AI Project Part B Report

Team Members: Jiacheng Ye
Zihao Chen

Our program is simple. We first create an abstract player, which has all the default functions. For example, __init__, update(), and other utilities functions. The good thing about this is that we can then extend it to many different models for testing. We have extended it to a RandomBot, GreedyBot, AlternativeGreedyBot, MaxNBot, and QLearningBot.

- RandomBot: only play randomly
- GreedyBot: after I play this move, without considering others' move, and result in a better state, then choose this move. This is an immediate evaluation. It is good when the opponent is using action-based evaluation functions. Because a good move may not result in a good state, but a good state is always resulted by the current best move
- AlternativeGreedyBot: uses action-based evaluation. This is only used for testing different evaluation purpose. This is just a bench mark.
- MaxNBot: search depth is 3, a slightly better player, but still not idea.
- QLearningBot: a bot should learn to play the game by reinforcement learning.

Result first. Our game bot is not optimized due to several reasons. It will only attempt to finish the game as soon as possible, assuming that opponents are not too smart and are too careful to make a move.

At start, our evaluation function is ∑features * weights. This is potentially like a perceptron, after the sum exceed a certain amount, it outputs a value. We think that maybe we can create a bot by Neural Networks, moreover, by neural revolution. We started to search about 'genetic algorithm'. But this idea was quickly killed. Because we realized that this is a non-deterministic environment. And to learn how to solve this issue by using neural revolution takes too much time. Then we found out about what is called "Q-learning", a type of reinforcement learning. This algorithm only needs 3 things to implement, an environment, an agent, and a reward function. We have the environment (referee module), we have an agent (player module), and only thing we need is to implement a reward function. This is thus a feasible solution.

After research and discussion, we finally decided to use two methods to implement the game program. The main idea of the first algorithm is based on the MaxN search algorithm. Another way we choose to use the Q-learning algorithm. However, before we start, we also should consider the greedy approach, since that is potentially the benchmark strategy.

The game confrontation program mainly involves the field of reinforcement learning. After discussion, we finally choose to implement our program using a classic algorithm, Q-learning. Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what actions to take under what circumstances. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations. The basic idea of Q-learning is to create a table of states and actions, each pair (*state*, *action*) shows the value of the "*action*" at particular "*state*". Before learning begins, the table is initialized to 0. Then, at each time t, the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1}, and update the table.

Qnew(s, a) =
(1-learning_rate) * Q(s, a) + learning_rate * (reward+max(expected_reward))
where expected_reward is the Q value of every possible future state after choosing every possible move in this state

The core of the algorithm is a simple value iteration update, using the weighted average of the old value and the new information:
Where r_t is the reward received when moving from the state s_t to the state s_{t+1}, α is the learning rate, and γ is the discount factor. In the learning process, in order to find the best way to success, we should explore as much available actions as possible. So, we define a parameter named $\varepsilon(0\leq\varepsilon\leq1)$, which means there is a ε probability we choose the next action randomly instead of looking up in the table. During the development of this program, we made many changes to improve the algorithm:

Q-Learning:

1. The original Q-learning is to create a table where the states are known at first and each state has identical actions. However, in our game, since there are too many possible states in the game, we cannot save all of them, and each state has different available actions. Considering to these facts, we use a dictionary data structure as the table, and we do not add any state into the table when initialize. Each time a new state appears, we add the new state and its available actions into the table. In this way, the table requires less space and the looking up would be faster. Even in this case, after 600 iterations, the text file size for table becomes > 2GB.
2. We believe the method we calculate the reward received in each state will seriously affect the learning outcomes. We tried a lot of methods in order to get a good performance. Detain in the reward section.
3. The first version of our program learned very slowly. So, we did some research to accelerate this process. Firstly, we used a slightly higher learning rate. Then we find a more appropriate method, using decrement ε. In the initial stages of learning, the program has a bigger chance to take actions randomly, and as learning progresses, this chance will continue to drop until the minimum number we defined.
   This process has to be done by a main function, which is the "run.py" file. It is a messy file, only used for testing.
4. However, we failed at the end due to time management. We do not have enough time to train it to let the q-value to converge. And there must be a logic flaw somewhere in our program to make this learning process slow.
5. We actually tried to create a gym-environment in OpenAI, although we made it, but our observation space and action space is too ambiguous, it does not result ideally.

MaxN:

6. A key difference between a Greedy and MaxN or Q-learning is the latter two strategy evaluate the state after each complete turn, but the Greedy will evaluate the state after its move immediately.
7. One of the potential issues with MaxN strategy is that, there is no dominat strategy. During intermedia steps, during strategy will have different opinions about how good

this state is. For example, a simple greedy strategy may think that a state is good since I can jump over a hex. But a more complexed strategy may think that even if I can jump over a hex, but it will be jumped over by another hex, thus it probably will end in a draw, thus it is probably a bad state. If MaxN uses the latter strategy, and it only do a shallow search, then it probably won't return a correct 'max-value' for that state.

8. However, there is one thing that all strategy will agree on is that if some player wins, that state is perfect for that play. Moreover, this issue can be solved if MaxN do a deep search. By experiment, if we increase the search depth to 5, even though it runs ages, but the chance of winning is high. A potential shallow search is possible, except we have spent too much time on Q-learning, thus we have to give up on this. That is why our bot is not optimized.
9. In order to achieve shallow pruning though, we must have a max_sum_scores, and a max_score for a single player can have at each state and assume the future value will not increase this max_score as well. And only when max_sum_scores < 2*max_score, we then can do prunning. For instance, at start of the game, max_sum_scores = 1000(win), each player can have max_score = 1000(win). Then at next_state, max_sum_scores are still 1000, player 'blue' still can have max_score = 1000, but player 'red' can only have max_score = 900, because it made a bad move, and in result, player' green' can have max_score = 1100. And we have to assume that no matter player red what to do, it can never get it losing scores back. This can guarantee some child nodes will not better result than others.
10. But the challenge is, how to restrict a check game score into a range, as it potentially has a large sample space. One thing we did is, to make the utility score of a given state as the sum difference among players. This ensures that the max_sum scores are limited (, and we set it to 1000).
11. Moreover, due to the nature of an evaluation function is a linear equation with positive slope, i.e. higher score when doing the right action or close to winning, it is really hard to set its ceiling. Particularly in a board game, it is possible for a player to earn its score back. Due to we have spent too much time on Q-learning, we have to give up on this. That is why our bot is not optimized.

Rewards:

1. The key idea is that a state is good implies a higher chance of winning. But a perfect state (i.e. no enemies around or have eaten a lot of opponents), does not guarantee winning, moreover, a draw might happen.
2. When we are thinking about weights, we think it as conditional probabilities. Given a set of features with a set of corresponding weights, what is the probability that it is a good state and ends in winning the game if possible.
3. If 'win' is the causes, what are some results? i.e, given win, what results are produced? If 'win' is the result, what are some causes? i.e, given win, how likely it is caused by something? But a probability network is hard to model since we don't have a huge valid date set.

4. Maybe we can try to use reinforcement learning to let the bot to learn a set of weights. Adjusting the weight during learning by using gradient descent is feasible. However, due to computer power, time limit, and most importantly, the huge sample space, we cannot explore this method.
5. Instead, we will try to manually give some features and some weights, hopefully, it can reduce the MaxN search time, thus, make it possible to search deeper.
6. Most of features can be treated as independent features, e.g. close to each other is independent of will be eaten. So P(win|S) = P(win|f1)*P(win|f2)*...
7. But some of them are dependent. e.g. If I have < 4 pieces, then the weight of 'eat' should be significant large
8. It turns out that state-related features, i.e. the features used to determine the goodness of a state, are likely to be independent. Such as how many pieces left and how far I am from the end. And the action-related features are likely to be dependent. Thus, I add an adjust weight in the 'sorted action' method, to balance the weight by dependence.
9. some rewards we used are:

features	weights
relatively close to each other (only 4 closest hexes counts) this is the same saying how many are in danger	1
relatively close to end (only 4 closest hexes counts)	1
how likely opponents will win (this does not matter for calculating probality of winning give a state)	1
how likely will be eaten with depth=2	1
how likely will eat with depth=2	1
have less than 4 pieces	1
how many have exited	1
how many avaliable actions	1
how many hexes in the spots?	1

.....

Initially all weights are set to 1. After severing games, we loop through each action a MaxN bot made, and determine which weight should be increased. And after we encounter some situation which is more complexed, such as if have > 4 hexes, and close to end, and has enemy available to jump over, don't. i.e. adjust the weight such that the sum of weight of "> 4 hexes" and the weight of "close to end" and the weight of "eat" will give less score than the sum of weight of "> 4 hexes" and the weight of "close to end" and the weight of "not eat".

Sides:

1. One huge flaw in our implementation is that we only keep tracks of the whole board. Each time we want to look up our own hexes or others' hexes, we have to loop through the whole board, this slows things down.
2. This implementation idea is inspired by Matt's code of Game.py. Before we read through his code, we were thinking of create a Player who keep tracks of two other instances of Player. This is a bad idea. It causes so much trouble when updating and creating, and lots of work is unneeded are repeated, thus reduce the speed significantly. However, by keeping tracks of the whole board only, every time we want to update or pretend to be another player, we can just simply change or colour then change it back!
3. We also did some trails, such as using sigmoid/tanh functions to keep every reward consistent. But this causes so much trouble to adjust the weights manually, so we discarded those methods.
4. Some of the mates find out that there are perfect spots in the game, which are the 6 corners. So we hardcode this into our program as well.

*links that insipired us

https://pdfs.semanticscholar.org/f244/450aea025edf6d191395a4bdd8e0d6d018cd.pdf
http://deeplizard.com/learn/video/nyjbcRQ-uQ8
https://www.youtube.com/watch?v=qhRNvCVVJaA&list=PLZbbT5o_s2xoWNVdDudn51XM8lOuZ_Njv&index=6
https://www.youtube.com/watch?v=4KGC_3GWuPY&t=753s
https://www.youtube.com/watch?v=4KGC_3GWuPY&t=753s
https://www.youtube.com/watch?v=aNuOLwojyfg&t=164s
https://www.youtube.com/watch?v=c6y21FkaUqw&t=523s