# A GOMOKU ARTIFICAL INTELLIGENCE AGENT

**Jiacheng Hou**
School of EECS, University of Ottawa, Ottawa, Canada

**ABSTRACT**

Gomoku is also named Five in a Row. It is a popular game in the world, especially in Asia. This report explores various Artificial Intelligence (AI) agents that play Gomoku. In particular, the Minimax and Minima with Alpha-Beta Pruning algorithms are utilized. In addition, this report proposes a heuristic evaluation function that assigns utility values to each game state. Heuristic evaluation functions with various parameters are also explored. Moreover, the report compares the performances of various AI agents against a baseline AI agent and a Minimax AI agent with a search depth of 1, respectively. Finally, future work is planned at the end of the report.

***K*eywords** Gomoku · Minimax · Minimax with Alpha-Beta Pruning · Heuristic Evaluation Function

## 1 Introduction

Gomoku, also called Five in a Row, originates from Japan. It is a two-player, turn-based and zero-sum game played with GO pieces (black and white stones) on a GO board. Black always goes first, and then players alternately place the stone in an empty intersection of the board, and stones can not be moved once placed. The first player who gets five pieces horizontally, vertically or diagonally wins the game.

Gomoku is an exciting game, and it is pretty popular across Japan, Korea and China. In China, we call the game Wuziqi, 'Wu' means five, 'Zi' means piece, and 'Qi' is a board game category in China. Gomoku brought much fun to the author's childhood, although the author did not have a wise strategy at that time.

In 2016, the birth of AlphaGo [1], an Artificial Intelligence (AI) agent, shocked the world by beating a Go world champion. Go is thought to be the most challenging board game globally because there are around $10^{172}$ possible board positions. It gives the author an insight into the power of AI and inspires the author to build a Gomoku AI player. The objectives of this report are as follows:

- Build various AI agents for playing Gomoku: (i) Minimax AI agents with various search depth limitations, (ii) Minimax with alpha-beta pruning AI agents with various search depth limitations, (iii) A baseline AI agent that only attacks, not defends.
- Simulate various AI vs. AI competition environments.

- Evaluate AI agents' performances based on three evaluation metrics: (i) Winning ratio, (ii) Draw ratio, and (iii) Average time per step.

The rest of this report is organized as follows: Section 2 overviews background. Section 3 introduces the methodologies and implementations. Section 4 presents experiments. Section 5 are results. Section 6 concludes the report.

## 2 Background

### 2.1 Adversarial Search and Adversarial Games

Adversarial search [2] refers to searching in competitive environments. A competitive environment has multiple agents, and agents' goals are in conflict. Games are good candidates to learn adversarial search for several reasons: (i) games are fun, (ii) games states are easy to represent, (iii) all agents are restricted to a limited number of actions in games, (iv) the outcomes of agents' actions are defined by precise rules. Therefore, this report explores adversarial games where players can apply adversarial search algorithms.

There are many kinds of games worldwide, such as Sudoku, Tic-Tac-Toe, Gomoku, etc. In Sudoku, when we are playing the game, no one else interferes with us, and thus it is a one-player game. Unlike Sudoku, Tic-Tac-Toe and Gomoku are two players games, and the two players compete with each other.

In this report, adversarial games that have the following properties are explored:

- Two players: one player and one opponent have opposing goals.
- Turn-based: players take turns when playing the game.
- Zero-sum: whenever one player wins, the other loses the game.
- Perfect information: each player has complete information about the opponent's position and available choices.
- Deterministic: there is no randomness to the game. It means that the outcome is defined whenever a decision is made, and there will be no surprise at all.

This report focuses on such an adversarial game, Gomoku.

### 2.2 The Game of Gomoku

As briefly described in Section 1 on Gomoku, Gomoku [3] is a board game that originated in Japan. It can be played on a 9*9 [4], 15*15 or 19*19 board. The game has two players, a white player and a black player. Two players take turns placing its stone on the intersection of the square rather than in the middle of the square, which is shown in Figure 1. Once a player has placed five stones in a row, in a column or in a diagonal, that player wins.

### 2.3 Gomoku and Adversarial Search Algorithms

There are a number of adversarial search algorithms available for implementing Gomoku AI agents. The most popular approach is Minimax, an algorithm that is further improved by the Alpha-Beta pruning algorithm. On top of this, a solid heuristic evaluation function is necessary to help reduce the decision time. In this project, the Minimax algorithm

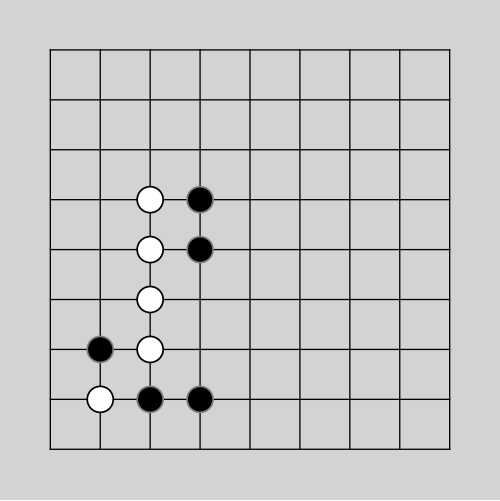


Figure 1: A Gomoku example on a 9*9 board

is first implemented; subsequently, the Minimax with the Alpha-Beta pruning algorithm is utilized. At the same time, a heuristic evaluation function is proposed.

## 3 Methodologies and Implementations

This section describes the adversarial search algorithms used in this project, namely Minimax and Minimax with Alpha-Beta pruning algorithms. For each algorithm, a pseudo-code and complexity analysis are presented. In addition, a heuristic evaluation function is proposed to assign utility values to both intermediate and terminal game states.

### 3.1 Minimax Algorithm

The Minimax algorithm [5] was initially being proposed for zero-sum games with n players in which the players take alternative or simultaneous decisions. It is widely used in games such as Tic-Tac-Toe, Chess and Gomoku. In a two-player game, there are two players, MAX and MIN. the MAX player always goes first, and they take turns until the game ends. A game can be formally defined as a search problem, as follows:

- **Initial state**: the initial configuration of the game. For example, the initial configuration of Gomoku is an empty board.
- **Successor function**: a list of (move, state) pairs specifying legal moves.
- **Terminal test**: a Boolean value indicating if the game is finished.
- **Utility function**: a utility value for the state of the game. It is worth noting that this utility value is from the MAX player's point of view.

In the Minimax algorithm, the goal of player MAX is to optimize the payoff, but the MIN player minimizes the payoff that MAX can obtain. Minimax works by recursively constructing a state-space tree, traversing possible successors until a leaf node is reached. Then, as the recursion unfolds, the utility of the leaf node states is backed up.

The Algorithm 1 [6] shows the pseudo-code of the Minimax algorithm, which accepts three parameters, *node* is the current game state, *depth* is the number of steps the player wishes to view further before making the current decision, and *maximizingPlayer* is a Boolean value that indicates whether it is the MAX player's turn or not. The base case is that when the search depth is 0, or the node is a terminal node, then the utility value of the node is returned. Otherwise, when it is the MAX player's turn, a recursion occurs for each child node of that node, where the function Minimax is called and the child node, search depth is reduced by one, and *maximizingPlayer* is FALSE. In the for loop, the utility value of the current node is updated to the larger of its utility value and that of its child. Outside the for loop, the utility value of the node is returned. On the other hand, if it is the turn of the MIN player, then the lower value between the utility value of the current node and the utility values of its children is assigned to the utility value of the current node. We can observe that if *depth* is initialized to $+\infty$, then the Minimax algorithm performs a complete depth-first search of the game tree.

Ideally, a minimization algorithm would search a complete game tree and make the most optimistic move assuming that its opponent plays optimally. However, this is a problem for games with large state spaces, such as Gomoku. Let $b$ be the branching factor of the state and $d$ be the depth of the state space tree. Then the time complexity of the minimization algorithm is $O(b^d)$, and the space complexity is $O(bd)$. In a 9 * 9 Gomoku board of depth 5, the time complexity is about $81^5 = 3486784401$. As the search depth increases, the time complexity increases exponentially. In reality, it is not feasible to search for a complete game tree.

**Algorithm 1** Minimax Algorithm

```
1:  function minimax(node, depth, maximizingPlayer)
2:  if depth = 0 or node is a leaf node then
3:      return utility value of node
4:  end if
5:  if maximizingPlayer then
6:      maxValue = −∞
7:      for Each child of node do
8:          value = minimax(child, depth - 1, FALSE)
9:          maxValue = max(maxValue, value)
10:     end for
11:     return maxValue
12: else
13:     minValue = +∞
14:     for Each child of node do
15:         value = minimax(child, depth - 1, TRUE)
16:         minValue = min(minValue, value)
17:     end for
18:     return minValue
19: end if
```

**Algorithm 2** Minimax with Alpha-Beta Pruning Algorithm

```
function minimax-alpha-beta(node, depth, α, β, maximizingPlayer)
if depth = 0 or node is a leaf node then
    return utility value of node
end if
if maximizingPlayer then
    maxValue = −∞
    for Each child of node do
        value = minimax-alpha-beta(child, depth - 1, α, β, FALSE)
        maxValue = max(maxValue, value)
        α = max(α, value)
        if β < α then
            break
        end if
    end for
    return maxValue
else
    minValue = +∞
    for Each child of node do
        value = minimax-alpha-beta(child, depth - 1, α, β, TRUE)
        minValue = min(minValue, value)
        β = min(β, value)
        if β < α then
            break
        end if
    end for
    return minValue
end if
```

### 3.2 Minimax with Alpha-Beta Pruning Algorithm

In order to improve the Minimax algorithm search complexity, a Minimax with Alpha-Beta pruning [5] algorithm is utilized. This pruning algorithm adds two parameters for each tree node, $\alpha$ and $\beta$, where $\alpha$ means the best (max) value can be achieved for the MAX player so far, and $\beta$ means the best (min) value can be achieved for the MIN player so far.

Algorithm 2 [7] shows Minimax with the Alpha-Beta pruning algorithm. unlike Algorithm 1, the algorithm accepts two more parameters $alpha$ and $beta$, which are initialized to $-\ infty$ and $+\infty$, respectively. If it is the MAX player's turn, then $alpha$ is updated to the greater of $alpha$ and its child utility value. If $beta$ is less than or equal to $alpha$, then its parent node, the MIN player, already knows a path that can lead to a lower utility value. Therefore, the MIN player will never choose this node, so the other branches of this node can be pruned away. Similarly, if the MIN player's turn

is, then $\beta$ will be updated to the lower value between $\beta$ and its child utility value. If $\beta$ is less than or equal to $\alpha$, which means that in the previous step, the MAX player has already known a path that leads to a larger utility value, then the MAX player will never choose this node, so the other children of the current node can be cut off. It is worth noting that the Alpha-Beta pruning technique does not affect the final decision. Whether the pruned branches are explored will not affect the final decision.

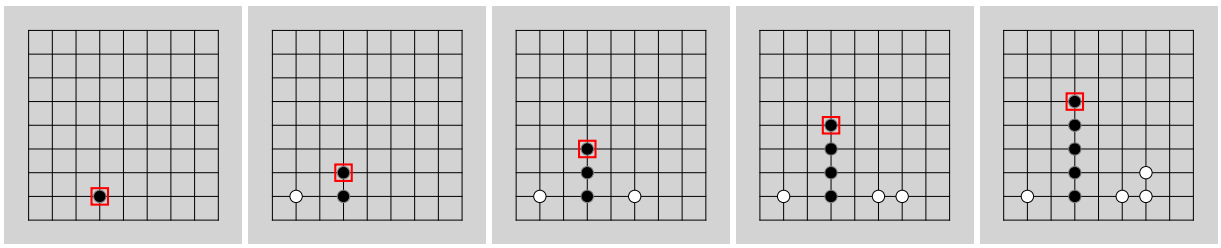
In the optimal situation where everything is ordered right, the time complexity of the Minimax algorithm with alpha-beta pruning is $O(b^{d/2})$ [8]. Compared to the Minimax algorithm, it can perform double the depth search within the same amount of time, or do half less work when searching the same depth of the state space tree.

### 3.3 Heuristic Evaluation Function

The Minimax Alpha-Beta Pruning algorithm can reduce lots of computation time compared with the Minimax algorithm. However, in reality, we may not allow the AI agent to search for a complete game tree, and we only want the AI agent to search for several further steps to judge how good or how bad the move is. This case involves a heuristic evaluation function to assign utility values to intermediate states. In order to estimate the utility value of each game state, the concepts of attacks and defences are introduced [9, 10].

#### 3.3.1 Attacks

An *Attack* is defined as increasing the number of consecutive stones a player has in a row, column or diagonal. Figure 2 shows examples of five attacks from the black player's perspective. After placing a stone in the red box, the black player can form one, two, three, four, and five in a column, respectively. In particular, Attack5 is also a winning state.



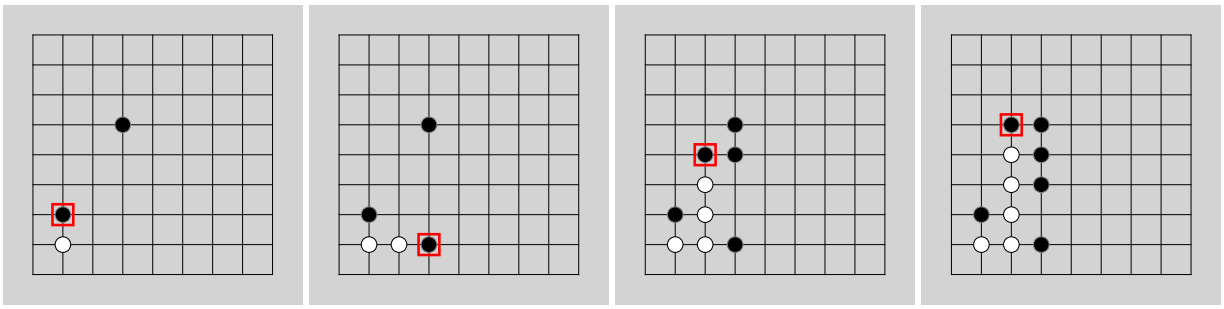
(a) Attack1 (b) Attack2 (c) Attack3 (d) Attack4 (e) Attack5/Win

Figure 2: On a 9*9 board, Gomoku Black **Attacks**, assuming it is now Black's turn and the stones in the red box are newly placed stones.

#### 3.3.2 Defences

A *Defence* is defined as breaking the opponent's consecutive stones in the row, column or diagonal. Figure 3 shows an example of four defences from the black player's perspective. After placing a stone in the red box, the black player can break one, two, three, and four consecutive white stones in a column, respectively. In particular, for all game states, both defences and attacks are formed when the black player places his stone in the red box.

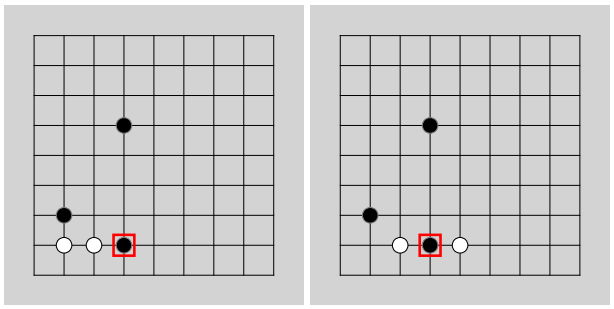
It is worth noting that there are more cases of Defense 2, as shown in Figure 4. Figure 4b is a case of breaking two consecutive white stones since there is a white neighbour on the left and one on the right, near the newly placed black



(a) Defence1 + Attack1 (b) Defence2 + Attack1 (c) Defence3 + Attack2 (d) Defence4 + Attack2

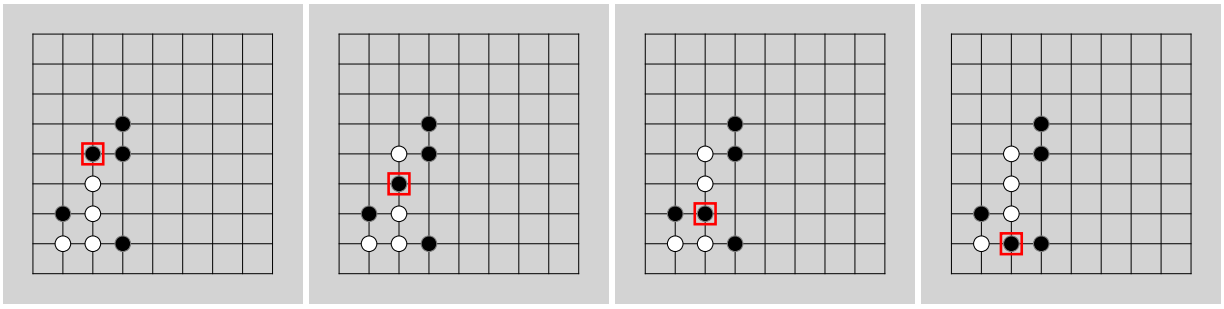
Figure 3: On a 9*9 board, Gomoku Black **Defences**, assuming it is now Black's turn and the stones in the red box are newly placed stones.

stone. In addition, the stones in the red box form an attack 1. Similarly, there are more cases of Defence 3 and Defense 4, shown in Figure 5 and Figure 6, respectively



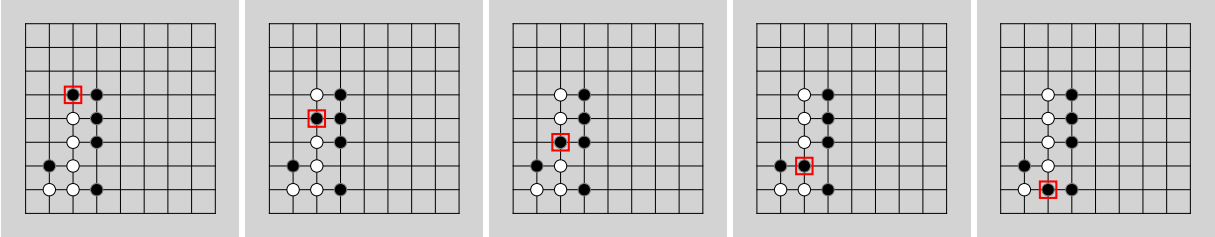
(a) Defence2 + Attack1 (b) Defence2 + Attack1

Figure 4: On a 9*9 board, Gomoku Black **Defence2**, assuming it is now Black's turn and the stones in the red box are newly placed stones.



(a) Defence3 + Attack2 (b) Defence3 + Attack3 (c) Defence3 + Defence1 + Attack2 (d) Defence3 + Defence1 + Attack2

Figure 5: On a 9*9 board, Gomoku Black **Defence3**, assuming it is now Black's turn and the stones in the red box are newly placed stones.



(a) Defence4 + Attack2 (b) Defence4 + Attack2 (c) Defence4 + Attack2 + Attack3 (d) Defence4 + Attack2 (e) Defence4 + Attack2

Figure 6: On a 9*9 board, Gomoku Black **Defence4**, assuming it is now Black's turn and the stones in the red box are newly placed stones.

#### 3.3.3 Evaluation Function

The utility value of each game state is defined as follows:

$$val = w1 * \text{No\_of\_Attack1} + w2 * \text{No\_of\_Attack2} + w3 * \text{No\_of\_Attack3} + w4 * \text{No\_of\_Attack4} + w5 * \text{No\_of\_Attack5} + w6 * \text{No\_of\_Defence1} + w7 * \text{No\_of\_Defence2} + w8 * \text{No\_of\_Defence3} + w9 * \text{No\_of\_Defence4},$$

where *w1, w2, w3, w4, w5, w6, w7, w8* and *w9* are constants, and No_of_Attack1, No_of_Attack2, No_of_Attack3, No_of_Attack4, No_of_Attack5, No_of_Defence1, No_of_Defence2, No_of_Defence3, No_of_Defence4 are the total number of Attack1, Attack2, Attack3, Attack4, Attack5, Defense1, Defense2, Defense3 and Defense4 formed on the row, column and diagonal after placing the stone, respectively. In addition, if the player loses the game, the game state's utility value from the player's point of view is $-\infty$.

## 4 Experiments

In this project, the Gomoku board size is 9 * 9. Three AI agents are simulated, a Minimax AI agent, a Minimax with Alpha-Beta Pruning AI agent and a baseline AI agent. For Minimax and Minimax with Alpha-Beta Pruning AI agents, the weights of *w1, w2, w3, w4, w5, w6, w7, w8* and *w9* are -1, 0.1, 0.2, 0.4, $+\infty$, 0.1, 0.2, 5, and 10 respectively. The reason *w1* is initialized as -1 is to prevent the AI agent from placing a stone that is far away from any other stones on the board. For the baseline player, the weights of *w1, w2, w3, w4, w5, w6, w7, w8* and *w9* are -1, 0.1, 0.2, 0.4, $+\infty$, -10, -10, -10, and -10 respectively. It is easy to see that the baseline player only attacks and does not defend.

Furthermore, fourteen AI agents competition environments are simulated, which are: the baseline AI agent where the search depth is 1 (Baseline-depth1), the Minimax AI agent with the search depth of 1 (Minimax-depth1), the Minimax AI agent with the search depth 3 (Minimax-depth3), the Minimax with Alpha-Beta Pruning AI agent with the search depth of 1 (Minimax-alpha-beta-depth1), the Minimax with Alpha-Beta Pruning AI agent with the search depth of 3 (Minimax-alpha-beta-depth3), the Minimax with Alpha-Beta Pruning AI agent with the search depth of 5 (Minimax-alpha-beta-depth5) and the Minimax with Alpha-Beta Pruning AI agent with the search depth of 7 (Minimax-alpha-beta-depth7) vs. a Baseline-depth1 and a Minimax-depth1, respectively. All the experiments are run 20 times, and evaluation metrics are calculated on average.

Three evaluation metrics are adopted to evaluate AI agents' performances [4]:

- Winning ratio: It is defined as the number of winning trades divided by the total number of trades.
- Draw ratio: It is defined as the number of draw trades divided by the total number of trades.
- Average time per step: It is defined as the average time an AI agent spent, in seconds, to make a single decision.

Table 1: AI agents' winning ratio, draw ratio and average time per step in seconds against a Baseline-depth1

| Agents | Winning ratio | Draw ratio | Average time per step (sec) |
|---|---|---|---|
| Baseline-depth1 | 0.5 | 0.0 | 0.00148 |
| Minimax-depth1 | **1.0** | 0.0 | **0.00144** |
| Minimax-depth3 | **0.0** | 0.0 | **6.05090** |
| Minimax-alpha-beta-depth1 | **1.0** | 0.0 | **0.00141** |
| Minimax-alpha-beta-depth3 | **0.45** | 0.0 | **0.18298** |
| Minimax-alpha-beta-depth5 | 0.45 | 0.0 | 10.21738 |
| Minimax-alpha-beta-depth7 | 0.7 | 0.0 | 167.90123 |

Table 2: AI agents' winning ratio, draw ratio and average time per step in seconds against a Minimax-depth1

| Agents | Winning ratio | Draw ratio | Average time per step (sec) |
|---|---|---|---|
| Baseline-depth1 | 0.0 | 0.0 | 0.00146 |
| Minimax-depth1 | 0.0 | **1.0** | 0.00077 |
| Minimax-depth3 | 0.0 | **0.0** | 6.91080 |
| Minimax-alpha-beta-depth1 | 0.0 | **1.0** | 0.00077 |
| Minimax-alpha-beta-depth3 | 0.0 | **0.0** | 0.93425 |
| Minimax-alpha-beta-depth5 | 0.0 | **0.0** | 12.83126 |
| Minimax-alpha-beta-depth7 | 0.0 | **0.0** | 185.80305 |

## 5 Results

Table1 shows the performance of the AI agent for Baseline-depth1. We can see that the average time for each step increases as the search depth increases. In addition, the Minimax Alpha-Beta pruning algorithm can substantially reduce the time spent on decision-making. Minimax-alpha-beta-depth1 searches 2% faster than Minimax-depth1 without affecting the winning ratio. Minimax-alpha-beta-depth3 searches faster than Minimax-depth3 is 97% faster and has an even higher winning ratio. We can conclude that the Minimax Alpha Beta pruning algorithm dramatically improves the computation time. In addition, we can observe that both Minimax-Depth1 and Minimax-alpha-beta-depth1 achieve a winning ratio of 100%; however, the winning ratio decreases a lot when the search depth increases. It is probably because the Minimax algorithm assumes the opponent's player is optimal. However, this is not the case when the opponent is Baseline-depth1.

Table 2 shows AI agents performances against a Minimax-depth1. Interestingly, no AI agent can compete with its opponent, a Minimax-depth1. In addition, the table shows that the draw ratio is 100% for the Minimax-depth1 and Minimax-alpha-beta-depth1 vs. Minimax-depth1. The reason is that the two players play the game optimally, and they are the optimal opponent against each other. However, when the search depth is larger, the Minima-depth1 is no longer an optimal opponent from the perspective of Minimax-depth3, Minima-alpha-beta-depth3, Minima-alpha-beta-depth5 and Minimax-alpha-beta-depth7, and thus they have worse performances.

## 6 Conclusion

In conclusion, this project utilized the Minimax and Minimax with Alpha-Beta Pruning algorithms to implement the Gomoku AI agent. A heuristic evaluation function was proposed to assign utility values to intermediate and terminal game states. Various AI competitive environments were simulated. Experimental results showed that the time spent on decision-making increases with the search depth. Furthermore, the Minimax with Alpha-Beta pruning algorithm reduces the decision time a lot compared to the Minimax algorithm. Last but not least, the Minimax algorithm and Minimax with Alpha-Beta Pruning are designed for optimal adversaries, so it is difficult to achieve satisfactory results when facing non-optimal adversaries.

In the future, simulated annealing could be used to allow a player to temporarily choose a worse move that may lead to a good state in future steps. In addition, games with three players could be explored to make the game more exciting and fun.

## References


[1] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[2] Adversarial search. `https://www.academia.edu/6954459/Lecture_6_Advanced_Search_Methods_Local_Beam_Search_Games_Alpha_Beta`. Accessed: 2022-04-20.

[3] Gomoku. `https://en.wikipedia.org/wiki/Gomoku`. Accessed: 2022-04-20.

[4] Ai agent for playing gomoku. `https://stanford-cs221.github.io/autumn2019-extra/posters/14.pdf`. Accessed: 2022-04-20.

[5] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[6] Shubhendra Pal Singhal and M Sridevi. Comparative study of performance of parallel alpha beta pruning for different architectures. In *2019 IEEE 9th International Conference on Advanced Computing (IACC)*, pages 115–119. IEEE, 2019.

[7] Gomoku ai player. `https://pats.cs.cf.ac.uk/@archive_file?p=525&n=final&f=1-1224795-final-report.pdf&SIG=a852f388b81ea43c0953ec0fb298084d16361371285eb9b836422360627fbd64`. Accessed: 2022-04-20.

[8] Search: Games, minimax, and alpha-beta. https://www.youtube.com/watch?v=STjW3eH0Cik&t=2031s&ab_channel=MITOpenCourseWare. Accessed: 2022-04-20.

[9] Jiun-Hung Chen and Adrienne X. Wang. Five-in-row with local evaluation and beam search. 2004.

[10] Louis Victor Allis, Hendrik Jacob Herik, and Matty PH Huntjens. *Go-moku and threat-space search*. University of Limburg, Department of Computer Science Maastricht, The . . . , 1993.