Playing Connect Four using Monte Carlo Tree Search

Jonathan Vincent Kilian Wohlleben

June 2022

1 Introduction

Connect Four is a two player perfect information board game, consisting of a 7 by 6 upright grid, where players take turns dropping coloured counters into the top of the grid. A player wins when their coloured counters form a connected row, column or diagonal of length 4. If the grid fills up without one player achieving a connected line of 4, the game ends in a draw.

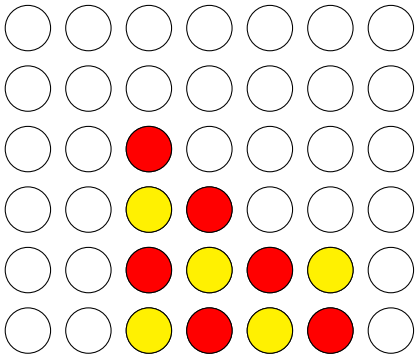

Figure 1: Red wins with a diagonal 4 in a row

While the game is simple in complexity compared to other popular board games such as chess or Go, there are still 4,531,985,219,092 possible board states, making brute force methods difficult, but not intractable. The game has been solved since 1988, and player 1 can force a win in 28 moves.[Tro]

In this project, we will use Monte Carlo Tree Search to build an agent that, while not perfect, can frequently beat a human, and compete reasonably well with a perfect player.

2 Monte Carlo Tree Search

Monte Carlo Tree Search is an algorithm to estimate the value of an action at each state. Different variants of the algorithm exist, with this one mostly taken from the Reinforcement Learning textbook by Sutton and Barto [SB18]. Whenever the agent needs to make a decision, the following 4 steps are carried out:

1. Selection: At a root node (i.e. a given board state), evaluate the child nodes (i.e actions) using the Upper Confidence Bound (UCB):

$$UCB_i = \frac{\text{wins}_i - \text{losses}_i}{n_i} + c\sqrt{\frac{\log(N_i)}{n_i}}$$

where n_i is the number of simulations from the ith node, N_i is the number of simulations from the parent node and c is the exploration parameter.

The left term indicates the current estimate of the value of the action, and the right term estimates the uncertainty of this estimate based on how many times that action has been chosen. For unexplored actions, n_i is 0 and thus UCB_i is infinite.

2. Expansion: Choose the child node with the highest UCB. These will be unexplored nodes at first, and then nodes that are promising but have not been explored too thoroughly.

3. Simulation: From the chosen child node, perform multiple 'rollouts', playing the game down the tree until a terminal state is reached. The rollouts can be performed randomly, or in our case, using UCB again.

4. Backpropagation: Once a terminal node is reached, move back up the tree, updating the evaluation of the node.

These steps are then performed many times per node, giving a reasonable estimate of how good each state, action pair is.

3 Implementation

We split our implementation into two parts: A game class and a MCTS class. The main focus of the game class is performance so the MCTS can play as many simulations as possible. We have kept the game class as simple as possible with only few methods to add and remove moves, a history and methods to get the legal moves, the results and the game state as a string which we use to fill the MCTS Q value dictionary. We can create Connect 4 games for arbitrary board sizes, but we have only evaluated the common version in our experiments. The MCTS class takes the game as an input, as well as parameters for c,$n\_branches$, the number of rollouts, and $symmetry$, which

we can use as Connect 4 is a symmetrical game. When trying to find the best move for a given state, the run function first checks, whether the game isn't already finished, in which case it returns the result. Then it performs $n\_branches$ rollouts and during the backpropagation phase saves the result into the Q dictionary. The Q dictionary is the heart of the algorithm. It saves the result for a gives game state by converting that state to a string and adding it to a list of the form $[draws, player\_1\_wins, player\_2\_wins]$. This is very neat as it allows access via the turn flag of the game class ($\in \{-1, 1\}$). In the policy function we find the node with the highest UCB by using the Q dictionary to calculate the average outcome of the move and the standard second part. The second part contains an additional explore part so we can turn it off when we actually decide on a move, as we don't want to perform any exploration at that point.

We have also created some helper scripts to play the game. optimize_c is a very rudimentary heuristic algorithm to give us a good value for c. train_Q is a script that "trains" Q, i.e. it plays a lot of games with a high number of rollouts to fill Q with good approximations for the results for early states. We only save the states that have been visited a lot, as the dictionary gets very large otherwise (With 100 games with 1000 rollouts it generated over 1 GB of data, which caused memory errors). The dictionary is then saved into a json file which can be reused later on. demo is a file that shows a game of the algorithm against itself and play_against_ai lets humans play against the algorithm.

4 Results

It is difficult to objectively quantify the quality of play in a game like Connect 4, in the absence of sensible 'baseline' players or readily available Connect 4 engines. We can however say with confidence, that it clearly outperforms 1 and 2 step look-ahead searches.

4.1 Without prior training

These experiments were performed without training the Q dictionary beforehand, using only the information gained from the rollouts. We used 1000 rollouts for each move. It won a game against an online version (`https://www.cbc.ca/kids/games/play/connect-4`) of connect 4 as player 2. It was also able to beat a few humans without training, while it lost against others.

Comparing one round of an untrained MCTS against itself with the theoretically optimal moves (taken from [Pon]), we saw that it played the best move in 22 out of 32 cases, the second best in 7 cases and the fourth, fifth and sixth in 1 case each. During the game the optimal result only changed three times, which means that in a game there were only three losing moves. While in many cases the optimal move is obvious, in others it isn't clear at all, whether any given move is perfect, good or even losing. The optimal strategy often relies on "Zugzwang" which is difficult to see far in advance.

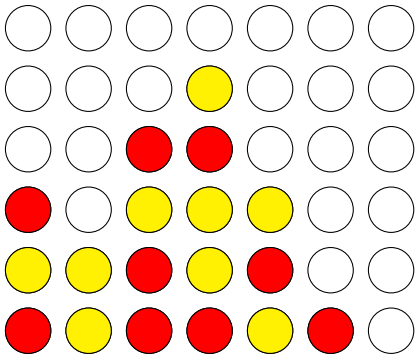

Figure 2: Example of over optimism when using too few rollouts. In this position the MCTS played its red counter into the second column, lining up for a diagonal 4 in a row on red's next move. The algorithm is optimistic, as 5 out of 7 of yellows moves would result in a win for red.

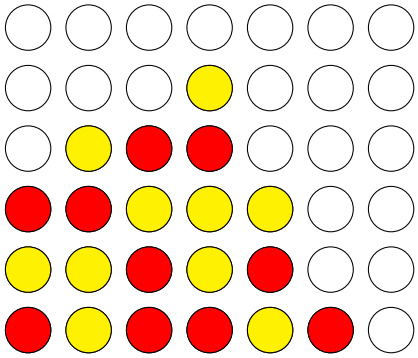

Figure 3: As soon as red plays this move, yellow has a 'mate in one', getting a 4 in a row diagonal

The MCTS can often be overly optimistic, as the UCB encourages it to play moves that win often, but not necessarily by force. In the example above, using an MCTS with only 100 rollouts, red plays a promising looking move which loses it the game. This is an example where MCTS fails where a more traditional game playing algorithm like minimax would succeed.

5 Conclusion

We have created an algorithm that performs well against human and non-human players. While not

being able to play perfectly, it achieved very good results. The main drawback is that it is "oblivious" about states it hasn't seen before. In other implementations of MCTS like Alpha Zero, this issue was overcome by using a neural network to evaluate the value of the state. While that would've been possible in our case, we never managed to improve the training loss of the neural network to a good level, likely due to a lack of training data. A more exhaustive parameter search could've solved that but we lacked the computing power to do so. What worked very nicely, was the fast execution speed of the MCTS, being able to perform 100 rollouts per second for the initial state on our machine.

Bibliography

[SB18] Sutton and Barto. *Reinforcement Learning: An Introduction*. 2018.

[Pon] Pascal Pons. *Connect 4 Solver*. URL: `https://connect4.gamesolver.org/en/`.

[Tro] John Tromp. *John's Connect Four Playground*. URL: `https://tromp.github.io/c4/c4.html`.