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

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

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

4 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 all environments to avoid negative rewards to the agent we have added a constant to all rewards such that the minimum reward received by the agent is 0. Since the difference between the rewards is same this will not affect the behaviour of the agent in any way.

4.1 Cheese Maze

This is a environment in which the agent is a mouse trying to find a piece of cheese in an two dimensional maze. The actions of the agent are:

Action	**Code**
Move Up	0
Move Right	1
Move Left	2
Move Down	3

The rewards are have been adjusted as

Action effect	**Reward**
Agent bumps into wall	0
Agent moves into free cell	9
Agent finds cheese	20

The details of the maze and the starting positions of the agent and the cheese are read from the configuration file. 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 of that node and an array of pointers which point to its neighbours. In case there is a wall on a particular side of the node then that pointer will be NULL.

4.2 Extended Tiger

This environment simulates two doors. At the start of each game the tiger will placed behind one of the doors with probability of 0.5 and behind the other there is a pot of gold. The agent begins sittnd 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 tries to listen the environment provides it with an observation which correctly describes where the tiger is with a probability as defined in the configuration file. Until the listen action is performed the observation remains 0.

Observation	Code
Agent has not performed listen action yes	0
Tiger behind left door	1
Tiger behind right door	2

The 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

4.3 TicTacToe

In this 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

4.4 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.

4.5 PacMan

This is game of PacMan in which the agent can only partially observer 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 very free cell has a 50% chance of having food in it. The agent receives 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.