# Model Based AI Assignment: Connect Four Agent

Arne Lescrauwaet (852617312)

October 2024

## 1 Introduction

Connect Four is a classic two-player game that has long intrigued both casual players and researchers due to its blend of simplicity and strategic depth. Despite the straightforward rules, the game offers complex decision-making challenges, making it an ideal candidate for computational analysis and artificial intelligence applications.

Answer Set Programming (ASP) has emerged as a powerful tool for solving combinatorial problems, including game solving. ASP, a form of declarative programming, is particularly well-suited for modeling complex rule-based systems such as Connect Four. Its logic-based nature allows for a clear representation of game rules, board states, and strategies, while its efficient solvers enable the exploration of optimal moves.

In this assignment, I want to explore the use of ASP to develop a Connect Four agent capable of determining the optimal move for any given game state. By encoding the game's rules and strategies in ASP, I aim to create an intelligent agent that not only identifies immediate winning moves but also anticipates and blocks opponent threats. The methodology includes implementing various strategic principles such as center control, parity, and blocking, with each tactic weighted to guide the agent's decision-making process. As an extension of this, I wanted to see if I could make an agent capable of winning against the average human player. This leads us to the two research questions of this assignment:

**RQ1: Can the optimal move be determined based on the current state of the game**
**RQ2: Can my agent defeat the average human player?**

## 2 Background

Connect Four, also known as Four-in-a-Row or Captain's Mistress, is a classic two-player connection game invented by Howard Wexler and Ned Strongin in 1974. The game is traditionally played on a vertical grid that is seven columns wide and six rows tall. Players take turns dropping colored discs into one of the columns, and the discs fall to the lowest available position within the column. The objective is to be the first player to form a horizontal, vertical, or diagonal line of four consecutive discs of their color.

Figure 1: Connect Four

## 3 Methodology

To develop an agent capable of returning the optimal moves based on the current state of the game, we must first model the game environment and then define a strategy for evaluating potential moves.

### 3.1 Components

Modeling Connect Four involves breaking the game down into its fundamental components. The game consists of a board, two players, a set of rules, possible actions, and win conditions.

**The Board:** The game board can be represented as a grid of rows and columns. In my implementation, each player is identified by a unique ID: Player 1 is assigned the ID 1, and Player 2 is assigned the ID 2.

**Rules:** Connect Four is conceptually simple, translating the rules into code requires some attention to detail. The first rule is that chips can only be placed in empty slots. To enforce this, we define what it means for a cell to be "occupied" or "empty." A cell is considered occupied if it contains a chip at specific coordinates. Conversely, a cell is considered empty if it falls within the valid row and column bounds and has no chip.

Listing 1: Empty and Occupied

```
occupied(Row, Col) :- cell(_, Row, Col).

empty(Row, Col) :- row(Row), col(Col), not occupied(
   Row, Col).
```

Another essential rule in the physical game—that chips must either be placed at the bottom of the board or directly on top of another chip—must also be explicitly programmed. Before allowing a move, the system checks whether the bottom-most row (row 1) is selected and empty or if the row directly below the chosen row is occupied and the selected cell is empty.

Listing 2: Valid moves

```
valid_move(1, Col) :- empty(1, Col).

valid_move(Row, Col) :- row(Row), Row > 1, col(Col),
    empty(Row, Col), occupied(Row-1, Col).
```

**Win Condition:** The goal of the game, as implied by its name, is to connect four chips in a row. To determine if a player has won, the system must check four possible patterns: horizontal, vertical, diagonal-up, and diagonal-down sequences. The Clingo code for implementing these win conditions can be found in the appendix under 9 Win Condition.

## 3.2 Strategy

To design a strategy for the agent, I incorporated several tactics gathered from online sources, including articles and YouTube videos [5, 1]. Each tactic was assigned a specific weight, and the program maximizes the sum of these weights to identify the optimal move for the current board state.

**Middle Control:** Controlling the center column offers a strategic advantage to both players. If player 1 begins by placing their chip in the middle column and plays perfectly from that point onward, they are guaranteed to win. Similarly, player 2 can force a draw by doing the same and following with flawless play [4]. This tactic is reflected in the agent's strategy by assigning a higher weight to moves in the middle column, with the weight decreasing as moves are made in columns further from the center.

Listing 3: Center control

```
score(Row, 4, center_weight) :- valid_move(Row, 4).

score(Row, Col, center_weight) :- valid_move(Row, Col)
    , Col = 3; Col = 5.

score(Row, Col, 2) :- valid_move(Row, Col), Col = 2;
    Col = 6.

score(Row, Col, 1) :- valid_move(Row, Col), Col = 1;
    Col = 7.
```

**Parity Strategy:** The parity strategy involves creating potential threats (situations that could lead to a sequence of 4 chips) on specific rows based on the player's turn order. If you play first, the goal is to create threats on odd-numbered rows, while if you play second, the focus shifts to even-numbered rows. To implement this, a weighted score is assigned to moves depending on the row number and the player's ID, encouraging the agent to prioritize these strategic positions.

Listing 4: Parity Strategy

```
odd(1).
odd(Row) :- row(Row), row(Prev_Row), odd(Prev_Row),
    Row = Prev_Row + 2.

score(Row, Col, odd_even_weight) :- valid_move(Row,
    Col), player(1), odd(Row).

score(Row, Col, odd_even_weight) :- valid_move(Row,
    Col), player(2), not odd(Row).
```

**Immediate Win Moves:** Once a threat is established, the algorithm must prioritize securing an immediate win. The first challenge is identifying all possible winning threats. As discussed earlier, there are four types of winning sequences: horizontal, vertical, diagonal-up, and diagonal-down. Each of these can be formed in different ways.

For instance, a horizontal sequence can be achieved by placing three consecutive pieces either to the left or right of the selected column, as modeled in the appendix under 10, Horizontal Consecutive. Another scenario involves a middle gap, where pieces are placed on both the left and right of the selected column and a chip dropped in the current column would complete the four-piece line. This case, along with other possible gap variations, is handled in 11, Horizontal Gap and can be found in the appendix.

The vertical sequence is more straightforward since there is only one way to stack four pieces vertically. In this case, the algorithm only needs to check for three consecutive chips in the same column, as shown below:

Listing 5: Vertical Threat

```
winning_move(Player, Row, Col) :- valid_move(Row, Col)
    ,
                                            cell(Player, Row
                                               -1, Col),
                                            cell(Player, Row
                                               -2, Col),
                                            cell(Player, Row
                                               -3, Col).
```

For diagonal sequences, the algorithm must account for two directions: upward-right and downward-right (as well as their leftward counterparts). The first diagonal variant checks for three consecutive pieces starting from the bottom-left or bottom-right (relative to the current move) and moving upward.

Listing 6: Diagonal Upward

```
winning_move(Player, Row, Col) :- valid_move(Row, Col)
    ,
                                            cell(Player, Row
                                               -1, Col-1),
                                            cell(Player, Row
                                               -2, Col-2),
                                            cell(Player, Row
                                               -3, Col-3).
winning_move(Player, Row, Col) :- valid_move(Row, Col)
    ,
                                            cell(Player, Row
                                               -1, Col+1),
                                            cell(Player, Row
                                               -2, Col+2),
                                            cell(Player, Row
                                               -3, Col+3).
```

Next, the down-right and down-left diagonal sequences are handled similarly, starting from the top-right or top-left (relative to the current move) and moving downward. The code for these variations can be found in the appendix under 12, Diagonal Down.

Lastly, the algorithm also handles diagonal sequences with missing gaps, where a single move can complete a winning sequence despite an interrupted line. The corresponding code for diagonal gap handling can be found in the appendix under 13, Diagonal Gap.

**Blocking:** A critical aspect of the game is preventing the opponent from completing a sequence of four chips, which is done through blocking. This requires placing a chip in the position the opponent would need to win. To implement this, the agent first checks for any potential winning moves from the opponent, as discussed earlier. If such a move is identified and is valid, the agent blocks it by placing its chip in that spot. This process is modeled as follows.

Listing 7: Blocking

```
blocking_move(Row, Col) :- valid_move(Row, Col),
                            player(1),
                            winning_move(2, Row, Col).
blocking_move(Row, Col) :- valid_move(Row, Col),
                            player(2),
                            winning_move(1, Row, Col).
```

## 3.3 Heuristics

Once the challenge of pinpointing threats and opportunities has been addressed, the next step is to assess the value of each potential move. This is achieved by assigning suitable weights to various strategies. The highest priority is given to "immediate win" moves, which directly lead to a victory. Moves that block the opponent's winning opportunities rank second in importance. Instead of merely depending on a knowledge base, I chose an optimization strategy where the agent aims to maximize the score of each move based on the game's current state.

The implementation begins by calculating the total score for each valid move, using the #sum aggregate to combine the weights assigned to various strategies. The #maximize aggregate is then applied to ensure that the move with the highest total score is selected as the best move.

Listing 8: Optimization

```
total_score(Row, Col, Total) :- valid_move(Row, Col),
    Total = #sum{S : score(Row, Col, S)}.

#maximize{Total, Row, Col : total_score(Row, Col,
    Total)}.

best_move(Row, Col) :- total_score(Row, Col, Total),
    Total = #max{T : total_score(_, _, T)}.
```

The weights for each strategy are assigned based on my understanding of the game. Immediate win moves are given the highest weight of 100, ensuring they always take priority. Blocking moves are assigned a weight of 80 to reflect their importance in defense. The center control strategy is weighted at 30, while the parity strategy is given a weight of 10. This balance ensures the agent prioritizes winning and blocking, while still considering positional strategies.

## 3.4 Experiments

In the first experiment, I evaluated the agent's performance using a series of game puzzles from Doron Zeilberger's *Introduction to Shalosh B. Ekhad's "120 Connect-Four End-Game Problems"* [2]. The puzzles are divided into five chapters, each increasing in difficulty. The first chapter consists of puzzles that require one move to win, the second chapter requires two moves, and so on. For this experiment, I input the final puzzle from each chapter and tested whether the agent could successfully solve it. To solve the puzzle player 1 (the agent) needs to win in X moves. When there are multiple options for player 2 I opt for the most optimal play according to [3] since going through all the game trees in the solution manual would be too big of a scope.

The first problem (Figure 2) is straightforward, requiring a piece in row 1, column 4 to win the game. The agent correctly identifies and selects this move.

In the second problem (Figure 3), two moves are needed to win. According to the solution manual, either playing row 1, column 4, or row 1, column 5 on the first turn will work, followed by capitalizing on the created threat. The agent selects row 1, column 4, which is expected due to its preference for the center column. After updating the game state, the agent selects row 2, column 4 as its second move, successfully solving the puzzle.

The third problem (Figure 4) requires three moves to win. The agent initially chooses (4,4), which deviates from the solution manual's recommendation of column 6 as the optimal play [3]. After updating the board and allowing player 2 to respond with row 3, column 6, the agent selects row 5, column 4, creating a vertical threat. player 2 blocks this, and the agent then creates another threat in column 5, ultimately failing to win in the expected three moves.

The fourth problem (Figure 5) requires four moves to win. The optimal move is column 3, but the agent opts for the middle column. After player 2 plays row 2, column 2, the agent blocks the horizontal threat with the optimal move, positioning itself to win in four moves, slightly underperforming compared to the solution manual. However, after some back-and-forth blocking, the agent plays (6,4), a suboptimal move, further delaying the win.

The fifth problem (Figure 6) requires five moves to win. The agent starts with (3,4), an optimal move, and player 2 responds with (3,5). The agent follows up with (4,4), still optimal, and player 2's response is (1,1). The agent then plays (5,4), another optimal choice. player 2 blocks the vertical threat with (6,4). The agent considers both (3,2) and (3,3), with (3,3) being the optimal move. Choosing (3,3) leaves player 2 two moves from defeat, while (3,2) would give player 2 three moves. After choosing (3,3), player 2 responds with (4,3). The agent then selects one of the optimal moves, (2,7), leaving player 2 with no solid counter. After blocking the most obvious threat at (4,5), the agent finishes the puzzle by playing (5,5).
Although the agent solved the puzzle in six moves instead of the optimal five, this slight deviation was most probably due to multiple equally weighted optimal choices. Nevertheless, the agent performed well, completing the game relatively quickly.

The full output of the agent for each problem can be found here.

In the second experiment, I aimed to assess whether the agent could successfully defeat an average human player in a game of Connect Four. To test this, my parents and I played several games against the agent. I automated the process of inputting new moves and running the ASP program on the updated game state using a Python script, which can be found in the project's GitHub repository here.

## 4 Results

**RQ1:**The agent performed well on early to mid-game puzzles but began to struggle with more complex late-mid-game and end-game scenarios. One issue I observed was the agent's overly strong preference for the middle column, which likely hindered its performance in tougher situations. This could potentially be improved by adjusting the weight assigned to the center control strategy. This limitation is expected, as the strategy weights were configured based on limited knowledge of Connect Four tactics and a small set of strategies. Despite these challenges, the agent performed surprisingly well on the most difficult puzzle, coming close to solving it.

**RQ2:** The agent was able to secure a few wins, largely due to occasional mistakes on our part. However, in more focused games, the agent typically lost. While the agent can defeat an average human player, it does so inconsistently.

## 5 Conclusion

The agent performs well in identifying the optimal move in early and mid-game scenarios, but tends to struggle in late-game or end-game situations. This is likely due to the limited set of strategies used and the suboptimal weighting of those strategies. Improvement could be achieved by expanding the agent's strategy repertoire and refining the weights, either through expert knowledge or by learning them dynamically.

Although the primary goal of this project was to compute optimal moves based on the current game state, testing the agent in full games added an interesting dimension. Since the agent lacks the ability to plan ahead, its struggles in real games are not unexpected. A possible enhancement would be to implement a lookahead mechanism, perhaps using a combination of game tree search and heuristics, to improve its performance in these scenarios.

## 6 Appendix

### 6.1 ASP Code

Listing 9: Win Conditions

```
win(Player) :- cell(Player, Row, Col),
                 cell(Player, Row, Col+1),
                 cell(Player, Row, Col+2),
                 cell(Player, Row, Col+3).

win(Player) :- cell(Player, Row, Col),
                 cell(Player, Row+1, Col),
                 cell(Player, Row+2, Col),
                 cell(Player, Row+3, Col).

win(Player) :- cell(Player, Row, Col),
                 cell(Player, Row+1, Col+1),
                 cell(Player, Row+2, Col+2),
                 cell(Player, Row+3, Col+3).

win(Player) :- cell(Player, Row, Col),
                 cell(Player, Row-1, Col+1),
                 cell(Player, Row-2, Col+2),
                 cell(Player, Row-3, Col+3).
```

Listing 10: Horizontal Consecutive

```
winning_move(Player, Row, Col) :- valid_move(Row, Col)
   ,
                                   cell(Player, Row,
                                      Col-1),
                                   cell(Player, Row,
                                      Col-2),
                                   cell(Player, Row,
                                      Col-3).
winning_move(Player, Row, Col) :- valid_move(Row, Col)
   ,
                                   cell(Player, Row,
                                      Col+1),
                                   cell(Player, Row,
                                      Col+2),
                                   cell(Player, Row,
                                      Col+3).
```

Listing 11: Horizontal Gap

```
winning_move(Player, Row, Col) :- valid_move(Row, Col)
   ,
                                   cell(Player, Row,
                                      Col-1),
                                   cell(Player, Row,
                                      Col+1),
                                   cell(Player, Row,
                                      Col-2).
winning_move(Player, Row, Col) :- valid_move(Row, Col)
   ,
                                   cell(Player, Row,
                                      Col+1),
                                   cell(Player, Row,
                                      Col-1),
                                   cell(Player, Row,
                                      Col-2).
```

Listing 12: Diagonal Down

```
winning_move(Player, Row, Col) :- valid_move(Row, Col)
   ,
                                    cell(Player, Row
                                       +1, Col-1),
                                    cell(Player, Row
                                       +2, Col-2),
                                    cell(Player, Row
                                       +3, Col-3).

winning_move(Player, Row, Col) :- valid_move(Row, Col)
   ,
                                    cell(Player, Row
                                       +1, Col+1),
                                    cell(Player, Row
                                       +2, Col+2),
                                    cell(Player, Row
                                       +3, Col+3).
```

Listing 13: Diagonal Gap

```
winning_move(Player, Row, Col) :- valid_move(Row, Col)
   ,
                                    cell(Player, Row
                                       +1, Col+1),
                                    cell(Player, Row
                                       -1, Col-1),
                                    cell(Player, Row
                                       +2, Col+2).

winning_move(Player, Row, Col) :- valid_move(Row, Col)
   ,
                                    cell(Player, Row
                                       -1, Col+1),
                                    cell(Player, Row
                                       +1, Col-1),
                                    cell(Player, Row
                                       +2, Col-2).
```

## 6.2 Game Configurations

## References

[1] en. URL: https://www.youtube.com/channel/UCq6XkhO5SZ66N04IcPbqNcw.

[2] URL: https://sites.math.rutgers.edu/~zeilberg/C4/Introduction.html.

[3] en. URL: https://connect4.gamesolver.org/en/.

[4] Louis Victor Allis. “A Knowledge-Based Approach of Connect-Four.” In: *J. Int. Comput. Games Assoc.* 11.4 (1988), p. 165.

[5] Lauren Cahn. *How to Win Connect 4 Every Time, According to the Computer Scientist Who Solved It*. en-US. Nov. 2023. URL: https://www.rd.com/article/how-to-win-connect-4/.

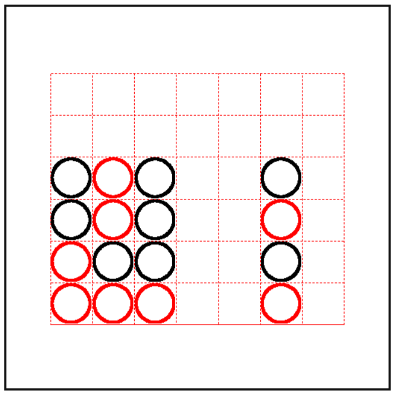

Figure 2: Chapter 1 problem 15

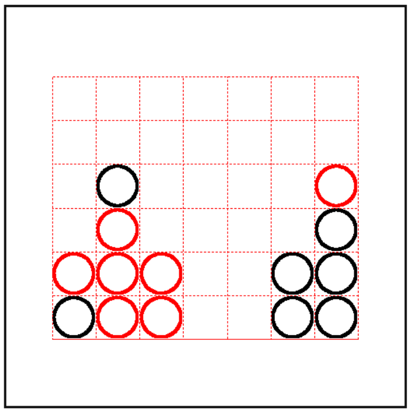

Figure 3: Chapter 2 problem 30

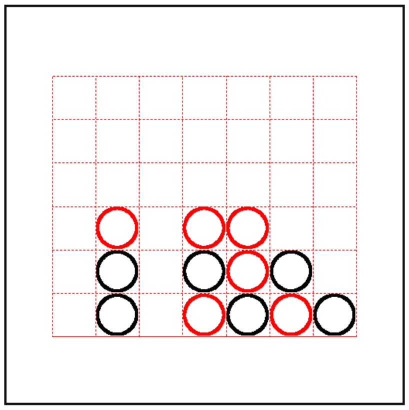

Figure 4: Chapter 3 problem 30

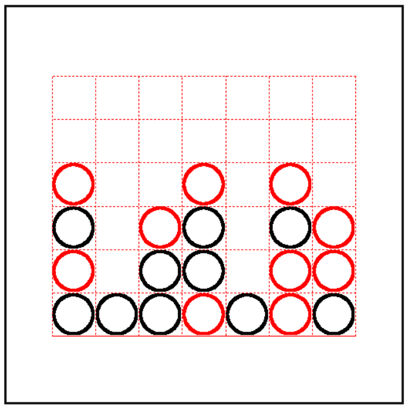

Figure 5: Chapter 4 problem 30

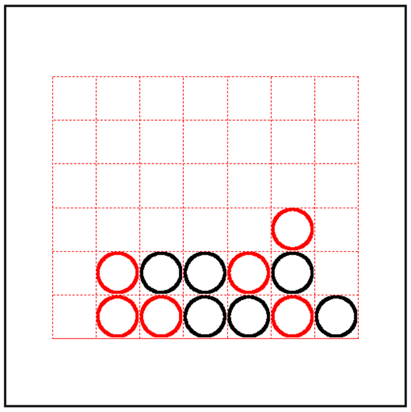

Figure 6: Chapter 5 problem 15