Minimax with Alpha-Beta Pruning for Gomoku

Zepeng Ding
School of Data Science
18307110088@fudan.edu.cn

Yuze Ge
School of Data Science
19307130176@fudan.edu.cn

Abstract

In this project, we used Minimax algorithm and α-β pruning to implement a Gomoku AI. We assign different scores to different gomoku patterns so that AI can reasonably evaluate the current state of the board and make choices that are beneficial to its own side. AI can quickly identify the gomoku pattern through the pattern auxiliary table we have built, and select the successor nodes according to the importance of the chess pattern. This can speed up the minimax search process so that we can set a larger search depth to increase the winning rate. In the end we can always beat MUSHROOM, and win the game most of the time when playing against FIVEROW.

1 Introduction

Gomoku, also called Five in a Row, is a two-player chess game famous in several countries. Two players use black or white stones respectively, and take turns to place a stone of their color on an empty intersection of the horizontal and vertical lines of the board. The winner is the first player to form an unbroken chain of five stones horizontally, vertically, or diagonally.

Minimax search algorithm is a classic algorithms for Gomoku. It assumes that the opponent is a rational agent, and recursively looks for actions that can maximize the benefit of itself or its opponent in each round. Each state has a corresponding value to evaluate itself. However, the state space of gomoku is so large that it is impossible for AI to enumerate all the states. α-β pruning can avoid access to a state that is already worse than the known state. In addition, we have also adopted the following methods to speed up the search speed and make the program run effectively: Limit the search depth, and evaluate the state of the board when the maximum depth is reached; Limit the branching factor; Adjust the order of searching for successors; Dynamically update and maintain the node's gomoku pattern data

Gomoku has been proven that the one who play first will have a greater advantage. In this project, we will use three fixed start states and exchange the first player.

Organization for the following report:

- **Section 2** Minimax with Alpha-Beta Pruning
- **Section 3** Heuristic Function: Score of State and Evaluation
- **Section 4** Code Implementation
- **Section 5** Experimental Results
- **Section 6** Conclusion and Future Word

2 Minimax Search with α-β Pruning Method

Our Gomoku agent is based on MiniMax Search algorithm. As described in introduction, α-β Pruning cut off those branches that are impossible to affect the decision, let the agent acts more effective. In addition, we pre-specified the maximum depth of each step of the search and queried the score in the constructed value table for each board state that might be approached. Then, we **adjust the search order** of MiniMax algorithm according to the score of these candidate states, and delete some unnecessary branches according to the preset branch factor.

A function **Getcandidates** is introduced to discover the next favorable move (e.g. Flexible-4, Blocked-4) and winning point (if exist), and order the different moves according to the score of each Adjacency state. This will help speed up the search section and help eliminate unnecessary branch searches.

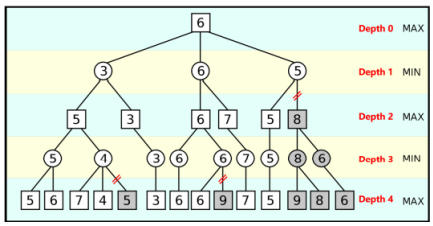

Figure 1: Alpha-Beta Pruning.

The whole algorithm is composed of three parts: **search, min-value and max-value**. **Search** implements initialization and the first hand assignment, **min-value** and **max-value** search for the best move from their respective positions (hunman player or agent).The pseudo-code is shown below.

3 Heuristic Function: Score of State and Evaluation

To choose a good move, it is needed to let the agent understand a fixed state, and know which points are good for itself, and which points are good for its enemy.

3.1 Point evaluation

For the chess pattern of a position, we recognize from four directions: horizontal line, vertical line, diagonal line, diagonal diagonal line, represented by 0, 1, 2, and 3 respectively. Different chess patterns correspond to different priorities. The higher the priority, the more points the chess type has.

Since a position can be black or white, the score of a position contains black score and white score two parts. The total score of a position is obtained by adding the scores up in the four directions.

Pattern	Shape	Score
Win 5	-ooooo-	5000
Flex 4	-oooo-	1200
block4	-xoooo-	1000
flex3	-ooo-	200
block3	-xooo-	12
flex2	-oo-	12
block2	-xoo-	5

Table 1: Pattern-Score.

3.2 Board evaluation

For board evaluation, we traverse all our chess pieces and the opponent's pieces, add up the scores (point evaluation) of the two pieces respectively,

Algorithm 1 Minimax Search with α-β Pruning

Input: initial state s_0

Output: the best position to place for current player

function ALPHA-BETA-SEARCH($state$)
  bestpos←Max-Value(0,state,-∞, +∞)
  return the best position to act
end function

function MAX-VALUE($depth$,$state$,α,β)
  if $depth == maxdepth$ **then**
    return Score($state$)
  end if
  $candlist \leftarrow Getcandidates(state)$
  $candlist \leftarrow candlist[0 : branchfactor]$
  $v \leftarrow -\infty$
  for each move in candlist **do**
    $v \leftarrow Max(v, Minvalue(move, \alpha, \beta))$
    if $v > \beta$ **then**
      return V
    end if
    $\alpha \leftarrow Max(\alpha, v)$
  end for
  return v
end function

function MIN-VALUE($depth$,$state$,α,β)
  if $depth == maxdepth$ **then**
    return Score($state$)
  end if
  $candlist \leftarrow Getcandidates(state)$
  $candlist \leftarrow candlist[0 : branchfactor]$
  $v \leftarrow +\infty$
  for each move in candlist **do**
    $v \leftarrow Min(v, Maxvalue(move, \alpha, \beta))$
    if $v < \alpha$ **then**
      return V
    end if
    $\beta \leftarrow Min(\beta, v)$
  end for
  return v
end function

and subtract the opponent's total score from our total score to get the current state score of the board. It is usually used in the evaluation of leaf nodes.

4 Details in implementation

4.1 Storage and recognition of patterns

In this project, we use 4 numbers to represent the piece on the board.

0: Our piece
1: Opposing piece
2: Empty
3: Outside the board

We get a 16-bit binary number by bit operation from the four pieces adjacent to the left of the target position and the four pieces adjacent to the right, and each 2 bits represents a piece. In this way, given the color of the chess piece at the target position and a 16-bit binary number, it can correspond to a chess pattern.Note that a target position has four directions, that is, four such 16-bit binary numbers need to be stored.

For example, in figure 2, the red box indicates the target location. We can get a 16-bit binary number from the four pieces on the left and the four pieces on the right. That is, b1010100000001001 in binary and 43017 in decimal. The pattern at this position is live 4 if we put a white stone and is dead 1 if we put a black stone.

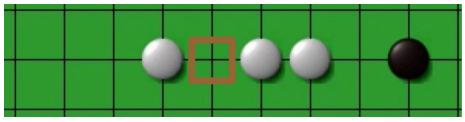

Figure 2: A 16-bit binary number represents a pattern

The range of 16-bit binary numbers is 0-65535. We can enumerate the gomoku patterns corresponding to these numbers and generate a list of 65535×2. In this way, we only need simple bit operations to get the gomoku pattern of the target position. As each gomoku pattern corresponds to a heuristic score, we can use the same method to construct a score list.

4.2 Dynamically update the board

Each position on the board saves the pattern of that position (including oneself and opponent), instead of setting each state node to a 20 by 20 chessboard.Whenever we play chess or regret chess, we first change the state of piece at the target position, and then update the pattern of its neighbors. Note that the neighbor's patterns will only change in a certain direction, so we only need to update each neighbor's patterns in one direction instead of four directions.

In addition, we will also dynamically update the number of neighbors at each location, which refers to the number of neighbors within two squares of the location.

In Figure 3, Only the points on the red line need to update the pattern, and the points in a certain direction only need to update the chess pattern in that direction.

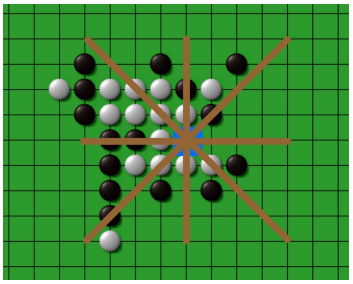

Figure 3: Dynamically update the neighbor's gomoku pattern

4.3 Candidate selection and depth control

On the one hand, in the world of Gomoku, there are many successor nodes of the current state node, and it is impossible for us to traverse all the successor nodes. Therefore, we need to control the number of successor nodes and the depth of search. On the other hand, the search order of successor nodes will affect the search speed of α-β pruning, so the order of successor nodes is also important.

- **Control of the number of candidates.**

  We first choose positions with empty piece that there are other pieces within two squares from themselves as candidates. (The number of neighbors of the position is greater than 0) Usually, there will still be a large number of successor nodes after selection. We set the branching factor b so that the number of candidates does not exceed b.

- **Sorting and selection of successors.**

  Generally speaking, we often hope to choose a position that is beneficial to us in the next

step. We will evaluate each successor node, the higher the priority of the pattern, the higher the score. The successor nodes will be ranked from highest to lowest score. We choose the first b successor nodes as candidates.

In particular, if we find that some successor nodes have special gomoku patterns, we will give priority to these successor nodes and adopt some strategies. For example: If we find a position that is live 4 or win 5, we will directly return that position, because if we don't take that position, we will lose the chance to win or the opponent will seize the opportunity to win; If the opponent has a live 3 pattern, it will be a big threat to us, and this position will become a candidate. In this case, there are many ways to contain the opponent: multiple positions to prevent the opponent from live 3, make our side form a block 4... At this time, we should select candidates from the successor nodes according to this strategy

- **Select branching factor b and search depth d.**

  We tested the thinking time of each step of AI under different b and d. We choose b=10 and d=6, so that most of the thinking time per step is within 5s. And when b=11 or d=7, the AI thinking time per step will be close to 10s or more.

5 Experimental results

We used minimax search with α-β pruning AI to compete with 12 AIs on the GomoCup website. Each competition uses 3 kinds of fixed openings, alternate starting order, 6 games in total.We set the thinking time for each step not to exceed 15s, and the entire game time not to exceed 90s. The result is as follows:

Agent	Minimax
YIXIN17	0
WINE17	0
PELA17	0
ZETOR17	0
EULRING	1
SPARKLE	0
NOESIS	5
PISQ7	5
PUREROCKY	6
VALKYRIE	6
FIVEROW	6
MUSHROOM	6

Table 2: Competition outcome

6 Conclusions and future work

In this project, we implemented Gomoku AI using minimax with alpha-beta pruning and achieved good results. By limiting the branching factor and search depth, rationally selecting candidates and dynamically storing chessboard information, our AI thinking time is shortened, the search depth is deeper, and the winning rate is increased. The highest score in the elo score is 1500.

But our AI also has some problems. When fighting against advanced AI such as YIXIN, we found some phenomena. YIXIN occasionally chooses positions far away from known nodes. (For example, the number of neighbors of nodes is 0).Our AI often plays chess in densely populated areas, eventually leading to the opponents always being able to easily block our chess pieces.For AIs like WINE and YIXIN, the thinking time can often be well controlled at about 10s, not more than 15s. Our AI has a shorter thinking time, but if we increase the search depth, the thinking time will be difficult to control and easily exceed 15s. In this way, our AI often cannot see farther than other AIs.

In the future, we intend to improve our AI in the following areas:

- **Threat sequence search**

  A winning threat sequence for a player is an action sequence which consists of a sequence of checkmate which leads to a must-win situation. Our AI will be improved if a threat sequence exist in the board.

- **Selection of successor nodes**

  In this project, our AI's selection of nodes is limited to densely packed positions. These positions are easy to produce high-priority gomoku patterns but are also easy to be blocked by opponents. We hope to find a better strategy for the selection of successor nodes, including consideration of locations farther from the dense area of the chess pieces.

- **Update search depth**

  We found that advanced AI can balance search depth and search time. We hope that the search depth of AI can be increased when there are few successor nodes, and the search depth of AI can be reduced when there are many successor nodes.