

Dynamic Programming & Reinforcement Learning Assignment 3

Qiuyang Fu (2721961)

December 7, 2021

Assignment Description

This assignment's goal is to develop an MCTS tree search algorithm for the game tic-tac-toe, and the implemented solution should be effective to beat a random agent. For the random agent, this agent's moves are purely random at all times. And the starting position is restricted to the following state:

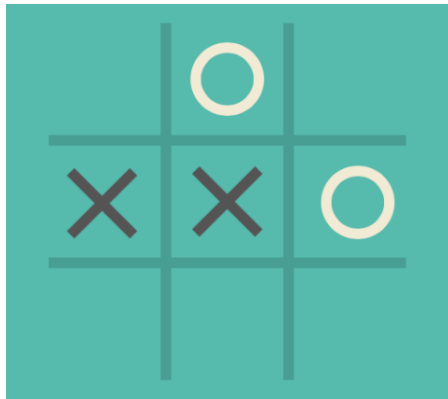


Figure 1: Starting State

a). Monte Carlo Tree Search Solution

In layman's terms, Monte Carlo Tree Search (MCTS) is basically creating a search tree using nodes based on simulated outputs. The four steps of the algorithm process are: Selection, Expansion, Simulation, and Backpropagation.

In "Selection", it starts at root node R, and it looks for the best child node through iterations until it reaches leaf node L. The following UCT formula provides an optimal way for selecting the best child node, and the code is followed below:

$$\frac{w_i}{n_i} + \sqrt{\frac{\ln N_i}{n_i}}$$

```

58     def best_child(self, c_param=1.4): # Return the best child node
59         choices_weights = []
60         for c in self.children:
61             choices_weights.append((c.q / (c.n)) + c_param * np.sqrt((2 * np.log(self.n) / (c.n))))
62

```

Figure 2: Select Best Child

In "Expansion", if L is not a terminal node (which means it won't cause the game to end), then create one or more child nodes and select one of them as C.

```

31     def expand(self): # Expand
32         action = self.untried_actions.pop()
33         next_state = self.state.move(action)
34         child_node = MCTSNode(next_state, parent=self)
35         self.children.append(child_node)
36         return child_node
37
38     def terminal_node(self): # Determine if it's leaf node
39         return self.state.game_over()

```

Figure 3: Expansion

"Simulation" serves the purpose to run a simulated output from C until the end of the game. In this assignment, I run 500 simulations for optimal result and run time.

```

8     def best_action(self, simulations_times): # Get the best action after chosen number of loops
9         for _ in range(0, simulations_times):
10             v = self.tree_policy()
11             reward = v.rollout()
12             v.backpropagate(reward)
13             return self.root.best_child(c_param=0.)

```

```

85 for i in range(0, 100): # Play 100 games
86     print(at + 1, 'game :')
87     while True:
88         new_state, n_board = init()
89         move1 = get_action(new_state)
90         new_state = new_state.move(move1)
91         new_board = new_state.board
92
93         board_state = TicTacToeState(state=new_board, next_to_move=1)
94         root = MCTSNode(state=board_state, parent=None)
95         mcts = MCTS(root)
96         best_node = mcts.best_action(500) # Return best moves after 500 simulations
97         new_state = best_node.state
98         new_board = new_state.board
99         if judge(new_state) == 1:
100             graphics(new_board)
101             break
102         elif judge(new_state) == -1:
103             continue

```

Figure 4: Simulation

And lastly, "Backpropagation" uses simulated output to update current action sequence. It uses the final value that rollout gets to update every node's T and N values in the path. The process is repeated until it reaches the solution and learns the policy of the game.

```

49     def backpropagate(self, result):    # Backpropagation
50         self._number_of_visits += 1.
51         self._results[result] += 1.
52         # Use the final value that rollout gets
53         # to update every node's T and N values in the path
54         if self.parent:
55             self.parent.backpropagate(result)

```

Figure 5: Backpropagation

b). Optimal Policy

While the game is not terminated (`current_node.terminal_node()`), the search tree will keep expanding (`current_node.expand()`) until it is fully expanded (`current_node.fully_expanded()`). If the tree is fully expanded then it will return the best child node.

Every iteration will update the current action sequence, and in theory, the higher number of iterations, the higher accuracy and better results because the reliability of MCTS algorithm is based on numerous repeated samplings and estimations. MCTS could be less reliable in the early stage due to very limited samples, but in the end if given enough time and information, the algorithm will eventually provide the most optimal results by reaching convergence in infinite time. However, it is hard to determine the exact number of iterations for convergence.

```

8     def best_action(self, simulations_times):    # Get the best action after chosen number of loops
9         for _ in range(0, simulations_times):
10             v = self.tree_policy()
11             reward = v.rollout()
12             v.backpropagate(reward)
13             return self.root.best_child(c_param=0.)
14
15     def tree_policy(self):    # MCTS policy
16         current_node = self.root
17         while not current_node.terminal_node():
18             if not current_node.fully_expanded():
19                 return current_node.expand()
20             else:
21                 current_node = current_node.best_child()
22         return current_node

```

Figure 6: MCTS Policy

c). Results

Due to the interesting restricted (starting) position, it is expected to ensure a win rate of 100% for MCTS agent after the first move with a correct implementation. If the opponent's moves are optimal as well, then it will win if it moves first just like MCTS agent.

PROBLEMS	3	OUTPUT	DEBUG CONSOLE	TERMINAL
2		X	—	0
97 game : MCTS Win!				
0		X	0	0
1		X	X	0
2		X	—	—
98 game : MCTS Win!				
0		X	0	0
1		X	X	0
2		X	—	—
99 game : MCTS Win!				
0		X	0	—
1		X	X	0
2		0	—	X
100 game : MCTS Win!				
0		0	0	X
1		X	X	0
2		X	—	—
Total Games Played: 100 MCTS Win Rate: 100.0 % Tie Rate: 0.0 % Random Agent Win Rate: 0.0 % fu@Fus-MacBook-Pro tictactoe_mcts %				

Figure 7: Results