Michael Moy (mmoy92)

CS4804 Final Project

Monte Carlo Connect Four Learner

I formulated this reinforcement learning problem as follows:

States – States are represented by snapshots of the current game board (a row by column grid) populated with either x's (the learner) or o's (opposing player, the environment). Empty cells are symbolized by underscores "_" for spacing reasons. The initial state contains a row by column grid (set by the user) containing only empty cells.

- An 'X_WINNER' state is a terminal state in which there are 4 (or an arbitrarily set number of) 'X' tokens adjacent to each other in the horizontal, vertical, or diagonal directions.
- An 'O_WINNER' state is a terminal state in which there are 4 'O' tokens adjacent to each other in the horizontal, vertical, or diagonal directions.
- A 'STALEMATE' state is a terminal state that is not an 'X_WINNER' or 'O_WINNER' state and has no blank '_' cells remaining.
- An 'IN_PROGRESS' state is state that is not an 'X_WINNER', 'O_WINNER', or 'STALEMATE' state.

There can be huge numbers of states, depending on the board dimensions. Therefore, transitions are defined as the transition of a state with an odd number of placed tokens (odd state) into a state with an even number of placed tokens (even state). Odd and even states alternate as the first and second players take actions.

Actions – Actions are represented by unsigned integers from 0 to the total number of columns minus one. A player can only place their token ('x' or 'o') during their turn, and in a column that is not full. Choosing a column will place the token "at the top" of the row in that column.

Rewards – The goal of the game is to reach the WINNER state for your perspective token, before the opponent can. The reward values are defined in the reward() method inside the Board.as class.

- (200 pts) The move causes the board to enter the X_WINNER state.
- (100 pts) The move blocks the opponent from entering their O_WINNER state.
- (0 pts) The move results in neither X_WINNER or O_WINNER state.

In addition, the total value for a state-action move is further calculated in the calculateReward() method inside the Main.as class, where gamma is 0.5:

- calculateReward(action) = (Immediate_action reward) + gamma * (Following_action reward)

The Program – The learner performs its algorithm in three stages, which repeat forever until paused by the user:

1. **Episode Generation** – Starts with a cleared state and plays out an episode by following the current policy. If there is no matching policy for the current state in the game, one of two choices are made:

a. If TD (temporal difference) is enabled, the method placeBest() is used to test each of the current possible moves for reward values. The move yielding the highest reward is then performed.
b. If TD is disabled, the method placeRandom() is used to select and perform a random valid move.

Every state-action pairing is pushed into an episode array to be analyzed in the next stage.

2. **Episode Evaluation** – Starts from the beginning of the episode that was just played. For each state-action pairing in the episode, the discounted q-value is calculated and averaged into a new array called “returns”. The returns list keeps a running total of all state-action pairings occurred so far, and is updated whenever a new q-value for a move is discovered.
3. **Policy Evaluation** – Starts from the beginning of the episode again. For each state that occurred in the episode, a lookup is performed on the returns list. A short list of matching returns for the state is built, and a new policy is determined:
    a. Exploitation – 90% of the time, the action correlated with the return entry with the highest average q-value is chosen as the policy.
    b. Exploration – 10% of the time, a random action (column) is chosen from the possible actions in the current state.

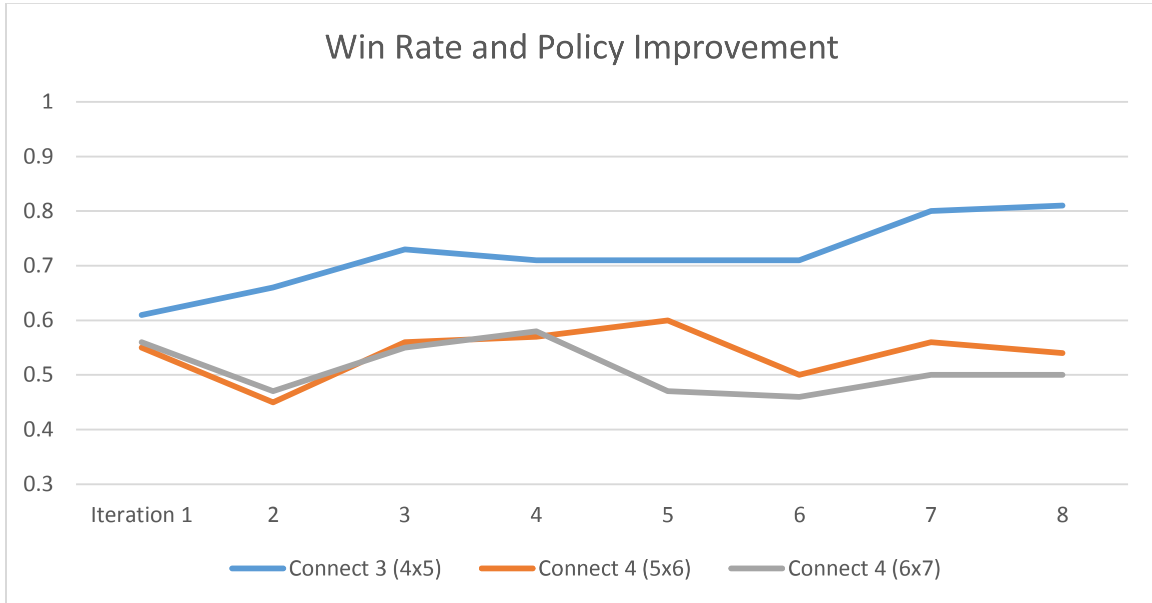
    The chosen action then replaces the corresponding state-action mapping in the policy list.

The program was written in Flash Actionscript 3.0, which may be unavailable to those who wish to compile it.

- The program is demo-able on my website: http://mikedmoy.com/codingdir/connectfour_ai/
- Or can be downloaded and run here: https://dl.dropboxusercontent.com/u/28783024/ConnectFour_MMoy.exe
- The source code is available on Github here: https://github.com/mmoy92/MonteCarloConnect4.git

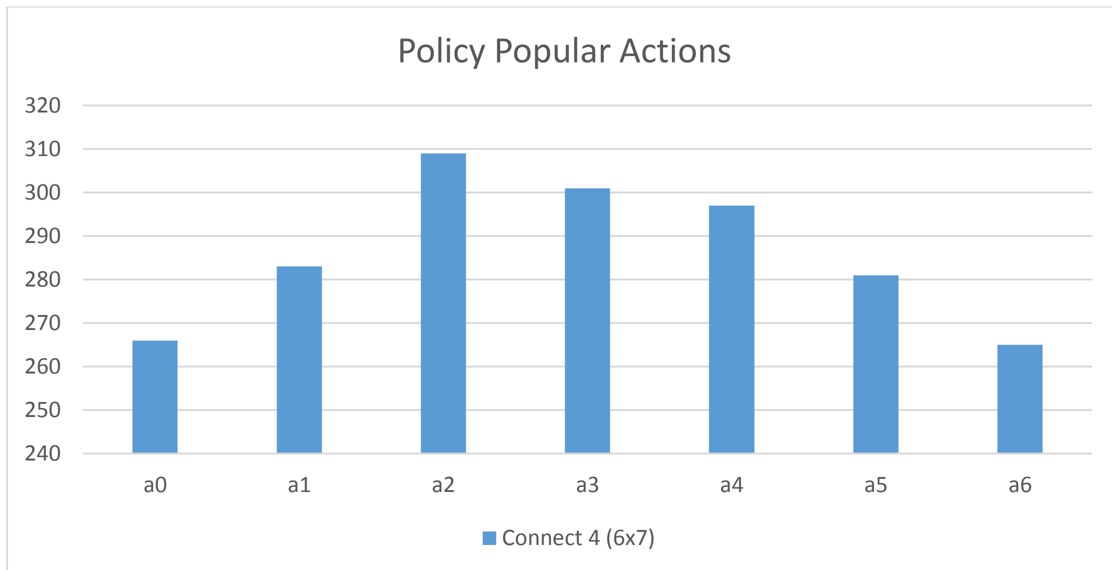
The finished demo allows the user to see the algorithm work in real time, as well as allowing you to specify a board dimension, connect-goal, and play against the learner. You can adjust the execution speed of the program using the >> arrows in the bottom-left corner.

Observations – The Monte Carlo learner does see improvement against the environment as policy improvement is run over time. The policy improvement rate is much faster when the game board is of smaller dimensions (and connect number goals). The q-values also seem to become more precise as time passes, a result of convergence on their "true" values.



After each policy iteration, 100 test games were played to analyze the learner's performance. It seems that boards with the connect-4 goal need much more than 8 policy iterations to learn.

One way to see if a certain strategy results in the best policy was to analyze the q-values after 2000 policies were created. I copied the data from my demo into Notepad++, and ran a search for the occurrence of moves a0 – a6 for a 6x7 board, following a connect-4 goal:



Policies for this board favor central columns more than outer ones.

Peculiarly, when inspecting the q-values for a 5x6 board, the policy popular actions varied, and seemed to have no pattern. This suggests that the center columns are only useful for boards with an odd number of columns. This is likely due to the fact that most winning combinations would have to involve the center (if they were horizontal/diagonal wins). So, it would be beneficial to go first for dimensions r x 3, r x 5, and r x 7, and place your initial token in the middle.

Most of the q-values with winning values (200), occurred with actions in the outer columns as well. Perhaps the optimal strategy is to initially place your tokens in the center columns, and look for finishing connections in the outer columns.

Temporal Difference – I implemented an optional temporal difference option that applies TD during the episode generation stage to see if it improved performance. If a policy doesn't already exist, then TD kicks in and looks toward a future reward for the current state, and picks the best action. The learner only applies TD when policies don't exist so that TD is not used for exploration. The reward-action "lookahead" slightly improves the rate at which the learner achieves winning policies.