COMP30024 Artificial Intelligence
Project B Report

Team Name: deep dark fantastic boys next door
Team Member: XuLin Yang (904904) Liguo Chen (851090)

Table of Contents

Structure of Game-Playing Program[1]

/deep_dark_fantastic_boys_next_door	Our group's Chexer game program package.
__init__.py	Package import.
train.py	The script to train q-learning agent.
Chexer-A3C.py	Unsuccessful attempt to use A3C algorithm.
Constants.py	Contains all constants in the code.
Player.py	Provide an interface for the Chexer game.
State.py	The object represents info from the game.
Util.py	Contains some helper functions.
Mozi.json	The setup of our final agent.
/agent	Module contains all attempted agents.
__init__.py	Package import.
AgentFactory.py	Static factory to create various agents.
env.py	The game environment for learning agents.
GreegyAgent.py	The greedy agent considers the current best choices.
HumanAgent.py	The agent can be controlled by a human for debugging.
MaxnAgent.py	The agent uses max^n search tree.
MoZiAgent.py	The agent uses our defined strategy based on maxn search tree.
MPMixedAgent.py	The agent considers the coalition between players.
OffensiveAgent.py	The agent only considers attacking the leading player.
ParanoidAgent.py	The agent uses paranoid search tree.
QLearningAgent.py	The agent performs policy learning.
RandomAgent.py	The agent makes a random choice.
RandomMaxnAgent.py	The agent randomly chooses the result searched by maxn agent with the same evaluation value.
TDAgent.py	Unadapted TD-learning agent interface.

N-Player Game Challenge

The well-approved minimax algorithm is no longer applicable. We need a new algorithm for the task. Being unable to use minimax algorithm, we also lose alpha-beta pruning, which is proved to give substantial improvement to the performance of minimax. Moreover, with the existence of the third player, there is unpredictability introduced into the game. In a two-player game, every action of a player has the same motivation: defeat the other player. In an n-player game, alliance might be a factor need

[1] Used codes are marked as **red and bold**.

to be considered in some cases. Also, deeper searching is required for n-player games, which gives challenges to the performance of the search algorithm.

Search Strategy

Max^n

Pseudo-code for (heuristic) max^n :

```
# Inputs: State s, Player ID p
# Output: (utility vector, best action)
function MAXN(s, p):
    if CUTOFF-TEST(s):
        return (EVALUATE(s), no-action)
    v-max  <- (-inf, -inf, ..., -inf) # n dimensions
    best-a <- no-action
    for each Action a in ACTIONS(s):
        (v, *) <- MAXN(RESULT(s, a), NEXT(p))
        if v[p] > v-max[p]:
            v-max  <- v
            best-a <- a
    return (v-max, best-a)
```

Figure 1 Max^n algorithm provided in the lecture

The max^n algorithm assumes each player will **only** choose their best action in their own perspective (maximize their evaluation values). As a result, our agent misses the chance to coalited with other players when one opponent is too strong. Our implementation of max^n search is quite similar to the provided pseudo code. However, we have added shallow pruning, lazy evaluation, reoccurrence avoidance and ACTIONS(s) has changed to NEXT_STATES(s) and we have applied directly pruning when we call this method. All these techniques will be discussed in the "Search strategy optimization" section. What's more, we have extended the EVALUATE(s) to evaluate the state for a particular player with a specified evaluation function, which is EVALUATE (s, p, *f*).

Paranoid

Algorithm 2.5 Pseudo code of the paranoid algorithm.

```
1: function ALPHABETA(node, depth, currentPlayer, α, β);
2:     if node.isTerminal() or depth ≤ 0 then
3:         if currentPlayer == rootPlayer then
4:             return evaluation(node);
5:         else
6:             return –evaluation(node);
7:         end if
8:     end if
9:     for all c ∈ node.children do
10:        if currentPlayer == rootPlayer or nextPlayer == rootPlayer then
11:            α = max(α, –ALPHABETA(c, depth–1, nextPlayer, −β, −α));
12:        else
13:            α = max(α, ALPHABETA(c, depth–1, nextPlayer, α, β));
14:        end if
15:        if α ≥ β then
16:            return β;
17:        end if
18:    end for
19:    return α;
20: end function
```

Figure 2 pseudo-code paranoid algorithm[2]

The paranoid algorithm assumes opponents only consider minimize root player's evaluation value. As a result, the deeper we searched, the more paranoid our agent behave. Our implementation adds the return of chosen NEXT_STATE for our agent. What's more, this paranoid search algorithm is based on the Negamax (discussed in the next paragraph) gives us the advantage of getting rid of duplicate codes. With the help of the max-min characteristic of the paranoid algorithm, we can apply α-β pruning in the algorithm which can increase the efficiency of searching.

Negamax search algorithm

Unlike the regular minimax, and paranoid search algorithm. Both of them require a max function representing max node and a min function representing min node. In Negamax, when we evaluate min node, we just shift the evaluation value for min node with a negative sign. As a result, a max node is sufficient for the whole search process.

[2] Pseudo code taken from page 22, https://project.dke.maastrichtuniversity.nl/games/files/phd/Nijssen_thesis.pdf

Offensive Algorithm

The offensive search algorithm is quite similar to the max^n search. The only difference is the root player minimizes the specified leading player's evaluation value. As a result, our agent could have a direct attacking towards a particular player.

MP-Mix algorithm

Algorithm 1: MP-Mix(T_d, T_o)

```
foreach i ∈ Players do
  | H[i] = evaluate(i);
end
sort(H);        // decreasing order sorting
leadingEdge = H[1] − H[2];
leader = identity of player with highest score;
if (leader = root player) then
  | if (leadingEdge ≥ T_d) then
  |   | Paranoid(...);
  | end
else
  | if (leadingEdge ≥ T_o) then
  |   | Offensive(...);
  | end
end
MaxN(...);
```

Figure 3 pseudo code of NP-Mix algorithm[3]

In MP-MIX algorithm, it takes customized parameter T_d and T_o as defensive and offensive threshold to activate various search strategy. *H[]* denote the evaluated value on each player's current status. LEADING_EDGE denote the difference between the two leading players. If we are the leading player and is currently leading the game, then it is reasonable If we are not the leading player, and leading player is too strong, then it is reasonable to assume we have the coalition with the third player to defeat the leading player. Otherwise, just behave as Max^n agent.

[3] Pseudo take from https://www.aaai.org/ocs/index.php/IJCAI/IJCAI-09/paper/viewFile/507/728

Evaluation Functions

Some terminologies in this section will be explained later with the help of a diagram.

Features

when evaluating the desirability of a state, we use several features:

- Cost to destination
    - Motivation: to tell the agent to move its pieces to the destination.
    - Cost for all pieces
        - This is our first version. For each piece, the cost is calculated as half of the hexagon distance between the piece and the destination point (the minimum one for all four destination points) plus one (for exit action), which is the same as the heuristic function in our project A submission.
        - The problem with this feature is that, when our agent captures a piece of others, the total cost is higher (extra contribution from the captured one), resulting in lower evaluation value. Hence, our agent resists capturing other players' pieces.

    - Average cost for all pieces
        - This is the improved version, aiming to solve the problem of our first version. In this version, the cost is calculated as the average of the total hexagon distance. When a piece of other players is captured, the average distance is lower, leading to a higher evaluation value.
        - However, this version has its own problem as well. Sometimes, our agent manages to capture all pieces of one of the players. It is now on its way to the destination, and so is the other player. When using this feature, our agent moves the farthest pieces, which gives the same evaluation value as moving the closest piece, and the opponent exits more quickly and wins the game.

    - Average cost for the four pieces of our agent that are the closest to the destination
        - This is our final version. In this version, the cost is the average distance of the four closest pieces. when our agent is trying to move the farthest piece, the evaluation value stays the same as the average distance of the four closest pieces stays the same. Whereas, when it is trying to move one of the closest pieces, the cost decreases, thus higher evaluation value.

- Cost to goal
    - Motivation: to tell the agent to move pieces to the strategy points.
    - This feature is used only in the trap formation phase as part of the evaluation function designed for our agent. During the formation of our trap, we do not want our agent to move pieces towards the destination. We want our pieces to lock up the four strategy points. So, the cost to the destination is replaced by cost to goal, with destination coordinates replaced by strategy point coordinates.

- Total distance to walls
    - Motivation: to tell the agent to stay as close as possible to the walls.
    - This feature is used only in the trap formation phase as part of the evaluation function designed for our agent.

- Total number of pieces of the current player (including those on the board and those that have exited)
    - Motivation: to tell the agent to protect its own pieces from being by other players.
    - This is the feature used by most of our evaluation functions.

- Total number of pieces of our player (including those on the board and those that have exited)
    - Motivation: to tell the agent to be more cautious in the trap formation phase.
    - This feature is used only in the trap formation phase as part of the evaluation function designed for our agent. By using this feature, we assume that opponents are very aggressive, and will only capture our pieces. we came up with this feature after the observation that, sometimes, our agent gives free pieces to opponents in the trap formation phase. As a result, it fails to form the trap properly and loses the game at the end. A side-effect of using this feature is that our agent misses good attacking opportunities from time to time.

- Total number of pieces of our player (those on the board only)
    - Motivation: to tell our agent not to exit pieces
    - This feature is used only in the hunting phase as part of the evaluation function designed for our agent.

- Number of pieces of our player that are in the trap (used only for cleaning trap for better hunting)
    - Motivation: to tell the agent to move its own pieces out of the trap.
    - This feature is used only in the hunting phase as part of the evaluation function designed for our player. When a piece of opponents is captured in the trap, it is better to move the one extra piece out so that the trap is ready for the next attack. We have overserved that the one extra piece stays in the trap, and the opponent gets all its pieces exited from the only one left exit point with no risks at all.

- Number of pieces of our player that are exited
    - Motivation: to tell the player to exit its pieces as quickly as possible.
    - This feature is used only in the exiting phase as part of the evaluation function designed for our agent. By the rules of the game, a player with eight pieces is not more advantageous than the player with four pieces when both are trying to exit their pieces. Hence, quick exiting is equally important for winning the game.

There are two features that we decide not to use in our final submission:

- Total number of partners around each piece
- Total distance between each piece and all its partners

Both have the same motivation: keeping pieces as close as possible. During our development of our intelligent agent, we find that once a piece is lost, it is pretty hard to get one back without the risk of losing more pieces. Therefore, we think that it is a good idea to keep the pieces together. When attacking or under attack, there are always backups.

the reason why it is not in our final submission is its drawbacks. Moving as a team means that it is not efficient. It takes more time for the team to reach the destination. Besides, from our observations, after adding this feature, our agent becomes much less aggressive. It refuses to capture any pieces, even when the piece is moving alone. Moreover, it causes an infinite loop of moving back and forth. A reason we think of for the repeated movement is that moving a piece one step closer to the destination reduces the value of the team component. When it decides its next action, moving one step back sounds good because that increases the value of the team component.

During state evaluation, we also apply standardization techniques to improve the performance of our evaluation functions. Bur standardized versions of the evaluation function don't seem to work properly. What's more, the maximum of each feature various from a different situation which makes it not only hard to determine but also requires a dynamically changes in the max feature evaluation value. As a result, we didn't adopt the feature standardization.

Other alternative feature selections (not tried but worth thinking):

The importance of each piece is a reasonable feature. The motivation is to enable the agent to choose the correct move among several pieces when these moves are sharing similar evaluation values. Due to our limited time, we didn't have enough time to come up with the way of evaluating this feature.

The aggressiveness of the player is also a reasonable feature. The motivation is to dynamically inform our agent to change the game playing strategy w.r.t. the behaviours of other players. However, it is even hard for a human to guess the opponent mental activity. So, it is too difficult for us to derive this function.

Functions

Our basic evaluation function:

$$f = -0.1 \times \sum_{piece}^{player's\ pieces} minimum\ cost\ to\ destination + \sum current\ player's\ pieces$$

Equation 1

During the formation of the trap:

for other players, we assume that they are very aggressive, the evaluation function is:

$$f = -(\sum our\ player's\ exited\ pieces + \sum our\ player's\ pieces)$$

Equation 2

for our player, we use the following evaluation function:

$$\begin{aligned} f = -0.1 \times \sum_{piece}^{\substack{pieces\ not\ in \\ strategy\ points}} & minimum\ cost\ to\ walls \\ & - 0.2 \times \sum_{piece}^{\substack{pieces\ not\ in \\ strategy\ points}} minimum\ cost\ to\ goal \\ & + 2 \times (\sum current\ player's\ exited\ pieces + \sum current\ player's\ pieces) \end{aligned}$$

Equation 3

After the trap is formed, we use two evaluation functions:

$$\begin{aligned} f = -0.1 \times \sum_{piece}^{player's\ pieces} & minimum\ distance\ to\ the\ destination\ strategy\ points \\ & + \sum current\ player's\ pieces - \sum_{piece}^{player's\ pieces} not\ in\ trap \end{aligned}$$

Equation 4

and

$$\begin{aligned} f = -0.1 \times \sum_{piece}^{player's\ pieces} & minimum\ distance\ to\ destination \\ & + \sum current\ player's\ pieces + \sum current\ player's\ exited\ pieces \\ & - \sum_{piece}^{player's\ pieces} not\ in\ trap \end{aligned}$$

Equation 5

In addition to those mentioned above, we also came up with many other evaluation functions, but we chose not to use them eventually due to their poor performance of helping our agent.

Game Playing Strategy

In the section, we explain our strategy from Red's perspective.

The following diagram helps to illustrate our strategy.

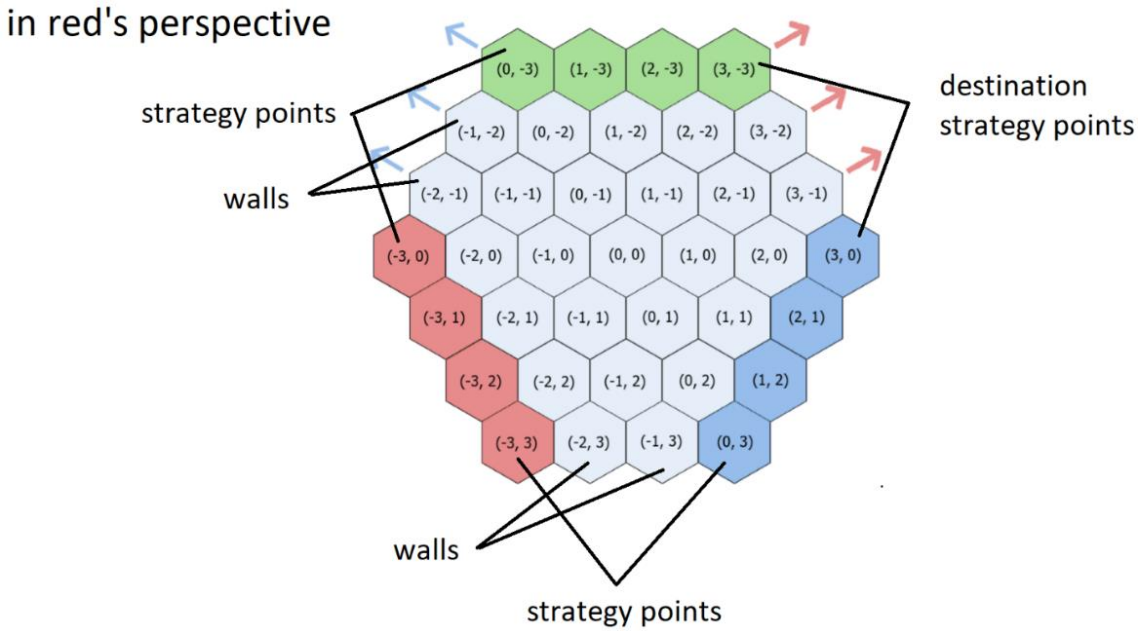

Figure 4 Board of the game

By analysing the board, we find that the six corners are the most important cells among all 37 cells on the board. In particular, the four strategy points in Figure 4 are important for Red to capture pieces of the other two players, and the two destination strategy points are important for Red to have a safe exit.

Except for the locations, appropriate timing is also an important factor in the game. If all three players tend to attack others, Blue will be the last one arriving at the battlefield, and this gives Blue extra advantages. There are cases where Red and Blue are playing well, and both have one last piece at the finishing line waiting for exiting. Red plays first, and Red wins. In these cases, Blue is disadvantageous due to the rules of the game.

We design our whole game-playing strategy based on the two observations above.

Beginning Game Strategy

When the game starts, our agent will try to move pieces to the two unoccupied strategy points using Equation 3.

To decide which one to occupy first, we divide opponents into two groups: upstream and downstream. The upstream is the one that played the previous turn, and the downstream is the one that will play the next turn. We choose to form the trap for the upstream first, which means our agent will move pieces to (0, -3) first, then (0, 3). The reason behind our decision is that there will be uncertainty before the

upstream player can respond to our action. It is possible that Green can unintentionally help us prevent an attack from Blue to stop the formation of the trap.

Setting up the trap for the downstream player also has its problem. Figure 6 shows a board configuration for the problem. Moving the free piece one cell to the right results in giving a piece to an opponent for free and failing to form the trap for Green. A rational movement to make is to move the free piece one cell down. Both options lead to one step closer to the goal, but the second one is safer. To tell our agent to choose the safer option, we use feature "total distance to walls" with slightly higher weight than feature "cost to goal", telling our agent that safety is more important.

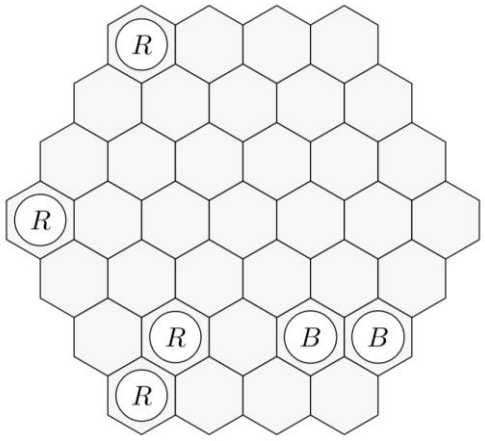

Figure 5 A problem when trapping downstream player

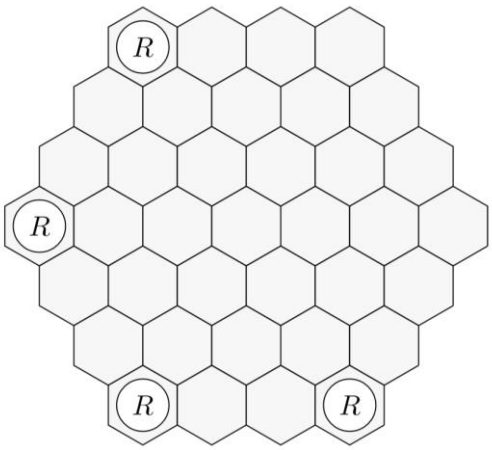

Figure 6 A properly formed trap

Mid-Game Strategy

When the trap has been formed, our agent begins to capture any pieces entering the trap. Equation 4 and Equation 5 are used in this stage of the game.

Our agent uses Equation 4 to capture opponents' pieces and move additional pieces to the finishing line.

When one or both of the opponents are not doing well, our agent may get free pieces from the trap. There is a risk of losing the strategy point, but we assign capturing in-trap pieces top priority as we consider this to be our original intention. It can then move captured pieces to destination guided by Equation 4, leaving the four pieces at the strategy points where they are. We call this "wandering phase" as captured pieces are wandering about. When captured pieces arrive at the finishing line, our agent does not exit them since there are not enough pieces for it to win and these pieces may be useful later. The "total number of pieces on board" feature helps to achieve this.

When our agent has 8 or more pieces in total, it switches the evaluation function to Equation 5 and starts exiting pieces. As captured pieces have already been moved to the finishing line during the "wandering phase", now our agent can just simply exit pieces.

When both opponents are doing well, no free piece will be given. Our agent then needs to make the hard decision: which of the four pieces to move. Figure 7 and Figure 8 show a series of possible actions (other pieces have similar logic). First, our agent will try to find all "JUMP" actions. If there are available actions, it chooses one randomly. If not (cells are occupied), it will then try to find all "MOVE" actions and chooses one randomly (we assume that there will always be at least one "MOVE" action).

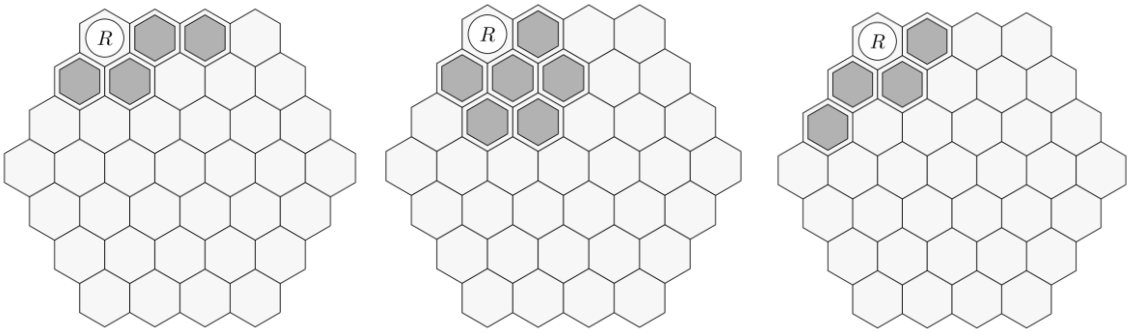

Figure 7 possible moves (shaded cells must be unoccupied)

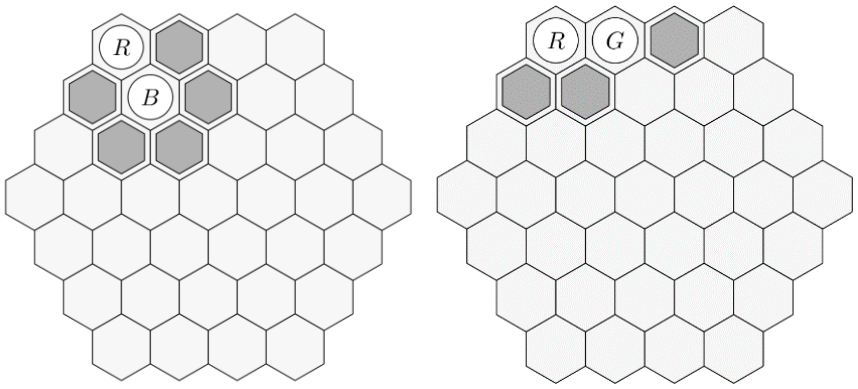

Figure 8 possible jumps (shaded cells must be unoccupied)

Endgame Strategy

Our agent enters this stage when it has 4 or more free pieces, which can be moved to the destination. We use Equation 5 as our evaluation function.

there are two possible scenarios:

- Our agent has 8 or more pieces in total as shown in Figure 10. In this case, as Green is eliminated, the two pieces on the bottom can now be freed and moved to the destination.

- One of the opponents is shared by our agent and the other opponent as shown in Figure 9. In this case, all four pieces at the strategy points still have their job to do and cannot be moved by our agent. It should, instead, move the four captured pieces to the destination.

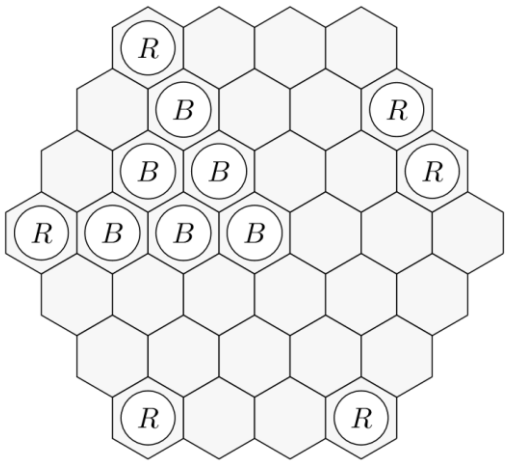

Figure 10 A board configuration where our agent captures two opponent's pieces

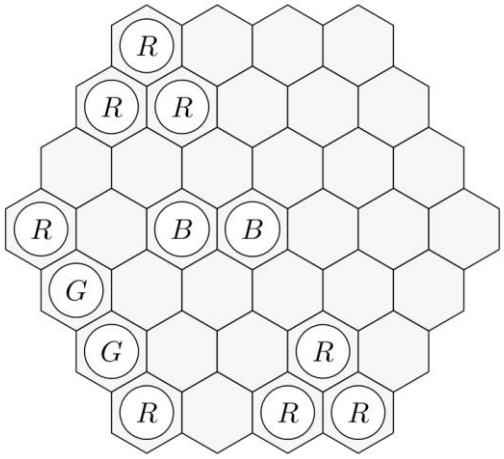

Figure 9 A board configuration where our agent captures four opponent's pieces

This Photo by Unknown Author is licensed

This Photo by Unknown Author is licensed under CC BY-SA

Artistic representation of our MoZi Agent

Coalition

As this is a zero-sum game, if we have a betrayal by the alliance (the third player is max^n agent), we might directly lose more. So, to avoid this potential invalid assumption. We didn't consider offensive agent and MP-MIX agent as our submitted version.

Other Techniques

Reinforcement Learning

We have considered reinforcement learning techniques, such as Monte Carlo Method, TD-leaf. Monte Carlo Method is not a good choice because it requires to run the whole episode. As a result, it consumes a large amount of training time. But TD-leaf is unlike the Monte Carlo Method. It doesn't require to run a whole episode to get the reward. As a result, it increases training efficiency. We have tried the Q-learning to learn the policy from the computer. However, Q-learning requires a Q-table, which requires a huge demand on memory. To solve this, we can use a neuro network which leads to DQN. But neither of us have deep learning experience. So, we give up this plan.

Open game database

When the game starts, the possible action is finite. As a result, an open game database can provide our agent with the best start setup to give the most advantage to the agent. Not only the open game database gives our agent the best choices, but also saves some time and memory space with first several turns.

Search strategy optimization

Shallow pruning

As discussed in lecture, we applied shallow pruning, but it seems not effective in majority cases. So, we didn't adopt this strategy.

Lazy evaluation

We only evaluate for one player in search tree when we want to use this evaluated value. Although this doesn't optimize algorithm big O time, it can reduce constant time. In practice, it does accelerate our program execution time. As a result, this is what we adapted.

Repeated board configure occurrence avoidance

Although initially, we didn't decide to record the occurrence of board configure for each player, our player is easily stuck in the game loop of jumps over an enemy piece back forth while a second enemy piece is also jumping back and forth. As a result, the game will result in a draw. To solve this problem,

Directly pruning

When we are returning NEXT_STATES, it is not necessary to return all possible next states when evaluating the opponents. For example, if there is an opponent's jump action over our pieces, we can only return a randomly chosen action among these. If no such action, then return a randomly chosen action among opponent's jump action over other pieces. If still no such action, return a randomly chosen action among opponent's actions left. The benefit of doing this is we can prune a lot of opponents' children. However, the disadvantage is obvious. We might lose quite a lot of accuracy (unaware of a lot of enemy's reaction).

Pieces reduction

Similarly, when we are choosing pieces to move to goals or make pieces wandering around goals when we don't have enough pieces to win. We can simply create a new State object, exclude all pieces in strategy points and start max^n searching. The advantage is we can get rid of several pieces. Thus reduces the branching factor.

Other optimizations

Extend the EVALUATE(s) to EVALUATE (s, p, *f*)

The motivation for doing this is to make our evaluation function more flexible. And we can customize it when we decide to use it in the searching algorithm.

Two-way look-up data structure

In our state class, we have a list of three lists named *player_pieces_list* which enables state can access pieces through player's index. And a dictionary of *pieces_player_dict* provides the reverse direction lookup which is accessing the piece's owner through coordinates.

Reduce to 2-player search when 1 player has knocked out

The motivation of doing this is to simplify the search tree to minimax tree. As a result, a 3-player game searching problem is reduced to a 2-player game searching problem. And pruning techniques of minimax tree can be applied. What we need to do is to transfer the search state to a 2-player version state and convert the searched 2-player version state back to 3-player search state which won't cost much calculation time.

Reduce to 1-player search

When the other two players have been knocked out, we can transform the problem to the shortest path problem. As a result, our agent can exit as quickly as possible. However, we didn't choose to adopt this plan. Because in our observations, games terminated or ended with no more than 60 turns. So, we have enough turns to move every single piece with move actions to exit 4 pieces.

Effectiveness of game-playing program

Random agent is definitely the worst agent with no doubt. Greedy is a little better. Our implementation of maxn agent can search depth with 3. Paranoid can search with depth 4. It is hard to compare their performance based on our observation. They win each other equally. RandomMaxn agent is a little better than maxn because it has some kind of randomization which brings some surprises in the practice. Finally, our chosen MoZi agent has a better overall performance compared to the agents mentioned above. When we test it on the battleground, it almost never loses and wins the agent that is weaker than maxn or maxn with bad evaluation. For the strong maxn agent, we can always result in a draw.