Alpha-Beta Pruning and MCTS Methods for Gomoku

Ziqin Luo
School of Data Science
18307130198@fudan.edu.cn

Yanda Li
School of Data Science
18307130151@fudan.edu.cn

***Abstract*—In this final report, we propose some significant improvements on our α-β pruning agent compared with the one in the midterm report. We re-construct our agent and divide the evaluation system into two parts to make the search more powerful. We add threat space search into our agent as well to further improve its performance. Meanwhile, we implement an effective MCTS method based on Monte Carlo Tree simulation, which conducts many simulations for current state and generate a game search tree. Then we analyze our final results and share some personal ideas and thoughts for this final project as the finishing touch.**

***Index Terms*—Gomoku, Minimax Tree search, α-β pruning, threat space search, MCTS, Heuristic**

I. Introduction

Gomoku, as a popular two-player strategical board game, has been the focus of artificial intelligence research for a long time. More and more methods and agents are developed for this adversarial game.

After the midterm, we carried out deeper research for Gomoku to improve the performance of our agents. In the process, we consulted a lot of document literature and got a general understanding of the running system about those powerful agents. Meanwhile, we learned that game tree search, Greedy search, Hashing method [1], Genetic algorithm [2] and Monte Carlo tree search all can be used for adversarial search. But different methods and algorithms are suitable for different situations. That's why Alpha-Go choose MCTS and we finally use Minimax Tree Search to get the best performance.

In this final report, we mainly practice two algorithms: Minimax Tree Search with Alpha-Beta pruning and MCTS. The first algorithm has been implemented before and the main work we have done after midterm is to adjust our evaluation system and implement some new methods such as threat space search. To implement the second algorithm MCTS is our another job. Then we analyze our final results and share some personal ideas and thoughts for this final project as the finishing touch. Below are the details.

II. Method

A. *Minimax Search with α-β Pruning*

*1) **Introduction**:* Minimax algorithm is often used in chess and other games that include two parties. The algorithm is a zero-sum algorithm, that is, one party should choose the one that maximizes the utility among the options, while the other party should choose the one that minimizes the utility. Fig.1 can provide us with an intuitive understanding of Minimax search.

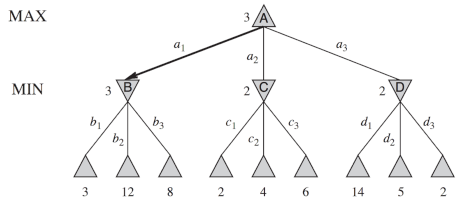

Fig. 1. Minimax Search Tree

Specifically, in Gomoku, under the assumption that the opponent is rational, we start from the current state (the board) and alternately predict the best position both in our turn and the opponent's turn to conduct the prediction recursively for several times. Finally, after the analysis, we derive the best position for our side in the current state.

*2) **Implementation**:* For one thing, in Gomoku world, the terminal state can't be found within several steps by minimax search in the most situations, that is, the leaf nodes are often quite deep. For another, there are always many possible successor states for current state, which implies that the number of child nodes of any internal node in the search tree can be very large, that is, the branching factor b is very large. Due to these two reasons, we can't directly apply the primal minimax search to Gomoku since it is quite time expensive.

We would like our search tree to be shallow and narrow. To reach the goal, we have made some modification:

- **Control the branching factor** b. We choose the empty positions which are important or have first-order neighbors around them as the possible successor states. [3] For any possible successor state, it can be represented by the

current state and one chess position that reaches it from the current state. Thus, we can use point evaluation to evaluate these positions' priority and pick up top b of them as our candidates to do the search.

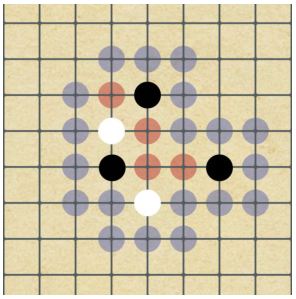

Fig. 2. Candidate selection

- **Limit the maximum search depth** d. When conducting the minimax search, we always record the depth of the state visited. As long as the depth equals to the maximum depth, we use board evaluation to evaluate the state's utility and propagate it to the root upward.

In this project, we choose $b = 10$ and $d = 6$ due to the time and memory limit. Details of point evaluation and board evaluation will be covered in the *Evaluation* section.

*3) **Improvment**:* For our previous pruning agent, we have hit some bottlenecks which are hard to overcome due to the limitation of our agent's design concept. To solve this problem, we have made some changes in our final pruning agent.

The changes can be divided into three parts:

- **Re-construct our agent**. For the board and the positions on the board, we have implemented two classes called `Board` and `Unit` respectively. Any operation on the board like minimax search can be done by invoking the methods of these two classes. This can modularize our agent and help us realize another two improvements.
- **More detailed evaluation system**. We divide our evaluation into point evaluation and board evaluation. Class `Unit` stores the patterns of each position and converts these patterns into corresponding scores using prior knowledge, that is, point evaluation. For the board evaluation, class `Board` records all the non-empty positions since the beginning of the game and goes through all these positions to get the evaluation, which is regarded as the utility of the leaf nodes, of the current board. By taking advantage of this more detailed evaluation system, our agent made significant improvement.
- **Threat-Space Search**. Our previous pruning agent can deal with the rival's attack well but behaves poor in finding rival's weakness to end the game. Thus, class `Board` maintains a 2-dimensional array to record special pattern combinations like straight four and double four which are the key to winning. We make use of this 2-dimensional array to implement Threat Space Search in our improved agent.

According to experiment results, it turns out that our modified agent works much better than the previous one.

*4) **Evaluation**:* In our modified minimax tree search, we divide the evaluation into two parts: point evaluation and board evaluation.

Point evaluation: As we mentioned before, point evaluation is used to select candidates. Specifically, for every possible successor chess position, we check its 4 directions, analyze and score its pattern in every direction. At last, we add up the scores in the 4 directions and derive the total score for each position. Then we use these scores to select candidates.

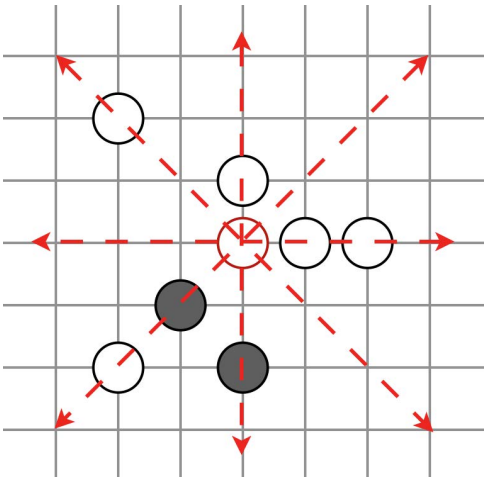

Fig. 3. Point evaluation

Take Fig.3 as an example, horizontally, the pattern is -ooo-. Vertically, the pattern is -oo-. Part of our patterns are shown in TABLE 1.

TABLE I
PATTERNS

Pattern	Score
-ooooo-	1000
-oooo-	200
-xoooo-	50
-ooo-	40
-xooo-	10
-oo-	8
⋮	⋮

In our project, for every position, we use eight 8-bit binary numbers to represent its patterns in four directions. When we need to update the patterns, simple bit operations are enough. Our agent stores these patterns internally throughout the whole game. It can make use of these patterns at any

time needed. This technique can help us store important information with less memory and speed up our minimax search.

Board evaluation: For board evaluation, we record all the positions that have been set pieces instead of empty positions. For each of them, we re-evaluate its score and sum the scores up to derive the total for both sides. At last, we use our score subtract the opponent's score to get the utility.

*5) α-β **Pruning**:* We apply classical α-β pruning to our Minimax Search Tree, which will cut off those branches that are impossible to affect the decision, and still return the same results as the minimax algorithm. α-β pruning can dramatically accelerate the search when the child nodes are explored in an appropriate order. The pseudo-code of pruning is shown below.

Algorithm 1 Alpha-Beta Search

Input: original state s_0;
Output: action a corresponding to the best value for the player;
function Alpha-Beta-Search($state$)
$v \leftarrow$Max-Value($state, +\infty, -\infty$)
return the $action$ in Actions($state$) with value v

function Max-Value($state, \alpha, \beta$)
if Terminal-Test($state$) **then**
return Utility($state$)
end if
$v \leftarrow -\infty$
for each a in Actions($state$) **do**
$v \leftarrow$ Max(v, Min-Value(Result(s, a), α, β);
if $v \geq \beta$ **then**
return V;
end if
$\alpha \leftarrow$Max(α, v)
end for
return v

function Min-Value($state, \alpha, \beta$)
if Terminal-Test($state$) **then**
return Utility($state$)
end if
$v \leftarrow +\infty$
for each a in Actions($state$) **do**
$v \leftarrow$ Min(v, Max-Value(Result(s, a), α, β);
if $v \leq \alpha$ **then**
return V;
end if
$\beta \leftarrow$Min(β, v)
end for
return v

*6) **Threat Space Search**:* In our modified minimax search method, we implement Threat Space Search (TSS) to improve our agent's performance. To explain threat space search clearly, we will take Fig.4 as a simple example first.
In Fig.4, we are black side and it's our turn to move. If we set our piece at **b1**, our opponent will set his piece at **w1** to block our five. For the same reason, we set our piece at **b2** and our opponent will set his piece at **w2**. Last, we set our piece at **b3** and we can get a straight-four pattern **-oooo-** and our opponent is impossible to block us no matter what he does. Then we can get a winning threat sequence consists

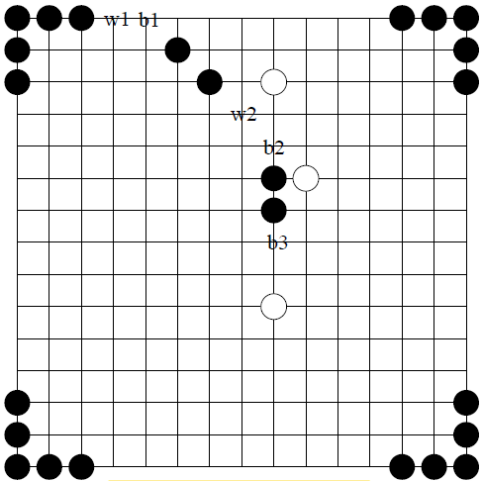

Fig. 4. A simple example of TSS

of threats {b1,b2,b3}. Similarly, the Threat Space Search [4] aims to find such a winning threat sequence that can lead us to victory. We implement TSS in a recursive way by continuously pose direct threat to force opponent to block. Meanwhile, we block our opponent as early as possible to prevent him from doing TSS.

We invoke TSS before the minimax search in each layer of the search tree. Since we only consider these positions which is likely to form three or straight four, it will not take too much time. Sometimes, it can even accelerate our search significantly.

B. Monte Carlo Tree Search

*1) **introduction**:* Monte Carlo Tree Search (MCTS) is a classic search algorithm for finding optimal decisions in a given domain y taking random simulations in the decision space and building a search tree according to the results. It has a long history within the numerical algorithms and significant successes in various AI games, such as Alpha-Go. An important feature of MCTS is that its estimated value will become more and more accurate with the increase of simulation times and nodes accessed. The basic process of MCTS is shown in Fig. 5. Briefly, it can be summarized as four steps: Selection, Expansion, Simulation and Backpropagation.

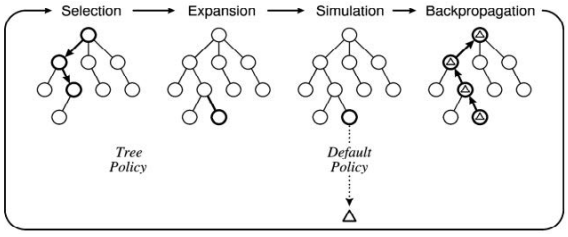

Fig. 5. One iteration of the general MCTS approach

- **Selection**: Start from root and select successive child nodes until a leaf node is reached. The root is the current game state. The move to use would be chosen by evaluation algorithm (maybe with heuristic information) and applied to obtain the next position to be considered.
- **Expansion**: Unless the leaf node ends the game decisively (e.g. win/loss/draw) for either player, create one (or more) child nodes and choose node from one of them.
- **Simulation**: If the node has not been simulated, then do a typical Monte Carlo simulation for Gomoku. Else, generate a random chld node for the leaf node and do the simulation until the result is decided. In our code, we call the function "rollout" [5].
- **Backpropagation**: Use the result of the rollout to update information in the nodes on the path from the leaf node to the root (generally 0 for lose and 1 for win). Add the number of visit timee for every node in the path.

*2) **Implementation**:* In our MCTS, we firstly implemented the Upper Confidence Bounds (UCB) algorithm to solve our selection problem [6]. The most famous usage of UCB is that it can balance the intention between exploration and exploitation, and this algorithm will help us find the result faster. Its simplest form is:

$$UCB \ = \ Q_i + \ C \times \sqrt{\frac{\ln N}{n_i}}$$

Where Q_i is the average winning rate from i_{th} simulation, N is the total nodes amount, and n_i is the child nodes number belong to node i. And C is a constant value determining the trade-off between exploration and exploitation.

However, the result is not satisfactory because of limited time and expanding the tree without checking utility of the nodes. The sample space is so large that it takes tremendous time to reach the state in which the result is decided.

*3) **Improvement measures**:* Consequently, we modify the original MCTS to add some heuristic knowledge [7] which can save more time in simulation than random sampling. The improvement measures are as belows.

- If four-in-a-row occurred in our side, the player will be forced to move its piece to the position where it can emerge five-in-a-row in our side.
- If four-in-a-row occurred in opponent's side, the player will be forced to move its piece to the position where it can block five-in-a-row in the opponent's side.
- If three-in-a-row occurred in our side, the player will be forced to move its piece to the position where it can emerge four-in-a-row in our side.
- If three-in-a-row occurred in opponent's side, the player will be forced to move its piece to the position where it can block four-in-a-row in the opponent's side.
- Besides, we also take some other special patterns into consideration. If these patterns occur in our or the opponent's side, the player will be forced to move its piece to the special position to emerge or block the situation.
- If there is no special pattern to be found, we will use evaluation function to help choose the candidate position with the probability to win. Therefore, we also adjust our UCB evaluation to reflect the difference.

With these methods, we improve the performance of our MCTS agent. We don't have to spend so much time in computing the next move. Besides, below is the brief skeleton of our modified MCTS with heuristic knowledge [7].

Algorithm 2 Modified MCTS with Heuristic Knowledge

Input: original state s_0;
Output: action a corresponding to the highest value of MCTS;
add Heuristic Knowledge;
obtain possible action moves M from state s_0
for each move m in moves M reward **do**
  reward $r_{total} \leftarrow 0$;
  while simulation times ¡ assigned times **do**
    reward $r \leftarrow$ Simulation($s(m)$)
    $r_{total} \leftarrow r_{total} + r$;
    simulation times add one;
  end while
  add (m, r_{total}) into $data$;
end for
return action Best($data$)

function Simulation(state $\mathbf{s_t}$):
if s_t is win **and** s_t is terminal **then**
  return 1.0;
else
  return 0.0;
end if
if s_t satisfied with Heuristic Knowledge **then**
  obtain forced action a_f;
  new state $s_{t+1} \leftarrow f(s_t, a_f)$
else
  evaluate action a_r with Heuristic Knowledge
  choose a_r with max probability to win;
  new state $s_{t+1} \leftarrow f(s_t, a_r)$
end if
return Simulation(s_{t+1})

function Best(**data**):
return action a #the maximum r_{total} of m from $data$

III. EXPERIMENT

Settings: To conduct the experiment, we set three fixed openings. The time of each move should be no greater than 15 seconds and the total game time should be no greater than 90 seconds. Each combat consists of 12 games.

We used our improved minimax search with α-β pruning agent and MCTS agent to combat with the 12 agents provided by the Gomoku official website. The results are shown in the table below.

The numbers in the table represent the number of times our agents have won among the 12 games. From the table, we can find that our improved minimax search with

α-β pruning agent has a better performance than the MCTS agent. It is as powerful as EULRING.

Gomoku Competition Result

Official Agent	Minimax	MCTS
YIXIN17	2	1
WINE	1	0
PELA17	0	0
ZETOR17	2	0
EULRING	5	0
SPARKLE	10	0
NOESIS	9	2
PISQ7	11	7
PUREROCKY	12	11
VALKYRIE	11	4
FIVEROW	10	6
MUSHROOM	12	12

IV. Discussion And Conclusion

Review the process of finishing Gomoku project, it can be easily found that Minimax Tree Search can perform better than MCTS in Gomoku. Besides, we learned that those powerful AI programs like Yixin also implemented Minimax Tree Search [3], despite written by C++ not Python. After discussion and analysis, we propose several reasons why Minimax Tree Search performs better than MCTS in terms of Gomoku.

First, the rules of Gomoku are relatively simple, which can be easily summarized as judging conditions. When selecting the child node, we can directly use the evaluation function to choose the best rather than simulating. We do not have to examine the whole board when we move our piece. On the other hand, due to the limitation of computing power and time constraints, we can not bring out the powerful ability of MCTS. The most serious problem about time consuming in MCTS is that MCTS must spend a lot of time on searching some unnecessary feasible actions. Compared to Minimax Tree Search, MCTS is more suitable to solve the board game with complicated situations, such as Go.

Meanwhile, we also found some interesting phenomena. On the one hand, minimax search agents with excellent performance have much to do with powerful evaluation system (time-saving and precise), and the search depth must be deep enough to reach a good performance. On the other hand, after many experiments, we found our minimax search agent performs better when the search depth is even compared with the odd depth. After discussion, we think this is because even depth implies that minimax search ends at the rival's turn, that is, MIN node of the search tree. That is to say, we optimize our worst cases and that will raise the lower bound of our prediction. According to the results, we think agent's robustness may be an important factor in terms of performance.

Overall, in this report we implemented methods for employing Minimax Tree Search with Alpha-Beta Pruning and MCTS to solve strategical games. In Minimax Search, we made some modification to optimize the minimax search tree. We also introduced a detailed and effective evaluation and Threat Space Search to improve our agent's performance. In MCTS, we introduced the UCB algorithm and some methods with heuristic knowledge to improve the MCTS agent.

However, our agents still have a certain gap with YiXin, the best AI program for Gomoku. One of the possible reasons is that these powerful AI programs are written by C++ not Python. Although developing a stronger AI for a certain board game is the ultimate goal, many algorithms still have their own potential to solve some special problems. In the future, we may try to employ deep neural network to make a better feature representation of the board in the next stage. Zobrist Hashing, a hash function often used in computer programs that play abstract board games like chess and Go, also may be a good method worth trying.

References

[1] A. L. Zobrist, "A new hashing method with application for game playing," *ICGA Journal*, vol. 13, no. 2, pp. 69–73, 1990.

[2] J. Wang and L. Huang, "Evolving gomoku solver by genetic algorithm," in *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*. IEEE, 2014, pp. 1064–1067.

[3] H. Liao, "New heuristic algorithm to improve the minimax for gomoku artificial intelligence," 2019.

[4] L. V. Allis, H. J. Herik, and M. Huntjens, *Go-moku and threat-space search*. University of Limburg, Department of Computer Science, 1993.

[5] H. J. K. Jun Hwan Kang, "Effective monte-carlo tree search strategies for gomoku ai," in *2016 International Science Press*. IEEE, 2016, pp. 1–9.

[6] X. Cao and Y. Lin, "Uct-adp progressive bias algorithm for solving gomoku," in *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2019, pp. 50–56.

[7] Z. Tang, D. Zhao, K. Shao, and L. Lv, "Adp with mcts algorithm for gomoku," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2016, pp. 1–7.