Connect 4: Artificial Intelligence

By Edward Cardenas, Erika Dickson, Klarissa Navarro

1. Problem Statement

Artificial intelligence (AI) is often utilized and tested in a variety of applications, most notably within game play and strategization. AI has experienced a tremendous amount of growth in different sectors in recent years, however, since its inception AI has consistently been utilized within game contexts for the furtherment of research in artificial intelligence and related subsets (Yannakakis). Games pose an interesting and potentially complex setting for AI models to train and test against, whilst also providing an easily interpretable benchmark for efficiency and comparisons (Yannakakis). One such game setting is Connect 4, a traditional board game in which players compete to stack their game pieces in a winning orientation through alternating turns. By utilizing a Minimax algorithm with Alpha-Beta Pruning to create an AI agent as a Connect 4 player, there presents an opportunity to research, build, and evaluate the proficiency of an AI model at various degrees of depth-limited search. The AI model was built to maximize an opportunity for victory through determining game routes at varying depth limits to discover the best possible utility value at that level. Given that Connect 4 has been a solved strategy game for many years, preexisting Connect 4 AI agents have been designed and implemented in a variety of manners, with many being unbeatable models. Thus, the goal of venturing into this field is to research the performance and effectiveness of a manually implemented AI agent in a game of Connect 4 and the extent of which it can compete with human adversaries and unbeatable Connect 4 AI opponents.

2. Approach

The approach taken to this project involves a four-faceted system. The procedure taken to this undertaking began with the research of existing similar projects. Many previous projects individually addressed separate aspects of our project, such as the graphical user interface (GUI) or the Minimax implementation with Alpha-Beta Pruning. Various sources were collected and synthesized into a roadmap from which each of the following stages of project implementation could draw inspiration or be built upon.

2.1 Research

The project combined insights from various resources to create a unique and functional Connect 4 game with AI capabilities. The GUI was implemented using PyGame, guided by a Youtube Tutorial for PyGame basics and another Youtube video that provided the structure for a basic Connect 4 game without AI. Additional UI elements such as fonts, mouse events, token rendering, and text display were used in order to enhance the visuals of the project using PyGame features. The AI implementation leveraged the Minimax algorithm with Alpha-Beta Pruning, based on an article from Robotics Project. To enhance the evaluation function, unique features like double threats and different depths for difficulty levels were added, improving gameplay. By utilizing these sources with custom features, the project successfully delivered a competitive AI and user-friendly Connect 4 experience.

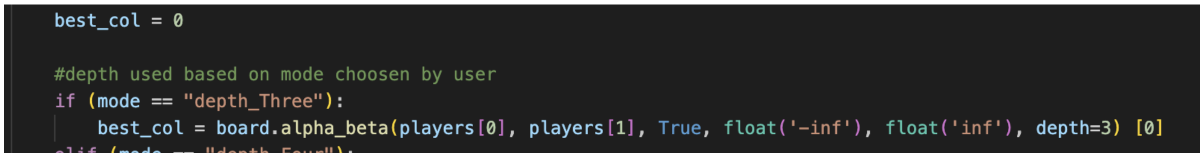
2.2 GUI Implementation

The second step of the program approach was the frontend, which was done through the use of PyGame to make an interactive, visually appealing graphical interface that mimics a traditional physical Connect 4 board. This allows the user to more easily use the game in a classic manner, such as physically dropping the discs into the intended slots, with color coordinated game pieces to distinguish the players' choices. This frontend implementation was chosen for a seamless user experience that eliminates the need for any special gameplay instructions to be communicated to the user beforehand.

2.3 AI Implementation

The AI agent was implemented through the Minimax algorithm with Alpha-Beta Pruning and an evaluation function into the `Board` class. The algorithm evaluates the state of the boards and assigns utility value with the `score_position()` and `evaluate_window()` functions. Here's how it was implemented:

1. **AI's Turn:**
    a. As shown in "Fig. 1," when it's the AI's turn, it calls the `alpha_beta()` function
    b. The function explores possible moves using a recursive approach, alternating between the maximizing player (AI) and the minimizing player (human) to determine what is the optimal column to place their token

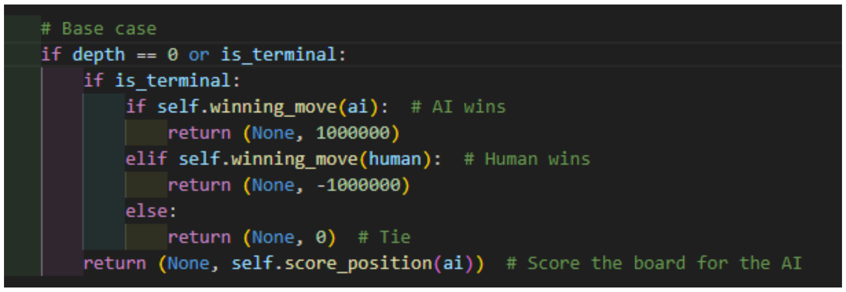


```
best_col = 0

#depth used based on mode choosen by user
if (mode == "depth_Three"):
    best_col = board.alpha_beta(players[0], players[1], True, float('-inf'), float('inf'), depth=3) [0]
```

Fig. 1

2. **Alpha-Beta:**
    a. The `alpha_beta()` function is our Minimax algorithm with Alpha-Beta Pruning
    b. This is a recursive function divided into three parts: 1) base case, 2) the Maximizer, and 3) the Minimizer
3. **Base Case:**
    a. As shown in "Fig. 2," The recursion ends when it reaches a specified depth (chosen by the user) or encounters a terminal state:
        i. **Win**: utility value of 1,000,000
        ii. **Loss**: utility value of -1,000,000
        iii. **Tie**: utility value of 0

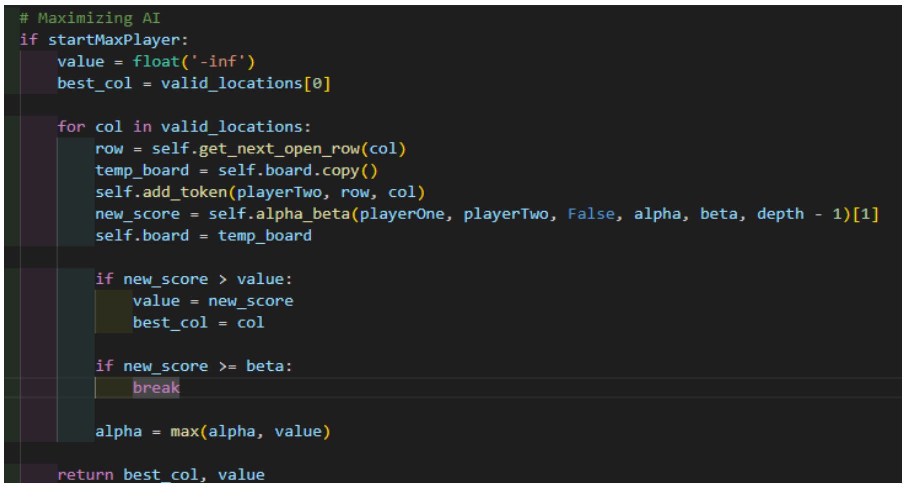
b. If none of these conditions are met, the board's state is scored using the `score_position()` function:
   i. `score_position()` evaluates the tokens and empty spaces for both players
   ii. It then calls `evaluate_window()` to assign utility values based on the number of discs in a row it found for both the player and itself (AI)



```
# Base case
if depth == 0 or is_terminal:
    if is_terminal:
        if self.winning_move(ai):  # AI wins
            return (None, 1000000)
        elif self.winning_move(human):  # Human wins
            return (None, -1000000)
        else:
            return (None, 0)  # Tie
    return (None, self.score_position(ai))  # Score the board for the AI
```

Fig. 2

4. **Recursive call:**
   a. The Maximizer simulates AI's moves (in Fig. 3)
      i. A temporary board is created to simulate AI's token being placed on various columns of the *available* rows
      ii. The function recursively calls itself to simulate the minimizer's response
      iii. After evaluating a move, an evaluated score is returned. Then, perform the following steps using the evaluated score:
         1. If the evaluated score is greater than the current value, update value
         2. If the evaluated score is greater than or equal to beta, prune.
         3. If the evaluated score is greater than the current alpha, update alpha



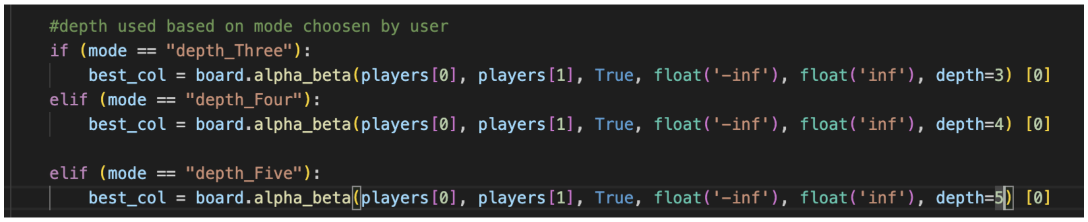
```
# Maximizing AI
if startMaxPlayer:
    value = float('-inf')
    best_col = valid_locations[0]

    for col in valid_locations:
        row = self.get_next_open_row(col)
        temp_board = self.board.copy()
        self.add_token(playerTwo, row, col)
        new_score = self.alpha_beta(playerOne, playerTwo, False, alpha, beta, depth - 1)[1]
        self.board = temp_board

        if new_score > value:
            value = new_score
            best_col = col

        if new_score >= beta:
            break

        alpha = max(alpha, value)

    return best_col, value
```

Fig. 3

   b. The Minimizer simulates the (human) player's moves

i. It mirrors the Maximizers logic *except* it aims to minimize the score, based on the following steps when the evaluated score is returned:
   1. If the evaluated score is less than the current value, update value
   2. If the evaluated score is less than or equal to alpha, prune
   3. If the evaluated score is less than the current beta, update beta

5. **Depth:**
   a. Depending on the choice of the user, a certain depth is passed to the AI (“Fig. 4”), which then evaluates possible moves for each turn to a certain depth of 3, 4 or 5.



```
#depth used based on mode choosen by user
if (mode == "depth_Three"):
    best_col = board.alpha_beta(players[0], players[1], True, float('-inf'), float('inf'), depth=3) [0]
elif (mode == "depth_Four"):
    best_col = board.alpha_beta(players[0], players[1], True, float('-inf'), float('inf'), depth=4) [0]

elif (mode == "depth_Five"):
    best_col = board.alpha_beta(players[0], players[1], True, float('-inf'), float('inf'), depth=5) [0]
```

Fig. 4

2.4 Testing/Evaluation

The last aspect of the assignment to be completed was testing and evaluation. Two main methods were employed in the evaluation process: 1) run the gameplay trials against the AI at various depths with both human and AI competitors, and 2) measure the time and space resources utilized by the AI agent at various depths. Approximately 198 gameplay trials of a human competitor were conducted against the agent, which broke down to 66 trials at each depth, in which various general strategies/approaches were utilized against the AI at various depths in order to have a roughly uniform gameplay approach for each model at differing levels. In doing so, we were able to evaluate the performance of the AI by a metric of victories, ties, and losses. It was discovered that the best performing AI version was the AI of depth five, and the worst performing AI variety was the AI agent at depth three. AI vs AI tests were also conducted against our AI agent, in which an outside Connect 4 AI was utilized to test the efficiency of our implementation. However, due to deviations in the respective board configurations after a certain point, it was no longer an appropriate test of competitiveness and results were deemed inconclusive. Additionally, various tests were run to determine the AI’s efficiency in terms of runtime and the number of nodes explored during its depth-limited search. The goal of this analysis was to compare the average runtimes and number of nodes explored during each AI agents’ turn to determine the extent of the tradeoff between their performance and their resource utilization.

3. Innovation:

Despite pulling inspiration and source code from various preexisting Connect 4 AI programs, our project exemplifies an innovative approach to a long-solved issue through a variety of novel implementations. One such unorthodox integration includes allowing the user to compete with AI agents of different depths in depth-limited search. This choice significantly impacts the game outcome and performance of the AI, which the user is able to directly witness as the correlation between AI performance and the depths to which the agent searches during its search is made apparent by the degree of sophistication of the AI's moves and the possibility of human/competitor victory diminishes with each increasing depth. In addition, a secondary method of including an innovative factor is implementing a double threat check within the evaluation function to determine the degree of which the board state is configured for opponent or agent win based on the number of threats present.

4. Description of the Software

4.1 Frontend, GUI, and AI algorithm

For our software implementation, we used Python along with libraries, such as PyGame, NumPy, math, and sys, to build the following components of our software:

- **Frontend**: provides a visual representation of the game board
  - Using PyGame, we were able to provide the visual components, such as the board and disc of each player for Connect 4. We also provided the human player the different mode options that they can choose to play.
  - Users can also see which player won the game, or if a tie occurred.
- **GUI**: allows the human player to interact with the graphical components created by the frontend
  - PyGame also allows the human player to click on any column of the board given that the column isn't full to place a token.
  - For every token placed on the board by either player, the NumPy array gets updated to reflect the players' positions. In the array, each index is displayed by the players' ID or a 0 (for empty cell).
- **AI algorithm**: drives the gameplay of the AI player
  - Using the NumPy array that reflected the state of the game board, we use our `alpha_beta()` function to help the AI find the optimal position to place their token in one of the columns
  - Then, using PyGame, the AI's disc will be drawn on the game board for the human player to see. Also, the AI's move would be added in the NumPy array in the respective position of the optimal move discovered by the AI algorithm.

No datasets were required for our problem.

4.2 Structure of Code and Classes

Our code includes three key components, each with a role

1. *main.py*
   a. **Role:** Manages the game flow, handles user interactions and serves as the coordinator of the entire program
   b. **What it does:**

        i. Initializes the PyGame GUI and manages the welcome screen
        ii. Creates the instances of `Player` and `Board` class
        iii. Alternates turns between players
        iv. Handles human player input and calls AI algorithm when its AI's turn

2. *player.py*
    a. **Role:** Represents a game participant, human or AI
    b. **What it does:**
        i. Identifies the player
        ii. Assigns the token color
    c. **Functions:**
        i. get_id()
        ii. get_color()
        iii. is_ai()
3. *board_game.py*
    a. **Role:** Represents the Connect 4 game board, manages game logic, and Minimax algorithm with Alpha-Beta Pruning with evaluation function
    b. **What it does:**
        i. Creates a 2D Numpy array that represents the current state of the board
        ii. It has functions that represent the game logic such as drawing the board, placing a token, checking for valid locations, and checks for winning moves.
        iii. Contains the AI algorithm to help it make the best decision to place token
        iv. Has a counter to check for how many nodes are checked.
    c. **Functions:**
        i. add_token()
        ii. is_valid_location()
        iii. get_next_open_row()
        iv. winning_move()
        v. draw_board()
        vi. valid_location()
        vii. alpha_beta()
        viii. score_position()
        ix. evaluate_window()
        x. is_terminal()
        xi. reset_counter()

5. Evaluation

5.1 Runtime and Number Counter Test Implementation:

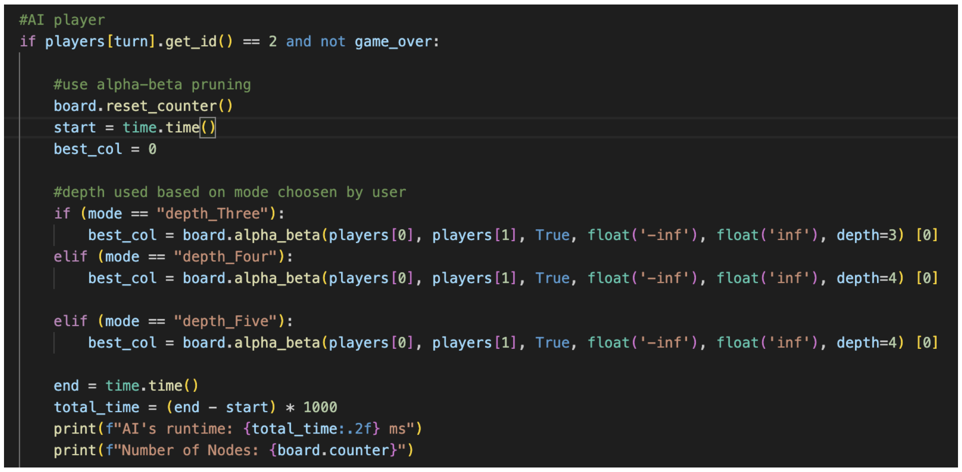
The depth of the Minimax algorithm's search tree has a significant impact on both the runtime and the number of nodes explored. As depth increases, the number of possible board states grows exponentially due to the algorithm's time complexity of $O(b^d)$, where 'b' is the branching factor and 'd' is the depth. Alpha-Beta Pruning improves efficiency by eliminating unnecessary branches, potentially reducing the time complexity to O(b^(d/2) in the best-case scenario, effectively halving the search depth. However, even with Alpha-Beta Pruning, increasing the depth still leads to higher computational demands, especially in the early game when branching factors are largest. Thus selecting an appropriate depth is critical to balancing decision quality and computational efficiency.

To better understand and test the performance of our AI and how depth can affect its efficiency we performed these test:

1. **Runtime**: How long does the algorithm take to make a decision at each turn?
2. **Node Count**: How many nodes the algorithm explores during each turn?

5.2 How We Implemented It:

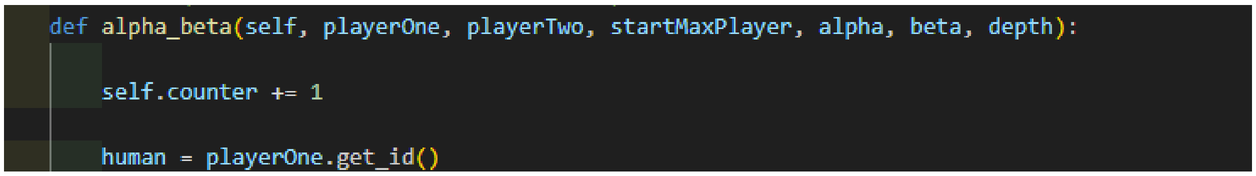
1. **Runtime Test Setup** :
    a. We imported the `time` library and added timing functionality around the algorithm calls in the `main()` function.
    b. For each AI move, we recorded the start and end times using `time.time()` and calculated the total run (in "Fig. 5")



```
#AI player
if players[turn].get_id() == 2 and not game_over:

    #use alpha-beta pruning
    board.reset_counter()
    start = time.time()
    best_col = 0

    #depth used based on mode choosen by user
    if (mode == "depth_Three"):
        best_col = board.alpha_beta(players[0], players[1], True, float('-inf'), float('inf'), depth=3) [0]
    elif (mode == "depth_Four"):
        best_col = board.alpha_beta(players[0], players[1], True, float('-inf'), float('inf'), depth=4) [0]

    elif (mode == "depth_Five"):
        best_col = board.alpha_beta(players[0], players[1], True, float('-inf'), float('inf'), depth=4) [0]

    end = time.time()
    total_time = (end - start) * 1000
    print(f"AI's runtime: {total_time:.2f} ms")
    print(f"Number of Nodes: {board.counter}")
```

Fig. 5

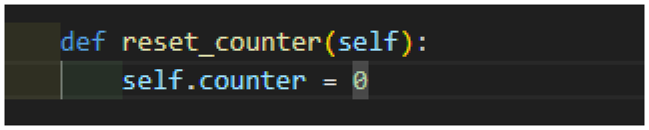
2. **Node Counter Setup:**
    a. A counter attribute was added to the `Board` class to track how many nodes the algorithm explored during each turn.
    b. The counter was incremented every time the `alpha_beta()` function was called since it recursively called itself when it entered in a new node (in "Fig. 6")



```
def alpha_beta(self, playerOne, playerTwo, startMaxPlayer, alpha, beta, depth):

    self.counter += 1

    human = playerOne.get_id()
```

Fig. 6

c. To ensure accurate counts, the counter was reset before each AI move using the `reset_counter()` function (in "Fig. 7")



```
def reset_counter(self):
    self.counter = 0
```

Fig. 7

5.3 Analysis of Results:

Link to google sheets with results:
https://docs.google.com/spreadsheets/d/1hVxx-eJ_0a1GkH__5h2B4gAOFoxV5CoWvfc_1r3cjfc/edit?usp=sharing

5.3.1 Depth 3

- **Average Runtime**: 19.48 ms to 147.67 ms
- **Average Nodes Explored**: 119.89 to 162.83
- **Results**: The algorithm performed efficiently at this depth, with relatively quick decision times. Early turns showed higher runtimes and node counts due to larger branching factors, while later turns saw a significant drop as the board became more constrained and Alpha-Beta Pruning reduced the number of nodes evaluated.

5.3.2 Depth 4

- **Average Runtime**: 53.32 to 127.96
- **Average Nodes Explored**: 385.24 to 907.57
- **Results**: The computational cost increased significantly compared to Depth 3 due to the exponential growth in the search tree b^4. However, Alpha-Beta Pruning effectively reduced unnecessary calculations, especially in mid to late game turns where the number of valid moves was lower. This resulted in a noticeable decrease in runtime and number of nodes explored during this depth.

5.3.3 Depth 5

- **Average Runtime**: 198.56 to 281.07
- **Average Nodes Explored**: 1399.06 to 2012.75
- **Result**: This depth signified the exponential growth of the Minimax algorithm, with early turns requiring the exploration of thousands of nodes. However, as the game progressed,

pruning became more effective, and the runtime and nodes dropped significantly. Despite this, some mid-game turns showed spikes in runtime and nodes due to less effective pruning depending on the board state.

The analysis showed that increasing the depth of the Minimax algorithm improved the AI's decision making quality but required higher computational resources. At Depth 3, the AI performed efficiently, maintaining low runtimes and node counts, but its strategic effectiveness was limited. Depth 4 balanced computational efficiency with drastically improved strategic play, making it a practical option for most scenarios. At Depth 5, the AI showed the highest strategic performance, winning nearly all games, but the computational demands were much higher, particularly in early turns with large branching factors. These results highlight the trade-offs between depth, computational cost, and decision making quality, emphasizing the importance of Alpha-Beta pruning in managing resource usage while maintaining strong gameplay across all depths.

5.4 Human and Outside AI Agent Gameplay Test Trials

An additional means of testing and evaluation conducted against the Connect 4 AI agent we built was running gameplay trials against the model by human and outside AI agent competitors. Approximately 198 trials were executed against the AI agent by a human competitor, in which all depths (three through five) of the AI model were subjected to 66 trials to determine the statistics of wins, losses, and ties to measure the overall performance of the model as it pertains to winning. Certain strategies, such as attempting to play defensively/offensively or trying to stack discs in a certain configuration throughout the game, were employed in order to provide a general standardization of gameplay against the AI to test its reaction and the effectiveness of its response to these moves at different depths. The outcome of these tests reveal interesting results:

	Depth = 3	**Depth = 4**	**Depth = 5**
Winner	34 (~52%)	64 (~97%)	64 (~97%)
Loser	31 (~47%)	2 (~3%)	0
Tie	1 (~1%)	0	2 (~3%)
Total Games	66	66	66

Fig. 8

From the trials, it was determined that the AI had the best performance at a search depth of five, with roughly 97% wins, 3% ties, and zero losses. At a close second place, the AI agent operating at depth 4 also had 97% wins, however, it was determined that the AI at depth five was a better performer because it experienced no losses and games not won resulted in only ties, whereas the AI agent at depth 4 did experience losses. The worst performer was the AI at a depth of three, which only won approximately 52% of its games, had 47% of its games result in losses, and had a singular tie.

Once trials against human competitors were completed, we sought to test our AI agent against a preexisting Connect 4 AI agent to measure its competitive ability through examining its responses and game outcomes in 20 gameplay runs. An online version of Connect 4 with an AI player was utilized as an opponent for our AI agent at depth five. Trials were conducted by manually placing the discs in the positions each AI selected on their respective boards during their respective turns. However, this experiment proved inconclusive because at a certain point, the AI agents would place a disc in a position that would alter the game state to no longer be consistent on both AIs' boards, thus it was not possible to evaluate the outcomes as if they were competing against each other because the changes would only propagate further as the games continued, ultimately resulting in the AI agents playing different games. Despite this limitation, useful analytical data for comparisons of our AI against other AI implementations was gathered. We discovered that although the AI agent at a depth of five won each game, it did not always select the optimal route toward victory and would occasionally make redundant choices that would delay a win. In comparison, the outside AI agent was comparatively quicker at finding a solution to the game, with respect to the number of moves.

6. Conclusion

With the advent of artificial intelligence being utilized in new, dextrous, and innovative ways, it raises the question of how these models can be trained and tested. Games have been historically utilized by AI researchers for a number of reasons, including, but not limited to, the understandability presented by game contexts, the ease of evaluation and analysis, and the fact that real world scenarios can often be modelled in game systems. It is for these reasons that we decided to utilize a game setting, Connect 4, to implement an AI model. Our methodology included four steps: researching preexisting similar projects from which we could pull inspiration from, building the GUI through PyGame, implementing the AI agent using the Minimax algorithm with Alpha-Beta Pruning at different levels of depth limited search, as well as analyzing and evaluating the model through human and outside AI competitors. In concluding this process, we discovered that our best performing AI agent version was at a depth of five, and though it was not optimal compared to preexisting Connect 4 AI implementations, it was a proficient competitor that won the vast majority of its games against human opponents without losses.

The main lessons gathered from this undertaking include an improved understanding of the significance of depth in game trees. This is illustrated in our evaluation of the AI's ability to win against human users at different depths, and how the statistics are vastly different between the different levels. Also, we analyzed the AI's efficiency by monitoring the runtime and the number of nodes explored at each turn, which highlighted the relationship between the differences in depth and the performance of the AI with respect to the average runtimes and the number of nodes explored. Additionally, a lesson we took from this project is the complexity of designing effective evaluation functions. These functions require considering multiple factors and establishing appropriate weights for different circumstances within the game state. This can be challenging since it requires assessing a wide range of scenarios to guide the AI agent in choosing the right strategies. Our endeavour into the study of artificial intelligence through an implementation of a Connect 4 AI agent has been incredibly informative in multiple fronts that help compose the incredibly dynamic field of artificial intelligence.

In attempting to improve this project in the future, one significant change that would enhance the performance of the AI agent is the tweaking of the evaluation function in order to allow the AI to consistently choose and prioritize the optimal path to victory. At present, the AI will occasionally postpone a win in favor of pursuing defense, which causes the AI agent to commit more moves to reach a victory than necessary, which reduces its efficiency. By extension, in utilizing more turns to reach a solution, the AI agent additionally requires more computation and time in order to process more turns, which also impacts its efficiency in resource utilization.

7. References

"Connect 4 Algorithm." *Connect 4 Algorithm - Connect Four Robot Documentation*, roboticsproject.readthedocs.io/en/latest/ConnectFourAlgorithm.html.

Galli, Keith. "How to Program Connect 4 in Python! (Part 1) - Basic Structure & Game Loop." *YouTube*, YouTube, Dec. 2017, www.youtube.com/watch?v=UYgyRArKDEs&t=0s.

"How Do I Measure Elapsed Time in Python?" *Stack Overflow*, stackoverflow.com/questions/7370801/how-do-i-measure-elapsed-time-in-python.

Yannakakis, Georgios. 2022. AI and Games: The Virtuous Cycle. In Proceedings of the 12th Hellenic Conference on Artificial Intelligence (SETN '22). Association for Computing Machinery, New York, NY, USA, Article 3, 1. https://doi-org.lib-proxy.fullerton.edu/10.1145/3549737.3549744

"How to Display Text in Pygame?" *Stack Overflow*, stackoverflow.com/questions/20842801/how-to-display-text-in-pygame.

Jayanam. "Python and Pygame Game Development Tutorial : Connect 4." *YouTube*, YouTube, Apr. 2019, www.youtube.com/watch?v=iWnbEvEAc5I.

“Pygame - Drawing Shapes.” *Tutorialspoint*, www.tutorialspoint.com/pygame/pygame_drawing_shapes.htm.

“Pygame - Mouse Events.” *Tutorialspoint*, www.tutorialspoint.com/pygame/pygame_mouse_events.htm.

Sanchez, Bryan. “ConnectFour.” *GitHub*, github.com/bryanjsanchez/ConnectFour.

“What Fonts Can I Use with Pygame.Font.Font?” *Stack Overflow*, stackoverflow.com/questions/38001898/what-fonts-can-i-use-with-pygame-font-font.