Gomoku

Shun Zhang *15300180012* Donghao Li *15307100013* Pingxuan Huang *15307130283*

I. Introduction

In this AI project, we first explored various algorithms for Gomoku, such as Monte Carlo Tree Search, Genetic Algorithm, Threat Space Search. However, we found that these algorithms have some unavoidable flaws: the Monte Carlo Tree Search Algorithm requires too much simulations. If we only perform a very small search, we will have a very high probability to get some bad solutions. Genetic algorithms are also facing the same problem, requiring larger populations and iterations. Besides, genetic algorithms rely on some parameter turning and it is difficult to set them well. The third one is Threat Space Search Algorithm, and it is slow because it needs to constantly create threats and challenge the threat. But its power is quite considerable compared to the previous two randomized algorithms.

Afterwards, we refer to some successful AI, especially Yixin and its blog. We found that since the rules for the victory of Gomoku are local and the number of steps is very short, we do not need to consider the steps as far as Go. Therefore, the state of arts Gomoku AI is mostly based on Adversarial search algorithm and Alpha-Beta pruning. So we turned our attention to the scoring system and Adversarial search algorithm . It turns out that we are right. The first version of our AI defeated some of the lower-rated AIs. After that, we spent a long time constantly enhancing it, mainly in two areas. The first one is to speed up the search so that we can search faster and deeper. The second is to increase the VCX module to solve the horizontal problem (short-sightedness problem) and improve the level of AI. The final version's performance is surprising, none of our team could defeat it!

Subsequent articles will be presented in the order described above, with some possible future improvements.

II. Algorithm Exploration

A. MCTree

At first, we try to realize the algorithm of Monte-Carlo Tree Search, which can be referred to *mcts.py*. However, we found that because of limited computation resource and time, the MCTS algorithm often returns a bad result of simulation. Also, our board is 20×20, which is much bigger than the standard 15×15. The MCTS algorithm becomes more 'stupid'. (It performs well on a small board such as 10×10). For example in Figure 1, it is obvious that MCTS failed to defend in the left figure and the other one on a small board succeeded to do so.

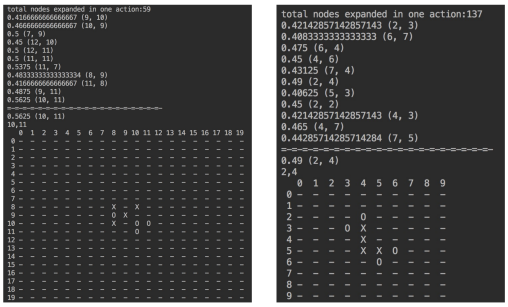

Fig. 1. 'stupid' and not so 'stupid' MCTS. Note that the board is printed below and those positions with win rate higher than 40% are printed above.

B. Mini-Max Tree Search

The most competitive and stable algorithm that we found is Mini-Max Tree Search, which is also known as adversarial search. The basic idea is covered in our course. Besides, there are some remarks:

- **leaf node** (evaluate)
  Given a certain deep, the leaf node is defined as the one with deep 0. And the evaluation at a leaf node is defined as follows:

  $$max_p\{score_{AI}(p)\} - max_p\{score_{opp}(p)\}$$

  where opp denotes the opponent of our AI, and p denotes available position.
- **generate potential nodes**
  A core question of Mini-Max Tree Search is that there are many available positions (nodes) at one layer, especially when the game is just started. Obviously, there are a lot of positions that are hopeless without any search. So how to generate fewer potential nodes without missing some valuable cases is quite essential.
  In our AI, we generate potential positions based on our scoring and we rearrange them in order of their scores, this will do great help in α-β pruning.
- α-β **pruning**
  This part is also covered in our course. In addition, another case for pruning is that when one player wins, there is no need to go deeper.

C. Genetic Algorithm

1) Algorithm: Generally speaking, We could roughly separate GA into 5 parts: coding scheme, original population, crossover and mutation operations, and the fitness function. limited by space, we will describe only the coding schema and fitness function:

First, according to original paper [2], we would set a list of coordinates as an individual, which means we could present an individual as $[A_1, O_1, A_2, O_2 \cdots]$(where A denotes agent and O denotes opponent). Then, simply put, our scoring/fitness function is based on a rating table, details will be presented latter.

By the way, GA-searching will converges when the best next move of the latest 5 generation are the same.

2) Implement: Implement process including two segments: fitness function & GA structure.

Fitness function (grading function):

To score one genome, we first grade each gene(coordinate) separately, then sum them up. When comes to single gene, we need to take both attack and defend into consideration, so we will check this table twice(set player as different color each time) and sum these 2 scores up. We need to set score as positive if it is Agent's move, vice versa. Because we need to take different directions, pieces and lines into consideration, we need to construct a system to match pattern:

First, we need to find all neighbors of one coordinate (pieces with same color which connect to this coordinate) in 8 direction(up, right...). Second, based on each neighbor, we will check its 8 directions to count its neighbors. After getting these data, we can do template matching for each neighbor, then we will select the best score as result.

GA structure: we coded 2 functions and 3 classes to implement GA. The 2 function are Mutation & Crossover. The 3 classes are Board, Population and Wuzi_GA:

- Board: Package cheeseboard, which could output available pieces, check weather one coordinate is free or not and update chessboard.
- Population: Simulate genetic process, which have the ability to generate primate population, to select individuals, to generate next generation and find out the best next move of one generation. By the way, according to paper [3], we choose to cross first and then mutate.
- Wuzi_GA: Apply GA on Gomoku, which will set select function, code individual and control general process.

3) Experiment & Enhancement: First, we added some optimization:

- To accelerate algorithm, we transformed duplicate checking(same gene cannot appear twice in one genome) from population-generation process to individual-evaluating process: if we find a duplicate gene, we will assign this individual -1 point.
- In order to encourage early victories, we directly assign the (negative) maximum value to the redundant steps(steps after agent/opponent win).
- if there is still no convergence when time runs out, we will choose the last one of most common coordinates from the best next move of latest 5 generation.

Then we did some enhancement in experiment:

First, we noticed that if one line are blocked in both side, grading function will see it as 2 lines blocked on only one side. So we made our select function check one whole line(in both directions) each time.

Then we noticed that our agent was not good at defending. First we think it is because too much steps considering will influence AI's judgment. So we set the length of genome as 2. Although not-defend problem can be resolved, AI will become short-sighted. So we refined our fitness function, to be specific, we directly assign the (negative) maximum value to the redundant steps if we meet 4 renju, and we changed the genome length into 4.

III. Algorithm acceleration

A. Fast evaluation

1) Make score dictionary: For a certain square, we assume that its engaging radius is 5. That is to say, only the 40 squares around it (5 squares on each side, 4 directions) that matters. Obviously, the four directions share the same number of patterns. So the dictionary for one direction is enough and we will only discuss the case of broad-wise direction.

Assume that now we have a square, denoted as x. the 10 engaging squares are denoted as $i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9, i_{10}$. For any $1 \leq j \leq 10$, $i_j = 1$ if this square is occupied by AI, or, $i_j = 2$ if this squared is occupied by opponent. Otherwise, $i_j = 0$ if the square is empty. This is consistent to the setting of *Piskvork*.

After that, for a pattern like $i_1, i_2, i_3, i_4, i_5, x, i_6, i_7, i_8, i_9, i_{10}$, it's represented as integer:

$$x * M^5 + \sum_{j=1}^{5} i_j * M^{11-j} + \sum_{j=6}^{10} i_j * M^{10-j}$$

which is equivalent to integer $\overline{i_1 i_2 i_3 i_4 i_5 x i_6 i_7 i_8 i_9 i_{10}}$ if $M = 10$.

However, here we choose that $M = 3$ to make the integers much smaller, in order to save disk usage also to avoid projection conflict. (Note that the pattern should be symmetric, but I enccounted some projection conflict during symmetric projection.)

Then, our job is to give every pattern a score. The basic score settings are:

TABLE I
THE BASIC SCORE SETTINGS

Pattern	Score	Pattern	Score
Five	10000000	-	-
Four	100000	BlockedFour	10000
Three	1000	BlockedThree	100
Two	100	BlockedTwo	10
One	10	BlockedOne	1

Our scoring is based on Table I and some Gomoku rules. Further details can be found in *getPointCache.py*.

2) Updating pattern: Till here, we have all patterns with scores, so the last thing to do is to update the patterns after taking a move.

A popular and sound method is to update within engaging area. After a move has been taken, only the 40 squares defined above should be updated according to our assumption. And with our score dictionary before, the update schedule is quite simple.

- Role x takes move (m, n)
  Also, we take a square $(m-2, n)$(if valid) for example. Note that (m, n) is the i_7 of square $(m-2, n)$, so the new pattern should be
  $$Pattern_{new} = Pattern_{old} + x * M^{10-7}$$
- Role x removes move (m, n)
  Again, we take a square $(m-2, n)$(if valid) for example. Symmetrically, the new pattern should be
  $$Pattern_{new} = Pattern_{old} - x * M^{10-7}$$

3) Get a square's score: In summary, one square has four patterns for four directions, respectively. The patterns are keys and their values are correlated scores. With all the preparations above, getting a certain square's score only requires the sum of scores for the four direction.

Also, with the score setting in Table reftab-score, the score of a square determines its actual gomoku pattern. For example, a (straight) four owns score more than 100000. A double three owns score more than 2000. This feature will do great help in other parts of our project.

B. Zobrist Hashing

Another method to accelerate our Mini-Max Tree Search as well as VCX is to create a cache for seen cases to avoid repeated calculations. Zobrist Hashing is a popular and easy way to do this job. It is quite suitable for games such as gomoku or chess.

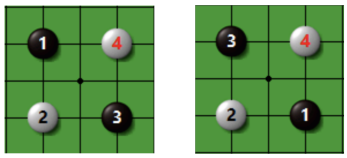

Fig. 2. An example for the advantage of Zobrist Hashing

For example in Figure 2, the different sequences lead to the same board, which means that when we decide our next move, the order of the past sequences does not matter. Zobrist Hashing is just an algorithm that can hash the board with same patterns, despite of how the board become to the current one. The steps are also simple:

- generate two random matrixes (large integer) of size $L_1 \times L_2$ for both AI and opp, denoted as M_{AI} and M_{opp}
- generate a large integer for the initial board, denoted as $code$
- when AI takes move (i, j), $code = code \wedge M_{AI}[i][j]$
- when opp takes move (i, j), $code = code \wedge M_{opp}[i][j]$
- when AI removes move (i, j), $code = code \wedge M_{AI}[i][j]$
- when opp removes move (i, j), $code = code \wedge M_{opp}[i][j]$

Here $\wedge$ is the XOR operation and $L_1 \times L_2$ is the size of the board.

IV. VCX

A. Introduction

In the minimax search, we calculate the score for all candidate points. According to the existing technology, namely Alpha-Beta pruning and limiting the number of candidate points (witch is not safe pruning), we can at most consider 6-level search. (If there are too many candidate points, we can't finish the 6-layer search!) This means that we can't perform calculations at any later point. And in many cases, we can force our opponents to make choices by constantly generating threats to win. This will reflect the problem of extremely small search, that is, the horizontal line effect.

In order to overcome the horizontal line effect, we first think of the method is to increase the depth of the maximum search. This method is not advisable because the amount of calculation increases exponentially with increasing depth. We do not have sufficient resources to deal with deep searches, and some killing pattern is often hidden after seven or eight steps. It is difficult for us to adopt a search with too much nodes.

Here we use the strategy of ”Killing”, that is, to find a winning strategy to win the game. In the process, we actually did a very small search, but in order to increase the depth of the search, we reduced the width of each layer. That is, the number of candidate points. Originally it would consider things such as ”Open Two” and but VCX would only consider Open Three and Closed Four, so it would reduce the number of candidate points.

Here we propose a new concept: VCX, where X could be ”T” for Three or ”F” for Four. This concept means that we could control the game and win.

In actual testing, we could even consider the situation after fifteen steps. Keep a quick pace. And, this max and mini search only has the concept of winning and not losing, there is no concept of utility value, if it finds it will return the optimal position. Note that what is generated here is a sufficiently unnecessary condition that there are some killings but we can't find them.

Since the VCX is a depth-first search, we hope to find the optimal solution, which is the fastest killing of the opponent's solution. Otherwise, there will be a situation of ”teasing opponents”, that is, constantly flushing without quick victory.

And we have adopted two strategies for VCX: VCF is introduced separately in VCT and below:

B. VCF

This situation can be understood as finding a winning strategy by using the four steps. The following is a brief introduction: Our candidate points will only produce points for Open Four and Closed Four, so the opposite may only consider themselves to be five or block us, not considering other positions such as three, Closed Four, and Opne Four. Therefore, there are fewer candidates. If it cannot be blocked, I will win.

C. VCT

This situation continues to threat, winning through the Double Three, the following is a brief introduction: Because our candidate points are three points and four points, we need to consider the opposite of the four points because it will win faster than us. Therefore, the candidate set generated by this situation is larger than the first case. It can be considered that the second case includes more killing patterns, but correspondingly, the speed will be slower. Therefore, we first perform VCF, and if it is found, it will return, otherwise it will perform VCT, which will greatly improve efficiency.

D. Iterative deepening search

Iterative deepening search is a method to quickly find the optimal solution. Its principle is very simple. Firstly, a depth-first search is performed for one layer. If no solution is found, a layer of depth is added until the solution is found. In this way, we can enjoy the low space complexity and time complexity of depth-first search, and we can find the optimal solution. And it seems that we have done a lot of unrelated searches, such as repeatedly traversing many shallow leaf nodes, but in fact these only occupy a small part of the search, and decisive or the last layer of time. So we can ignore the part that does not count duplicates.

E. Pruning for VCX

In the killing, we also need pruning to ensure the speed of the algorithm. First of all, we need to pay attention when running the iterative deepening algorithm. Many times, when we run to a certain depth, we find that there is no new threat behind, so we should terminate the iterative deepening algorithm. We designed a module to detect whether the leaf nodes are still threatening, so we saved a lot of unnecessary search and improved the search efficiency.

F. End the game with killing!

In the game, we observed a pity: our AI occupies a favorable situation, but we have been reluctant to behave. Finally we missed the time and missed the success. Therefore, we will give priority to the VCX process. If we have found a killing strategy before, then we will return to the killing with the wind speed after the next opponent's action, killing the game and not wasting any chance!

V. Current bottlenecks and future improvements

Our current bottleneck mainly has two parts. The first is the computational problem. Due to the speed limitations of Python itself and possible defects in some of our functions, our computing power is relatively weak and we can only search 4 to 6 layers. If we can accelerate we will achieve better results. The second is to limit the ability to kill the module. If we can consider more situations, we can find out the winning strategy in more situations to win. So we want to try to speed up the code and try to threaten the space search and kill it.

References

[1] Jun Hwan Kang,Hang Joon Kim, Effective Monte-Carlo Tree Search Strategies for Gomoku AI, IJCTA, 9(10), 2016, pp. 4833-4841

[2] Junru Wanga, Lan Huangb,Evolving Gomoku Solver by Genetic Algorithm,IEEE WARTIA,2014

[3] JINXING XIE and JIEFANG DONG,Heuristic Genetic Algorithms for General Capacitated Lot-Sizing Problems,2001

[4] Louis Victor Allis,Searching for Solutions in Games and Arti cial Intelligence,Version 8.0 of July 1, 1994