# Gomoku Artificial Intelligence Project

**Josh Adkins**
**Henry Paek**
**Ian Fair**

# Table of Contents

## Task Definition

**Background Information:**

Gomoku, also known as “Five in a Row” is a strategy board game played on a 15 x 15 grid. The game is played with two different colored stones. Stones are colored “black” and “white” and correspond to the player’s preference. Whichever color stone a player chooses at the start of the game is their color for the entire duration of the game. During the game, players take turns placing their chosen colored stone. The first player to align five stones of the same color in a row, be it horizontally, vertically, or diagonally, wins the game.

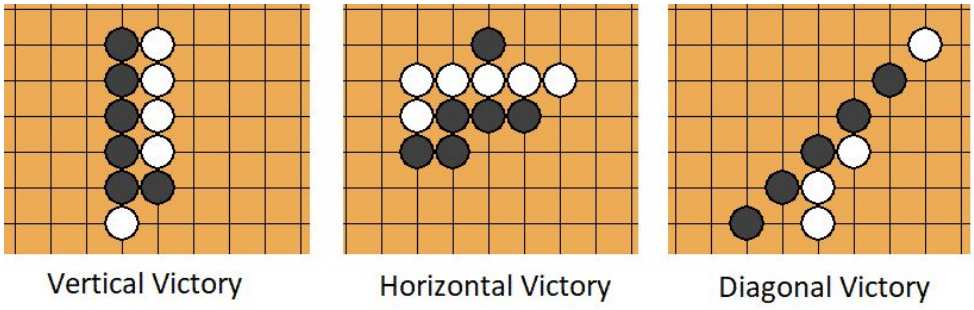



Figure 1: Showing example victories in vertical, horizontal, and diagonal directions.

**Task Objective:**

For our project, our group has decided to create a program that solves a Gomoku game. In our program, a user takes turns playing against an A.I. computer on a digitally rendered 15 x 15 game board. Once a player makes a move (input) the A.I. calculates a counter move(output) by taking into account the player's possible future horizontal, vertical, and diagonal actions. The end goal of our program is to have a successful A.I. play against the player.

## Infrastructure

**Experimental Setting:**

In order to set up our foundation and starting point for our game-based Gomoku project, we needed to choose a programming language that best suited our goal. For this, we decided on using the Python programming language because it best fit our needs for designing a game with its limitless modules and great adaptability for artificial intelligence. For our programming environment and development setting, choosing an IDE (or lack-there-of) had no impact on this project, as long as the proper modules were imported.

**Required Libraries/Packages:**

To help with the design and implementation of our Gomoku project, we elected to use the following modules; Python3 Pygame, Numpy, Sys, and Time. The reason we decided to use the Pygame module is because it allowed us to create the 15 x 15 game board, game pieces, and game rules. The Pygame module also helped us with the implementation of our game board by using a 2D array to indicate piece position and determine neighbors. The rest of the imported modules such as Numpy, Sys, and Time are required because they allow the A.I. to

interact with the established framework in order to gain information and make decisions.

**Computer Hardware (Recommended):**

CPU: 2.8 GHz Dual-Core Intel i7 or better

GPU: Intel Iris 1536MB or better

Memory: 8GB RAM or better

OS: macOS Catalina or Windows 10

## Approaches And Experiments

### Initial Approach vs Decided Approach:

For our initial approach, we decided to implement a form of hill-climbing and involve simulated annealing. After further thought, we then decided to try and use Markov Decision Process or MDP. After meeting with professor Bo during our midterm presentation and hearing his comments on using MDP, we removed that idea and decided on implementing alpha-beta pruning with a decision tree and a greedy approach to scoring. Our reason for deciding on using alpha-beta pruning was because it takes into account both the player and the A.I. Alpha-beta pruning removes unnecessary move trees, which lowers the amount of time required to generate moves.

### Baseline:

The baseline was initially planned as an algorithm that applies hill-climbing for picking the best position. After running the scoring function on every empty position, the algorithm chooses the best neighbor among the ones that have the same highest score. But as we progressed, we realized that the most pertinent algorithm for continuing the game is the scoring function, not the algorithm for

picking up one element among the already scored neighbors. We decided to abandon the idea of applying a hill-climbing algorithm and started focusing on more refined ways of building algorithms for the scoring function. One of the ideas that we came up with was using MDP to score each empty position when it's the A.I.'s turn - which is accomplished by giving a specific amount of reward for each empty position. But, we decided to abandon the idea of applying MDP too, because calculating the different reward for each empty position was already a huge obstacle when applying the MDP.

## Algorithm:

Our algorithm makes use of Alpha-beta pruning by searching through a decision tree for possible moves that could be made by both the player and A.I. Our algorithm attempts to maximize the value of A.I. moves while simultaneously minimizing the value of player moves. The algorithm we created checks for wins by searching the x-y position of each stone added to a line. If the length of the line is 5 (meaning 5 stones in a line of the same color) it will return true. If no such line or lines are found, it will return false. Lastly, our algorithm makes use of move depth to establish intelligence difficulty. When our algorithm is set to a depth of 3 the A.I. plays at a beginner level, when the depth is set to 1 it has the potential to beat the player.

## Scoring Function:

The scores for specific movements were determined by three main factors: The number of pieces in a line, the number of player pieces blocking, and the closeness of certain pieces. The number of pieces in a line helps determine how likely the A.I. is to create a winning move for this position. It also takes into

account piece counts above 5, where multiple rows can intersect and create unbeatable moves. These values are worth exponentially more score, both due to their difficulty in executing and increasing worth. However, the player can block certain placements, lowering the overall value of these positions. These do not set the value to 0, because certain moves can still set up other moves. Finally, the A.I. should slightly value placing pieces closer together in order to set up future moves and prevent opportunities to cut moves off. These are not so valuable as to overcome the scores for pieces being placed in a row, but they can be the determinator in ties.

```
def get_diagonalScore(board, xpos, ypos):
    diagonal = numpy.diagonal(board, ypos-xpos)
    length = len(diagonal)
    max_Score = 0
    score = 0
    row_pos = (xpos-ypos)
    if row_pos >= 0:
        abs_pos = xpos - (xpos-ypos)
    else:
        abs_pos = ypos - (ypos-xpos)
    for i in range(length):
        for j in range( min(length-i, 5) ):
            #print("Slide Diagonal")
            if diagonal[i+j] == ai_turn:
                #print("Diagonal: ", i, j)
                score += 1 + (2/ (abs(i+j-abs_pos) + 1))/NEARNESS_SCORE
            elif diagonal[i+j] == player_turn:
                score -= 1
        if max_Score < score:
            max_Score = score
        score = 0
    return max_Score**IN_A_ROW_SCORE
```

```
def get_antiDiagonalScore(board, xpos, ypos):
    anti_diagonal = numpy.diagonal(numpy.fliplr(board), 14-xpos-ypos)
    length = len(anti_diagonal)
    max_Score = 0
    score = 0
    row_pos = 14-xpos-ypos
    if row_pos >= 0:
        abs_pos = (4-xpos)-(4-xpos-ypos)
    else:
        abs_pos = ypos-(4-ypos-xpos)
    for i in range(length):
        for j in range( min(length-i, 5) ):
            #print("Slide Anti-Diagonal")
            if anti_diagonal[i+j] == ai_turn:
                #print("Anti-Diagonal: ", i, j)
                score += 1 + (2/ (abs(i+j-abs_pos) + 1))/NEARNESS_SCORE
            elif anti_diagonal[i+j] == player_turn:
                score -= 1
        if max_Score < score:
            max_Score = score
        score = 0
    return max_Score**IN_A_ROW_SCORE
```

```
def get_horizontalScore(board, xpos, ypos):
    lmax = max(0, xpos-4)
    rmax = min(14,xpos+4)
    max_Score = 0
    score = 0
    for i in range(lmax, rmax-4):
        for j in range(5):
            #print("Slide Horizontal")
            if board[i+j][ypos] == ai_turn:
                #print(i+j, ypos)
                score += 1 + (2 / (abs(i+j-xpos) + 1))/NEARNESS_SCORE
            elif board[i+j][ypos] == player_turn:
                score -= 1
        if max_Score < score:
            max_Score = score
        score = 0
    return max_Score**IN_A_ROW_SCORE
```

```
def get_verticalScore(board, xpos, ypos):
    umax = max(0, ypos-4)
    dmax = min(14,ypos+4)
    max_Score = 0
    score = 0
    for i in range(umax, dmax-4):
        for j in range(5):
            #print("Slide Vertical")
            if board[xpos][i+j] == ai_turn:
                #print(i+j, xpos)
                score += 1 + (2 / (abs(i+j-ypos) + 1))/NEARNESS_SCORE
            elif board[xpos][i+j] == player_turn:
                score -= 1
        if max_Score < score:
            max_Score = score
        score = 0
    return max_Score**1.3
```

Figures 1-4 (left to right) displaying our scoring function.

# Literature Review

## Literature Review:

Gomoku is a strategy game similar to that of tic-tac-toe. The only difference is that Gomoku is played on a larger board, usually a 15 x 15 grid. The goal of the game is to get 5 in a row of the same color. Because of the game's simplicity, it is largely used to experiment in the field of artificial intelligence. Prior groups, independent from each other, have used various algorithms to establish an A.I. vs. player game. In these games, the A.I.'s goal is to defeat the player and the player's goal is to defeat the A.I. One such approach involved using two units for training the A.I., the first being a policy-value network and the second unit being a Monte Carlo Tree Search or MCTS (Xie, Z., Fu, X., & Yu, J.). A policy-value network functions like the brain of a human being. It functions like a human brain by observing the current board and generating judgments similar to human intuition (Xie, Z., Fu, X., & Yu, J. ). The way an MCTS works is that it simulates multiple potential outcomes all stemming from the current state of the player and A.I., then uses those simulations to make a move. Another approach by a different individual involved implementing a minimax algorithm combined with alpha-beta pruning (Salamone, L.). Both approaches involved various levels of complexity, but both achieved the same end goal. When deciding how our group should

approach our project after reading these papers, we elected to use a form of alpha-beta pruning because the approach was simplistic as well as less taxing on our computer hardware as opposed to using a policy-value network combined with an MCTS.

## References:

Xie, Z., Fu, X., & Yu, J. (2018, September 27). AlphaGomoku: An AlphaGo-based Gomoku Artificial Intelligence using Curriculum Learning. Retrieved November 30, 2020, from https://arxiv.org/abs/1809.10595

Salamone, L. (2020, May 19). Creating an AI for Gomoku. Retrieved November 30, 2020, from https://medium.com/@LukeASalamone/creating-an-ai-for-gomoku-28a4c84c7a52

# Analysis

## Data Analysis:

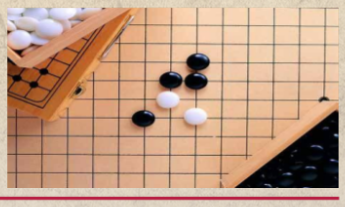
While the depth of the search improves the capabilities of the A.I., there are certain factors that remain constant. The A.I. is able to adequately interpret and follow the rules while acting in a way that moves toward achieving its goal (with increasing competency as depth increases). The A.I. at a depth of 1 is capable of playing intelligently but is not particularly proficient. But, it is possible for it to play well enough to beat a beginner sometimes. Due to its depth of 1, it is unable to think in regards to the player, making it unable to plan ahead well or act offensively. At a depth of 3, the A.I. is able to play more intelligently, acting similar to a beginner in terms of adaptability. A depth higher than 3 was intractable with our hardware but is presumed to increase the competency and adaptability with decreasing returns.

## Error Analysis:

One of the largest sources of error is the small sample size. We are unable to perform experiments at very large depth values, limiting the number of samples as well as the efficiency of the results. Also, the scoring function takes into account A.I. moves more than it does player moves, making it less effective at

determining the value for a player's move or the value in preventing a player's move. Once again, due to hardware limitations, we weren't able to build a large decision tree from the beginning, requiring the value to be calculated for each move. This both limits the number of turns the A.I. can predict ahead as well as limits the accuracy at the tradeoff of speed and tractability.

# GOMOKU PROJECT



BY: JOSH ADKINS, HENRY PAEK, IAN FAIR

## PROJECT DESCRIPTION

- Our group has decided to create a program that solves a Gomoku game.
- Gomoku is a simplified form of Go.
- Played on a 15 x 15 grid
- Two players take turns placing colored stones (black or white)
- Players choose their color at the beginning of the game
- First to align 5 in a row (horizontally, vertically, diagonally) in their colored stone wins.

## MODEL

- Uses the Python3 Pygame module to create the game board, pieces, and implement the rules.
  - Implement the board using a 2D array to indicate piece position and determine neighbors.
- Allow the AI to interact with this framework in order to gain information and make decisions.

## ALGORITHM

- Initially trialled using an MDP, but decided on Alpha-Beta Pruning with a Greedy Approach to scoring.
- Searches through the possible moves the AI and player can make.
  - Attempts to maximize the value of AI moves
  - Attempts to minimize the value of Player moves
  - Is limited to a certain depth of moves, in order to restrict hardware usage.
  - Alpha and Beta Pruning removes unnecessary move trees, lowering the amount of time required to generate moves.
  - Checks for wins by searching the x-y position of each stone added to a line, and if the length of the line is 5 (which means there are 5 stones in the line of the same color), it returns True. If no such lines are found, return False.
- Scores based off the amount of pieces intersecting and how close they are together.

# ALGORITHM CONT.

```
def alphabeta(copy_positions, depth, alpha, beta, aiTurn_bool, xpos, ypos):
    #print("Depth: ", depth)
    if depth == 0: # If the maximum amount of later turns to look at is reached...
        #print("Score Pos: ", xpos, ypos)
        #print("Returning Score: ", get_value(copy_positions, xpos, ypos))
        return get_value(copy_positions, xpos, ypos) # Return the heuristic

    # Finds every empty spot on the board.
    empty_spots = [] # Creates an array to hold empty spots
    for i in range(len(copy_positions)):    # Iterates through the passed board to find an
        for j in range(len(copy_positions[i])):
            if copy_positions[i][j] == 3:
                empty_spots.append(tuple([i, j]))    # Adds them
    #print("Empty Spot Found: ", empty_spots)

    if aiTurn_bool: # If it would be the AI's turn..
        #print("AB, ai")
        value = (-sys.maxsize-1)    # Set the min value

        # GO through each possible move you can make..
        for move in empty_spots:
            #print("Move: ", move)
            # Make a new board to simulate each move
            new_positions = deepcopy(copy_positions)
            new_positions[move[0]][move[1]] = ai_turn   # Perform the move on the new board
            #for i in range(len(new_positions)):
            #    print(new_positions[i])
            #input("Enter to continue...")

            #print("Value going in: ", value)
            #value = max(value, alphabeta(new_positions, depth-1, alpha, beta, False, move[
            temp = alphabeta(new_positions, depth-1, alpha, beta, False, move[0], move[1])
            if temp > value:
                #print(temp, value)
                global last_move
                last_move = (move[0], move[1])
                #print("LM Updated: ", last_move)
                value = temp
            #print("Value coming out: ", value)
            #input("Enter to continue . . . . . ")
            #print("DEBUG HERE BITCH")
            #print(alpha, beta)
            alpha = max(alpha, value)    # If it's better than what has been found, make it

            if (alpha >= beta): # If we're beating the player's best move, break immediate
                break
        return value
    else:   # If it would be the player's turn
        #print("AB, player")
        value = sys.maxsize # Set value above what is possible on the board

        # GO through each possible move you can make..
        for move in empty_spots:
            #print("Move: ", move)
            # Make a new board to simulate each move
            new_positions = deepcopy(copy_positions)
            new_positions[move[0]][move[1]] = (ai_turn+1)%2 # Perform the move on the new board
            #for i in range(len(new_positions)):
            #    print(new_positions[i])
            #input("Enter to continue...")

            #print("Value going in: ", value)
            value = min(value, alphabeta(new_positions, depth-1, alpha, beta, True, move[0], move[1]))
            #print("Value coming out: ", value)
            #input("Enter to continue . . . . . ")
            beta = min(beta, value)


            if (beta <= alpha):
                break
        return value                                    # Have to make it general between AI score and player score
```

# SCORE FUNCTION

- Gives +1 Point per piece within 5 spaces of the position; positions with more pieces nearby considered more valuable.
  - This is allowed to go above 5, to allow for setup attacks between two different directions.
- As well, gives an marginal and increasing reward based on the proximity of the piece to the position; positions that are closer to other pieces more valuable, preventing cutoff attacks as easily.

```
def get_horizontalScore(board, xpos, ypos):
        lmax = max(0, xpos-4)
        rmax = min(14,xpos+4)
        max_Score = 0
        score = 0
        for i in range(lmax, rmax-4):
                for j in range(5):
                        #print("Slide Horizontal")
                        if board[i+j][ypos] == ai_turn:
                                #print(i+j, ypos)
                                score += 1 + (2 / (abs(i+j-xpos) + 1))/20
                if max_Score < score:
                        max_Score = score
                score = 0
        return max_Score
```

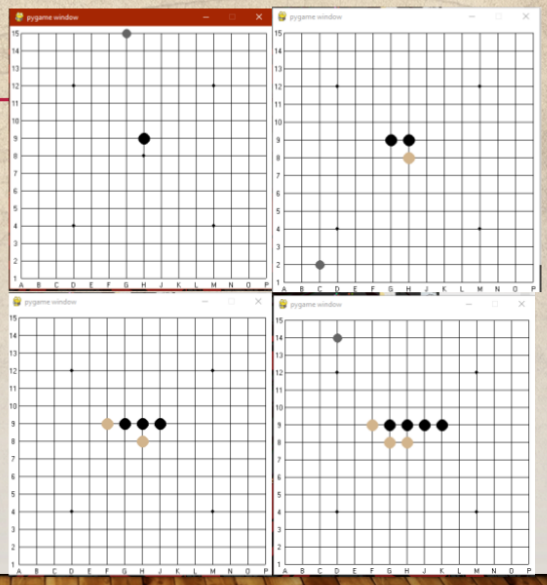
```
def get_diagonalScore(board, xpos, ypos):
        diagonal = numpy.diagonal(board, ypos-xpos)
        length = len(diagonal)
        max_Score = 0
        score = 0
        for i in range(length):
                for j in range( min(length-i, 5) ):
                        #print("Slide Diagonal")
                        if diagonal[i+j] == ai_turn:
                                #print("Diagonal: ", i, j)
                                score += 1
                if max_Score < score:
                        max_Score = score
                score = 0
        return max_Score
```

```
def get_antiDiagonalScore(board, xpos, ypos):
        anti_diagonal = numpy.diagonal(numpy.fliplr(board), 14-xpos-ypos)
        length = len(anti_diagonal)
        max_Score = 0
        score = 0
        for i in range(length):
                for j in range( min(length-i, 5) ):
                        #print("Slide Anti-Diagonal")
                        if anti_diagonal[i+j] == ai_turn:
                                #print("Anti-Diagonal: ", i, j)
                                score += 1
                if max_Score < score:
                        max_Score = score
                score = 0
        return max_Score
```

```
def get_verticalScore(board, xpos, ypos):
        umax = max(0, ypos-4)
        dmax = min(14,ypos+4)
        max_Score = 0
        score = 0
        for i in range(umax, dmax-4):
                for j in range(5):
                        #print("Slide Vertical")
                        if board[xpos][i+j] == ai_turn:
                                #print(i+j, xpos)
                                score += 1 + (2 / (abs(i+j-ypos) + 1))/20
                if max_Score < score:
                        max_Score = score
                score = 0
        return max_Score
```

# DATA

- Here's a short sample of moves, with Black as AI, and White as Player.



## DATA ANALYSIS

- The AI shows ability to determine generally good moves and to implement them.
- It is definitely limited by hardware, unable to think very far ahead, especially in terms of reacting to the player.
- The Score Function does not perfectly reward certain moves, especially in response to Player actions, making the AI value personal score more than preventing Player score.

## Future Work

- The AI's decision time could be improved but this could also be a hardware limitation
- The decision tree could be implemented in a one-time only fashion if we had a powerful enough computer to start, allowing the AI to simply choose the best decisions based on inputs, instead of generating decisions every turn.
  - This would also allow for a more intelligent agent, since the depth is limited due to hardware concerns.
- Implementation of difficulty levels (easy, medium, hard) when playing against the AI