COMP30024 Artificial Intelligence Project B Report

Team Name: Admin
Team Member: Yifeng Pan (955797) & Ziqi Jia (693241)

1. Overview

The project began with considering the approaches of rule-based algorithm, game-theory based minimax algorithm, and machine learning approaches such as model-free Q learning approach, and finished the implementation of two algorithms, one being a complicated variant of game-theory based minimax algorithm and the other being a rule-based greedy algorithm that depends on set. The project was done in parallel by the team members. The greedy algorithm required less effort and therefore was finished first and was later put to testing and optimizing the more complicated game-theory based algorithm.

Through extensive tuning and optimizing, the final model we chose is the game-theory based algorithm since it turns out to have a more balanced performance than the greedy algorithm.

2. Unimplemented Algorithms

2.1 Supervised learning

Supervised learning was considered to be implemented for the project since the algorithm could learn from the training cases and produce an inferred function that summarize the rules of taking which move that could lead to a victory. Since each state (turn) of the game is largely dependent on the previous state of the game and combined with the zero-sum characteristics of the game, we regarded any model that assumes the independence of attributes would not be appropriate for the task.

The implementation we conceptualized for a generic supervised learning model is to treat **Win/Lose, number of tokens on board, number of enemies, closest target distance** (closest distance from a friendly token to an opponent's vulnerable token), **closest threat distance** (the closest distance from a friendly vulnerable token to an opponent's token), and **friendly/opponent's remaining throws** as the features, and labelling opponent's action with four categories of **offensive throw, defensive throw, evasive move** and **aggressive move**, training the model and hence using the model to predict the opponent's next decision. However, with such attributes, we could already imagine that many of entries would have the same values, or even same combinations would have different actions outcomes, making the problem a hard one for the model to learn. And besides that, the attributes could not accurately reflect the real situation on the board since it is missing the actual tokens' positions which should truly be the most decisive factor influencing the next decision. But however, this feature is hard to be made into a machine learning attribute since not only that both the number of tokens on board and their types and position are undetermined and changes constantly throughout the game, but also each token is not unique (meaning that a token representing paper that has been removed previously could not be distinguished from another token of the same type added later in the game). Besides, the decision category needed to be manually labelled by the team, and many moves cannot be clearly identified into the four categories stated above (for example, a token that is evading from an opponent's chasing token in the direction towards its nearest target).

Therefore, with the above considerations of challenges and drawbacks, the supervised machine learning method was put out of our consideration. But still, the model-free machine learning method looks promising, our next choice was to try to fit a model-free learning method, the Q learning.

2.2 Reinforcement learning

The Q learning is a type of model-free reinforced learning method. Since the method does not assume a model of the environment it is implemented (the model-free aspect), it looks promising for this task. The model follows the below function:

$$Q(action, reward) = (1 - \alpha)Q(action, reward) + \alpha(reward + \gamma \, max\,(next\ state, all\ actions))$$

The hyperparameters, α value, is the learning value set between 0 and 1 which decides the extent of the update of the old Q value, while γ determines the importance of the future rewards. The idea of the algorithm is to through continuously exploring the state space through available actions, learn the rewards of each action and update the Q value according to the above equation. The learned Q values are stored in a Q value table, with columns being all the available actions and rows being all the possible states. After the model is trained, when

encounter a state, the model will look up the table and select an action that has the maximum reward and hence choose that action.

However, as for the project, the actions available for a single token will have maximum of 6 slide actions in all directions, and if there are nearby tokens, the action space could be extended to maximum of 12 swing actions in all directions. If only taking account of one token's respective actions and treat throw action as one single action, the Q table will need to have $6 + 12 + 1 = 19$ columns. As for the rows of the table, the game states, there are 61 slots a single token could be in, giving a size of 61 rows for the Q table. The resulted table size for a single token would be a 61×19 matrix. Such approach would be effective if the game is relatively static, however, the opponent would be making moves at the same time, making the last learnt Q table obsolete. If the algorithm needs to be effective for solving the game, the agent needs to learn the given layout again for every single turn and for every single token on board. This will seriously undermine the performance of the algorithm, and thus it may not be a feasible approach for this project.

The above is not the only issue with this algorithm we have considered. How to set the reward was also a challenge for this algorithm. In this game, the ability to perform evasive moves are also as important as the attack moves. If only giving reward upon the state of an enemy being eliminated, the algorithm might deteriorate into a greedy approach, and if also assigning rewards for each evasive move, the reward might come too easily and confuse the algorithm of what will truly end the game. Hence, the Q learning was decided to be put aside as an optional task if there were extra time left, ended up not being implemented.

3. Implemented Algorithms

Our team members worked in parallel and developed two algorithms, one being a simpler but effective model based on simple evaluation function and preset conditions (we call it the greedy approach), the other being a more complicated minimax-variant algorithm that utilizes a complex evaluation function developed on the prior's basis and solve the problem with payoff matrix. The latter algorithm also absorbed the features of certain moves under some preset conditions to help improve the problem-solving efficiency.

3.1 Greedy algorithm

3.1.1 Search algorithm

The search algorithm used for this agent is based on the A* search done in Part A. However, the search problem is relatively simplified in this part, and the features of considers blocks, avoiding friendlies to reduce the probability of failing the game due to an eliminated friendly token and exiting from dead-ends is no longer needed. Therefore, the search function has been simplified to improve the efficiency.

3.1.2 Evaluation function

As this is the first algorithm implemented, the evaluation function h(x) considers simpler aspects: the distance between the position of a token and its target piece, the distance between the position of a token and the position of the closest enemy. The idea is to generate a value through the two features and maximize the value. The heuristics function is structured as the following:

$$h(x) = \frac{1}{closest\_dist\_to\_target} - \frac{1}{closest\_dist\_to\_threat}$$

We want the distance from a token to its target to be as small as possible, making the first term a growing term, and distance from a token to its threat to be as big as possible, hence a diminishing second term. The distance is inversed to avoid the situation that if a token gets closer to goal as well as the threat, the evaluation function returns the same value and could not distinguish between these two situations (e.g., 4 moves to target and 3 moves to threat. If no inverse, the value would be 5-3=2 and for a move that minimizes both distances, the score would be 4-2 =2, being the same). Since the reciprocal of each distance is used in this function, it is possible to have a denominator of 0 (when two tokens overlap). When this situation appears, the denominator will be set to 0.1 (i.e., when *closest_target_distance* or *closest_beat_it_able_distance* = 0, we adjust each value to 0.1) to give a big bonus without causing a calculation error. If the overlapping tokens have the same symbol, it does not affect the calculation. The overall goal is to maximize the value generated from the equation.

3.1.3 Implementation

We firstly calculate the value for the current position of the token according to the heuristics, then the value for the position after taking an action. During each turn of the game, we put all our possible actions into a heap and use the heuristic function to value each action, returning the action with the largest value after searching for all the actions, so that we can maximize the total output by maximizing the output of each turn. The 'greedy' aspect of this algorithm is that it disregards the information asymmetry issue by ignoring the opponent's actions and treat each turn as a static game.

The advantage of this feature is that the time cost is quite low as the time complexity of each turn is O(n) since we are only considering our n tokens on board. However, the drawback is also obvious. The optimal value by the heuristics might not truly be the best move combining with the opponent's actions. Therefore, a few rules were set on the actions to avoid it being to balance out the drawbacks. The game was categorized into three scenarios, and under different scenario the algorithm will have a different preferred action.

Open Game Scenario:

At the beginning of the game, a random piece is thrown among all the allowed positions for no computation is required for an empty board.

Mid-Game Scenario:

During the mid-game, we set preferences for different actions. If no preference was set for the player's action, the player will keep throwing until the number of throws is exhausted. The considerations taken into the throw action are whether there is an opponent's token in the range that we can throw, assuming that the opponent's token is beaten upon the throw action, and then calculate the value of this new token with our evaluation function. However, if the opponent's token has a token nearby that can protect it (use a variable called *beat_it_able* to mark it), or if the protectable token can slide or swing towards opponent's token within the throw range, position will not be chosen. If there is no opponent's token within the throw range, the algorithm will try calculating the value if perform throw within the range. However, the preferences for the throw action are set to be low as we need to be careful of quickly exhausting all the throw moves, which will likely lead to a defeat. For example, if a throw action and a move/slide action results in the same optimal, the algorithm will go with the slide/swing decision instead of the throw action. And if a slide action and a swing action results in the same optimal, the algorithm will choose the swing decision.

Endgame Scenario:

This scenario handles situation when all the throws are used up. We will just keep using our evaluate function to evaluate each action and pick the action with the highest value until the game ends.

3.1.4 Performance evaluation:

The performance is evaluated from three perspective, whether the agent wins the game, how many turns it takes to finish and time it needs to finish. We have made some optimizations to the algorithm: Optimized the data structure: instead of using the entire board to represent tokens, we categorize the tokens into different symbols and store them by using a dictionary list so that any access to token information can be done in the shortest possible time; Applied pruning to cut out unnecessary nodes, so that the algorithm will not explore the unpromising positions.

The algorithm turns out to be a highly efficient agent. Among the 10 trial tests against random player on the battleground, the greedy wins all of the trials, with an average turn of 37.5 to finish, and the average computation time used is 0.0125s when testing on the local machine, which is impressively low. Similarly, among the 10 trials against the greedy opponent on the battleground, our algorithm wins all of the trials, with an average turn of 42.8 to finish, and the average computation time for a game is around 0.008s. The trials against the random player prove that the evaluation function is effective, and the victory battles against an actual greedy opponent suggests that the preset conditions of the algorithm has prevented our agent to deteriorate into a truly greedy algorithm. In both cases, the computation time is amazing low, meaning the algorithm would be time efficient, which just as the upper O(n) bound predicted above.

But tests against random and greedy player may not suggest that the agent is intelligent. Therefore, a sequence of 5 trials is conducted against other students' agents. Among the five trials, the result was 1 victory, 1 defeat, and 3 draws due to repetitive state. The performance is reasonably acceptable. From the results of the trials, one issue reflected is that the algorithm, due to preferences and preset conditions, is set to be too conservative and does not have intelligent behaviors such as sacrifice tokens to victory.

Overall, the algorithm is a good balance between performance and efficiency, but it does not address the game's simultaneous play characteristics, which might be a drawback when facing intelligent agents that uses algorithm that addresses this. The algorithm is used for developing the latter model.

3.2 Minimax-variant algorithm

The minimax algorithm is a good approach for a game of perfect information. However, the problem we tried to solve is not necessary the case: Although the agent and the opponent has the knowledge of what the current layout of the board is, they do not have the knowledge of what each other's action will be due to simultaneous play. If translated to a turn-based game, which player take turns to act, this would be that the agent would only have the knowledge of opponent's action two turns ago, not their latest action. Besides, although with alpha-beta tuning and cut-off techniques, the search tree in the original algorithm can be reduced to a relatively small size. But for this problem, as for the big branching factor which consists of all the actions player can take, the tree would grow so rapid as the number of tokens on the board increases that the memory would be drained. For example, even if the current layout is 1 friendly token and 1 opponent token, and we set the cut-off depth to be 2 turns away, if at any position both tokens have 6 positions to move, with the throw action player could perform, we would still have a tree of size $7^4 = 2041$ possibilities. Although with alpha-beta pruning we could avoid expanding some tree branches, this size is still too enormous. Therefore, modifications to the original minimax algorithm are needed to make it feasible for this problem. The idea of the minimax is retained, but the actual implementation is done through a payoff matrix, and the simultaneous play is addressed with game theory.

3.2.1 Evaluation function and payoff matrix

3.2.1.1 Evaluation function

To realize the payoff matrix, we would need to have values representing the state of the board. Simple setting such as 0 for nothing, 1 for defeating one enemy and -1 for losing one token is too generic since during most of the turns, there would be no battling happening and will cause the matrix to be a matrix of 0s which the algorithm could not decide the best move from. In the prior algorithm, the heuristics function is set to evaluate the difference between distance

between the position of a token and its target piece and the distance between the position of a token and the position of the closest enemy. Although it turns out to be effective, we would like to know if there will be improvements by a more comprehensive evaluation function. We took the idea of the evaluation function $EVAL = \sum weight \times f(s)$ of a chess game and modified it to this problem. The basic evaluation function for n tokens on the board is:

$$EVAL(board) = winning\_coef \times penal\_coef \times overlap\_adjustment\_coef \times \sum_{i=1}^{n} w_{t_i} \times f(t_i)$$

, where w_{t_i} is the weight of a single friendly token and $f(t_i)$ is the evaluated score of it given by the following.

$$f(t_i) = \frac{(enemy\_throws\_left + number\_of\_enemies\_on\_board + 1)}{(enemy\_throws\_left + 1) \times num\_of\_friendlies\_exposed\_to\_enemy\_throw} \times \frac{closest\_dist\_to\_threat}{num\_of\_enemies\_t_i\_exposed\_to \times conservation\_coef} - closest\_dist\_to\_threat \times num\_of\_enemies\_t_i\_exposed\_to$$

The desirable situation is for an individual token to be as far away to the threat as possible, while not exposing to opponent throws and being as close to the target as possible. In this case, the prior term in the evaluation function would grow large as the second term diminishes, maximizing the f value. Hence, the larger the value, the more desirable the current situation is. When no tokens are exposed to enemy throw, we reward to value by setting num_of_friendlies_exposed_to_enemy_throw to 0.5, rewarding the function by doubling the final score, and penalizes the function by add 1 to the variable for every token exposed. However, even if the token is in the throw range, there is a chance that enemy might not throw since the opponent might have better action for other tokens or might not throw to this token due to multiple tokens are exposed. We regard the probability of opponent perform throw action to be $\frac{enemy\_throws\_left}{enemy\_throws\_left + number\_of\_enemies\_on\_board}$, by this we are also assuming the enemy values "Throw" action. As the number of throws decreases to 0, the chance of throwing also decreases to 0. If we have multiple tokens in the throw range, then mathematically the probability would need to be multiplied by how many tokens are within throw range, however we need to be careful to set this behavior in the eval function since the algorithm might interpret it as a 'reward' for exposing more tokens in opponent's throw range. Therefore, in the final model we decided to penalize the score for more tokens entering the throw range, with the inverse the probability expression, deriving the part $\frac{(enemy\_throws\_left + number\_of\_enemies\_on\_board + 1)}{(enemy\_throws\_left + 1) \times num\_of\_friendlies\_exposed\_to\_enemy\_throw}$. In the function we also penalize the algorithm for letting tokens exposed to nearby threats even it is neighboring its target under current circumstance, therefore divide the $num\_of\_enemies\_t_i\_exposed\_to$ to the first term and multiply it to the second term, reducing the score. The conservation coefficient was put to determine the level of aggressiveness, larger coefficient would reduce the aggressiveness of the algorithm.

But it would not be reasonable to assume every token has the same importance, for example, for a token with no threat and target only does not share the same importance of another token that has both threat and target, since for the first one nothing can eliminate it and we could move this token later while the second has a chance of being eliminated and result in a defeat. This is why we assign the w_{t_i} to each token. The default weight is set to 5, and we deduct 1 for not having targets, 1 for not having threats, 1 for it is covering friendly and being covered, and add 1 for having 1 target, multiply by 5 for immediate danger (nearby threats)

Some layouts of the board are not in favor to us and would easily lead to a defeat, therefore we penalize the final score for having such layouts by multiplying penal coefficients. We would not like multiple tokens of the same type to be stacked on the board, since enemy might eliminate multiple friendly tokens by one throw, thus we multiple an overlapping adjustment coefficient to penalize the score for every such situation occurs. Also, if there are opponent tokens that we cannot eliminate, we penalize the score by multiplying the penal coefficient and grant the algorithm an extra reward for performing throw action.

Besides all the contents above, we also set a simple measure of winning progress by taking the difference between friendly remaining throw and opponent remaining throw. If we have more throws than the opponent, we multiple the difference to the final score to reward the function, and else we divide the difference to penalize the score.

3.2.1.2 Payoff matrix

With the evaluation function ready, we could proceed to generating the evaluation function. The generation of a payoff matrix is to take all combinations of available moves from both sides (for example, my throw action "A" and opponent slide action "B") and hypothetically update the board with these moves, generating an if-score for the combination of (e.g., for combination of A and B, update the board hypothetically and get a score of x). The result matrix P will be of $M \times N$, where M is the number of my available moves on rows and N is the number of opponent's available moves, the entry P_{ij} is the score if I take the i^{th} move from the available move set and the opponent take the j^{th} move from his available move set. Then, we use the minimax idea. For each row/action, we look up the minimum score we get from all possible combinations of opponent moves with this action and record the value. We compare the minimum value across all my possible actions, and we choose the action with the largest value, meaning that although we do not know what the opponent's move will be, but for this move, even in the worst case, is better than every other

move in my action list. This approach can be viewed as a minimax algorithm of cut-off depth of 1.

3.2.1.3 Optimization of the payoff matrix

As the matrix size is of $M \times N$, it would be very inefficient to generate such a matrix every turn and look up the desired value. But given we have little knowledge on the opponent's action, it would be unwise to exclude any opponent actions from their action set, meaning reducing the size N might cause the algorithm to miss important information. But however, through reducing the size of our potential moves and hence the size of M, we could effectively improve the algorithm's performance. The idea is to limit our action space to be O(n) bound, and with more tokens our action space only grows linearly. Therefore, our approach is to categorize the moves under different behaviors, for example, attack move, defensive move, aggressive throw move, (the actual category is more detailed than this) only add the actions that can best achieve the move purpose to the action set. And with such reduction, for n friendly tokens, on the board, instead of having an action space of $O(k^n)$ where k is the branching factor, we could reduce it to $O(\alpha n)$ where α is the number of movement categories we selected.

3.2.2 Implementation and optimization

The nature of such payoff matrix approach is that it will pick the "least good" action from your action set instead of choosing the truly good moves, for example, it would ignore a bold decisive attack move that has a chance of losing a token, preferring the moves where it can safely attack. In the initial experiment with the algorithm, we found that using the algorithm on its own would result in a too conservative agent that could almost never end the game unless the opponent is a super aggressive one. Therefore, to compensate for this downside, we did similar things as to the first algorithm, categorizing the scenarios and moves, and we decide under what scenario which set of actions are fed to the algorithm and ask it to decide the best among our chosen moves.

3.2.2.1 Scenario and move categorization.

The categories we set for the actions are: **Offensive move** which returns the best action to the token's nearest target; **Defensive move** which the token evades from an incoming threat, which is further divided into **running for cover move** which a token moves towards a friendly token that can protect it, **purely evasive move** which it simple increases the distance to its threat, and **evasive attack** which the token moves away from the nearest threat while moves towards its target; **Covering move** which it moves towards a friendly vulnerable token to protect it, or with a potential threat to a friendly token nearby, move onto the friendly token that result in elimination of all three tokens; **Offensive throw** which the agent tries to throw directly onto or near the enemy as close as possible; And **protective throw** which the agent tries to throw a token near a vulnerable friendly token, or onto a friendly token that have a threat nearby and ready to attack it, causes elimination of three tokens.

We also ask the algorithm to perform in certain ways under these scenarios:

Open Game: When there are no friendly or enemy token. Randomly throw a token onto the board within the throw range.

Mid Game: This should last for the most turns of the game. Here we set 3 possible scenarios: 1. When there is no enemy token on the board, then no throw action is necessary for withholding the throw action gives more advantage, and no aggressive action is needed to be fed to the matrix, hence in this scenario only the pure defensive and evasive actions would be considered; 2. When there are no friendly tokens on the board, then we only feed all sets of throw actions to the matrix and return the best action; 3. Any other situations during mid game we would proceed through the normal payoff matrix procedure with all categories of actions fed to it.

End Game: This state is entered as one of the players exhausts the throw actions. The scenarios are: 1. The opponent drains the throw actions first, it means that all the tokens visible to us are its last tokens, and we do not need to hold the throw actions. Therefore, perform attacking throws that benefits the most until we do not have throws left. 2. When both players throws are emptied, then we could assume this might be close to the ending of a game. Therefore, to end the game fast, we deteriorate this algorithm to the greedy approach above; 3. When the player does not have any throws left and the remaining token on the board is less than 3. And if the player has less than 1 paper 1 rock and 1 scissor on the board, this would mean that continue to be aggressive might be very risky. We would check losing which token might cause a defeat, give up all the aggressive moves and only perform evasive action for that token and saving it until a draw happens.

3.2.2.2 Weight assignment

With the above categorizations, the algorithm was tested much better than the original one. However, we still found that it loses a lot of games. We investigated some cases and found that during normal mid games the agent used up throw actions too soon. Therefore, we decide to implement some weight to each move and result. The final if-score for the matrix is changed according to following:

$$final\_score = original\_if\_score \times penalty\_coef$$

where the penalty is initialized as 1 and changed according to the following rules:

1. If the action is a "THROW", then multiply the penalty by $\frac{1}{8}$; However, if the opponent does not have throws left, we encourage throw actions and set the penalty to 2 for extra reward; If we already possess 1 paper, 1 rock, 1 scissor on the board, we dislike throw actions and set the penalty to 0.01.

2. If a move action on both sides would cause to lose tokens, for every token lost we divide the penalty for 3 for every friendly token is lost and multiply by 4 for every opponent defeated. If enemy is defeating its own tokens, we multiply the penalty for this is less likely to happen.

Only the final score is passed on to the payoff matrix to help determine the optimal actions.

3.2.3 Performance evaluation

We follow the same evaluation rules as assessing the first model, and using win/lose, turns to finished and computation time as our standards. With the final finished model, among the 10 trials against a random player, the agent won all the games with an average turn of 32.2 and computation time of 6.23s. The turn it takes to finish a game is reduced slightly compared to the previous greedy model, however the computation time is significantly increased as the size of processing the payoff matrix.

The ten trials against the greedy opponent shows that, there indeed are some slight improvements seen on this agent compared the previous one, at the cost of computation time. The minimax approach won all ten trials against the greedy opponent, with an average turn of 36.4 to finish and the mean computation time of 5.67s. Compared to the results of greedy algorithm, we saw a reduction of around 7 turns to finish the game. From the above trials, we have seen this model, while gaining some insights into looking at the opponent's moves, retained the reasonable efficiency from the previous model for solving a game. To test whether this algorithm is also an effective one against another adversarial search algorithm, we testes 5 trials of minimax versus the previous greedy agent. The minimax won two of the five trials and achieved a draw due to invincible tokens on both sides. Among the two victories, it took 31.5 turns on average to win.

In conclusion, although a victory is not guaranteed, the probability for winning a game with the minimax-variant is higher against the previous algorithm we developed, although the improvement comes with the trade-off on computation time and memory usage (storing the payoff matrix).

4. Conclusion and possible future insights

In this project, we explored four possible approaches to solve the game, including generic machine learning, reinforcement learning, greedy approach integrated with rules, and the minimax-variant algorithm, and implemented the last two models. Both of them are able to solve games against random and greedy opponent with consistency and shows reasonable efficiency in the turn time and computation time for finishing the game. And when playing against another adversarial search agent, from the limited trials, the minimax-algorithm shows a better chance of winning. However, there are downsides of this algorithm. With the minimax-variant, it only looks at the actions that are results in the least cost, which assumes the opponent is smart enough to take the 'best' action, while missing the true best action. We did not actually implement a model that could predict the opponent's behavior like a human agent would do when playing similar games. One possible approach is that based on the minimax-variant model we have add a feature that collects the past opponent's behavior and calculate the probability of it taking an aggressive /defensive/throw move. Then in the payoff matrix, we could assign a higher coefficient (in favor to us) to the combination with the opponents moves that are not so likely they would take, and a lower coefficient for the moves they are likely to take (not in favor) and use this adjusted final value to get the final action.

5. Appendix

5.1 Table of test results

Restricted Greedy Approach vs Greedy			
Trial	Turns to finish	Computation time	Result
1	84	0.016	Win
2	30	0	Win
3	46	0	Win
4	36	0	Win
5	44	0.016	Win
6	26	0.016	Win
7	49	0.016	Win
8	38	0.016	Win
9	41	0	Win
10	34	0	Win
Average	42.8	0.008	

Restricted Greedy Approach vs Random			
Trial	Turns to finish	Computation time	Result
1	46	0.031	Win
2	41	0.031	Win
3	37	0	Win
4	34	0.031	Win
5	23	0	Win
6	61	0.016	Win
7	37	0	Win
8	38	0	Win
9	32	0.016	Win
10	26	0	Win
Average	37.5	0.0125	

Modified Minimax Approach vs Random			
Trial	Turns to finish	Computation time	Result
1	28	5.375	Win
2	22	1.047	Win
3	30	6.234	Win
4	57	10.031	Win
5	24	3.019	Win
6	35	6.922	Win
7	29	8.219	Win
8	34	5.141	Win
9	41	10.953	Win
10	22	5.359	Win
Average	32.2	6.23	

Modified Minimax Approach vs Greedy			
Trial	Turns to finish	Computation time	Result
1	25	3.281	Win
2	27	5.406	Win
3	58	6.406	Win
4	33	5.906	Win
5	23	1.047	Win
6	35	4.328	Win
7	60	6.969	Win
8	47	7.453	Win
9	34	9.75	Win
10	22	6.156	Win
Average	36.4	5.6702	

Restricted Greedy Approach vs Another online agent			
Trial	Turns to finish	Computation time	Result
1	53	0.016	Win
2	78	0.031	Draw (Repetitive)
3	45	0.016	Lose
4	22	0.031	Draw (Repetitive)
5	12	0	Draw (Repetitive)

Modified Minimax Approach vs Greedy			
Trial	Turns to finish	Computation time	Result
1	24	5.109	Win
2	58	2.656	Draw (Invincible)
3	22	3.703	Draw (Invincible)
4	55	7.125	Draw (Invincible)
5	39	4.781	Win