

**LAB MID**

**SUBMITTED BY:**

Faizan Maqbool (SP22-BCS-022)

**SUBMITTED TO:**

Sir Umer Nouman

**COURSE TITLE:**

Artificial Intelligence

**Monto Carlo Tree search**

**Introduction and working of tic tac toe**

Monte Carlo tree search (MCTS) is a heuristic based algorithm for making decisions in games by simulating possible actions. In tic-tac-toe, the simulations are played to the end of the game. The winner of the simulated game is used to update the value of the board that led to that simulation. This process is repeated many times. The MCTS algorithm then chooses the action (i.e., the move on the board) that has the highest value.)

Code below will implement the game Tic-Tac-Toe using Monte Carlo Tree Search (MCTS) for the computer player. Let's walk through how the code works with an example game:

1. **Initialization** :

The game initializes with an empty Tic-Tac-Toe board represented by a tuple of 9 values, where each value can be None, True (for 'X'), or False (for 'O'). The game starts with 'X' ('True') as the first player.

1. **Player Input**:

The human player inputs their move by specifying the row and column numbers (1-3) separated by a comma. For example, "1,2" means placing their mark in the first row, second column.

1. **Game Progression:**

After each player's move, the board is updated accordingly and displayed. Then, the computer player uses MCTS to choose its move. It simulates multiple random games (rollouts) from the current board position to determine the best move.

1. **Winning Condition :**

The game continues until either a player wins, the board is full (a tie), or the human player decides to quit. The winning condition is determined by checking all possible winning combinations on the board.

1. **Outcome:**

Once the game ends, the result is displayed, indicating whether 'X' wins, 'O' wins, or it's a tie.

1. **PRACTICAL IMPLEMENTATION :**

- Human player inputs: "1,1" (placing 'X' in the first row, first column)

X - -

- - -

- - -

- Computer player's move (after simulating rollouts):

X O -

- - -

- - -

- Human player inputs: "2,2" (placing 'X' in the second row, second column)

X O -

- X -

- - -

- Computer player's move (after simulating rollouts):

X O -

O X -

- - -

- Human player inputs: "3,3" (placing 'X' in the third row, third column)

X O -

O X -

- - X

- Computer player's move (after simulating rollouts):

X O -

O X -

O - X

- Human player inputs: "3,1" (placing 'X' in the third row, first column)

X O -

O X -

X - X

Computer player's move (after simulating rollouts):

X O O

O X -

X - X

- The game ends with a tie.

This process continues until the game is over, and the program allows you to play against the computer using MCTS.

**PYTHON CODE :**

"""

A minimal implementation of Monte Carlo tree search (MCTS) in Python 3

Luke Harold Miles, July 2019, Public Domain Dedication

See also https://en.wikipedia.org/wiki/Monte\_Carlo\_tree\_search

https://gist.github.com/qpwo/c538c6f73727e254fdc7fab81024f6e1

"""

from abc import ABC, abstractmethod

from collections import defaultdict

import math

class MCTS:

    "Monte Carlo tree searcher. First rollout the tree then choose a move."

    def \_init\_(self, exploration\_weight=1):

        self.Q = defaultdict(int)  # total reward of each node

        self.N = defaultdict(int)  # total visit count for each node

        self.children = dict()  # children of each node

        self.exploration\_weight = exploration\_weight

    def choose(self, node):

        "Choose the best successor of node. (Choose a move in the game)"

        if node.is\_terminal():

            raise RuntimeError(f"choose called on terminal node {node}")

        if node not in self.children:

            return node.find\_random\_child()

        def score(n):

            if self.N[n] == 0:

                return float("-inf")  # avoid unseen moves

            return self.Q[n] / self.N[n]  # average reward

        return max(self.children[node], key=score)

    def do\_rollout(self, node):

        "Make the tree one layer better. (Train for one iteration.)"

        path = self.\_select(node)

        leaf = path[-1]

        self.\_expand(leaf)

        reward = self.\_simulate(leaf)

        self.\_backpropagate(path, reward)

    def \_select(self, node):

        "Find an unexplored descendent of node"

        path = []

        while True:

            path.append(node)

            if node not in self.children or not self.children[node]:

                # node is either unexplored or terminal

                return path

            unexplored = self.children[node] - self.children.keys()

            if unexplored:

                n = unexplored.pop()

                path.append(n)

                return path

            node = self.\_uct\_select(node)  # descend a layer deeper

    def \_expand(self, node):

        "Update the children dict with the children of node"

        if node in self.children:

            return  # already expanded

        self.children[node] = node.find\_children()

    def \_simulate(self, node):

        "Returns the reward for a random simulation (to completion) of node"

        invert\_reward = True

        while True:

            if node.is\_terminal():

                reward = node.reward()

                return 1 - reward if invert\_reward else reward

            node = node.find\_random\_child()

            invert\_reward = not invert\_reward

    def \_backpropagate(self, path, reward):

        "Send the reward back up to the ancestors of the leaf"

        for node in reversed(path):

            self.N[node] += 1

            self.Q[node] += reward

            reward = 1 - reward  # 1 for me is 0 for my enemy, and vice versa

    def \_uct\_select(self, node):

        "Select a child of node, balancing exploration & exploitation"

        # All children of node should already be expanded:

        assert all(n in self.children for n in self.children[node])

        log\_N\_vertex = math.log(self.N[node])

        def uct(n):

            "Upper confidence bound for trees"

            return self.Q[n] / self.N[n] + self.exploration\_weight \* math.sqrt(

                log\_N\_vertex / self.N[n]

            )

        return max(self.children[node], key=uct)

class Node(ABC):

    """

    A representation of a single board state.

    MCTS works by constructing a tree of these Nodes.

    Could be e.g. a chess or checkers board state.

    """

    @abstractmethod

    def find\_children(self):

        "All possible successors of this board state"

        return set()

    @abstractmethod

    def find\_random\_child(self):

        "Random successor of this board state (for more efficient simulation)"

        return None

    @abstractmethod

    def is\_terminal(self):

        "Returns True if the node has no children"

        return True

    @abstractmethod

    def reward(self):

        "Assumes self is terminal node. 1=win, 0=loss, .5=tie, etc"

        return 0

    @abstractmethod

    def \_hash\_(self):

        "Nodes must be hashable"

        return 123456789

    @abstractmethod

    def \_eq\_(node1, node2):

        "Nodes must be comparable"

        return True

tictactoe.py

"""

An example implementation of the abstract Node class for use in MCTS

If you run this file then you can play against the computer.

A tic-tac-toe board is represented as a tuple of 9 values, each either None,

True, or False, respectively meaning 'empty', 'X', and 'O'.

The board is indexed by row:

0 1 2

3 4 5

6 7 8

For example, this game board

O - X

O X -

X - -

corrresponds to this tuple:

(False, None, True, False, True, None, True, None, None)

"""

from collections import namedtuple

from random import choice

from monte\_carlo\_tree\_search import MCTS, Node

\_TTTB = namedtuple("TicTacToeBoard", "tup turn winner terminal")

# Inheriting from a namedtuple is convenient because it makes the class

# immutable and predefines \_init, \_\_repr, \_\_hash, \_\_eq\_, and others

class TicTacToeBoard(\_TTTB, Node):

    def find\_children(board):

        if board.terminal:  # If the game is finished then no moves can be made

            return set()

        # Otherwise, you can make a move in each of the empty spots

        return {

            board.make\_move(i) for i, value in enumerate(board.tup) if value is None

        }

    def find\_random\_child(board):

        if board.terminal:

            return None  # If the game is finished then no moves can be made

        empty\_spots = [i for i, value in enumerate(board.tup) if value is None]

        return board.make\_move(choice(empty\_spots))

    def reward(board):

        if not board.terminal:

            raise RuntimeError(f"reward called on nonterminal board {board}")

        if board.winner is board.turn:

            # It's your turn and you've already won. Should be impossible.

            raise RuntimeError(f"reward called on unreachable board {board}")

        if board.turn is (not board.winner):

            return 0  # Your opponent has just won. Bad.

        if board.winner is None:

            return 0.5  # Board is a tie

        # The winner is neither True, False, nor None

        raise RuntimeError(f"board has unknown winner type {board.winner}")

    def is\_terminal(board):

        return board.terminal

    def make\_move(board, index):

        tup = board.tup[:index] + (board.turn,) + board.tup[index + 1 :]

        turn = not board.turn

        winner = \_find\_winner(tup)

        is\_terminal = (winner is not None) or not any(v is None for v in tup)

        return TicTacToeBoard(tup, turn, winner, is\_terminal)

    def to\_pretty\_string(board):

        to\_char = lambda v: ("X" if v is True else ("O" if v is False else " "))

        rows = [

            [to\_char(board.tup[3 \* row + col]) for col in range(3)] for row in range(3)

        ]

        return (

            "\n  1 2 3\n"

            + "\n".join(str(i + 1) + " " + " ".join(row) for i, row in enumerate(rows))

            + "\n"

        )

def play\_game():

    tree = MCTS()

    board = new\_tic\_tac\_toe\_board()

    print(board.to\_pretty\_string())

    while True:

        row\_col = input("enter row,col: ")

        row, col = map(int, row\_col.split(","))

        index = 3 \* (row - 1) + (col - 1)

        if board.tup[index] is not None:

            raise RuntimeError("Invalid move")

        board = board.make\_move(index)

        print(board.to\_pretty\_string())

        if board.terminal:

            break

        # You can train as you go, or only at the beginning.

        # Here, we train as we go, doing fifty rollouts each turn.

        for \_ in range(50):

            tree.do\_rollout(board)

        board = tree.choose(board)

        print(board.to\_pretty\_string())

        if board.terminal:

            break

def \_winning\_combos():

    for start in range(0, 9, 3):  # three in a row

        yield (start, start + 1, start + 2)

    for start in range(3):  # three in a column

        yield (start, start + 3, start + 6)

    yield (0, 4, 8)  # down-right diagonal

    yield (2, 4, 6)  # down-left diagonal

def \_find\_winner(tup):

    "Returns None if no winner, True if X wins, False if O wins"

    for i1, i2, i3 in \_winning\_combos():

        v1, v2, v3 = tup[i1], tup[i2], tup[i3]

        if False is v1 is v2 is v3:

            return False

        if True is v1 is v2 is v3:

            return True

    return None

def new\_tic\_tac\_toe\_board():

    return TicTacToeBoard(tup=(None,) \* 9, turn=True, winner=None, terminal=False)

if \_name\_ == "\_main\_":

    play\_game()