COMP30024 Artificial Intelligence
Project Part B Report

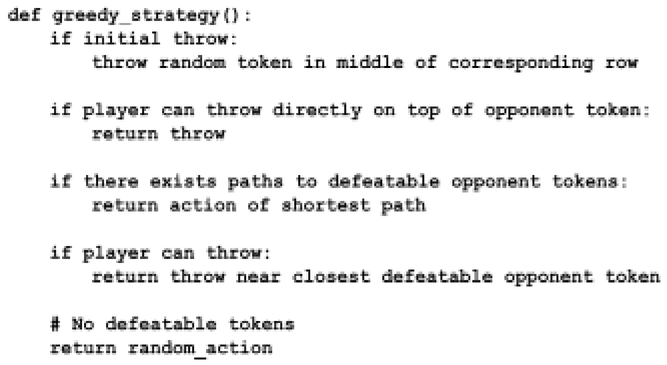
Tony (Hoang) Dang 1080344
Jennifer Xiang 1080696

1. Describe your approach

For this project, we mainly considered three types of game-playing strategies to solve and play the game of RoPaSci 360. The three approaches are Greedy Player, Single-Stage Mixed Strategy Nash Equilibrium, and Multi-Stage Mixed Strategy Nash Equilibrium. From here on, Mixed Strategy Nash Equilibrium will be referred to as MSNE.

Approach 1 - Greedy Player

To begin simple, our team first focused on developing a greedy player that was capable of consistently beating a random player. Our logic for the greedy player is as follows. We use astar search to determine the optimal path between tokens.



```
def greedy_strategy():
    if initial throw:
        throw random token in middle of corresponding row

    if player can throw directly on top of opponent token:
        return throw

    if there exists paths to defeatable opponent tokens:
        return action of shortest path

    if player can throw:
        return throw near closest defeatable opponent token

    # No defeatable tokens
    return random_action
```

(1.1) The Greedy Strategy

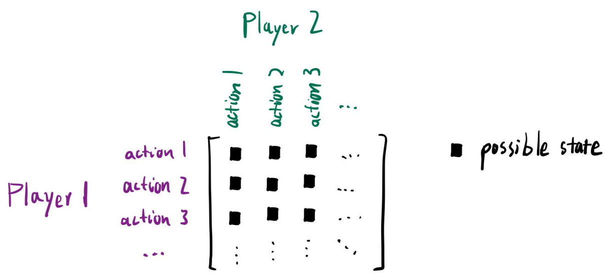
As expected, this greedy player was able to beat a random player 100% of the time with high efficiency (consistently in less than 50 states with <1 second run times). Similarly, when Greedy Player was tested on the online battleground, it beat the default random and greedy bots 100% of the time. Greedy Player was also tested against real players on the battleground, where its win/draw/loss rate was about 40/40/20 (win/draw/loss). We shall use this greedy strategy as the benchmark for any following strategies.

Approach 2- Use of Single-Stage Mixed Strategy Nash Equilibrium

For our second approach, we explored the MSNE method.

Why we picked this approach: MSNE is popular for being capable of solving simultaneous-play games as it generates new states for each pair of simultaneous actions from both players (in the case where there are 2 players). Thus, MSNE is able to take into consideration what the other player might do and produce a strategy accordingly. Since MSNE generates a probability vector, it also introduces some unpredictability into our player's actions. It is important to note that MSNE assumes that the opposing player is a rational agent and is trying to gain the optimal utility score for themselves.

How we implemented MSNE: At each state of the game, we generate a list of actions (or so called strategies) for each player, this includes generating all attack actions, run actions, and throw actions for our player and the opponent player. This list of actions only includes actions we deem to be relevant, the pruning strategy is explained in [3. Other Aspects]. Each pair of actions (our action, opponent action) corresponds to a different new state, meaning we can build a matrix of possible states from these 2 lists of actions. From this matrix of possible states, we can evaluate the score for each state using our evaluation function (described below), and thus compute a utility matrix for each player where each possible state evaluation is represented by a single number in the utility matrix. This utility matrix has the same dimensions as the possible state matrix, and the higher the utility score, the better the state is deemed to be. For each possible state in the matrix, the corresponding utility score was calculated by subtracting the opponent player's evaluation score from our player's evaluation score. This means that for the combined utility matrix, our player will be the maximiser, and the opponent player will be the minimiser, as a positive score means our evaluation was higher than the opponent's, and a negative score means the opponent's evaluation was higher than ours. After generating this combined utility matrix, we pass this into the solve_game function provided to us which uses the Linear Programming function from Python's scipy library to solve the linear system of equations embedded in the combined utility matrix and produce a probability vector (one probability for each of our actions), which is the equilibrium mixed strategy that our player should employ. This probability vector adheres to the principles of MSNE and guarantees a mixed security level for our player. Thus, we pick our next action according to the given probabilities.

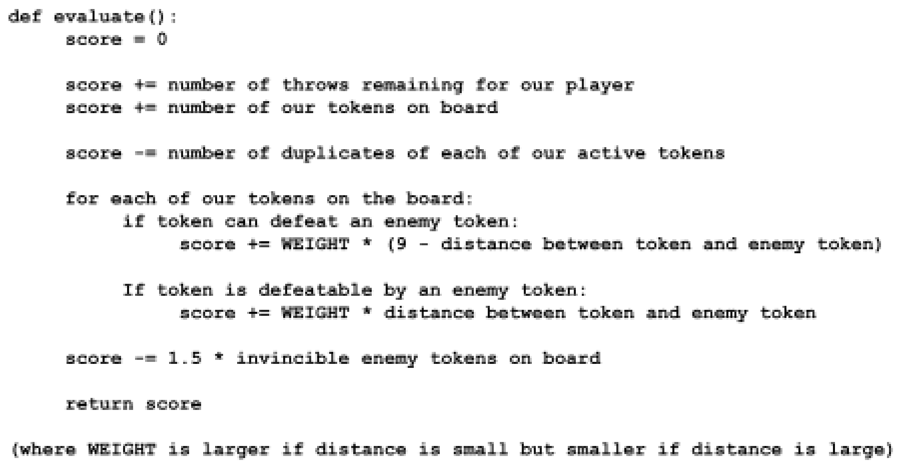


Example of a possible state matrix (corresponding utility matrices have the same shape as their state matrix)

Our evaluation function follows the logic described below (1.2). The features of the evaluation function and their corresponding strategic motivations are as follows:

- Throws remaining - the more throws we have left, the better, as there are fewer vulnerable targets on the board and there is a lower chance the opponent will end up with invincible tokens.
- Duplicate tokens on board - try to avoid having many duplicates of the same type, as they can become easy targets for the opponent player and can also increase the likelihood of an invincible opponent token.
- Token positions relative to defeatable opponent tokens - the closer we are to tokens we can defeat, the higher the chance we can execute a kill.
- Token positions relative to dangerous opponent tokens - the further we are from those tokens, the lower the chance we will be defeated.
- Invincible enemy tokens - the fewer of such tokens there are, the better.

We also assign weights to the distance measures, which dictates that opponent tokens close by are more significant and have more potential to drastically change the evaluation of the game state than opponent tokens further away.



```
def evaluate():
     score = 0

     score += number of throws remaining for our player
     score += number of our tokens on board

     score -= number of duplicates of each of our active tokens

     for each of our tokens on the board:
          if token can defeat an enemy token:
               score += WEIGHT * (9 - distance between token and enemy token)

          If token is defeatable by an enemy token:
               score += WEIGHT * distance between token and enemy token

     score -= 1.5 * invincible enemy tokens on board

     return score

(where WEIGHT is larger if distance is small but smaller if distance is large)
```

(1.2) The Evaluation Function

Approach 3 - Use of Multi-Stage Mixed Strategy Nash Equilibrium
Building upon the basis of our Approach 2, Approach 3 further extends MSNE to be able to handle 2 step lookahead. Similarly to Approach 2, given a set of viable actions for our player and the opponent player, it will create next possible game states for every combination of relevant actions. However, in the case of multi-stage MSNE, it will go a step further and recursively create next states for those next states, resulting in an abstract tree-like structure of next game states. We use backwards induction as means to propagate information up this tree. Here, each node in the tree is a potential game state with a corresponding utility matrix, and each branch represents a pair of player actions (our action, opponent action) that lead to further states. At the leaf nodes, the evaluation function gives an evaluation score of that state and propagates that value up into the utility matrix of its parent node. Once all values in the parent node are filled, it will solve this utility matrix and return a guaranteed minimum expected value (as we play the maximiser) for the equilibrium mixed strategy, which will in turn be further propagated up the tree. Once the propagation is complete and reaches the root node, we once again solve this utility matrix at the root node and return a probability vector which we use to determine our play action. For a clearer visualisation of our implementation of backwards induction with MSNE, please refer to diagram (1.3) attached at the bottom of this document.

2. Performance evaluation

We evaluated the effectiveness of each of our approaches by comparing its performance against one another. As mentioned previously, our Greedy Player was used as the main benchmark for performance evaluation and testing. We developed a simple script that plays two programs against each other for a set amount of games, totalling and calculating outcomes across these games.

To begin, we tested our Approaches 2 and 3 against our Approach 1, the Greedy Player. The results are as follows.

v/s	**Approach 2: Cutoff = 1** (150 games as upper, 150 games as lower)			**Approach 3: Cutoff = 2** (150 games as upper, 150 games as lower)		
	Win Rate	Draw Rate	Loss Rate	Win Rate	Draw Rate	Loss Rate
Approach 1: Greedy Player	0.97 291	0.02 6	0.01 3	0.96 288	0.036 11	0.003 1

Here we can see that both approaches result in high win rates against Greedy Player with relatively similar statistics. Since our Greedy Player plays predictable moves, and Approaches 2 and 3 were developed specifically to target this behaviour, these results are to be expected.

Next, we tested our Approach 2 against our Approach 3. The outcome is shown below.

v/s	**Approach 3: Cutoff = 2** (150 games as upper, 150 games as lower)		
	Win Rate	Draw Rate	Loss Rate
Approach 2: Cutoff = 1	0.43	0.23	0.34

We can see that on average Approach 3 outperforms Approach 2. This goes to show that the two step lookahead utilised by Approach 3 is, in the long term, beneficial. However, a major flaw of Approach 3 is that it is quite time inefficient. Despite using a more aggressive pruning strategy than in Approach 2, most Approach 3 games take over 60 seconds to complete, which violates the restrictions of the task. We also note that Approach 3 does not yield significantly higher performance against Approach 2. Hence, due to the time constraints placed upon this project, we choose Approach 2 as our final agent.

As a safety check, we also played our Approach 2 against a random player. The results are to be expected with an almost absolute win rate. The observed draw from our tests is likely the result of our Approach 2 player being confused with the illogical moves made by the random player.

v/s	**Approach 2: Cutoff = 1** (150 games as upper, 150 games as lower)		
	Win Rate	Draw Rate	Loss Rate
Random Player	0.996	0.003	0

Furthermore, utilising our Approach 2 on the online battleground, we consistently saw an approximate 70/25/5 (win/draw/loss) performance. However, as we matched up randomly against other team agents, it is likely the case that these statistics are unreliable. From what we observed, the higher draw rate was a result of many of the games ending in repeated states where the opponent player behaves irrationally, leading to our agent making illogical decisions in response. This is due to the fact that our agent assumes the opponent will likely make a logical move, hence leading to suboptimal performance when the opponent's behaviour is highly unpredictable. Another flaw noted is that many times our player will reach a stalemate where it is continuously chasing a fleeing opponent token.

3. Other aspects

Pruning of player actions:
As we were developing Approach 2 and 3, we ran into the issue where our program would exceed the given time limit. This is due to the fact that the number of possible states grows exponentially with the number of possible actions. In order to combat this, we chose to only consider the most relevant actions of each player. For each player, instead of considering all possible actions, we only consider the actions that “attack” the closest defeatable opponent tokens and the actions that “run” from the closest opponent tokens that can defeat us, as well as a maximum of three best throw actions (one of each token symbol). Here, we make the assumption that these actions are the most beneficial for a player to consider. With the restricted number of possible actions, Approach 2 was able to run reliably within the time limit. However, in Approach 3 with search depth of two, the number of possible next states grows at an even steeper rate. Thus, we used a more aggressive pruning strategy, taking into account only a single closest defeatable token and a single closest threat. This allowed the Approach 3 program to run much more efficiently.

Strategy combination:
While testing our agent, we noticed that, although extremely rarely, there would still be some instances where our Approach 2 agent would exceed the given 60 seconds of play time. To guarantee a timely finish, we implemented a combination of our Approaches 1 and 2. The program functions as per the specifications of our Approach 2, but would switch to the greedy strategy of our Approach 1 if the time remaining is less than 15 seconds, ensuring that we do not violate the time constraints of RoPaSci 360.

Defense mechanism:
Another issue we encountered while testing our player was that we realised a fair amount of draws resulted from a continual loop of our token running away from an opponent’s attacking token. Hence, we implemented a series of strategies to counter this. The logic is as shown below. Here, “threat” describes the closest opponent token that is able to defeat one of our tokens. If the below function returns None, we would run our MSNE strategy as per usual.

```
def run_strategies():
    if a threat exists:
        if threat is adjacent to the vulnerable token:
            if there exists an opponent token that our vulnerable token can defeat:
                return move towards closest defeatable opponent token

            if an ally token that can defeat the threat exists:
                return run_to_ally

            if an opponent token of the same type as the vulnerable token exists:
                return run_to_enemy

            if threat is in our throw zone:
                return throw a token that can eliminate the threat on top of threat

        elif threat is within 3 steps of our vulnerable token:
            threat_path = path the threat will take to defeat our vulnerable token
            if there exists an ally token that can defeat the threat AND this token is
                    able to move onto the immediate next hex of threat_path:
                return ally_cut_in

            if there exists a throw action that can throw onto the immediate next hex of
                    threat_path:
                return throw_cut_in

        if there exists no opponent tokens our tokens can defeat:
            return throw as close as possible to an opponent token

    return None
```

(3.1) Run Strategies

4. Supporting work

test_evaluate.ipynb - When developing our evaluation function, we created a Jupyter Notebook script that is able to generate random board states, and given an evaluation function is able to output the given evaluation score for that board state. Using this, we were able to compare different board states and assess the appropriateness of our evaluation function, determining whether the numerical outputs aligned with our logical assessments.

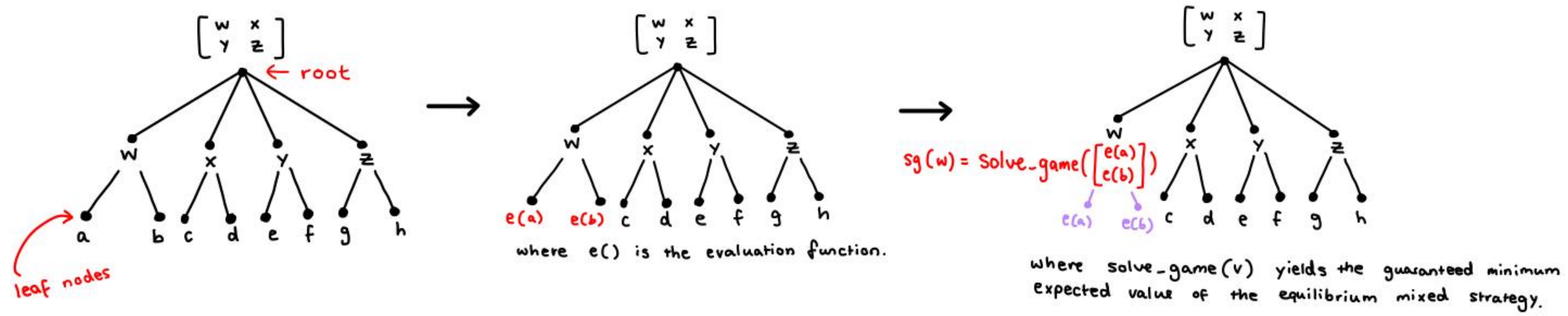
test.txt - As mentioned previously, we also created a bash script that enabled us to play programs against one another for a set amount of games, totalling and calculating outcomes across these games. This helped us determine the effectiveness of different strategies and approaches against one another, yielding win/draw/loss ratios for comparison and analysis.

random_player - When building the foundations of the program, we first built a random bot that would do random actions, this later became a basic benchmark test for our new players.

greedy_player - As discussed in Approach 1, we built a greedy bot that would go after the seemingly easiest kill at every turn. This bot can find the optimal path to chase down tokens as well as good positions to throw a token (on top of a defeatable token or close to it). This greedy player helped solidify the foundational structure of our program and later became very useful as a benchmark test (as it consistently beats both generic bots that were provided on the online battleground).

Diagram (1.3): Backwards Induction with MSNE and Evaluation Function

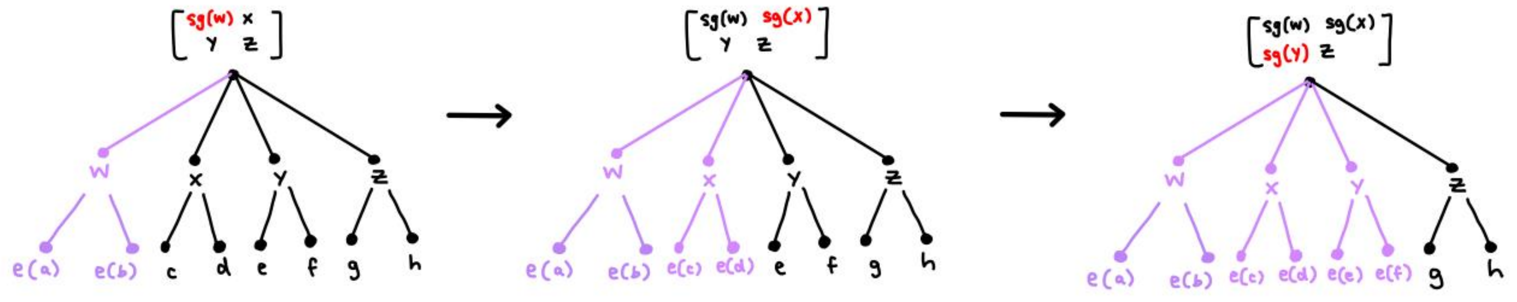
Let a, b, …, g, h, w, x, y, z be game states. Each branch represents different pairs of player moves resulting in the next state.



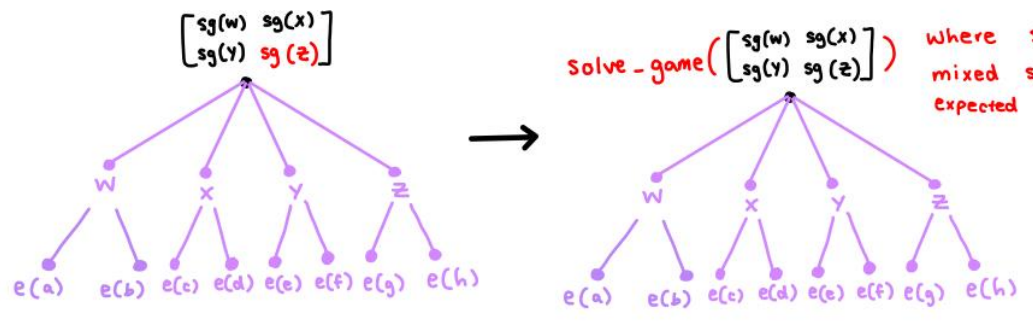
$\begin{bmatrix} w & x \\ y & z \end{bmatrix}$
root
leaf nodes
w x y z
a b c d e f g h
e(a) e(b) c d e f g h
sg(w) = Solve_game($\begin{bmatrix} e(a) \\ e(b) \end{bmatrix}$)

where e() is the evaluation function.

where solve_game(v) yields the guaranteed minimum expected value of the equilibrium mixed strategy.



$\begin{bmatrix} sg(w) & x \\ y & z \end{bmatrix}$
$\begin{bmatrix} sg(w) & sg(x) \\ y & z \end{bmatrix}$
$\begin{bmatrix} sg(w) & sg(x) \\ sg(y) & z \end{bmatrix}$
w x y z
e(a) e(b) c d e f g h
e(a) e(b) e(c) e(d) e f g h
e(a) e(b) e(c) e(d) e(e) e(f) g h



$\begin{bmatrix} sg(w) & sg(x) \\ sg(y) & sg(z) \end{bmatrix}$
solve_game($\begin{bmatrix} sg(w) & sg(x) \\ sg(y) & sg(z) \end{bmatrix}$)
w x y z
e(a) e(b) e(c) e(d) e(e) e(f) e(g) e(h)

Where solve game yields a probability vector: an equilibrium mixed strategy over the rows ensuring a guaranteed minimum expected value.

$\Downarrow$

Use this result to determine our next action.