Midterm Report: Minimax Search With Alpha-Beta Pruning for Gomoku*

1st Keyu Mao
School of Data Science
Fudan University
220 Handan Road, Shanghai, China 200433
20307130241@fudan.edu.cn

2nd Jianzhi Shen
School of Data Science
Fudan University
220 Handan Road, Shanghai, China 200433
20307130030@fudan.edu.cn

***Abstract*—This project aims to create a gomoku agent with alpha-beta pruning minimax algorithm. Since the search height and branching factor are relatively high, a depth cap and a heuristic utility measure of truncated paths are introduced. In order to perform better in a limited time, the agent has to search faster, thus deeper. Therefore, this implementation features several optimization methods, like threat detection, action sorting and greedy pruning based on local heuristic and Zobrist hash. The effect of those methods are tested.**

***Index Terms*—Alpha-Beta pruning, Minimax, Gomoku, Zobrist hashing**

I. Introduction

In 1997, the chess AI Deep Blue beat the top human for the first time. Then in 2006, the human beat the top chess AI for the last time. Then, with AlphaGo, developed by a team led by Demis Hassabis of Google's DeepMind company, defeating the world's top chess players Lee Sedol and Ke Jie in 2016 and 2017, the ability of artificial intelligence players has far surpassed that of humans. . Chess-playing agents built by artificial intelligence algorithms are starting to attract attention.

Gomoku is a strategic chess game in which two players play against each other. Usually, both sides use black and white chess respectively, and alternately place the chess on the intersection of the straight line and the horizontal line on the chessboard. The one who first forms 5 chess in a line will win.

Louis Victor Allis used a computer to prove that the first mover will win on a standard 15*15 board in Gomoku if there are no additional rules such as other forbidden moves or exchanges are introduced [1]. After that, it has also been proved that there is also a strategy in Renju to make the first mover win [2]. However, there is currently no AI that can achieve an ensured win by taking first move.

In this report, we will specifically introduce the simple AI we construct to play Gomoku with intelligence. In this experiment, we will mainly rely on minimax tree search to solve the chess move problem.Besides, $\alpha-\beta$ pruning method is used to ensure the agent can solve the problem more efficiently and rapidly, which can help to cease the searching process of a sub-tree when we ensure that the rest of that sub-tree cannot give contribution(or a better value than present value) to the root node. Moreover, we use some specific ways to estimate the utility of a move through concerning chess patterns across the whole board. The strategy we use can help the agent to defend when facing extreme danger and seize the opportunity to kill when find certain patterns. Additionally, some methods are applied for optimization purpose. For instance, we shall introduce a heuristic function to help us choose a proper move more quickly.

The structure of the report is shown as follows:

1) The strategy we use to estimate utility
2) Alpha-Beta pruning algorithm
3) Optimization for the algorithm
4) Results display
5) Future work
6) Conclusion

II. Strategy

In this section we will introduce the strategy we use in the experiment to evaluate the utility of a situation on the board and some special methods use to recognize the pattern more orderly and efficiently. Since our goal is to win the Gomoku game, the chess agent is expected to possess some knowledge of Gomoku, including the game rules, how to get advantages in the game, and some details concerning recent board.

It is worth mentioning that when we start to compute the utility of one certain role(white or black), we will first choose a rather appropriate location and assume we have already placed the chess on that location. Then we can derive the utility of the whole board with respect to given role by recognizing patterns and computing values on a fixed board. We will introduce the strategy we use concerning utility with details in the following 2 subsections.

A. Calculate the utility

To compute the utility, it's essential for us to understand that chess patterns play an important role in a Gomoku game. In a Gomoku game, in order to first construct a winning pattern(a line with 5 pieces of chess in same color), players should devote themselves to forming patterns in longer length or with more threat(a great number of simple patterns sometimes will be more threatening than a single complex pattern). Due to that, we hope that our agent can perform well in finding

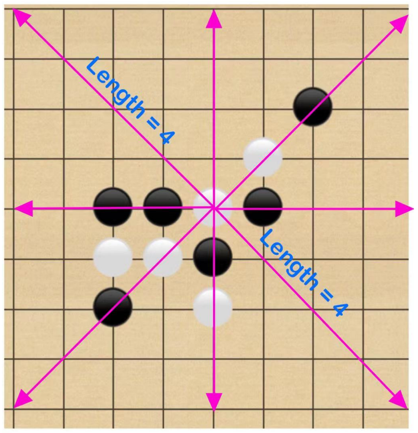

Fig. 1. Calculate the utility in 4 directions

patterns and then choose an optimal situation due to the patterns a move will bring on the board.

Besides, according to the rule of Gomoku, for a chess piece a certain location, it can contribute to chess patterns in 4 directions:

1) from top to bottom
2) from left to right
3) from bottom left to top right
4) from top right to bottom left

In addition, rules claim that 5 pieces of chess in a same color will bring a win. In that case, for a certain situation (x,y), only 9(4+1+4) grids in a certain direction can make sense to form a pattern. The figure1 will visually show you how this strategy work in terms of a certain situation.

Since the utility is a criterion for the whole board, we must avoid a same pattern for more than one time. To specify, we use a 3-dimension array to record whether a certain direction of a grid has been considered. The element of the array looks like $Array(x, y, direction) = 1$ or 0. By repeatedly updating and querying the array, we can ensure that every direction of a certain grid will be considered only once.

Still, if we choose to consider all the vacant grids and compute the utility by assuming they have a chess piece, the whole procedure will be extremely time-consuming and complex. There we will introduce some methods to optimize this procedure in the 4th section.

B. Recognize the pattern

We have already introduce 4 directions we use to find pattern, and then it's important to teach agent which patterns are valuable. Hence, We enumerate almost all possible chess patterns and give different patterns points based on our playing experience. To give a brief introduction, we will simply list some of typical chess patterns in the tableI as an example. It should be highlighted that there many patterns will can be concluded as a same type. For instance, we only list a kind of "Sleep Four" in the 7th line of the tableI, but there are many different kinds like "XMMXMMO".

TABLE I
(EXAMPLE) PATTERNS AND POINTS

Pattern	Points
XOMX	0
XMX	0
XMMO	1
XMMX	10
XMMMO	100
XMMMX	1000
XMMMMO	10000
XMMMMX	100000
MMMMM	10000000

[1] M:my chess piece
[2] O:opponent chess piece
[3] X:vacant grid

Note that sometimes the group of simple patterns can be much more threatening than a rather complex patter. We do want out agent to have the ability to find such threatening pattern and offer to form those patterns which can directly lead to wining the game. For instance, take a look at the circumstance of figure2 and figure3. Although figure2's choice can form a "sleep four", which is more valuable than "live three", the figure3's choice can form a "double live three" than can absolute win the game. In that case, the agent will perform as figure3 rather than figure2

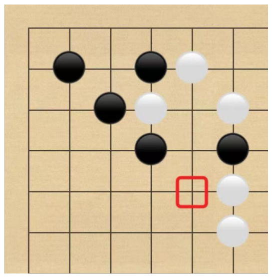
Fig. 2. Sleep Four

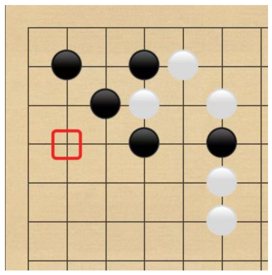
Fig. 3. Double Live Three

For a fixed board, we will recognize all the patterns on the present board and then compute the score of the board. The agent should iterate all "appropriate" vacant grids and choose the one with highest score to perform a move.

In the "Optimization" section, we will introduce how to choose "appropriate" situation and how to highlight the value of certain combination of simple patterns(like double live three).

C. Global score acquisition

In the above part we have talked about how attain the score estimation of a single grid due to a given fixed board. Taking the whole board into consideration, we will use the utility of white and black to generate the final score which the agent will use as judgment criterion.

Assume $m\_utility$ serve as the global utility of agent, and $o\_utility$ serve as the global utility of opponent player. Then we will use the equation blow to generate final scores:

$$Final\_Score = w * m\_utility - o\_utility \tag{1}$$

where w is a coefficient to choose different styles of moving strategy. When w is large, the agent will focus more on patterns belong to itself, so it will be more likely to take aggressive moving. On the other hand, if w is small, the agent will pay more attention to defensive moving.

III. Alpha-Beta Pruning

In the experiment, the main algorithm we use to search is Alpha-Beta pruning search based on Minimax algorithm. At first we should talk about the basic hypothesis in Minimax algorithm. In the Minimax algorithm, we just postulate the opponent is also a wise agent and will tend to maximize its utility and decrease our utility. As a result, the search tree in Minimax search will use Min node and Max node alternatively. In Min node, our agent will focus on maxmizing our score, while in the max node we just postulate the most likely step enemy will take.

The Alpha-Beta pruning method can help use to reduce time and cost a great deal. As we mentioned before, Alpha-Beta pruning will stop when the agent realize the sub-tree will never contribute the root node. The pseudo-code of Alpha-Beta pruning goes like figure 4.

```
function ALPHA-BETA-SEARCH(state) returns an action
 v ← MAX-VALUE(state, −∞, +∞)
 return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
 if TERMINAL-TEST(state) the return UTILITY(state)
 v ← −∞
 for each a in ACTIONS(state) do
   v ← MAX(v, MIN-VALUE(RESULT(state, a), α, β))
   if v ≥ β then return v
   α ← MAX(α, v)
 return v

function MIN-VALUE(state, α, β) returns a utility value
 if TERMINAL-TEST(state) the return UTILITY(state)
 v ← +∞
 for each a in ACTIONS(state) do
   v ← MIN(v, MAX-VALUE(RESULT(state, a), α, β))
   if v ≤ α then return v
   β ← MIN(β, v)
 return v
```

Fig. 4. Pseudo-code of Alpha-Beta pruning

Just like the pseudo-code, the Alpha-Beta search will use Max node to update alpha and use Min node to update Beta. If the value is beyond $[\alpha, \beta]$, the whole sub-tree will be ignored since other value will not be chosen by both our agent or opponent agent. Then figure 5 visually demonstrate an example procedure of Alpha-Beta pruning. For instance, after the 5th node in the 3rd layer pass the value 5 to its parent, the right part of sub-tree will not be searched since 5 is smaller than Alpha value(α =6 after searching 2nd sub-tree).

IV. Optimization

In the above sections, we mainly talk about the basic framework of our project and introduce some theories of

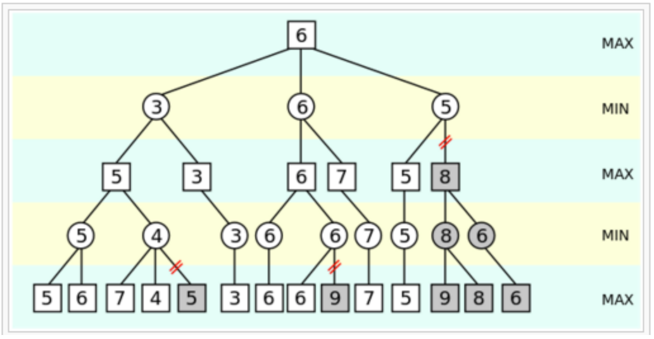

Fig. 5. Alpha-Beta pruning procedure

algorithms or methods used in the experiment. In the following section, we will concentrate on some details in code which can help the agent work more effectively and orderly, some complements for special cases, and some practical techniques we apply to optimize the algorithm.

A. Code optimization

According to the way we use to compute the utility of chess pieces, it's of vital importance for us to accelerate the process of recognizing pattern and choosing proper situation. Thus, we consider many rather special cases in the code.

1) Adjacent Judgement: First, we try to ignore some grids that are unlikely to possess high utility. As we all know, the pattern is formed by a series of chess pieces, indicating that a single chess piece has little value. In that case, for a five grid (x,y), we will first get the neighborhood information of (x,y). We will only choose to take (x, y) as a candidate move when (x,y) have neighborhoods(both our chess or opponent chess can be regarded as neighborhoods) since only such type can contribute to patterns or destroying opponent patterns.

2) Pattern recognition: Second, we use 4 variables in code to help us recognize complex pattern and abandon actions with no additional value. The variables are respectively: left_edge, right_edge, left, right.

The first 2 variables will travel from the center grid (x,y) in a certain direction until find an opponent chess piece or reach the boundary of that direction. As a result, we can compute the

which is not the same type as center chess piece (Namely, travel until O or X appears). By "right" and "left", we can rapidly compute the usable space in a given direction.

$$Usable\_space = right\_edge - left\_edge \tag{2}$$

If the length of usable space is no more than 4, our agent cannot place 5 chess pieces continuously in this direction, so every move in the space will contribute nothing to wining. Given that, we should give up any action in this blocked line. Take figure 6 as an instance, the red circle should not become our target although it can form a series of 4 chess pieces.

Although "right_edge" and "left_edge" can help us to get rid of some valueless move choices, they cannot help us to

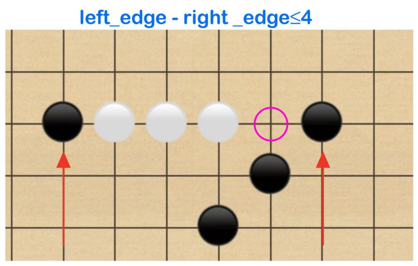

Fig. 6. Block Four

determine the pattern precisely. ”left” and ”right” variables will also travel from center and figure out continuous pattern around the center chess. This will help us to determine some vague patterns. For example, 4 chess pieces in usable space can be a ”sleep four” or 2 ”live two”.

B. Threat Pattern

In the test process, we find that if we only use the value evaluation standard to consider the utility pattern, the agent may lose the attention of some potential risks like double ”Live Three”. Additionally, there are some certain cases that the agent have only one choice to take move. If we can attach great importance to such circumstances, the agent can prevent some loss and be more likely to catch some potential chances to win the game.

First, when encountering ”Live Four”, ”Five” of ours or ”Five” of enemy, the movement is fixed(Fig 7). For the former one, the agent should choose to add a chess piece to the ”Live four” can win the game. For the latter one, the agent should block the pattern or Sleep four will turn to a ”Five” in the next turn for a wise enemy. When facing these 2 patterns, the agent will not do further search and take a move immediately, helping to reduce the time consumption.

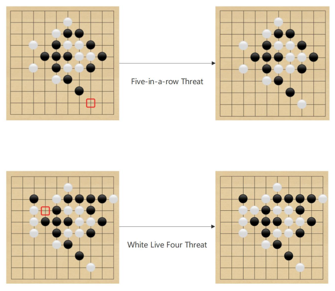

Fig. 7. Certain choice due to threat patterns

Second, as we have mentioned, some simple patterns can form a threat. In our utility computing method, the difference between patterns is very large, ensuring the agent will consider complex patterns first. Nevertheless, we will ignore some threatening patterns in this way. Due to the score we give each pattern, we need over 10 ”Live Three” to let agent give up ”Sleep Four” and choose ”Live Three” instead. So we take some important patterns into account. For example, since Double ”Live Three” will lead to an absolute win, we make its score higher than ”Sleep four” but lower than ”Live Four”(”Live Four” also leads to an absolute win and takes fewer steps).

C. Heuristic function and greedy pruning

Alpha-Beta pruning is an efficient method and can surely find the right value for the root node. But there are still some problems.

To start with, the algorithm will be slow if the order of nodes is not good. Like the example in figure 8. If we can sort the nodes with respect to a certain heuristic function, the pruning process can be more efficient since the agent can be aware of whether the sub-tree should be cut much more earlier.

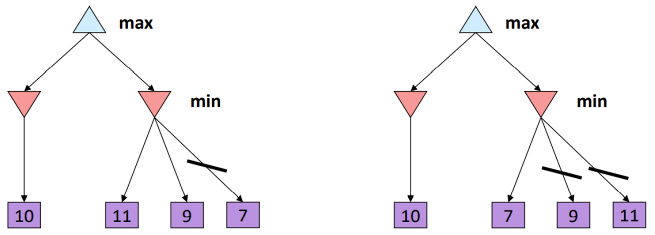

Fig. 8. Bad order vs Good order

Next, if we choose to consider all vacant grids in the board(Though we have abandoned grids without neighborhoods), the whole process will also be time-consuming. Hence, we can simply evaluate the utility of vacant grids, and then only choose grids that are valuable enough to search deeply and compare.

Considering 2 problems mentioned above, we decide to create a heuristic function. Let’s think about a simple sub-problem. When computing the utility of a move, we don’t compute global utility and won’t go deeper to follow Minimax Algorithm. Instead, we only consider the utility around the position we choose to place chess piece. We will form heuristic function based on this sub-problem.

Before calculating global utility of each possible move, the heuristic function will consider partial pattern each possible move will bring. To be specific, given a certain move, consider patterns the center chess involved in 4 directions, add the scores of those patterns, and use the sum as heuristic value. This process should be fast, since we only consider one position and won’t go deeper. Afterwards, we sort the grids by the heuristic value and choose the top k largest grids (k is an adjustable parameter). Those k grids will be action list that agent should choose from. This greedy pruning method can greatly reduce our time consumption and will have little negative influence on the strategy.

It’s worth mentioning that by taking the evaluation step, we can find threatening pattern mentioned in the last subsection very quickly and then take an immediate move without doing

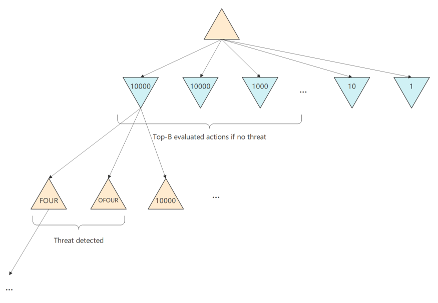

Fig. 9. Greedy pruning

any searching mission. The framework of greedy pruning and the case of finding threatening patterns are shown in fig 9.

D. Zobrist hashing

In the Gomoku game, we there several cases where different movement series will lead to a same board state(Take figure 10 for an instance). Actually, such cases will cost a large amount of extra time and will lead to the huge drop of the efficiency of Alpha-Beta pruning search.

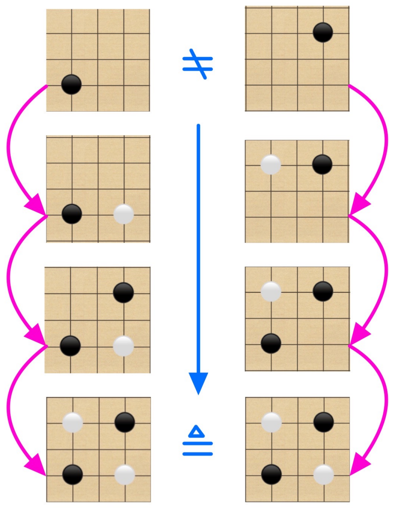

Fig. 10. Different movement may lead to same state

Hence, we decide to use Zobrist hashing to optimize. Zobrist is a fast hash algorithm ideal for using in various board games and is named for its inventor, Albert Lindsey Zobrist [3]. At the beginning of, Zobrist hashing we will randomly generate a series of bitstrings for every possible element of a board game, i.e. for each combination of a piece and a position. For instance, in the game of Gomoku, we will use 2 pieces × (M × M) board positions, where 2 is number of colors and M is board size). Now we can use independent piece/position components to denote any board configuration, which are mapped to the random bitstrings generated earlier. Every time we place a chess piece on the board, we only need to a single XOR calculation to gain hash value.

In our experiment, we will generate random numbers in 64 bits. This is a quite safe choice since it's almost impossible to generate 2 same numbers in the board.Before searching, we generate a big random board shaped as (2,20,20). Each time before the agent carries out Alpha-Beta pruning based on Minimax search, the agent will check whether the present hash value(By do bitwise XOR with the corresponding random number generated in the random board) has already existed. If no, the agent will continue to search and store the utility and hash value in a dictionary. Otherwise, the agent will immediately gain the utility by querying in the dictionary.

Due to the property of XOR calculation, we know that every number does XOR with a same number for even times, the first number will stay same. Making use of this property, we can do XOR after searching in a certain layer, and the board state(denoted by hash value) will just recover! As computer can finish XOR in a extremely short period of time, Zobrist is expected to improve the efficiency of the agent a great deal.

V. Results

A. Speed improvement

In the above section, we talk about the use of Sorting which can improve pruning process, greedy pruning method which can improve global efficiency of algorithm, and Zobrist hashing which can help to update board state in a splendid way by making use of XOR. However, we only offer some theoretical explanation and some empirical postulation. In this subsection, we are going to test the speed improvement.

Specifically, we will let our agent to play with AI "five row" with different strategies, and count the average time consumption per step. The strategies consist of original algorithm(without sorting and greedy pruning), algorithm with sorting, algorithm with sorting and greedy strategy, and algorithm carried out on Zobrist board(sorting and greedy strategy are also used). Besides, we consider depth that ranges from 2 to 4. The results are shown in figure 11 and details can be viewed in table II. Remark: The y-axis unit is milliseconds, and the x-axis unit is the number of layers.

TABLE II
Specific Time consumption

Optimazation\depth	2	3	4
Original Algorithm	154ms	2640ms	14000ms
With sorting	94ms	614ms	2616ms
With sorting and greedy strategy	88ms	300ms	1686ms
With Zobrist	83ms	200ms	1022ms

Obviously, sorting method, greedy strategy and the use of Zobrist hashing great boost the speed of searching process and the efficiency of agent. Furthermore, we can find that in shallow layers, 2 in particular, the difference is slight among 4 lines. However, as depth goes further, the difference becomes significant.

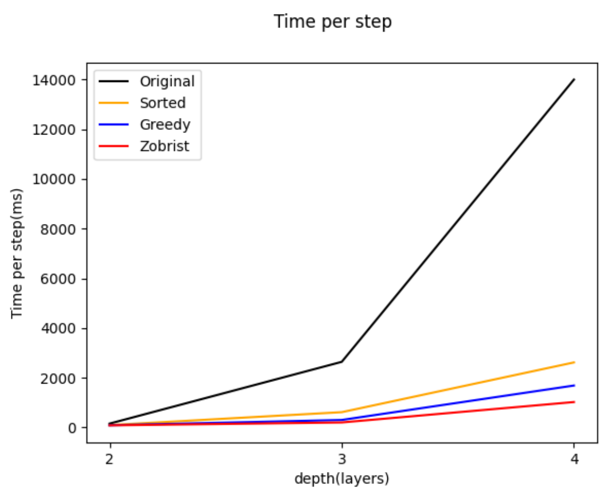

Fig. 11. Speed improvement based on different optimization methods

B. *Experiment Comparison*

In the final part of our work, we use our agent to play gomoku game against a series of given AIs, and we will display the latest competition results in table III. Notice that in this test we use searching algorithm with depth of 4.

TABLE III
COMPETITION RESULTS

AI	Win	Loss
Eulring	0	12
Fiverow	5	7
Mushroom	12	0
Noesis	2	10
Pela17	0	12
Pisq7	1	11
Purerocky	6	6
Sparkle	1	11
Valkyrie	8	4
Wine	0	12
Yixin17	0	12
Zetor17	0	12
rate	0.243056	0.756944

VI. Future Work and Conclusion

During the trials, the agent is able to make a reasonable move in a short period of time, and defeat some simple agents. We make use of sorting method, greedy strategy and Zobrist hashing, helping improve the efficiency of original Alpha-Beta pruning. Besides, we find that certain patterns should be dealt with particular movement in case that the agent may ignore some potential threat or wining chances.

However, there are still some optimization can be done in the future. We only list some of them.

- The selection of action at start can be further accelerated. We can construct a hash table for basic openings, for example, which requires scrutiny over numerous variations and nuances.
- Moreover, in order to combat those more sophisticated AI's, a utility evaluation with more than simple dummy variables and linear combinations is required. We can introduce threat-space search, which can detect potential threat on an earlier stage.

References

[1] L. V. Allis *et al.*, *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.
[2] J. Wágner and I. Virág, “Solving renju,” *ICGA journal*, vol. 24, no. 1, pp. 30–35, 2001.
[3] A. L. Zobrist, “A new hashing method with application for game playing,” *ICGA Journal*, vol. 13, no. 2, pp. 69–73, 1990.