Monte Carlo Tree Search for Two Player Snake

Algot Johansson Eric Guldbrand

May 8, 2020

Abstract

We implement a game playing system that uses Monte Carlo Tree Search (MCTS) to play a two-player snake game where both players move simultaneously each turn. The game playing system outperforms random, simple scripted and human players, but has a tendency to seek out draws.

1 Background

In a previous module we implemented Minimax tree search for Tic-tac-toe but had no time to implement MCTS. We now wanted to implement MCTS as well, but for some other game.

The game chosen is a two-player Snake game inspired by the game *Achtung, die Kurve!*[1]. Each player controls one snake. Snakes move on a grid, one block every turn, but also grow one block longer every turn, meaning that the snake's tail remains still.

This game differs from Tic-tac-toe in two important ways. Firstly, both players move simultaneously at each turn, without knowing how the other player will move. Secondly, while the board is usually much bigger than a Tic-tac-toe board, there is only a maximum of three moves available to the player at each turn: continue forward, turn left, or turn right. However, each game tends to be longer than a game of Tic-tac-toe played on a 3-by-3 or 5-by-5 board.

2 Method

A game arena was implemented for matching different types of players against each other. This included a few baseline players: one random, one rule-based and one player for humans to use.

[1] `achtungdiekurve.net`

A GUI was also created to display the current state of the game. A screenshot from the GUI can be seen in figure 1. Here we can also see that the game implements the possibility for snakes to travel through walls.

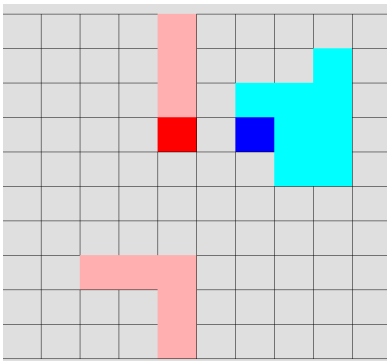


Figure 1 A screenshot from the GUI created for this application. With human (red) playing against the AI (blue).

However, the most important player is the MCTS agent. This player is implemented with random rollout, although the random moves will always pick a move that does not cause the agent to lose immediately, if such a move is available. (It will not turn into an already existing wall, but it might turn into a dead-end that will force it to lose after the next turn.)

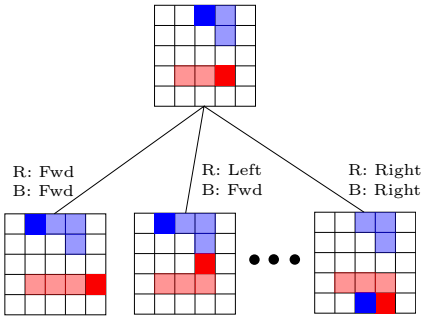
Since each turn of Tic-tac-toe involves both players moving, it is not given how child nodes should be generated in the state tree. Our first approach was to generate all combinations of moves between both players each turn, see Figure 2a. However, this did not work out very well since it is often easier for the other player to lose than it is for you to win. Thus the system would tend to ”choose” the worst possible move for its opponent, and not explore its own best move as well as it should, although it still outperformed a random opponent.

Another approach, and the one chosen for the final system, was to pretend that each turn in the game is actually two turns, and that only one player moves at each pretend turn, see Figure 2b. This reduces the problem to the same form as a regular turn-based game such as Tic-tac-toe and minimaxing behaviour can be utilized when exploring states.

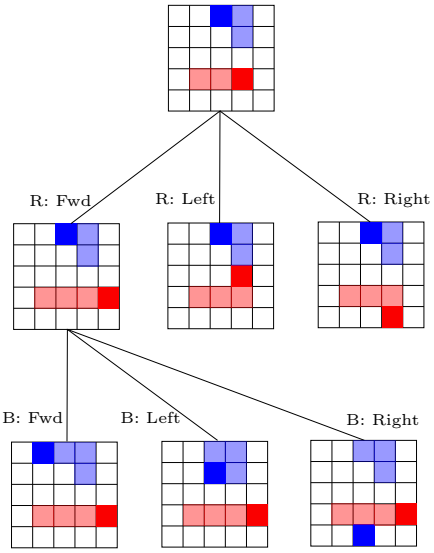
In the final agent, the tree is rebuilt from scratch after each game turn. Maybe there could be some way to use the values calculated in the previous to obtain information on the current state.

3 Results

The final agent plays quite well, but has a tendency to seek out draws. Table 1 shows how the agent performed on a 10 by 10 board against a random player, a scripted player, human players and itself using different numbers



(a) The child layer contains all possible combinations of moves from both players. This tree more accurately represents the flow of the game, but is more difficult to search using minimax-like behavior.



(b) Even though players actually move simultaneously, the agent will pretend that players have one turn each. In the first child layer, a state exist for each possible move for the red player. In the second child layer, a state exists for all possible moves of the blue player, given the red player's move.

Figure 2 Two different types of state trees for the two-player snake game. One of these is the tree that is searched using MCTS by the agent. Labels show the move taken to get to each child for B(lue) player and R(ed) player. Our final agent uses the tree in 2b.

of rollouts (n). The agent playing is using 100 rollouts each turn. To note is that the random agent is not allowed to take an action that will make it lose immediately, if another action is available. The scripted agent always goes forward until it encounters an obstacle and then turns to a direction in which it will not immediately lose. Human players consisted of a small number (4) of Chalmers students playing with a round-time of 0.5 seconds.

Table 1 MCTS agent performance with $n = 100$ rollouts when playing against random, scripted and human players, as well as against itself using different numbers of rollouts. Random players were not allowed to make a move that would immediately lose them the game. Scripted player goes forward until it encounters an obstacle, then chooses a direction at random in which it will not immediately lose. Human games were played against a small number (4) of computer science students.

Opponent	Wins	Losses	Draws	Games Played
Random	65%	2%	33%	10 000
Scripted	57%	4%	39%	10 000
MCTS, $n = 10$	20%	13%	67%	1 000
MCTS, $n = 100$	13%	10%	77%	1 000
MCTS, $n = 10\,000$	17%	3%	80%	1 000
Humans	40%	13%	47%	15

4 Discussion

A problem with the turn based state tree is that, in practice, the AI is likely to draw, since it tries to enter the same square at the same time as the opponent does. This tendency to draw can be clearly seen in Table 1. This is likely due to the fact that the simulations are turn-based, so the agent thinks that it can get an advantage by taking the square just in front of the opponent, as that will greatly limit which moves the opponent can make. One possible solution to this problem would be to, in the simulation, add extra ruling to indicate that the game will result in a draw if the opponent crashes into the head of our snake.

Studying the results of the agent playing against versions of itself, there seems to be some advantage for player 1 over player 2, as one might expect that the MCTS $n = 10\,000$ agent to otherwise outperform the MCTS $n = 100$ agent that played against all opponents. This might be because of the fixed starting positions and starting directions for both players. These were chosen somewhat arbitrarily and might give a slight advantage to one player. It was not possible to redo the tests more thoroughly with other starting positions due to time constraints, but this might be of interest in the future. However, interestingly, the difference in number of wins was smaller when the agent

played against a version of itself with the exact same $n = 100$, so perhaps the poor performance of $n = 10\,000$ could lie elsewhere as well.