COMP30024 Artificial Intelligence 2019 S1

Project B – Chexers

Haichao Song, Haolin Zhou

Introduction

To develop an agent with higher winning rate, we've tried machine learning and various algorithms. We finally adopted the Max^n algorithm since it is the best choice according to time and memory constraints. The data structure to store the states has been improved thus the tree can expand deeper in limited time.

Structure of Program

Before we finalized our agent, some agents were designed to test and modify. These agents can be list as follow:

- **Random Agent**

An agent that randomly selects the valid action. It was created to modelling basic functions of the game. This agent does not use any algorithm which causes it to be the fastest but also the weakest agent.

- **Greedy Agent**

An agent that select the action closest to the destination in all available actions. When there are opponent pieces around it which can be eaten by it, the agent will have the priority to eat the pieces. The greedy agent can win the random agent under all circumstances. If we compare it with the Max^n agent, it will be the agent only have one depth and features of eating and going to the destination and exit.

- **team_404/Paranoid.py (Paranoid Agent)**

An agent implements the paranoid algorithm. This agent assumes all opponents are our enemies and minimize our value, and our consider maximize our own value.

After testing and trying these agents repeatedly, the results lead us to our applied agent which implements the Max^n algorithm.

- **$\mathbf{Max^n}$ Agent for team_404**

An agent implements the Max^n algorithm. The agent assumes that all players in the game would like to maximize their own benefits in order to win the game, and we will consider other players pick the actions to get most benefits then we choose the action to get the most benefits for ourselves among them.

- **team_404/State.py**

This class generates the features of a board state and contains a data structure. It has essential information in the board including each player's pieces, their destinations, their exit pieces number, the latest past action and the turns in the game. It also includes some basic utilities like the function that can get the next available states of the current state, check if one piece is on the board and find the middle piece when jump action is taken.

- **team_404/Evaluation.py**

This class represents the evaluation function used by Max^n agent. It contains different features and the weights of features. Each state of the board will be evaluated by the combination of different evaluation functions in this class.

- **team_404/Maxn.py**

The implementation of the Max^n algorithm. We also applied lazy evaluation In order to improve the performance of the agent. It has depths three to consider all players action in one round.

Machine Learning

There were several attempts of machine learning to train the weights of our features. Since there is a significant amount of states and learning needs millions of games to improve performance, it is difficult to train out a better accuracy of the perfect weight in such a short time.

- **Q – learning**

Q - learning is an algorithm of reinforcement learning which will try to learn the value of a specific action base on a given state. Q - Learning will be presented in the form of a table. We will get a Q value table after training. The meaning of each specific value is the long-term expected reward of taking this action in this state. We initialized from a consistent Q value table to start and then updates the corresponding value of Q values in the table according to the return on the various actions we observe. Bellman function was used in Q – learning:

$$Q(s, a) = r + \gamma(max(Q(s', a')))$$

However, Q-learning takes longer time than we expected and hard to generate good results before deadline.

- **Genetic learning**

The genetic algorithm will solve the problem need to be solved as a process of biological evolution. A group of players is first sown by a random feature weighting vector. Then we generate next-generation players through operations such as copying, crossover, mutation, and gradually eliminate players with low scores and increase players with high scores. After the evolution of the N generations, it is very likely that an individual with a higher winning rate will evolve.

Our well-trained weight vector is not optimal because we cannot train until significant convergence is achieved. Compared to Q-learning and neural network, it does not perform well in board game since the decision of mutation and parts of crossover is hard to be distinguished. Many features are highly related, and most generations are useless so that the process will be inefficient.

- **Neural network**

We use eigenvectors as input to the neural network. We have selected several features. By defining these features, we can obtain eigenvectors by traversing the game. TensorFlow library can be used to iteratively train the neural network model. This is the most efficient way in Machine Learning but due to the time, space and ability limitation, the training was no succeeded.

- **Limitation**

As a three-player game, there are a lot of possible states. And the hexagonal board with MOVE and JUMP actions also creates a large number of possibilities. It is hard to generate a good evaluation function or make decisions due to a single state.

Effectiveness of Different Programs

For Multi-player games, there are some algorithms can against opponents. These algorithms are all derived from the minimax algorithm which only fit for 2-player games. In this game, greedy is also an algorithm that worth mentioning. Two main algorithms we considered is Max^n and paranoid.

- **$\mathbf{Max^n}$ Algorithm**

Max^n algorithm can take the decision of three players in consideration, rather than simply treating both players as enemies. The consideration of Max^n algorithm will be more comprehensive. But the pruning of Max^n algorithm is very inefficient. Max^n has two pruning methods, shallow pruning and lazy evaluation. Shallow pruning has no obvious effect, and lazy evaluation can save a lot of time. In this game, compared to paranoid algorithm, the Max^n algorithm is more fit for normal situations so that make more realistic actions.

- **Paranoid Algorithm**

The paranoid algorithm inherits the pruning method of the minimax algorithm so it can search deeper depth than Max^n algorithm within the same time. J. (Pim) A. M. Nijssen and Mark H. M. Winands claim that the paranoid algorithm treats all players except themselves as enemies, regardless of the league (2012). This also led to the program being more conservative in the game. In our experiments, although the paranoid algorithm can explore deeper, its winning rate is not as high as Max^n algorithm.

Approach of the Game

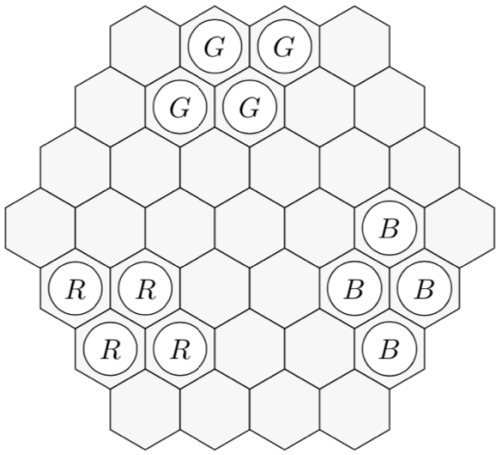
As mentioned above, we implemented the Max^n algorithm as our final agent. Max^n will consider 2 opponents' choice respectively rather than regard them as one enemy. The depth of the Max^n tree is 3 which means we can simulate the situation after one round. Within a limited amount of time, the depth of 3 is the most suitable.

To improve the time complexity, we apply the lazy evaluation. Lazy evaluation only calculates the values of players on the considered situations which saved time for calculating other players. What's more, we apply a dictionary to store the different color of pieces which save time for each evaluation calculation since the time needed to getting value in a dictionary can be ignored.

To make the opponent's evaluation function realistic, we only take the eat actions and the actions toward destinations into consideration since these two kinds of actions have a higher probability to be chosen.The evaluation function is the most important part of this program. The winning rate depends on the features of the evaluation function. The Max^n agent can even lose to the greedy agent with a less considerable evaluation function. But this doesn't mean that the evaluation function is as good as the more of features are.

For all the following diagrams, we consider red is our player and the other two are opponents. We use red pieces to explain features and their weight.

- **Fixed start**



(Figure 1)

Based on the investigation of various algorithms and the observation of the board, we have come up with an optimized startup mode, which is to let the pieces move forward together. This prevents us from losing our pieces at the beginning, and this state is also the closest to the end. This startup pattern has been stored in a dictionary, and we can make the best choices without consuming time. After this processing, our Max^n algorithm can get more efficient to traverse the states of the board. However, we can only set up the first few steps, the latter states of 3 players are varied and can only be given to the Max^n algorithm to make a better decision.

- **Evaluation Function**

The factors of the evaluation function can be listed as follow:

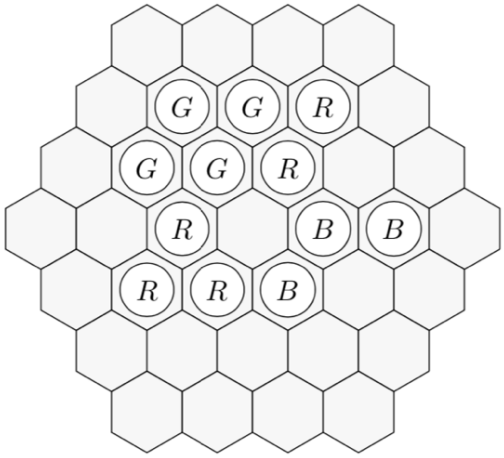
- **EAT**

To represent the eat feature, we used eat = # of current player's pieces + # of current player's exit pieces. We would like to have as many pieces as we could. However, if we can win the game by exiting, eating will waste turns.

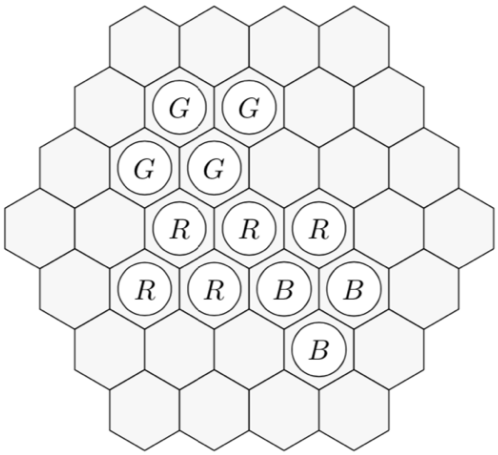
The weight of eating feature has been set as 100. Eat is an important action since the situation will become more optimistic because of eating. The range of this feature is

between 0 to 1200 since the maximum number of pieces could have is 12 and the worst-case scenario is having 0 pieces.

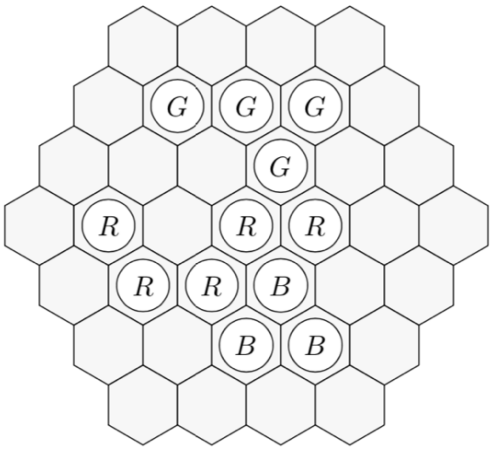
- **DANGER_COLOUR AND DANGER_PIECES**



(Figure 2)



(Figure 3)



(Figure 4)

If a piece has opponent pieces around and it can jump through our piece, this piece is dangerous. This feature is very important to protect our pieces from eating and distinguish from a simple greedy algorithm.

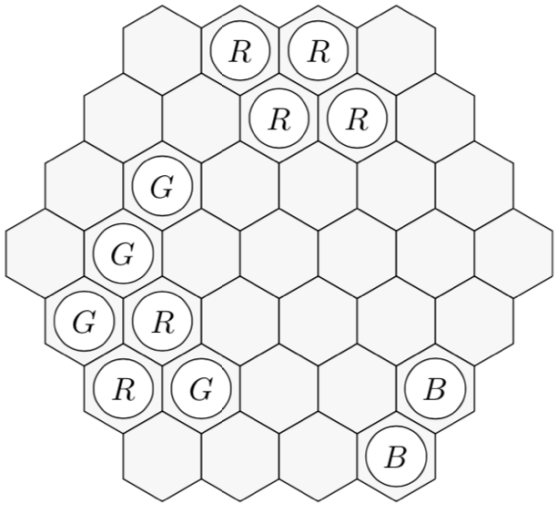
The DANGER_COLOUR value returns the number of opponents our pieces are exposed to. The worst situation is that we expose our pieces to two opponents in the same turn. This factor has a maximum number of 2 and minimum number 0. The situation in Figure 4 diagram below needs to be avoided since we might lose 2 pieces directly in the same turn. However, the situation represented by Figure 2 and Figure 3 needs to be considered since it is worthy to gain one piece and taking the risk with one piece in danger.

The DANGER_PIECES value returns the number of pieces we exposed to opponents. Under similar situations of eating and moving, we need to expose fewer pieces to the opponent as possible. The factor has a maximum number to the pieces we have on the field and a minimum number of 0. In Figure 2, the danger pieces we exposed is 2 and in Figure 3 is 1.

The weight of DANGER_COLOUR has been set to 60 because its priority is lower than EAT. The range of DANGER would be 0 to 160. If the situation is that we can trade one for one, the evaluation value for this state will be 100(EAT) – 60(DANGER_COLOUR) = 40, then we choose to take the action. However, if eating will expose our pieces to two opponents, we are taking the risk of losing two pieces in one turn, which is not worthy, and the value for this state will be 100(EAT) – 120 (DANGER_COLOUR) = -20, then we consider not to take the action.

The weight of DANGER_PIECE is 5 which is lower to both EATING AND DANGER_COLOUR. At similar situations, we will consider protecting pieces but eating has higher priority. When we exposed two pieces to the same opponent in the same round, he could still only eat one of them so it is not that influential than the previous features.

- EXIT



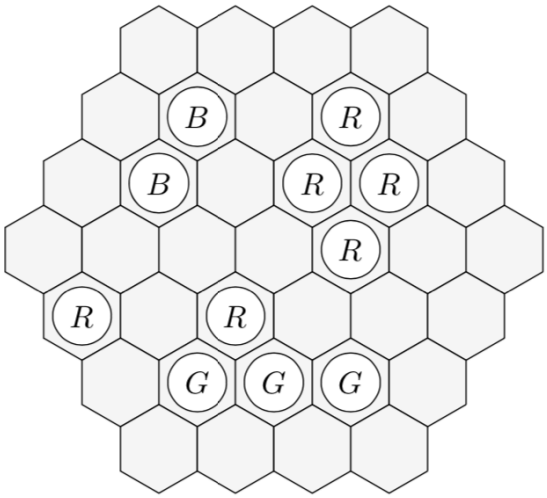
(Figure 5)

When a piece can exit without influence of others. In this situation, although the red player can eat the green piece, he could exit 4 pieces directly to win the game (Figure 5). Therefore, eating might waste rounds and lose games.

There is a situation that when all the pieces exit but still lose the game since the exit pieces are lower than four, then the current pieces will not exit anyway. The Exit value is -1 when the player couldn't win the game with the exit actions of all pieces on the board, otherwise the EXIT value would be the number of pieces that have already exited. Before we adjust -1 of EXIT value into the evaluation function, the program simulated interesting actions such as exit 3 pieces and lose the game directly.

The weight of EXIT is 70. Therefore, we only consider eating prior to exiting when we eat others without taking any risks. Eating exposed our pieces to enemies will have value 100 (EAT) – 60 (DANGER_COLOUR) = 40 which is smaller than 70 and we will consider exiting prior to this situation.

- DIST

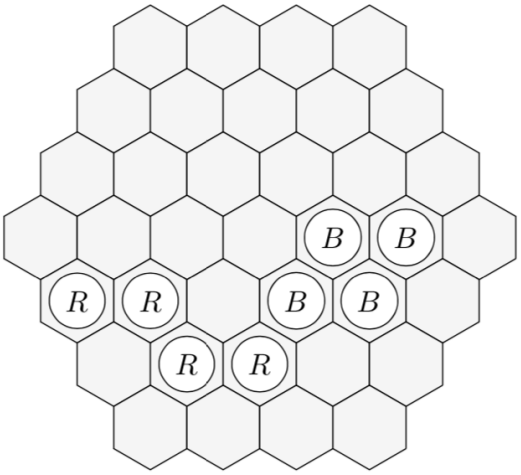


(Figure 6)

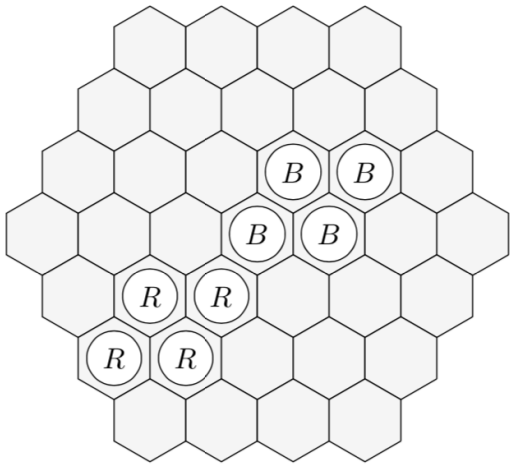
This feature is represented by the average distance to the destination for the closest four pieces. If the player has equal to or less than 4 pieces, the DIST value would be the average distance of all pieces. If there are 4 exit pieces or we lose all pieces, the EXIT value would be 0. If the player has more than four pieces, the EXIT value would be represented by the average distance of the closest four pieces.

In the premise of eating and bound. We need to move 4 pieces to the destination and exit. When we have more than 4 pieces, we only consider the closest 4 pieces since moving farther pieces is useless.
The weight of DIST is 10. action may cause the value to change from +10 to -10, which is smaller than eating.

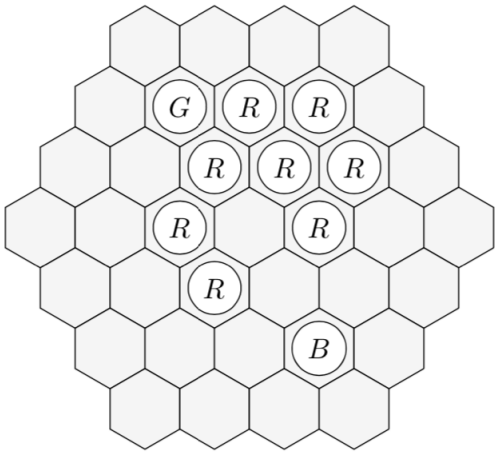
- **BOUND**



(Figure 7)



(Figure 8)

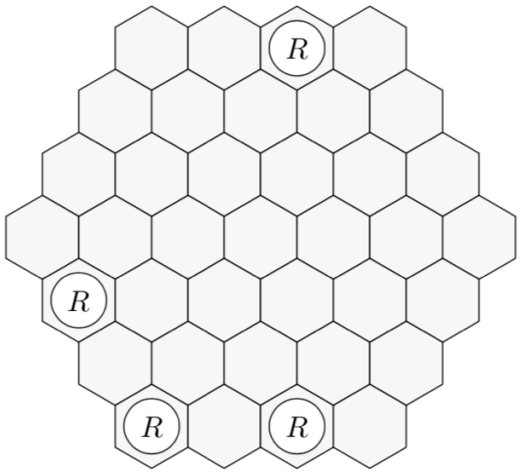


(Figure 9)

Pieces had better to bound together in order to eat back our pieces and decrease loses (The situation in Figure 8 is better than the situation in Figure 7). The BOUND value is # of player's pieces surrounding to each piece / # of player's pieces.

The weight of BOUND is 1 since the priority of DIST must higher than it. The situation of Figure 9 would appear if the BOUND weight is higher than DIST weight. The range of BOUND is 0 to 5. One round may cause about +1 to -1 which is not as high as previous features so it is only considered after eating and moving to destinations.

- SIDE



(Figure 10)

The value of at the side or corner of the board, since the side is safer and corner is safest. This value is used when we want to protect or pieces.

The SIDE value is represented by # of pieces along with the side / # of pieces. And the weight of this feature is 0.01.

Overall, the Evaluation Function is：

EAT * EAT weight - DIST * DIST weight + EXIT * EXIT_weight + BOUND * BOUND weight – DANGER_COLOUR * DANGER_COLOUR weight + SIDE * SIDE weight – DANGER_PIECES * DANGER_PIECES weight

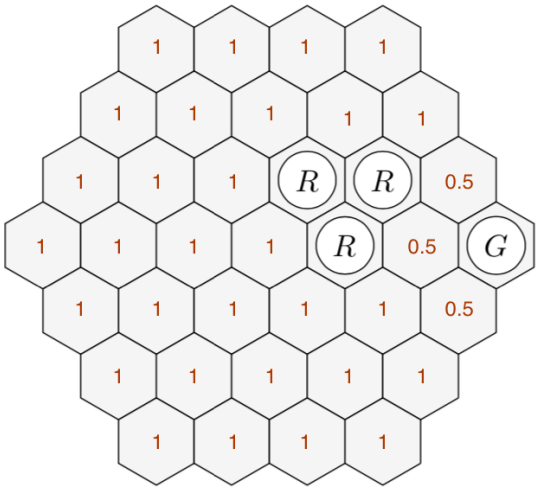
Each range of the low priority factor should be smaller than the factor with higher priority. Otherwise, the evaluation function is not accurate.

What's more, we considered other opponents only considered eating and moving to destiastions to exit, which predict normal opponent behaviors more aggressive and avoid making bad decision to lose pieces. The Evaluation Function is:

EAT * EAT weight - DIST * DIST weight

Other Features we Considered

- **Weight for places on the board**



(Figure 11)

We stored the positions of the board as key and the rewards of the positions as value. By adjusting the value of the specific position, we can control the action of pieces.
This feature is useful in exiting when others block our destinations since we could make the weights around their pieces lower so that avoid making contact to blocks (RED player in Figure 11).

However, DANGER and SIDE could handle the simple situation of exiting already. Assigning different values in other places may cause errors in moving to the destination process since the values of places are hard to change according to the state on the board

- **AVOID**

This feature was used to avoid enemies when exiting and we are at an inferior position. However, it made the performance of the agent too conservative, so it is replaced by DANGER and SIDE which is more flexible.

Other Approach we Considered

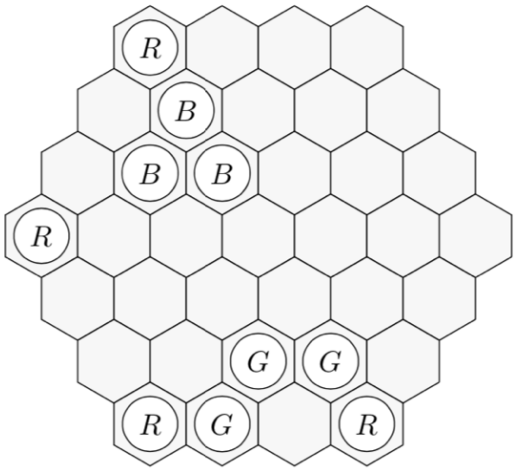
- **Different phase in different situation**

Under different situations of our pieces, we adjust our evaluation function for better action. If the piece is at the destination and can exit without influence other pieces (such as other

pieces wouldn't be eaten because of the target piece's exit action), the evaluation value of exit will directly increase to the level that can be recognized as winning the game. Also, we added a function to search surrounding enemies' pieces for this situation to avoid the target piece to be eaten. Another situation is when we do not have enough pieces to win the game and cannot get more pieces at the current state, we will move our pieces to the corner of enemies' destinations and wait for chances to get more pieces.

However, it is very hard to distinguish when to switch to different phases. Going to opponents' destination and trying to eat pieces back with few pieces is still have little hopes to win. With a few pieces, we may only stop one opponent from exiting but may still lose by the other. With high intelligent agent blocking their destination is not that useful. The if statement to distinguish different phases is left in the code as comment with explanation, but it is not used for now.

- **Blocking Opponents' Destinations At the Start of Game**



(Figure 12)

By moving pieces to the corner of opponent's destinations at the start of games, we can eat other enemies' pieces as much as possible until we have enough pieces to exit (RED player in Figure 12).

Limitation: cannot block high intelligent agent. If at the middle one opponent is lose to the other and one opponent has more than 5 pieces bounded when exiting. Blocking is hard to stop it.

This approach may be considered as a negative approach and since it needs a series of if statement to distinguish situations but not agent itself, it may not be fit for the purpose of this project.

- **Increase the Depth of the Search Tree**

We are inspired by the Best Reply Search Algorithm mentioned by Schadd, M. P., & Winands, M. H. (2011) and came up with this strategy.

Since the possible move of enemies significantly increases our branching factor, there is a way to reduce time complexity that only chooses one move of the enemy to expand. Enemies have the priority to return the action that eats our pieces or the third party's pieces. If there are no such actions then randomly return one action. By this way, the time needed reduces and the depth of the tree can be increased and in this way, we sacrifice accuracy for time. In this way, the time complexity reduced but still couldn't search to the depth of 6 in 60 seconds.

Conclusion

There are a lot of approaches to design the Artificial Intelligence agent. Because of space and time limitations, Artificial Intelligence is likely to lose to humans. The human agent can have a winning method. Among these limitations, Max^n algorithm with good evaluation function is undoubtedly the most effective algorithm. What we can do is to adjust the evaluation repeatedly to improve the winning rate.

Reference

Nijssen, J. A. M., & Winands, M. H. (2012). An overview of search techniques in multi-player games. In Computer Games Workshop at ECAI (pp. 50-61).

Schadd, M. P., & Winands, M. H. (2011). Best reply search for multiplayer games. IEEE Transactions on Computational Intelligence and AI in Games, 3(1), 57-66.