A Comparative Analysis of Connect 4 AI Agents

Overview

Great strides have been made in the field of artificial intelligence, especially in recent years. With AI agents becoming more advanced and efficient at learning, game-playing AI agents have been a popular area for research as they can be developed and analyzed in a structured environment. They also tap into the creative, fun side of computer science, making them a great introduction for someone wanting to learn more about AI models and their implementations. The problem I aim to tackle in this report is creating an AI agent for Connect 4, a classic two-player connection board game. I will explore the design and implementation of various AI agents as well as evaluate their effectiveness against humans. Additionally, I will detail how well the agents perform against each other. My goal is to create an agent with a 70% confidence level that it can beat a rational human player with average skill at the game.

Initial Progress: Basic AI Agent

In my endeavor to create an AI agent, I explored multiple avenues of implementation, some we had learned in class and some I learned independently. I began by first coding a simple version of the game using Python which the user can play using the command line. The game showed a Connect-4 board and prompted the first player to select a column to place their piece. Then, it would update the board and ask for the next player's input, ending when a player got four pieces in a row in any direction (horizontal, vertical or diagonal). From there, I began by using a similar tactic to how AI agents worked in Pacman. The program used a heuristic function to guide characters on the maze, allowing them to make informed decisions by processing data about locations of different items and obstacles. Similarly, I created a heuristic function for the AI agent which assigns a score to different possible board states by factoring in the position of

pieces on the board. The heuristic function shown below shows an example of how factors were considered. The function took in an arrangement of a board and the piece that the AI was playing as. In this snippet, the first section of code evaluates the number of pieces there are in the center column. Placing pieces in the center maximizes the possibilities of making a sequence of four as there is more space in every direction to expand. The second block of code evaluates horizontal arrangements in the board by calling a method named windowEval. This function adds scores depending on the number of possibilities of a winning sequence in the board horizontally. Here is a list of how situations are scored:

```
# heuristic function for minimax
def scorePos(board, piece):
    score = 0

    # center column score
    centerArr = [int(i) for i in list(board[:, COLUMNS//2])]
    centerCount = centerArr.count(piece)
    score += centerCount * 3

    # horizontal score
    for r in range(ROWS):
        rowArr = [int(i) for i in list(board[r, :])]
        for c in range(COLUMNS - 3):
            window = rowArr[c : c + WINDOW_LENGTH]
            score += windowEval(window, piece)
```

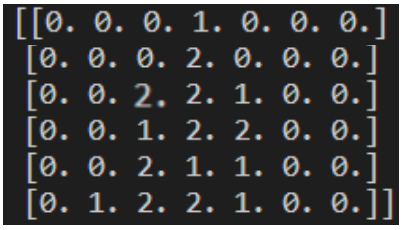
- A piece in the center column: +3
- A window of four slots with four connected pieces: +100
- A window of four slots with three pieces and a free slot: +5
- A window of four slots with two pieces and two free slots: +2
- A window of four slots with three opponent pieces and a free slot: -50
- A window of four slots with two opponent pieces and two free slots: -6

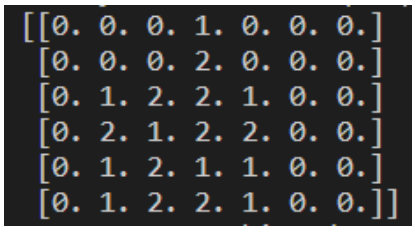
It should be noted that these scores are all arbitrary and a result of me trying different values until I found one that worked well. After some games, it was clear that the agent had made progress and was making somewhat informed decisions. It started to block any slots in which I would have an obvious game winning move, and it also began to try to create sequences with its own pieces. The results of me playing 10 games against the agent are listed in the table below.

Agent Game Result	L	L	W	L	L	L	W	L	W	L
First player	H	H	H	H	H	A	A	A	A	A
Moves	13	31	20	27	23	22	35	26	25	24

H = Human, A = Agent, L = Loss, W = Win

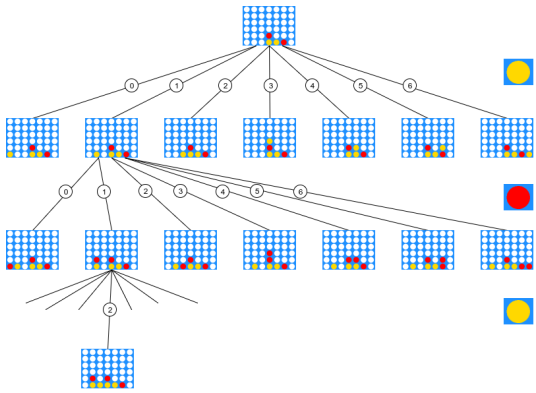
The agent had a win rate of 20% when I played first, but it won at a 40% rate when it was allowed to go first. The average number of moves when the agent won was 26.7 compared to 23.7 when it lost. While playing these games there were some details I noticed about the agent's playstyle and tendencies. When I went first, I found that it was relatively easy to win a game. In one of my two losses, the agent did create a scenario in which it was winning with two different sequences that I could not block, however this did not repeat in other trials so it is likely a coincidence. The agent performed much better when given the chance to play first, as is the case with most players. However, a tactic that proved to be effective was to set up my moves by anticipating what the agent was going to play next. For example, when given the position on the right, I placed my piece in the second column from the left. I could predict that the agent would then place its piece in the same column as it valued the diagonal made by three of its pieces; it would not be able to predict that I could win in the next turn (shown below). This is a disadvantage of having an agent that can only evaluate current positions without any knowledge of the future. Based on the moves data, it can also be noted that the agent performed much better when the game went on longer. This might be because keeping track of the complexity of pieces in a single turn is easy for a computerized agent, while a human may make mistakes.





Second Implementation: Minimax

While this version of my agent had shown promising results, I seeked to create a more complex version of the AI and utilized an algorithm used extensively in game theory: Minimax. I created a recursive function to call the minimax algorithm and used the same heuristic function as the previous algorithm to evaluate a score value of any board position. This score value guided the algorithm to make decisions. The diagram below is a simple representation of how the



minimax function works. It creates a tree of possible moves from a given position and the number of nodes increases exponentially as the depth increases. It then traverses the nodes in a depth-first manner, recursively exploring each branch until it reaches a terminal node or the depth limit. The algorithm aims to find the move that increases its own utility while decreasing the opponent's utility, assuming that the opponent plays optimally. Within this implementation, I also added alpha beta pruning: a method in which moves that are deemed to be evidently detrimental are removed from the tree. This decreased the time and space complexity of the algorithm by allowing it to process less nodes: at depth 6, each move had tens of thousands less nodes to process. With this implementation, I was able to consistently play the game with the algorithm having a depth of six. Any higher would result in long, inconsistent wait times. After successfully integrating minimax and alpha-beta pruning, the next step I took was to incorporate iterative deepening to further optimize the agent's performance. This technique allows the agent to explore the game tree at increasing depths, allowing for a more flexible approach to time management and further reducing the computational cost of the search. Now, instead of limiting the depth at which my

algorithm ran, I limited the time that it had to process a move to 5 seconds. The algorithm would run as deep as possible within the given timeline to produce the best result. This method was already proving to be much more effective than the previous as it was able to process hundreds of thousands of nodes per move compared to a maximum of 7 per move by the other agent. To evaluate the performance of the AI agent using minimax with alpha beta pruning and iterative deepening, I played a series of games against it once more. The results are shown below.

Agent Game Result	W	D	W	W	L	W	W	W	W	W
First Player	H	H	H	H	H	A	A	A	A	A
Moves	24	42	26	28	29	21	27	27	35	31

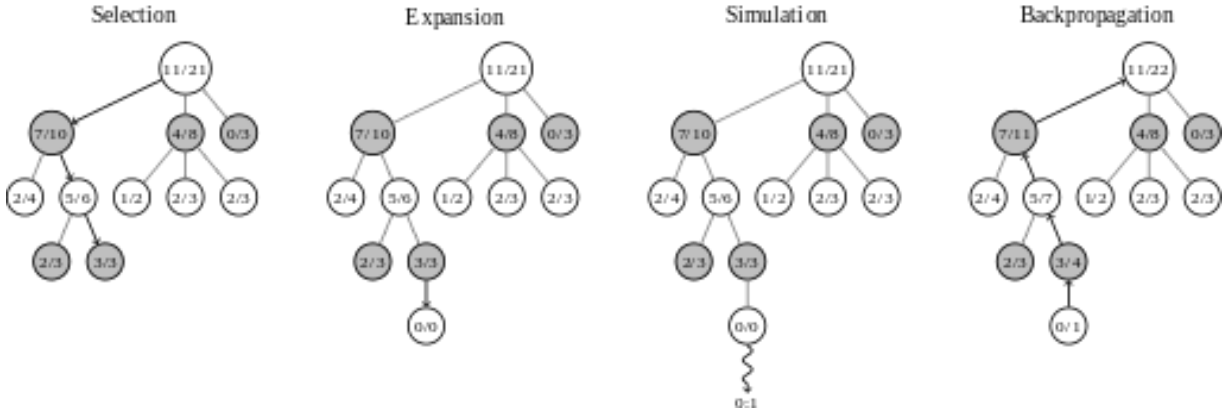
H = Human, A = Agent, L = Loss, W = Win

Out of these ten games played, the AI agent won 8 times with one game ending in a draw and one in a loss. It won 100% of games when it got the chance to play first and 60% of games when it played second. The margin by which the agent consistently defeated a rational human player provides valuable insights into how effective the model truly is. This massive increase in effectiveness against a rational human player is reflective of how much more information this agent has access to while making a decision. The tactics that I employed in the previous matchup against the basic AI did not work as the agent now exhibited a deep understanding of the game. It was making moves that not only aimed at winning, but strategic moves that blocked my attempts to secure a victory. The times in which I managed to draw or defeat the AI were moments when I had to play almost perfectly, thinking several moves ahead and capitalizing on the rare lapses in the agent's decision making. Additionally, this time the agent achieved victories in primarily mid to late game situations as shown by the data (range of 27 to 35 moves); it stuck

through matches and played strategically, not falling for early game mistakes. This can be attributed to the fact that as the game goes on, the number of total nodes that can be analyzed decreases and the game has a more complete understanding of which move to take. From this data, I can conclude that I achieved my goal of creating an agent with a 70% confidence level that it can beat a rational human player with average skill at the game as it defeated me 80% of the time. Furthermore, when I played my first implementation of an AI model against this version of the minimax, there was a consistent result every time: games started by the first model ended in 36 moves and games started by minimax ended in 17 moves. Each time, the minimax model won.

Third Implementation: Monte Carlo Tree Search

In my search for an effective Connect 4 AI algorithm, I wished to create an algorithm to rival this version of the minimax using a different methodology. Thus, I decided to employ another effective algorithm used in game theory known as the Monte Carlo Tree Search (MCTS) algorithm. This algorithm relies more on reinforcement learning principles by creating random simulations and conducting a statistical analysis rather than a deterministic evaluation function. What makes this algorithm more flexible is that there is no need for a heuristic function in order to implement this model. It consists of four main steps performed iteratively as shown below.



Starting from a root node, in this case the starting position of the board, the algorithm traverses the nodes of the game tree following the upper confidence bound for trees (UCT) algorithm until it reaches an unexpanded node. Then, it expands this node by adding one or more child nodes. A random simulation is performed with the newly added child node as the root until a terminal node is reached (a win or a loss). The result of this simulation is then propagated back up the game tree, updating the values of win rate and visit count of each of the nodes in the path. Similar to iterative deepening, I employed this algorithm using a 5 second time limit, after which the AI agent would select the move with the highest win rate. While this algorithm is efficient, it has some drawbacks. For example, it performs poorly when there is a need for long-term planning as it may not be able to explore as many states as minimax. This is because it needs to reach a terminal node each time in order to evaluate an outcome and backpropagate data. It is only able to traverse a few thousand nodes per move while the minimax algorithm processes hundreds of thousands of nodes. Once again, to evaluate the algorithm's performance, I played a series of games against it.

Agent Game Result	W	W	L	L	W	W	D	W	W	W
First Player	H	H	H	H	H	A	A	A	A	A
Moves	22	28	31	26	30	21	24	29	23	25

H = Human, A = Agent, L = Loss, W = Win

This algorithm has produced results almost identical to the minimax algorithm, showcasing that MCTS is just as formidable as minimax. It has dominated to almost the same degree by winning 70% of matches and only losing two while drawing the other. As previous AI research has noted, this algorithm closely converges to minimax over time and the results obtained from

experimentation support this finding so far. The manner in which the algorithm played was similar to minimax as well, as it showed a clear understanding of strategy. The algorithm was also more consistent in the amount of moves it took to win a game (standard deviation of 3.60 moves in games won compared to 4.24 moves with minimax). Additionally, unlike the other two models, when given a starting board, the best move determined by Monte Carlo may be different on each iteration due to random iteration. Theoretically, this would also make it harder to predict what move the algorithm would make. Yet, when pitting MCTS against minimax with alpha beta pruning and iterative deepening, the results are interesting.

Winner	MI	MI	MI	MI	MI	MI	MI	MI	MC	MC
First Player	MI	MI	MI	MI	MI	MC	MC	MC	MC	MC
Moves	21	37	39	37	37	38	42	38	39	27

MI = Minimax, MC = Monte Carlo

It seems as if the difference between Monte Carlo and Minimax's effectiveness seems small when facing a human opponent, but when facing each other, minimax consistently outperformed MCTS. This clear difference can be attributed to a number of differing factors between the two models. One factor is efficiency as the minimax algorithm, with alpha-beta pruning and iterative deepening, may have been more efficient in traversing the game tree compared to MCTS. This would lead to an optimized, higher quality sample of data from which the algorithm can make a decision. Another factor may be that the heuristic function used in the minimax algorithm might be extremely accurate in guiding the agent to evaluate situations. This is compared to MCTS, which relies on random simulations and may not always converge to an optimal solution with a limited amount of time. MCTS only processes a few thousands nodes per move compared to minimax which can process hundreds of thousands in the same time frame.

The effectiveness of these algorithms is also dependent on the characteristics of Connect 4 itself. Since it is a solved game, the deterministic nature of the game seems to favor a deterministic algorithm such as minimax which seeks to maximize utility. While MCTS is effective, its reliance on randomness may be less effective in a game with a set perfect strategy. These results suggest that minimax with alpha-beta pruning and iterative deepening is one of the most effective algorithms for Connect 4, however this outcome is taken from a small sample and is not compared to other AI models. In regards to MCTS converging to minimax over time, the differences between algorithm performance can also be attributed to the time limit given to each algorithm. It is likely that in such a scenario, MCTS requires more time to traverse nodes in order to select moves more effectively. Further research and experimentation can help refine these AI agents and uncover new strategies to improve their performance in different gaming scenarios.

This comparison between various AI models achieves the highest milestone in the grade contract as it compares the implementations of MCTS and minimax with alpha-beta pruning and iterative deepening against an average player as well as against each other. Since this was an individual project, I coded the entirety of the program, the game itself and the algorithms, I gathered all of the data and I wrote the entire report. I referred to various online sources throughout my work on this project and I have linked them below along with the github link for the program.

Github Link:

https://github.com/rahildacoder/AI-Final-Project

Sources:

Deep Reinforcement Learning and MCTS with Connect 4

When should MCTS be chosen over MiniMax

Iterative Deepening in Chess Programming

Trappy Minimax

Implementing Minimax