Research School of Computer Science

College of Engineering and
Computer Science

An Implementation of MC-AIXI-CTW

Jarrah Bloomfield* Luke English† Andrew Haigh‡ Joshua Nelson§ Anthony Voutas¶

COMP4620 - Advanced Artificial Intelligence
Assignment 2

October 31, 2012

u4669875* u4667010† u4667844‡ u4850020§ u4519169¶

Chapter 1

Description of the MC-AIXI-CTW implementation

The AIXI agent is a formal, mathematical agent which represents a solution to the general reinforcement learning problem. Principally, AIXI consists of an expectimax search over a Bayesian mixture of Turing machines in order to choose optimal actions by predicting future observations and rewards based on past experience. Given infinite computational resources, AIXI represents the optimal reinforcement agent: maximising future expected reward in any unknown environment.

In the far more limited world of what is tractable, we require an approximation to AIXI. Here, we approximate AIXI via a combination of UCT search (Monte-Carlo Tree Search with the Upper Confidence Bound) (Kocsis and Szepesvári, 2006) and Context Tree Weighting (Willems et al., 1995), yielding MC-AIXI-CTW.

Chapter 2

User Manual

2.1 Arguments

The agent can be compile with the `make` command. The agent then can then be run using

```
./main <environment> <logfile>
```

`<environment>` is a compulsory argument, which specifies the environment configuration file the agent is to use. In this implementation, it is one of the following

- `coinflip.conf`: Biased coin flip enviroment
- `grid.conf`: Gridworld environment
- `kuhnpoker.conf`: Kuhn poker environment
- `pacman.conf`: Pacman environment
- `rps.conf`: Biased rock paper scissors environment
- `tiger.conf`: "Tiger" Environment
- `composite.conf`: A combination of the above environments

`<logfile>` is an optional argument, which specifies the name of a log file to output results to.

2.2 Configuration files

`.conf` files are *configuration files*, specifying which environment is to be used, and relevant parameters for each environment. Each configuration file has the following parameters

- `environment`: The name of the environment to use. One of {4x4-grid, kuhn-poker, tiger, biased-rock-paper-scissor, pacman, composite}.
- `exploration`: The rate at which the agent explores, by making random decisions.
- `explore-decay` : The rate at the exploration rate decreases

In addition to this, some configurations have parameters that are specific to their environments.

- Coinflip
  - `coin-flip-p`: The probability of a flipping heads ($0 \leq$ `coin-flip-p` ≤ 1).
- Kuhn poker

  - `gamma`: A constant that determines the environment's Nash equilibrium strategy. ($0 \leq$ `gamma` ≤ 1)

- Pacman
  - `mapfile`: The location of the map file for the pacman board.

- tiger.conf
  - `left-door-p`: The probability that the gold is behind the left door
  - `listen-p`: The probability that a listening observation is correct.

- composite.conf
  - `environmentN`: Specifies the N^{th} environment, where $0 \leq N \leq 10$. The value of this parameter is an integer ≤ 10, and indicates which environment environmentN represents.
  - `startN`: Specifies the time step that at which the N^{th} environment starts, where $0 \leq N \leq 10$.
  - Paremeters required for the environemnts 1..N specified in `environment.cpp`.

Chapter 3

MC-AIXI-CTW Implementation

3.1 Design Choices

We have implemented MC-AIXI-CTW in the C++ coding language, which has a number of benefits and drawbacks.

3.2 Context Tree Weighting

The algorithm for implementing context-tree weighting (CTW) mainly consists of the methods and objects given in the file `predict.cpp`, along with its corresponding `.hpp` file. We run through these from the beginning of the file to its end.

3.2.1 Objects

- `CTNode`

  This object represents a node in the context tree; and so it needs to store both the Laplacian estimate of the probability, along with the weight given to it by the actual CTW algorithm. For mathematical stability, we store these floating-point numbers as logarithms, so we can just add them instead of multiplying them. In addition, it has a 2-array of pointers to its left and right child, and it stores the counts for these nodes in the context of the history (which is also required for the CTW algorithm).

- `ContextTree`

  Here we have the entire context tree represented as an object, which is a glorified pointer to the root node of the tree (as `CTNodes` store their own children). In addition to storing the root, `ContextTree` also stores a double-ended queue of symbols (boolean integers) which represents the current history of the tree, and an unsigned integer which represents the maximum depth of the tree (so that we don't add nodes past the depth, and take up more memory than we want).

3.2.2 Methods

`CTNode`

- `logProbWeighted, logProbEstimated, visits, child`

  These methods are simply defined. Since the values stored in variables such as `m_log_prob_weighted` are declared privately, it is required to use a public method to get them. `visits` grabs the sum of the counts for the child nodes - the number of times that we have seen this particular node in our travels; and `child` takes a symbol, and returns a pointer to the correct (left or right) child.

- `update`

  `update` is one of the most important methods in the entire CTW implementation. We recurse through the tree, starting from the node that calls this method, popping symbols from the end of the history until we reach a leaf; as in the CTW algorithm. When we do, we update the counts for the leaf based on the symbol that is passed to `update`, and then follow the branch back up the tree, using the equation given for P_w:

  $$P_w = \frac{1}{2} P_{KT}(n) + \frac{1}{2} P^0 P^1.$$

  This method doesn't change the history, as we push the popped symbols back onto the stack when we're done with each call.

- `size`

  This is a simple method; it counts the total size of the subtree with the calling node as the root.

- `logKTMul`

  This method calculates the Laplacian estimator P_{KT} for the node, given a symbol $n \in \{0, 1\}$, where

  $$P_{KT}(n) = \frac{\#n + 0.5}{\#n + \#(1 - n) + 1}.$$

  This forms the base multiplier for the probabilities in the context tree.

- `revert`

  If `update` is the method which adds a symbol to the context tree, then `revert` is the method which removes it. We again recurse down the tree, removing the symbol from the leaf, and pushing the changes back up the tree using the current history. Again, we don't change the history with this method.

- `prettyPrintNode`

  Mainly used for testing, this method prints the counts and probabilities at a node, indented given the depth in the tree.

`ContextTree`

- `clear`

  Instead of using the generic destructor `ContextTree`, this method simply deletes the existing history and disassociates the pointer to the root, making a new root and associating the tree with that root instead.

- `update`

  This is an overloaded method, which can either take in a symbol, or a list of symbols. If passed a list, it simply runs itself on each of the symbols in turn; and if passed a symbol, it first checks whether we have enough pre-history to generate a tree. If we don't, it simply pushes the symbol to the back of the history (where we can easily access it again), and if we do, then we need to actually change nodes; so we call the `update` method from `CTNode` instead, before pushing the symbol onto the history.

- `updateHistory`

  Here we simply push a list of symbols onto the back of the history.

- `revert`

  This version of `revert` pops the last symbol off the history, and uses it to pass to the `CTNode` version. It is useful because only the `ContextTree` has access to the variable `m_history`.

- `revertHistory`

  Unlike `revert`, this method takes in a size to revert back to (this corresponds to a previous age of the context tree), and simply pops symbols from the history until we get back to that size.

- `genRandomSymbols`

  This method utilises the following `genRandomSymbolsAndUpdate` method to make a string of random symbols on the history, utilising `revert` in order to remove the changes that the following method makes to the tree.

- `genRandomSymbolsAndUpdate`

  As the name implies, this method uses the context-tree weights to form a random distribution, upon which it generates `bits` bits and pushes them to the given symbol list, and then updates the context-tree weightings.

- `logBlockProbability`

  This is a simple getter method for the weighted probability of the root - which is the probability of the entire context tree.

- `nthHistorySymbol`

  Again, this method is aptly named: it returns the nth most recent history symbol. (Of course, when it can't find a symbol back that far, it just returns `NULL` instead.)

- `depth, historySize, size`

  These are simple getter methods for the private variables contained within a `ContextTree`.

- `predictNext`

  This method uses the weighted context-tree to make an educated guess about what the next symbol may be. It predicts the probability of a 1 being the next symbol, then uses a random generator to figure out whether it should guess 1 or 0.

- `prettyPrint, printHistory`

  These methods are printing methods - `prettyPrint` uses the `prettyPrintNode` method from `CTNode` to recursively print the entire tree, indented by level; and `printHistory` just prints a single string of each symbol in the history, with spaces for separators.

3.3 Upper Confidence Tree

3.4 Revert Function

Chapter 4

Experimentation

4.1 Sequence prediction

The CTW tree is the primarmy prediction mechanism in the MC-AIXI-CTW model. The CTW implementation of MC-AIXI-CTW was tested in isolation from the rest of the agent on a range of deterministic and non-deterministic sequence. This was useful for debugging purposes, and the results are an interesting biproduct of MC-AIXI-CTW.

4.1.1 Deterministic sequence prediction

Several simple sequences were given to the CTW tree, and the CTW tree was asked to continue the sequence. This can be seen in Figure 4.1. We can see that the CTW tree is able to correctly predict the next bit, or multiple bits, in the sequence.

4.1.2 Non deterministic sequence prediction

A partially non deterministic sequence was given to CTW for testing purposes. The sequence was given to CTW was of the form

$$S := x_0^{\text{obs}},\ x_0^{\text{rew}},\ x_1^{\text{act}},\ x_1^{\text{obs}},\ x_1^{\text{rew}}, x_2^{\text{act}},\ x_2^{\text{obs}},\ x_2^{\text{rew}}, \ldots$$

Where

$$x_i^{\text{act}} := r_{\text{act}}, \quad x_i^{\text{obs}} := r_{\text{obs}}, \quad \text{and} \quad x_i^{\text{rew}} := \begin{cases} 0 & \text{if } x_i^{\text{obs}} = x_i^{\text{act}} \\ 1 & \text{if } x_i^{\text{obs}} \neq x_i^{\text{act}} \end{cases}$$

Where

$$r_{\text{act}}, r_{\text{obs}} \text{ are random bits}$$

Sequence	Prediction	Sequence	Prediction
0^{1000}	0...	1^{1000}	1...
$(01)^{1000}$	01...	$(10)^{1000}$	10...
$(0110)^{500}$	0110...	$(1100)^{500}$	1100...
$(110)^{500}$	110...	$(001)^{500}$	001...

Figure 4.1: Some of the sequences given to the CTW tree, and the prediction of the next symbol.

	$x^{\text{rew}}_{3000} = 0$	$x^{\text{rew}}_{3000} = 1$
$x^{\text{rew}}_{3000} = 0$	1	0
$x^{\text{rew}}_{3000} = 1$	0	1

Figure 4.2: Predicted values for x^{rew}_{3000}

The idea of this sequence S is to simulate a coinflip environment history sequence. The random bits x^{obs}_i and x^{act}_i simulate random coin flips, and the reward bit x^{act}_i compares them. The action bit x^{act}_i is random in this sequence.

A sequence of this type with size 9000 (3000 iterations) is given to the CTW tree, up to the point

$$S = x^{\text{obs}}_0,\ x^{\text{rew}}_0,\ x^{\text{act}}_1,\quad \ldots\quad ,x^{\text{act}}_{3000},\ x^{\text{obs}}_{3000}$$

The CTW tree is then asked to predict which bit is the next in the sequence. The idea of this experiment is to determine if the CTW tree "understands" the rules of the game - if it understood, it would predict $x^{\text{rew}}_{3000} = (x^{\text{act}}_{3000} \wedge x^{\text{obs}}_{3000}) \vee (\neg x^{\text{act}}_{3000} \wedge \neg x^{\text{obs}}_{3000})$.

The results of this experiment are show in Figure 4.2. We see that it correctly predicts the reward bit given the action and observation bits. This provides some verification that the implementation of the CTW tree is correct.

4.2 Coin flip results

Coin flip is a simple environment where the agent is given a reward of 1 for correctly guessing the next bit, where the next bit is random with a certain bias such that the next bit is 1 with probability $\theta \in [0, 1]$, else the reward is 0. For a regular coin, $\theta = 0.5$, and there is no dominant strategy. However, if $\theta \neq 0.5$, dominant strategies emerge ($\theta > 0.5 \Rightarrow$ predict 1, $\theta < 0.5 \Rightarrow$ predict 0). The variable θ is hidden from the agent.

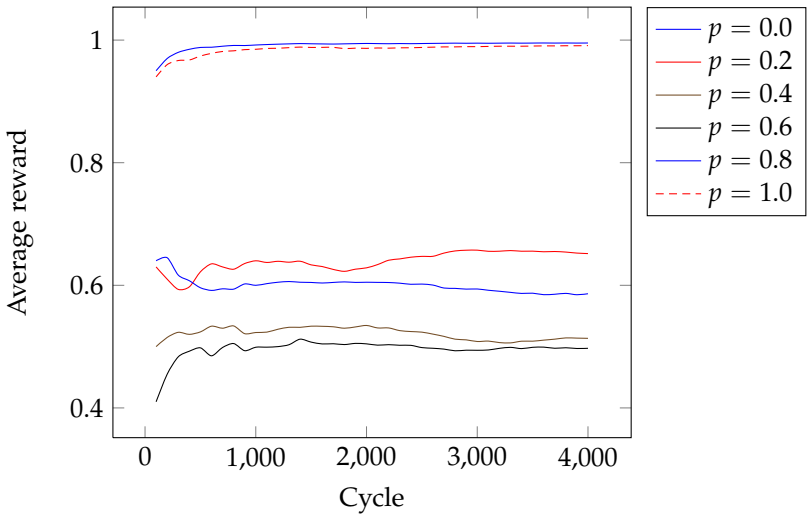

Figure 4.3: Coin flip results

4.3 Tiger results

The parameters used were:

mc-simulations	50
exploration	0.01
explore-decay	0.999

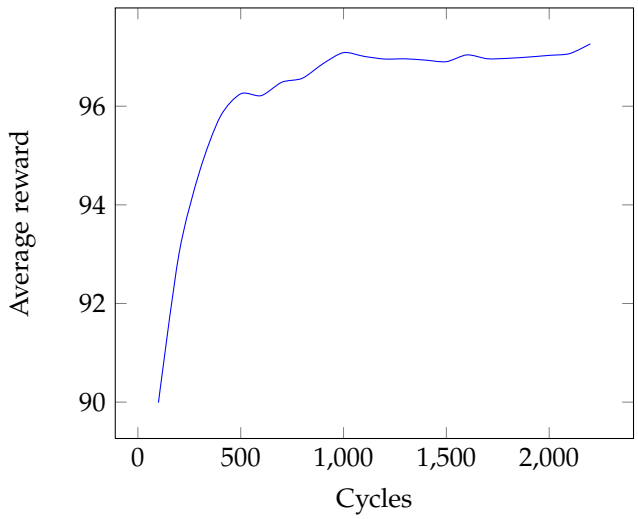

Figure 4.4: Tiger results

4.4 Grid world results

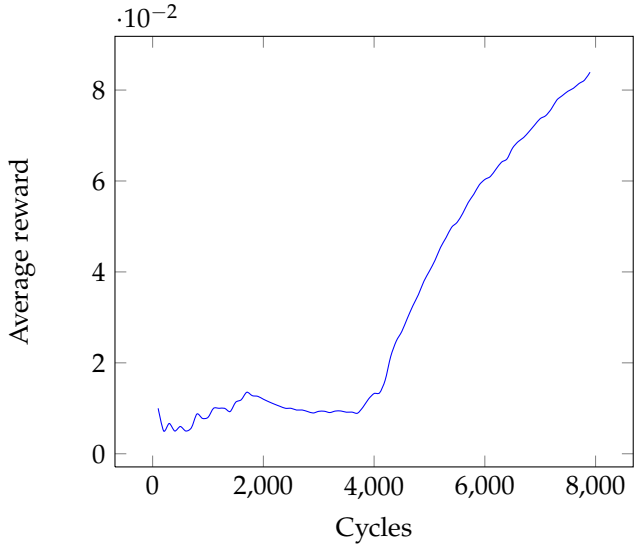

Figure 4.5: Grid world results

mc-simulations	50
exploration	0.01
explore-decay	0.99999

4.5 Biased Rock paper scissors results

In Biased Rock Paper Scissors (RPS), the agent plays against an environment which will randomly select rock, paper or scissors; but after winning with rock, the environment will repeat a rock move. This is interesting because it creates a bias in the environment's distribution, and the agent needs to try to exploit this bias.

The parameters used were:

mc-simulations	50
exploration	0.1
explore-decay	0.999

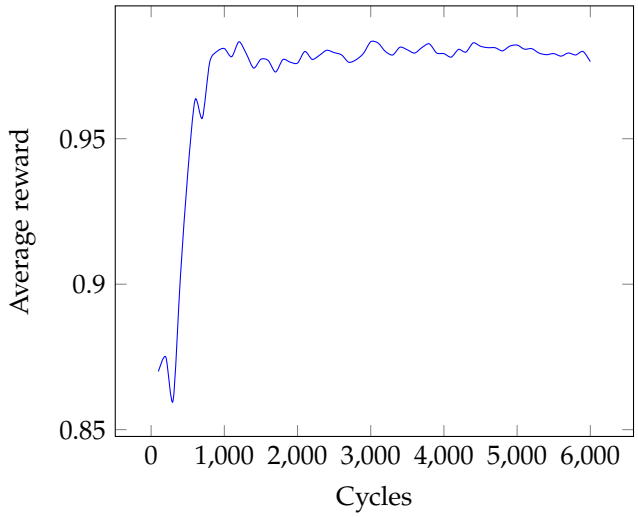

Figure 4.6: Biased RPS results

4.6 Kuhn Poker results

In Kuhn Poker, the agent plays against a Nash Equillibruim AI, with gamma set to 0.5. The agent is given a reward of 0 for a loss, 2 for a win or 1 if no showdown has occurred so far.

The parameters used were:

mc-simulations	50
exploration	0.01
explore-decay	0.9999

4.7 Pacman results

In Pacman

The parameters used were:

mc-simulations	50
exploration	0.01
explore-decay	0.99999

Table 5.1 Number of problems where complete solutions were found

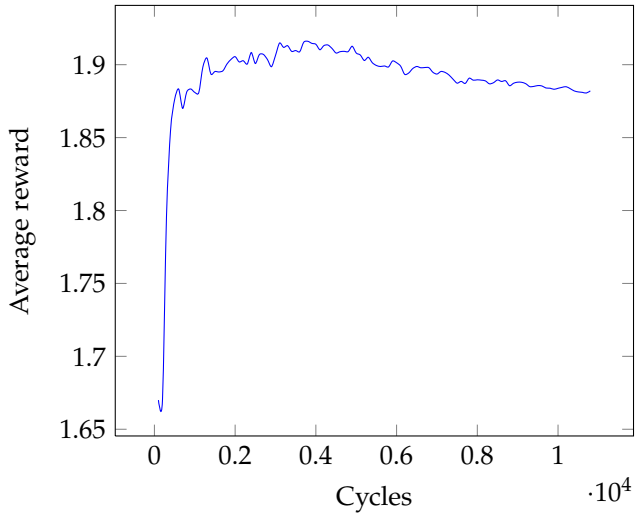

Figure 4.7: Kuhn poker results

The parameters used were:

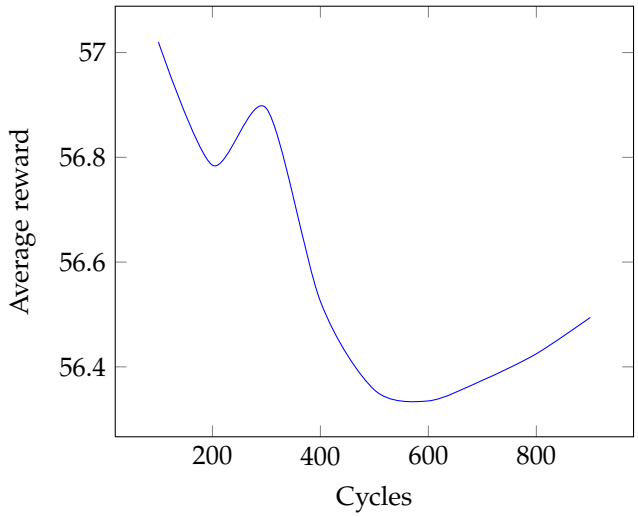

Figure 4.8: Pacman results

Bibliography

L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. *Machine Learning: ECML 2006*, pages 282–293, 2006.

F.M.J. Willems, Y.M. Shtarkov, and T.J. Tjalkens. The context-tree weighting method: Basic properties. *Information Theory, IEEE Transactions on*, 41(3):653–664, 1995.