

COMP8620: MC-AIXI-CTW

Group 3

Jarryd Martin, John Aslanides, Yadunandan Sannappa,
Nrupendra Rao, Cheng Yu, Ryk Budzynski

October 2015

Abstract

We outline an implementation of Veness et al.'s Monte Carlo AIXI approximation[?] (MC-AIXI-CTW), and report our simulation results on a number of toy domains.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 2 | MC-AIXI-CTW Implementation | 1 |
| 2.1 | Agent-environment interaction loop | 1 |
| 2.2 | Monte Carlo Tree Search (MCTS) Algorithm | 1 |
| 2.2.1 | High level description | 1 |
| 2.2.2 | Class structure | 2 |
| 2.2.3 | Other Considerations | 2 |
| 2.2.4 | Efficiency/performance | 3 |
| 2.3 | Context Tree Weighting (CTW) | 3 |
| 2.3.1 | High level description of algorithm | 3 |
| 2.3.2 | Class structure | 3 |
| 2.3.3 | Code snippets | 3 |
| 2.3.4 | Efficiency/performance | 3 |
| 3 | Environments | 3 |
| 3.1 | Cheese Maze | 3 |
| 3.2 | Extended Tiger | 4 |
| 3.3 | Tic-Tac-Toe | 5 |
| 3.4 | Biased Rock-Paper-Scissors | 5 |
| 3.5 | PacMan | 6 |
| 4 | Experiments | 6 |
| 4.1 | Experimental Setup | 6 |
| 4.2 | Experiment Summary | 6 |
| 4.3 | Cheesemaze | 7 |
| 4.4 | Extended Tiger | 7 |
| 4.5 | TicTacToe | 8 |
| 4.6 | Biased Rock-Paper-Scissors | 8 |
| 4.7 | Pacman | 9 |
| 4.8 | Cross Domain Simulation Results | 10 |
| 5 | Conclusion | 10 |
| A | Files | 10 |

1 Introduction

Recall that the AIXI agent is defined by its actions, which for each cycle k are given by

$$a_k^{\text{AIXI}} = \arg \max_{a_k} \sum_{o_k r_k} \cdots \max_{a_m} \sum_{o_m r_m} [r_k + \cdots + r_m] \xi(o_1 r_1 \dots o_m r_m | a_1 \dots a_m),$$

where the o_n and r_n are the observation and reward provided by the environment at cycle n , and ξ is a Bayesian mixture model for the environment.

Following Veness et al., we approximate a_k^{AIXI} using Monte Carlo tree search (upper confidence bound) to approximate the expectimax, and we compute a mixture over variable-order Markov models using the context-tree weighting algorithm.

We present a lightweight C++ implementation of MC-AIXI-CTW, along with implementations of a number of simple games: PACMAN, TIC-TAC-TOE, BIASED ROCK-PAPER-SCISSOR, EXTENDED-TIGER, and CHEESEMAZE.

2 MC-AIXI-CTW Implementation

In this section we outline the implementation of the agent, and its interface with its environments.

2.1 Agent-environment interaction loop

For each agent-environment interaction cycle, we run the following for each experiment (see Algorithm ??):

Algorithm 1 Main loop.

```

1: while  $cycle < max\_cycles$  do
2:   while environment is not finished do
3:     generate  $(o, r)$  from environment
4:     update agent model with  $(o, r)$ 
5:     if explore then
6:        $a \leftarrow randomAction()$ 
7:     else
8:        $a \leftarrow MCTS()$  ▷ Monte Carlo Tree Search
9:     end if
10:    perform action  $a$ 
11:    update agent history with  $a$ 
12:     $cycle++$ 
13:   end while
14:   reset environment
15: end while

```

The *MCTS* algorithm follows as in Algorithms 1-4 in Veness et al., and is found in `search.cpp`. Model updates are handled in methods in `agent.cpp`, which interfaces with the context tree defined in `predict.cpp`.

2.2 Monte Carlo Tree Search (MCTS) Algorithm

2.2.1 High level description

Since the environment is only partially observable, we have no explicit notion of state; instead, we only have a history of actions and percepts $h = (a_1 o_1 r_1 \dots a_n o_n r_n)$. For the purposes of choosing the optimal action, we treat each possible (hypothetical) history as a node in the search tree, with the root being the tip of the current (realised) history.

The search tree is comprised of alternate layers of decision nodes and chance nodes with the root being a decision node. The maximum branching possible from decision nodes is the number of available actions in the given environment while

the maximum branching possible from chance nodes is equal to the number of possible observations times the number of possible rewards. We do however restrict branching in general to 100 to avoid memory issues. This number is configurable in `search.cpp`.

Each node is implicitly labelled by its history as chance nodes record the hypothetical action taken while decision nodes record the hypothetical observation and reward from the environment. The expected value of each node in the search tree is equal to the expected total (non-discounted) reward that would be accumulated from that node, where the expectation is under the agent’s current policy and the agent’s current model for the behavior of the environment.

Thus, for each node we keep a value estimate \hat{V} , and a count of the number of times T that node has been visited in search. This is used to determine how we explore the search space using the UCB algorithm, which, for each decision node picks (assuming $T(ha) > 0$ for all actions)

$$a_{\text{UCB}} = \arg \max_{a \in \mathcal{A}} \left\{ \frac{1}{m(\beta - \alpha)} \hat{V}(ha) + C \sqrt{\frac{\log T(h)}{T(ha)}} \right\},$$

where \mathcal{A} is the set of all permissible actions, m is the search horizon, $\beta - \alpha$ is the difference between the minimal and maximal instantaneous reward, and C is a parameter controlling the propensity of the agent to explore less frequently-seen histories.

Note that the expectimax is a stochastic, partially observable game between the agent and its environment. In the following, call nodes corresponding to agent moves ‘decision nodes’, and nodes corresponding to Nature’s moves ‘chance nodes’.

2.2.2 Class structure

To represent our search nodes, we define a base `SearchNode` class, from which `ChanceNode` and `DecisionNode` inherit. `ChanceNode` and `DecisionNode` each have a `sample` method defined on them; each of these methods is mutually recursive. For a given node n , we keep its children in a dictionary keyed on the action or (observation, reward) used to generate each child.

2.2.3 Other Considerations

In addition to the pseudocode presented in Veness et al., we implemented a solution to allow us to decide on a per cycle basis whether or not to build a search tree from scratch. In addition, we constructed routines which given an action taken by the agent and an observation/reward received from the environment, would prune the search tree accordingly.

The pruning of the search tree removes all subtrees beginning with the chance nodes directly below the root which correspond to the actions not taken by the agent and as such are impossible future paths. Additionally, all subtrees beginning with decision nodes below the chance node (corresponding to the action taken) that do not match the observation/reward received from the environment are pruned. As cycles progress throughout an experiment, this represents a significant reduction in memory consumption.

Our final solution however did not implement pruning as our experimental results showed unusual behaviour whereby the agent would appear to get stuck in certain game states.

2.2.4 Efficiency/performance

Between calls to `search`, we retain much of the search tree, pruning those nodes that are now inaccessible from the realised (a, o, r) tuple that happened during the cycle. This allows us to avoid re-generating similar search trees from similar positions.

2.3 Context Tree Weighting (CTW)

2.3.1 High level description of algorithm

2.3.2 Class structure

2.3.3 Code snippets

2.3.4 Efficiency/performance

3 Environments

To test the effectiveness of the agent we developed 5 environment simulations which range in complexity from the relatively simple cheese maze to the much more complicated partially observable pacman game. The details of these environments with respect to their behaviour and implementation is discussed below. For technical convenience, our agent only handles non-negative rewards; for this reason, our environments are all implemented so as to have their minimum reward set to zero. In this way, our implementations differ from the environment specifications. For the purpose of plotting and analysis of the results, we transform all rewards back to their original specified ranges, so as to make comparison with Veness’s benchmark easier. Note that this additive reward transformation has no effect on the agent’s behavior.

In all of the environments, the agent is given a static interface of possible actions it can perform. At any given game state, some (indeed many) of these actions may be illegal; in all games, illegal moves are punished with significant penalties.

3.1 Cheese Maze

Cheese Maze is an episodic, deterministic and partially observable game in which the agent is a mouse trying to find a piece of cheese in a two dimensional maze. The objective is to find the cheese in the fewest amount of moves, while avoiding running into walls. The episode ends when the agent finds the cheese. The actions available to the agent are:

| Action | Code |
|------------|------|
| Move Up | 0 |
| Move Right | 1 |
| Move Left | 2 |
| Move Down | 3 |

The transformed rewards are now given by

| Action effect | Reward |
|----------------------------|--------|
| Game start | 0 |
| Agent bumps into wall | 0 |
| Agent moves into free cell | 9 |
| Agent finds cheese | 20 |

The configuration of the maze and initial states of the agent and the cheese are read from the configuration file, and are identical to those specified in Figure 5 in Veness et al. The maze is represented as the list of nodes as they would be visited by the depth first search algorithm. Mouse position and cheese position are simply the order of the node. Each node of the maze is a structure which stores the percept corresponding to that node, along with an array of pointers to its neighbours. In case there is a wall on a particular side of the node then that pointer will be NULL.

The optimal strategy in each episode of Cheese Maze is to perform the sequence of actions(`gamestart` → `right` → `down` → `down`). This yields a total reward of 8 utils in 4 cycles, giving an upper bound for the average reward per cycle of 2. Average reward per cycle is the main metric with which we evaluate the agent’s performance in this and all other environments.

3.2 Extended Tiger

Extended Tiger is an episodic stochastic game. This environment simulates two doors, behind one of which is a tiger, and the other a pot of gold. At the start of each episode the tiger is placed behind one of the doors with probability 0.5. The agent begins sitting down and it can perform the following actions:

| Action | Code |
|-----------------|------|
| Stand | 0 |
| Listen | 1 |
| Open left door | 2 |
| Open right door | 3 |

When the agent performs the **listen** action, the environment provides an observation that correctly identifies the location of the tiger with probability p (parametrised in the configuration file). Otherwise, its observation is 0:

| Observation | Code |
|--------------------------------------|------|
| Agent does not perform listen action | 0 |
| Tiger behind left door | 1 |
| Tiger behind right door | 2 |

The agent is penalised for performing illegal moves, or for opening the door with the tiger. The episode ends when one of the doors is opened. The transformed rewards the agent receives in this environment are as follows:

| State | Action | Reward |
|----------|----------------------|--------|
| sitting | stand | 99 |
| sitting | open door | 90 |
| sitting | listen | 99 |
| standing | stand | 90 |
| standing | open door with tiger | 0 |
| standing | open door with gold | 30 |
| standing | listen | 90 |

Clearly the optimal strategy in Extended Tiger will be the sequence (**gamestart** \rightarrow **listen** n times \rightarrow **stand** \rightarrow **opendoor**). To play optimally given a false observation probability of $q = 1 - p$, it is sufficient to maximise our expected reward with respect to n using this sequence of moves. Hence the expected average reward under the optimal policy is given by

$$\mathbb{E}[r_{\text{avg}}]^* = \frac{30(1-q)^n - 100(1 - (1-q)^n) - n - 1}{n + 3}$$

where the denominator is equal to the number of actions taken. For example, with $q = 0.15$, the optimal n is 2.

3.3 Tic-Tac-Toe

In this game, the agent plays games of TicTacToe against an opponent who makes random moves. The agent moves are coded as 0-8 referring to the different positions of the board.

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Observation completely describes the current state of the board using 2 bits for each cell of the board.

| | |
|----|--------------------|
| 00 | Cell empty |
| 01 | Agent's cell |
| 02 | Environment's cell |

The agent rewards are as follows:

| Status | Reward |
|-------------------|--------|
| Illegal move | 0 |
| Game is a draw | 4 |
| Game won by agent | 5 |
| Agent lost | 1 |

In terms of an optimal agent strategy, an upper bound on reward per cycle is 0.5 given that the minimum number of cycles required for the agent to win the game is 4.

3.4 Biased Rock-Paper-Scissors

The agent will play games of Rock-Paper-Scissors against an opponent who plays the same action every time it wins the game, and otherwise plays a random move. For this agent we encoded the actions of the agent and the environment as

| | |
|----------|---|
| Rock | 0 |
| Paper | 1 |
| Scissors | 2 |

The agent receives a reward of 1 for a win, 0 for a draw and -1 for a loss.

In terms of an optimal agent strategy, an upper bound on reward per cycle for Biased Rock-Paper-Scissors is derived as follows.

Let X_t be a R.V. representing the outcome of cycle t . Under an optimal strategy, the agent will win after the environment won in cycle $t - 1$ as the environments action in cycle t is predictable. Otherwise, the agent plays randomly. As such we have

$$P(X_t = \text{win}) = \sum_{X_{t-1}} P(X_t = \text{win}, X_{t-1}) = \sum_{X_{t-1}} P(X_t = \text{win} | X_{t-1}) P(X_{t-1})$$

which is a 1st order Markov chain with the following right stochastic transition matrix

$$\mathbf{P} = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 1 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} \quad (1)$$

Therefore, we want a steady state distribution for \mathbf{P} , i.e. a solution to $\boldsymbol{\pi} = \mathbf{P}\boldsymbol{\pi}$, which is an eigenvector $\boldsymbol{\pi}^*$. The average reward at time t is then

$$\mathbb{E}[r_{\text{avg}_t}] = 1 \cdot P(X_t = \text{win}) + 0 \cdot P(X_t = \text{draw}) - 1 \cdot P(X_t = \text{lose}) = \boldsymbol{\pi}_0^* - \boldsymbol{\pi}_2^* = 0.25$$

3.5 PacMan

This is game of PacMan in which the agent can only partially observe the environment unlike the classic game in which the entire game state is known to the player at any given time. The agent does not know the structure of the maze but only receives a 4 bit wall configuration of the node it is currently in. The agent also receives a 4 bit observation of the presence of any ghosts in its direct line of sight. The location of food pellets can change every episode as every free cell has a 50% chance of having food in it. The agent receives a 3 bit observation indicating the presence of food within a Manhattan distance of 2, 3 or 4. It also receives a 4 bit string indicating food in its line of sight similar to the ghosts. The agent also knows when it is under the effect of the power pill.

| Action | Reward |
|-----------------------------|--------|
| Agent runs into a wall | -10 |
| Agent caught by a ghost | -50 |
| Agent moves into empty cell | -1 |
| Agent eats a food pellet | 10 |
| Agent collects all the food | 100 |

To calculate the actual reward, the rewards from the above table are added to 60. This is because the agent can potentially be affected by multiple rewards e.g. it can run into a wall and get caught by a ghost in the same turn.

In terms of an optimal agent strategy, an upper bound on reward per cycle for Pacman is unknown.

4 Experiments

4.1 Experimental Setup

4.2 Experiment Summary

Following is a complete summary of all experiments conducted

| | Bits ¹ | CT-depth | Cycles | Timeout ² | Horizon | Exp. Rate | Exp.-Decay | UCB-weight ³ |
|----------------------------|-------------------|----------|--------|----------------------|---------|-----------|--------------|-------------------------|
| Cheesemaze | 11 | 96 | 10 | 0.5 | 8 | 0.999 | 0.999769818 | 1.4 |
| | 11 | 96 | 10 | 1.6 | 6 | 0.999 | 0.999769818 | 1.4 |
| | 11 | 144 | 2.5 | 3 | 8 | 0.999 | 0.9990795899 | 1.4 |
| | 11 | 48 | 10 | 0.1 | 4 | 0.999 | 0.999769818 | 1.4 |
| | 11 | 96 | 10 | 0.5 | 8 | 0.999 | 0.999769818 | 1.4 |
| Biased Rock-Paper-Scissors | 6 | 32 | 10 | 2.5 | 4 | 0.999 | 0.999769818 | 1.4 |
| | 6 | 48 | 5 | 5 | 8 | 0.999 | 0.999539689 | 1.4 |
| | 6 | 96 | 4.5 | 4 | 4 | 0.999 | 0.9994885564 | 1.4 |
| | 6 | 96 | 10 | 0.5 | 4 | 0.999 | 0.999 | 1.4 |
| Extended Tiger | 13 | 96 | 16 | 2.5 | 6 | 0.999 | 0.99985613 | 1.4 |
| | 13 | 96 | 2.5 | 12 | 4 | 0.999 | 0.9990795899 | 1.4 |
| | 13 | 96 | 25 | 0.8 | 4 | 0.999 | 0.9999079208 | 1.4 |
| | 13 | 52 | 5 | 8 | 4 | 0.999 | 0.999539689 | 1.4 |
| Tic-Tac-Toe | 25 | 96 | 4.5 | 4 | 9 | 0.9999 | 0.9994884564 | 1.4 |
| | 25 | 192 | 5 | 8 | 9 | 0.9999 | 0.999539599 | 1.4 |
| | 25 | 256 | 20 | 8 | 9 | 0.9999 | 0.9998848799 | 1.4 |
| | 25 | 512 | 25 | 3 | 9 | 0.9999 | 0.9999079028 | 1.4 |
| | 25 | 512 | 30 | 2 | 9 | 0.9999 | 0.9999232518 | 1.4 |
| Pacman | 26 | 256 | 10 | 4 | 8 | 0.99 | 0.999 | 1.5 |
| | 26 | 512 | 20 | 2 | 4 | 0.9999 | 0.9998848799 | 1.4 |
| | 26 | 320 | 10 | 4 | 8 | 0.99 | 0.999 | 1.4 |
| | 26 | 256 | 10 | 1 | 6 | 0.9999 | 0.999769773 | 1.4 |

Table 1: MC-AIXI-CTW Experiments

4.3 Cheesemaze

Here we present the environmental setup and learning rate of our best results training AIXI on the Cheesemaze environment.

Interpreting the results, we conclude that ...

| Bits | CT-depth | Cycles | Timeout | Horizon | Exp. Rate | Exp.-Decay | UCB-weight |
|------|----------|--------|---------|---------|-----------|-------------|------------|
| 11 | 96 | 10 | 0.5 | 8 | 0.999 | 0.999769818 | 1.4 |

Table 2: Environment Setup

4.4 Extended Tiger

Here we present the environmental setup and learning rate of our best results training AIXI on the Extended Tiger environment.

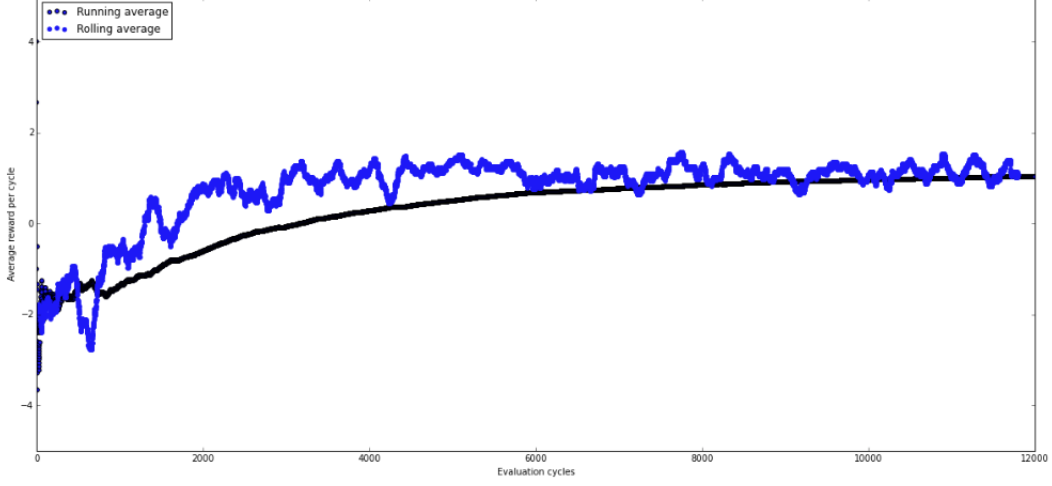
Interpreting the results, we conclude that ...

¹total bit length of an ora cycle

²timeout value for each select action

³UCB exploration bias parameter

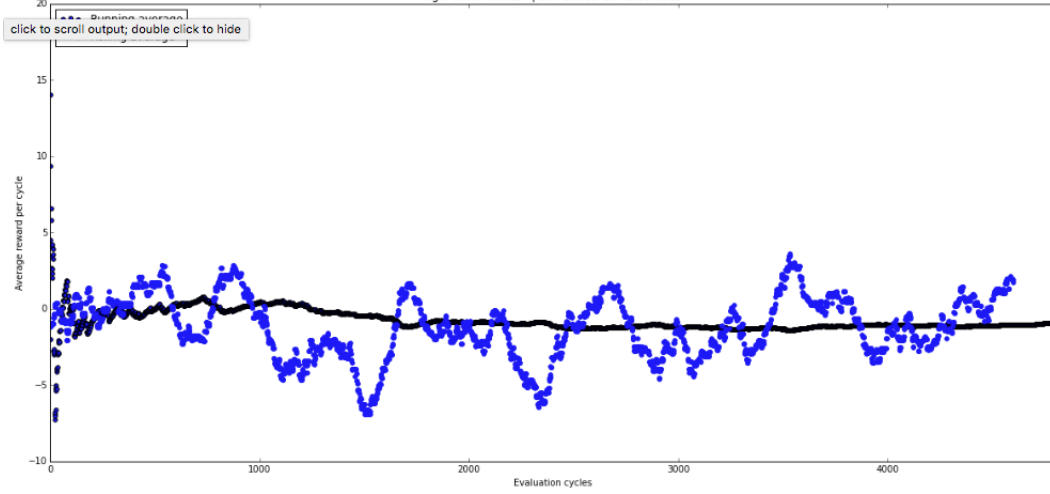
Figure 1: Learning Rate



| Bits | CT-depth | Cycles | Timeout | Horizon | Exp. Rate | Exp.-Decay | UCB-weight |
|------|----------|--------|---------|---------|-----------|-------------|------------|
| 11 | 96 | 10 | 0.5 | 8 | 0.999 | 0.999769818 | 1.4 |

Table 3: Environment Setup

Figure 2: Learning Rate



4.5 TicTacToe

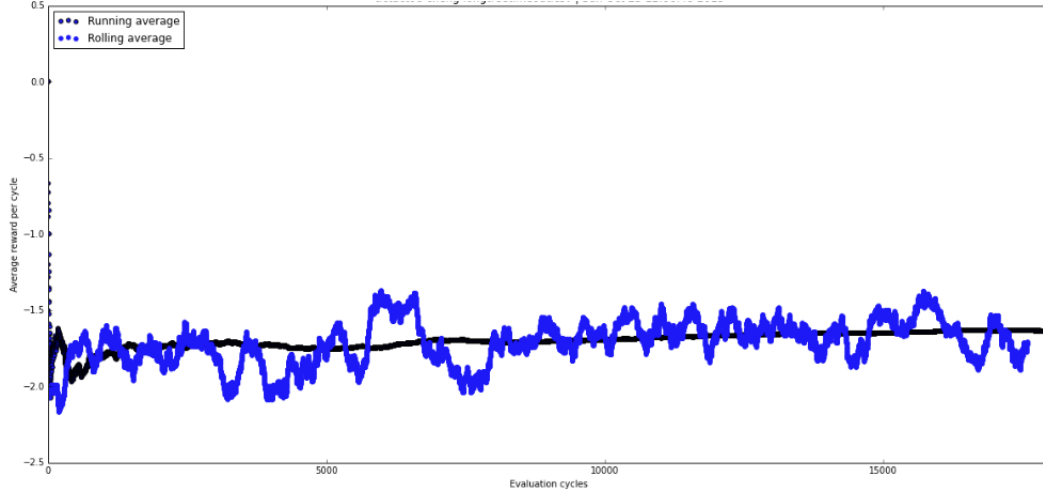
Here we present the environmental setup and learning rate of our best results training AIXI on the Tic-Tac-Toe environment.

Interpreting the results, we conclude that ...

| Bits | CT-depth | Cycles | Timeout | Horizon | Exp. Rate | Exp.-Decay | UCB-weight |
|------|----------|--------|---------|---------|-----------|-------------|------------|
| 11 | 96 | 10 | 0.5 | 8 | 0.999 | 0.999769818 | 1.4 |

Table 4: Environment Setup

Figure 3: Learning Rate



4.6 Biased Rock-Paper-Scissors

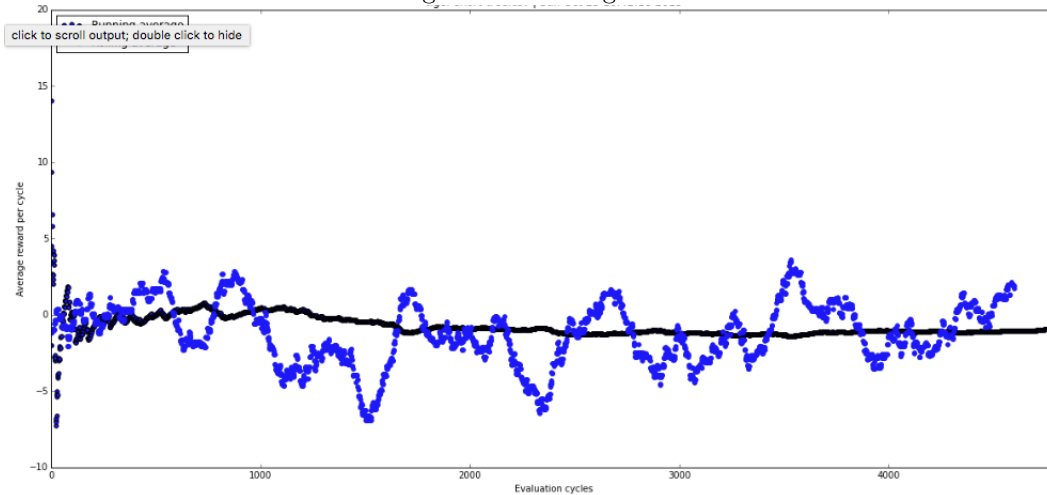
Here we present the environmental setup and learning rate of our best results training AIXI on the Biased Rock-Paper-Scissors environment.

Interpreting the results, we conclude that ...

| Bits | CT-depth | Cycles | Timeout | Horizon | Exp. Rate | Exp.-Decay | UCB-weight |
|------|----------|--------|---------|---------|-----------|-------------|------------|
| 11 | 96 | 10 | 0.5 | 8 | 0.999 | 0.999769818 | 1.4 |

Table 5: Environment Setup

Figure 4: Learning Rate



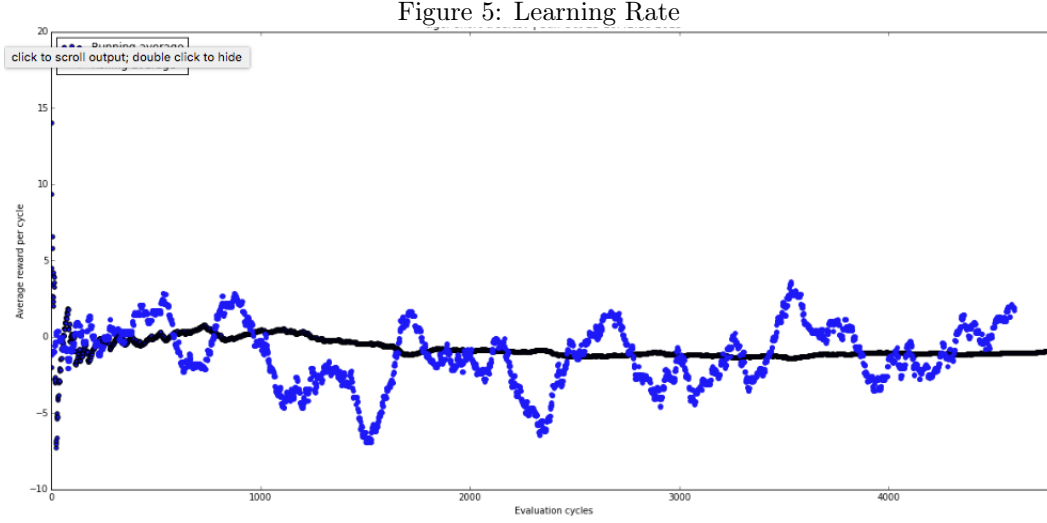
4.7 Pacman

Here we present the environmental setup and learning rate of our best results training AIXI on the Pacman environment.

Interpreting the results, we conclude that ...

| Bits | CT-depth | Cycles | Timeout | Horizon | Exp. Rate | Exp.-Decay | UCB-weight |
|------|----------|--------|---------|---------|-----------|-------------|------------|
| 11 | 96 | 10 | 0.5 | 8 | 0.999 | 0.999769818 | 1.4 |

Table 6: Environment Setup



4.8 Cross Domain Simulation Results

- ◇ Cheesemaze and Biased Rock-Paper-Scissors
- ◇ Cross domain simulation on more difficult environments...

5 Conclusion

Appendix

User Manual

To build from source on linux, run `g++ -std=c++0x *.cpp -o aixi` **TODO check**

To run, invoke `./aixi $ENVNAME.conf`.

A Files

The report archive should contain the following:

```
MC-AIXI-CTW-Grp3.zip
\report
  report.pdf // this report
  report.tex
  cheesemaze_best.png // results plots
  tiger_best.png
```

```
rockpaper_best.png
tictactoe_best.png
pacman_best.png
\src
main.hpp
main.cpp
environment.hpp
environment.cpp
agent.hpp
agent.cpp
search.hpp
search.cpp
predict.hpp
predict.cpp
util.hpp
util.cpp
README.md
cheesemaze.conf // environment configuration files
rockpaper.conf
tictactoe.conf
coinflip.conf
tiger.conf
```

B Graphs..