MAE-5370-Optimization for Engineers
Final project report

Daniel Garner; A#02321462

Abstract

Zero-sum games are games in which the rewards of the winning party are always equal to the losses of the losing party. This project aims to solve a zero-sum game, connect 4. This project aims to do so by developing an optimal playing agent. While connect 4 is a solved game[1], the number of possible positions is roughly half a million [1], while the number of possible games (dictated by the order of moves, not just the position of the pieces) is more than $1.6 * 10^{13}$ [2]. This makes it much too large for a real-time brute force solution on standard hardware, like my computer. In order to still solve this game, this project does not brute-force solve the game, it instead seeks to optimize a Heuristic Evaluation Function that allows an agent to make near-optimal decisions within a limited search depth.

This project implements a Minimax algorithm with Alpha-Beta pruning to explore the possible future games. The main challenge is tuning the weights on the heuristic function, or how valuable the program sees each potential position. This was accomplished using the Genetic Algorithm. Over 5 generations and approximately 8 hours of training, the Genetic Algorithm converged on a set of weights that prioritized defensive play and getting two pieces in a row. This algorithm achieved a 100% win rate against the training population.

Problem Description

The primary problem is picking the best spot to go in any given board state within a limited amount of time. In Connect 4, two players, usually red and yellow, take turns placing tokens in a 6x7 grid. The pieces fall straight down, occupying the lowest available space within the column. The objective is to be first to form a horizontal, vertical, or diagonal line of four distinct pieces in a row. (see Fig.1)

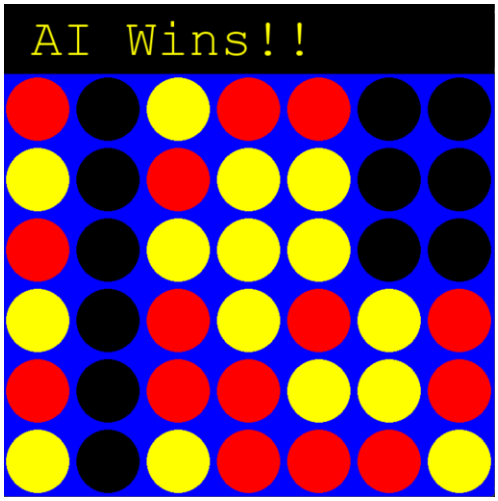

Fig. 1 A connect 4 board, in which the AI (yellow) has won

This game is discrete, deterministic and perfect-information as actions are done in turns that the players have full control over and all players can see everything on the board at once. At any point in time the player has up to 7 choices, one for each column. Looking just 7 moves ahead results in 7^7 possible states, or 823,543 possible states. One of the main challenges is to develop a function that can assign a scalar value to the board state. This value from the function represents the player winning the game, without having to exhaustively calculate the entire game.

As such, this project actually contains two different optimization problems. The first is evaluating the current board state and picking the action that maximizes the future utility. Basically put, it will choose the best move, one that increases it's own position "score" while simultaneously keeping the opponents "score" as low as possible.

The second optimization is tuning the function. This means finding the best possible weights in the heuristic function, the function that evaluates the "score" at each board state. There are also constraints that must be followed that correspond with the rules of the game. They are as follows: a disc can't be placed in a column if the column is full, there is gravity and the disc will fall to the bottom, and finally there is a turn order and the players must alternate.

Optimization Formulation

In order to solve this problem using optimization methods, we need to mathematically define the board state, search objectives, and the restrictions that the we need to follow as per the rules of the game.

We can represent the board as two binary matrices, X and Y, each of size 6 x 7. Let i represent the row index (i=1…6, where 1 is the bottom row). Let j represent the column index (j=1…7). Within these matrices, x_(i,j)=1 if Player 1 has a token at position (i,j), else 0. Similarly, y_(i,j)=1 if Player 2 has a token at position (i,j), else 0.

The other rules of the game are defined as follows:

Rule 1: Can't have more than 1 token per tile-

$$x_{i,j} + y_{i,j} \leq 1$$

Rule 2: There is gravity, the slot underneath must be full-

$$x_{i,j} + y_{i,j} \leq x_{i-1,j} + y_{i-1,j}$$

Rule 3: There is a turn order, there can only ever be 1 more tile of one kind than another-

$$\sum_{i,j} x_{i,j} - \sum_{i,j} y_{i,j} \in \{0,1\}$$

In this solution, we will assume that the opponent plays optimally, making this a Zero-Sum game. Since the goal of the problem is to maximize the players value while minimizing the other player's value, we can represent this as a function. Let $f(S, W)$ be the heuristic score of state S given weights W. The value of a current node v as depth d is:

$$v(S) = \max_{a \epsilon A}(\min_{b \epsilon B}\left(\dots f\left(S_{leaf}, W\right)\right))$$

Since we can't traverse the leaf node of the entire game, we use estimate how good a position is using a linear combination of features. The decision variables that the function uses to evaluate score are as follows:

w_{center}- the value of controlling the center column

w_{block}- the value of denying an opponent 3 pieces in a row

w_2- the value of creating 2 pieces in a row

w_3- the value of creating 3 pieces in a row

w_4- the value of creating 4 pieces in a row

N_{center}- the number of pieces in the center column

N_4- the number of 4 pieces in a row

N_3- the number of 3 pieces in a row

N_2- the number of 2 pieces in a row

N_{opp_3}- the number of 3 pieces in a row the opponent has

The optimization goal is to maximize the utility score f at a future depth d:

$$f(S) = w_{center}(N_{center}) + w_4(N_4) + w_3(N_3) + w_2(N_2) - w_{block}(N_{opp\_3})$$

Our goal is now to maximize the weights in this function. To do this, we treat the weight vector (a vector containing all the weights) as the genome in the genetic algorithm as discussed below.

Algorithm and Implementation

The solution was implemented in Python using numpy for the matrix operations and multiprocessing in the implementation of the genetic algorithm as will be discussed below.

The algorithm used while playing the game is Minimax with Alpha-Beta Pruning. The core game engine, that actually does the thinking and decision making is Minimax, which does it's best to maximize your value while minimizing the opponents value [3]. The way Alpha-Beta pruning is it maintains two different values, α (the best value maximizer value so far) and β (the best minimizer value so far). If at any node the algorithm find a move that is worse than a move previously found in a neighbor branch (worse being defined as $\alpha \geq \beta$), than it stops searching that branch instantly [4]. Introduction of alpha-beta pruning reduces the search complexity from order $O(b^d)$ to approximately $O(b^{\frac{d}{2}})$. This allows the algorithm to search deeper within the same time constraints. Thanks to Alpha-Beta pruning, I am able to run the minimax algorithm at a depth of 7.

In order to find the best possible weights for the function, I used the genetic algorithm. The genetic algorithm can be represented with the following equation.

$$W = \sum_{k=1}^{N} Outcome_k(W)$$

Subject to:

$$W \in \mathbb{Z}^+$$

To find the optimal W, I implemented an evolution algorithm with the following stages.

1. Initialization: A population 32 bots was created with random weights.
2. Evaluation: Each both plays 5 games against random opponents from the population. The fitness score of each bot was totaled, with 1 point for winning a game, 0.5 points for drawing a game, and 0 points for losing a game.
3. Selection: The population was sorted by fitness score. The top 50%, comprising 16 bots, were selected to reproduce.
4. Mutation: The selected bots were cloned. These clones then undergo mutation, where a random noise factor $\delta \in \{-5, -2, -1, 1, 2, 5\}$ was applied to one of their weights.
5. Iteration: This process was repeated for 5 generations.

In an ideal world, I would keep running this for more generations, however; for reasons discussed in results, I only ran 5 generations.

The heavy lifting during the genetic algorithm portion of the code was done by the multiprocessing library in python. This library allows the python script to utilize all 16 cores on my CPU. This reduced the genetic algorithm run time from an estimated 4 days to approximately 8 hours.

Results and Interpretation

The genetic algorithm was run on a Ryzen 7 5700G with 16 cores. The search depth for training was set to 7, the depth the model uses in game. Below is the progression of the best bot from each generation.

Generation	w_{center}	w_4	w_3	w_2	w_{block}	Score (/10)
1	3	100	1	10	29	9
2	3	100	1	10	29	9
3	3	100	1	8	29	9
4	9	100	8	9	29	9
5	3	100	1	8	29	10

The algorithm converged in the 5th Generation with a perfect win rate. The resulting weights (W=[3,100,1,8,29]) provide an interesting insight to how to best play connect 4. The optimization from the genetic algorithm assigned a significantly higher value to blocking an opponent than to creating a 3 tiles in a row of their own. This confirms that avoiding a loss is much more important than just going blindly for a win.

Interestingly, the model favored getting 2 tiles in a row over 3. This is likely because having more 2 tiles in a row offers much more options more quickly than 3 tiles in a row. In a situation

where the model has many 2 in a rows, the opponent can't block everything. The center control remains low and constant. This is insightful because, although the center is the most important column to control, block opponents and creating opportunities for oneself is much more important.

To verify the solution, the optimized weights are hard coded into the final program. This final program was tested against a random agent at a reduced depth of 5 (as opposed to the depth of 7 used when playing a person), the program had a win rate of 100%. When the program was tested against a "greedy" agent, that goes for instant gratification (depth of 1), it also had a 100% win rate. To further verify, the model played against 20 real people, winning 20/20 games.

In the future this can be further optimized. The results can be improved by more generations in the Genetic Algorithm and increasing the number of games each model plays during the Genetic Algorithm training. Additionally, the model can be improved by implementing more measures in the heuristic function. Things such as, assigning weights to each column, not just the middle, implementing a weight that would prevent the model from making a "dead-end" 3 pieces in a row, would greatly benefit the model, allowing it a more complete "vision" of the game.

Suboptimal performance is noted in this model. Although the model remains undefeated against normal people, I think a person who is thinking more than 7 moves ahead, outside of the model's maximum depth, would be able to win. Additionally, playing against models that use the solved map of connect 4, this model either loses or draws. Usually, it loses. This is evidence of the fact that this model still has room for improvement. Also, when playing people, although it never lost, the model sometimes chose not to go in a slot that would connect 4 in a row, and instead go somewhere else, passing up the opportunity to instantly win.

A further optimization could be implemented to speed up the Alpha-Beta pruning. This process could be sped up by beginning the search in the center column (typically the most valuable column to go in), which would allow it to prune other branches much quicker. Additionally, allowing the model to "remember" it's Alpha-Beta tuning from the previous turn (typically called a transposition table) would speed up the model's execution in game.

Conclusion

This project successfully applied engineering optimization techniques to solve a Zero-Sum game. By formulating Connect 4 as a heuristic formula, I was able to implement a Minimax solver capable of playing the game better than any human. Additionally, by applying the Genetic Algorithm, I was able to automate the tuning of the decision variables in the heuristic function.

From this project, I have learned that in Connect 4, the value of defense greatly outweighs the importance of offense. The algorithm discovered that an optimal strategy was to build as many pieces of 2 in a row as possible, while defending from opponents' threats. This strategy proved much better than other aggressive strategies that appeared during the Genetic Algorithm training phase. The final deliverable is a Python-based application with a GUI that implements these optimized weights. Additionally, I have attached the files for the Genetic Algorithm training, and the play against different agents.

References

[1] Allis, V. (1988, October). A knowledge-based approach of Connect-Four the game is solved: White wins. https://tromp.github.io/c4/connect4_thesis.pdf

[2] MIT. (n.d.). Introduction solvability rules computer solution implementation connect four. https://web.mit.edu/sp.268/www/2010/connectFourSlides.pdf

[3] Shannon, C. E. (1988). Programming a computer for playing chess. Computer Chess Compendium, 2–13. https://doi.org/10.1007/978-1-4757-1968-0_1

[4] Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. Artificial Intelligence, 6(4), 293–326. https://doi.org/10.1016/0004-3702(75)90019-3

Code

The final code is attached to the submission, not included in this document.