Assignment 6 - Game-Playing systems

February 27, 2024

1 Assignment 6 - Game-playing System

- Student 1 - Luca Modica
- Student 2 - Hugo Alves Henriques E Silva

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from icecream import ic


sns.set_style()
%matplotlib inline
np.random.seed(0)
```

1.1 Game setup

```
class TicTacToe:
  def __init__(self, size):
    self.size = size
    self.board = [' ' for _ in range(size*size)]
    self.current_player = 'X'

  def print_board(self):
    hline = ('------'*(self.size)).replace("-", "", self.size)
    print(hline)
    for i in range(0, self.size*self.size, self.size):
      print('|', end='')
      for j in range(self.size):
        print(' ', self.board[i + j], '|', end='')
      print('\n', end='')
      print(hline, end='')
      if i != (self.size*self.size) - self.size:
        print('\n', end='')
```

```python
def make_move(self, position):
  if position < self.size*self.size and self.board[position] == ' ':
    self.board[position] = self.current_player
    self.current_player = 'O' if self.current_player == 'X' else 'X'
  else:
    print('Invalid move. Try again.')

  return self.clone()

def get_winner(self):
  winning_combinations = []
  for i in range(self.size):
    winning_combinations.append([j for j in range(i*self.size, (i+1)*self.size)])  # rows
    winning_combinations.append([j for j in range(i, self.size*self.size, self.size)])  # columns
  winning_combinations.append([i for i in range(0, self.size*self.size, self.size+1)])  # diagonal 1
  winning_combinations.append([i for i in range(self.size-1, self.size*self.size-self.size+1, self.size-1)])  # diagonal 2

  for combination in winning_combinations:
    if all(self.board[i] == self.board[combination[0]] and self.board[i] != ' ' for i in combination):
      return self.board[combination[0]]

  if ' ' not in self.board:
    return 'Draw'

  # no winner yet
  return None

def is_game_over(self):
  return self.get_winner() in ['X', 'O', 'Draw']

def clone(self):
  clone_game = TicTacToe(self.size)
  clone_game.board = self.board.copy()
  clone_game.current_player = self.current_player
  return clone_game

def get_legal_moves(self):
  return [i for i in range(self.size*self.size) if self.board[i] == ' ']

def is_winning_move(self, move):
  clone_game = self.clone()
```

```python
    clone_game.make_move(move)
    return clone_game.get_winner() == self.current_player

  def get_random_move(self):
    # If there's a winning move, take it
    for move in self.get_legal_moves():
        if self.is_winning_move(move):
            return move

    return np.random.choice(self.get_legal_moves())

  def __repr__(self) -> str:
    return f'TicTacToe({self.size}, {self.board}, {self.current_player})'
```

```python
game = TicTacToe(3)

print('Welcome to Tic Tac Toe!')
game.make_move(0)
game.make_move(1)
game.make_move(2)
game.make_move(4)
game.make_move(3)
game.make_move(5)
game.make_move(7)
game.make_move(6)
game.make_move(8)


print('\n')
game.print_board()
print('\n')
game.get_winner()
```

```
Welcome to Tic Tac Toe!



---------------
|  X |  O |  X |
---------------
|  X |  O |  O |
---------------
|  O |  X |  X |
---------------
```

```
'Draw'
```

1.2 Game strategy setup

```python
import math
from typing import Literal

class MCTSNode:
    def __init__(self, game_state: TicTacToe, parent=None, move=None):
        self.game_state = game_state  # An instance of TicTacToe representing the state
        self.parent = parent
        self.move = move  # The move (index in the board list) that led to this state
        self.children: list[MCTSNode] = []
        self.visits = 0
        self.wins = 0

    def add_child(self, child):
        self.children.append(child)

    def select_child(self):
        return max(self.children, key=lambda child: child.uct_score())

    def expand(self):
        # Assume game_state has a method get_legal_moves() that returns all possible moves
        legal_moves = self.game_state.get_legal_moves()
        for move in legal_moves:
            # Assume applying a move returns a new game state
            new_game_state = self.game_state.clone()
            new_game_state.make_move(move)
            new_node = MCTSNode(new_game_state, self, move)
            self.add_child(new_node)

    def simulate(self) -> Literal['X', 'O', 'Draw']:
        current_state = self.game_state.clone()
        while not current_state.is_game_over():
            move = current_state.get_random_move()
            current_state.make_move(move)

        return current_state.get_winner()

    def backpropagate(self, simulation_result):
        self.visits += 1
        if simulation_result == 'Draw':
            self.wins += 0.5
        elif (simulation_result == 'X' and self.game_state.current_player == 'O') or \
```

```
            (simulation_result == 'O' and self.game_state.current_player ==␣
↪'X'):  # AI won
                self.wins += 1
            else:  # AI lost
                self.wins -= 1

            if self.parent:
                self.parent.backpropagate(simulation_result)

    def uct_score(self, C=math.sqrt(2)):
        if self.visits == 0:
            return float('inf')  # Ensure unvisited nodes are prioritized
        parent_visits = self.parent.visits if self.parent else 1  # Use 1 as a␣
↪fallback for the root
        return (self.wins / self.visits) + C * math.sqrt(math.
↪log(parent_visits) / self.visits)

    def __repr__(self) -> str:
        return f"MCTSNode(move={self.move}, visits={self.visits}, wins={self.
↪wins}, children={len(self.children)})"
```

```
def select_node(node: MCTSNode):
    """
    Traverse the tree from the given node down to a leaf node using the␣
↪select_child method.
    This involves choosing the child with the highest UCB1 score at each step.
    """
    current_node = node
    while current_node.children:
        current_node = current_node.select_child()
    return current_node

def run_mcts(root_node, iterations=100):
    for _ in range(iterations):
        # Selection step
        leaf_node = select_node(root_node)

        # Check if the game at the leaf node is not over
        if not leaf_node.game_state.is_game_over():
            # Expansion step
            leaf_node.expand()

            # Choose a child node from the newly expanded ones for simulation
            # This step assumes there's at least one child after expansion
            if leaf_node.children:
                selected_child = leaf_node.select_child()
```

```
            else:
                selected_child = leaf_node
        else:
            selected_child = leaf_node

        # Simulation step
        simulation_result = selected_child.simulate()

        # Backpropagation step
        selected_child.backpropagate(simulation_result)
```

1.3 Strategy test and evaluation

```
from IPython.display import clear_output
import timeit

def play_game(grid_size=3, mcts_iterations=10000):
    game = TicTacToe(grid_size)
    current_player = "Human"
    root_node = None
    mcts_times = [] # Store the time taken for each MCTS iteration

    while not game.get_winner():
        clear_output(wait=True)

        if current_player == "Human":
            print("Current Board State:")
            game.print_board()
            print('\n')
            print("Your turn, Human!")
            is_valid_move = False
            while not is_valid_move:
                move = int(input(f"Enter your move (0-{grid_size**2}): "))
                if move not in game.get_legal_moves():
                    print("Invalid move. Try again.")
                else:
                    game.make_move(move)
                    is_valid_move = True

            current_player = "AI"
        else:
            print("AI's turn. AI is calculating its move...\n")
            root_node = MCTSNode(game.clone(), parent=None)

            start_time = timeit.default_timer()
            run_mcts(root_node, iterations=mcts_iterations)
```

```python
            end_time = timeit.default_timer()
            mcts_time = end_time - start_time
            mcts_times.append(mcts_time)


            best_node = max(root_node.children, key=lambda child: child.wins/
 ↪child.visits)
            game.make_move(best_node.move)

            current_player = "Human"

        if game.get_winner():
            print('\n')
            game.print_board()
            print('\n')
            if game.get_winner() == 'Draw':
                print("It's a draw!")
            else:
                print(f"{game.get_winner()} wins!")

            avg_mcts_time = sum(mcts_times) / len(mcts_times)
            print(f"Average MCTS computational time: {avg_mcts_time:.6f}␣
 ↪seconds")
            break



play_game(grid_size=5, mcts_iterations=10000)
```

```
Current Board State:
-------------------------
|  X |  X |  X |  O |  O |
-------------------------
|  O |  O |  X |  O |  X |
-------------------------
|  O |  O |  O |  X |  X |
-------------------------
|  O |  O |    |  O |  X |
-------------------------
|  X |  X |  O |  X |  X |
-------------------------

Your turn, Human!



-------------------------
|  X |  X |  X |  O |  O |
```

```
-------------------------
| O | O | X | O | X |
-------------------------
| O | O | O | X | X |
-------------------------
| O | O | X | O | X |
-------------------------
| X | X | O | X | X |
-------------------------

It's a draw!
Average MCTS computational time: 7.472431 seconds
```