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

u4838292[*] u4667844[†] u4667010[‡] u4850020[§] u4519169[¶]

Contents

Chapter 1

Introduction

The field of Artificial Intelligence (AI) is involved in the study and development of rationally intelligent agents that have been deliberately constructed. A recent trend in AI has been moving from agents who attempt to optimise their performance in a particular environment towards agents who attempt to optimise their performance in a wide range of environments. This field of study is known as Artificial General Intelligence. Within this paper, we describe an implementation of the AIXI agent from Universal AI (Hutter, 2005).

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. AIXI interacts with an environment in discrete cycles $k = 1, 2, \dots, m$ by taking an action a_k and receiving an observation o_k and reward r_k. Then AIXI is formally defined by

$$a_k := \arg\max_{a_k} \sum_{o_k r_k} \cdots \max_{a_m} \sum_{o_m r_m} [r_k + \cdots + r_m] \xi(o_1 r_1 \cdots o_m r_m \mid a_1 \cdots a_m),$$

where $\xi(x_{1:n} \mid a_{1:n}) = \sum_{\nu \in \mathcal{M}} w_0^{\nu} \nu(x_{1:n} \mid a_{1:n})$ is a mixture environment model, $\mathcal{M}$ is a model class of all chronological probability distributions, and $w_0^{\nu} > 0$ is a prior weight for each ν such that $\sum_{\nu \in \mathcal{M}} = 1$.

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`: Grid World 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++ programming language, which has a number of benefits. Firstly, C++ is a better option for implementation than C or another imperative language, as objects allow us to pass recursive and enumerated structures between processes. Secondly, C++ is a low-level language - more than any other object-oriented language, which allows us to process more optimally; and dynamic memory allocation means that we can ensure that our program doesn't take up more memory than we need it to, so that it will take less time to run overall.

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; 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 add them instead of multiplying them. In addition, it has a pair 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 `CTNode`s 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:

  $$P_w = \begin{cases} P_{KT}(n) & \text{for leaves} \\ \frac{1}{2}P_{KT}(n) + \frac{1}{2}P^0P^1 & \text{otherwise} \end{cases}$$

  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

Most of the Upper Confidence Tree (UCT) algorithm is implemented in the files `search.cpp`. The UCT algorithm assumes we have some model of the environment (CTW) and searches this model for the best action.

3.3.1 Objects

- `SearchNode`

  This object represents a node in the search tree. The `SearchNodes` form an expectimax tree, alternating between decision and chance nodes. `SearchNode` contains most of the methods related to traversing this tree, and calculating optimal actions from it.

- `ModelUndo`

  This object represents a snapshot of an agent's model of the environment at a given time. The agent includes functions to revert its model (the CTW tree) to the time at which the `ModelUndo` was created. This functionality is necessary in the `sample` function, as it makes changes to the CTW tree that need to be undone.

3.3.2 Methods

`SearchNode`

- `selectAction`

  Determines the next action to play, starting from this search node. `selectAction` looks through the possible actions it could take from this point, and chooses the action that would yield the best expected score (based on the sampling this node in the `sample` function). Included in this expected score is the `ucb_bound` variable, which gives a bonus to nodes that have not been well explored.

- `sample`

  Performs a trial run through this node, and through its children. `sample` calculates the expected reward along the way, which updates the expected score (`m_mean`) value for `selectAction`. This function generates actions to take as it travels down the tree using the CTW tree (`agent::genPerceptAndUpdate` and `agent::modelUpdate`).

  If `SearchNode` is a chance node, it generates an observation and reward from the CTW tree, and continues down through `SearchNode`'s children. If it is a decision node, it uses `SelectAction` to choose an action. On the first visit, it uses the `playout` function to determine a reward.

- `playout`

  Generates actions randomly, and updates the CTW tree based on them. Used in the first run through of the `sample` function.

`ModelUndo`

- `ModelUndo`

  The constructor for a `ModelUndo` takes the age, reward, history size, and last `SearchNode` type (chance or decision) of an agent and saves this information. This information is sufficient for an agent to revert itself to the time at which the `ModeUndo` was made.

- `Agent::modelRevert`

  Takes a `ModelUndo` object and reverts an agent's model to the state it was at when this `ModelUndo` was taken. It iterates through the history and rewinds objects, actions, and rewards seen, peeling them off the history array, and reverting the CTW tree to reflect this removal of information (performs the reverse of an agent's model update procedure).

General functions

- `search`

  The top level function for determining an action based on an agent's state. `search` forms the search tree as it searches. Using the `SearchNode::sample` function, `search` samples from the root multiple times to form the MCTS (Monte Carlo Tree Search) tree. This function also uses `ModelUndo` to save an agent before modifying it and its CTW tree. It can then revert it back to its previous state before the sampling run took place.

  After the tree has been explored sufficiently (`numSimulations`), the best action is determined by taking the maximum expected reward over all of the possible actions. This best action is the one that search returns.

Chapter 4

Experimentation

4.1 Sequence prediction

The CTW tree is the primary 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

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

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...

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

	$x_{3000}^{\text{rew}} = 0$	$x_{3000}^{\text{rew}} = 1$
$x_{3000}^{\text{rew}} = 0$	1	0
$x_{3000}^{\text{rew}} = 1$	0	1

Table 4.2: Predicted values for x_{3000}^{rew}.

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

$$S = x_0^{\text{obs}},\ x_0^{\text{rew}},\ x_1^{\text{act}},\quad \dots\quad ,x_{3000}^{\text{act}},\ x_{3000}^{\text{obs}}.$$

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

$$x_{3000}^{\text{rew}} = (x_{3000}^{\text{act}} \wedge x_{3000}^{\text{obs}}) \vee (\neg x_{3000}^{\text{act}} \wedge \neg x_{3000}^{\text{obs}}).$$

The results of this experiment are shown in Table 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.

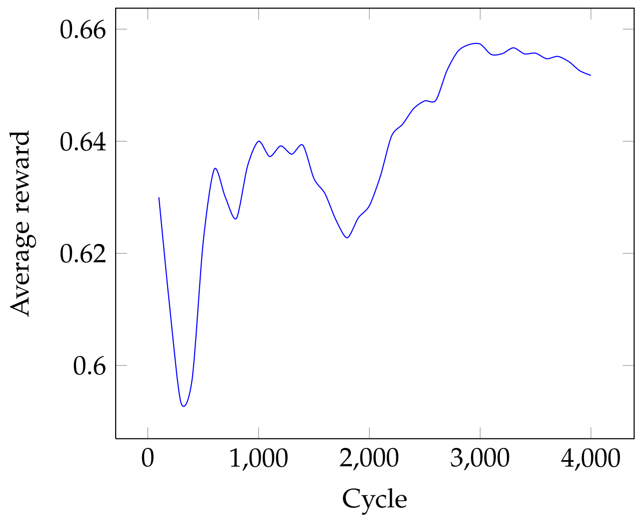

Figure 4.1: Coin Flip simulation for $\theta = 0.2$.

Figure 4.1 shows the results for $\theta = 0.2$. The theoretical best strategy for this environment would be to predict 0 always, and gain a reward of 0.8. We can see that our implementation of

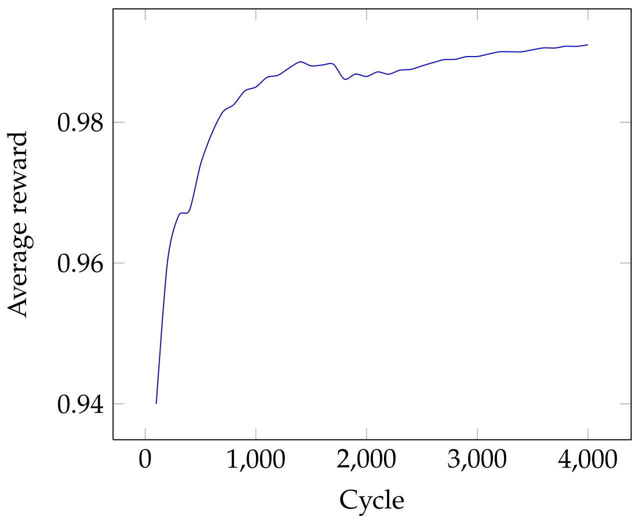

Figure 4.2: Coin Flip simulation for $\theta = 1.0$.

MC-AIXI-CTW acheives better than a random player (who would receive an average reward of 0.5) but does not acheive the maximum theoretical average reward in 8000 cycles.

Another graph with $\theta = 1$ is shown in Figure 4.2. With $\theta = 1$, the maximum theoretical reward is 1, with the strategy of picking 1 always. We can see that our implementation of MC-AIXI-CTW quickly converges to this strategy of picking 1 always, with the average reward approaching 1.

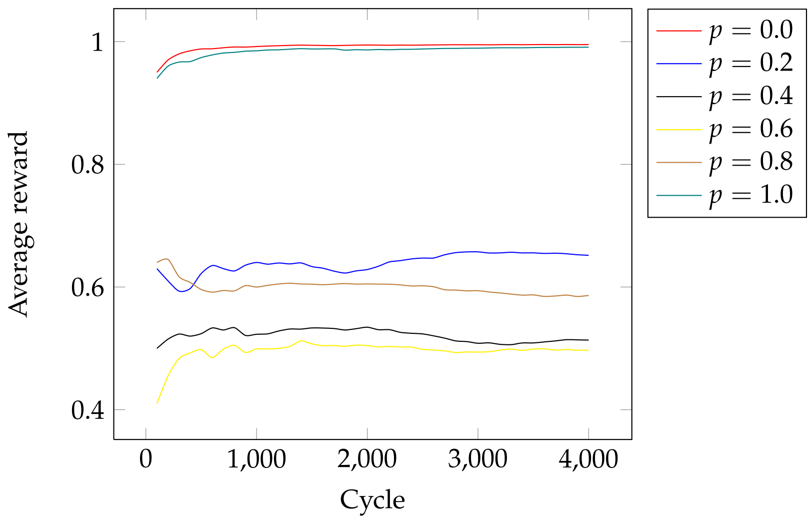

Figure 4.3: Coin Flip results.

We also include in Figure 4.3 a graph of all results for MC-AIXI-CTW with p ranging from 0 to 1. Note the discrepancy that the agent has difficulty correctly exploting p=0.8 or o=0.2, settling on 0.6 reward. This is likely caused by the extra depth of the context tree confusing the algorithm by asking it to store useless information, which in some sense 'dilutes' the precision. This greatly increases the number of cycles required for convergence. p=1 or p=0 converge well because the sequence becomes completely deterministic, greatly reducing the difficulty required to predict the next bits of the sequence.

4.3 Tiger results

Tiger is a game where the agent chooses out of two doors. Each door has a 50% chance to contain a moderate reward of 110, or contain a reward of 0. A reward of 99 is given for listening, which has an 85% chance to correctly reveal (as an observation) the size of the reward behind the door. This is a non-deterministic environment. Tiger is an interesting environment because effective exploitation requires patience. Selecting a door immediately will result in a far lower expected reward than listening until 3 confirmations that a door does not contain the tiger (low reward). Because of this, the optimal action sequence is long, even though greedily picking a door immediately can result in high reward.

mc-simulations	50
ct-depth	30
agent-horizon	4
exploration	0.01
explore-decay	0.999

Table 4.3: The parameters used in the Tiger environment.

The CT depth was chosen to be 30 so as to be large enough to store several percept and reward sequences. The agent horizon was set to 4 in order to enable optimum play (listen 3 times then choose the door. The exploration is quite low but decays slowly; this is to assist the agent in discovering that while listening has low immediate reward, it can increase future reward.

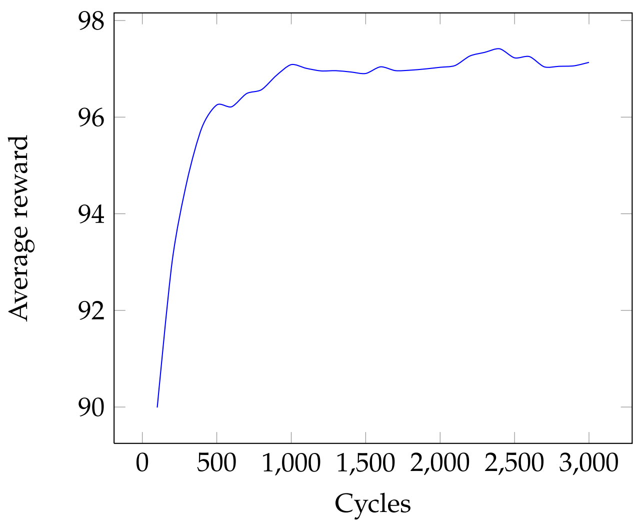

Figure 4.4: Tiger results.

The initial learning shows a steep curve, and the agent learns after 500 cycles that listening is gives safe reward (as a reward of 99 is better than picking a random door resulting in an average reward of 55). From this point onwards the agent seems to begin to notice the correlation between the percept provided by listen and the reward given by a door, and there is another small spike. The agent has likely had insufficient time to learn, so the reward has not converged, and is still far from the theoretical optimum of 107 average reward. This is caused by the difficulty in learning the observation reward sequence in the CTW; as the number of reward bits is 7, CTW learning takes a long time.

4.4 Grid World results

Grid World is a world where a blind agent (never given any observation) in a 4x4 grid needs to reach the bottom right corner, which will grant a reward of 1. All other rewards are 0. Moving outside the grid returns the agent back to the nearest square inside the grid. Once reaching the bottom right corner, the agent is teleported to a random other square in the grid. Note that under the specification, the agent may be teleported to the bottom right corne. The agent will not receive extra reward for this, but has an advantage in that either RIGHT or DOWN will result in an immediate reward. The environment is non-deterministic, however, contains an optimal sequence of moves that is guaranteed a reward within 6 moves, namely RIGHT, DOWN repeated 3 times, which would give an average reward of 0.33. The parameters were chosen such that the agent has enough depth in the CT to store the optimum strategy (depth 12), and will search a reasonable distance in the future (given the small environment).

mc-simulations	50
ct-depth	12
agent-horizon	4
exploration	0.01
explore-decay	0.99999

Table 4.4: The parameters used in the Grid World environment.

These parameters are chosen such that the agent has enough depth in the CT to store the optimum strategy (depth 12), and will search a reasonable distance in the future (given the small environment). The exploration was chosen to be low in order to promote early exploitation so the agent would find a good strategy quickly. The decay rate is set low to allow the agent to keep exploring in order to optimise strategies.

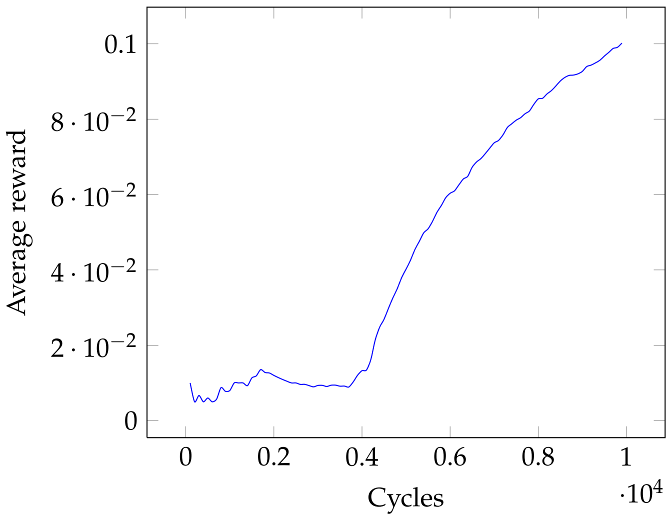

Figure 4.5: Grid World results.

The results for this experiment are particularly interesting because the graph shows the agent clearly wandering around lost for the first 4000 cycles, but then as the agent starts to correctly identify the better strategy, the reward sharply increases. This creates a snowball effect; better strategies are found from making explorations on good strategies. As shown, the agent strategy has clearly not converged yet and needs more time. The actions taken shows the agent clearly favouring sequences with RIGHT and DOWN, dramatically increasing the average reward. The average reward continues to increase but has not quite reached the optimum yet.

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 environment is non-deterministic. Note that optimum play on this environment would involve the agent playing scissors until a loss, then playing paper directly afterwards.

mc-simulations	50
ct-depth	8
agent-horizon	3
exploration	0.1
explore-decay	0.999

Table 4.5: The parameters used in the RPS environment.

The CT depth is chosen such that 2 observation and reward pairs can be stored, which will enable optimal strategy. The agent horizon is 3 for similar reasoning, slightly longer than the horizon required for optimal strategy, to ensure that the agent isn't immediately unfairly biased towards optimal strategy. The exploration is set to be quite high with a slow decay to enforce some random play from the agent in order to evaluate all the different possible combinations.

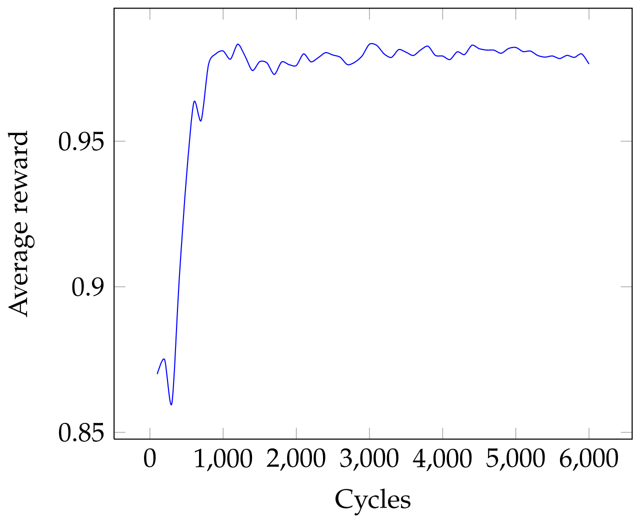

Figure 4.6: Biased RPS results.

The results show that the agent is initially thrown off by the curious distribution of the environment, actually performing worse than random. While this may seem odd, it must be kept in mind that the CTW will initially perform poorly for non-deterministic sequence prediction as it can be thrown off by chance. If the agent is at a point where it predicts that scissors is the best move, it is more likely to predict that scissors will be the best next move as well. The problem is that the bias of playing two rocks in a row is quite subtle and will take a while to learn the correct paths in the CTW, making it difficult to learn that a loss on scissors must be immediately followed by paper. A sharp learning curve happens in iterations 200-600 as the agent learns how to handle this distribution (by not playing scissors after losing on scissors) and the average reward becomes close to 1 (reward from performing random). However, even after 6000 iterations, the agent still hasn't managed to learn that a loss on scissors should be followed by paper for optimum reward.

mc-simulations	500
ct-depth	32
agent-horizon	4
exploration	0.001
explore-decay	0.999999

Table 4.6: New parameters for biased RPS.

To provide MC-AIXI-CTW with enough leeway to store sufficient information to model the more complicated Markov process in RPS, an extensive simulation with was run with the parameters specified as in Table 4.6.

This yielded an average reward per cycle that can be seen in Figure 4.7, showing that the agent performs better than random play, but still does not achieve the optimal reward of 1.25 per cycle.

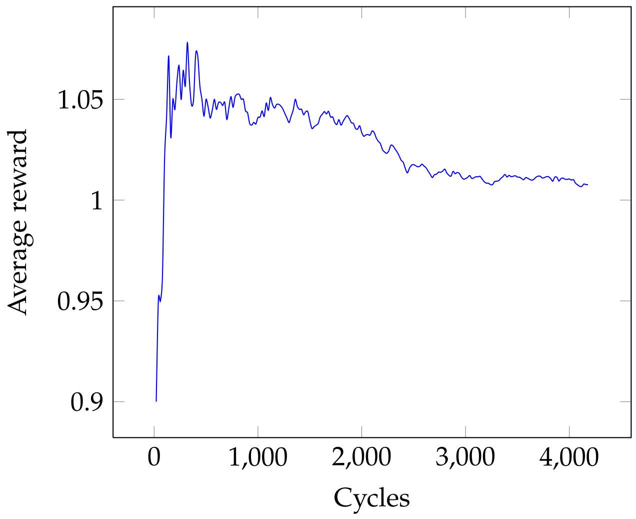

Figure 4.7: Biased RPS results with larger CTW depth.

4.6 Kuhn Poker results

In Kuhn Poker, the agent plays against a Nash Equilibrium AI, with gamma set to 0.5. More details about the environment can be found in Hoehn et al. (2005). The agent is given a reward of 0 for a loss, 2 for a win or 1 if no showdown has occurred so far.

mc-simulations	50
ct-depth	12
agent-horizon	2
exploration	0.01
explore-decay	0.9999

Table 4.7: The parameters used in the Kuhn Poker environment.

The CT depth was chosen in order to be slightly more than enough to play optimally. The horizon was chosen in order to enable optimal strategy. While this could possibly be set to 3 to make sure there is no unfair bias, this is reasonable because the maximum iterations per environment cycle are 2. The exploration is set to be initially low and decay slowly in order to increase the agent's ability to learn the non-deterministic environment.

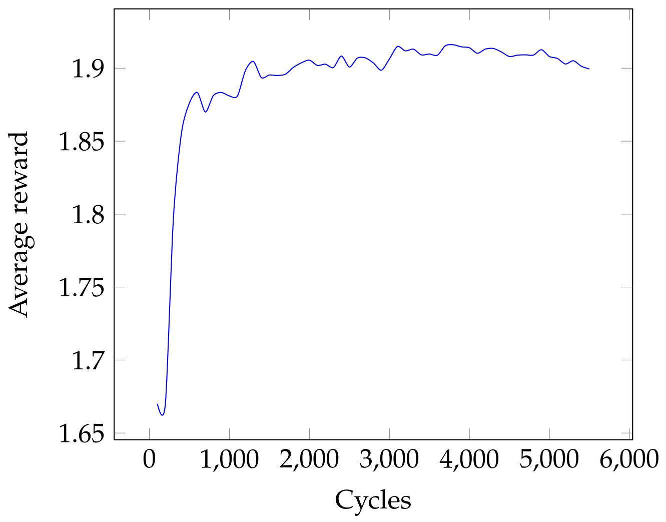

Figure 4.8: Kuhn Poker results.

Kuhn Poker initially performs poorly against the Nash player, losing significantly more often than winning. However, interestingly, within 800 cycles, the agent begins to learns how to play the basic game (such as betting when seeing a king and passing with a jack), seen by the steep improvement which cuts out just above 1.87. Some finer points are learned up to 4000 cycles, possibly such as bluffing with a queen, reaching an average reward of 1.9. However, the agent does not have enough time to start learning how to exploit the Nash player and still loses more times than it wins.

4.7 Pacman results

In partially observable Pacman, the agent plays in a maze, getting reward for consuming food but losing reward for running into walls and getting eaten by ghosts, according to the classical Pacman game. However, the agent cannot see the entire map, with a 16 bit percept based on 4 bits for line of sight for food, 3 bits for smelling food (based on manhattan distances of 2, 3 and 4), 4 bits for line of sight of ghosts, and finally 1 bit for the power pill. The environment is non-deterministic because the ghosts move randomly if not chasing pacman, and food is randomly allocated.

Several assumptions were made on top of the specification. While under the influence of the power pill, the agent is able to eat ghosts instead. Note that there is no reward given for this, unlike the classic pacman game. Ghosts also do not respawn, differing from traditional pacman. The effect this has is the agent should learn that if it eats a power pill, a good strategy is to try to eat ghosts, which will maximise future reward. However, given the limitations of balancing search size against computational limits, it is unlikely that this will be seen to a large degree.

The rewards were more complicated because the agent can trigger multiple events with one action and thus receive multiple rewards. A reward of -60 was given for running intot a ghost, -10 for a wall, -1 for moving, 10 for eating a pellet and 100 for finishing all pellets. Multiple events had their rewards added together, and 71 was added to the total to ensure the reward remained positive. Ghosts will do a depth first search on possible moves in order to find pacman, and if he is within 5 tiles, they will agressively pursue him for 10 moves, after which they will forget about him for 10 moves. Ghosts do not flee from pacman with a power pill.

Whilst the CT depth and horizon chosen as parameters for pacman are far below the sizes required to play optimally, searching the depth of an entire game would be computationally infeasible. The parameters were chosen in order to balance computational time required. The agent horizon of 4 is very small but can still enable pacman to learn some advantageous behaviours

mc-simulations	50
ct-depth	32
agent-horizon	4
exploration	0.01
explore-decay	0.99999

Table 4.8: The parameters used in the Pacman environment.

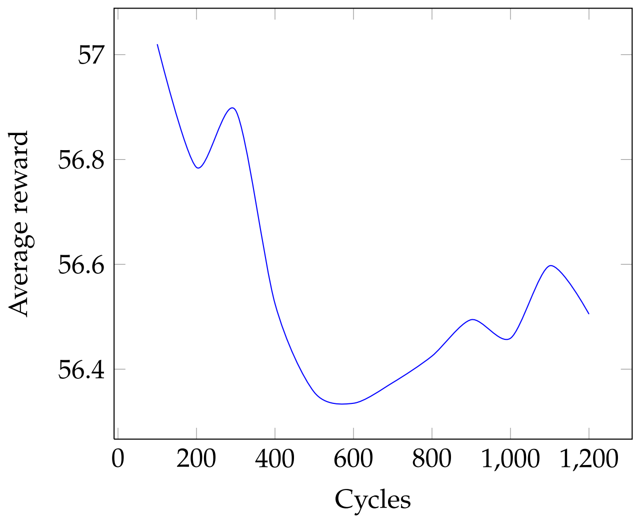

Figure 4.9: Pacman results.

in the environment, such as not walking into the direction where a ghost can be seen in line of sight. The exploration rate was set low in order to make the agent try to exploit the environment early on. The decay is low so that the agent continues to explore for a long time.

The results show that the agent clearly did not have enough cycles to learn optimal behaviours. The average reward shows the agent has considerable difficulty learning not to run into walls or ghosts in the first 600 cycles. However after this it seems to begin to have enough information in the context tree to predict suboptimal reward for running into walls in one or two directions; this causes the learning curve to become bumpy.

4.8 Composite environments

A composite environment is an environment which behaves like another environment for a certain number of time steps and then switches to behaving like another environment. In this way it becomes possible to examine whether the agent can develop transfer knowledge from one environment to another. It is also possible to examine whether the agent will remember the original environment when reintroduced to it, and therefore more quickly conform to optimal behaviour in that environment the second time it is introduced to it. The first test is to change between coinflip environments with different weighted coins. The second is to change from a coinflip environment to a tiger environment, and then back to the original coinflip environment.

In the example case (Figure 4.10), the weights are chosen so that an action of 1 is optimal in the first and third environments and an action of 0 is optimal in the second environment. The agent does not seem to lose too much when the environment is switched, as it quickly learns that tails is a better choice. When it is switched back, it still has a context which says that if it sees a few heads, it should probably pick heads. Note that the environment switches at cycle 500 and then switches back at cycle 1000.

The composite environment must have a standard ct-depth and a standard number of bits

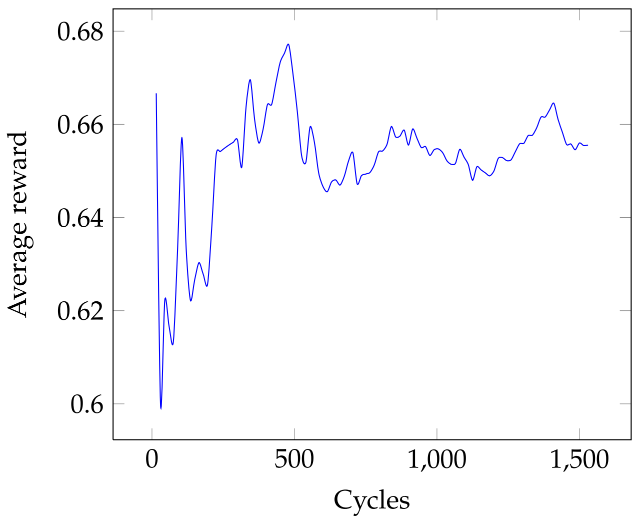

Figure 4.10: Composite Coin Flip environment $p = \{0.9; 0.1; 0.9\}$.

for actions, observations and rewards over all environments that it runs, because the agent takes these values at the start. So, the maximum values (over the tested environments) of these are taken. That is, in the case with just Coin Flip environments, we take just the standard values for Coin Flip. In the case of, say, Coin Flip and Tiger, we take the standard values for Tiger because these are higher. Extra actions not pertinent to the normal environment return the minimum reward over all environments (0 in the case of just Coin Flip environments and -100 in the case of composite environments including Tiger, for example), and are thus discouraged.

As the code for composite environments is generic, we can experiment with running longer and more varied simulations (such as Coin Flip and Tiger, or Grid World and Kuhn Poker), as well as resetting the average reward when the environment changes. This would most likely help to understand the composite environments' effect on the MC-AIXI-CTW learning paradigm further.

Bibliography

B. Hoehn, F. Southey, R.C. Holte, and V. Bulitko. Effective short-term opponent exploitation in simplified poker. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 783. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

M. Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Springer, Berlin, 2005.

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.