Tackle the Game of Gomoku by Minimax with Multi-layer Heuristics

Yang Su, Yu Li, Yumeng Yao
{y242su, y2455li, y89yao}@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Abstract

Gomoku, also called five-in-a-row, is a popular abstract strategy board game developed by humans in the 19th century. Classical methods to tackle the game by purely tree-based Minimax algorithm does not achieve good results because of its limitation in both search depth and width. In this paper, we try to assign heuristics from the surroundings of each possibly winnable grid and build a multi-layer threat space searching algorithm on top of that. Earlier layers perform mainly for restriction and enforcement in sensing threat and later ones utilize trainable ranks for pruning the search space. Experiment results show that our approach outperforms traditional Minimax from closest neighborhood searching and generates interesting outcomes against amateur human players. The algorithm parameters are constructed in a neural-network like and self-correlated way, and it can be easily adopted by reinforcement learning methods like Monte Carlo Tree Search. This paper serves as a new approach of including multi-layer heuristics in RL for board game AI.

Introduction

Gomoku is listed as one of the competition events in Chinese National Intelligence Games, so the experience of growing up in China has given us a great number of opportunities to not only get familiar with, but also try to master this game. The rules of Gomoku is that both sides of the competitors, with black side makes the first move, take turns to move the stones across the empty intersections on the board until one side wins by forming a sequence of unbreakable five stones horizontally, vertically and diagonally in sequence.

The most famous success of board game AI would probably be the Deep Blue by IBM and AlphaGo by Google which mastered Chess and Go respectively. The infamous success, however, is that their AI utilized the underlying technology of playing games in Internet of Things (IoT), big data, cloud computing and much more. Their AI labs have then participated and worked with UK National Grid and the field of health and insurance industry, aiming to use AI's infrastructure as a way of improving the efficiency of the British power grid and automating a quarter of insurance jobs in the near future (Yu 2016). Game AI has paved the way for innovation in many fields, that is why it is so important and we as researchers want to explore the methodology within board game AI.

Inspired by AlphaGo, which combines Deep Neural Networks and Reinforcement Learning, we would like to develop an algorithm that can play against human in a non-linear way, which is similar to AlphaGo, and try to improve the efficiency and the winning rate of the algorithm as much as possible. In particular, we choose Gomoku because of its long-lasting history in China and its simple rule but complex variations. The difference between human strategy and the performance of AI algorithm would be an interesting research topic and makes applying AI on Gomoku worth exploring. In 1994, Allis has come up with an algorithm called treat-space search which mathematically proves that Gomoku is a solved game. However, until today, no actual AI engine has kept a winning rate of 100 percent playing as black (the first-mover) in standard Gomoku. Like Google always keeps exploring how to make AlphaGo perfect, we want to create a more strategic and offensive algorithm on Gomoku.

In the previous studies of Gomoku, using Minimax search is a common strategy. However, this approach requires a long time to generate results and the algorithm cannot 'learn' anything during the game. In this research, we are interested in designing a new Gomoku solver based on an extension of Minimax search by adding $\alpha - \beta$ pruning and utilizing Victory of Continuous Four (VCF) and Victory of Continuous Three (VCT) heuristics to achieve higher effectiveness. We will also focus on its advantages compared with the traditional game-tree-based solvers. Since Gomoku has been proved to be sure-win for the first player (Allis, 1996), in our problem, we will use the Swap2 rules in worldwide Gomoku competition to ensure fairness on both players.

The method primarily consists of Minimax search, $\alpha - \beta$ pruning and Genetic algorithm. Minimax search is a recursive algorithm for choosing the next move in

an n-player game by maximizing the minimum gain. And $\alpha - \beta$ pruning is a search algorithm that helps to reduce the number of nodes in the search tree. Genetic algorithm is a search heuristic that reflects the process of natural selection by selecting the fittest individuals. We will use VCF and VCT to determine how fit each situation is.

We test the traditional game-tree-based algorithm with our modified Minimax search algorithm by

1. Let these two algorithms fight themselves 1000 games to check the algorithms' performances both as an attacker and as a defender, and also compare their efficiency.
2. Let the two algorithms fight with each other 2000 times (for 1000 times, let Algorithm A be the attacker, for the other 1000 times, let Algorithm B as the attacker) and compare their winning rate and efficiency.
3. Compare the amount of calculation each algorithm required with regards to the number of pieces on board.

The algorithm performs well in general since AI 0.0 can beat the amateur human players in most cases with a winning rate of approximately 70%, even though AI 0.0 demonstrates the lowest intelligence level compared to AI 1.0 and AI 2.0. The result also shows that AI 2.0 has a higher winning rate because we introduce another layer "rank" and the algorithm will not choose the candidate moves with worse ranks than the move with the top rank, which increases the winning rate and reduces the running time. In terms of running time performance, three versions all take within 0.5 seconds for each move on average, which is good enough for an AI agent to respond to its opponent.

The contributions of our algorithm for the AI agent are described as follows:

• Beats the human players in most cases

• Improves the classic Minimax algorithm with alpha-beta pruning in terms of the winning rate and the running time by introducing multi-layer heuristics

• Combines the concept of neural network with heuristic search and introduces parameters "depth" and "width" to allow the AI agent to anticipate multiple future moves

• Discovers the limitations of this approach and discusses potential fixes

Related Work

1. Adversarial Search

Adversarial search is a search method where there exists an opponent changing the State of the problem every step in a direction you cannot control. Usually the search refers to a confrontation problem with two sides, each in a different party. Any party hopes that the situation will try to be beneficial to themselves, which should also be as unfavorable as possible for the other side. Minimax is a popular algorithm to solve this type of problem.

Minimax is widely used in strategic board games such as chess, Go and Gomoku where two players compete with each other. The goal of Minimax is to enumerate all State spaces in an exhaustive manner to predict the result at the current State. To achieve this, Minimax recursively evaluates the worst possible situation from our viewpoint by taking into account the best possible move of the opponents, and then it tries to maximize our outcome given this worst case scenario. We will assign each State a score by establishing heuristic functions for Minimax to determine the favorability of each move. In actual game-play of Gomoku, however, the search space of the game is extremely large, and it can be easily verified that Minimax is not able to exhaust the search space (Kuan Liang Tan et al. 2009). In fact, on a normal machine, Minimax can only successfully predict the solution that is 3 turns ahead. As a result, we usually perform Minimax in the neighborhood, or search to a fixed depth.

To get rid of unnecessary branches, we use $\alpha - \beta$ pruning where α and β keep track of the currently discovered highest score of ourselves and the lowest score of the opponent. If $\beta \leq \alpha$, the branch is discarded as there is already another branch outperforms it. This pruning approach improves efficiency of Minimax on Gomoku significantly according to (Liao 2019).

2. Heuristic Tree Search by Reinforcement Learning

Another algorithm comes into play in recent years is the heuristic tree search algorithm, one of the most successful ones is called Monte Carlo Tree Search (MCT). It is firstly widely known at the end of 20th century when "Deep Blue" surpass world known human experts in the Game of Chess; and later in 2016, "Alpha Go" defeat the best human player in Go utilizing MCT with neural networks.

Instead of establishing search trees using fixed heuristics, MCT pursues optimal decisions by random sampling and doing self-play simulation in the adapted game domain and builds a search tree according to the result of simulation. Compared to Minimax, MCT does not store intermediate State result to generate search trees, so it greatly decreases space complexity; meanwhile, due to the fact that MCT highly depends on the combination of the generality of random sampling and search trees, an improper sampling mechanism or search depth would lead to bad performance of the algorithm and cause huge time waste.

Due to the limitation of MCT in the sense of randomness and finite depth, it has been considered not suitable for complicated board games for a long time. It was not until 1997 that Deep Blue made remarkable progress in Chess, using the MCT algorithm developed by Philipp Rohlfshagen and Colton. Later on the MCT framework of Gomoku was also proposed and it began to give people the hope that MCT can be effectively used in board games (Chaslot et al. 2008). After that, the Game of Go was considered the last human-made board game challenge of Artificial Intelligence.

Different from using purely MCT, AlphaGo uses neural network models to perform the MCT simulation process. It

trains a 13-layer so called policy network with 30 million positions from previously human matches and a new reinforcement learning framework instead of traditional MCT selection method. With AlphaGo as an inspiration, in the same year, a Gomoku AI is developed using MCT with neural network based on Adaptive Dynamic Programming (ADP) (Tang et al. 2016). Instead of AlphaGo, this AI extracts the neural network outside of MCT. In particular, it uses ADP on neural network to generate 5 candidate moves given some board configurations, then by setting these candidates as the root State, MCT selects the best of them by assigning weights and winning probabilities according to its inline structure.

3. VCF, VCT and Genetic Approach

What we will possibly try is the most successful heuristics used in Gomocup (worldwide tournament of AI playing Gomoku) based on $\alpha - \beta$ pruning, the Victory of Continuous Four (VCF) and Victory of Continuous Three (VCT), which refers to continuously manufacturing a blocked four attack or open-ended three attack like the pattern below to force the opponent to have less and less opportunity to fight back until no more VCF or VCT sequence of moves can be generated. By utilizing Minimax, this can also predict the best possible move of the opponent and fight back accordingly.

On top of this heuristic that tends to be deterministically aggressive to the winning goal, we would like to give AI some variation and learn based on its opponent and itself. Mimicking the Gomoku AI with ADP, we can use VCF, VCT to generate several possible candidate moves as the root State, and performs a genetic algorithm to further operating crossover and mutation to choose the "best" candidate by a continuously growing fitness function. The result would also potentially modify the given heuristics value of each pattern configuration at the beginning and accelerates the process of Minimax, as suggested in the research (Junru Wang and Lan Huang 2014).

We would also try to do further optimization around threat space search reduction, board heuristic function estimation on offensive piece factor. See details in methodology (D2).

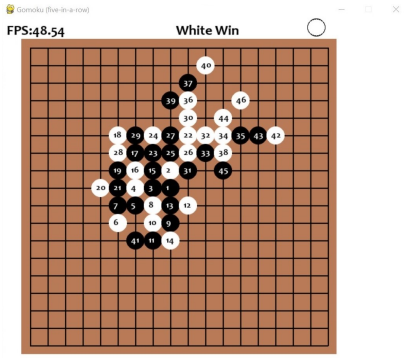
Methodology

The details of the design and implementation of the Gomoku game AI will be discussed in this section. Both the game engine and the game AI were implemented using Python.

Game Rules

Gomoku is a game played with black and white stones on a 18×18 board. Two players alternatively place a stone of their color on an empty square. Black plays first. The first player who forms a continuous chain of five stones of the same color on the same row, column, or diagonal will win.

The Gomoku game engine enables human player vs. human player, human player vs. AI, and AI vs. AI. During each turn, human players can select the desired move by clicking the board. The game engine also enables users to set the difficulty by changing the depth level - the number of steps AI thinks ahead, and width level - the range of boards AI thinks of. The final result (who wins the game) will be shown on top of the board.



Game AI

The empty squares on the game board can be viewed as possible moves for one player, and where to put the next piece can be viewed as a "search-for-optimal-solution" problem, which is the main reason we use minimax algorithm and alpha-beta pruning. For each move, a heuristic value is assigned to each of the empty squares. Then, the AI agent considers possible moves of opponents by recursively maximizing the minimum gain to choose the best move for itself. Since the exponential increasing of the number of search nodes, alpha-beta pruning is introduced to decrease the number of nodes that are evaluated by the minimax algorithm, which guarantees the search space will not be exhausted and the solution is found efficiently and optimally.

1. Heuristic

The heuristic function evaluates the gain of each move for the current AI player. For each move, the AI agent will not only consider itself but also consider how this move will affect the opponent. The algorithm classifies each move as offensive move and defensive move. For offensive moves, the AI agent tries to form an unbroken chain of its color. For defensive moves, the AI agent tries to block the opponent from forming an unbroken chain. The algorithm also classifies the move as move with "open end" and move with "dead end". Move with an "open end" refers to moves that have a side next to an empty square. Move with a "dead end" refers to moves that have a side next to a stone of the opposite color or wall. There are a few rules to follow:

1. A non-empty square will have the lowest value.
2. A move that will make current player win in the next move will have the highest value.
3. A move that will block the opponent from winning in the next move will have the second highest value.

4. With the same number of a continuous chain of stones in the next move, a move with an "open end" will have a higher value than a move that has one or more "dead end".
5. With the same number of a continuous chain of stones in the next move, an offensive move has a higher value than a defensive move. (consistent with the 2nd and the 3rd rule)

Note that with more than one move with the same heuristic values, the AI agent will randomly choose from those moves. The value of each move is arbitrarily chosen and summarized in 1 and 2.

Offensive Move	Value	Defensive Move	Value
occupied square	-10	occupied square	-10
5 in a row	100000	5 in a row	90000
4 in a row	10000	4 in a row	5000
3 in a row	350	3 in a row	250
2 in a row	50	2 in a row	10
1 in a row	1	1 in a row	0

Table 1: Values of Different Moves With an "Open End"

Offensive Move	Value	Defensive Move	Value
occupied square	-10	occupied square	-10
5 in a row	100000	5 in a row	90000
two blocked ends	0	two blocked ends	0
4 in a row	500	4 in a row	400
3 in a row	30	3 in a row	7
2 in a row	1	2 in a row	0
1 in a row	1	1 in a row	0

Table 2: Values of Different Moves With One or More "Dead End"

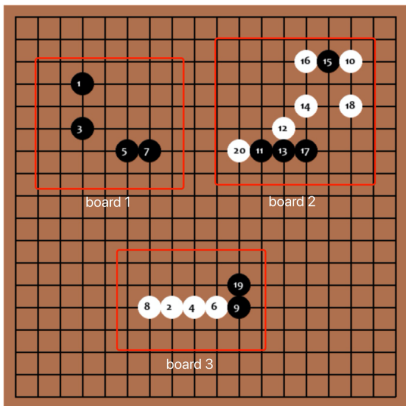
2. Minimax Algorithm with Alpha-beta Pruning

The AI agent uses minimax algorithm to decide the next move. The minimax algorithm assumes that two players take alternate moves. For the current player, the goal is to maximize its gain. However, for the opponent who moves next, the goal is to minimize the player's gain. The AI agent always starts at depth 0, which indicates the current player's turn. At this level, the algorithm goes through all possible neighbors based on width (range of boards AI thinks of) to construct level 1. At level 1, it's the opponent's turn, so AI will go through all the opponent's neighbors and construct level 2. This process continues recursively until the algorithm reached the depth limit. The heuristic value is only assigned to leaf nodes when the algorithm reaches the limit, it backtracks, and at the opponent's level, it finds the minimum value, at the current player's level, it finds the maximum value.

To eliminate the branches of minimax search tree, alpha-beta algorithm is introduced. Alpha represents the best (max) choice AI has found so far, and beta represents the best (min) choice AI has found so far. To initialize, set alpha to -1000000, which is equivalent to negative infinite considering the heuristic values, and beta to 1000000, equivalent to positive infinite as the worst possible cases. At the opponent's level, multiply the heuristic value by 0.9, consistent with the values of defensive moves. Alpha is only updated when the current move is better (greater value) than the last time. Similarly, beta is only updated under the same condition, but when the current move is better (lower value) than last time.

Different from many other minimax algorithms for Gomoku, this algorithm evaluates heuristic values of points instead of the whole board. It is "greedy" since it weighs more on attacking than defending. In other words, facing a similar situation (e.g., both the player and the opponent has "3 in a row with one open end"), this algorithm will prefer to develop itself (e.g., create "4 in a row with one open end" for the player) to block the opponent. Based on this difference, we introduce a variable $rank$ additionally to classify different heuristic values into three ranks.

$rank$ can be viewed as a forced-attacking move, whereas the heuristic function mentioned in Tables 1 and 2 can be seen as a voluntarily-defending move. In function fetch_neighborhood of the pseudocode on the right side, it will process and calculate the possible rank points around the neighborhood in a loop, if there is no rank point found and no new rank point to be assigned, the function will update the corresponding heuristic values instead. Therefore, in the alpha-beta pruning process, the algorithm first looks for the intersection with the highest rank, if found then the rest of the board is ignored, which will improve the program efficiency to some degree. Note that rank 1 and rank 2 are corresponding to the highest (100000) and the second-highest(90000) heuristic value. However, rank 3 and 4 are assigned by looping through the neighborhood and determining the rank value based on the heuristic values and the previous rank values. The rank will be utilized in Minimax with additional pruning parameters rank_max, for the algorithm to discard any position choices with a lower rank.



Rank examples drawn on the same board

Rank 1 and rank 2 are defined as deterministic situations

Algorithm 1 Minimax with Alpha-beta Pruning

Input search_set, pos_list, conf_list, eval_list, depth, width, max_depth
Output max(pos_list[depth])

```
1:  procedure MINIMAX
2:      if depth = 0 ∨ board_is_full then return
3:      for child_pos in search_set do
4:          if depth % 2 = 0 then
5:              simulate_board(child_pos, my_move)
6:          else
7:              simulate_board(child_pos, opponent_move)
8:          next_search_set = []
9:          max_pos = None
10:         max_conf = alpha_conf
11:         max_eval = global_dict(alpha_conf)
12:         for winnable_pos in fetch_neighbor(width) do
13:             curEval = evaluate(winnable_pos)
14:             if curEval < max_eval then
15:                 max_pos = winnable_pos
16:                 max_conf = global_dict(winnable_pos)
17:                 max_eval = curEval
18:                 next_search_set.add(winnable_pos)
19:         if depth % 2 = 0 then
20:             if max_eval > max(eval_list[depth]) then
21:                 clear(pos_list, eval_list, conf_list, depth)
22:                 for prev_eval in eval_list do
23:                     if prev_eval == max_eval then
24:                         conf_list[depth] = max_conf
25:                         eval_list[depth] = max_eval
26:                     maxEval = max(maxEval, eval)
27:         else
28:             if max_eval < max(eval_list[depth]) then
29:                 clear(pos_list, eval_list, conf_list, depth)
30:                 for prev_eval in eval_list do
31:                     if prev_eval == max_eval then
32:                         conf_list[depth] = max_conf
33:                         eval_list[depth] = max_eval
34:                     maxEval = max(maxEval, eval)
35:         if len(next_search_set) == 0 then
36:             pos_list.remove(child_pos)
37:         else
38:             Minimax(next_search_set, depth - 1)
39:         if depth % 2 = 0 then
40:             restore_board(child_pos, my_move)
41:         else
42:             restore_board(child_pos, oppo_move)
```

which means fetch_neighborhood would just return one value in search_list. The piece arrangement in board 3 of the figure above represents rank 1 and rank 2 in general, where if it is the white player's turn, white wins by placing the piece on the intersection next to piece 8, and rank 1 is assigned to this particular intersection. On the other hand, if it is the black player's turn, the same piece will be assigned rank 2 for this case where the opponent already has 4 pieces in a row with one side blocked and the current player needs to block the opponent from winning immediately. In board 1, the situation is a bit more complicated since there are empty intersections between black pieces. The algorithm loops through each neighborhood by looking at if rank 1 and rank 2 intersections are found, if not found (which is the case here) then we continue to see if the heuristic values of the possible next move positions satisfy the condition of rank 3. If so, then we update the rank value of the certain intersection on the board. Rank 3 refers to the situations where there are three pieces in a row with both ends to be "open end"s (OXXXO where X represents pieces that are already set and O represents intersections that have not been occupied), and where there is one empty intersection between the three pieces (e.g. OXXOXO). If one intersection has been assigned rank 3 in the loop which we use to determine the rank value twice, then the rank of this piece is upgraded to rank 2, since this is a move to form 2 unbreakable chains in total which increases the winning possibility a lot. Similarly for rank 4, the algorithm will increase the rank value by 1 to rank 3 if the same piece is found to be rank 4 twice in the loop. However, there are 3 situations that we assign rank 4. The first one is two pieces in a row with both ends to be open (e.g. OXXO, OXOXO), which is indicated in board 1 of the figure above. In this case, the square on the intersection of the vertical and horizontal chains of pieces is assigned rank 4 twice, which will be upgraded to rank 3 finally. The other two situations are two pieces with empty intersections in between (e.g. OXOXO) and three pieces in a row with one open end and one blocked end. Therefore, defining four rank values in total can help identify which move is the best move more clearly and improve the efficiency in the pruning process.

Results

Since this is a game with only three results-winning, losing and tie, the experimental design is to compare the winning rate and the efficiency with different players and algorithms.

Measures and Metrics

We provide depth level and width level as parameters when the game starts, which can influence the performance, including running time and the winning rate of the algorithm. The first measure is to play against an AI agent as a human player to see if the AI agent is smart enough to block the offensive moves from the human player and make the offensive moves to win the game. If the AI agent is making some random moves on the board instead of placing its piece to form an unbreakable chain, the algorithm likely has a very low power of prediction. After the AI agent can beat amateur human players several times, we make 2 players both AI agents as opponents to play against each other for 100 rounds and document the number of times each player wins or tie. In the process of adjusting heuristic values and the pruning process, we also measure if the adjustment improves the algorithm or not by making the AI agent play against the previous algorithm to see the winning rate. By making two algorithms play against each other, it is easier and straightforward to tell which one is better than the other one and to determine which heuristic values and rank values are better when depth and width are set.

The maximum time that each move that it takes for AI is documented as well to evaluate the efficiency of the algorithm. In real life, when two human players play against each other, there must be a time limit each round for players to think about the next move, and the winning rate is also an important standard to evaluate the capability of one player, which is why we use these 2 metrics as our performance metrics.

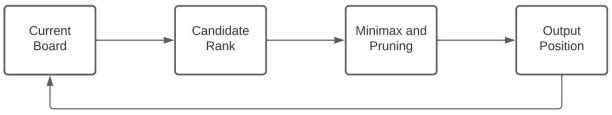
Finally, to adjust the advantage given to black piece players, we will swap the AIs' position and measure the result again, and at the end, we will take the average winning rate of black and white. To formulate the "intelligence level" of agents, we set $50 - 50$ winning rate to amateur human players like the unit for "intelligent level" (IL, which is 1.00 for human players, and 2.00 if the winning rate is 100%) and take the average level of all players that one particular agent has played against to with regards to the winning rate:

$$IL_A = \frac{1}{|O(A)|} \sum_{i \in O(A), i \neq A} (W(A:i) + W(i:A)) \cdot IL_i$$

where $O(A)$ refers to all players that A have played against to and $W(A:B)$ is the winning rate of A when A is black and B is white.

Implementation

The general architecture of the AI is as follows. Whenever the computer is to make the next move, it first generates a rank for each candidate. The candidates are then evaluated by the heuristic function and assigned a score to each of them. After that, AI uses minimax search with pruning to determine the position of the next move and update the board.



We developed three different versions of AI - AI 0.0, AI 1.0 and AI 2.0.

For AI 0.0, we use "one step with rank". We define 7 different ranks as described before and AI 0.0 assigns the ranks to each possible next move. Then the AI 0.0 chooses the candidate with the highest rank as the next move. If multiple candidates have the same rank, AI 0.0 chooses randomly from them. We want our AI agent to be more "intelligent", so we introduce the heuristic - score in the net version, AI 1.0.

For AI 1.0, we use "one step with rank + score". Based on the heuristic function mention above, AI 1.0 chooses the one with the highest "score" (heuristic) from the candidates with the same rank. There could hardly be a tie since the "score" depends on the neighbor of the candidates in all eight directions, which is very complex. To reduce the randomness more, we use minimax algorithm in our final version - AI 2.0.

For AI 2.0, we use "minimax with rank". Minimax algorithm deepens the search space and takes into account all the possible moves that players can take at any given time during the game. Given the depth and width, AI 2.0 chooses the candidates that are no worse than the existing ones, keep them in next_search_set and call minimax recursively on them. This significantly increases the winning rate, we will discuss this in the following section.

Performance

The performance of the algorithm is evaluated by AI's winning rate and the run time complexity of each move. To evaluate the which version has a higher winning rate, we wrote an automation script to make different algorithms play against each other for 100 rounds consecutively by setting different parameters when construct the "Computer" object. However, we first play against different version ourselves and find it hard to beat the AI without applying some tricky techniques.

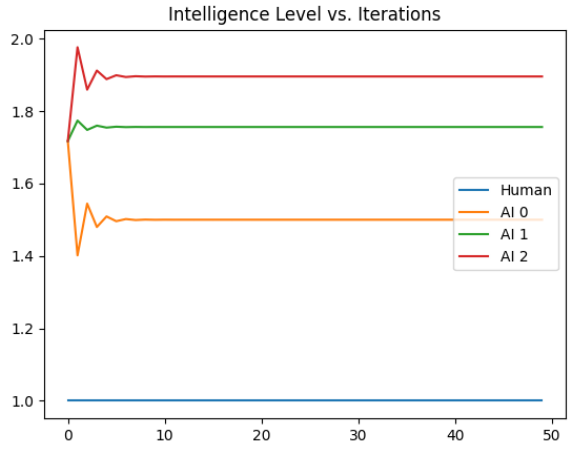
This is a system of equations that mimics the Page Rank algorithm where each AI gets some partial scores of other AIs. To get a better visualization of the result, we firstly created a table to store all the wins and losses of each traceable gameplay (some untraceable ones of Human vs. AI is not

documented since it is hard for a human to win, which may lead to inaccuracy of the result), then follows Markov process with each AI initialized with the same scores (rank). In the end, the scores are normalized according to humans' baseline score of 1.

	Human	AI 0.0	AI 1.0	AI 2.0
Human	-	26 : 49 : 25	-	-
AI 0.0	68 : 15 : 17	45 : 39 : 16	22 : 35 : 43	31 : 45 : 24
AI 1.0	-	40 : 28 : 32	48 : 34 : 18	43 : 46 : 11
AI 2.0	-	45 : 29 : 26	53 : 33 : 14	55 : 38 : 7

Table 3: Game Play result

The row and column represent the black and white player respectively, where each cell indicates (number of wins of Black: number of wins of White: number of Draws) in 100 gameplays. Human refers to average players with some experience in playing Gomoku; AI 0.0 refers to single-step greedy best search AI utilizing rank only; AI 1.0 refers to single-step greedy best search AI utilizing both rank and scores with a one-step look ahead, and AI 2.0 refers to general Minimax search AI with specified search depth and width (the above result is documented from depth = 4 and width = 3).



The run time complexity is not too bad in general since we apply pruning to reduce the amount of time to explore useless paths. We keep track of the duration time in seconds that the AI agent takes in each move, and then calculate the average response time in each game for an AI player. We can see that the more sophisticated the algorithm is, the more time it takes to predict the opponent's move. The data is documented in the following table (parameters are the same):

	AI 0.0	AI 1.0	AI 2.0
Duration (seconds/move)	0.00315	0.00298	0.28890

Table 4: Average running time for each move

Lessons We Learned

The biggest problem we encountered in Minimax is the balance between pruning and discovering. As mentioned previously, alpha-beta pruning can reduce search space significantly by pruning all moves that are worse than discovered ones. However, this traditional pruning strategy does not perform well. If we set heuristic values very differently for each pattern, then it is very rare for two configurations to have the same heuristic, which yields fast pruning but bad move generations; on the other hand, if we set them to be very similar, then it is very likely for two configurations to have the same heuristic, which leads to intelligent move generations but very slow pruning.

We solved this by a neural-network like approach: set a higher layer of heuristic called $rank$, we prune if and only if the rank is lower, and the final move choice is still done by the lower layer heuristic $score$. Patterns with similar scores are grouped together as the same rank, and will only be pruned if a definitely better move is already found. High end $rank$s serves more as a move-restriction technique (i.e. my winning move rank $>$ opponent winning move rank) and low end $rank$ contributes more to pruning the search space. To make the final move choice more accurately represent our goal, we also added a regularization score to each move in the search space, instead of dropping the score directly in the process of pruning.

The next hardest challenge comes from the restriction of Minimax itself. Recall that Minimax always assumes the opponent is perfectly rational like the AI itself and tries to minimize our payoff, whereas in reality this is not what humans would do. A move that is optimal in 8 look ahead is not always better than the one in 3 look ahead, unless the former one leads to a guaranteed win no matter what the opponents' strategy is. In other words, since at the very start of the game, AI does not have the time to exhaustively find a "guaranteed win", it is trying its best to predict the highest payoff given the opponent is using the same strategy as itself – this is not the case in general. From our experiment, AI 2.0 outperforms other AIs, but it can still lose, because it assumes AI 1.0 and AI 0.0 to have the same intelligence level as itself, it prunes many strategies that are not optimal but actually used by them, which leads to potential loses if it does not find the "guaranteed win" in its dedicated search depth and width. Observing this drawback gives us potential possibilities to modify heuristic opponent wise accordingly, which may be optimized in future implementations by deep neural networks (see discussion in D4).

Both obstacles illustrates potential issues of Minimax on Gomoku and its limitations. Although it can sustain good winning rates against amateur human players, it is far from perfect rational. We also get a better understanding from other papers work of why neural network and MCTS would be eventually more rational, because they would learn from opponents' moves and make adjustments accordingly. A perfect rational AI should not assume the opponent to be rational, but rather assumes nothing but possibilities.

Discussion

Implications

Our Algorithm mainly redesigns the Minimax used for Gomoku in two ways, point-wise Minimax and Multi-layer heuristic. Since this change is finalized by a explicitly built-in heuristic dictionary, it can be easily adopted to other Minimax based board games by constructing a neural-network like weighted matrix from this dictionary and be potentially combined with reinforcement learning approaches like MCTs.

The first and most important breakthrough is the idea of point-wise Minimax. As we mentioned above, at a particular search depth, the traditional Minimax algorithm tends to configure the board as a whole and assigns heuristic based on all possible combinations of threat space searching patterns. To contrast, point-wise Minimax wants to configure each unobserved and winnable point explicitly, and then summing them up by assigning possibly different weights. In other words, this method extends the heuristic that can only be used for the current board to all future boards (constrained by some time limit, which is what Multi-layer heuristic tries to overcome) by Minimax. Unfortunately, this method is fundamentally flawed – the time, or the depth that the move is generated is totally ignored. We cannot sum up the heuristic at different stages like what we did in the traditional Minimax since deeper stages of the board are guaranteed to have higher heuristic than earlier stages, as long as we keep heuristic non-negative. A classical way to solve this issue is to introduce discount factor at each depth. We extend this idea one step further, instead of using a fixed or trainable discounted factor, we stores all the heuristics at every depth and assign different weighted factor on each of them. This factor can be later trained by neural-network like approach as well, just like the heuristic. We can clearly see that this can be generalized to other Minimax based board games much better than using a single discounted factor.

The second innovation comes from neural network itself, which is called Multi-layer heuristic. We essentially want to build some architecture to reduce the search space of Minimax as much as possible without hurting its comprehension for the stage we are currently in. Because we use heuristic instead of full-board configuration as the building block of Minimax, this becomes plausible. Similar to how we split a single discounted factor to differently weighted ones, we can also do the same thing to heuristics. In our algorithm, we try to add an upper level heuristic called $rank$ and the lower (original) level is now $score$. We constructed 7 layers of $rank$ and each of them can serve as their own purpose. Some of them try to convey the message of exploration, i.e. current board have little threat to me, so I can go further and try more allocations of my pieces; others may restrict the moves and prune the search space, which leads to exploitation, i.e. current board is threatening, so I may want to search deeper to see what the opponent's strategy looks like. the number of layers and their intensity of sending the message is a big topic to discuss, what we do here can only serve as a baseline.

Due to time and machine constraint, we cannot build this fully connected network to see its performance, instead we manually input all the initial parameters by experiments, even at the current stage, it shows some good winning rate compared to the traditional method. We hope to see its further implications in the future.

Impact and Limitations

The intention of this paper is to create a new version of Minamax algorithm to expert gomoku. Unlike traditional Minimax algorithm, our algorithm balance between pruning and discovering by a neural network like approach. We introduce two layers of heuristics - rank and score. Rank has higher priority than score, so when comparing two moves, we no longer need to calculate the score of each pattern. Instead, patterns are grouped by different ranks, we compare the ranks of the patterns at first and calculate score of patterns only when they have the same rank. Moreover, instead of dropping the heuristic at each time in pruning, we memorize and add a regularization score to each move in the search space. In our experiments, we figured out this approach has significant better performance (i.e., higher wining rate) than the algorithm that only consider one heuristic.

Overall, we present a method by employing Minimax algorithm combined with two-layer heuristic. However, there is still a certain gap between our method and best AI program for gomoku like YiXin and AlphaGo. Our algorithm assumes that both the and the opponent follow the same strategy. However, in real world, people have preferences when making decisions. Different from AlphaGo, our AI cannot learn anything from self-playing. Further improvements in the specialization of the heuristic are one of our next research goals. Integrating with reinforcement learning techniques will be a powerful approach to improve the performance. Introducing adaptive dynamic programming which penalizes the last move taken by the loser and rewards the last move selected by the winner seems to be a suitable approach in the next stage. In the future program improvements, we will consider whether to employ deep neural network to improve the performance.

Conclusion

In summary, this report focuses on exploring a new approach of including multi-player heuristics for the AI agent player in the game Gomoku, which is a popular abstract strategy board game, also called five-in-row. The topic of this project is inspired by the development of the famous "Alpha-Go", and the motivation is the common enthusiasm and interest towards Gomoku and learning Artificial Intelligence among the group members. Since the classical Minimax algorithm does not achieve good results due to its width and depth limitations, we implement the algorithm based on it with

multiple layers of heuristic values and alpha-beta pruning by trying to assign heuristics from the neighbors of every winnable location. In short, we introduce the earlier layers to sense the attacking moves from the opponent player, while the later layers are designed to generate trainable ranks for pruning the unnecessary moves. With respect to algorithm parameters, they are constructed in a way that is similar to the neural network and they are self-correlated. Experiment results show that the algorithm can beat amateur human players in most cases with only several seconds of thinking time in each round. Also, different versions of AI agents have different levels of intelligence level. In general, AI 2.0, which uses "minimax with rank" strategy, is the best version because it has the highest intelligence level and the winning rate, and the running time for each move is within 0.5 seconds on average.

The algorithm in general demonstrates a good winning rate and short running time, but we still run into a few obstacles in the process such as potentially losing caused by the Minimax itself. It would be interesting to try a different approach: deep neural networks (Reinforcement Learning) because neural networks can train models and learn from human moves, and we can construct multiple layers to adjust the parameters to see which ones perform the best. The related topic "Game Theory" is also interesting because Gomoku is a 2-player strategy game, applying probability statistics and math models may lead to interesting findings and results.

References

Chaslot, G.; Bakkes, S.; Szita, I.; and Spronck, P. 2008. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'08, 216–217. AAAI Press.

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

Kuan Liang Tan; Tan, C. H.; Tan, K. C.; and Tay, A. 2009. Adaptive game ai for gomoku. In *2009 4th International Conference on Autonomous Robots and Agents*, 507–512.

Liao, H. 2019. New heuristic algorithm to improve the minimax for gomoku artificial intelligence. In *Creative Components*, volume 407.

Tang, Z.; Zhao, D.; Shao, K.; and Lv, L. 2016. Adp with mcts algorithm for gomoku. *2016 IEEE Symposium Series on Computational Intelligence (SSCI)* 1–7.

Yu, H. 2016. From deep blue to deepmind: What alphago tells us. *Predictive Analytics and Futurism* (13):42–45.