Designing A Gomoku AI Using Alpha-Beta Pruning

Yuqing Zhang
y3593zha@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Abstract

Introduction

Gomoku (also called Five in a Row, Connect 5, or Gobang), is a popular board game played with two people. It is played usually on a 20 * 20 grids Go board, where two players alternatively put their signs (black and while or cross and circle) on until one of the players can achieve a continuous line of five points in either the horizontal, vertical or diagonal direction. Above is the simple rules of Gomoku playing in freestyle (players with the empty opening).

In the freestyle Gomoku, there is no limitation on the opening stage, which can not ensure the fairness of the game. From the mathematical aspect, the player who moves first is supposed to have a winning strategy using specific algorithms in the freestyle Gomoku (Allis and others 1994). However, the artificial intelligence topic related to Gomoku is still challenging in computer science, as the theoretical values of all legal positions have not been solved yet. The intriguing difference between human strategic abilities and the performance of the state-of-the-art AI computer programs lets Gomoku to be an attractive research domain.

Inspired by the remarkable achievement of AlphaGo created by DeepMind (Silver et al. 2016), game AI researchers are dedicated to improving the strategy of algorithms. With simpler rules, Gomoku is popular to be tested for advanced artificial intelligence algorithms. Its rules are easier to implement at the coding level under some traditional game theories, such as the Mini-max algorithm with alpha-beta pruning. Also, many researchers prefer to use neural networks and deep learning techniques to improve the performance of their algorithms in recent years.

In fact, many AI algorithms have some unavoidable flaws. For example, Monte Carlo Tree Search Algorithm (MCTs) requires too many simulations. Searching with shorter steps will lead to a high probability of bad results; the Genetic Algorithm also requires a larger number of iterations. Besides, GA relies on some intractable parameter converting; Threat Space Search Algorithm is relatively slow because it needs to create and challenge the threat constantly.

Since Gomoku has simpler rules than Go, there is actually no need to consider too many future steps as far as Go. Therefore, the majority of the state-of-the-art Gomoku AIs are based on Adversarial search algorithm (especially Mini-max algorithms) and Alpha-Beta pruning. This algorithm assumes that both players will try to maximize their lowest gains to guarantee a rational choice. So it simulates the real situation in a game in a large measure.

The research question our project will address is to program one competitive AI to play the Gomoku game based on the Mini-max algorithm with the alpha-beta pruning. We hope to achieve a higher winning rate compared to some baseline models provided on the Gomoku Cup Website.

We mainly focus on the improvement of the pruning. A novel heuristic based on the Gomoku threat theory (Allis and others 1994) and some wiser moving strategies were implemented to speed up the game tree search and reduce the space complexity and time complexity. We implemented the algorithm and adapted it with a template provided on the GomokuCup website to generate an executable file, which is our agent.

To measure the performance of this algorithm, the best version of our AI was used to run an informal tournament with the existing Gomoku AIs in GomokuCup (Gomocup 2020). We downloaded five AI agents in different rankings to play against as baseline models. Those AIs are based on various algorithms, such as the Monte-Carlo search or alpha-beta pruning with a different way of improvements. After running an offline tournament using our AI, we recorded our victory rate in 20 games versus those AI agents. To ensure fairness, we let both the players move first in 10 games. We chose the AI model with the lowest ranking as our lowest baseline model and we expected to win all the games. Also, we hoped to beat other AIs as many games as possible.

The main contributions of this work are: *(to be continued)*

Related Work

Threat-Space Search in Gomoku

Back to the last century, Dutch scientists Allis and van der Herik proposed the threat-space search where they defined many types of threat (Allis, Herik, and Huntjens 1993). For instance, a straight-four (in Figure 1b) is four continuous signs in a line with both ends empty. The theory stated that a player needs to create either a straight-four or a double threat in order to assure a win of one game. Then the scientists were able to build the dependency tree of the gain square and cost square based on this theory.

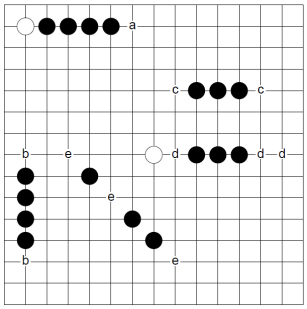

Figure 1: The threat defined in the threat-space search. The pattern of signs which is specially formed is called a threat.

Monte Carlo Tree Search And Adaptive Dynamic Programming in Gomoku

Monte Carlo Tree Search (MCTs) is a classic heuristic search algorithm. The main purpose of MCTs is to make optimal decisions by doing simulations randomly and building search trees(Browne et al. 2012b). There are four steps containing in the MCTs process: Selection, Expansion, Simulation, and Backpropagation as Figure 2 (Browne et al. 2012a). It has been widely used in board games after the extraordinary success in AlphaGo (Silver et al. 2016).

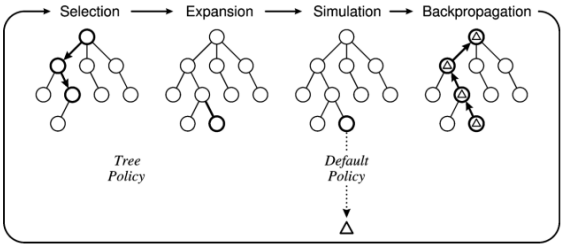

Figure 2: The basic process of MCTS. There are four steps containing in the MCTs process: Selection, Expansion, Simulation, and Backpropagation.

However, using MCTs needs a large amount of time and effort in simulation in order to get a satisfying solution. Additionally, complex and advanced domain knowledge is acquired in MCTs algorithm. Because of these inevitable flaws, traditional MCTs did not perform better than adversarial search with improved Alpha-Beta Pruning algorithm. A remarkable and important improvement for MCTs is adding a tree selection policy, called the Upper Confidence Bounds for Trees (UCT) (Patil, Amrutkar, and Deshmukh 2012). In an attempt to reduce the computation time and get better accuracy, in 2016, scientists introduce a combination method of UCT and heuristic searching, which is called the Progressive Bias. (Kang, Kim, and IJCTA 2016).

Together with MCTs, Adaptive Dynamic Programming (ADP) can be used to design Gomoku AIs as well. ADP needs much less information than the Markov decision process (MDP) as a programming method based on the Temporal difference (TD) learning, a branch of reinforcement learning. Instead, the action decision can be described in the continuous form. The training process of ADP is shown in Figure 3 (Zhao, Zhang, and Dai 2012), where the critic network is a feed-forward three-layer fully connected neural network (Zhao, Zhang, and Dai 2012). Combining MCT with ADP into Gomoku is also a novel approach to solve Gomoku, as the new architecture described in Cao and Lin's work (Cao and Lin 2019).

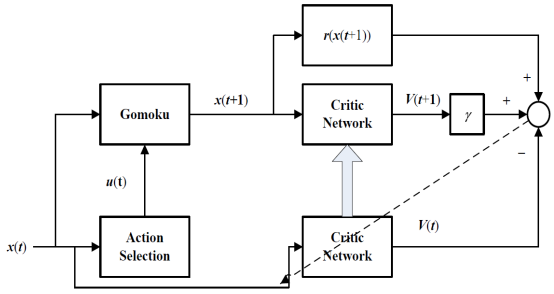

Figure 3: Basic ADP structure

Neural Network in Gomoku

In 1995, the neural network was already presented to be used in playing Gomoku by Freisleben (Freisleben 1995). Reinforcement Learning was used to train the network to evaluate the empty positions on the board with a reward and punishment mechanism.

With the advent of deep learning, some scientists are keen on using the advanced neural network techniques on almost every topic related to computer science. There is no exception for this board game. Experiments have shown that neural networks, such as ResNet, can simplify the Gomoku AI architecture and build a relatively competitive model (Yang et al. ; Li et al. 2019).

Mini-Max Algorithm With Alpha-beta Pruning in Gomoku

Mini-Max is a typical decision rule used in game theory, with the basic idea of maximizing the minimum gain. It is an excellent strategy used in the adversarial search, as it was especially formulated for the two-player zero-sum game. A minimax algorithm assumes that the player always

chooses the move maximizing the minimum value of the positions generating from the opponent's possible next moves. It can be seen as a recursive algorithm (Russell and Norvig 2016).

To speed up the search, alpha-beta pruning is a great heuristic pruning method which dramatically improving the result (Schaeffer 1989). Other details will be included in the Methodology section.

Although Alpha-Beta Pruning algorithm has a long history for almost half a century, with some powerful heuristic functions it still worked better than the other algorithms as showed in the Gomocup website. Based on the above fact, this paper will focus on the improvement of Alpha-Beta algorithm.

Methodology

We use Mini-Max algorithm with pruning to simulate the real scene when playing Gomoku. It is more suitable when being used in the adversarial search than other algorithms, such as the Monte-Carlo search. Another reason for choosing this algorithm is that it simulates the real reaction of the players in a game to a large extent. With the alpha-beta pruning, this competitive algorithm has a much shorter convergence time.

Mini-Max algorithm assumes that both players will try to maximize their lowest gains to guarantee a rational choice. To accelerate the process, Alpha-beta pruning is introduced to abandon unnecessary exploration. Also, we introduce the heuristic function we used to assign value to specific patterns in Gomoku. In this section, we will introduce more details of these parts.

Mini-max Algorithm

In the general game theory, the value of a state can be assigned as the best achievable outcome of that state.

The value has been fixed and described for a terminal state. Through the process of backtracking the game tree from the bottom to the top, we can get the optimal value of the non-terminal states as

$$V(s) = \max_{s' \in \text{children(s)}} V(s')$$

To generalize this to an adversarial game, the zero-sum game is a good representative. As an illustration, Gomoku is a zero-sum game. After the agent puts a sign on the board, the opponent takes another action. For each sequence of alternating action, the game might end and utility is associated. Specifically, at the end of the game, two players have opposite utilities with a sum of zero. Both players are attempting to maximizing their own outcomes, but in fact, maximizing one's own value means minimizing the other's outcome in this pure competition. Therefore, we can think about one outcome instead of two outcomes, where our agent maximizes this single value, and the opponent try to minimize it.

In the game tree, the layer of nodes contains states under the agent's control and the opponent's control alternatively.

For the states under our agent's Control, we can calculate the value as

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

In contrast, for the states under the opponent's control, we can assign the value as

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

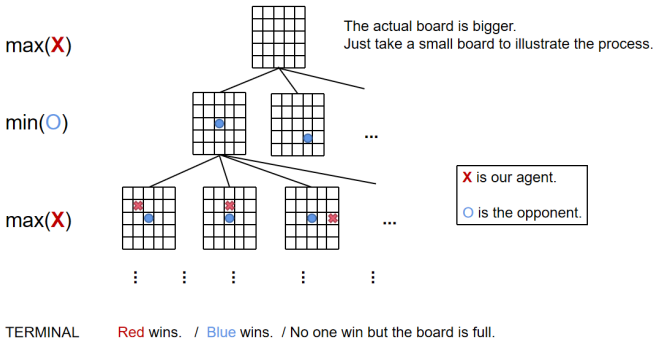

Figure 4: Minimax process in Gomoku

Overall, as Figure 4 shows that the minimax search algorithm is computing the best achievable utility against a rational opponent. The efficiency of Mini-max is like an exhaustive depth-first search, with exponential time complexity. Note that the game has a large branching factor up to 400 (in a 20*20 board), which leads to an enormous game tree. The exact solution is completely infeasible. So it is critical to adopt some techniques such as the depth-limited search or alpha-beta pruning.

Alpha-beta Pruning

There is no need to look at an entire search tree to find a solution. Alpha-beta pruning manages to decrease the node number, stopping evaluating a move which is assumed to be worse than an examined move we have explored before. Pruning these branches will not affect the final decision.

As an illustration, in figure 5 we take the MIN part of the minimax algorithm. We are currently computing the value of node n. When we loop over n's successors, the minimum value of n's successors keeps dropping. If a is the currently best value that MAX can get at present, and if n is considered has a worse outcome than a, we can stop considering n's other successors.

In alpha-beta pruning, alpha represents the best option of MAX on the path to the root, while beta represents the best option of MIN. The pseudo-code for the MiniMax algorithm, including Alpha-Beta pruning in implementing Gomoku is below.

Heuristic Evaluation

For an arbitrary node, it is almost impossible to reach the terminal state and return an optimal result. To limit the search depth, exploiting a heuristic function to evaluate

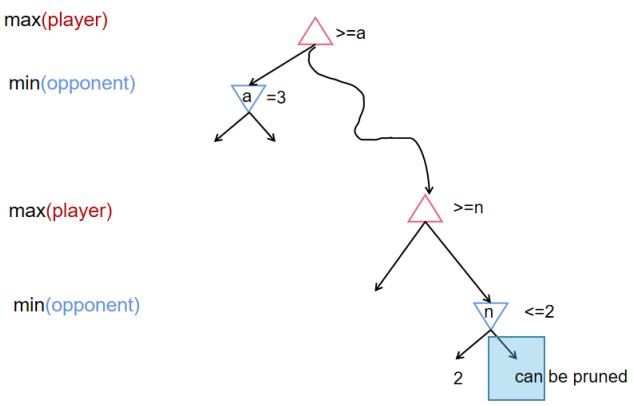

Figure 5: Alpha-beta pruning process in Gomoku

Algorithm MiniMax, changed for Alpha-Beta pruning

```
function INITIALCALL
    getMinValue( board, −∞, ∞)

function GETMINVALUE( board, alpha, beta )
    if board.isTerminal() then return evaluateState( board )
    else
        score = ∞
        for move in board.getLegalMoves() do
            score = getMaxValue( board.createMove(move), alpha, beta )
            beta = Min(beta, score);
            result = Min(score)
            if beta <= alpha then
                break
        return result

function GETMAXVALUE( board, alpha, beta)
    if board.isTerminal() then return evaluateState( board )
    else
        result = −∞
        for move in board.getLegalMoves() do
            score = getMinValue( board.createMove(move), alpha, beta)
            alpha = Max(alpha, score);
            result = Max(score)
            if beta <= alpha then
                break
        return result
```

Figure 6: pseudo code for Minimax process in Gomoku (Russell and Norvig 2002)

which moves are more relevant to our agent at the current state is necessary.

Our heuristic will first gather moves that can induce some predefined patterns together with some existing points. Based on the Gomoku mathematical threat theory introduced in 1993 (Allis, Herik, and Huntjens 1993), our heuristic function assign different scores for different patterns (as shown in table).

Pattern Name	Explanation	Examples	Score
(1,1)	1 piece, 1 live end	xo_	1
(1,2)	1 piece, 2 live ends	_o_	10
(2,1)	2 pieces, 1 live end	xoo_	10
(2,2)	2 pieces, 2 live ends	_oo_	100
(3,1,"S")	3 pieces, 1 live end, split	xo_oo_	100
(3,2,"S")	3 pieces, 2 live ends, split	_o_oo_	1000
(3,1)	3 pieces, 1 live end	xooo_	100
(3,2)	3 pieces, 2 live ends	_ooo_	1000
(4,1)	4 pieces, 1 live end	xoooo_	1200
(4,2)	4 pieces, 2 live ends	_oooo_	100000
(4,0,"S")	4 pieces, 0 live end, split	xoo_oox	100000
(4,1,"S")	4 pieces, 1 live end, split	xoo_oo_	1100
(4,2,"S")	4 pieces, 1 live end, split	_oo_oo	100000

Table 1: Heuristic evaluation for each pattern

The state evaluation function above can evaluate how likely we will win given the current condition. Besides, we also need a method to evaluate how likely to win if we make a move at a certain point (x, y) on a board. After one move is executed, we can add scores of every pattern of two players respectively, and update the current score. Thus, we exploit a point evaluation method:

$$V(x,y) = \sum V(\text{Pattern}_i),$$

where Pattern_i is the pattern made by our move at (x, y), $V(\text{Pattern}_i)$ is the value of the pattern according to our simple version pattern score of Table 1). V(x,y) indicates how smart a move is. It will be of great use in the pruning ordering.

Since we are using the Minimax algorithm and evaluate only one output value, the final value will be assigned as

$$V = V(\text{agent}) - \gamma V(\text{opponent}),$$

where γ should be larger than 1 to emphasize the adverse effect when the opponent gets a bigger chance to win.

Results

Details of Implementing the Algorithm

The order of exploring nodes can dramatically affect the searching efficiency. We use a heuristic to decide the order. After we generate a list of candidate nodes to explore,

we sort them by the estimated value mentioned in the section above. And each time we only choose the top scores moves and add them to the tree.

We believe that a node with a higher value is more likely to be the optimal move, and then lead to the pruning of other unnecessary nodes. However, this method may be risky because this greedy approach abandons some valuable nodes which have lower values but can contribute more in the future steps. But overall this method accelerates the searching process a lot so that we have chances to search deeper. So what we do is to seek a trade-off between the search breadth b and the search depth d. We choose several parameter pairs (d,b) to implement the algorithm and find the best one as the final default setting.

In Gomoku, we also need to find the trade-off between attacking and defending. Here we introduce the killing strategy. When we find that we are about to win (like _oooo_), we can react more aggressively. Also, when we find that we have no alternative but to block the threat of the opponent, otherwise we will lose, we also need to be decisive to defend.

In the killing process, we only take those positions with specific patterns into account. For example, in the MAX-node, we search nodes to attack(_ooox _, _oooo _), and in MIN-node, we search for those where the opponent can defend our attack or they can make a more severe attack. As the nodes satisfying the conditions are much fewer, we can easily search deeper with negligible time consumption. Once we find a "killing-move", we can achieve success directly. Therefore, we can summarize the process as the figure 7 7 followed.

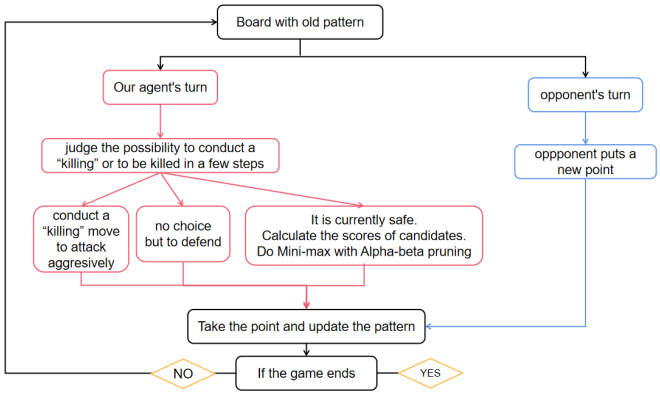

Figure 7: searching process in Gomoku.

As we assign the final value to each node

$$V = V(\text{agent}) - \gamma V(\text{opponent}),$$

where γ is a parameter larger than 1. We also need to take experiments on it to make sure the most appropriate value.

Evaluation Measurement

After implementing the algorithm, we used the template provided on the GomokuCup website to generate an executable file, which is our agent. Then we played our agents versus other agents provided by the platform using the downloaded interface. We played against five different opponents (named MUSHROOM, Valkyrie, PureRocker, Pisq7 and Noesis with a ranking from low to high) each for 20 times. The MUSHROOM is our baseline model in the experiments. Among these trials, our agent took the first move 10 times, and the opponent takes the first move in the other 10 times.

Codes Design

The project code is posted on the github [1]. The UML diagram in Figure 8 and table in Table 2 show the design of my codes. The main structure contains two files: one is the algorithm file to implement the MiniMax searching algorithm, while the other file has various utility classes and functions to choose the candidates, compute the score, define the pattern, conduct the Kill movement, etc.

file	class	explanation of the functions
miniMax	MiniMax	the main AI algorithm - Minimax with alpha-beta pruning
	TTEntry	create the transition table
utils	Board	obtain the candidates points
	Kill	conduct a kill movement
	Pattern	define the heuristic(assign value to each pattern)
	Score	calculate the score of both players

Table 2: information about the code design

Technical Challenges

There are several technical challenges during the implementation:

First, the transition from the algorithm in theory to the game in practice is quite difficult. A bunch of problems needs to be addressed, such as how to move a point and how to apply the algorithm into the real game. We start merely from the pseudo-code in theory, so the process of making it to a complete project is demanding.

Additionally, the performance is relatively hard to quantify, thus making the implementation far more challenging. Unlike other machine learning techniques (such as in some classification problems), you can get an accuracy on the training set or the test set. In our project, the competitiveness of the model is mainly reflected in the performance (win or loss) in the real games. A higher winning rate is the most important.

Another challenge is to find a balance when designing the algorithm.

The balance includes the trade-off between defense and attack, whether to attack more aggressively or to move more defensively and wait for the opponent to make a mistake. This mainly depends on the value assigned to each pattern.

The second trade-off is to balance the running efficiency and the quality of the result. If we explore more nodes we can choose the best candidates with a higher possibility.

[1] github website: https://github.com/aaazyq/Gomoku

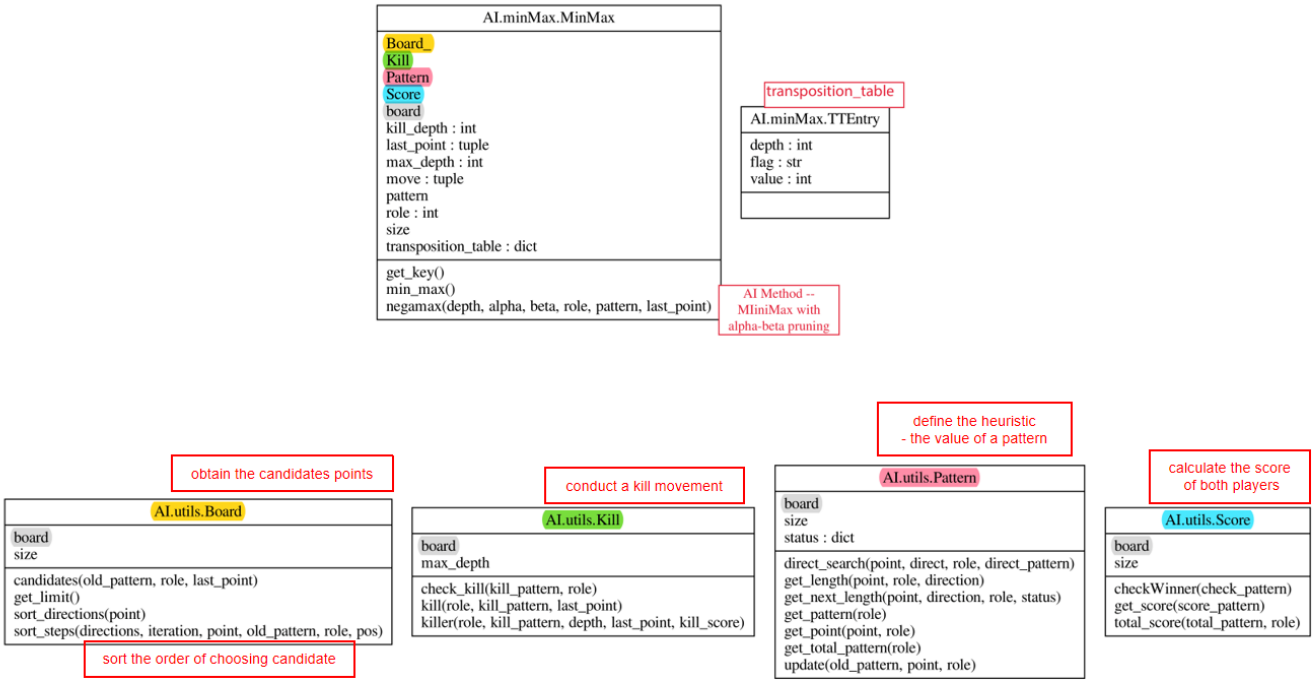

Figure 8: UML Class Diagram. The classes, functions or attributes in the same background color show the close relationship between each other.

However, it needs more computation and the convergence speed is slower, which may exceed the time limit of each turn.

Performance

To optimize the performance of the code, I made several efforts. Besides the new heuristic function we came up with above, the main point is the use of Negamax and the Transposition Table to accelerate the computation speed. Negamax is a variant form of MiniMax search, which relies on the property that $\max(a, b) = -\min(-a, -b)$ and the value of the first player is the negation of the value of the second player. It uses only one function instead of two functions in MiniMax thus largely simplifying the implementation.

Transposition Table is a novel way to store the states. During the searching process, we may encounter many identical states. It's a waste to compute them repeatedly. To avoid this situation, we can store the state and value in a dictionary. However, our representation of the state is a 20*20 matrix, it can hardly be possible to use as the key to the dictionary. So we represent a board state by a randomly generating 64-bit integer and maintain a dictionary (a transition table) of board value. The method is also called the Zobrist Hashing algorithm. In this way, the algorithm could get deeper into the tree in a shorter time, resulting in a better move being eventually chosen.

In our experiment, simple Minimax will cost 9.24 seconds in each turn (calculated by the average time when our AI agent plays against itself for 50 turns). Using the Negamax search and Transposition Table can decrease by about 8.5% of the computing time (8.43 seconds per turn).

To choose the most appropriate hyper-parameter (max-depth), I compare the running time of models with max depth = 1,2,3,4 when they compete with themselves. The box-plot in Figure 9 intuitively proves that as the searching depth grows, the running time per turn also increases. If we confine the running time per turn to 30 seconds, there is a possibility where `max_depth` = 3 or 4 can not return the result. Also, the time difference between `max_depth` = 1 and 2 is relatively small.

The line chart in Figure 9 shows that after how many turns the game ends when the agents play against themselves. Interestingly, when `max_depth` = 2, the moves in the game are the most. This is because as the searching depth increases, the agent is smarter, and easier to catch the opponent's mistake so the game ends earlier. Meanwhile, when `max_depth` = 1, the agent is so 'stupid' that it may make simple mistakes and after 30 steps the game ends.

So based on the result above, I choose `max_depth` = 2 in the model. This figure seems quite small, but with the proper heuristic function and Kill Movement, the model is competitive enough.

In the informal tournament, We recorded the winning rate against other AIs. The competition result in Table 3 shows the competition result versus five other agents. It is understandable that when our AI moves first, we have a larger opportunity to win than when we move later.

What reaches our original expectation for this project is that the model is much better than MUSHROOM (the base-

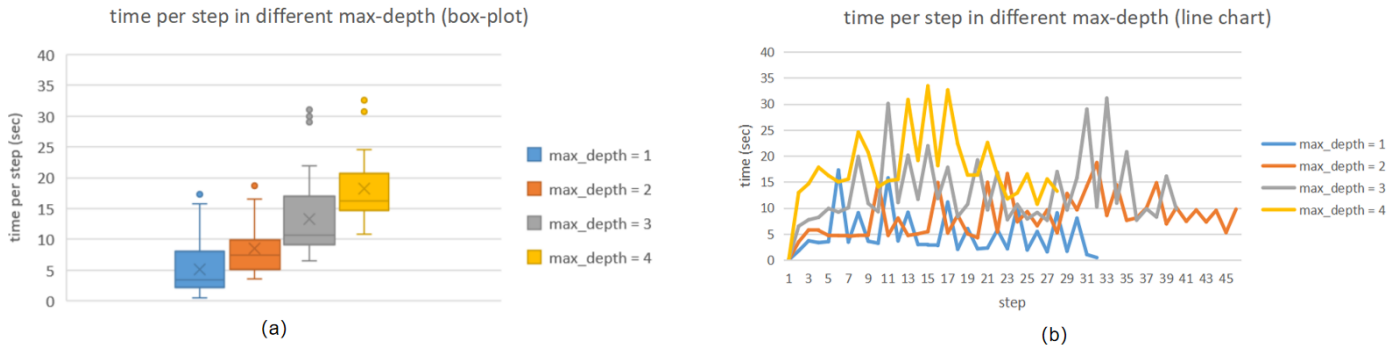

Figure 9: running time per turn in different max-depth(1,2,3,4). Left is the box-plot showing the time distribution. Right is the line chart showing time per step in details and the information of when the game ends, for example, model with max_depth = 2 ends after 46 moves.

AI Name	our wins	our losses
MUSHROOM	20(10+10)	0(0+0)
Valkyrie	18(10+8)	2(0+2)
PureRockery	16(8+8)	4(2+2)
Pisq7	11(7+4)	9(3+6)
Noesis	2(2+0)	18(8+10)

Table 3: competition result versus other AIs. The AI listed is downloaded in the gomocup website. The format in the "win" and "losses" is $w(w1+w2)$ and $l(l1+l2)$. The number of w or l shows the number our agent won or lost in the 20 games, and the number of $w1$ or $l1$ shows the game number won or lost when our agent moved first, while $w2$ or $l2$ shows the number of our wins or losses when the opponent moved first.

line model). Besides, our AI has a complete advantage over Valkyrie and PureRockery and had a nearly-even race versus Pisq7. It's a pity that we still have some deficiencies against the strong opponents like Noesis.

Lessons I learned

Honestly, this is the first time for me to implement a CS project individually. From this project, I learned how to apply the algorithm to a real situation. I knew the whole implementation process of MiniMax search and I made some progress on various aspects of coding. Also, I knew I should be more organized and avoid getting into a rush for quick results in future projects.

More experiments in practice are crucial. As was previously stated, it is often hard to find the most appropriate value for each pattern. Sometimes only in the experiments, we can find where the problem exactly is. Also, it is necessary to analyze and summarize the reasons for the loss. For example, in the replay of one trial I found that when there is already a three-in-a-row, our AI is likely to choose to skip one grid to put the sign (xxx_x), rather than form a four-in-a-row (xxxx) (seen in Figure). During the analysis, I found that I assign the two patterns (xxxx) and (xxx_x) the same value. After giving the (xxxx) pattern a higher value, we make progress on the result.

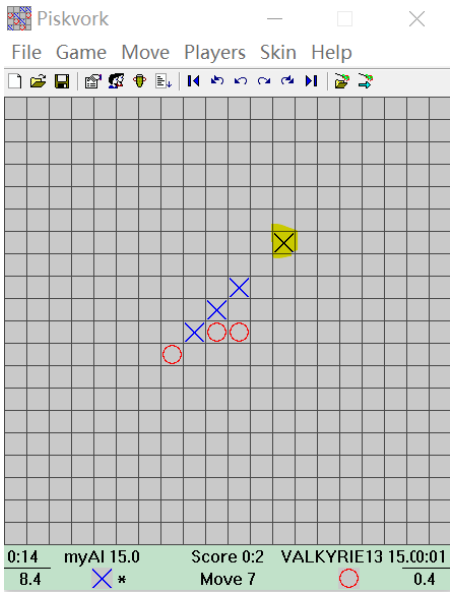

Figure 10: One mistake occured in the experiments. The blue X is our AI and the red O is the oponent AI. The point in the yellow background is our last move, which is not the best in this situation.

Furthermore, I learned that 'flexibility' is another key to winning. For example, the strategy and emphasis when you face a week opponent and a powerful opponent should be different. The ability to adjust yourself according to the circumstances is an underlying factor to winning.

References

[Allis and others 1994] Allis, L. V., et al. 1994. Searching for solutions in games and artificial intelligence. Ponsen & Looijen Wageningen.

[Allis, Herik, and Huntjens 1993] Allis, L. V.; Herik, H. J.; and Huntjens, M. 1993. Go-moku and threat-space search. University of Limburg, Department of Computer Science.

[Browne et al. 2012a] Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012a. A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games 4(1):1–43.

[Browne et al. 2012b] Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012b. A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games 4(1):1–43.

[Cao and Lin 2019] Cao, X., and Lin, Y. 2019. Uct-adp progressive bias algorithm for solving gomoku. In 2019 IEEE Symposium Series on Computational Intelligence (SSCI), 50–56. IEEE.

[Freisleben 1995] Freisleben, B. 1995. A neural network that learns to play five-in-a-row. In Proceedings 1995 Second New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems, 87–90.

[Gomocup 2020] Gomocup. 2020. Gomocup, the gomoku AI tournament. https://gomocup.org/.

[Kang, Kim, and IJCTA 2016] Kang, J.; Kim, H. J.; and IJCTA. 2016. Effective monte-carlo tree search strategies for gomoku ai *.

[Li et al. 2019] Li, X.; He, S.; Wu, L.; Chen, D.; and Zhao, Y. 2019. A game model for gomoku based on deep learning and monte carlo tree search. In Chinese Intelligent Automation Conference, 88–97. Springer.

[Patil, Amrutkar, and Deshmukh 2012] Patil, T.; Amrutkar, K.; and Deshmukh, P. K. 2012. Monte carlo tree search method for ai games. The International Journal of Emerging Trends Technology in Computer Science 2.

[Russell and Norvig 2002] Russell, S., and Norvig, P. 2002. Artificial intelligence: a modern approach.

[Russell and Norvig 2016] Russell, S. J., and Norvig, P. 2016. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,.

[Schaeffer 1989] Schaeffer, J. 1989. The history heuristic and alpha-beta search enhancements in practice. IEEE transactions on pattern analysis and machine intelligence 11(11):1203–1212.

[Silver et al. 2016] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T. P.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. Nat. 529(7587):484–489.

[Yang et al.] Yang, G.; Chen, N.; Niu, R.; Patel, V.; Ni, Y.; Jiang, H.; and Weinberger, K. Learning the game of renju with neural network and tree search.

[Zhao, Zhang, and Dai 2012] Zhao, D.; Zhang, Z.; and Dai, Y. 2012. Self-teaching adaptive dynamic programming for gomoku. Neurocomputing 78(1):23–29.