Dynamic programming
Assignment 3

By

Victor Retamal Guiberteau, 2741820

Daniël Boelman, 2639457

December 7, 2021

Contents

1 Tic Tac Toe

The objective for this project is, given a restricted position where our agent ”X” can potentially beat the game, to find the optimal policy for this agent to play moves and beat the game. The optimal policy will be learned using Montecarlo Tree Search.

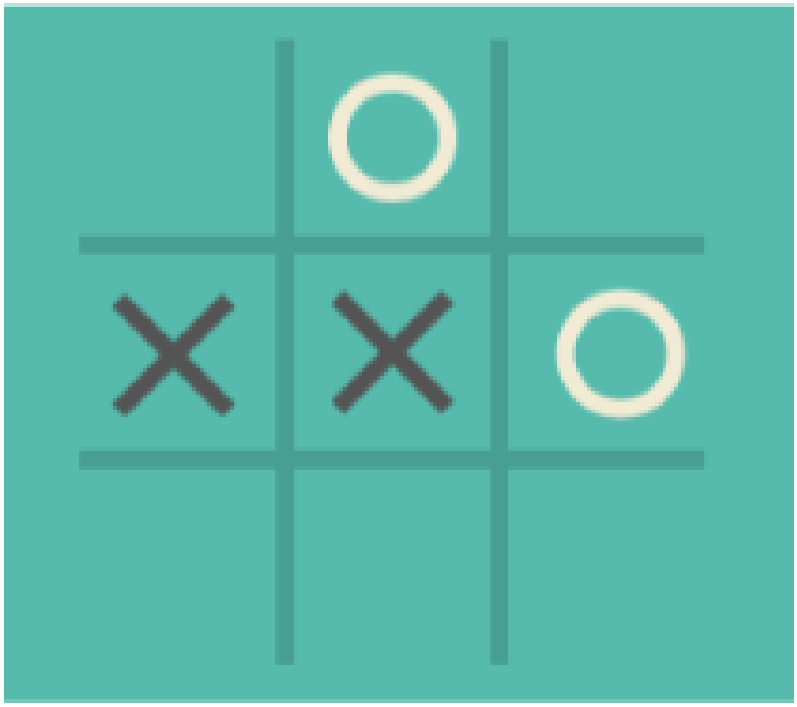

Figure 1: Restricted Initial position

Given the game’s simplicity, the whole search space could be covered without the need for excessive computational power. However, a clever solution is applied using Montecarlo Tree search.

The first part of the experiment assumes the opponent to take random actions to train our agent. In the second part, the opponent agent always performs the optimal action given the situation thanks to a Minimax algorithm. This second approach allows our agent to fine-tune the policy.

2 Implementation to beat Tic Tac Toe

The implementation to create an intelligent agent to beat Tic Tac Toe, starts with the representation of the game as a tree. Given the tree structure used, we can represent the game states in nodes and keep track of different events happening.

Different events and values are stored in the nodes. Number of visits, score and the expected value for the action taken to end in this particular node.

After selecting the representation, the next step is to implement the Montecarlo search tree. For this game, we decided to make some variations in the original implementation. Given the restricted position, we end in a search space with less than 128 positions. With the original algorithm, the sampling of the whole tree will be possible, and the stochastic will be lost, resulting in an agent who learnt the whole game by playing every possible position. To avoid this deterministic search, we restricted the expansion of the nodes to two levels of depth, that means, our agent select an action, the opponent selects an action, and a simulation is done.

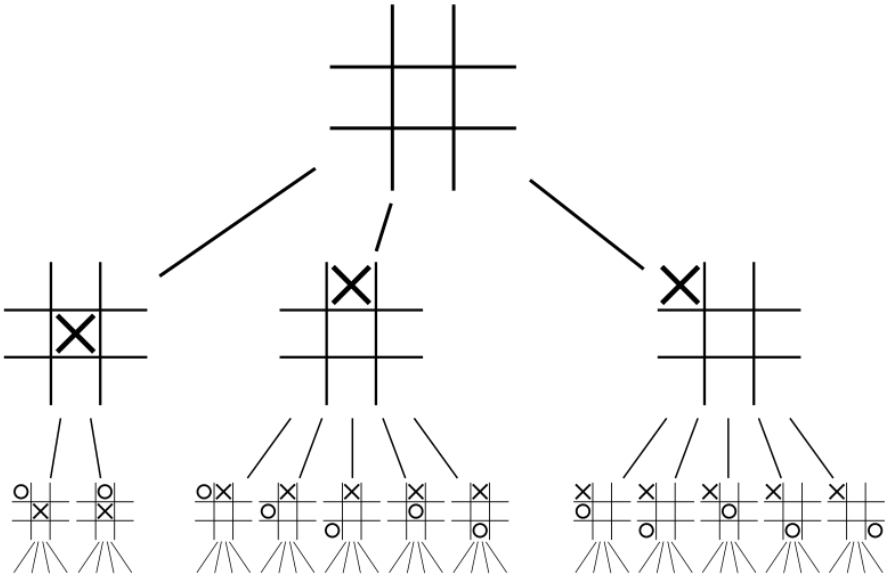

Figure 2: Representation of the game using a tree
https://commons.wikimedia.org/wiki/File:Tic-tac-toe-game-tree.svg

In order to calculate the optimal policy, or a good approximation, expected rewards are calculated per available initial move. This calculation is done through iteration until convergence. The convergence condition is:

$$Q_t - Q_{t-1} \leq \varepsilon$$

where ε is a really small quantity. The calculation of the expected rewards is:

$$Q(x,a) = \frac{\sum_{x'}(r + \max_{a' \in A} Q(x',a)}{n(x,a)})$$

That means, the new value is equal to the immediate reward plus the estimate of optimal future value. We are using the information obtained in the Montecarlo Sampling to update our previous knowledge and adjust the actions in the policy.

2.1 Opponent plays at random

For the first experiment, the action selected by the opponent will always be randomly selected. Given this restriction, the learning process from the Montecarlo sampling could not be optimal. testing our agent against an opponent playing optimal moves, could result in not and optimal performance. The pseudo code for the calculation of the expected rewards is:

```
findNextMove():
-> Initializes root node
    While not converge:
        -> Select one child
        -> Randomly explore the child (traverse until leaf node)
        -> Get the result of the leaf node
        -> Store current Q_value in Q_value_prev
        -> Update the Q_values with Back propagation.
        if Q_value - Q_value_pre < epsilon:
            converge = True # end of the iteration
```

The initial actions are mapped from top left to bottom right being [(0,0), (0,2), (2,0), (2,1), (2,2)] if we see the grid as a 3x3 matrix The results for the firs part of the experiment are:

Q-values for the initial moves actions: $[1.0, 0.875, 1.0, 0.375, 0.875]$

The win probabilities for these possible actions: $[0.75, 0.66, 0.72, 0.30, 0.66]$

2.2 Opponent plays optimally

The second experiment train and test our agent against an opponent performing optimal actions. To simulate this behaviour we utilized Maximin

2.2.1 Maximin

Maximin is a decision rule utilized to maximize the gains in the worst scenario. In this experiment, the worst scenario for our agent, is an opponent playing optimal moves. The formal definition of the rule is:

$$Q_t(s,a) = \max_{a_{p1} \in A_{p1}} \min_{a_{p2} \in A_{p2}} Q_{t+1}(s')$$

```
findNextMove():
-> Initializes root node
    While not converge:
        -> Select one child
        -> Randomly explore the child (traverse until leaf node)
        -> Get the result of the leaf node
        -> Store current Q_value in Q_value_prev
        -> Update the Q_values with Back propagation.
        Maximin(player):
            if player == "X":
                -> new_q value = get max Q_value
            elif player == "O":
                -> new_q_value = get min Q_value
        if Q_value - Q_value_pre < epsilon:
            converge = True # end of the iteration
```

The expected value is now selected taking into account the best move for the opponent. Since the goal of our agent is to maximize the reward, our opponent is trying to minimize it.

3 Conclusions

The Montecarlo Sampling is an excellent approach to estimate an optimal policy utilizing a low computational budget. In this experiment, we could see how the optimal policy, which was first randomly generated, is updated after every iteration where we sample with Montecarlo, adding new knowledge about future states in the game given a specific initial action. Given the restricted initial case, the two implemented methods have similar final policies because some moves are strictly wining for X player if it play optimal moves. After

approximately 100 iterations of Montecarlo, we found the agent learnt those moves and could generate an optimal policy for the first movement. tested against the optimal enemy, at first, both behave randomly, but after a specific time, the opponent learns to take advantage of the sub-optimal moves of player ”X”, that is, trying to end the game in a draw, but the ”X” agent soon learn to optimize it moves to avoid this draws.