BITS Pilani, K. K. Birla Goa Campus

Department of Computer Science & Information Systems

CS F407 - Artificial Intelligence: Programming Assignment 2

Nimish Wadekar

2019A7PS1004G

December 4, 2021

Contents

1 Abstract

This report contains details on the methodology and thought processes that were used to develop two competent reinforcement learning algorithms, *Monte Carlo Tree Search (MCTS)* and *Q-Learning*, to play the Connect4 board game. Both algorithms were implemented from scratch using Python3.

This report is divided into two main sections:

- The first section contains detailed information about the implementation of the **Monte Carlo Search Tree** algorithm, and analysis of multiple games being played between two MCTS algorithms while varying various parameters of the program, with the two algorithms themselves differing only in the number of playouts they perform before taking an action.

- The second section contains information about the methodology and development of the **Q-Learning** algorithm. The implemented Q-learning algorithm is a modified Q-learning algorithm that uses the concept of *afterstates*, instead of state-value pairs, to learn and store Q-values.

2 Monte Carlo Search Tree

2.1 Background

Monte Carlo Tree Search is a heuristic search algorithm. It uses the Monte Carlo method of estimating value functions. Monte Carlo methods are algorithms in which the task is divided into episodes. The algorithm then repeatedly performs actions in the episodes, and the value function is updated at the end of an episode based on the reward received.

A game tree is a tree of states of a game, where each state represents the state of the game at some point in time. Game trees can be traversed according to the actions taken at any given state, and they also be searched to attempt predicting the future states of a game.

MCTS is an algorithm that combines the Monte Carlo method with the game tree search. It tries to analyse the most promising actions, and expands the game tree in the direction of said actions. The Monte Carlo method is applied here as follows: before choosing a move, the algorithm performs several simulations, each simulation lasting till a game-ending state is found, called *playouts*, from its current state. Each playout updates the state values of all states that were used to reach that playout's terminating state. The actual move is then chosen based on these state values.

Each round of MCTS consists of the following four steps:

1. **Selection**:
   Starting from the root node R, traverse the game tree until a leaf node L is reached. A leaf node is a node that has not been explored, i.e. a node that has not gone through the *Expansion* phase of MCTS. The choosing of the nodes in the traversal is done using the UCT (Upper Confidence Bound applied to trees) formula, which balances exploration with exploitation. The formula is:

$$UCT(node) = \frac{node.rewards}{node.visits} + C\sqrt{\frac{\ln\left(node.parent.visits\right)}{node.visits}} \tag{2.1}$$

where $node$ represents a node of the game tree, $node.rewards$ represents the state value modified during a playout, $node.visits$ represents the number of times $node$ has been visited during playouts, $node.parent$ represents the parent node of $node$, and C represents the exploration parameter.

The first part of the formula corresponds to exploitation; it increases with increase in the value of a node. The second part corresponds to exploration; it increases when a particular node is visited less compared to its parent node.

2. **Expansion**:
   From node L obtained from the *Selection* phase, create child nodes representing possible immediate next states for node L. This does not occur if node L is a terminating node.

3. **Simulation**:
   Start from node L and make uniformly random moves until a terminating state's node is reached.

4. **Backpropagation**:
   Use the reward obtained at the terminating node of the *Simulation* phase to update all the nodes traversed from the root node R to the leaf node L. The reward is added to each node's reward value, along with incrementing the counter that tracks the number of visits to the node.

After performing several playouts, a move is chosen greedily based on the values of the immediate next nodes.

2.2 Method

This subsection will explain the thought process that was followed while developing and implementing the MCTS algorithm.

2.2.1 Connect4 Board

- The game used a board of 6 rows and 5 columns.
- The Connect4 board state was represented as a double-dimensional array, with each element's possible values being
  - 0 - Empty cell
  - 1 - Player 1
  - -1 - Player 2
- Since each element has only 3 possible values, the entire board can be treated as a number in a ternary number system, and can thus be encoded into a single number. Assuming n is the number of rows and m is the number of columns in the board, the formula to encode a board is:

$$Encode(board) = \sum_{i=0}^{n} \sum_{j=0}^{m} Val(board[i][j]) * 3^{mi+j} \tag{2.2}$$

$$\text{where, } Val(e) = \begin{cases} 2 & \text{, if } e = -1 \\ e & \text{, otherwise} \end{cases} \tag{2.3}$$

  The maximum value of an encoded 6×5 board, including unreachable states, is $2 \cdot 3^{29} + 2 \cdot 3^{28} + 2 \cdot 3^{27} + \cdots + 2 \cdot 3^{0}$, which is equal to $3^{30} - 1$. This value ($\approx 2.06 \times 10^{14}$) is lesser than $2^{63} - 1$ ($\approx 9.2 \times 10^{18}$, which is the maximum signed number that can be stored in one register in modern 64-bit architecture. Thus, overflows will never happen.

2.2.2 MCTS

- Two versions of MCTS, MC_{40} and MC_{200} (where MC_n denotes an MCTS algorithm that performs n playouts before making a move), were played against each other for 100 games.
- The rewards decided for each state were:
  - +2 for a win state.
  - -2 for a loss state.
  - +1 for a draw state.
  - 0 for non-terminating state.
- While selecting a child node for making a move, the nodes that were left unexplored during playouts are not chosen.
- The UCT exploration parameter was set to be root 2 based on observations found from games between MC40 and MC200. $C = \sqrt{2}$ allows MC200 to win the most times (73% wins, 9% losses, and 18% draws).

  Other values of C did not give MC200 the advantage $C = \sqrt{2}$ did. The following are some statistics gathered by varying the value of C:

  - $C = 1.0$: 65% wins, 15% losses, 20% draws.
  - $C = 1.8$: 68% wins, 18% losses, 14% draws

  The reason for this ideal value could be that this is the ideal C value when held in the context of the rewards ($+2$, -2, $+1$); i.e. the value of $\sqrt{2}$ is comparable to the magnitudes of the rewards that are given. Figure 2.1 shows how varying the value of C changes how good MC_{200} performs against MC_{40}.

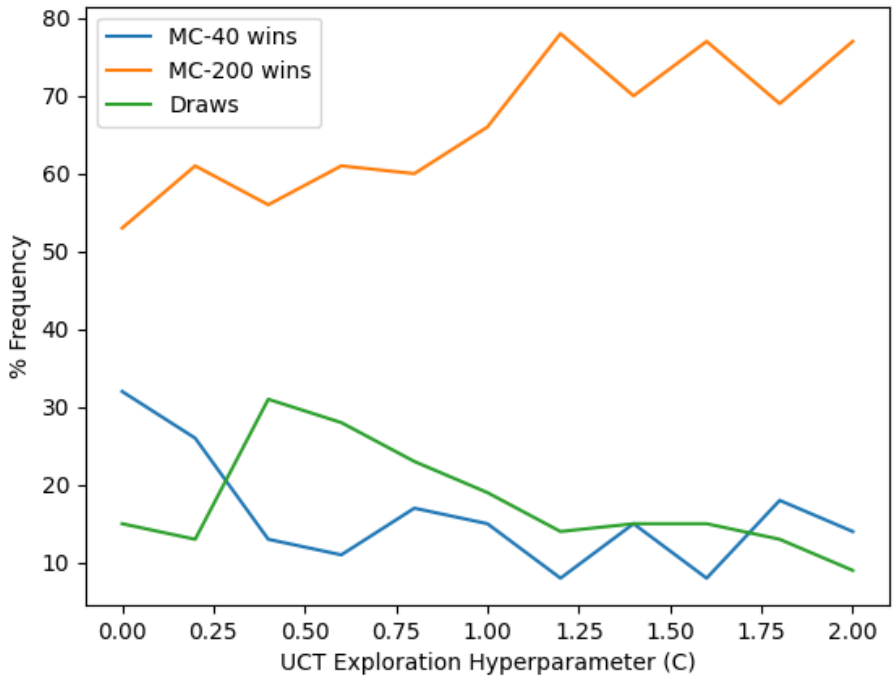

Figure 2.1: Effect of variation of C on the advantage given to MC_{200} over MC_{40}.

3 Q-Learning

3.1 Background

Q-learning is a reinforcement learning algorithm for learning the value of a $state - actionpair$. It is similar to the Monte Carlo method for estimating values but has one key difference. Unlike the Monte Carlo method, where the values are updated at the end of each episode, a Q-learning algorithm performs update operations after each action that the agent takes.

The update rule for Q-learning is as follows:

$$Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot max_a Q(s_{t+1}, a)) \tag{3.1}$$

$Q(s_t, a_t)$ represents the value of a state-action pair at time t. The learning rate (α) determines the extent to which new information that is acquired after an action overrides old existing information. The discount factor (γ) determines how important future rewards are. The lower the discount factor, the lesser rewards that are farther into the future affect the current state's value.

3.2 Method

- The Connect4 board was simplified to be of 4 rows and 5 columns, to decrease the time it takes to converge to optimal values.
- The Q-learning algorithm implemented comprises the following steps for each episode, until a terminal state is reached:
  - Choose action A from state S using an ϵ-greedy policy derived from Q.
  - Obtain reward R and next state S' after taking action A.
  - Update the value of $Q(S, A)$ using the update rule.
  - Move to state S'.

- The Q-learning algorithm was trained against the previously implemented MCTS algorithm, with MCTS' playouts ranging from 0 to 25, both inclusive.
- The Q-table was initially implemented as a table of state-action pairs mapping to Q-values.
- Since Connect4 is a two-player game, the Q-learning agent cannot consider the state of the game after performing a move as the next state S' to be considered in the algorithm. This is because at such a state S', the opponent will perform a move, and as such will not update this agent's Q-table. To resolve this, the opponent, and any action it performs, is considered a stochastic part of the environment. The next state S' is then considered to be the game state that occurs after the opponent has finished making its move, and the update rule is applied accordingly.

  Thus, $max_a Q(s_{t+1}, a)$ in the Q-value update rule denotes the maximum Q-value of all possible actions from the state reached after the opponent has made its move.
- The reward provided to the agent were:
  - $+2$ for a win
  - -2 for a loss
  - 0.2 for a draw
  - -0.01 for all non-terminating states.

  There is a slight negative reward for each state the game moves to, that is not a terminal state. This is to disincentivise the agent from taking too many moves to finish the game and push it to attempt to win with as few moves as possible.
- The concept of *afterstates* was considered and then implemented for the Q-Learning agent. An afterstate is a state that is reached after the agent takes

an action. For example, if the agent reaches state S' immediately after taking action A at state S, state S' is the afterstate of the state-action pair (S, A). Note that this state is not the next state the agent is in control, but the one where the opponent does. Consider the following situation:

Suppose the agent reaches a game state in two possible ways:

$$(S_1, A_1) \rightarrow S_3, \text{ and}$$

$$(S_2, A_2) \rightarrow S_3$$

In the traditional state-action value representation, since the state S_3 was reached from two different states using two different actions, there would be two distinct Q-values in the Q-table for reaching the same state.

In a game like Connect4, it does not matter how a particular state is reached, but rather how probable it is to reach a win state from that particular state. Thus, afterstates can be used to exploit that fact.

- The table was then modified to map an afterstate to a Q-value, and the new update rule was as follows:

$$Q(s_t) \leftarrow (1 - \alpha) \cdot Q(s_t) + \alpha \cdot (r_t + \gamma \cdot max\, Q(s_{t+1})) \tag{3.2}$$

  Here, s_t represents an afterstate, and $max\, Q(s_{t+1})$ represents the maximum Q-value of all possible afterstates that can be reached after taking an action from the state reached after the opponent picks its move. This change ensured that one state had only one Q-value associated with it, and any updates to that Q-value would be propagated to all the paths that contain that state.

- The Q-learning algorithm converged fastest to the optimal values for the following values of parameters:
  - $\alpha = 0.1$
  - $\gamma = 0.8$

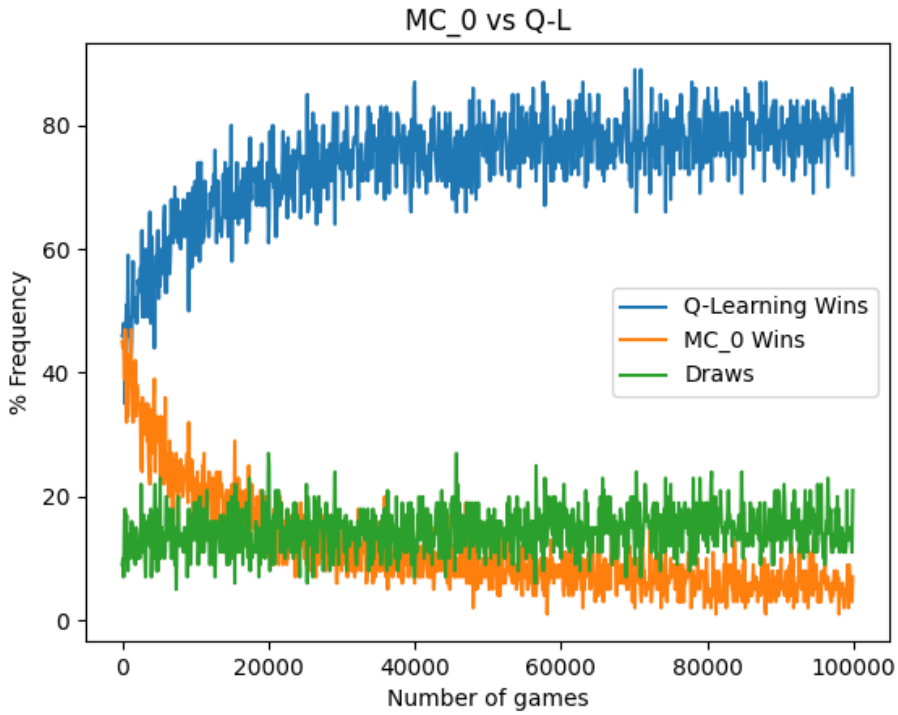

Figure 3.1: Performance of a newly created Q-Learning agent against the MC_0 algorithm, using optimal parameter values.

  - $\epsilon = 0.05$, with it being multiplied by a factor of 0.99 periodically while training.

  Modifying the above parameters from their values led to the algorithm taking much longer to converge to optimal values, and in some cases, not converge at all.

- The Q-learning algorithm was trained to be competent against a range of MC_n algorithms by training it against different MC_n algorithms incrementally; i.e. initially trained against MC_0, then against MC_5, then MC_{10} and so on until MC_{25}. Since it was trained against MC_{25} the last, those values were the

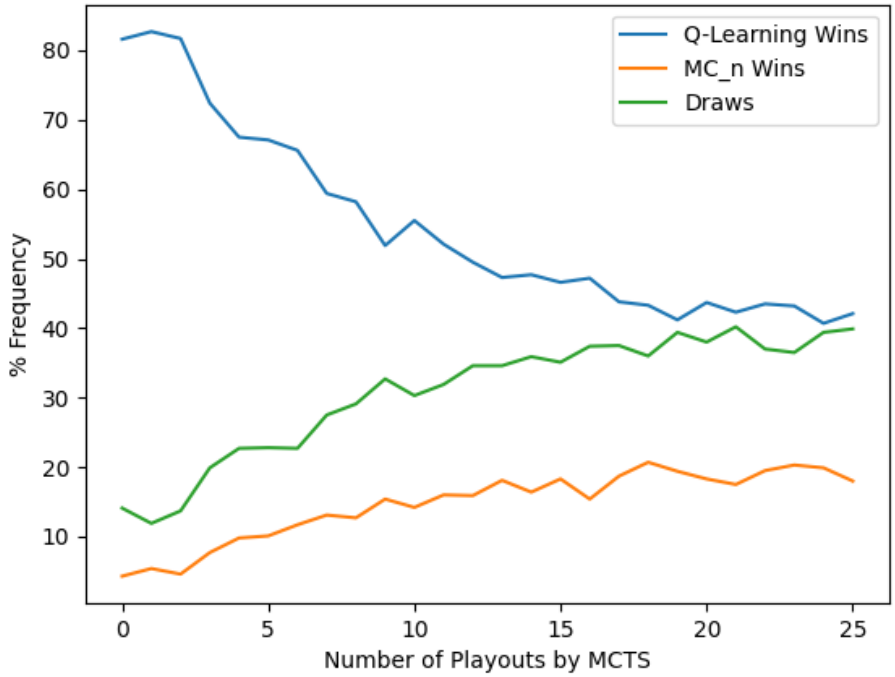

Figure 3.2: Variation of n against a trained Q-L agent.

most prevalent, while also having experience against other MC_n algorithms. Training against MC_{25} also taught it against the best moves it can face in the range of MC_0 to MC_{25}, thus improving its skill level much more than training against lower MC_n algorithms could.

Figure 3.3: Games between a trained Q-L agent and an untrained Q-L agent.