AI for gomocu

Anonymous ACL submission

Abstract

Computer Artificial Intelligence (AI) of gomoku has long been a focus on game research.This paper designs three different effective algorithms, fast move on local valuation (FMLV), Multi-level Negascout, and algorithm on Monte-Carlo Tree Search (MCTS) to play gomoku intelligently. In addition, VCT, VCF and an opening library serve as the supplement of the algorithms, considering the special properties of the gomoku rules. With experiments on these three algorithms, FMLV turns out to be the best, with a winning percentage larger than 98 percent versus humans or other AI.

1 Introduction

Application of AI to board games has been an advaced issue in player centric research for a long period of time. Chess is the best-known example and has received particular interest culminating with Deep Blue beating Kasparov in 1997. Still, more research lies in learning techniques that allow a computer to learn to play chess, rather than being told how it should play.

In the past few years, AI board game players have been created to be world class ones in chess, gomoku, go, *etc*. In 2015, Alpha Go beat the top human professional Go players, enabling board game AI to attract the public's attention again. A variety of board game algorithms have been created and proved potent in actual competitions.

Gomoku, as one of the most popular board games in the world, has developed a variety of professional AI, most of which are based on the Minimax theorem raised by John von Neumann. Since every stone has no particular value itself (unlike the chess hierarchy), the board situation can only be judged on the locations and distributions of stone pieces. It's quite hard to form an accurate value function. But a complete search of n moves requires $p!/(p-n)!$ board situations, where p is the current number of legal moves. It's almost an impossible task due to the huge amount of calculation.

To solve the problem, we proposed FBLV, using the valuation of one move to decide the next step, which outperforms the majority of humans and other AI. Besides, optimization of Minimax method is another algorithm in this paper, using Principal Variation Search (PVS) to prune better, and adopt different depths in nodes expansion to decrease the wasting time searching some bad nodes. Finally, we create a method on MCTS, using sampling instead of traditional searching.

Considering the specialty of gomoku, we use gomoku strategy like Victory of Continuous Three (VCT) and Victory of Continuous Four (VCF), so as to win the game quicker. Opening library is also used to guarantee a best starting of the whole game.

2 Background

Gomoku is a traditional two-player strategical board game. Two players with Go pieces (black and white stone) on a board with 15×15 intersections. The winner is the player who first obtains an unbroken row of five pieces horizontally, vertically or diagonally. Black plays first, and has great advantage. *L. Victor Allis* has been proved that black could force a win. So many extra rules are applied to reduce that advantage.

In the field of gomoku AI, the mainstream algorithms are based on alpha-beta pruning. In Gomocup, which is an international gokumo AI competition, Yixin from China ranked first in 2017, core of which is also based on alpha-beta pruning. Methods like MCTS, machine learning can also be applied the gomoku AI.

3 Fast Move Based On Local Valuation

3.1 Background

Most of Gomoku AIs espacially those based on $\alpha\beta$ pruning, are using value functions that evaluate the whole chessboard. For $\alpha\beta$ pruning, for which a evaluate method is needed, those value functions performs better. However, when $\alpha\beta$ pruning is not necessary, we may use a faster and easier way to evaluate which point is the best for the situation. That is the original algorithm, **Local Valuation**, we adapted.

3.2 General Idea

3.2.1 Searching Domain

For a Gomoku game, not all points are reasonable. For example, it has been proved that move on or near tengen[1] is better than boarders. Also, for most cases, you will not move away from other chess pieces.

In this project, we used a *list* to save which points are reasonable, namely *temp*. In order to adjust speed of different algorithms, we made the searching domain adjustable, namely *dist*. The concrete algorithm is that, we chose a square of side length 2×*dist* and centering the chess piece for each chess pieces on the chessboard. You can find the searching domain by the dots drawn in local valuation algorithm, the dots are colored gradual red to blue.

3.2.2 Valuing the Step

The chief thought of this algorithm is to evaluate only based on where you will move. That is, to maximize the value of next step. The value is devided into two parts, one is the reward of get a better shape, the other is the reward of destroying opponent's shape.

[1]The center of the chessboard.

3.3 Implementation

3.3.1 Generating Valued Lists

The code of this algorithm is implemented in *value_function4.py* and *sub_structure.py* . In *sub_structure.py* , we get four lists of a certain point on the chessboard. The four lists indicate four directions, row, column, left diagonal and right diagonal. In these directions, the lists contains 0, 1 and -1, representing spaces, black chess pieces and white chess pieces on the chessboard. The list also contains a 2, representing the point we are calculating.

After receiving these lists, we can calculate three values, original value, self value and opponent value, respectively regard the 2 in the list as 0, 1 and -1 if the next player plays black, and 0, -1 and 1 if he plays white.

3.3.2 Evaluating the Lists

When calculating the values of the list, there are several methods. The traditional one is to ergodic the list to find what shape there is in the list. However, since this part is one of the most frequently used functions, we need a high-performance method to calculate the values.

Shortening the List First of all, if we move at somewhere on the chessboard, only places near the position we moved is affected. Without loss of generality, we assume the next player plays black. Actually, the precise region is, from the position he want to move, seek for the first chess piece occurs in the list in two directions (since the list is one-dimensional), only the region between these two pieces will be affected when calculating the shape of black chess pieces. Specially, if there is no white chess pieces in any direction, the maximum size of the region is 5 on each direction, because what we want is to link a 5-in-a-row, chess piece with distance more than 5 has no contribution to the chess piece we move.

Evaluating the Shorten Lists After shortening the lists, the length of the four lists of four directions are not more than 9, because the length of each direction is 4.

We can easily ergodic the shorten lists to find shapes and evaluate them. But there is a faster method. Since there are only 0s and 1s

in the list, the number of different types of list is less than $2^10 = 1024$, which means we can just calculate the list, instead of evaluating by looping.

In *vv.py*, we saved all the possible lists and their types in a *dictionary*. In *value_function4.py* , we load the dictionary and evaluate the different types. The reason why we didn't just evaluate the list is that marking their type could make us easier to change the values of different types of shapes. For future work, we can use this as descending dimension for reinforcement learning.

3.3.3 Finding Next Move

In function *score_in_table* , we calculate the three values mentioned above. In practice, actually, we calculated four values, because the original values are always different for black and white.

It is easy to find that, compared to original list, moving a chess piece, for example, moving a black one, will make the value of black larger. Thus, we calculated the difference between moved situation and original list.

Typically, because the advantage of one more chess piece, we will encourage black to be more aggressive, especially in opening stages. Thus, we gave the opponent a punish coefficient, and for black player, the coefficient is larger at the beginning of the game. Furthermore, in order not to make all the game same, a random number is added into the punishment.

Finally, we calculate every reasonable point in the searching domain, the point that gets largest value is the point that we will move. This is what we do in *valueFunc4*.

3.4 More Applications

This algorithm is much faster than others, but it is no use to continue calculate more layers. Because we can not calculate the reward of several steps. Generally, if you get more reward in one step, your opponent will get a larger reward in next step too.

But because of its high speed and fairish moving quality, we can use this method in many other algorithms, for example, the MCTS we used in the following sections.

4 VCT and VCF

4.1 Introduction

Calculating VCT is a basic skill for Gomoku and Renju players, and it is also a important part for Gomoku AIs to judge whether there is a certain win for the player.

VCT is short for Victory of Continuous Three and VCF is short for Victory of Continuous Four. In Gomoku, if a player can not connect into five, he must response to the opponents' four, or his opponent will win in the next turn. Thus, if we connect into four, we can easily know where the opponent moves. By continuously connect into four, we might get several different points that the opponent must defend, but he can only move one chess piece at on step, so we might win by continously four. This is how VCF works.

VCT is a little bit more complicated than VCF. Another situation that the opponent must defend is called *living three* , which means a shape that if the opponent does not defend, the player can make a single move to make two or more points that is possible to connect into five in one step. Instead of continuous four, VCT allows you to continuously living three or four. What needs to be explained is that, in VCT, the opponent may not defend, but achieve his VCF, because living three needs two steps to achieve victory.

4.2 Preparing

4.2.1 Detecting Fours and Living Threes

We used a original method to detect living threes. The method can also be slightly changed into detecting fours, but it has been used by others.

The chief thought is, if you want to connect into five, there must be a continuously list with a length of 5 that does not contants any opponent's chess piece. Then, if there is four chess pieces of your color in the list, we say you have a four. If your opponent does not response this four, you can win in next turn.

If there is three of your color, we can only say you have a three, but we want to see whether you have a living three. To solve this problem, we should see how we store these fours and living threes in our program.

4.2.2 The Magical Living

In class *TABLE*, we implemented *TFCheck* method to search for fours and living threes, and *DFour*, *LThree* to save them. After a move, we will update the points for the two players to connect into five, four or living three. The points to connect into five is easy to understand. Let's have a look at the points to connect into four. As we know, in Gomoku, there are two types of fours, one is those has two different points to connect into five, the other only have one. In our algorithm, since we ergodic the list to detect all the possible list with length five, the points to connect into fours of the first type will be calculated twice.

After understanding what is done in detecting fours, it is similarly for detecting living threes. The points to connect into fours of the first type will be calculated twice or three times in one line. Thus, we can only count the points occured for two or more times in the detection of a single list.

4.2.3 Another Local Searching

We used local search again in searching for fours and living threes. If we want to a move at somewhere, we will delete the fours and living threes in the lists of its four directions. Then, after make the move, calculate these four directions again to update. In this method, only 8 lines needs to be calculated, compared to over 60 lines to detect the whole chessboard.

This subsection is what we did in *TFCheck* and *InLine* method.

4.3 Detecting VCF

4.3.1 Background

The chief algorithm we used is found in *Baidu Tieba*. Unfortunately, when I want to add it into bibliography, the post seems has been deleted. Some other algorithms in *csdn* and *GitHub* are not as good as we want. Some of them can not search all the VCTs while some are based on $\alpha\beta$ pruning, which are not what we called VCT in Gomoku.

4.3.2 Algorithm

The algorithm is implemented by iteration. The chief thought is, for each point that we can connect into four, move on that point, then the opponent has the only response–to defend this four. After these two steps, we iterately detect VCF of the new situation, *i.e.*the situation after these two steps. When all the points to connect into four is used up or the opponent connects into five, then the VCF is failed, if we find a point to connect into five, the VCF is succeed.

The green points on the chessboard indicates the AI is searching for VCF on that point. This subsection is what we did in function *VCF* in *VCT.py* .

4.4 Detecting VCT

VCT is much harder than VCF, mainly in two reasons. One is what we have mentioned above, the opponent may win by his VCF. The other is that the opponent may have different defend methods towards your living three.

The first one can be converted into the second one, because four can also be considered as a defend method.

To solve the second problem, we should try all the defend methods of the opponent, if all the methods are proven to be useless, VCT is succeeded.

This subsection is what we did in function *VCT* in *VCT.py* . The main difference to VCF is that there is a loop for checking different defend methods.

5 Opening Library

As we all know, Gomoku has been proved to be certain win for black player. Thus, if we can tell AI how to achieve this certain win, then it will be unbeatable when using black.

The key algorithm for the opening library is to combine similar situations into one situation to reduce the size of the library. We implemented transpose and rotating to acheive it. Since it is easy and not so important, we will not introduce the specific method. We saved the library in a *txt* file. When the game starts, we load and transform it into a dictionary, if the situation is in the dictionary, there will be no calculation.

We think this method is mainly implemented by mechanical labour. Thus, not all library are added. But when using a library with around 100 situations for black, combined with other algorithms and VCT, the AI can hardly lose when using black.

6 Adversarial Search

According to competition records of Gomoku Cup, most of the top AIs in gomoku are based on the method of adversarial Search, alpha-beta prunning specifically. Using this method, we can build a game tree, simulating the following movements of both sides and decide the current move. Compared to the method of local valuation, the adversarial search focuses on the overall situation in space and time.

6.1 Traditional Minimax and Alpha-beta Prunning

The Minimax algorithm is a recursive method when deciding the next move in the multi-player game. The current state of the game is the root of the tree, which is depicted as *deep* = 0 in our program. The tree's nodes expand simulating the game of both sides. A value is calculated to indicate the superiority of the player over the other side, using the a heuristic evaluation function. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible moves.

Alpha-beta pruning is a common way reducing the nodes to be calculated in the Minimax algorithm. The algorithm dismisses the nodes that can be proved to be worse using history nodes data. This can improve the algorithm's speed by three or four times.

6.2 Negamax

Negamax search is a variant from the Minimax search applied in two-player games. It is relied on the fact that $\max(a, b) = -\min_{(}-a, -b)$. Instead of using two separate subroutines for the Min player and the Max player, it passes on the negated score to the opponent. This is just a code simplification over the traditional Minimax, without any improvement in the core of the algorithm. Apply the Alpha-beta pruning to the Negamax search, and we get the pseudocode below.

6.3 NegaScout

The algorithm is also called PVS(Principal Variation Search), which is a successful improvement of the algorithms above. The core lies in that it's much faster to judge whether a new node to be expanded is better, than to expand the node and obtain the exact value of it.To be more specific, NegaScout just searches the first move with an open window, and then every move after that with a zero window, whether alpha was already improved or not. If the node is proved to be better, than expand it. Otherwise, just skip to the next node. This "trick" method can prune to a large extant, reducing an order of magnitude according to the experiments.

In our program, we colored the first expanded dot in grey, and other dots based on the depth of the search.

Algorithm 1 Negamax

```
1: function ABNEGAMAX(board, depth, temp, maxDepth, α, β)
2:     if board.isOver() or depth == maxDepth then
3:         return board.evaluate()
4:     end if
5:     for next_node in temp do
6:         new_board = move(board, next_node)
7:         new_score = −ABNEFAMAX(new_board, depth + 1, −β, −α)
8:         if new_score > β then
9:             return new_score
10:        end if
11:        if new_socre > α then
12:            α = new_score
13:            states = next_node
14:        end if
15:    end for
16:    return alpha
17: end function
```

6.4 Multi-Depth Search

Based on our experiment results, we can find that with the increasing of the searching depth, the time is much longer. But for some nodes in our searching area, it's vain to expand in that level. Our idea is to abandon the nodes whose performance is bad in shallow search, and only search the remains nodes in a deeper level.

Take the human players' feeling into consideration, we take the "low" depth as two-level. During the 2-level search, we prune half of the nodes judging from the result of this level, and then transfer the remains nodes to a

deeper search.

We use the combination of multi-depth search and negascout.

7 Monte-Carlo Tree Search

7.1 Development and Strengths

From the first days of game programming mini-max enhanced with alphabeta pruning has been the algorithm of choice. All of this changed in 2006 when the first experiments with Monte-Carlo Tree Search (MCTS) began to appear and research on MCTS has exploded in the past decade. MCTS differs from classical mini-max game tree search in two major ways. Firstly, no evaluation function is needed in MCTS. Instead, random game playouts, sometimes called simulations or rollouts, in the MCTS act as a kind of sampling of the possible outcomes from various board positions, which in turn can be used to rate (evaluate) these different positions. Random playouts are essentially random games played from a given position until the end of the game with the goal of trying to judge who has the advantage in that position. The random games are often skewed so that more reasonable moves are more likely to be made . Secondly, MCTS builds the search tree so that more promising lines of play are more thoroughly explored in the tree than less promising ones. This differs from mini-max algorithms that examine different lines of play in a much more uniform fashion. We have learned that MCTS can dramatically outperform mini-max based search engines in games where evaluation functions are difficult to obtain, and especially in games with large branching factors.

7.2 Algorithm

We adopted UCT which is a standard algorithm of Monte Carlo tree search (MCTS) in our program. UCT has achieved remarkable successes in various domains including the game of gomocu. For game tree search, UCT constructs and evaluates a game tree through a random sampling sequence. At each time step, a playout involving Monte Carlo simulation is performed to improve the empirical estimation of the winning ratio at a leaf node and that of the ancestors of the leaf. For each playout, a leaf in the game tree is selected in a best-first manner by descending the most promising move with respect to the upper confidence bound of the winning ratio, UCB1. After it reaches the leaf, the score of the playout is determined by the terminal position, which is reached by alternatively playing random moves. Therefore, UCT works effectively without heuristic functions or domain knowledge. The fact is a remarkable advantage over traditional game tree search methods based on alpha beta search , because such methods require adequate evaluation functions to estimate a winning probability of a position. Generally, the construction of such evaluation functions requires tremendous effort.

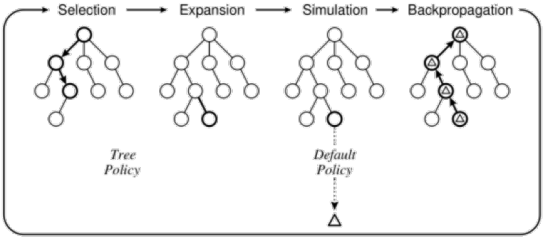

Figure 1: Principle of MCTS

7.3 Experiment

First, the search area is determined through the functions described in table.py. The depth of the search is set to be 20. Then, in each dot that can be chosen to be the next step, we do an iteration to update the winning rate of the dot. To illustrate this process, we shall take the first few steps as an example. After the black chooses (7, 7) as a start, we take (6, 7) to cope for example. Afterwards, the black will take the neighbors of (6, 7) as the search area whose distance from (6, 7) is less than $3\sqrt{2}$. For each dot in this area, we update their winning rate by motivating 1000 games which last for 20 steps at most. Then, the new rate is summed up by the rate achieved in these 1000 games and the former rate in a radio set in the function. With the updated winning rate, we choose the one with the highest rate as our choice. If there exist two or more dots whose rate are the same which may occur more often in the first few steps, we would choose the dot whose Manhattan distance is the least from the former choice.

With the steps introduced above, we finished a basic MCTS AI used for gomocu. Above all, it is a well-behaved AI that performs up to our expectation. However, this model has a few problems needed to be solved. First, AI may act ridiculously in the first few steps for it has no previewed knowledge about the situation. Second, the iteration may last for almost 1 min before AI can make the final decision. Though we can decrease the number of iterations, the performance of AI can be affected to some extent.

8 Experiments and results

8.1 Win rate

Due to the advantage of black, we cannot assess two algorithm simply by playing gomoku with them as black and white. Instead, we should consider its winning percentage on both sides synthetically.

Table1 shows the trials of different algorithms.These are the three main algorithms with the supplement of VCT and VCF. We do not include the opening library just because when using it, black's winning rate is 100 percent.(This indicates that our algorithm with opening library is quite powerful!) Based on the data, we conclude that the performance of FMLV ranks first, and Multi-level Negascout ranks second, MCTS ranks last. The bad performance of MCTS stems from lack of simulating rounds, which can be improved at the cost of time.

Table 1: Win rate of different algorithms

	Local	MCTS	Negascout
Local		50:0	49:1
MCTS	9:41		13:37
Negascout	33:17	49:1	

8.2 Time consuming

Figure 2 shows the time consuming in each step on average. We can see that when considering the speed, FMLV also performs the best. This is because it considers only the local valuation, while Multi-level Negascout consider the whole board, and MCTS uses the value calculated by it.

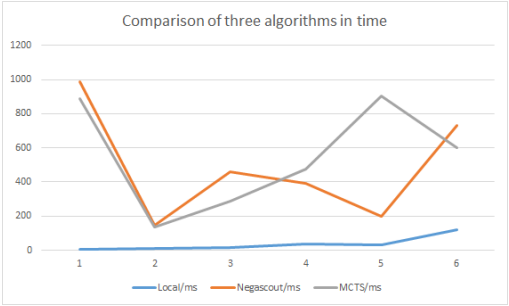

Figure 2: Speed of Algorithm

9 Conclusion

Three different algorithms have been proposed to act as a gomoku AI player, based on local valuation function, alpha-beta pruning and MCTS respectively.

The first method creates the valuation of one move, thus has an accurate and quick response to the current board. The second one using PVS and multi-depth concept to cut prunes as many as possible. The third way abandons the traditional valuation function and update the next move's performance by selection, expansion, simulation and back-propagation.Auxiliary skills of VCT, VCF and opening library are also applied to improve performance. These algorithms are tested with human players and each other (AI vs AI), which suggests the superiority of method FMLV in both time and winning rate.