Gomoku Agent: An Efficient Minimax Algorithm

Jiancong Gao
School of Data Science
15300180050@fudan.edu.cn

Zilong Liang
School of Mathematical Sciences
zlliang15@fudan.edu.cn

1 Introduction

Gomoku is a zero-sum adversarial chess-playing game. To design an agent for this game, methods like Minimax Search (Russell and Norvig, 2016), Genetic algorithm (Wang and Huang, 2014) and Monte Carlo Tree Search (Tang et al., 2016) are proposed.

In this report, we basically investigate the Minimax approach. We implement the algorithm, propose an efficient state evaluation policy and discuss some methods to promote its performance.

In section 2, we review the basic Minimax algorithm. Next in section 3, we propose some efficient strategies on evaluating search states. And in section 4, we dig the performance of the Minimax algorithm and carry out some experiments.

2 Adversarial Search: Minimax

For zero-sum adversarial games like Gomoku, we can formulate the chess playing process as a state space, each state including every detail of the Gomoku board, then use Minimax Tree Search algorithm (Russell and Norvig, 2016) to build our agent.

The Minimax algorithm simulates adversarial game playing. If we can evaluate states (here, boards for Gomoku), we can search all possible moves with a certain depth, and recursively select the best move we can already know. Formally, in the deepest layer with depth n, we get scores of boards:

$$V_n(s) = \text{Evaluate}(s). \tag{1}$$

Then recursively, for min-nodes, which represents the rival, lowset values representing

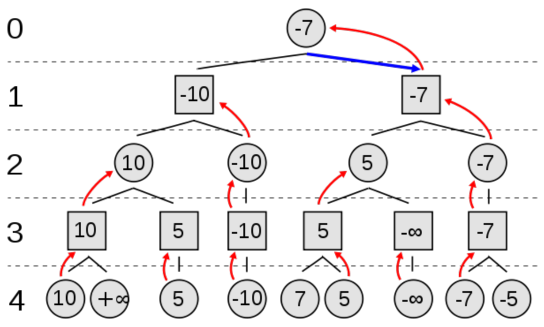

Figure 1: Minimax Search algorithm

moves that would beat our agent will be chosen:

$$V_k(s) = \min_{s' \in \text{candidates}(s)} V_{k+1}(s'), \tag{2}$$

and in max-nodes, which represents our agent, highest values will be chosen:

$$V_k(s) = \max_{s' \in \text{candidates}(s)} V_{k+1}(s'). \tag{3}$$

Here, candidates means possible moves on the certain state.

Figure 1 shows this process clearly. We can see that, as a result, the algorithm can extract the best move to take, taking next several moves into consideration.

However, state spaces in Gomoku is tremendously large. For example, if we assume we can take the next move in every vacant points, the branching factor, or the *breadth*, of search trees will be about 400. So if we try to perform a search process with depth 4, we would have to check about 400^4 board states, which cannot be completed.

As a result, here we have to adapt some pruning strategies to make our search tree relatively small. A basic method is Alpha-beta pruning (Russell and Norvig, 2016), which is effective and garantees optimality. We can also use our prior knowledge of Gomoku. For example, just select points that already has piece on the neighbor points as candidates is a rational choice, which can decrease the branching factor effectively. For more pruning strategies, we will discuss in Section 4.

3 Evaluation

No matter what solving approach we use, it is necessary to design a efficient heuristic to evaluate Gomoku states. An evaluation function with abundant prior knowledge of Gomoku and can detect precise details of the board is able to lead our search algorithm to a right way. Meanwhile, it should be with high-performance because we will call it frequently.

3.1 Point Evaluation

Intuitively, we use points as our ingredients of evaluation process. According to Gomoku rule, we check four directions of a certain point like Figure 2, score the line if it satisfies specific patterns, and finally sum scores of the four directions as the score of this point. Here, pattern recogonization is important prior knowledge, we use Table 1 for our agent[1].

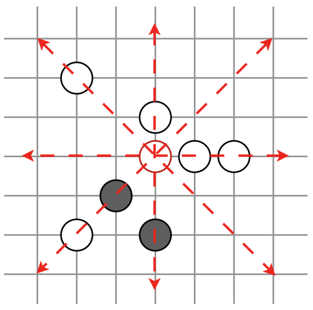

Figure 2: Point evaluation

Note that it is not only the points with piece on it will be scored, but also vacant points with patterns around it. Later in next section, we will se that this feature can help us select moving candidates and keep our search trees relatively small.

Table 1: Pattern recogonization

Pattern	Score
-xo-	1
-o-	10
-xoo-	10
-oo-	100
-xooo-	100
-ooo-	1000
-xoooo-	10000
-oooo-	100000
ooooo	10000000

3.2 Board Evaluation: Point-wise Approach

Having got point scores, we can evaluate the whole board through integrating these scores as the following formula:

$$S = r \cdot (\tilde{S}_1 - \tilde{S}_{-1}), \tag{4}$$

$$\tilde{S}_r = \max \left\{ S^r_{\text{point}}(p) \mid \text{Board}(p) = r \right\}. \tag{5}$$

Here, $r = 1$ represents our agent, $r = -1$ the rival. p are the points in current board, and S^r_{point} is the point score of r, described in Section 3.1. Obviously, this policy can both dig details and find most important patterns.

3.3 Speed-up Strategies

According to our evaluation policy described in Section 3.1, we can find that every time there are a new move, we have to re-score points on the board. Unfortunately, because movings are very frequent in our searching processes, it can be extremely slow if we re-score all the points on the board in every direction.

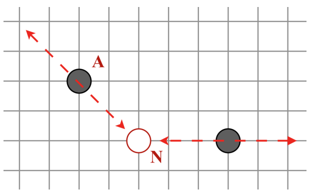

Figure 3: Score updating

To speed-up the score updating process, we take two stategies here. Firstly, because

[1]Different orders of the patterns and scoring for either players (agent and rival) are equivalent. Also, compositions and other more complicated patterns are not listed here

Gomoku chess playing usually occurs in a local area (like a 6×6 block), we here assume a new move on the board only influence its neighbor points. Secondly, We only update score of neighbor points in one direction, other than all the four directions. For example, in Figure 3, a new move N will trigger score updating processes. If a certain point A is near to N, like with distance $\leqslant 6$, we perform score updating for A. And we only update *right diagonal* score of A here, beacuse N has no impact on patterns on other directions of A. With appropriate cache technology, we can finally get evaluation system with high-performance.

4 Optimization Methods

Our Minimax Search implementation with alpha-beta pruning has already got good performance, but it can only search 3 layers. Otherwise, it will exceed the time limit. We use several methods to optimize our minimax search from different perspectives.

4.1 Candidate Pruning and Sorting

In Section 2, we introduced how to generate our candidates: select those points with neighbour in distance 1. Compared to take all empty points on the board, this has largely reduced the width of each search layer. But we know that when you make an open row of three `(-ooo-)` or a blocked four `(-oooox-)` or open row of four `(-oooo-)` or even a row of five `(-ooooo-)` which will lead you to the win the opponents have to block you immediately, or they are definitely about to lose. This is called **forced move**. So in this case, there are some nodes that can be pruned.

In detail, when we want to generate candidates for next layer, we examine the point score of every point with neighbour. If any of point can make a row of five, we directly use this single move as our candidate. If there are any points that can make a row of four, we return all of those four-points.

Moreover, we recognize that the performance of alpha-beta pruning is greatly infected by the order of search nodes. For example, a max-node should acquire the max-value from candidates as soon as possible. If we sort all candidates properly, the speed can be improved greatly. So, we sort all candidates according to its point score described in Section 3. Experiments show that it can largely reduce the nodes we expanded.

4.2 Greedy Pruning

In Minimax Search, alpha-beta pruning is a safe pruning method, in other words, that will surely find the best result in fixed depth. We want to relax it to a sub-optimal version, in order to find the approximate 'best' result in order to increase search depth.

A simple but effective way is to use *greedy pruning*. In Section 4.1, we introduced candidate pruning (which is safe) and candidate sorting via point value. We know that best move of either player or opponent has a large probablity to lie in those with high point value. In Greedy Pruning, we only keep top-b candidates in each layer, thus to explore higher depth in same amount of time.

This is a trade-off bwtween search breadth b and search depth d. The complexity of search tree can be estimated as b^d. With close complexity b^d, will larger b or larger d improve the result? We conducted a contest on several AIs with close complexity(and close computing time). The result is distinguished.

We conducted an experiment on opening 1 given on website for each pair of AI with different (b, d). The result is shown in Table 2. It is shown that the balance is about 6 candidate each layer.

4.3 Checkmate Seeking

Checkmate is a term often used in Chess, in which a player's king is in check (threatened with capture) and there is no way to remove the threat. We have a similar case in Gomoku, where one player keep attacking with force move and the other player has to do blocking - the forced move. If the attacker can win no matter where the opponent place its piece, we call it a checkmate. Example is shown in

Table 2: Contest between AIs with different search breadth and depth

(b,d)	**(7,4*)**[1]	**(6,5)**	**(5,7)**	**(4,12)**	**(3,-)**[2]
(7,4*)[1]	-	2:0	2:0	1:1	1:1
(6,5)	0:2	-	1:1	0:2	1:1
(5,7)	0:2	1:1	-	0:2	1:1
(4,12)	1:1	2:0	2:0	-	0:2
(3,-)	1:1	1:1	1:1	0:2	-
Total	2:6	**6:2**	**6:2**	1:7	3:5
Score	2	**6**	**6**	1	3

[1] Due to computing complexity, we use 4 candidates in first 4 layers and 3 candidates in latter 3 layers.
[2] We use all the candidates in this AI.

Figure 4.

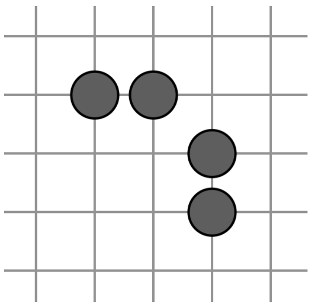

Figure 4: Checkmate example

Checkmate is also a strategy to increase the search depth. It is like Minimax Search in many aspect, but each node returns a boolean variable (True/False) instead of a value. The boolean variable represents whether this way can definitely lead to a win. In max-node, we only search for candidates that can attack, those can have force move `(-ooo-, -oooox-, -oooo-, -ooooo-)`. In min-node, we only search for those candidate that can defend this attack (block) or make a more severe attack (like using '`-xxxx-`' to defend '`-ooo-`').

The complexity of checkmate is also b^d, but because b is very small here, we can search for more than 10 layers in negligible time. So we use checkmate each time before our minimax search to find the oppotunity to directly get the win.

5 Conclusion

In this report, we implement Minimax Search, introduced point-wise policy to establish an efficient evaluation system, and adapt candidate pruning, greeady pruning and checkmate seeking to increse the depth of tree search. As a result, All of these proposal can effectively promote our agent's capability and robustness.

References

Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Zhentao Tang, Dongbin Zhao, Kun Shao, and Le Lv. 2016. Adp with mcts algorithm for gomoku. In *Computational Intelligence (SSCI), 2016 IEEE Symposium Series on*. IEEE, pages 1–7.

Junru Wang and Lan Huang. 2014. Evolving gomoku solver by genetic algorithm. In *Advanced Research and Technology in Industry Applications (WARTIA), 2014 IEEE Workshop on*. IEEE, pages 1064–1067.