Improved Alpha-Beta Search in GOMOKU

Midterm 1 report for final project

Xuetian Chen 19307110189 Yifan Li 19307110499

November 13th 2021

Abstract

We implemented a Gomoku AI using Nega-Max Search with alpha-beta pruning. In details, an efficient evaluation function is designed to evaluate current board and return a score when the search depth touches its limit. Instead of copying the whole board, a dynamic updating operation is taken when getting a successor of a node. Neighbors searching and forward pruning for possible actions and pre-evaluating for each action can extremely improve the performance in time. Besides, when the AI meet a checkmate, it will immediately react without considering other unimportant actions.

1 Gomoku Introduction

Gomoku, also known as “Five in a Row” is a classic strategy game that is known in many countries around the world. In Gomoku, each player takes turn to place their own piece, and whoever gets 5 straight pieces in a row in either vertical, horizontal or diagonal direction wins. Pieces are never taken off the board, and once the whole board is filled, the game draws.

Nowadays, Gomoku can be solved with computer. The difficult point is, for example, for a 20×20 chessboard, we have $(20 \times 20)!$ states, due to the outcome of Gomoku can only be judged at the end of a game, limited by the calculation power of the computer, we cannot exhaust all the game situations to help us to win. And it is also not necessary, for defeating most humans. Therefore, we consider using some algorithms to reduce our calculations. Fortunately, the effect is completely acceptable. The main algorithm used in this article is Minimax with Alpha-Beta Pruning, which is commonly used in human-machine games.

2 Adversarial Search

In this implementation, the basic adversarial search algorithm we used is Minimax search. Further, Alpha-Beta pruning is used to reduce the search branch. After several rounds of code version updates, in order to make our code more concise and transferable, we use the optimized version of minimax, Negative Max search.

2.1 Minimax Search

Minimax search is a decision-making algorithm. This algorithm is for two-player games, in which both the players are playing optimally. It considers two players Min and Max, and Min always picks up a minimum score from state and Max always picks up maximum score. The score for each game move is decided based on some heuristics. The algorithm constructs trees and follows a backward approach.

However, the Gomoku chessboard has a total of 400 points that can be chose, making the number of branching factors in search quite terrible. A good way for improvement Alpha-Beta Pruning.

2.2 Alpha-Beta Pruning

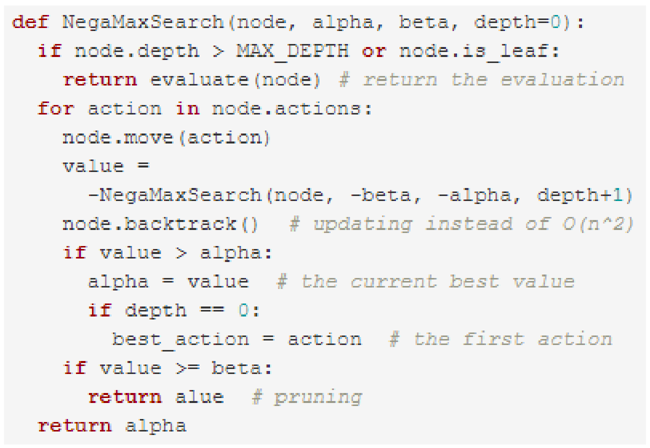
Alpha Beta Pruning is a search algorithm that is used to decrease the number of branches that are evaluated by the Minimax search. It passes two additional parameters in the Minimax function, namely Alpha and Beta, that represents the best score that Max guarantees as well as the best value that Min guarantees at that level. Alpha here is negative infinity and Beta positive infinity, that is, both side begin with with their worst possible score..The specific pseudo code[1] is as follows:
The effect of this algorithm is very significant. It limits the search time to more promising sub-trees, which enables a deeper search and reduces computation during the Minimax search.

2.3 Negative Max search

In the implementation, we use Negative Max (Knuth and Moore, 1975) Search to simplify our Minimax Search. In the Minimax Search, we consider two types of nodes: max and min. In the max node, the agent will maximize its successors' scores, and in the min node, the agent will minimize them. And the score of a board is the same for both max node and min node. But in Negative Max Search, all nodes are max nodes, and a node's score is the max of

$$-1 \cdot \textit{its successors' scores.}$$

Besides, the idea of alpha-beta pruning can be used in Negative Max Search with little modification: only update alpha when $value > alpha$, only prune when $value \geq beta$, and the (alpha, beta) of a successor is (-beta, -alpha) of its predecessor. Here is the pseudo code of Negative Max Search:



```
def NegaMaxSearch(node, alpha, beta, depth=0):
  if node.depth > MAX_DEPTH or node.is_leaf:
    return evaluate(node) # return the evaluation
  for action in node.actions:
    node.move(action)
    value =
      -NegaMaxSearch(node, -beta, -alpha, depth+1)
    node.backtrack()  # updating instead of O(n^2)
    if value > alpha:
      alpha = value  # the current best value
      if depth == 0:
        best_action = action  # the first action
    if value >= beta:
      return alue  # pruning
  return alpha
```

Figure 1: pseudo code of Negative Max Search

However, NegaMax Search can't do better than Minimax in time. It is just a simplification in code implementation.

3 Evaluation

3.1 Setting Points

Given a chessboard state, we need to get its score. To do this, we firstly to learn Gomoku's basic strategy. From a human point of view, no matter which side wins, they are closely related to the following several chess types, which are listed in reverse order of importance:

- *Five* : It means that one player's chess are connected to five in a certain legal direction, that is, it wins. In Gomoku, five or more consecutive chess are considered to win.
- *Open Four* : Four identical chess are in a row, and neither side is blocked by the opponent's chess (such as · ○ ○ ○ ○·). An *Open Four* means an immediate win for the player who makes it. This is a chess

type whose importance can almost compete with *Five*.

- *Four* : This chess type is still composed of four chess, but it may be blocked by the enemy on one side (· ○ ○ ○ ○●), or broken in the middle(○ ○ · ○ ○). This type of chess seems to be about to make five in a row, but in fact it is easy to be cracked by the enemy.
- *Open Three* : In short, this chess type can form *Open Four* (such as · · ○ ○ ○ · ·). We should treat it carefully to prevent it from becoming *Open Four*.
- *Three* : Corresponding to *Open Three*, this chess type can generate *Four* (such as · ○ ○ · ○●). Obviously, its importance is greatly reduced because it is easy to crack, and we can even wait until it forms *Four* before defending.
  But it should not be underestimated. There are many situations in *Three*, and some situations are also related to the formation of forbidden hands.
- *Open Two* : Which can form *Open Three*(such as · · ○ ○ ··). The *Open Two* chess type may seem harmless, but in fact, it is very important, especially in the opening stage. If we can form more *Open Two*, we will be able to form more *Open Three* in the follow-up, increase the chance of win.
- *Two* : Which can form a *Three*, this kind of chess type has the least hidden danger (● ○ ○ · ·).

Of course, not only these situations will occur during the Gomoku game, but other situations are of lower importance, so we will ignore them.
Based on the above analysis of chess type, we assign 8 kinds of chess patterns in the program as follows:

Other type	Two	Open Two	Three
0	10	100	100
Open Three	Four	Open Four	Five
1000	1500	10000	1000000

Supplementary explanation for the above score setting:

1. The *Open* chess type of each level will have the same score as the ordinary chess type of the upper level. This is because even if the ordinary chess type of the upper level contains more chess, it will also suffer more restrictions (such as being blocked on one side), so we simply think that their utility is close.
2. If you look closely, you will find that we set the score of *Four* to 1.5 times higher than that of *Open Three*. This is because when it is possible to create a *Open Three* or a *Four*, we will give priority to creating a *Four*, which has a higher chance of winning. However, the score of a *Four* cannot be much higher than that of a *Open Three*, because once the opponent establishes two *Open Three*, he must win. In other words, the scores of two *Open Three* should be at the same level as the *Open Four*.

3.2 Chessboard Evaluation

For a given board, we respectively search for all the chess types formed by both players on the board, and weight them to get the final board score. When searching for a chess type on a chessboard, although the number of chess we enumerated before does not exceed 5, we still need to use 9 chess as a complete chess pattern while judging. This is because we don't know the position of the chess to be evaluated in the pattern (for example, it may be in the middle or on the edge). The specific process of evaluation can be abbreviated as the following two steps:

1. Obtain the true range of the chess pattern to be evaluated, that is, the longest sequence that does not include the opponent's chess.
2. Judge the distribution of our chess in the range of this chess pattern by different conditions, and match chess types above.

In addition, when evaluating a certain chess, all chess in the chess pattern should be recorded as *Evaluated* to avoid double scoring. Finally, all the chess types are weighted according to the scores of the chess types enumerated above, and the scores of both parties can be obtained.
Then, when both players score separately, the entire score is obtained through the following formula:

$$S_{board} = S_{black} - r \cdot S_{white} \qquad , r < 1$$

Here, Black represents the first mover, and white represents the second mover. $r < 1$ is because in Gomoku without variants, the first mover is always more advantageous than the second mover. If we are the first player, then we should pay more attention to our own scores, increase our own advantages, and win as soon as possible. If the opponent is the first player, then we should pay more attention to the opponent's scores, defend in time, and reduce the advantage brought by his first move. In the actual implementation, after many attempts, we finally took $r = 0.9$, which showed a relatively good performance.

4 Updating

In the Gomoku game, there is only little difference between the boards before and after an action is executed (what we called a predecessor-successor pair). Hence, it is unnecessary to use *copy.deepcopy()* to generate a node's successor in the depth-first NegaMax Search algorithm. We can take an action on the current board, compute the evaluation of the board after moving, and then withdraw the action.

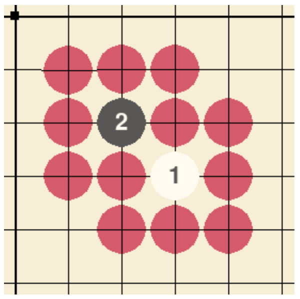
4.1 Withdraw

In the implementation of *Board.withdraw()*, we dynamically updating the attributions in the class Board, such as empty positions, black chess positions, last action, and the "hall" (see section 5.1). With the method *Board.withdraw()*, the generation of a successor only take $O(1)$ time and $O(1)$ space, which tremendously improves the performance of search algorithm.

5 Reduction

5.1 Neighbors Search

When playing Gomoku, taking an action far from the center of the current game is not a good decision. We only care about the positions that has nonempty neighbors. In our implementation, the neighbor of a position **P** is defined as the positions in the radius 1, centered in **P** square. When there are multiple nonempty positions on the board, we define the "hall" of the current board as: the set of empty positions that has at least one nonempty neighbor. The following figure displays the "hall" of the board with only two nonempty positions 1 and 2.



Besides, we have tried to change the radius of the hall to 2. The computation complexity rises rapidly, while the most decisions made are still in the radius 1 hall.

5.2 Pre-evaluating

When performing Alpha-Beta pruning, the order of successors matters a lot. In a max node, if its successors are in a decreasing order with respect to their true values, we can only consider the first successor and prune the others, similarly for min nodes. However, we do not know the true value of the successors, but we can pre-evaluate each successor. For every possible action in the hall, we evaluate the board score after taking this action, and use the evaluated scores (in *0-depth*), instead of true value, as the key for sorting (increasing for min node and decreasing for max node, similarly for the NegaMax algorithm), in order to increase the possibility of pruning in the future.
The additional computational cost is tiny, compared to the improvement of pruning. The idea of this method, compute to avoid computing, may be seen again and again in the improvement of minimax search.

5.3 Forward Pruning

When performing Minimax search, even with neighbors searching and pre-evaluating, the high branching factor still brings unacceptable computational cost. To make a further improvement on pruning efficiency, we introduce the forward pruning in branches where an intermediate evaluation leaves little hope for success (Go-moku and threat space search, L.V. Allis H.J. van den Herik)[2]. The idea of forward pruning is that the actions consider better in the deep depth would be also better (at least not bad) in the current situation. Therefore, we can just consider some successors with higher *depth-0* scores (see Pre-evaluating in Section 5.2) and prune the others directly. But there is no guarantee to find a best action in *depth-4* search by just consider the top few actions in *depth-0* search. Furthermore, it is reasonable that the number of possible successors can diminish in depth. For example, in *depth-0*, we compute all successors and preserve 10 successors, and then in *depth-1*, we compute the 10 and only preserve 8 successors, $\cdots$ The reason is that the risk of forward pruning decision in the shallow layer is higher than in the deep layer, because it may cause more mistakes with irreversible bias.

5.4 Necessary Reduction

It is obvious that if there exists a live four, its owner guarantees to win in the next turn. Hence, if we find a position being a key point of my (or opponent's) *Open Four*, it's necessary to take this action promptly. Otherwise, we will lose the chance to win (or immediately lose) the game. Similar patterns are seen in *Open Three* circumstances. Therefore, for every possible action in the hall, before computing the board score after the action, we can find if it is an important position in current game, for both players. If there is some important positions, the set of possible actions shrinks to the set of important actions, which gains a lot improvement when the necessary action exists.
There are 4 important cases that guarantee to win or loss.

1. My *Five*
2. Opponent's *Five*
3. My *Open Four*
4. Opponent's textitOpen Four

We detect every possible action in the hall if it is the key action of these 4 specific types. Note that the important levels of the above 4 cases decrease in order. For example, even if an opponent's *Five* point exists, when I take action on the key positions to generate my textitFive, the game is terminated and I win immediately.
If there are no important positions to accept, we perform Pre-evaluating (in Section 5.2) on

the board scores and arrange actions to accelerate pruning.

6 Looking into the Future

6.1 Checkmate

After implementing the necessary reduction (Section 5.4), our AI will not make trivial mistakes that give the opponent chances to win simply. However, a clever opponent will not make trivial mistakes, either. Offensive strategies are needed to gain a victory, which our AI lacks the ability to take now. Checkmate is a terminology in many board games, which means the opponent's move forces me to take defensive measures, otherwise, you will lose the game immediately. The pattern of *Open Three* and *Four* are both checkmates. According to L.V. Allis [2], almost all non-trivial winning is the result of a threat sequence. If such a winning threat sequence exists in the current board state, we can simply follow the sequence and take actions, we will win in some turns. Therefore, our AI will improve a lot if we implement threat sequence search into it. Necessary reduction, shown in Section 5.4, is the simplest case of checkmate sequence, which we have successfully implemented.

6.2 Forbidden moves and the Evaluation

There are still something needs to be improved in the evaluation function, which else will result in some stupid decisions. For example, crossed *Open Threes*, known as forbidden moves in *Renju*, guarantee to win. Therefore, when detecting necessary key positions (Section 5.4), we can add the cross threats into the important cases. Furthermore, we can take advantage of the forbidden moves' ideas. It is its effectiveness that makes a decision a forbidden move. In Gomoku of free rules, use these forbidden moves can easily form an effective threat.

6.3 Transposition Table

Another feature of board games such as Gomoku and Go is that there is more than one way to get to a current state. For example, just change the order of one's actions. This feature may result in repeated evaluation of a board and a single position. If we construct a big hash table recording evaluated boards and positions (more accurately, a line), it just take $O(1)$ time to fetch the information when the next time we need to evaluate them. Thanks to Zobrist Hashing (Albert Lindsey Zobrist, 1969), an effective way we encode the board game, we can update the hash value in $O(1)$ time, with little probability of hash collisions.
Zobrist hashing starts by randomly generating random $64-bit$ integers for each 20×20 positions on the board game for 2 colors. Then the hash value of a state can be computed by performing XOR operations of all 400 $64-bit$ integers. Whenever an action is taken on the board, the hashing value of the new board can be computed by performing two XOR operations on the hashing value of the old board.

References

[1] Russell, Stuart J., Norvig, Peter. 2010. Artificial Intelligence: A Modern Approach (3rd ed.). , Upper Saddle River, New Jersey: Pearson Education, Inc.

[2] L. Allis, H. Herik, and M. Huntjens. Gomoku and threat-space search. Computational Intelligence, 12, 10 1994.