Report on MC-AICI-CTW Approximation Implementation and Experiment

Jiayan Liu, Yuxi Liu, Andrew Tanggara, Yan Yang

October 20, 2019

1 Introduction

AIXI, the universal optimal agent, is constructed in [Hut05], but it is incomputable. For practical applications, [Ven+09] reported a computable approximation named MC-AIXI-CTW.

We report a reimplementation of this agent in python, as well as the results of some numerical experiments from running the agent in a few simple environments.

1.1 Reinforcement learning

AIXI is a reinforcement learning (RL) agent. In an RL problem, there is an environment and an agent in an interaction loop. At time n, the agent take an action $a_k \in \mathcal{A}$, and the environment replies an observation $o_k \in \mathcal{O}$ and a reward $r_k \in \mathcal{R}$. The agent attempts to find the action a_k that maximizes its expected future reward, conditional on its future actions,

$$\mathbb{E}(r_k + r_{k+1} + \cdots | a_k + a_{k+1} + \cdots).$$

Since it is in general impossible or impractical for any finite agent to optimize the total reward from infinitely many future steps, it is often assumed that the agent has a finite horizon m. That is, at step k, the agent only attempts to maximize

$$\mathbb{E}(r_k + r_{k+1} + \cdots + r_{k+m} | a_k + a_{k+1} + \cdots + a_{k+m}).$$

In general, most RL agents have two components: learning and planning. The learning component observes past interactions with the environment, and uses them to build an accurate model of the environment. The planning component uses the model of the environment to predict the future effects from actions, and attempts to pick a good action based on such predictions.

1.2 AIXI

AIXI's learning component is a Bayesian learner. It starts with a Bayesian prior ξ over the set of all computable environments $\mathcal{M}_U$. Then, as more data $aor_{<k}$ comes in, it computes the posterior $\xi(|aor_{<k})$ as a model of the environment. The prior ξ gives higher weighting to environments that are simpler in Kolmogorov complexity (this is essentially a form of regularization to prevent the learner from overfitting). From algorithmic complexity theory, for any environment $\mu \in \mathcal{M}_U$, the

posterior $\xi(|aor_{<k})$ quickly converges to μ with high probability. That is, it is very likely that AIXI quickly learns a good model of any computable environment.

AIXI's planning component merely computes the action that maximizes expected future reward. In one formula [Hut05]; [Ven+09],

$$a_k^{\xi} = \arg\max_{a_k} \sum_{or_k} \ldots \max_{a_{k+m}} \sum_{or_{k+m}} (r_k + \cdots + r_{k+m})\, \xi(or_{k:k+m}|aor_{<k}a_{k:k+m}), \tag{1}$$

where

$$\xi(or_{k:k+m}|aor_{<k}a_{k:k+m}) = \sum_{\rho \in \mathcal{M}_U} 2^{-K(\rho)} \rho\,(or_{k:k+m}|aor_{<k}a_{k:k+m}) \tag{2}$$

is the Bayesian prior over computable environments $\rho \in \mathcal{M}_U$, with each having Kolmogorov complexity $K(\rho)$.

1.3 MC-AIXI-CTW

AIXI as formulated above cannot be used in practice, since the planning component uses expectimax search, which is not efficiently computable, and the learning phase uses the incomputable Kolmogorov complexity over all computable environments. MC-AIXI-CTW replaces each of these with computable approximations.

1.3.1 Learning by CTW

Instead of learning by a universal prior ξ over all computable environments, which is incomputable, MC-AIXI-CTW learns by a simplicity prior Υ_D over Prediction Suffix Trees (PST) $M_i \in C_{D_i}$ of maximum depth D_i,

$$\Upsilon(x_{1:k}|a_{1:k}) = \sum_{M \in C_{D_1} \times \ldots C_{D_k}} 2^{-\sum_{i=1}^{k} \Gamma_{D_i}(M_i)} \Pr(x_{1:k}|M, a_{1:k}) \tag{3}$$

with $\Gamma_{D_i}(M_i)$ length of encoding of PST M_i embedded in the weight of each PST, resulting larger weight for smaller PST.

The CTW algorithm [WST95] efficiently updates this prior to posterior as new information is received.

1.3.2 Planning by rho-UCT

For planning, MC-AIXI-CTW uses a variant of the UCT algorithm: Upper Confidence Bound (UCB) applied to Monte Carlo tree search (MCTS) [Aue02].

MCTS is an algorithm explores a search tree randomly. UCB is an algorithm that solves the Multi-Armed Bandit (MAB) problem. In a game of MAB, one is faced with a set of gambles, each having an uncertain payoff in the range $[0, 1]$. The gambles are independent. The problem is to devise an algorithm that efficiently maximizes total average reward over repeated games of MAB.

UCT exploits the fact that at each tree search node of MCTS, one is in essence faced with a game of MAB: each action is a gamble, with reward being the value of the node following the action. UCB can efficiently find a good action in game of MAB, so it can find a good action at a node efficiently too.

The usual UCT is for two-player games, where the players alternate in making moves, each intent on maximizing their own scores. In ρUCT, the opponent of the agent is a stoic force of nature that has no preferences and no payoff. As such, there are two game tree nodes: "decision nodes" where the agent is making a decision, and "chance nodes" where the environment/nature is making a random move without attempting to optimize any kind of score of the game.

1.4 Exploration factor

The actual agent we implemented was not a pure MC-AIXI-CTW agent, but instead could be termed a ϵ-MC-AIXI-CTW agent. That is, in each round, there is a ϵ probability that the agent takes a random available action, and a $(1 - \epsilon)$ probability of using the MC-AIXI-CTW policy to choose its action. Here, ϵ is termed the exploration factor.

The idea is that taking completely random actions once in a while helps the agent to break out of inaccurate self-perpetuating environment models. For example, imagine an environment with two slot machines A and B. Machine B has average payoff μ_B better than that of A, μ_A, but the agent managed to get very bad early experiences with B compared to A. This distorted its environment model to such an extent that its internal model has $\mu'_B < \mu_A$, and the agent would spend a very long time playing A. As its internal model of A become very accurate, it would still think that $\mu_B < \mu_A$.

Throughout the exploration period, ϵ decays as time goes on. This is essentially the same idea as simulated annealing, the general metaheuristic technique in optimization, whereby an optimizer starts with a great deal of random exploration ("hot"), but as time goes on, it becomes less random ("cools down").

The length of the exploration period T, the starting exploration factor ϵ_0, and the exploration-decay factor γ, are all configurable. See Section 2.2.

After the exploration period ends, the agent would still interact with the environment, but it would no longer update its internal model of the environment, in effect freezing its learning component, leaving only its planning component functional. This somewhat unnatural design choice was essentially for diagnostic purposes. The idea being that, after a period of learning about the environment, we let the agent continue interacting in the environment without further learning, thus allowing us to evaluate the quality of the final learned model of the environment without interference due to further learning. If it was found that the final learned model gives an average reward per cycle close to the theoretical optimum, it would be deemed a good model.

1.5 Environments

We implemented four environments taken from [Ven+09], then tested our implementation of MC-AIXI-CTW on them using various configurations.

1.5.1 Cheese Maze

The cheese maze is a 7×5 maze with 11 rooms, with a cheese located at one specific room. The agent is a mouse inside the maze, starting at a specific room. This is shown in Figure 1. The episode restarts when the agent moves into the room of the cheese.

The agent has 4 available actions, encoded in 2 bits: moving up, down, left, or right.

The agent is unaware of its absolute location in the maze. It can only observe whether a wall exists to its up, down, left, and right. This is encoded as a 4-bit observation. The possible observations are when translated from binary to demical, $\{5, 7, 8, 9, 10, 12\}$, shown in Figure 1.

For example, the room with label 9 is observed to have:

1. 1 wall on its top;
2. 0 walls on its right;
3. 0 walls on its down;
4. 1 walls on its left;

giving a binary observation code 1001, which is 9 in decimal.

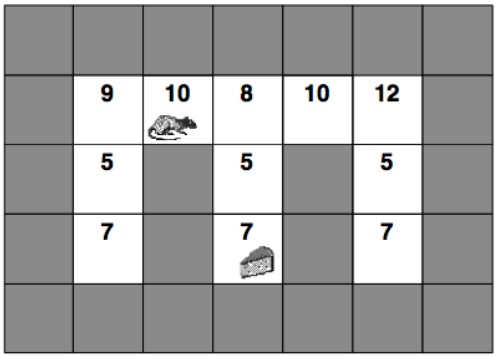


Figure 1: Cheese maze.

The possible rewards range from -10 to 10, encoded in 5 bits:

- -10, if the agent makes an invalid move, that is, attempting to move into a wall.
- 10, if the agent moves into the room of the cheese.
- -1, if the agent moves into a free room.

1.5.2 Extended Tiger

This is a basic extension to the classic test problem Tiger [KLC98].

At the start of each episode, a tiger and a pot of gold are randomly hidden behind one of two doors respectively. The agent starts sitting in a chair. The goal of the agent is to open the door with gold behind. To accomplish that, the agent should listen for the door behind which hides the tiger, then stand up to open the other door.

The agent can be in one of 2 states: sitting and standing.

The agent has 4 available actions, encoded in 2 bits: stand, listen, open left door, open right door.

There are 3 possible observations, encoded in 2 bits: left, right, none.

The possible rewards range from -100 to 30, encoded in 8 bits:

- -100 by choosing the worst action: opening the door with tiger hiding behind.

- -10 by choosing an 'invalid' action: e.g. stand while standing.
- -1 by choosing an 'valid' action: e.g. stand while sitting.
- 30 by choosing the best action: opening the door with gold behind.

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	130
standing	listen	90

Table 1: Rewards for extended tiger.

This is essentially the original Tiger problem with one extra step. Instead of listening until confident, then immediately opening the door, the agent has to listen until confident, then immediately stand up, and open the door.

By using the POMDP solver [Cas], the optimal strategy is found:

$l := 0, r := 0.$
While $|l - r| < 2$:
  Listen.
  If heard tiger on the left:
    $l := l + 1$
  Else:
    $r := r + 1$
Stand up.
Open the left door if $r > l$, else open the right door.

In short, listen until one door is heard to contain the tiger twice more than the other door, then stand up and open the other door.

Using this strategy, it can be calculated that the theoretical optimal average reward per round is $\frac{3485}{549} = 6.34791$.

1.5.3 Partially Observable Pacman

This environment is a partially observable version of the classic PacMan game. The maze structure and game are the same as the original arcade game, however the PacMan agent is hampered by being able to observe the game state only partially.

The agent controls PacMan, who must navigate a 17×17 maze and eat all the food pellets in the maze, and avoid hitting the walls and the ghosts. Eating a power pill turns PacMan into super PacMan. The effect of eating a power pill lasts for 100 steps, during which time PacMan can eat ghosts.

At each step, both the ghost and the PacMan can move 1 step up, down, left, or right. The PacMan must move, while the ghosts can choose to stay stationary. PacMan cannot move into walls. It moves normally into spaces containing nothing, pellets, power pills, and ghosts. When it moves into one end of a magic tunnel, it is immediately moved to its other end.

The agent has 4 available actions, encoded in 2 bits: moving up, down, left, right.

The are 4 ghosts. At each step, each ghost measures its Manhattan distance d from PacMan, and move based on 3 strategies:

- If $d > 5$, move randomly.
- If $d \leq 5$, and PacMan is not super PacMan, move to decrease d.
- If $d \leq 5$, and PacMan is super PacMan, move to increase d.

The observation has 16 bits, structured as follows:

- 4-bit describing the wall configuration at its current location.
- 4-bit indicating whether a ghost is visible (via direct line of sight) in each of its direct line of sight.
- 3-bit indicating whether a food pellet or power pill exists at a Manhattan distance of 2, 3, 4 from PacMan's location.
- 4-bit indicating whether there is food in each of its direct line of sight.
- 1-bit indicating whether PacMan is a super PacMan.

At the start of each episode, the environment reads a file that describes the location of the PacMan, the 4 ghosts, the walls, and the 2 ends of the magic tunnel, then a food pellet is placed down with probability 0.5 at every empty location on the grid. The episode resets if PacMan is caught or if it collects all the food.

If a ghost is eaten by super PacMan, the ghost will be sent to a random location in the maze that is not occupied by a wall, another ghost, or PacMan.

After each step, the reward is calculated by adding up each event that occurred:

- -1 for movement,
- -10 for running into a wall,
- 10 for eating a food pellet/power pill,
- -50 for being caught by a ghost,
- 30 for eating a ghost as super PacMan,
- 100 for eating all food pellets,

The reward ranges from to -61 to 139, encoded in 8 bits.

The theoretical optimum is too complicated to calculate.

1.5.4 Kuhn Poker

This environment implements a restricted form of the original Kuhn Poker. The environment is an opponent that always goes first.

At the beginning, both the agent and the opponent have 1 chip they can bet, and there are 2 chips in the pot. A deck of three cards (J, Q, K) is shuffled, and one card is dealt to the opponent and to the agent. The opponent decides to bet or pass.

The agent then observes the card it has been dealt with, as well as whether the opponent bet or passed.

Then, the agent must pass or bet.

The environment then plays the opponent's final pass/bet, if necessary, then calculates the reward, and immediately starts a new round, and plays the opponent's new first move. The agent observes the reward from the last game as well as the observation from the new game.

The opponent plays the Nash equilibrium strategy [Kuh50]:

> In turn one, always pass.
> If agent passes, then showdown, and there won't be a turn two.
> Else, agent bets, and
> If opponent has J, pass.
> If opponent has Q, bet with probability 1/3.
> If opponent has K, bet.

Against such an opponent, the agent could at most achieve an average reward of 1/18 per cycle.

The agent has 2 available actions, encoded in 1 bit: pass, bet.

There are 6 possible observations, encoded in 3 bits:

- Agent has J, Opponent bet.
- Agent has Q, Opponent bet.
- Agent has K, Opponent bet.
- Agent has J, Opponent passed.
- Agent has Q, Opponent passed.
- Agent has K, Opponent passed.

The reward is defined as the amount of chips the agent have by the end of the game, which could be any number from $\{0, 1, 3, 4\}$. This is encoded in 3 bits.

1.6 Encoding of environments

Each environment interacts with the agent through binary channels. The agent's actions are encoded in binary strings before being sent to the environment, and the environment encodes the observations and rewards for the agent in binary strings before sending to the agent. The number of bits in action, observation, and reward for each environment are tabulated in Table 2.

Environment	Action bits	Observation bits	Reward bits
Cheese Maze	2	4	5
Extended Tiger	2	3	8
Kuhn Poker	1	3	3
PacMan	2	16	8

Table 2: Length of binary encoding of the environments.

2 User Manual

This section is a user's manual of our implementation of MC-AIXI-CTW agent in Python 3. It describes its parameters and configurations.

2.1 Getting started

To run the scripts, a Python 3 interpreter and the standard Python 3 library are required.

The code is hosted at GitLab. Download and extract from it, and run the following command:

```
cd  /mc-aixi-ctw
python aixi.py -e coin flip
```

Alternatively, one may run a python script to run the agent with different configurations file, also the visualization of performance change is provided.

2.2 Configuration file format

A configuration file is composed of lines of key-value pairs:

```
key = value
```

To run the agent with a custom configuration file, run the following command:

```
python aixi.py -v file.conf
```

To swap between different environment, run the following command:

```
python aixi.py -v -s extended_tiger,1250,3750 conf/kuhn_poker.conf
```

Here, `-s` option indicates the use of a second environment. `extended_tiger,1250,3750` means that the second environment is `extended_tiger`, which replaces the first environment, `kuhn_poker`, at time 1250, and the first environment will be swapped back at time 3750.

For completeness, we copy from [19] descriptions of configuration options for `aixi.py`.

- **ct-depth:** maximum depth of the context tree used for prediction.
- **agent-horizon:** the number of percept/action pairs to look forward.
- **exploration:** probability of playing a random move.

- **explore-decay:** a value between 0.0 and 1.0 that defines the geometric decay of the exploration rate.
- **terminate-age:** how many agent/environment cycles before the agent needs to finish.
- **mc-simulations:** the number of MC simulations per cycle.
- **environment:** the environment the agent is to interact with.
- **learning-period:** the cycle count.
- **second:** swapping between two different environments, where the following format is used: `parameter -s <name of second enviroment,time to run second environment, time to change back to initial environment>`

2.3 Experiment Command Script

Python script `experimental_script.py` was created to run experiments under different configurations and plot the results. It requires the `tee` package.

2.3.1 Parameter options

- **experimental_result:** The folder to store the experimental result. The script will create it if there doesn't exists one. And, all information is stored as log file.
- **conf_director:** The folder contains the different configuration files. Only the file end up with ".conf" will be experimented.
- **performance_increase_graph:** Reading the specified log file, and generate a single graph about reward. The format for setting the parameter is < log file > <'title of the graph'>
- **interval_performance_increase_graph:** Reading the specified log file, and generate a single smoothed graph about the average reward. The interval can be set by using interval parameter. And, the format for setting the parameter is < log file > <'title of the graph'>
- **compare_performance_graph:** By supplying the director, it will reading a list of log file. Then, generate the graph for them. The format to giving the value is < where store the log files > <"['name of labels',]"> <'title of graph'>. Be careful, the order for the name of labels need to corresponds with the alphabeta order of log files.
- **custom_name:** The log file for different configuration files wil be stored as the name of configuration file plus the custom_name. if no custom name is supplied, the current time will be considered as custom name.
- **interval:** In order to smooth the history reward, the interval need to be supplied. The default value will be 50.
- **type of reward:** Used to indicate whether we should read average reward or reward from the log file.

2.3.2 Running experiments

Here is a example to run the experiments.

```
python experimental_script.py -c conf_director -e experimental_result
```

2.3.3 Generating graphs

Here is two example to generate the graph, and the demonstration of graph style.

```
python experimental_script.py -v experimental_result \['coin flip','coin flip compare']"
flip' -g 10
python experimental_script.py -i 'experimental_result/coin_flip.log' 'coin flip'
-g 10
```

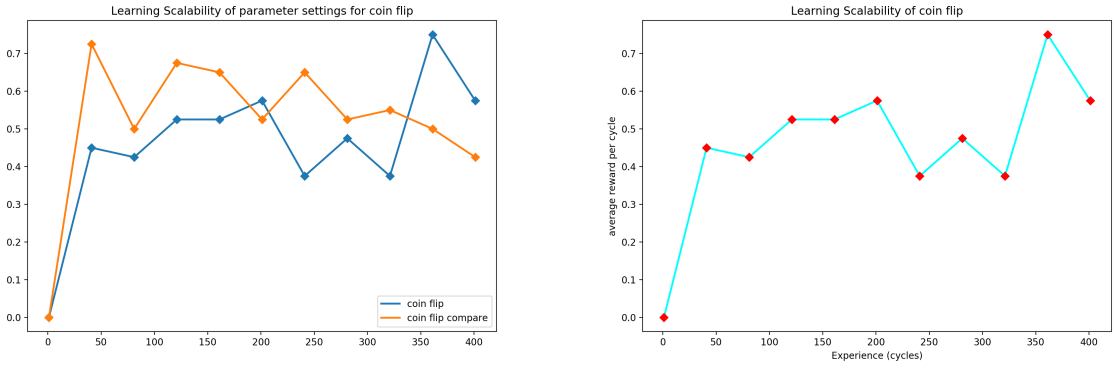


Figure 2: Sample Reward-cycle Graph

2.4 Log file format

During a run, the python scripts output log files, which are plain text files containing the results of each round of agent-environment interaction.

As an example, here is a section in a log file, recording the results of one round during a run with the agent in the coin flip environment.

```
prediction:  heads, observation:  heads, reward:  1
Agent is trying to choose the best action, which may take some time...
2, 1, 1, 1, False, 0.099500, 1, 0.500000, 0:00:00.504950, 1
cycle:  2
average reward:  0.500000
explore rate:  0.099500
```

From top to bottom, the meaning of each line are:

1. The agent is predicting heads, and observed heads, and received reward 1.
2. Filler text.
3. `cycle, observation, reward, action, explored, explore_rate, total reward, average reward, time spent in the round, model size.`

4. This is cycle 2 in the run.
5. The average reward per cycle so far is 0.5. Average reward is defined as $\frac{\text{total reward}}{\text{cycle number}}$.
6. The exploration rate of this cycle is 0.0995. For a definition of the exploration rate, see Section 1.4.

All of the environments in our implementation tries to follow the same format as much as possible. For the complex game, such as pacman. The string representation of map is also provided.

Figure 3: An example terminal printout from an interaction cycle of the agent. Here, a game of PacMan is in progress. P stands for pacman, S stands for power pill,the other letters stand for monsters, * stands for food pellets, and % stands for walls.

3 MC-AIXI-CTW Implementation in Python 3

This section explains the design decisions in our implementation of MC-AIXI-CTW agent and its environments.

Our implementation of MC-AIXI-CTW is split into four major parts;

1. Agent,
2. Monte Carlo Search Tree (MCTS),
3. Environment,
4. Context Tree Weighting (CTW),

with corresponding (parent) classes `MC_AIXI_CTW_Agent` , `MonteCarloSearchNode` , `Environment`, and `CTWContextTree` .

In general, the class `MC_AIXI_CTW_Agent` serves as an interface for the other classes. An `MC_AIXI_CTW_Agent` o stores objects of the other three classes, and calls their methods. For each time step in the lifecycle of the agent object, the agent performs one round of a loop, in which the agent plans for the next action, sends the action to the environment, and receives percepts from the environment. This loop is defined in `aixi.py`. A flowchart is in Figure 4.

Each run of `aixi.py` would call up the AIXI agent to interact with an environment with certain specified configurations. Configurations can be loaded from configuration files, as detailed in Section 2.

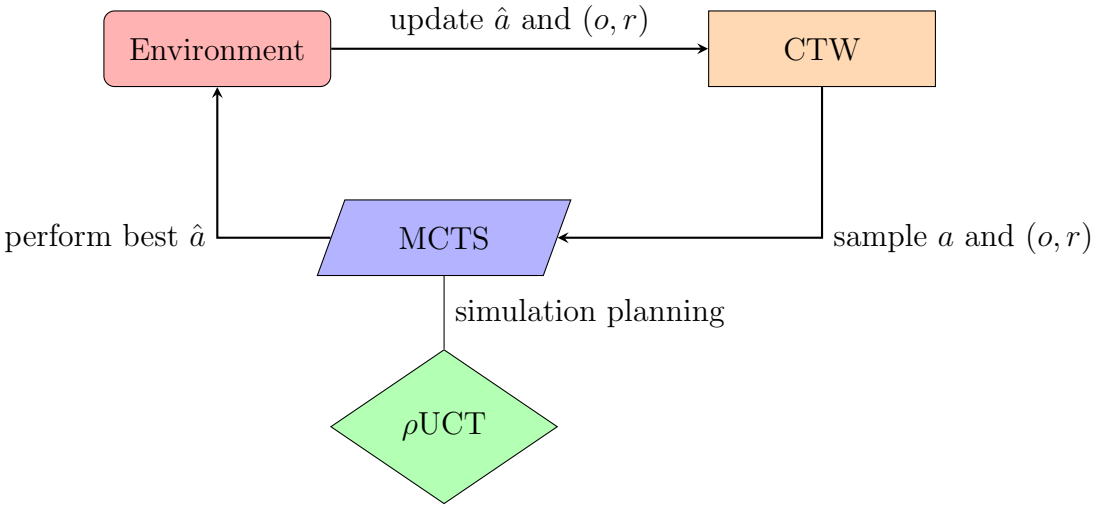


Figure 4: Flowchart of the Python implementation of MC-AIXI-CTW agent and its environment.

3.1 Environments

Each environment is a subclass of `Environment`. Each stores:

- the latest action, observation, and reward;
- a `list` of valid actions, observations, and rewards;
- maximum and minimum action observation, and reward values;
- maximum length of binary encodings of them.

The observations, rewards, and actions are all defined as `enum` objects, which are essentially named positive integers. In detail, `enum` objects are defined with `util.enum()` function from `util.py` module is then paired with `int` values.

Percepts are returned by `perform_action()`, which, unless otherwise noted, is a `if-else` structure enumerating all outcomes of actions and returning the appropriate rewards and observations.

3.1.1 Kuhn Poker

Since each game only lasts one round long, in each round, the environment initializes a new game by calling `restart()`. It uses `random.choice()`, to randomly deal one card each to the agent and the opponent (internal to the environment).

3.1.2 Cheese Maze

The initializer of the class `CheeseMaze` reads from the `cheese_maze.txt` file, which encodes, in plain text, the map of initial positions of the walls and the cheese.

The map is parsed and represented in an internal map object. A `dictionary` is simultaneously generated from the map, which stores the observed alias of each room in the true map.

3.1.3 Extended Tiger

Class `ExtendedTiger`, has a class variable `default_probability = 0.85`, which represents the probability that the agent can correctly hear the location of the tiger in any attempt.

3.1.4 Partially Observable Pacman (POCMAN)

The initializer of the class `PacMan` reads from the `pacMan.txt` file, which encodes in plain text, the map of initial positions of Pacman, the ghosts, the walls, the power pills, and the two magic tunnel entrances.

An `if-else` control flow is used to detect the location of each character to determine location of objects, and calculates distance between them using `find_Positions()`. This is necessary since when the distance between super Pacman and the ghosts is less than 5 characters, the ghosts moves once every time Pacman moves twice.

3.2 Context Tree Weighting

In `ctw_context_tree.py`, Context Tree Weighting (CTW) is implemented as a recursively-defined tree data structure `CTWContextTreeNode` , with root of the tree stored in a wrapper class `CTWContextTree` that may be interpreted as a subtree along with maximum tree depth, and its size. Each `CTWContextTreeNode` stores

- the log probability of the subtree rooted at it;
- symbol count;
- the log KT-estimator;
- its children.

The weighted probability calculation method from MC-AIXI-CTW to estimate the probability of having the tree is adopted [Ven+09]. Most mathematical formulation are implemented in class `CTWContextTreeNode`. Class `CTWContextTree` contains high-level interfaces for the agent to use the CTW with.

Central to the CTW implementation is the `update()` function in `CTWContextTree` class, which updates

1. the KT-estimator (for leaf nodes),
2. log probabilities (for non-leaf nodes),
3. history,
4. find context of the CTW with each new bit.

As each path in the `CTWContextTree` represents a unique context, which is implemented as a list of `CTWContextTreeNode` from the tree root to the leaf, we update the context by reading the context from root to node with the most recent context of a node being 1 if it is the left child of its parent, otherwise 0, and context of the root node is ϵ (empty string). Additionally, count of each symbol is also kept in each node in a Python `dictionary`.

As the length of context is equal to the the depth of tree, a node will not be updated if its binary-encoded history length is less than the depth of the subtree. Similar to `update()`, the function `revert()`, which reverts the context of relevant nodes in the tree, is also implemented from the leaves to root of the tree. Furthermore, for memory efficiency, a code to trim the children nodes in the tree with zero symbol count is implemented within `revert()`, for more efficient memory use, as memory issues occurred in early experiments.

Another key function is `predict()`, which returns the probability of a binary sequence, given all the past history received by the CTW.

The construction of CTW algorithm assumes that the CTW object is initialized with a history h of length of at least the tree depth D, but this is not necessarily true when our agent begins its operation. To deal with this boundary case, uniformly generated random binary sequence is added to h when `predict()` is called, but $l(h) < D$.

In sampling a random sequence of symbols from the tree,
`generate_random_symbols_and_update()` function generates a random binary sequence from the mixture of PST implicitly represented in the CTW object. Random binary sequences are generated by comparing a random values from `random.random()` and `predict()` functions which reflects the conditional probability $P(b_{1:d}|h)$ of a binary string $b_1 \dots b_d$ given history h. However in this implementation, the sample was taken one bit at a time alternating with updating the conditional distribution because $P(b_{1:d}|h) = p(b_1|h)p(b_2|hb_1)\dots p(b_d|hb_{1:d-1})$ with the intention to save computational cost of considering 2^d possible strings in directly sampling $P(b_{1:d}|h)$.

To save memory, We rewrote the history saving function implemented in the original library, so that only part of history is kept. This is allowed, since the agent only needs the last D bits of history for the CTW algorithm, where D is the depth of the CTW tree. These D bits are used as the context for the new coming in bits, and any previous bits are not needed.

Note that, for the purpose of MCTS, where storing more than D bits of history is needed for D depth of CTW, we provide `estimate_size`, an additional argument for `CTWContextTree` , which indicates how many additional bits we need to store in history for reverting the CTW tree after a round of MCTS sampling.

To lastly note an important function in `CTWContextTree` , `generate_random_symbols_and_update()` is called in the playout phase of the MCTS to simulate the environment.

3.3 Monte Carlo Tree Search

The file `monte_carlo_search_tree.py` contains methods and classes to implement ρUCT, the planning component of the MC-AIXI-CTW agent.

Its main part is the class `MonteCarloSearchNode` , which implements a recursively-defined data structure representing a Monte Carlo Search Tree (MCST), consisting of two kinds of nodes: decision and chance, defined as `enum` objects, `decision_node` and `chance_node` . A `decision_node` represents a state of the environment where the agent is about to perform an action, and a `chance_node` node represents a state right after the agent performed an action, and the environment is ready to respond with observation and reward.

Thus, each `decision_node` contains a dictionary with key-value pairs

$$\{\text{action} : \text{action node}\},$$

each key `action` maps to the resulting state after the action is undertaken by the agent. Similarly

for each `chance_node` ,

{observation : decision node}.

As only the agent, not the environment, uses the MCST to make decisions, all constructed MCSTs throughout lifetime of the agent are rooted at a `decision_node` .

For convenience in implementing the ρUCT algorithm, each `MonteCarloSearchNode` node stores as attributes:

1. the remaining agent horizon;
2. the sum of future rewards from all sampled futures;
3. the number of times the node has been visited.

During the planning phase, the agent performs a fixed number of samplings of the search tree. This is a configurable option.

The tree traversal is done in each node within the `sample()` function, which contains a `if else` control flow statement indicating whether the agent should perform a uniform random rollout with `playout()` function, traverse to a `decision_node` by sampling a percept, select an action with UCB method, or stop the traversal.

The agent is initialized with an initial percept before taking up any action in a `decision_node` . Creation of child nodes is done within the `sample()` function in each MCST node where an action child will be created from a percept node through a uniform random rollout policy if the percept node is visited for the first time, and otherwise selected using UCB policy with exploration constant $\sqrt{2}$ as the original implementation [Ven+09] does.

After choosing an action and update the CTW, `sample()` will then sample a percept from CTW that will be created as a new child node from the recently performed action node if it has not been observed before from that action node, or will be visited otherwise.

These two actions repeat until a percept node with no child is reached, where its expected reward will be approximated through the uniform random rollout policy that samples a playthrough with length equalling the remaining agent horizon. Finishing the sampling action, the recursive structure for MCTS came in conveniently as reward received in a percept node is returned to its parent node, which then update its average reward, and then return that reward to its parent node for the rewards to be summed up. This recursive action then will result in an update to the average reward of each action at the root node accordingly to rewards from trajectories of sequence of actions-percepts. Both action and percept nodes take count of how many times they are visited, as this is essential for calculating UCB scores.

As `sample()` are typically called multiple times according to the specification of how many samples the agent will take during planning time, interface function `sample_iterations()` is used to iterate sampling accordingly to agent's specification while also performing reset on agent's state appropriately as we will discuss in Section 3.4. This function is further interfaced by function `mcts_planning()` that will be called by the Agent to do a ρUCT planning, where an action from corresponding children nodes of the root of MCST with largest ρUCT reward will be chosen as the next action that the agent will perform in the true environment.

3.4 MC-AIXI-CTW Agent

The Python class `MC_AIXI_CTW_Agent` implements the MC-AIXI-CTW agent. An instance of this class contains:

- an `Environment` object, the environment that the agent is interacting with.
- a `CTWContextTree` object, responsible for learning.
- `age`, the number of cycles the agent has been alive.
- `total reward`, the total reward the agent has received so far.
- `options`, the configuration options for the run. Including: the depth of context tree, number of MC sample taken in a time step, the agent's horizon, and number of time steps it can learn.
- `learning_period`, the number of cycles that the agent would learn from. If it is set to 0 or below, it would learn forever. During the learning period, the agent explores, and would update its CTW upon receiving a new action-percept pair. After the learning period is finished, exploration stops, and the agent never updates its CTW again, and would continue planning using whatever its previously learned CTW.
- `last_update`, the last update that the agent has received. Can either be `action_update` or `percept_update`.

In cycle n, after the agent receives the previous percept $o_{n-1}r_{n-1}$ from the environment, if it is still in the learning period, it learns by updating its CTW tree with the new percept. Otherwise, it merely stores the new percept in the history attribute of its CTW tree, without updating the CTW tree. This is the learning phase of the cycle.

In the planning phase of the cycle, the agent runs the ρUCT algorithm by initializing an `MonteCarloSearchNode` object

As same instance of Agent with its corresponding CTW is used for MC sampling simulation, it is convenient to store information of the Agent's and CTW's state at a point of time as an object that we defined as `MC_AIXI_CTW_Undo` class. This object is created particularly before every MCTS sampling is done for planning to save the current state h. As the agent receives multiple percepts and did multiple actions in the planning process, these percepts and actions are removed by restoring the agent's previous state h, which at this point, the UCB-optimal action is identified. These are done simply by storing a `MC_AIXI_CTW_Undo` object within `MC_AIXI_CTW_Agent` , which attributes replace the corresponding attributes of agent's state in `MC_AIXI_CTW_Agent` . As saving is done before every planning session, which consisted of several MCTS sampling, this functionality is called in `sample_iterations()` function in `MonteCarloSearchNode` object as the agent state reverted to the saved state at the end of every single sampling.

Being the interface for all three components, `MC_AIXI_CTW_Agent` serves as an intermediary between all three components as it includes functions to sample and update actions and percepts from both CTW and true Environment. As the process of planning and sampling discussed previously, the functions to sample and update CTW are repeatedly called during ρUCT planning from `sample()` function in `MonteCarloSearchNode` , indicating that `MC_AIXI_CTW_Agent` serves as an information intermediary between `MonteCarloSearchNode` and `CTWContextTree` . While this also happens to `Environment` object, it happen less frequently only after a ρUCT is done and the best action is chosen, where the information of the best action is communicated from `MonteCarloSearchNode` to `Environment` through `MC_AIXI_CTW_Agent` .

In addition to the three objects utilized by the Agent that are explained in detail above, a technical yet necessary functionality that is also utilized by the agent is to convert percept and

action from and to binary strings in order to sample and update the CTW. This is done through calling functions defined in the python script `util.py` where functions `encode()` and `decode()` are used to do this translation by converting integer to binary with Python standard library function `bin()` and padding it with zeros up to a specified length to prevent bias of notion of simplicity for integers that are encoded with fewer binary digits, and using built-in Python standard library function `int()` to convert a binary string back to integer.

4 Experiment Results

This section discusses the setup, choice of configurations, and results of experiments on MC-AIXI-CTW in the environments described in Section 1.5. All experiments were ran on Intel 2.80Ghz E5 Xeon 2690 v3 processors.

4.1 Setup and Configurations

Command

```
python3 aixi.py <configuration file name> | tee -a <log file name>
```

outputs a log file with results of each cycle for each environment.

The following is an example of output from the log file for a single cycle

```
Agent is trying to choose the best action, which may take some time...
5850, 5, 3, 0, False, 0.600450, 11395, 1.947863, 0:00:07.024427, 1
cycle:  5850
average reward:  1.947863
```

To study the effect of different configurations, we ran the agent on the same environments using different configuration options. Table 3 tabulates expected effects on performance of agent after changing configuration options.

Configuration options	Change	Expected effect on agent performance
agent-horizon	Larger	Large computational overhead on MCTS for smaller environment, but may give more rapid convergence in early cycles.
ct-depth	Larger	Large computational overhead on CTW for smaller environment, but may give more rapid convergence in early cycles.
mc-simulation	Larger	More rapid convergence to optimum reward after a sufficient learning.
learning-period	Larger	More rapid convergence to optimum reward after a sufficient learning.

Table 3: Expected effect of configurations on agent performance.

In our experiments, for each environment, we defined a "default" configuration, and from that we derived several variant configurations, each changing one configuration value, to check if they indeed had the expected effects.

Tables given below for each respective environments lists the default configuration in the middle column, and variant values in the last column.

4.2 Experimental Results

4.2.1 Kuhn Poker

We ran 5 experiment with Kuhn Poker environment, one default and 4 variants, shown in Table 4.

Parameters	Values	Variant
Exploration	0.99	None
Exploration-decay	0.9999	None
Agent-horizon	2	4
mc-simulation	500	1000
ct-depth	42	21
Learning-period	5000	8000
terminate-age	10000	None

Table 4: Kuhn Poker experiment configurations

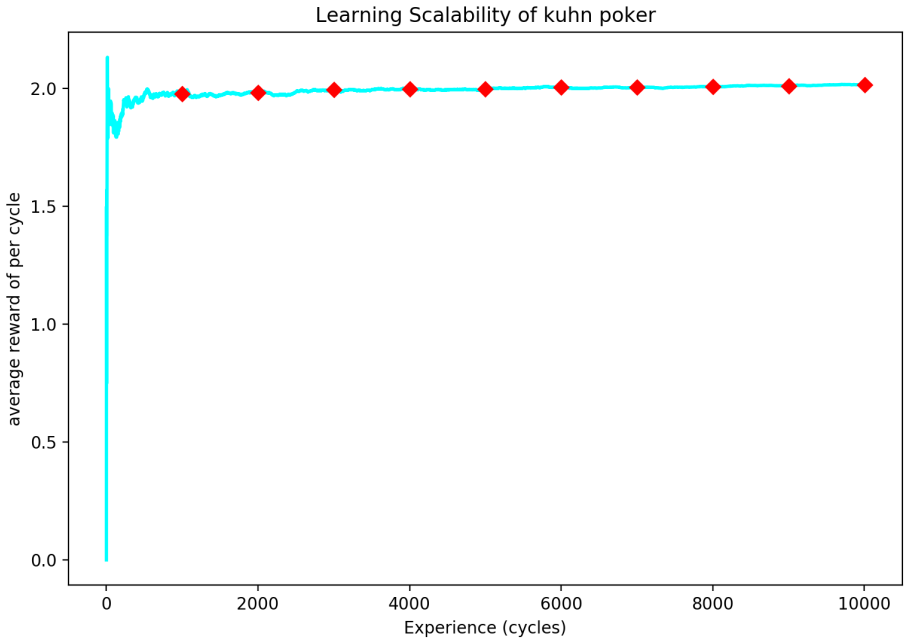


Figure 5: Learning to play Kuhn Poker under default configurations. The x-axis is the number of rounds, while the y-axis is the average reward achieved.

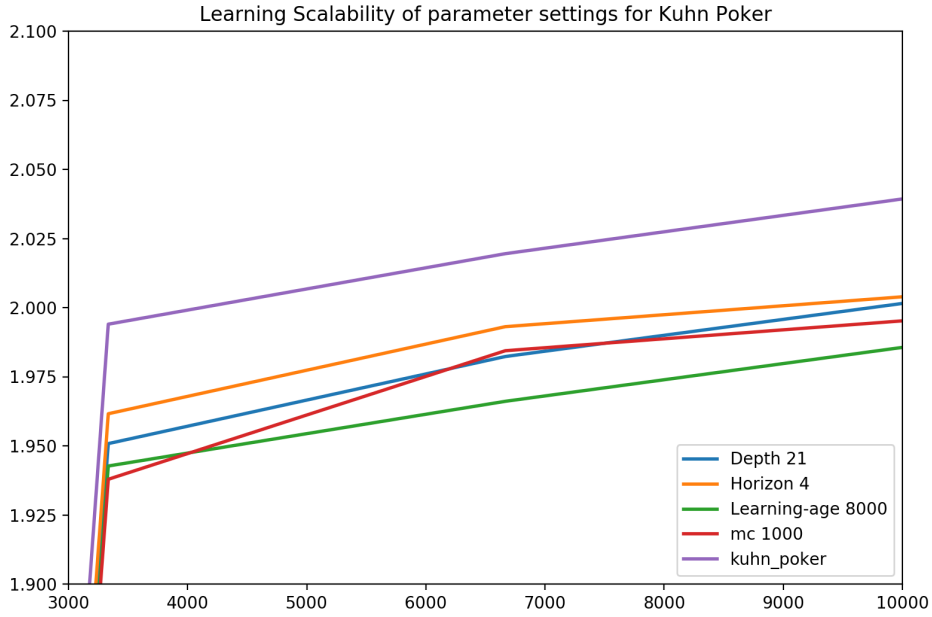


Figure 6: Comparison of rates of learning to play Kuhn Poker under different configurations. The x-axis is the number of rounds, while the y-axis is the average reward achieved. Under all configurations, The agent approaches the theoretical optimal of 2.055. The default configuration turned out to allow for fastest convergence for the agent.

Since Kuhn Poker environment is very small, with $|\mathcal{A}| = 2$, $|\mathcal{O}| = 6$, and $|\mathcal{R}| = 4$, we set the value of `horizon, mc-simulation, ct-depth` smaller relative to configuration for other environments to prevent unnecessary overhead in computational cost for the reasons will be discussed below.

1. `horizon = 2`: Since each game of Kuhn Poker lasts for one round where the agent take a single action, given enough time for CTW to converge sufficiently to Kuhn Poker environment, horizon of 2 will be sufficient for the agent to try all possible combinations of successive actions $a_1 a_2 \in \mathcal{A}^2$ in MCTS.

2. `mc-simulation = 500`: This was set to a small value to save computational cost. The variant of `mc-simulation = 1000` was tested to see if 500 is sufficient.

3. `ct-depth = 42`: This number was set as large as possible within available computational resources. A variant of `ct-depth = 21` was tested to see if a shallower context-tree is sufficient in modeling Kuhn Poker environment.

Figure 6 shows that all configurations allowed the agent to converge to the best average reward of $\frac{37}{18} \approx 2.0555$, despite different convergence speed. It can be inferred that the default configuration yields fastest convergence.

4.2.2 Extended Tiger

Parameters	Values	Variant
exploration	0.99	None
exploration-decay	0.99999	None
agent-horizon	4	8
mc-simulation	500	100
ct-depth	96	42
learning-period	5000	None
terminate-age	10000	None

Table 5: Extended Tiger experiment configurations

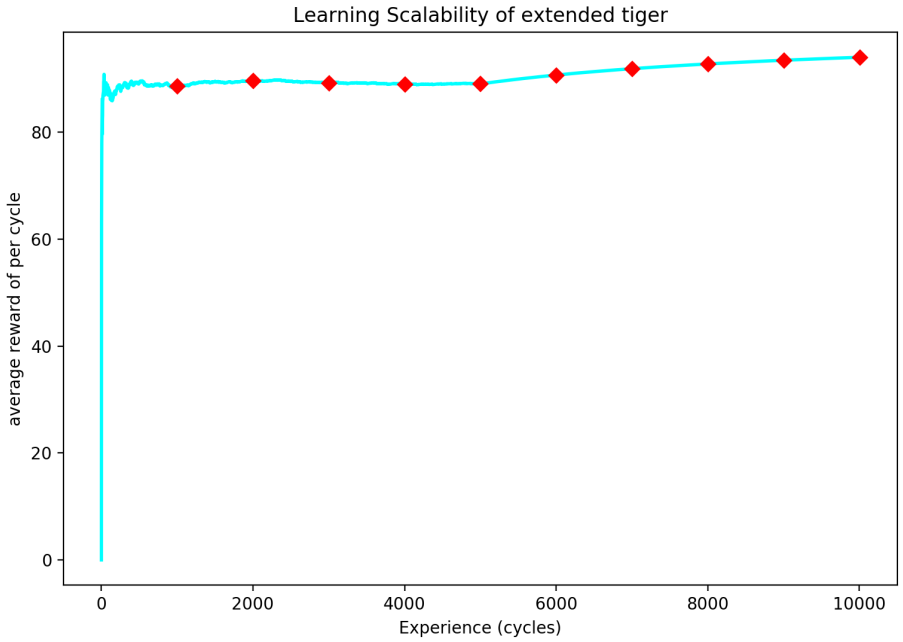


Figure 7: Extended Tiger default configuration

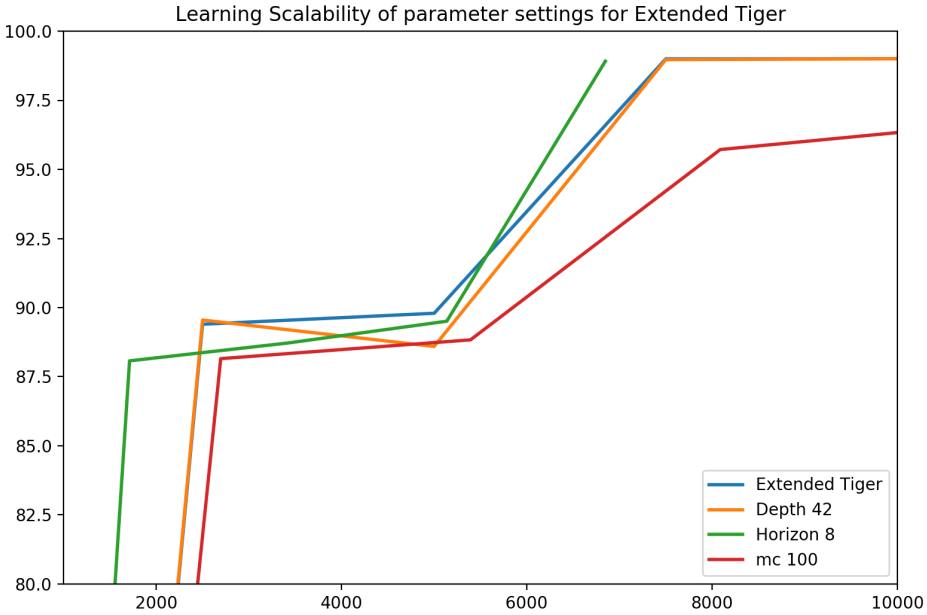


Figure 8: Comparison between average reward under different configuration settings of Extended Tiger. The x-axis is the number of rounds, while the y-axis is the average reward achieved. While appears to be failing to achieve the optimal expected reward of ≈ 106, our experiment using default configuration appears to roughly agree with the original experiment in [Ven+09].

As performance of our best model for Extended Tiger is shown in Figure 8, we now will discuss implication of changing value of each parameters:

1. `horizon = 4`: As $|\mathcal{A}| = 4$, therefore, when the agent have learnt the environment for the CTW to model the environment sufficiently close, any value of `horizon` larger than 4 will cost redundant computational cost as `horizon = 4` is sufficient for the agent to be able to perform an action with optimal reward during MCTS. A variant of horizon $= 2$ (which should be sufficient to allow the agent to stand up and open the door), was tested to see if the agent's performance would degrade notably.

2. `mc-simulation = 500`: As a smaller configuration parameter value `mc-simulation = 100` was tested to see if 500 can be reduced to save computational cost, no evidence of same or better convergence was found for smaller value of 100 as it can be inferred from Figure 8.

3. `ct-depth = 96`: This number was set as large as possible within available computational resources. A variant of `ct-depth = 42` was tested to see if a shallower context-tree is a sufficiently large mixture of classes in model class $\mathcal{M}$ which includes the Extended Tiger environment.

As shown in Figure 7, for all configuration, the agent achieved average reward < 90 during the learning phase as it explores the Extended Tiger environment and at times opening the door with the Tiger in it as that is the only action with reward less than 90.

Observing the best configuration (which is the default configuration), as after learning period ended (cycle > 5000) the agent kept getting an average reward of 99, we can infer that the agent needs a longer learning period for the CTW to model the environment close enough to the true environment.

4.2.3 Cheese Maze

Parameters	Values	Variant
exploration	0.999	None
exploration-decay	0.9999	None
agent-horizon	8	4
mc-simulation	500	100
ct-depth	96	42
learning-period	5000	None
terminate-age	10000	None

Table 6: Cheese Maze experiment configurations.

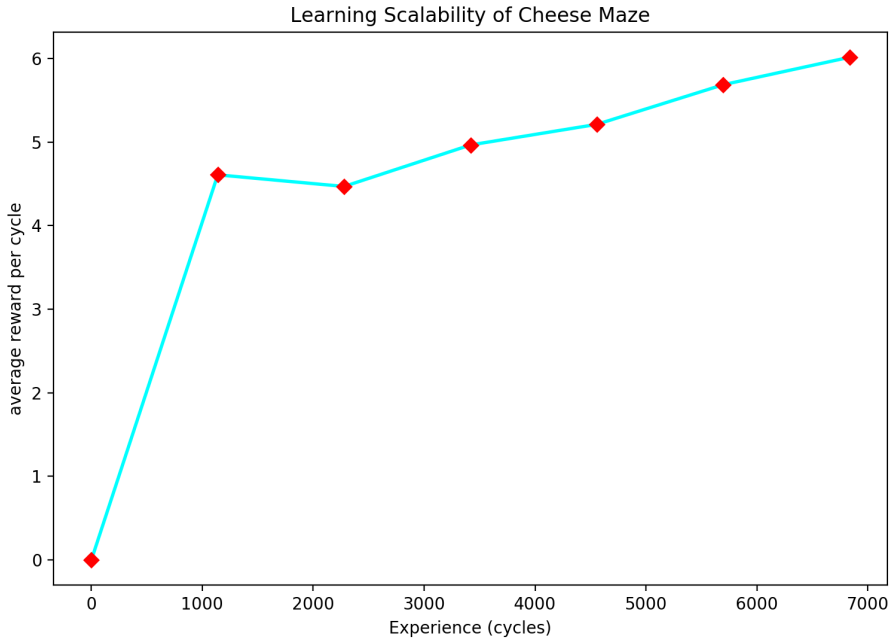


Figure 9: Learning to play Cheese Maze under default configurations. The x-axis is the number of rounds, while the y-axis is the average reward achieved.

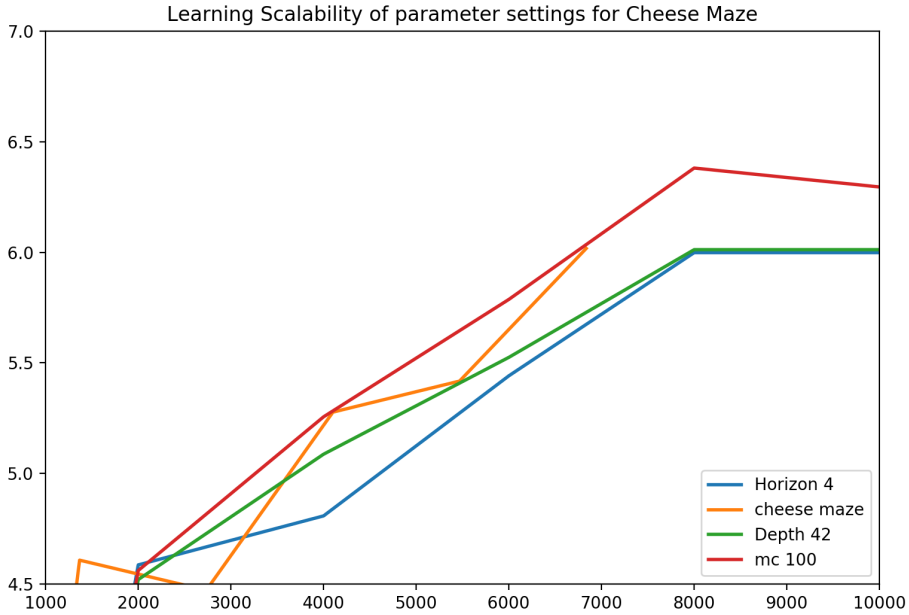


Figure 10: Comparison between average reward under different configuration settings of Cheese Maze. The x-axis is the number of rounds, while the y-axis is the average reward achieved.

Explanation of configuration options:

1. `horizon = 4`: the agent would get an optimal reward by performing three actions: right → down → down. Thus, looking forward 4 steps would be a reasonable setting. A Variant of horizon = 8 to see if agent will be performing better by looking forward more steps.

2. `mc-simulation = 500`: This was set to a small value to save computational cost. The variant of `mc-simulation = 100` was tested to see if 500 is necessary.

3. `ct-depth = 96`: This number was set as large as possible within available computational resources. A variant of `ct-depth = 42` was tested to see if a shallower context-tree is sufficient in modeling Cheese Maze environment.

We did not compute the optimal reward for the Cheese Maze, so we cannot say how much convergence occurred. What we can say is that, since the average reward was rising, the agent was learning.

Initially, the agent acts randomly. Since at each location, there are two valid actions and two invalid actions, the expected average reward by taking random actions would be slightly above $\frac{9+0}{2} = 4.5$. As shown in Figure 9, the average reward gradually increased as agent learned to avoid invalid actions. However, it never quite learned to get the cheese. We believe that if the model were to be trained for longer, the agent would be memorize the optimal strategy: right → down → down.

4.2.4 PacMan

Parameters	Values	Variant
exploration	0.9999	None
exploration-decay	0.99999	None
agent-horizon	4	2
mc-simulation	500	100
ct-depth	96	42
learning-period	5000	8000
terminate-age	10000	None

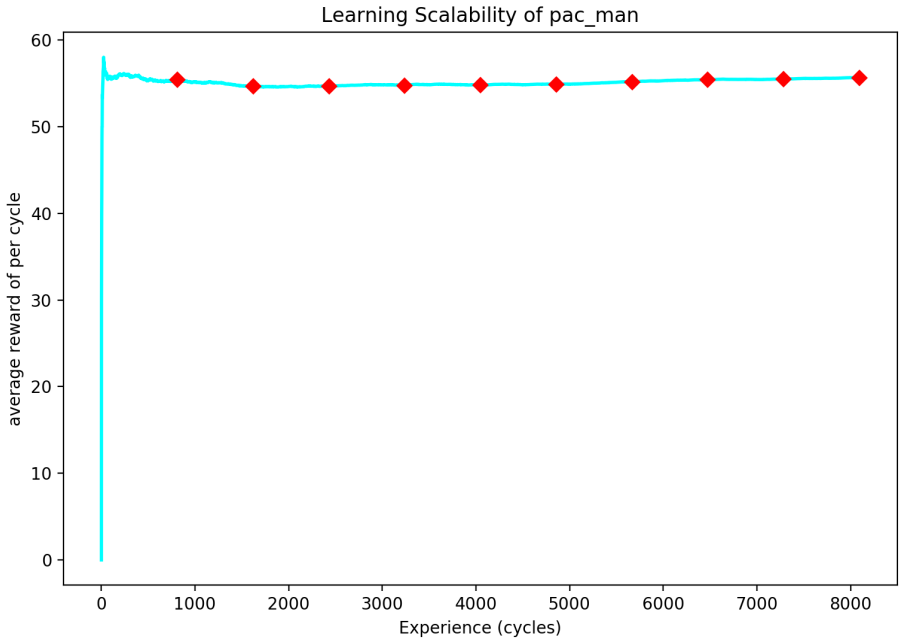


Figure 11: Learning to play PacMan under default configurations. The x-axis is the number of rounds, while the y-axis is the average reward achieved.

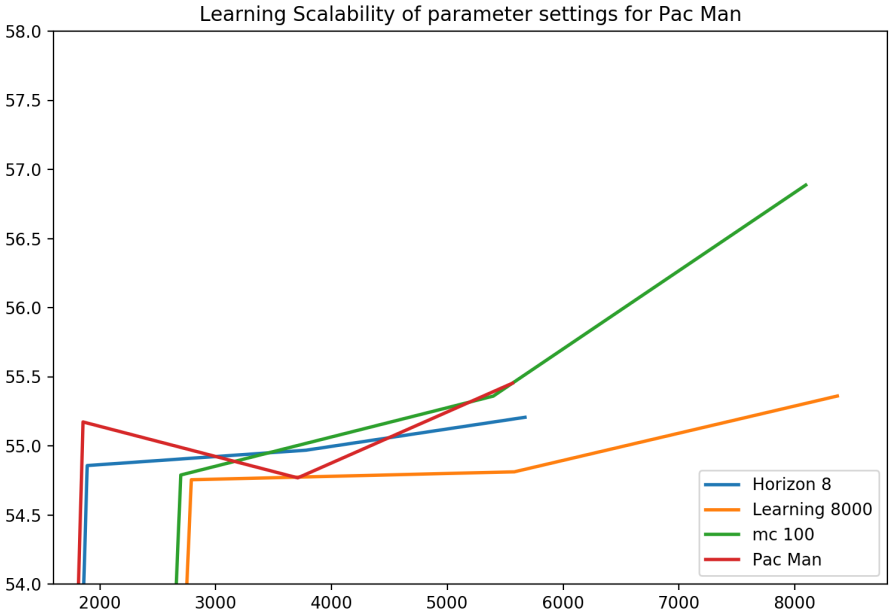


Figure 12: Comparison of different settings

The performance of our best model for PacMan is shown in Figure 11.

Experiment Setup : run command `python3 aixi.py pacman_conf/pac_man.conf | tee -a pac_man.log` will output a log file with results of each cycle of PacMan environment.

Explanation of configuration options:

1. agent-horizon: the search horizon affect the capabilities of agent. For partially observed PacMan, the larger horizon would improve the performance of the agent with the expense of significantly increased computation time.

2. mc-simulation: Due to the time issue, we can not expect a large simulation to be used on a complex environment such as PacMan. Thus we use 500 and an variant of 100 to compare the result

3. ct-depth = 96: this number was set as large as possible while still comfortably within our computational resources. A variant of ct-depth = 42 was tested to see if a shallower context-tree is good enough to model this environment.

4. learning-period = 5000: represent the cycles that the agent explores. Because evaluation phase is just more time consuming. We use an variant of learning-period = 8000 to see how that affect our result.

1. `horizon = 4`: the agent would get an optimal reward by performing three actions: right → down → down. Thus, looking forward 4 steps would be a reasonable setting. A Variant of horizon = 8 to see if agent will be performing better by looking forward more steps.

2. `mc-simulation = 500`: This was set to a small value to save computational cost. The variant of `mc-simulation = 100` was tested to see if 500 is necessary.

3. `ct-depth = 96`: This number was set as large as possible within available computational resources. A variant of `ct-depth = 42` was tested to see if a shallower context-tree is sufficient in modeling Kuhn Poker environment.

4.2.5 Environment Swap - Kuhn Poker, Extended Tiger, and Cheese Maze

In the environment swap experiment, we started by training the agent on Kuhn Poker environment for 1250 cycles, then switch to Extended Tiger or Cheese Maze and learn for for 2500 cycles before then switching back to Kuhn Poker for 1250 cycles. Both experiment results where a switch to Extended Tiger and to Cheese Maze were plotted and discussed below using the default configuration for Kuhn Poker in Table 4.

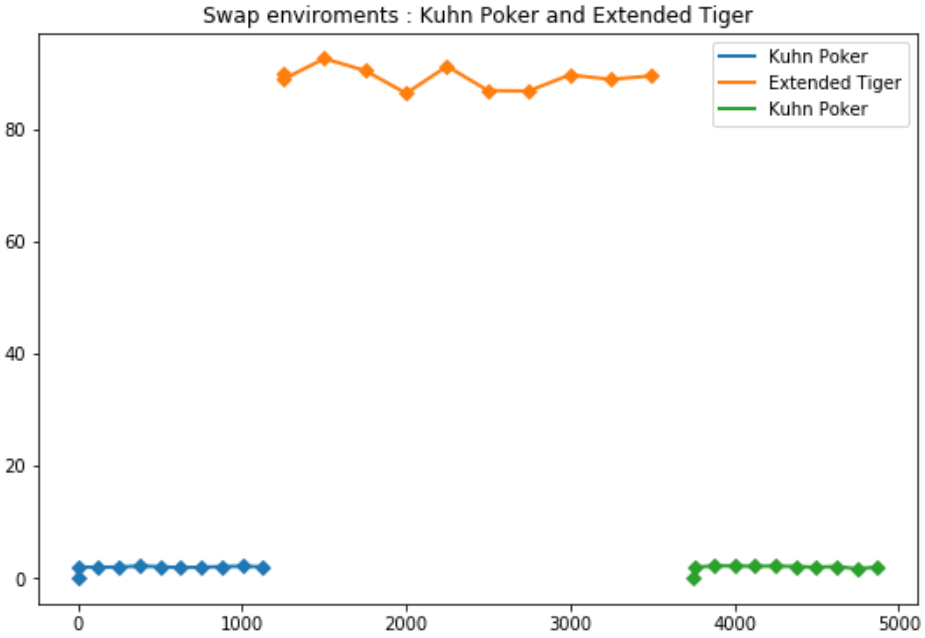


Figure 13: Learning to play Kuhn Poker, then Extended Tiger, then Kuhn Poker again. The x-axis is the number of rounds, while the y-axis is the average reward achieved.

Figures 13, 14 give the result of performance change by swapping between Kuhn Poker and Extended Tiger environments, and between Kuhn Poker and Cheese Maze Environment.

By comparison with the expected reward in the experiment with Kuhn Poker environment in 4.2.1, we can infer that after a swap to either Cheese Maze or Extended Tiger, the agent can roughly maintain optimal average reward in the case after a switch to Cheese Maze, and regain optimal average reward in the case after a swap to Extended Tiger. Recalling the CTW, which can be regarded as a mixture of probability distributions that resembles the environment, we then are able to infer that because the agent is still biased towards the learning gained before the first swap was done in the second environment, then the CTW still preserve some information from learning in the first environment. Thus, this preserved information from the first environment is again being recalled.

After the first environment swap, it also appears that the average reward was not influenced significantly by a previous learning in a different environment. This can be seen from 15c and 15b

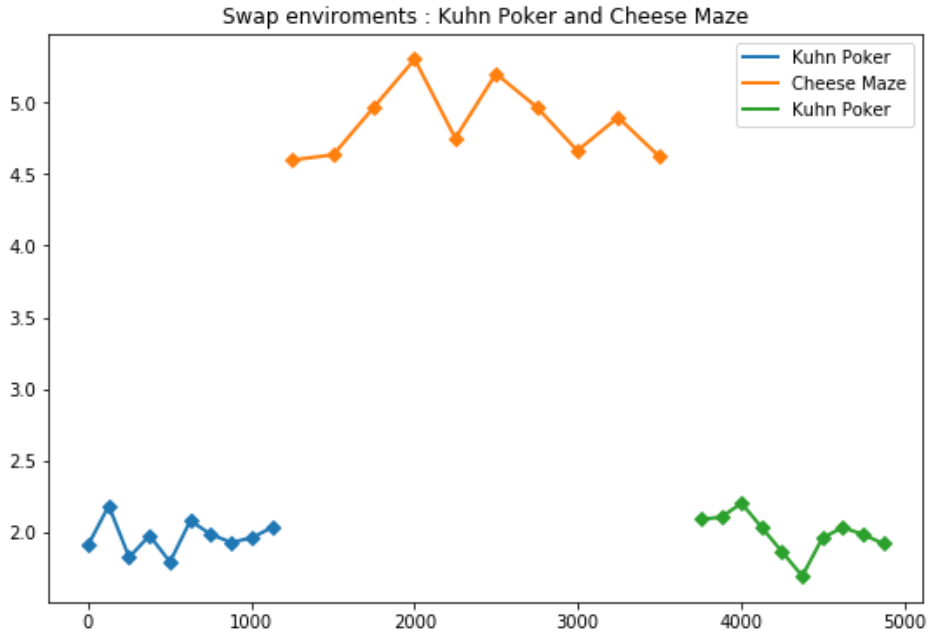
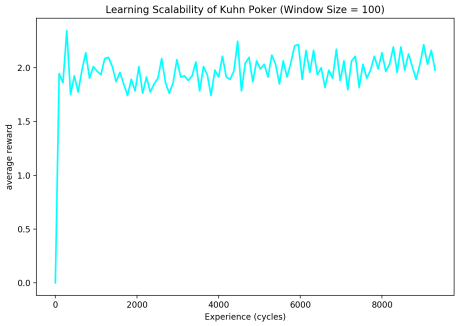


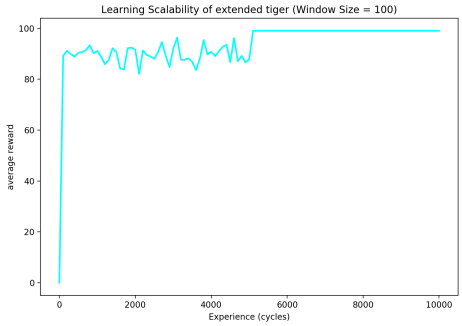
Figure 14: Learning to play Kuhn Poker, then Cheese Maze, then Kuhn Poker again. The x-axis is the number of rounds, while the y-axis is the average reward achieved.

as the average reward during the learning phase for the agent in the environment swap experiment are roughly the same compared to experiments with no environment swap.

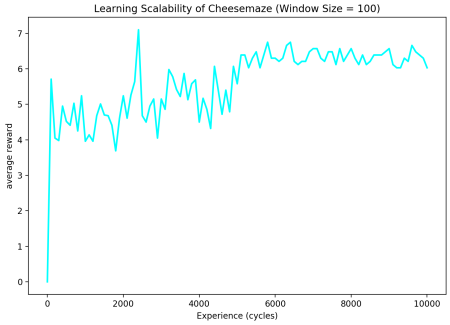
4.3 Summary of results



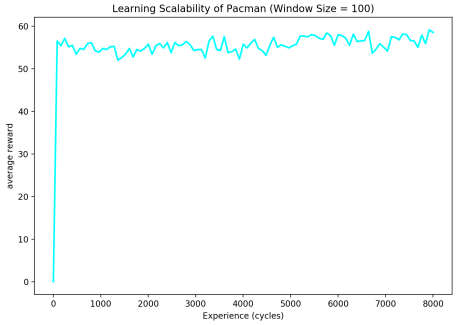
(a) Kuhn Poker



(b) Extended Tiger



(c) Cheese Maze



(d) PacMan

Figures (a) - (d) present the average reward with window size = 100 of the experiment results using the configuration that retains the highest average reward.

References

[19] *COMP4620/COMP8620 Advanced topics in AI Foundations of AI, Practical Assignment 2*. Australian National University, 2019.

[Aue02] Peter Auer. "Using confidence bounds for exploitation-exploration trade-offs". In: *Journal of Machine Learning Research* 3.Nov (2002), pp. 397–422.

[Cas] Anthony R. Cassandra. *pomdp-solve: POMDP Solver Software*. https://www.pomdp.org/code/index.html/.

[Hut05] Marcus Hutter. *Universal artificial intelligence: sequential decisions based on algorithmic probability*. Springer, 2005.

[KLC98] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial intelligence* 101.1-2 (1998), pp. 99–134.

[Kuh50] Harold W Kuhn. "A simplified two-person poker". In: *Contributions to the Theory of Games* 1 (1950), pp. 97–103.

[Ven+09] Joel Veness et al. “A Monte Carlo AIXI Approximation”. In: (Sept. 3, 2009). arXiv: 0909.0801 [cs, math].

[WST95] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens. “The Context-Tree Weighting Method: Basic Properties”. In: *IEEE Transactions on Information Theory* 41.3 (May 1995), pp. 653–664. ISSN: 0018-9448. DOI: 10/dcfdg9.