Final Report for AI Final Project: Gomoku

Jialun Shen 16307110030   Jinming Liu 17307110101

I. Introduction

In this project, we have created an AI agent(whose name is Eggplant) to play the Free-style Gomoku game. We have implemented two methods: Minimax search with *α-β* pruning and Monte Carlo tree search (MCTS) with Upper Confidential Bound(UCB). This report covers the improvements in Minimax, and details and results of MCTS.

II. Minimax Search

As a midterm progress, we have implemented our first version of agent with Minimax. However, we found that it has a poor performance in the contest against other AI's. In our second version of Minimax agent, as planned in our midterm report, we have improved the evaluation function, which is vital for the performance of AI, with more complicated rules, considering more patterns on board. The new evaluation process is implemented in *grader.py*. In *minimax.py* is the minimax searching and pruning process.

2.1. Improvements

The value for a search node is evaluated by *grader.eval_individial*(*board, players, moves*), where *board* is the actual board at the root node; *players*=(p_1, p_2) ∈ {(1, 2), (2, 1)} stands for players, where *p1* is the next to move; *moves is* a sequence of moves to be taken (in order).

Our improved version of evaluation process is shown as the following algorithms:

Algorithm 1: Evaluate a single move for a player

```
function evaluate_single_move(board, player, move):
Input: board: the original board, player: index of player, move: a move (x, y)
Output: evaluation value for move under board, i.e., the
find all neighbuors of move in all directions
# between move and one of its neighbour, there are only chess of the same color as player(no vacancy)
Initialize max_point=0
for each neighbour:
    Detect the pattern neighbour is in, count number of free ends, and get the corresponding point
    if point is equal to win_point then return point
    # win_point is the point of winning the game
    else max_point = max(max_point, point)
    end if
end for
return max_point
```

The following table shows the definition of a “neighbour” intuitively: suppose we are using black(●) and move x (c3) is our move of interest, it has neighbours a1, b2, d2 and e4 because they are continuous and in a line with x. c1 is not a neighbour because it’s blocked by a opponent’s chess; a3 is not a neighbour because it’s blocked by a vacant place; d1 is not a neighbour because it is not in a line with x. Note that the neighbours of x include x itself.

Red: Not a neighbour
Green: Is a neighbour

	a	b	c	d
1	🟩 ●	○	🟥 ●	🟥 ●
2	○	🟩 ●	○	🟩 ●
3	🟥 ●		🟩 x	🟩 ●
4	○	○	○	○

Figure 1: Definition of “neighbour”

In *grader.py*, the function *find_all_connect*(*board*, *player*, *move*) finds all neighbours of a *move*, *find_direction_connect*(*board*, *player*, *move*, *direction*) finds neighbours of a move in a certain *direction*. Algorithm 1 is implemented as the function *eval_point*(*board*, *player*, *move*). Algorithm 1 only considers the offensive case, i.e., the point it returns is only for chess of one colour. To consider both offensive and defensive strategies, we add points for both players as a final evaluation point.

The points for different patterns and number of free ends are shown below:

Pattern	**Number of free ends**	**Point**
●●●●●(5 or more) (already win)	>=0	100000(win_point)
●●●●(an alive 4) (will win for sure)	2	5000(will_win_point)
●●●●*2(two 1-end dead 4) (will win for sure)	1	5000
●●●●+●●●(a 1-end dead 4 and an alive 3) (will win for sure)	1 for four, 2 for three	5000
●●●*2(two alive 3)	2 for both	1000
●●●*2(an alive 3 and a 1-end dead 3)	2 for one and 1 for the other	500
●●●(an alive 3)	2	200

●●●●(a 1-end dead 4)	1	100
●●*2(two alive 2)	2	50
●●●(a 1-end dead 3)	1	10
●●(an alive 2)	2	5
●●(a 1-end dead 2)	1	3

Table 1: Points for different patterns and number of free ends

Algorithm 2: Evaluate a sequence of moves

```
function evaluate_moves(board, players, moves):
Input: board: the original board, players=($p_1$, $p_2$): ordered tuple of player, moves: ordered list of moves
Output: evaluation value for moves under board
Initialize value=0, multiplication sign=1
Set (indicators for win of me and opp) me_win, opp_win to be False
for each move in moves:
    if me have already won then add 2×win_point to value
    else if opp has already won then add minus 2×win_point to value
    else: # nobody wins
        value1, value2 = evaluate_single_move(move) for $p_1$ and $p_2$
        if value1 == win_point or will_win_point then me_win = True
        else if value2 == win_point or will_win_point then opp_win = True
        end if
        Add sign×(value1+value2) to value
        Set sign = -sign, reverse players, and put $p_1$ on board
    end if
end for
return value
```

Algorithm 2 is implemented as the function *eval_individual*(*board*, *player*, *move*), which is the final evaluation function we use.

Besides the evaluation function, we have made several other improvements in Minimax search (in *minimax.py*). We defined a *Board* class to clarify a state in the searching tree. We generalized the searching process to make it work for searching tree of height larger than 2.

2.2. Results

Our improved version of Minimax scored higher than the old version in a contest. Although we planned to use a search tree higher than 2 layers, a search tree of higher than three layers is too slow to be useful in practice. To be specific, it takes less than 1 second per move if n=2(search depth = 1), 10 to 60 seconds if n=3(search depth = 2).

With a more complicated board, time cost increases significantly. Therefore, n>2 is not practical, we can only afford to set n=2.

III. Monte Carlo Tree Search (MCTS)

3.1. Details

The general process of MCTS(Monte Carlo Tree Search) is: select a board state as root, expand each node by randomly(flat MCTS) or heuristically (Heuristic MCTS, or HMCTS) choosing next move as successor, repeatedly choose the best successor to simulate randomly until reaching a terminal state and AI gets reward, and finally propagate the reward back to all the parent nodes visited in the simulation. This can be summarized as 4 steps: Selection, Expansion, Simulation and Back-propagation. We can implement them with the following algorithm:

Algorithm 3: HMCTS(Heuristic Monte Carlo Tree Search)

```
Input: origin state s0
Output: action a corresponding to the highest value of MCTS
Add Heuristic Knowledge;
Obtain possible action moves M from state s0;
For each move m in moves M do
    reward $r_{total} = 0$;
    While simulation times < assigned times do
        reward r = Simulation(s(m));
        $r_{total} = r_{total} + r$;
        simulation times add one;
    end while
    add (m, $r_{total}$) into data;
end for
return action Best(data)

Simulation(state $s_t$)
    If ($s_t$ is win and $s_t$ is terminal) then return 1;
                                   else return 0;
    end if
    if ($s_t$ satisfied with Heuristic Knowledge)
        then obtain forced action $a_f$
            new state $s_{t+1} = f(s_t, a_f)$;
        else choose random action $a_r$ from untried actions;
            new state $s_{t+1} = f(s_t, a_f)$
    end if
    return simulation($s_{t+1}$)

Best(data)
    Return action a //the maximum $r_{total}$ of m from data
```

Therefore, one of the problems is to definite forced actions as heuristic knowledge, which means that in some cases we should not choose the next move randomly, such cases are common sense foe Gomoku players. Our forced actions are listed as follows:

Forced Pattern	**Priority**
●●●●(me 1-end dead or alive 4)	4
○○○○(opp 1-end dead or alive 4)	3
●●●(me alive 3)	2
○○○(opp alive 3)	1

Table 2: Forced patterns for MCTS

For each kind of forced pattern, we assign a priority value. An action with higher priority is chosen first. If there are several actions with the same priority, we randomly choose one of them.

Another problem is how to choose the best child node of current node in the step "expansion". And thus we need to balance the exploitation of known rewards and the exploration of unvisited nodes in the tree. For this sake, we use UCB(Upper Confidence Bound) instead of reward. The UCT(Upper Confidence Tree) algorithm is originated from HMCTS, but its difference to HMCTS is that the UCB can help to find out the suitable leaf nodes earlier than original algorithm. Thus, UCT can save more time than the original version. UCB is defined as:

$$UCB = \frac{Q(v')}{N(v')} + c\sqrt{\frac{\ln N(v)}{N(v')}}$$

where *v* is the current node to be expanded, *v'* is a child node of *v*, *Q*(*v'*) is the quality value for *v'*(in this case, reward is total number of wins), *N*(*v'*) and *N*(*v*) are number of visits for *v'* and *v*, *c* is a constant which is usually chosen empirically as √2. The child node we finally choose to take after enough number of simulations is the maximizer of UCB.

Last but not least, we have set limit time when we simulate, for it is sometimes impossible to reach the terminal state in under given computational budget. If AI fails to reach a terminal state within the required period of time, we force it to choose a relatively reasonable action.

3.2. Results

Our AI with MCTS does not perform well due to limited computational budget. For a board as large as 20×20, it would take as long as 40 seconds to simulate once -- which is impossible for MCTS to converge! It seems that our AI is not wandering on the board only if he or his opponent constitutes a danger to each other, thanks to the forced moves. The performance of MCTS can be improved if we have enough time for simulation for each step.

Furthermore, MCTS can be accelerated if we combine it with reinforcement learning methods such as adaptative dynamic learning(ADP) as discussed in [6].

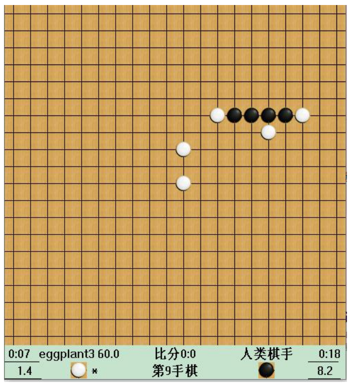

Figure 2: AI wanders until he’s going to die

IV. Conclusion

Since our MCTS agent cannot converge, it has a bad performance in games against human or other AI players. Therefore, our final AI agent still uses Minimax search. With better-defined rules, minimax agent can have a good performance.

References:

[1] Stuart J. Russell, Peter Norvig (2009) Artificial Intelligence A Modern Approach, 3rd Edition.2009, Prentice Hall

[2] Go-moku and threat-space search(1993), Louis Victor Allis and Hj Van Den Herik.

[3] 五子棋 AI 教程第二版，言川, https://github.com/lihongxun945/myblog/issues/11

[4] Searching for Solutions in Games and Artificial Intelligence(1994), Louis Victor Allis.

[5] Effective Monte-Carlo tree search strategies for Gomoku AI(2016), J H Kang and H J Kim.

[6] ADP with MCTS algorithm for Gomoku(2016), Zhentao Tang, Dongbin Zhao, Kun Shao, and Le Lv.

[7] Browne C B , Powley E , Whitehouse D , et al. A Survey of Monte Carlo Tree Search Methods[J]. IEEE Transactions on Computational Intelligence and AI in Games, 2012, 4(1):1-43.