COMP8620: MC-AIXI-CTW
Group 3

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

October 2015

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.

1 Introduction

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

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

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 User Manual

To build from source, run `g++ *.cpp *.hpp -o aixi` **TODO check**

To run, invoke `./aixi envname`.

3 MC-AIXI-CTW Implementation

3.1 Main loop

For each agent-environment interaction cycle, we run the following for each experiment:

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`.

3.2 Monte Carlo Tree Search (MCTS) Algorithm

3.2.1 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 $h = (a_1 o_1 r_1 \ldots a_n o_n r_n)$. 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

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 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'.

3.2.2 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.

3.2.3 Code snippets

3.2.4 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.

3.3 Context Tree Weighting (CTW)

3.3.1 High level description of algorithm

3.3.2 Class structure

3.3.3 Code snippets

3.3.4 Efficiency/performance

3.4 Environments

3.4.1 Cheesemaze

3.4.2 Extended Tiger

3.4.3 Biased Rock-Paper-Scissor

3.4.4 Tic-Tac-Toe

3.4.5 Pacman

4 Simulation Results

4.1 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
Extended Tiger	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

Table 1: MC-AIXI-CTW Experiments

[1]total bit length of an `ora cycle`
[2]timeout value for each select action
[3]the C parameter from Veness et. al.

4.2 Cheesemaze

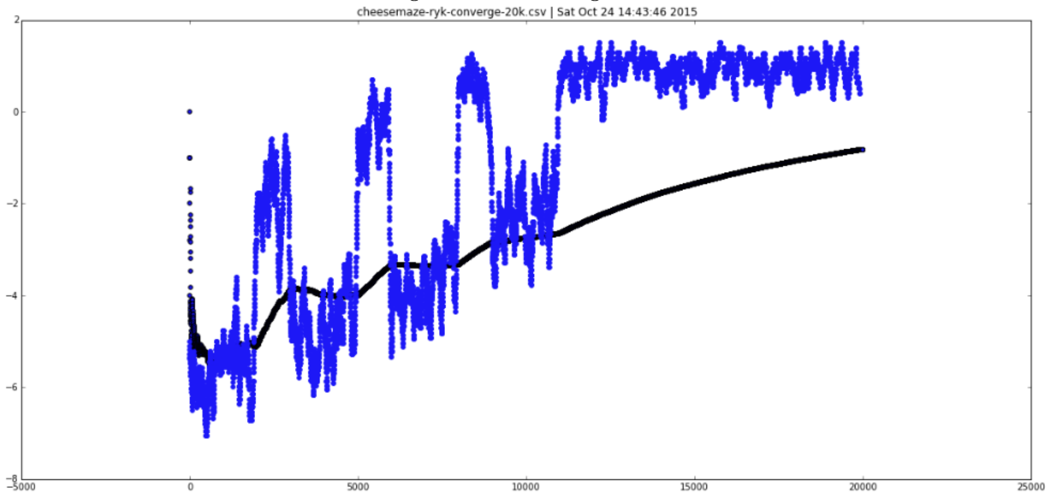
Here we present the environmental setup and learning rate of our best results training AIXI on the Cheesemaze environment.

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

Figure 1: Learning Rate



4.3 Extended Tiger

Experimental setup ...
Plots ...

4.4 Biased Rock-Paper-Scissor

Experimental setup ...
Plots ...

4.5 Tic-Tac-Toe

Experimental setup ...
Plots ...

4.6 Pacman

Experimental setup ...
Plots ...

5 Cross Domain Simulation Results

◇ Cheesemaze and Extended Tiger

◇ Cross domain simulation on more difficult environments...

◇ Separate CTW for Obs and Rews...

6 Discussion and Conclusions

Appendix

A Files

The report archive should contain the following:

```
MC-AIXI-CTW-Grp3.zip
    \report
        report.pdf // this report
        report.tex
        cheesemaze_01.png // results plots
        extended_tiger_01.png
        biased_rock_paper_scissor_01.png
        tic_tac_toe_01.png
        pacman_01.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 Grpahs..