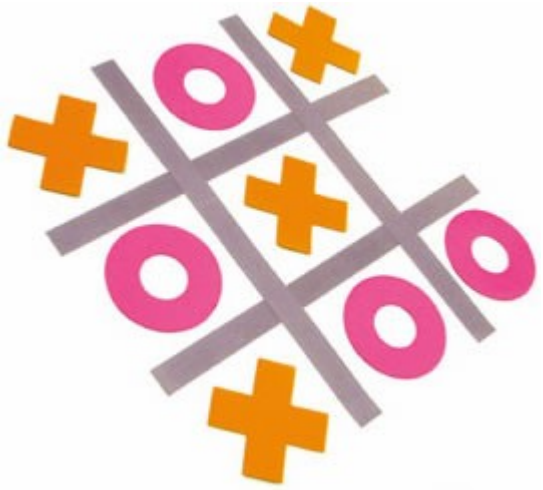
CS670 Reinforcement Learning

MENACE Programming Assignment 1

Gabe Dulac-Arnold <gabe@squirrelsoup.net> [CS09F004]
Johannes H. Jensen <johannj@stud.ntnu.no> [CS09F005]



1.MENACE vs. Various Fixed Opponents

We have trained our MENACE agent against 3 fixed opponents: a 'first-free' opponent, a random player, and an optimal player. For the optimal player we have chosen to integrate an implementation of the minimax player. Each scenario is composed of 1000 matches, and the scenario is repeated 30 times. Each plot represents average performance over the 30 iterations of a scenario.

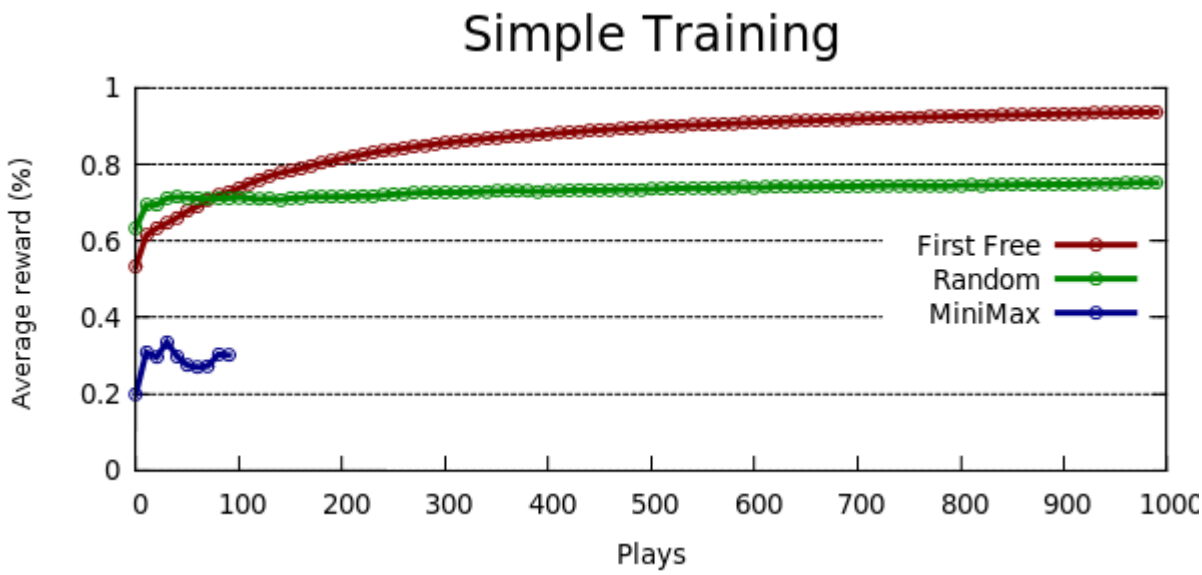


Illustration 1: Simple Training Learning

nb: the minimax player is only run for 100 iterations due to a slow minimax algorithm

2. Exploiting Board Symmetries

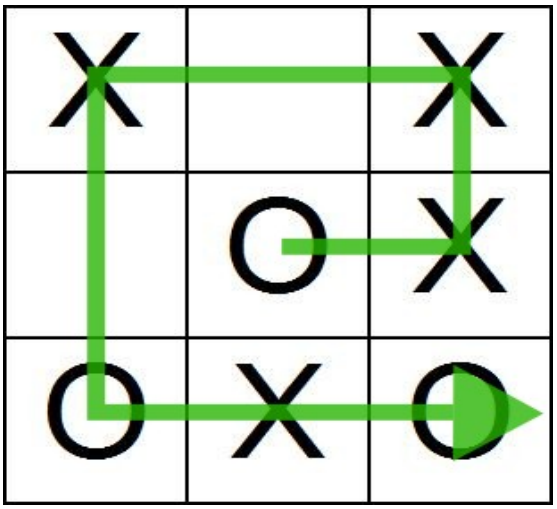
To reduce the state space for the MENACE agent, we can take into account board homomorphisms. The easiest way to accomplish this is to present the agent with the same state vector, regardless of variations in board symmetries. We have chose to accomplish this through a hash algorithm that is insensitive to symmetries. The hash algorithm takes a particular board as input, and outputs a 9 digit integer that is the same for all boards that are equivalent.

Symmetry Hash Algorithm

The following hash algorithm has been proven to be bijective if symmetrical boards are considered equal.

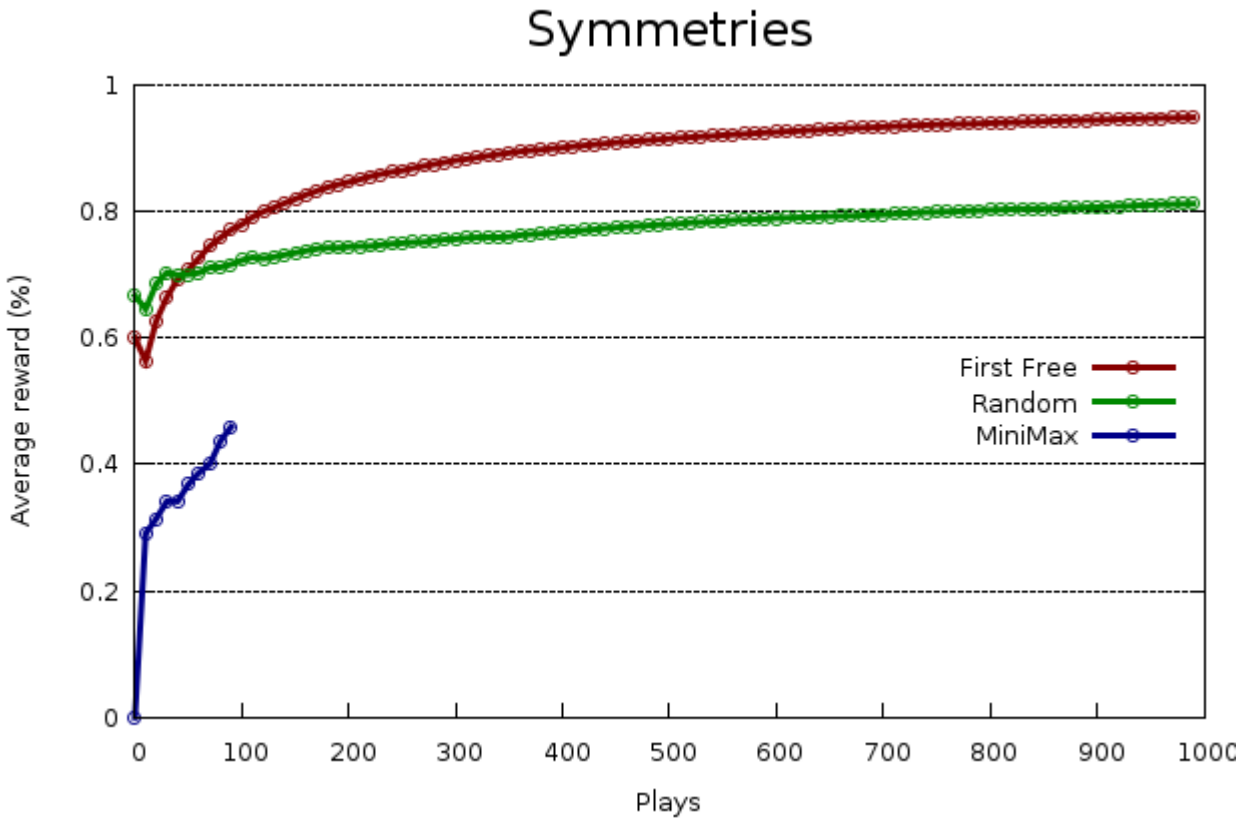
The idea is to traverse the board in such a way that regardless of orientation, the path followed will be identical for equivalent boards. We have therefore chosen the following algorithm for calculating the hash's 'path':

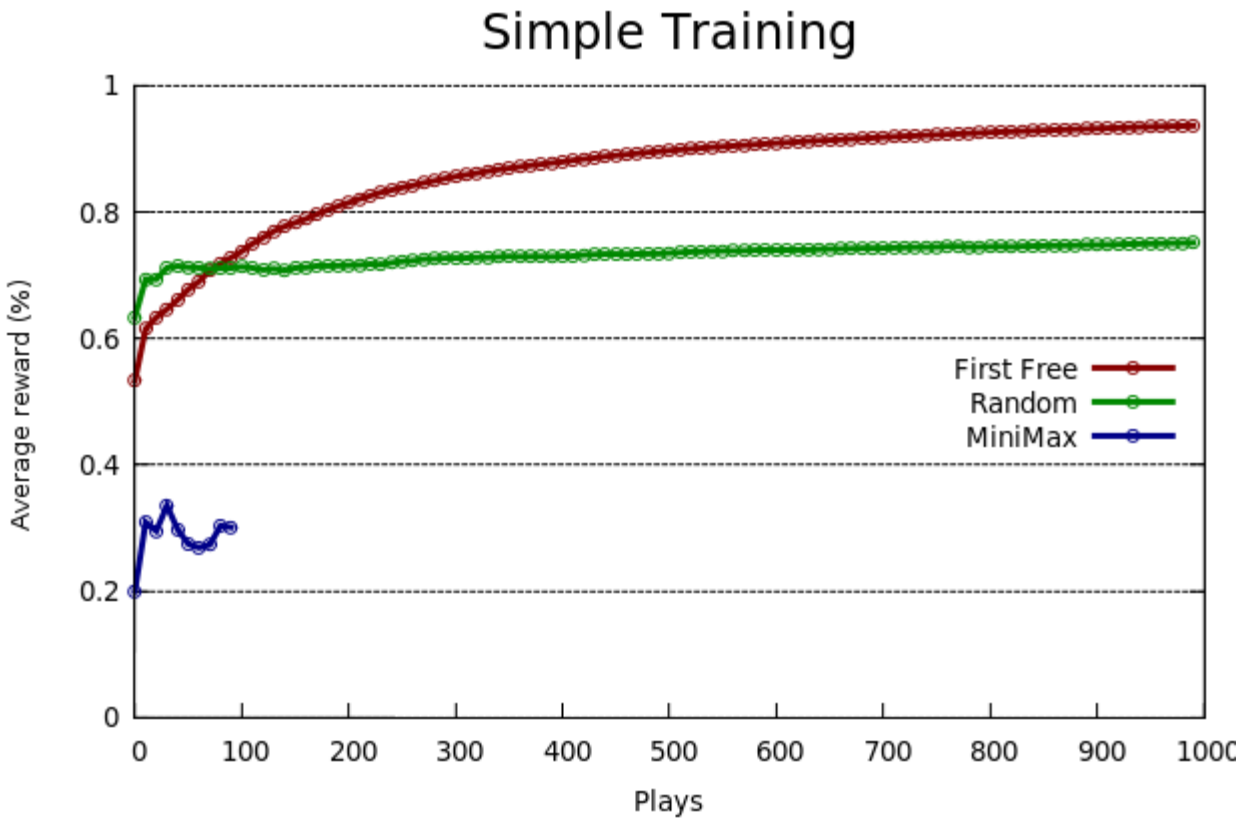
1. Convert the board pieces to integers: an empty spot is 0, O is 1, and X is 2.
2. Start in the middle of the board, as this is an invariant position.
3. Move towards the edge of the board with the biggest value.
4. Move towards the corner of the board with the biggest value.
5. Continue in this direction until the path covers all the spots.



In the event that more than one path is generated by this algorithm, we calculate the hashes for all the possible paths, sort by integer value, and then pick the largest hash amongst them. This guarantees that we will have a unique hash for a specific board configuration, regardless of orientation. Sometimes there are multiple paths that have the maximum value, but in that case the board has internal symmetries, so it does not matter which path is actually being represented.

Here are performance comparisons between the symmetry agent's performance and agents that do not take into account symmetries.

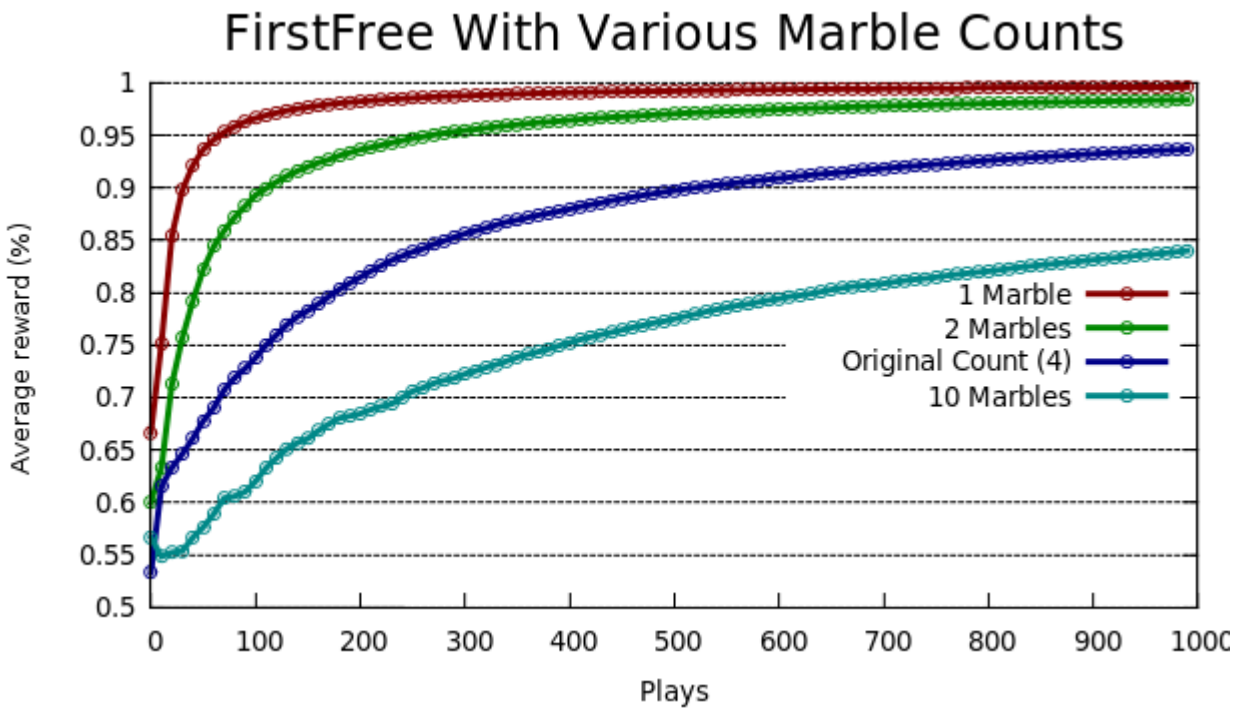


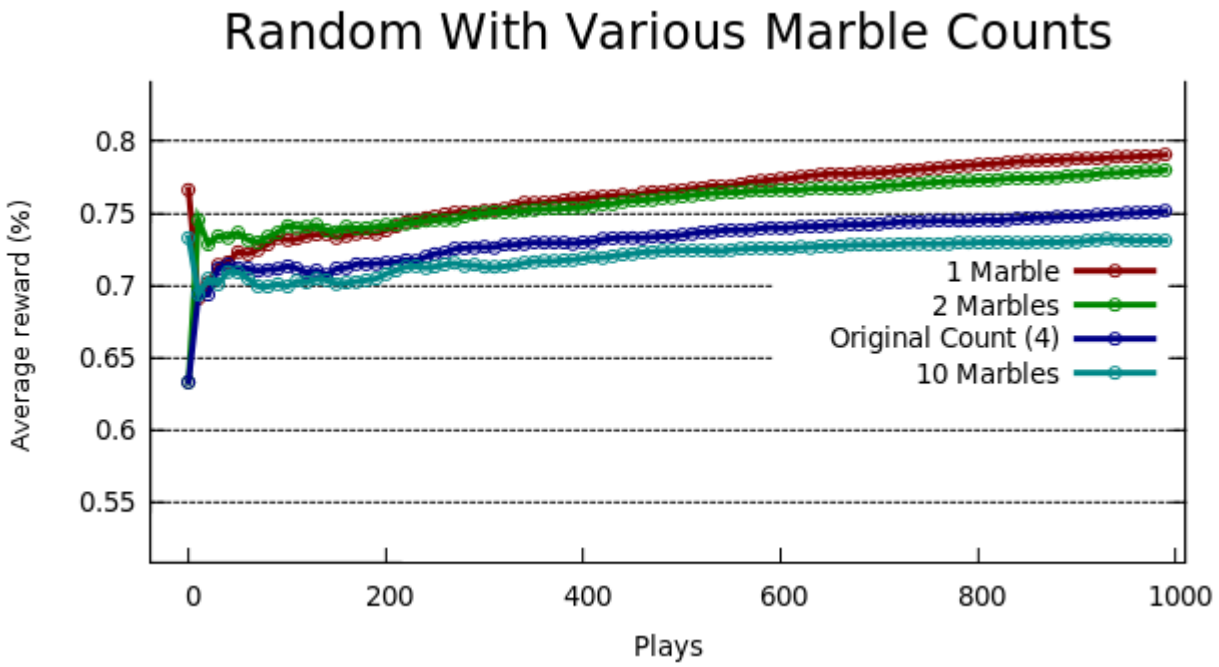


There does seem to be a significant difference with the MiniMax environment, but for the other environments that do not take adavantage of symmetries it does not seem to provide an advantage (and perhaps a disadvantage for random it would seem).

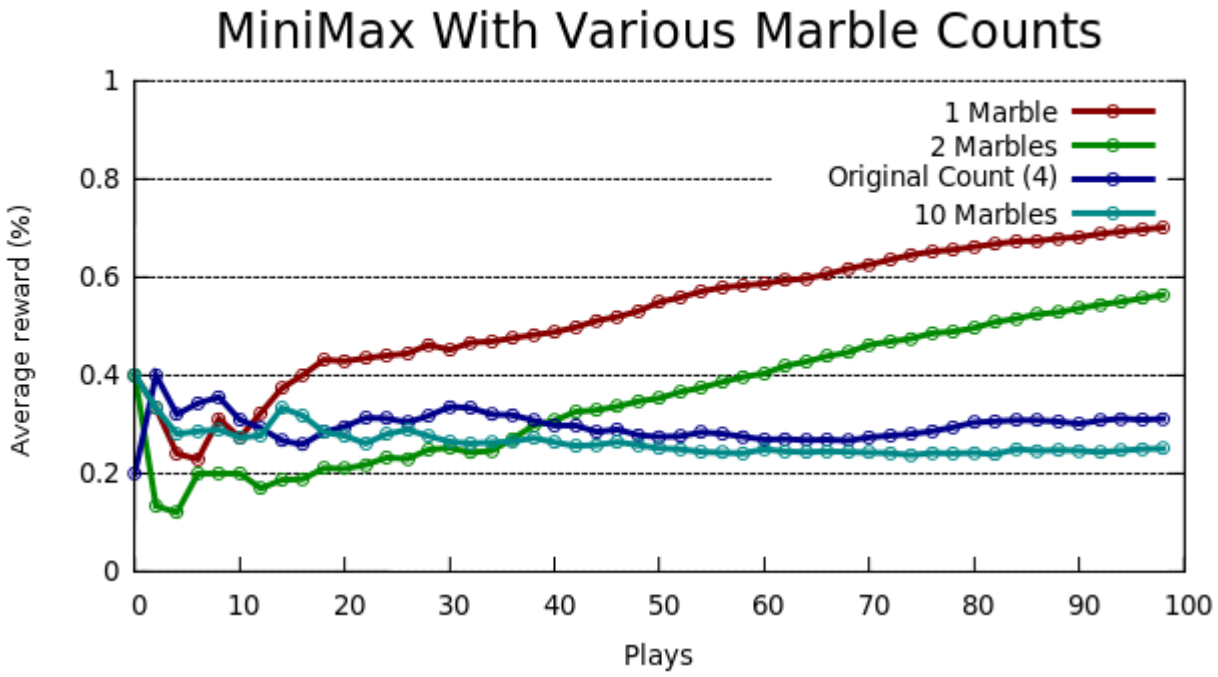
3. Varying Initial Marble Parameters

We repeated the initial experiments described in part 1 with differing amounts of initial marble counts. Here are performance graphs for the different amounts of marbles.



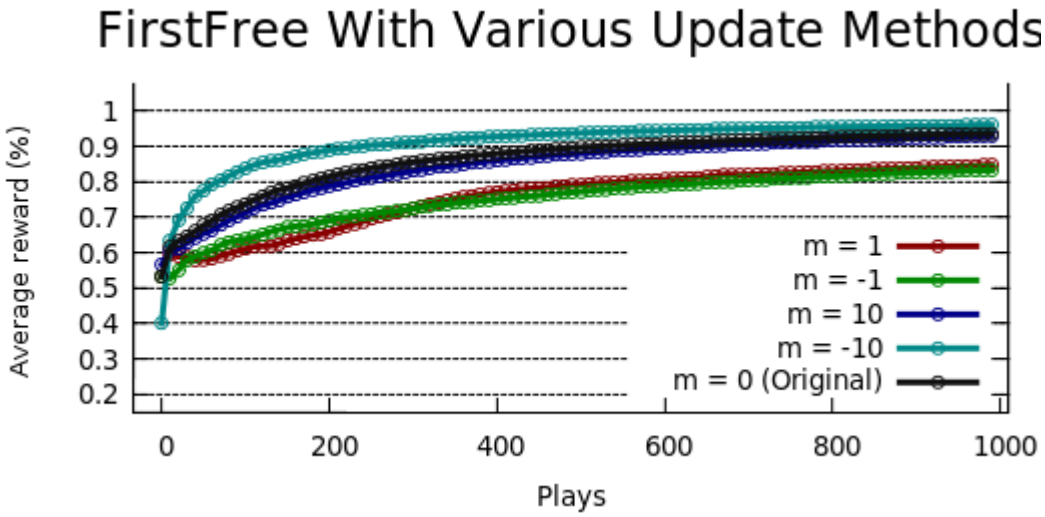


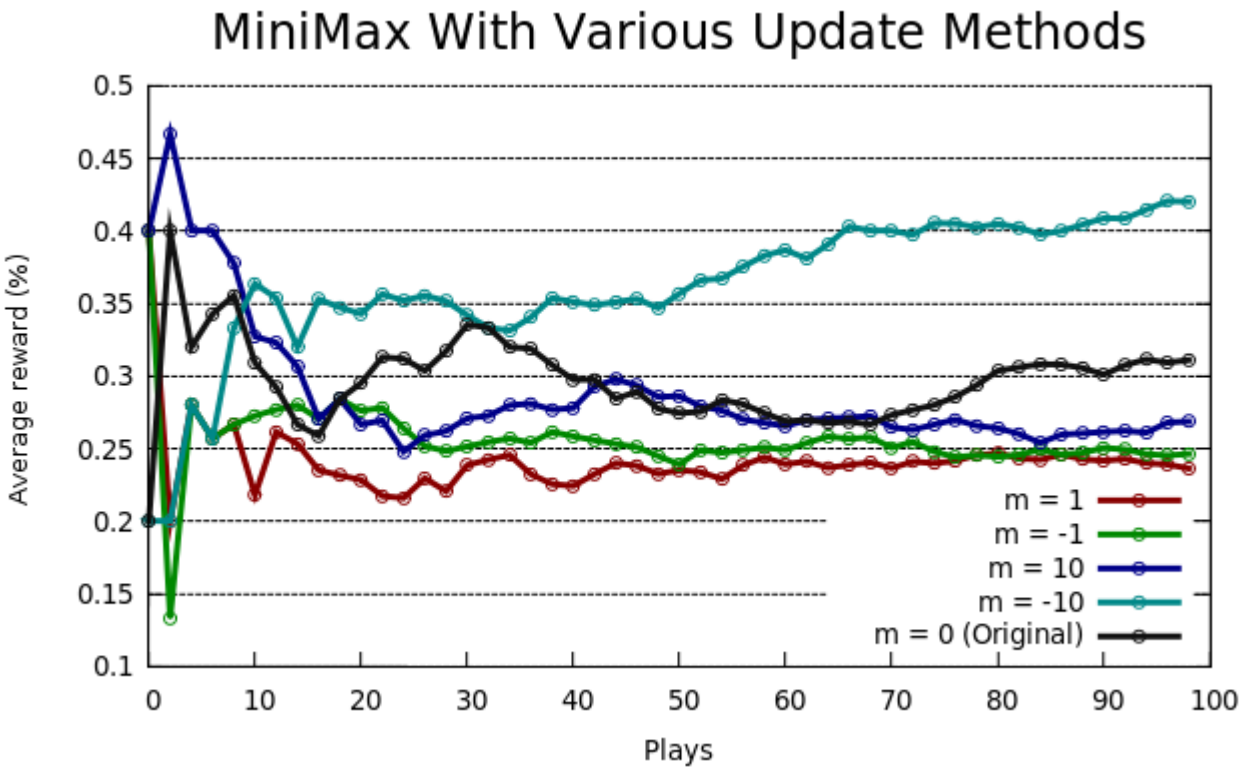
nb: the average reward has been zoomed in to show more detail



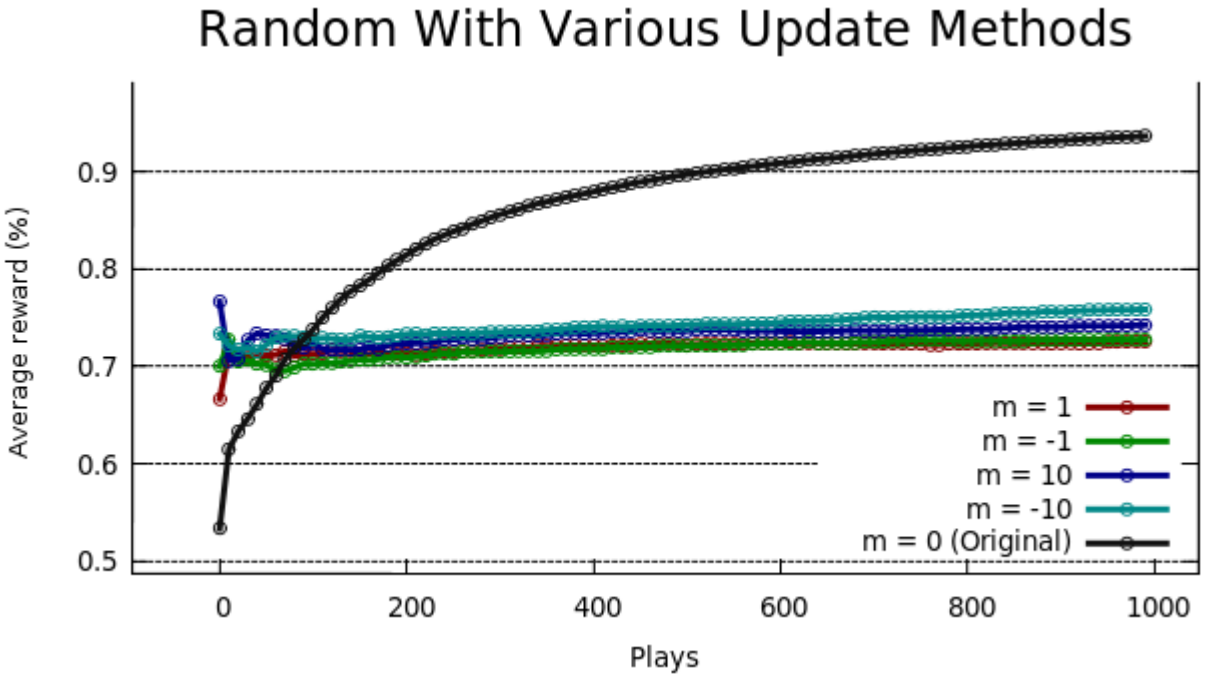
4. Varying Matchbox Update Schemes

Here we repeated the experiments described in part 1 with different update methods for varying the marble count in matchboxes. All these methods were linear, and we varied the slopes. If 'y' is the number of marbles added to the x^{th} box (starting at the empty-board box), then the number of marbles added to the various boxes evolved as y=mx+b. Below is a graph with the various values of m. If m is negative, that means that the first move was rewarded the most, and the last move the least. Otherwise, the rewards are larger the more a move is close to victory. The constant c is set such that the least-rewarded box would receive a reward of 1.





nb: the average reward has been zoomed in to show more detail



nb: the average reward has been zoomed in to show more detail

5. Self Play

To be able to self play, we simply need to extend our environment so that it is able to implement the agent's playing functions. In practice this means inheriting the environment class and the agent's class into a new class, and implement a play function that passes the player's actions to a new instance of the agent as the state, and returns this agent's actions as the new state to the player.

We implemented this change and trained a MENACE agent against another instance of itself, and then let it play with the three original environments.

Against FirstFree: Accumulated reward: 21 (70%)

Against Random: Accumulated reward: 29 (96%)

Against MiniMax: Accumulated reward: 27 (90%)

It's interesting to note that the FirstFree environment fairs rather well against our trained MENACE agent. This may be because the agent was not trained to play with such a bad player, and so the board situations that FirstFree consistently presented were not properly described in the agent's state space.

6. Last But Not Least

How will you modify MENACE to learn a general strategy by playing against different players?
One approach we believe could work well would be to present the MENACE agent with a round-robin response from the different environments (on a game level, not a move level). This way the MENACE agent could learn many different opponents without ever over-specializing on a certain technique.