

# 1 Introduction

A game is a zero-sum game in which two players take turns, with complete information and certainty. Gomoku is such a two-player strategical board game. The freest rule of Gomoku is that during the game, both sides take turns to place a black or white piece on the board, and the first to achieve five or more consecutive pieces of the same color in the horizontal, vertical or diagonal direction wins.

Although it seems that there are various moves in Gomoku, since Gomoku is a complete information zero-sum game, each move can be expanded out into a huge game tree. In this game tree, the score of each node represents how bad the situation is for the player, and the two adjacent layers are player actions and enemy actions respectively.

When the player moves, he chooses the node with the highest score, and when the enemy moves, he chooses the node with the lowest score, which is the most disadvantageous to the player. Thus, we can create an AI for Gomoku based on the alpha-beta algorithm.

To make the alpha-beta algorithm available, we must first figure out the way to evaluate situations on the board, as described in Section 2. Besides, considering exponential time cost, techniques are applied to make a well-performed AI in following section. In Section 4, we show experimental results against other AI. In the end, we point out several work to do in the future for further improvement.

# 2 Alpha-beta Algorithm

For the sake of unity of expression, we assume that the AI plays black and the opponent plays white.

## 2.1 Evaluation

In the ideal alpha-beta pruning algorithm, agent should search as deep as possible, which means if could reach leaf nodes where one side of the game win. But in real life, we are not capable to do it due to the exponential complexity of the tree search. Therefore, an excellent proximal evaluation is a key factor to the success of alpha-beta pruning algorithm. Here we design the evaluation strategy as follows.

### 2.1.1 Score For Each Pattern

We summarize the basic patterns of the pieces in Gomoku and assign scores to different patterns in table1. As for the name of patterns' names, the N refers to the sequence that can be added one move to become N+1, while the BLOCK\_N refers to a sequence that only can become BLOCK\_(N+1) with one specific move.



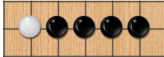






Pattern Type	Definition	Value
FIVE		10000000
FOUR		100000
BLOCKED_FOUR		10000
THREE		1000
BLOCKED_THREE		100
TWO		100
BLOCKED_TWO		10
ONE		10
BLOCKED_ONE		1

Table 1: Score For Each Pattern

### 2.1.2 Point Evaluation

The score of a move is evaluated based on how well it attack and how much it weaken the opponent. Here We divide dots into two categories, empty points and colored points.

For assigning a value to a colored point, we use the radius of 6 grid, scan its horizontal(h), vertical(v), and diagonal directions(d1/d2). Calculate patterns with the same color formed in each direction respectively, and take the largest in that direction. Finally, sum up the scores in the four directions to get a colored point score.

For assigning a value to a empty point, we assume it's black and white and calculate the score respectively. Thus it has a positive white score for the enemy and black score for AI player.

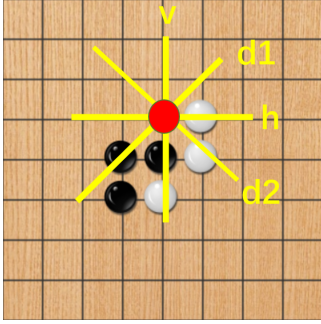


Figure 1: An example of evaluating a empty point

In this example the white score of red point is:

$$\begin{aligned} S_{white} &= S_{TWO} \times 2 + S_{BLOCK\_ONE} \times 2 \\ &= 100 \times 2 + 1 \times 2 \\ &= 202 \end{aligned}$$

Besides, the black score of red point is:

$$\begin{aligned} S_{black} &= S_{BLOCK\_TWO} + S_{TWO} + S_{BLOCK\_ONE} \times 2 \\ &= 10 + 100 + 1 \times 2 \\ &= 112 \end{aligned}$$

### 2.1.3 Board Evaluation

When we assign a value to the board, we first traverse the board and add up the scores of all the black points as our AI's score in this state, and apply the same action to the white points to get the enemy's score. We choose to use a linear combination of these two scores as an assessment of the current board state:

$$boardScore = \sum S_{black} - (1 \pm r) \times \sum S_{white}$$

In this formula, we introduce the importance of the order of attack. For a static board, the player's order will affect the judgment of the state, because the opportunity to get the first move means more freedom to attack. So we change the coefficient  $(1 \pm r)$  according to the player's turn of the board state. If it is our turn to play next, the coefficient will be  $(1 - r)$ , and if it is the opposite, the coefficient will be  $(1 + r)$ . Here we set  $r$  as 0.1.

## 2.2 Alpha-beta Pruning

Based on the evaluation, we write the following alpha-beta algorithm :

---

**Algorithm 1** basic alpha-beta algorithm

---

**Input:** board,  $\alpha$ ,  $\beta$ , depth, maxdepth

**Output:** value, action

```

1: if maxPlayer's turn then
2:   score =  $-\infty$ 
3:   get the Reasonable subsearch space
4:   for each child node in subsearch space do
5:     update the board
6:     score, move = alpha-beta(board, alpha,
7:       beta, depth-1, maxdep)
8:      $\alpha = \max(\text{score}, \alpha)$ 
9:     action = move
10:    if  $\alpha \geq \beta$  then return score, action
11:    end if
12:  end for return score, action
13: else
14:   score =  $\infty$ 
15:   get the Reasonable subsearch space
16:   for each child node in subsearch space do
17:     score, move = alpha-beta(board, alpha,
18:       beta, depth-1, maxdep)
19:      $\beta = \min(\text{score}, \beta)$ 
20:     action = move
21:     if  $\alpha \geq \beta$  then return score, action
22:     end if
23:   end for return score, action
24: end if

```

---

Here we used a few strategies to make the algorithm more efficient.

## 2.3 Active-move List

Another key issue of applying the alpha-beta algorithm is to determin points where players can

make a move. We do not consider all the points that are free on the board as active. We think that only points that can be played within the 6 grid radius of the piece. This strategy reduces the size of search space. Besides, instead of traverse the whole board to find them every time we call the `abp`-function, We use the idea of storage to reduce the number of searches. The `activeMove` list is stored during initialization and updated when a piece is placed.

## 2.4 Diminishing Branching Factor

While it reduces some of the unnecessary search space, the total time cost still grows exponentially, because there are a huge number of points available which can form sub-search spaces for each state. If we set the right branching factor and pick subspaces properly, we can effectively control the scope of the search. In the process of choosing chess positions, there are tons of terrible choices among the legal points, so it's natural to eliminate them from our search space and, more generally, we only pick the ones that are good enough to expand. After experiments and adjustments, we finally set the branching factor to 6, that is to say,

we only take the top 6 moves as the sub-search spaces.

On this basis, we further optimize: because intuitively, the deeper the search depth is, the closer it is to the optimal solution, the smaller the search space is required. So the branching factor is decreasing. After experiment, the diminishing branch factor is finally determined as  $[8, 8, 8, 4, \dots, 4]$ .

## 2.5 Flexible Strategy

During the testing, we discovered an interesting phenomenon that the AI will sometimes make lengthy moves to win when it is sure to win. In essence, the reason to this choice is that the AI only compares the result and does not take the length of the winning path into account. For example, when there are two identical games with a very high score on both the second and fourth layers, it will randomly choose one, instead of choosing the fastest option. Thus, when the situation is favorable, the AI choose the shortest path among the highest-scoring moves and the longest path among the highest-scoring moves when it is about to lose.

# 3 Realization And Techniques To Make A Well-performed AI

## 3.1 Score\_graph Class

For a board state, we use a class named `score_graph` to store the information of the board, such as the distribution of pieces, the scores of both sides. We also integrate a series of function methods such as assigning values to points to evaluate each point on the board and evaluate the whole board. See the `score_graph` class in the code for more details.

## 3.2 Evaluation Update Strategy

Since each state change of the board only involves one point position. A reasonable strategy to speed up the process is that We just update the score based on the existing results. when we update the points' scores, we update points in its horizontal, vertical and diagonal directions with radius of 6 grid like the point evaluation, while keeping oth-

ers'.

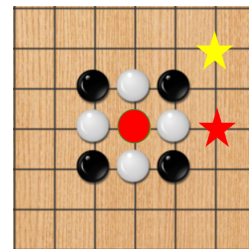


Figure 2: An example of point update

In Figure 2, we can see if we assume that the red dot is a black piece, then when we place another black piece at the position of the red five-pointed star, the value of the red dot will not change, and place a black piece at the position of the yellow five-pointed star, then its value Will change from 2000 to 200000. For the situation on the picture, placing any black piece in the horizontal and vertical directions of the red dot will not affect its

score, because it is blocked from the influence of the other pieces in these two directions. So during a update process of a black piece, when AI meets a white piece, it just skip this direction, vice versa.

### 3.3 Zobrist hash

When scoring each checkerboard state, it takes a considerable amount of time to go through each grid point, and the same state can be reached through different sequences of choices, which cause time-consuming unnecessary duplicate searches. We can use the Zobrist hashing method to record the board state with it's related information in order to avoid duplicate evaluations.

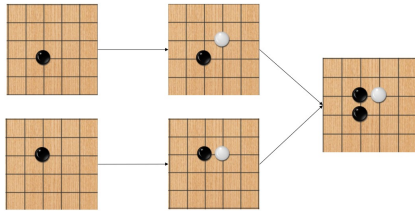


Figure 3: An example of reaching the same state

Zobrist hashing, named after its inventor Albert L. Zobrist, is an encoding method specifically designed for board games. Zobrist hashes encode the checkerboard on an integer with a very low collision rate by using a special permutation table that assigns a coded index value to each possible state at each position on the board.

It's implemented as follows, first generating a random number in the range of 64-bit integers for the black and white states at each point on the (15\*15) board. For a specific board state, the random number corresponding to the color states at each point is XOR operation, and the result is the hash value. Whenever a node is searched, we store the result in the hash table with the Zobrist value as its key, and when a node is searched, we try to retrieve it's information from the table to use it. Zobrist is so efficient that only one XOR is required for each move, and the performance cost of this XOR is negligible compared to the score of each move. However, because each board state exists in the score\_graph, it iterates over its own sub-methods when updating the board. This causes the value in the hash table to be the score\_graph

object which is deep-copied.

After analysis, we believe that the efficiency of deep copies in python is not high enough and the board state of Gomoku is not complex enough, resulting the efficiency improved by it compared to the time spent on storage is not cost-effective. So we didn't use it in the end, but we still provides the interface to turn it on.

### 3.4 Threat Space With The Essential Move

In Gomoku, a threat is an important notion. According to the previous pattern definition, BLOCK\_FOUR and THREE can be considered as a threat. According to Victor Allis's theory, to win the game against any opposition a player needs to create a double threat. During a real backgammon game, there are often some moves that need no extra thought, because if we don't take those certain actions, we will lose directly or miss the chance to win.

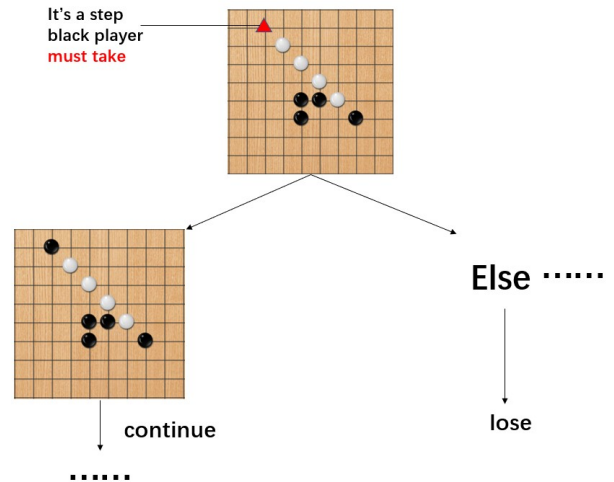


Figure 4: An example of a step that player must take

By shifting the thinking mode of human to the modular program, we can create a function focus on searching in the threat space. Notably, the size of the search space is considerably reduced by first searching for the attacker's side. Thus, we creat a function called seek\_must, whose basic algorithm is as follows.

---

**Algorithm 2** seek essential point

---

**Output:** value, action

```
    if we have win point then
2:      make that move to win
    else
4:      if opponent has win point then
        we should take it
6:      else
        if we have patterns of the THREE then
8:          we make that move
        else
10:         if opponent has patterns of the
            THREE then
                we should defuse the threat
12:         end if
        end if
14:    end if
    end if
```

---

### 3.5 Check Kill

Kill steps refer to a method that a player through the continuous THREE and BLOCK\_FOUR to attack, until the win. We tend to divide kill steps into two categories. One is VCF (Victory of Continuous BLOCK\_FOUR), and the other is VCT (Victory of Continuous THREE).

In general, when calculating the kill steps, after making sure both sides do not have a point that leads to win, we carry out VCT first and then carry out VCF when no results are found before. Obviously, with the same depth and frame, the calculation is much more efficient than the previous search. Because in the case of checking kill, each node only counts the children of THREE and BLOCK\_FOUR.

Because of the high efficiency of checking kill, we can use more time to calculate at very large depth. To avoid overtimeing, we set a maximum running time for it and keep checking for timeouts. This will moderately increase the game power of the AI.

## 4 Experimental results

-	abpruning
pela	0 : 6
Sparkle	0 : 6
Wine18	0 : 6
zotor2017	0 : 6
Yixin2018	0 : 6
Noesis	2 : 4
Eulring	2 : 4
valkyrie	3 : 3
pisq7	3 : 3
abpruning	-
PureRocky	4 : 2
fiverow	5 : 1
mushroom	6 : 0
总比分	25 : 47
胜负比	0.532

Figure 5: experiment result

The experimental results against other AIs can be found in Table 2. Our AI has been able to beat MUSHROOM completely, and beat PURE-ROCKY, FIVEROW by a large margin. Also,

it's almost as strong as EULRING, VALKYRIE, PISQ7 and NOESIS. However, compared with other AIs, there is still a certain gap, which suggests that we should continue to improve it through future work.

For the sake of time limit, currently our parameters are determined based on the goal of making a move within 5 seconds. And in order to ensure better robustness, the parameters were set very conservatively. Now every time a decision is made, a 8-deep-level abp search is performed, and the branching factor is decreased to [8, 8, 8, 4, 4, 4, 4, 4], and then the Check Kill module is used to do 8-deep-level search starting from the leaf node. We firmly believe the parameter performance still can be further improved.

## 5 Future Work

### 5.1 Activemove Update Optimization

We can optimize the update process of the Activemove list, since each state change of the board only involves one point position. In this way, when we are looking for effective Activemove s candidate

points, we only need to delete the new point in the original list and add the expanded points around the new point, instead of going through the board and searching again.

## 5.2 Proof Number Search

Proof number search selects the next node to be expanded using two criteria: the potential range of subtree values and the number of nodes which must conspire to prove or disprove that range of potential values. These two criteria enable pn-search to treat efficiently game trees with a non-

uniform branching factor.

It is shown that in non-uniform trees proof number search outperforms other types of search, such as alpha-beta iterative-deepening search, even when enhanced with transposition tables. Thus, we plan to use pn-search to make check kill module more efficient.

## References

- [1] L. Allis, H. Herik, and M. Huntjens. Go-moku and threat-space search. *Computational Intelligence*, 12, 10 1994.
- [2] Pearl, Judea. "The solution for the branching factor of the alpha-beta pruning algorithm and its optimality." *Communications of the ACM* 25.8 (1982): 559-564.
- [3] Albert Zobrist (1970). A New Hashing Method with Application for Game Playing. Technical Report, Computer Science Department, The University of Wisconsin, Madison, WI, USA.
- [4] J. Lawrence Carter, Mark N. Wegman (1977). Universal classes of hash functions. *STOC'77*