α-β Pruning Algorithm in Gomoku

Qinyi Zhao
School of Data Science
Fudan University
18307110468

Yuefan Ding
School of Data Science
Fudan University
18307110477

Abstract

This is the mid-term report for our course project Gomuku. The main aim here is to implement a Gomuku AI agent with α-β pruning algorithm. Besides implementing the mandatory algorithm, our team constructed a basic framework for Gomuku problem and defeat our opposite MUSHROOM successfully. Moreover, there are several aspects expected to be optimized. We are continue to implement more advanced agent in the final.

1 Introduction

Gomuku is a board game originating from ancient China. Two sides use black and white stone respectively and play at the intersection of the board's straight line and horizontal line. The one who firstly get a five stones in a row continuously will win the game. Rules in Gomuku could varies. However, the rule refered in our project is a free-style Gomuku with 3 openings. Free-style Gomuku means a row of five or more stones for a win. The goal of our project is to develop a AI agent for Gomuku competition. As what is being required in the mid-term report, we implement our AI with α-β pruning algorithm successfully.

The outline of this paper is as follows. The following section introduces the overall design of our agent. In the third part, we conduct a experimental comparison. The last comes to our conclusion.

2 Basic framework and algorithms

AlphaGo[2] proposed a classic framework for a Gomuku agent, which composed by a search algorithm MCTS, value network and policy network. Inspired by that, we design different modules to tackle our problems and our agent consists of three following parts:

- Minmax with α-β pruning
- Board evaluation
- Action prediction

Minmax with α-β pruning provides a basic logic of how the agent search and prune. Board evaluation part gives a empirical evaluation method to evaluate any given current board, in order to prune the tree further and avoid computing with limitless depth. Action prediction function predicts what action the opposite may take and what respond we should take.

2.1 Minmax with α-β pruning

α-β pruning algorithm maintains two values, α and β, which means the minimum score and the maximum score during the tree search respectively.

α is negative infinity and β is positive infinity at first. Whenever the maximum score that the minimizing player guaranteed comes less than the minimum score guaranteed by maximizing player

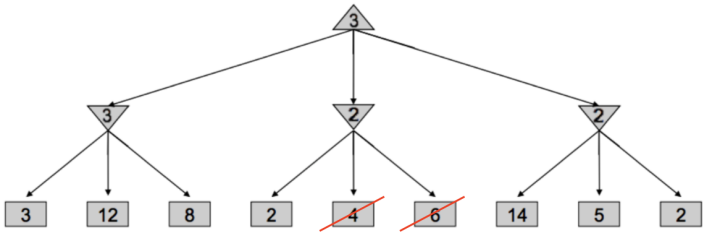

Figure 1: Minmax with α-β Pruning

(i.e. $\beta < \alpha$), we can prune the children of this node. Take Figure 1 as an example. Assume the agent search from left to right. When it explores the children of the second minimizing node, it returns 2. At this time, $\alpha = 3 > 2$ and we prune the node.

2.2 Board evaluation

To choose a good move action for our AI, it is needed to let the agent understand whether the situation is good for itself or good for the enemy. To realize this, a score evaluation function is needed. The score contains two parts: Attacker score and Defensive score.

For the black player, when he set a piece at point (x,y), he get the score as attacker score. At the same time, the white player's score for this board situation is defensive score. Thus, the final score for the black player is Attacker score - Defensive score. For a point already filled with black chess, its position equals to 1 while when a point filled with white chess, its position equals to 2, and the position is 0 when the point is free.

As for how high the score is, it depends on the importance of this point for either black or white, and the degree of importance is measured by the specific chess types. In practice, we consider 10 chess types, and the meanings of them are in the following table:

Type	Meaning	Example	Abbreviation
Renju	At least five pieces connected together	●●●●●	Long
Livefour	Has two points can get Renju	0●●●●0	H4
Punchingfour	Only has one point can get Renju	●0●●●	C4
Livethree	Has three pieces and can get Livefour	0●●●0	H3
Sleepthree	Has three pieces and can only get Punchingfour	●0●0●	M3
Livetwo	Has two pieces and can get Livethree	0●●0	H2
Sleeptwo	Has two pieces and can only get Sleepthree	0●00●○	M2
Diedfour	Has four pieces but both ends are blocked	○●●●●○	S4
Diedthree	Has three pieces but both ends are blocked	○●●●○	S3
Diedtwo	Has two pieces but both ends are blocked	○●●○	S2

Figure 2: Chess Type

When the AI consider where to set piece, it need to evaluate the score for the situation formed after falling in each position. So it should count how many chess types in rows, columns and two diagonals. In general, we need think the combination effect by multiple chess types. For example, when the AI get two Punchingfour or get one Punchingfour and one Livethree at the same time, then the other side can't defend. So we should set a higher score for those profitable combine chess types. Thus, we define the score map as Figure 3.

In the game practice, our AI will extract all rows, columns and two diagonals, and use character string to represent the board situation. To determine its chess type, the agent use regular expression matching the chess types with the board situation and find all the chess types.

Renju	100000
Livefour	10000
Double Punchingfour	10000
Punchingfour & Livethree	10000
Double Livethree	5000
Livethree & Sleepthree	1000
Punchingfour	500
Livethree	200
Double Livetwo	100
Sleepthree	50
Livetwo & Sleeptwo	10
Livetwo	5
Sleeptwo	2
Diedfour	-5
Diedthree	-5
Diedtwo	-5

Figure 3: Score Map

2.3 Action prediction

Action prediction function predicts probable actions our opposite may take and also returns the action we should take. As for now, our action prediction part is very intuitive and naive. Basically, it searches for the following position towards current board, regardless of the player.

We impose two conditions to filter the next position: Firstly, the position must be free now. Secondly, intuitively we hope there are some pieces already around the position AI choose. If there is nothing around, it might be meaningless to choose this position. So we set a parameter called *scale*, and for every free position, the agent will search its around and find if there are some other pieces. The *scale* parameter is used to adjust the size of the range.

3 Expansion pruning

During the construction of search tree, it is common to expand too many child node for a given node. That is too say, the opposite have too many position to pick. Admittedly, it is comprehensive to consider all the possible future but it is costly. In order to meet the time limit in every step, some further pruning is needed when the node expands their children.

Here, we take advantage of our board evaluation function. When the expansion of one's children is beyond a threshold *max_expand*, we sort all the child node by its evaluation score and filter the top *max_expand* children. In implementation, the sorting is maintained by a priority queue structure during the expansion of child node.

4 Experimental comparisons

In experimental practice, we set 1 for search depth and change a set of values for the parameter max_expand and see how does this parameter affect the results.

We mainly use our agent to compete with MUSHROOM and the pela which is built in piskvork. All the positions are free in the beginning. We change the offensive and defensive positions for them and record all the results of competition. The results are in the following:

	Offensive	**Defensive**
max _expand < 28	Lose	Lose
$28 \leq$ max _expand < 31	Win	Lose
max _expand ≥ 31	Win	Win

Figure 4: Our agent VS MUSHROOM

It is hard to win the competition with pela now, so for our agent now, the aim is to reduce mistakes and keep the games last more rounds.

max _expand	Number of fights	
	Offensive	Defensive
35	25	25
38	28	27
48	32	29
58	38	35
68	40	35
78	48	35

Figure 5: Our agent VS PELA

From the tables above we can see when max_expand gets larger, it can exactly promote the performance of our AI. While from the competitions we can also find the defect caused by the score evaluation algorithm and it is no use to increase the value of max_expand all the time.

In order to deal with this problem, we hope to optimize our agent in future works.

5 Conclusion and future works

In this report, we construct a basic algorithm frame for Gomuku. We implement Minmax with α - β pruning, with a empirical evaluation function and a prediction function. Our agent defeats MUSHROOM successfully in our experiments. In the future, we are expecting to optimize our AI with following aspects:

Threat space search Threat space search can improve for there exists some action sequence that one has to act accordingly or defend fails [3]. If the agent could detect those sequence during tree search, all things left is to follow the sequence and thus save the costs.

Value and policy network For now, the evaluation and action prediction function are naive and empirical-based. The expansion of our search tree is loaded both in width and depth. However, it is possible to introduce some other technique to improve our tree search process, like reinforcement learning [1].

References

[1] Allis, L. V. (1994) Searching for Solutions in Games and Artificial Intelligence. PhD thesis, Univ. Limburg, Maastricht, The Netherlands

[2] Silver, D. & Huang, A. & Maddison, C. *et al.* (2016) Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 484–489

[3] L. Allis, H. Herik, & M. Huntjens. (1994) Go-moku and threat-space search. (*Computational Intelligence*