AIFour: Computer AI Playing Connect Four

Eugene Chan

CS152 Spring 2018

See the Jupyter Notebook Implementation here

LOs Used:

#search: Both algorithms are applications of naive search in multiple agent settings

#aiconcepts: NegaMax uses AI concepts like MiniMax and graph search to determine the best move. Monte Carlo Tree Search uses graph search to determine the best move.

#aicoding: See the Jupyter Notebook for full implementation of NegaMax and Monte Carlo Tree Search in Python

Problem Definition

Connect Four is a two-player game where the aim is to have four of the same colored disc horizontally, vertically or diagonally. Players will take turn to place their own colored disc on one of the seven columns in the 7x6 board. The discs of the same column stack on top of each other.

It is a zero-sum game because either player would win, or both would draw the game. It is also a complete game because each player has perfect information on what the current board look like and what are the available moves for the other player.

What makes Connect Four an interesting AI search problem?

Although there is limited steps that each player can take (at most 7, unless the board has a different number of columns), the number of possible state of the board can be up to four trillion.

To build a computer bot that can play against a human player, it has to understand the board and determine a move in a reasonably short time. Therefore it is impossible to search through all the possible states before determining the move

How can AI search algorithms be applied in this bot?

This AI bot deploys either of the two search algorithms: NegaMax (a variance of the MiniMax algorithm) and Monte Carlo Tree Search. These two search algorithms does a partial search on all the possible steps given a state of the board and return the best move it can find. While one is breadth-first search (Monte Carlo Tree Search), while the other is depth-first search (NegaMax), both are able to search through a large number of board states and determine the best step given the steps simulated.

Solution Specification

There are two solutions implemented, NegaMax (a variation of MiniMax search) and UCT variant of Monte Carlo Tree Search.

NegaMax Search Algorithm for Connect Four

NegaMax search algorithm is a variation of the MiniMax search algorithm. The MiniMax search algorithm seeks to maximize the utility function when it’s the algorithm's turn to play, the minimize the utility when its the other player's turn (hence maximizing the algorithm's utility).

The algorithm simulates the possible steps by alternating players, until it reaches a terminating stage (win, lose or draw)

NegaMax has the same aim of maximizing the algorithm's utility during its turn, and minimize the utility during the other player's turn. But it is a simplified version of MiniMax, utilizing the fact that in connect four, the outcome is dichotomic, meaning that the minimum utility is the negative (losing has the utility of -N) of the maximum utility (wining has the utility of N)

Inner Working of the Nega-max Search Algorithm

NegaMax works with a recursive call. At every call, it will create a leaf node based on a legal action. If the state of the leaf node is not a win, lose or draw, it will branch again and form a child node with another action. Detailed procedures and implementation can be found in the Jupyter Notebook linked on the first page, under the section *"Nega-Max for Connect Four"*.

What if the game does not terminate after the algorithm reaches maximum search depth?

As mentioned above, not every path will lead to a terminal state when the maximum search depth is reached. In that case, we cannot assign a simplistic win-lose-draw utility to that particular leaf node.

We employ a heuristic function that assign utility based on the number of chains in the current state for each player. The underlying principle is that a longer chain implies higher chance of winning. Therefore different lengths of chain will have different score weightings. The difference between the scores will signal which player has a higher chance of winning if we continue to simulate the game. The detailed implementation can be found in the *evaluate()* function under the *ConnectFour* class.

Simplistic Monte Carlo Tree Search

The limitation of the NegaMax search algorithm (or any variation of the MiniMax search algorithm) is that the number of search steps is usually very large to run through a large amount of simulation in order to obtain the best move. A way to mitigate this problem is using the evaluate heuristic introduced above.

Another search algorithm that can mitigate the problem with MiniMax when it encounters high branching factor is Monte Carlo Tree Search. Monte Carlo Tree Search chooses the node with the best result (wins/game simulated) and simulate the rest game based on the action of that best node. Each time it simulates the game, it will reach a conclusion (win, lose or draw). It will then update the win/simulation ratio for all of its parents.

The benefits of Monte Carlo Tree Search over NegaMax algorithm in theory are:

1. It can simulate more full games than NegaMax, so to provide a better estimate of the end result given a move
2. Every search is a greedy decision: selecting the node with the best outcome currently to explore. It saves a lot of search space compared to NegaMax's depth first search regardless of the outcome of the path

Implementation of Monte Carlo Tree Search

There are **four steps** to Monte Carlo Tree Search: Selection, Expansion, Simulation and Back propagation. Detailed implementation and explanation is in the Jupyter notebook under the section *"Implementation of Monte Carlo Tree Search"*

UCT Variant of Monte Carlo Tree Search

The problem with simple Monte Carlo Tree Search is that it is at a tug of war with two different goals: exploitation and exploration.

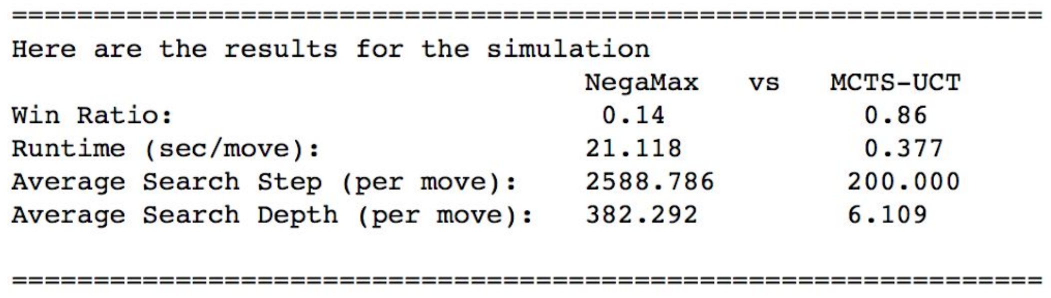
On one hand, it wants to exploit the child node of the root with the best win/simulation value, going deeper into the branches of that node. On the other hand, it needs to explore other actions that may give a higher win/simulation value than the ones explored. The simple implementation exploits the child node with the best win/simulation, until there is enough loses to render it suboptimal. However, it may miss the exploration of other nodes that may later on yield better win/simulation ratio

The UCT variant of Monte Carlo Tree Search that tries to balance the exploitation and exploration goals of the search algorithm by assigning a search weight to each node based on both the win/simulation ratio and the number of times the node has been visited. The detailed implementation is in the Jupyter notebook under the section *"UCT Search Weights"*.

Negamax vs Monte Carlo Tree Search

To evaluate the practical performance of the two major search algorithms. I built a simulation and allow a NegaMax bot play against a UCT-Monte Carlo Tree Search bot. The simulation plays 50 games.

The metrics measured after the simulation for both bots are: Search depth (per move), branching factor/search steps (per move), search time (per move), and win percentage



```
===============================================================
Here are the results for the simulation
                                   NegaMax   vs   MCTS-UCT
Win Ratio:                          0.14            0.86
Runtime (sec/move):                21.118           0.377
Average Search Step (per move):    2588.786        200.000
Average Search Depth (per move):   382.292         6.109

===============================================================
```

Fig. 1 -- Result of the 50 game simulation. Detailed result per game and analysis of the results can be found in the Jupyter Notebook linked in the first page under the section "Negamax vs Monte Carlo Tree Search"

References

For NegaMax Search Algorithm:

Pons, P. (2017, February 19). Part 3 – MinMax algorithm. Retrieved April 18, 2018, from http://blog.gamesolver.org/solving-connect-four/03-minmax/

Ahmadi, H. (n.d.). An Introduction to Game Tree Algorithms. Retrieved April 18, 2018, from http://www.hamedahmadi.com/gametree/#negamax

For Monte Carlo Tree Search:

Bradberry, J. (n.d.). Introduction to Monte Carlo Tree Search. Retrieved April 18, 2018, from https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

Brown, C. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transaction on Computational Intelligence and AI in Games,4*.