Assignment 3

Michele Vannucci (2819493)
Arturo Abril (2813498)

December 2024

The code used to obtain the results of this report can be found here[1].

Abstract

In this work we code the connect four game and use Monte Carlo Tree Search (MCTS) to play against a random agent. First, we present the basic implementation, in which the game starts with columns 4 and 6 of the board empty and the rest filled as described in the assignment description. The MCTS will choose the best actions leveraging the Upper Confidence Bound (UCB) rule to balance exploration and explotation. Later we describe how our implementation is already able to start from any initial configuration of the game board (including and empty one), and discuss how the performance of the MCTS can be improved if we run the algorithm for several times and average the action values obtained for the root node to select a move.

Game environment

To tackle the problem we first had to find a way to encode the state of the game board, how to update it and the termination conditions. The most important code for this can be found in the `GameBoard` class in the Appendix A.1.1. To encode the state of the game we use a matrix of the right dimensions (6 rows and 7 columns) with zeros for empty positions and ones (twos) for player one (two) discs. To update the state of the game we just update the values of the matrix as desired, making sure the move we are making is possible (see `GameBoard.step()` and `GameBoard.play()` in the implementation). For the termination condition we check, after every update of the board, whether any of the two following conditions is true: one of the two managed to *connect four or* the board is full. If any of these is true, the game is finished. The termination functions can be found in the Appendix A.1.2

MCTS algorithm

We build a directed acyclic graph (DAG) to perform the MCTS. The nodes in this graph will have a (frozen) copy of the corresponding game state. The edges represent the actions taken by player one. The (random) moves of player 2 are considered part of the transitions. Under this conditions, when a node is expanded choosing a certain action (a certain column to play on the board), a new child node is created and its state is the result of the action chosen and the random move of player 2. Both moves are performed sequentially on the game state of the parent node and the updated game is the state stored in the child.

The following algorithm is repeated for a fixed number of times: Starting from the **root** node, we start selecting child nodes recursively until we reach a node that is either a **leaf** or a **terminal** node (note that root itself can be leaf or terminal). A leaf node is defined as a node that hasn't been expanded to all its possible children, this is, the game state stored in this node has more possible actions than the number of children that the node points to. A terminal node is a node that cannot be expanded anymore (the game stored in it is a finished game). When we reach a leaf node, we **expand** it choosing one of the possible actions at random[2]. We take the game state of the leaf node and update it with the chosen action, and play a random move for player 2 (if possible at all) at random. From this expanded node, we **rollout** random game rounds until finishing a game. We **backpropagate** the result of this random game up the tree until the root node. This updates consist of both adding up the result of the game to the existing value of the node *and* updating the number of visits to each node.

[1] https://github.com/michelexyz/RL-assignments/tree/main/Assignment3
[2] Note that *possible actions* will be those possible given the game state and the already existing edges (actions) from that node.

We have skipped the **selection** step details in purpose to discuss them in more detail here. Every time we need to select a child from a parent node we select the children with the highest UCB value, defined as:

$$\text{UCB value} = \bar{x} + c\sqrt{\frac{\log N}{n}}$$

where $\bar{x}$ is the mean value of the node, N is the number of visits of the parent node and n the number of visits of the node. The first term correspond to exploitation and the second term to exploration. The latter goes to zero as the number of visits of both parent and child grow, which makes sense since the more times you have visited a node the more certain you expect to be about its value. See Appendix A.1.3 for implementation. We now give some details about the algorithm that runs the MCTS.

Nodes Nodes are instances of the `Node` class (see A.1.5). Each node has as attributes the parent node, the action from which it was created, the number of times the node was visited and the total reward received so far. It has two important properties: `Node.is_leaf` and `Node.is_terminal` that return the right boolean values depending on the game state stored in the node and the node's existing children. The `Node.select()` method returns the children node with highest UCB value.`Node.update_value()` adds up the passed value to the current total value of the node and adds one to the visits counter.

Tree Seach The MCTS is performed by the `MCTS` class. The implementation can be found in the Appendix A.1.4. `MCTS.select()` calls the node's `select()` method recursively starting from the root, until a leaf or terminal node is reached. The `MCTS.expand(parent: Node)` method picks random actions, out of the available ones, from the selected node. This adds a new node to the tree storing the action picked, the parent and the resulting game state. Finally, we roll-out playing random moves until we reach the termination condition and `MCTS.update(leaf=leaf, value=reward))` is used to recursively go backward in the tree following the chain of parent nodes until the root is reached, updating the node's total reward and number of visits. This method returns a dictionary with the action values of the root node.

Playing the game

In Figure 1 we can observe a sequence of game states that are produced picking the action that yields the highest mean value out of the root node's children . The tree is recreated for every turn of the gree player with the new game state as root and the MCTS is run through $2^{11} = 2048$ iterations. This will certainly exhaust the tree as the gree player can take up to 6 binary decisions considering the given initial state, additionally the possible game states are $\lesssim 2^{11}$ considering that winning states terminate the game earlier. On the top part of each of the game state graphs in fig. 1 we can observe the expected q-values computed by the MCTS algorithm for the available actions at the root.

Convergence

We store frozen copies of the game states upon creation of child nodes. We expect to achieve a better performance if, every time a tree pass is performed, we would recalculate the game state from root during transversal. Since this implementation is way harder, we do the following: we do 25 independent runs of the MCTS algorithm and store, for each possible action to be taken from root, the 25 obtained values. The action with the highest mean value over this 25 independent runs is considered optimal. The values from the independent runs for each action (from the initial configuration) can be seen in fig. 2. The mean value for action 4 (column 4 in the board) is 0.70 and for action 6 is 0.52. We conclude that action 4 is the optimal first action to be taken as the first player. Of course, even though this numbers are not directly winning probabilities, a higher value for a given action means that we have higher probabilities of winning a game if we take such action. Repeating this process for subsequent actions yields as best moves: $4 \to 6 \to 4$. A quick visual inspection of the game confirms this.

Bonus

The code implementation can be used to play a game from an empty board. One just have to initialize the `Game` class without any given initial grid (just `Game()`) and a matrix full of zeros will be used as initial state. See the GitHub repository for implementation details.

A Appendix

A.1 Implementation

A.1.1 The Game Board

The following is the `GameBoard` class that implements the game logic, updates the board after every action and defines the legal moves. Only the core methods are displayed

```
class GameBoard:
    nrows: int
    ncols: int
    _grid: np.ndarray

    @property
    def available_actions(self) -> Set[int]:
        return available_actions(self._grid)

    def game_result(self) -> GameResult:
        if not self.is_finished:
            raise GameNotFinishedError

        match self.check_winner():
            case PlayerType.US:
                return GameResult.WIN
            case PlayerType.OPPONENT:
                return GameResult.LOSE
            case None:
                return GameResult.DRAW

    def check_winner(self) -> Optional[PlayerType]:
        return check_winner(grid=self._grid)

    def step(self, action: int, player: PlayerType) -> None:
        action = self.validate_action(action)
        row = np.max(np.where(self._grid[:, action] == 0))
        self._grid[row, action] = player

    def play(self, first_action: int, second_action: Optional[int] = None) -> None:
        # First play
        self.step(action=first_action, player=PlayerType.US)

        # Second play
        second_action = (
            second_action
            if second_action is not None
            else random.choice(list(self.available_actions))
        )
        if not self.is_finished:
            self.step(action=second_action, player=PlayerType.OPPONENT)
        return

    def rollout(self) -> GameResult:
        while not self.is_finished:
            self.play(first_action=random.choice(list(self.available_actions)))
        return self.game_result()
```

A.1.2 Helper functions

More in detail the `check_winner` and other relevant common functions:

```python
def check_winner(grid: np.ndarray) -> Optional[PlayerType]:

    for player in [PlayerType.US, PlayerType.OPPONENT]:
        # Horizontal check
        for row in range(grid.shape[0]):
            for col in range(grid.shape[1] - 3):
                if np.all([grid[row, col + i] == player for i in range(4)]):
                    return player

        # Vertical check
        for col in range(grid.shape[1]):
            for row in range(grid.shape[0] - 3):
                if np.all([grid[row + i, col] == player for i in range(4)]):
                    return player

        # Diagonal down (\) check
        for row in range(grid.shape[0] - 3):
            for col in range(grid.shape[1] - 3):
                if np.all([grid[row + i, col + i] == player for i in range(4)]):
                    return player

        # Diagonal up (/) check
        for row in range(3, grid.shape[0]):
            for col in range(grid.shape[1] - 3):
                if np.all([grid[row - i, col + i] == player for i in range(4)]):
                    return player
    return

def available_actions(grid: np.ndarray) -> Set[int]:
    return set(np.where(grid == 0)[1])

def is_game_finished(grid: np.ndarray) -> bool:
    return (check_winner(grid) is not None) or (0 not in grid)
```

A.1.3 UCB selection

```
class UCB(SelectionStrategy):
    C = 2
    @staticmethod
    def strategy(node: Node) -> float:
        if not node.n_visits > 0:
            raise ZeroDivisionError
        return node.mean + (
            UCB.C * math.sqrt(math.log(node.parent.n_visits / node.n_visits))
        )
```

A.1.4 The MCTS tree search

This is the main method for the MCTS. For more details about the `MCTS` class, see A.1.6

```
class MCTS:
    ...
    def run(self, game_state, strategy):

        self.game = GameBoard.from_grid(game_state)
        self.root = Node.from_root(state=self.game.snapshot())

        for _ in range(self.maxiter):
            # Select a node **starting from root** (see MCTS.select())
            parent = self.select(s=strategy)

            # Expand it (or just return the same node if terminal)
            leaf = self.expand(parent=parent)  # Can be terminal

            # Play random rounds
            reward = self.game.rollout()

            # Backprop
            self.update(leaf=leaf, value=reward)

        return self.qvalues
```

A.1.5 The nodes

Below is the implementation for the `Node` class. Only the most core methods were reported

```python
class Node:

    game_state: np.ndarray
    parent: Optional[Node]
    from_action: Optional[int]

    children: Set[Node]
    depth: int
    n_visits: int
    value: float

    def __init__(
        self,
        game_state: np.ndarray,
        parent: Optional[Node] = None,
        from_action: Optional[int] = None,
    ) -> None:
        self.game_state = game_state
        self.parent = parent
        self.from_action = from_action

        self.children = set()
        self.depth = parent.depth + 1 if parent else 0
        self.n_visits = 0
        self.value = 0.0

    def select(self, strategy: SelectionStrategy) -> Node:
        if (
            self.is_leaf or self.is_terminal
        ):  # You only select when you have all your children
            return self
        assert self.children
        return strategy.select(nodes=self.children)

    def add_child(self, child: Node) -> None:
        # Check that a parent doesn't have two kids with the same action
        assert child.from_action not in [ch.from_action for ch in self.children]

        self.children.add(child)
        # Check that we don't have more children than permitted
        assert not (len(self.children) > self.game_state.shape[1])

    def update_value(self, value: int) -> None:
        self.value += value
        self.n_visits += 1
```

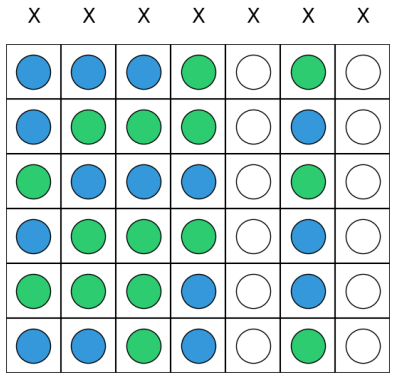
A.1.6 The MCTS

The core of our algorithm is the `MCTS` in the following we can observe its implementation.

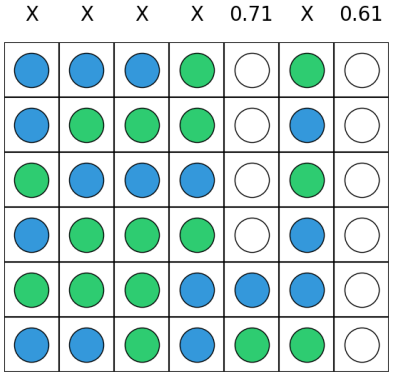
```
class MCTS:

root: Node | None
game: GameBoard | None
maxiter: int

def select(self, s: SelectionStrategy) -> Node:
    """Returns either a LEAF or TERMINAL node"""
    # Start always from the top
    node = self.root.select(strategy=s)
    # Iterate through nodes until reaching a leaf / terminal node
    while not (node.is_leaf or node.is_terminal):
        node = node.select(strategy=s)

    # VERY important: set game state to start playing from
    self.game.set_state(node.game_state)

    return node

def expand(self, parent: Node) -> Node:
    """Expands LEAF node. Make sure TERMINAL nodes are not expanded
    A LEAF node always has available actions to take, by definition
    """
    # Cancel expanding if node is terminal
    if parent.is_terminal:
        return parent

    # Choose random action (from available) to expand parent and play one round
    action = random.choice(list(parent.available_actions))
    self.game.play(first_action=action)

    # Create the child with the results from the just played game
    child = Node.from_parent(
        state=self.game.snapshot(), parent=parent, action=action
    )
    parent.add_child(child)

    return child

def update(self, leaf: Node, value: int) -> None:
    # Backpropagate the value up until root
    leaf.update_value(value)
    parent = leaf.parent

    while parent:
        parent.update_value(value)
        parent = parent.parent  # This will be `None` for root, so exit loop
```

The `MCTS.run()` method has already been presented.

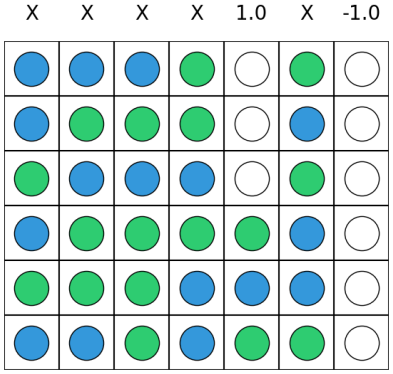
A.2 Playing the game



(a) Empty game state.



(b) The second game step.



(c) Green player wins.

Figure 1: The game steps of a game played by selecting the highest-value action of the MCTS tree for the green player and playing randomly for the opponent. The top part of the figure shows the expected Q-value for the given action-column. The moves of the shown game are: $4 \rightarrow 4 \rightarrow 4$

A.3 Convergence

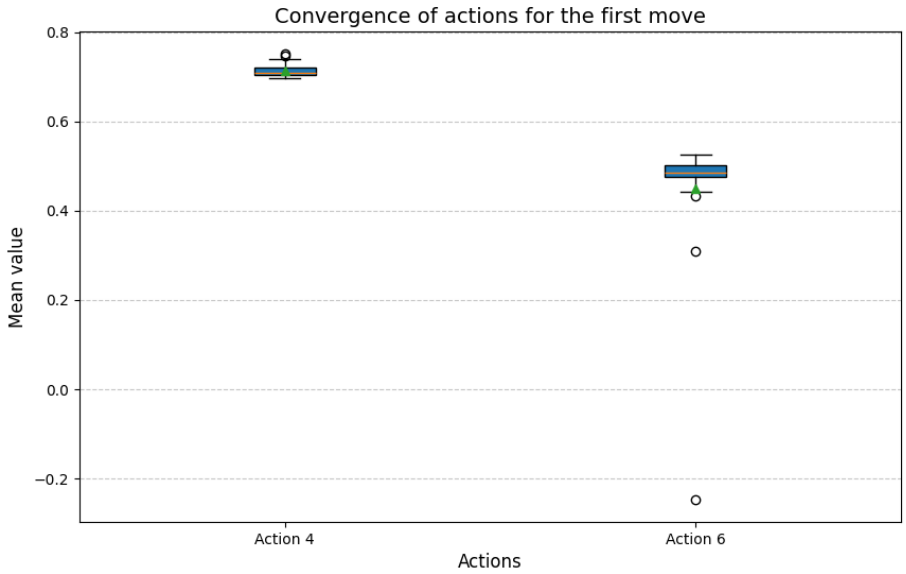


Figure 2: Mean values for each action starting for the first round. The mean values are 0.70 and 0.52 respectively.