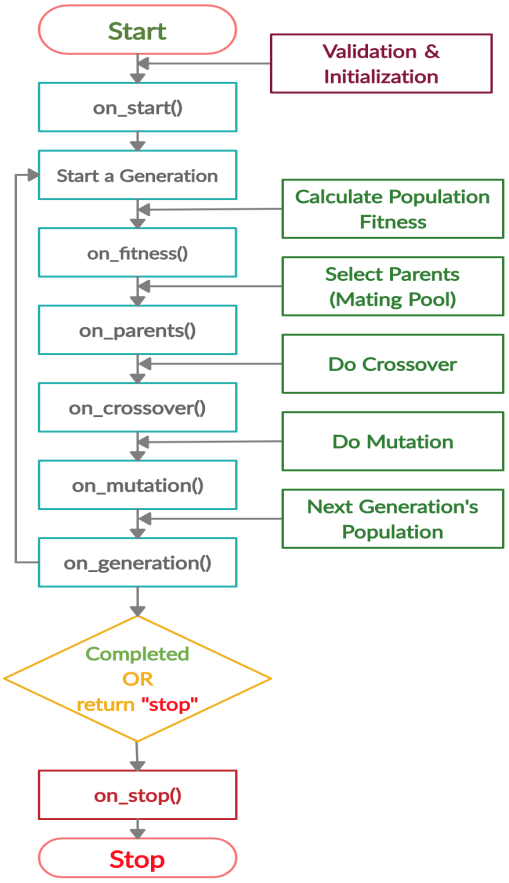

Figure1. *Pygad documentation*

Population initialization

GA begins by initializing the population with random values. We first specify the population size and number of inputs for the initial population.
Since the number of inputs corresponds to the number of features, which is the number of weight, num_inputs will be 10, here's one of the population example:
[2.12345008 1.58668476 2.05608363 0.92559664 0.14207212 0.03918779 0.0202184 1.66523969 2.48086769 4.35006074]

Calculating population fitness

We first calculate the population fitness by playing the game against opponent with fixed weight, such as the greedy opponent mentioned above:

```
w = [20, 0, 0, 10, 10, 0, 10, 10, 20, 20]
```

The game result will be returned as a fitness score.

Selecting parents

Population with the highest fitness is selected as parents.[1]
The number of parents can be specified in GA, and the learning rate can be derived from the number of parents, as parents are preserved in the next generation, so the higher the number of parents, the lower the learning rate as the number of mutations is lower.

Crossover

Crossover is the process of creating children for next generation by mixing and matching parts of two parents

Mutation

To prevent the population from converging, we introduce variation by mutating the offspring, we do this by specifying the range of values that will be randomly selected and randomly added to the offspring.

Applying GA to RoPaSci360

Due to the simultaneous-play nature of the game, we couldn't predict the opponent's action and we have to choose an action with the corresponding probability matrix. This implies that the fitness of the first population and second population might vary greatly despite having similar weighting.

Therefore, we will first be using GA in an environment with fixed seed.

```
np.random.seed(0)
```

Once we have a population of weight that is similar to each other. We can then set the next seed and repeat GA but this time with initial population = weighting of the last GA iteration.

After playing against the greedy opponent multiple times in different fixed seed, we select one of the weightings from the final population as the opponent, then play against it for multiple times in a fixed seed environment.

---

[1] Jason. B (2021). *Simple genetic Algorithm from Scratch in Python*
https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/

Performance

According to the game theory approach, we evaluate each player actions with each possible opponent action. The algorithm is in $O(n^m)$, where n is the number of available player actions and m is the number of available opponent actions.
Time taken each game is roughly 50 seconds.
Space used in each game is roughly 5MB.

The program can only look one move ahead since the algorithm will become $O(n^{m}*n'^{m'})$ if we expand another matrix for every possible combination, where n' and m' is the available player and opponent actions of the state created by n and m, and it will exceed the time limit if we increase the search depth to 2-ply

Improvement to be made

Performance

As mentioned above, the program is not efficient enough to search at a depth of 2-ply, this can either be improved by implementing pruning techniques in game theory or improving the efficiency of the loop that is responsible for creating the pay-off matrix. The program may also exploit the given resources (time and space limit) to explore more possibilities at a depth of 2-ply when efficiency can be compromised or when the matrix is small at the end and initial stage of the game.

Evaluation

The evaluation function can be further improved by adding new features to it, one of the features that can be added is num_moves_available, the more number of moves available, the more advantageous it is to the player.

Adding this feature also means that the program will favour swing action over slide action, and will group the same player's tokens together to create more swing possibilities.

We tried to add this feature in the evaluation function. Unfortunately, evaluating the number of moves available means that we have to add another loop in the evaluation function which iterates through every possible player and opponent actions.

This results in exceeding the time limit, thus we couldn't add this feature to the evaluation function.

Genetic algorithm

The learning process took roughly 5hours if the population is of size 15 and the number of generations is 10, so optimisation can be made to speed up the process and increase the number of generations.

Result

Here's one of the weighting produced by GA:
[1.58345008 1.76668476 1.93608363 0.46559664 2.79852716 0.20754596
 0.43674726 1.44523969 3.90086769 4.76871113]
We can see that the program values invincible tokens the most as predicted.
We tested the program against different opponents and got the following results:

- 100% win rate against opponent who pick random action out of 10 games
- 90% win rate, 10%draw rate against greedy opponent out of 10 games

Reference List

Pygad Documentation. (2020) *Pygad documentation.*
https://pygad.readthedocs.io/en/latest/README_pygad_ReadTheDocs.html#pygad-ga-class

Jason. B (2021). *Simple genetic Algorithm from Scratch in Python*
https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/