COMP4620 Assignment 2

Joseph Meltzer, Xi Du, Tianyu Wang, Xavier O'Rourke

October 2018

1 Agent Implementation

1.1 Context Tree Weighting

The code which implements the Context Tree Weighting algorithm may be found in the `predict.cpp` and `predict.hpp` files. The implementation is commensurate with the provided interfaces. The implementation uses a tree data-structure composed of several `CTNode` objects. Each `CTNode` encapsulates data about the number of 0s and 1s we have seen in that node's context, the value of the log of the KT estimator at that node, as well as the log of the weighted probability at that node (referred to as P_w^n in [1]), as well as pointers to that node's children.

The most difficult part of this module's implementation was the `ContextTree::Update` and `ContextTree::Revert` methods. When we want to update our tree with a new symbol, this means updating the symbol counts, KT estimators and weighted probabilities of the node corresponding to this context, as well as all ancestors of that node. This is achieved by first following a path down from the root node of the tree to the bottom (checking the previous symbol in the history at every step to decide which branch to traverse) while updating the symbol counts. Once we reach a leaf node at maximum depth, we then retrace our steps moving back up the tree while updating the probability estimates.

When we update the weighted probability estimates we use equation (27) in [1], which defines the weighted estimates as follows

$$P_w^n := \left\{ \begin{array}{ll} Pr_{kt}(h_{T,n}), & \text{if } n \text{ is a leaf node.} \\ \frac{1}{2}Pr_{kt}(h_{T,n}) + \frac{1}{2}P_w^{n0} \times P_w^{n1} & \text{otherwise} \end{array} \right\}$$

In our original implementation we calculated this estimate by naively evaluating the expression

$$\log P_w^n = \log\Big(0.5 \times \exp x + 0.5 \times \exp(y \times z))\Big)$$

Where $x = \log(Pr_{kt}(h_{T,n}))$, $y = \log P_w^{n0}$, and $z = \log P_w^{n1}$.

This implementation resulted in erratic agent behaviour when the size of the agent's history grew large. The problem here is that when the history is long, our probability estimates become extremely close to zero, meaning we take the log of a very small number, which can lead to underflow. To correct this, we borrowed a trick from the original implementation and calculated our log weighted probability estimates according to the equivalent formula:

$$\log P_w^n = \log 0.5 + a + \log(1 + e^{b-a})$$

Where

$$a = \max\{\log Pr_{kt}(h_{T,n}), \log(P_w^{n0} \times P_w^{n1})\}$$
$$b = \min\{\log Pr_{kt}(h_{T,n}), \log(P_w^{n0} \times P_w^{n1})\}$$

Our CTW algorithm makes predictions about the next bit in our history h using the law of conditional probability:

$$\mathbb{P}(0|h) = \frac{\mathbb{P}(h0)}{\mathbb{P}(h)} \qquad \mathbb{P}(1|h) = 1 - \frac{\mathbb{P}(h0)}{\mathbb{P}(h)}$$

This means, when calling `ContexTree::genRandomSymbolsAndUpdate()` or `ContexTree::predict()` we need to update the tree *as if* we have just observed a 0, and compare the estimated likelihood of the old history to the new history to determine with what probability we should predict the next bit to be a 0.

1.2 Monte Carlo Search Tree

The code which implements the Monte Carlo Search Tree may be found in the search.cpp and search.hpp files.

The tree structure is implemented as follows. As provided in the interface, each node is a SearchNode class. The edges between nodes are implemented with C++ map object, which is a member of the parent node, containing pointers pointing to its child nodes. C++ map is essentially a red-black tree which allows insert and find objects according to keys in $O(\log n)$ time. In our implementation, the keys stored in an action node (the keys of all the child nodes of the action node) is the decoded binary representation of the actions. The key stored in an chance node (the keys of all the child nodes of the chance node) is the decoded concatenated binary representation of both observation and reward. Observation is appended to the end of the reward.

The sampling process is an implementation of Algorithm 1,2,3,4 in [1]. A random ties breaker of incremental fashion is used when multiple max values

can be found. For a list **v**, the elements of which are real values, the largest value can be selected through one iteration of **v** without knowing the length of **v** or the number of elements that have the largest value. The detailed ties breaker is as follows.

Data: **v**, $num\_ties = 1$, $v\_max = -\infty$
Result: Select the element with largest value with uniformly random ties breaker
for v *in* **v do**
  if $v > v\_max$ **then**
    $v\_max = v$;
    $num\_ties = 1$;
  else if $v\_max == v$ **then**
    $v\_max = v$ with probability $1 - \frac{num\_ties}{num\_ties+1}$;
    $num\_ties = num\_ties + 1$;
end

Algorithm 1: Ties Breaker

Assume that there are in total n elements have the max value x. According to this algorithm, the probability that one element is selected is $\frac{1}{n}$ for all n elements. Thus, we achieved a strict uniformaly random ties breaker.

In order to debug the MCST, we also added interface to let the AIXI directly samples from the true environments during the role out.

2 Implemented Environments

CoinFlip This is just the CoinFlip example in the provided template. The value for the `environment` option is `coin-flip`.

1d-Maze. The 1d-maze has been implemented as specified in the assignment specification. The length of the maze is configurable and so is the position of the goal/reward location within the maze. The environment will continue to run even after the agent has reached the goal once.
The value for the `environment` option is `1d-maze`.

Cheese Maze. This domain has been implemented as per the specification, with modifications to the reward values. The rewards have been scaled to non-negative integers as follows: Bumping into a wall rewards 0. Moving into a free cell that does not have the cheese rewards 9, and finding the cheese rewards 20. Again, the agent finding the cheese does not reset or stop the the environment.
The value for the `environment` option is `cheese-maze`.

TicTacToe. TicTacToe is implemented with an opponent (the environment) which makes random moves as per the specification. The rewards have been

scaled to be non-negative as follows: Making an illegal move rewards 0. Losing to the environment rewards 1. Making a legal but not game-ending move rewards 2. Ending the game with a draw rewards 3, and winning rewards 4. The games of TicTacToe are repeated, with a new game starting as soon as the previous is completed.
The value for the `environment` option is `tictactoe`.

Biased Rock-Paper-Scissors. Rock-Paper-Scissors is implemented with the opponent bias as specified. The rewards are scaled such that 0 is a loss, 1 is a draw and 2 is a win.
The value for the `environment` option is `biased-rock-paper-scissor`.

PacMan. This version of the PacMan was implemented with the effort to minimise the number of states remaining across cycles. As a by-product, the resulting code is less than 400 lines long, complete with power pill effect and comments. It were implemented to meet the requirements, including follow-up ones from communications, as closely as possible. The relevant words from the requirements are placed in the source code as comments enclosed in quotation marks, alongside corresponding implementations. The rewards were shifted with 128 but in this report they were shifted back.
Most importantly, we ended up programming a visualisation tool to detect bugs, although not required. The visualisation simply prints the game screen as ASCII characters to `stdout`. Because the output quickly fills up the console, the latest game screen will always on the bottom, effectively becoming animation if you stare at it, and the program is not running too fast (which true for meaning for simultaions). The visualisation is hardcoded and cannot be disabled, unless commented out in the source. During experimenting we mostly piped the `stdout` to a `.txt`file. In this case, running in Linux shell

```
tail -f <the output file containing stdout>
```

would then print the game in realtime. Actually it looked better with `tail -f` because of output buffering, hence showing no "stream of lines" but whole blocks at once.
The value for the `environment` option is `pacman`

3 Usage and Testing

3.1 Basic usage

The way to compile and run the code is largely unchanged from that in the assignment requirement. Assuming the necessary POSIX tools, to compile, `cd` into the source tree and run

```
make main
```

To execute the binary, run

```
./main coinflips.conf logfile
```

where the semantics of the arguments are exactly the same as in the requirement.

3.2 Test harness

A minimalist test harness, complete with parallel execution, was implemented with a single line of Bash script in `streambatch.sh`.
There are a few precautions before use. All `.conf` files of interest need to be under `cf/`. There needs to be a writable directory `logs/`, under which are corresponding sub-directories if `.conf` files are in sub-directories of `cf/`. The script does not create directories on its own.
For example, if we are to test `cf/abc/123.conf`, there has to be a directory `logs/abc/` for the script to work.
Then we simply pipe into the `stdin` of `streambatch.sh` the paths of `.conf` files to be tested, one line each.
For example, to test all `cf/short/*-coinflip.conf` files, run

```
ls cf/short/*-coinflip.conf | bash streambatch.sh
```

and then the outputs will be written to `cf/short/*-coinflip.txt`, `cf/short/*-coinflip.log` and `cf/short/*-coinflip.csv`. The `.txt` files are for the human-readable output, while `.log` and `.csv` being the longer, detailed logs as specified in the assignment requirement. If there are errors or warnings, they will be in the `.err` files.
At most 6 jobs by default will be run in parallel, which is suitable for most desktop computers.

3.3 Post-processing

To keep the reward values nonnegative, some of the environments have their rewards translated. However, the reward of the first cycle was always zero and thus rendered the figure of average rewards less informative. To compensate this without re-running all the experiments, the plots in this report use recalculated

$$\frac{\text{total reward}}{\text{cycle} - 1}$$

as the average reward.

4 Results

4.1 CoinFlip

For the CoinFlip we initially considered five standard configurations with

```
ct-depth = 4
agent-horizon = 2
```

and a head-probability of 0.1.
The first configuration "Always Exploit" has `exploration` set to 0 to keep the agent from exploring at all.
The second configuration "Always Random" has both `exploration` and `explore-decay` set to 1 to imitate pure random play.
The third and fourth configurations "0.1 Explore" and "0.95 Decay" represent two strategies of mixing exploration and exploitation, with

```
exploration = 0.1
explore-decay = 1
```

and

```
exploration = 1
explore-decay=0.95
```

respectively.
The last configuration "Real Env" is configured to always exploit and use an authentic `Environment` as an "oracle" to do the prediction. Note that the oracle does not tell the agent what the next toss will be head or tail, but instead toss a coin with the true probability (0.1), which is independent from the "real" toss in the next cycle.
Then we tested a final configuration specific to the CoinFlip environment, which always exploits with

```
ct-depth = 0
agent-horizon = 0
```

In Figure 1, we first display the average reward with respect to time.
To show the initial process of learning more clearly, a log-time plot with first 20 cycles removed (to cancel the fluctuation of average heads initially) will be displayed next.
We can see that the average reward of pure random playing converges to 0.5, and other strategies converge to higher values as expected.
However, we also found that search trees with "more than enough" capabilities actually worsen the performance. Only in the "D=0 H=0" configuration,the agent learnt to deterministically guess tail, which is the optimal strategy, and achieved an average reward of 0.9.
Otherwise, even with exploration turned off and true probabilities given, the agent tend to randomly guess with the 0.9 chance of tail and ends up with an average reward of around 0.8.

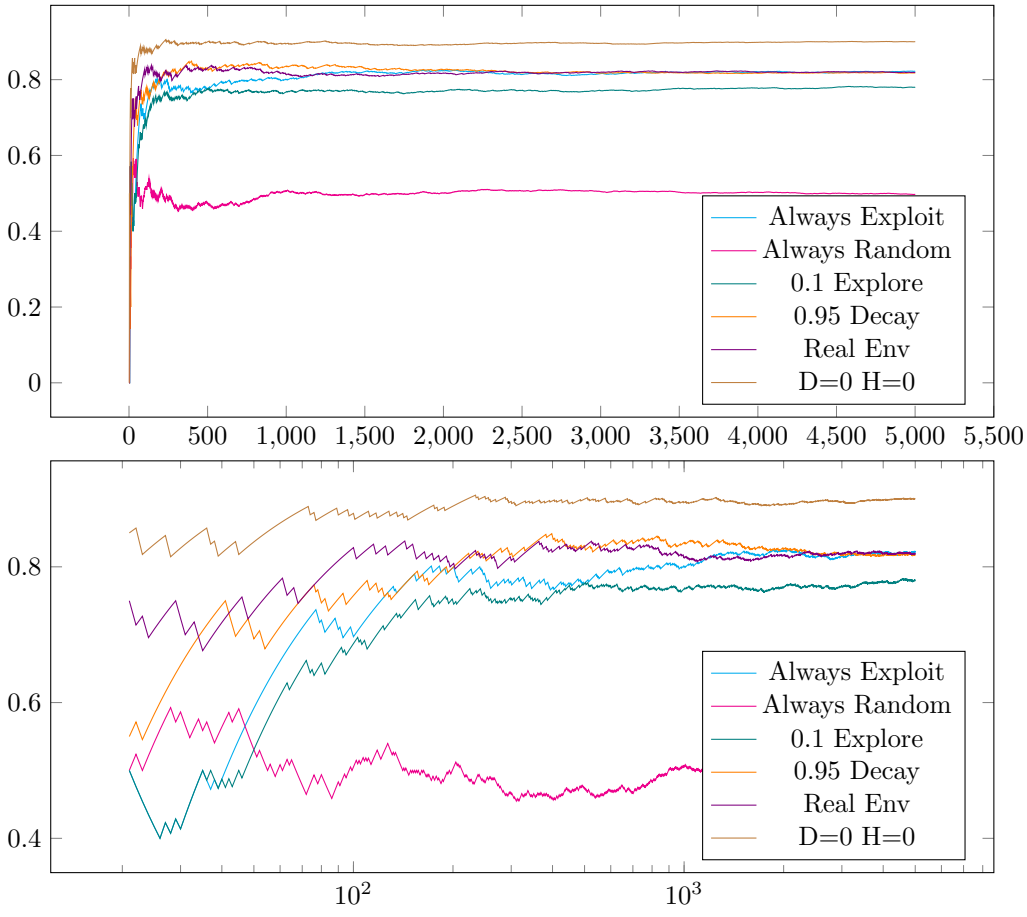


Figure 1: Average reward vs. time for CoinFlip

4.2 1d-Maze

First we tried the aforementioned five standard tests in the 1d-Maze environment, results shown in Figure 2 We can see that a fully random strategy converges to an average reward of around 0.25. The long-period oscillation of the other strategies hint a lack of memory. Constant exploration seems to be able to break this oscillation though, at the cost of peak performance.

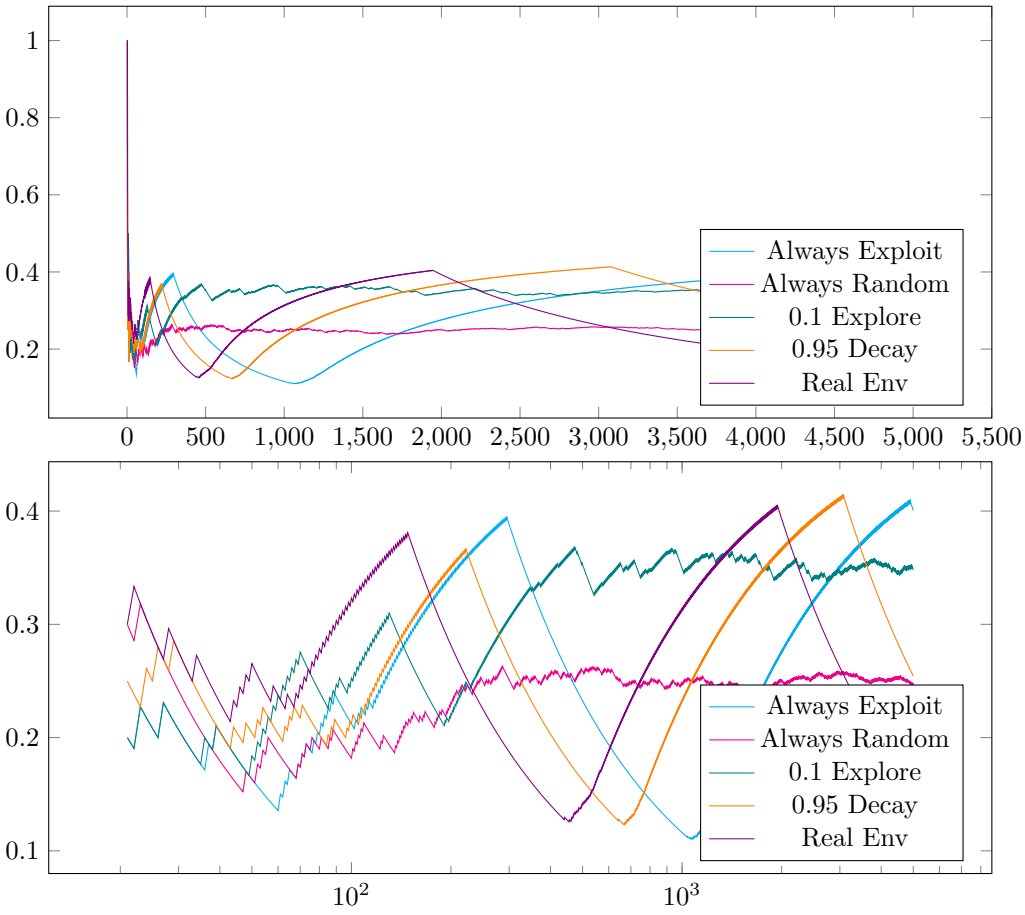


Figure 2: Average reward vs. time for 1d-Maze

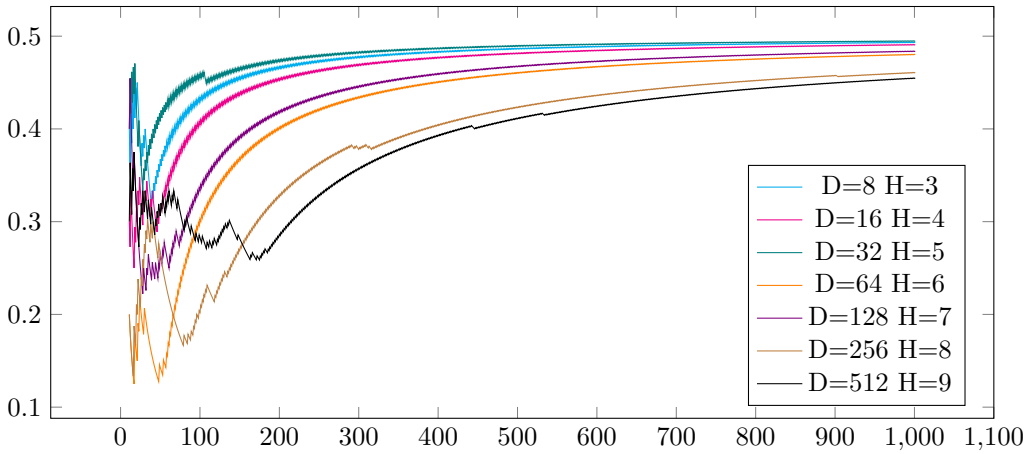


Figure 3: Average reward vs. time for 1d-Maze of Various (D,H)

Then we ran multiple tests with 0.95 decay but different tree depths and agent horizons, results shown in Figure 3. The optimal strategy would be to keep moving away from and then moving back to the target cell, with an average reward of 0.5. In this case all configurations seemed to converge to this maximum of 0.5, even with "more than enough" depth and horizon, different from that in CoinFlip.
Best results with more cycles are shown in Figure 4 along with log-time graph.

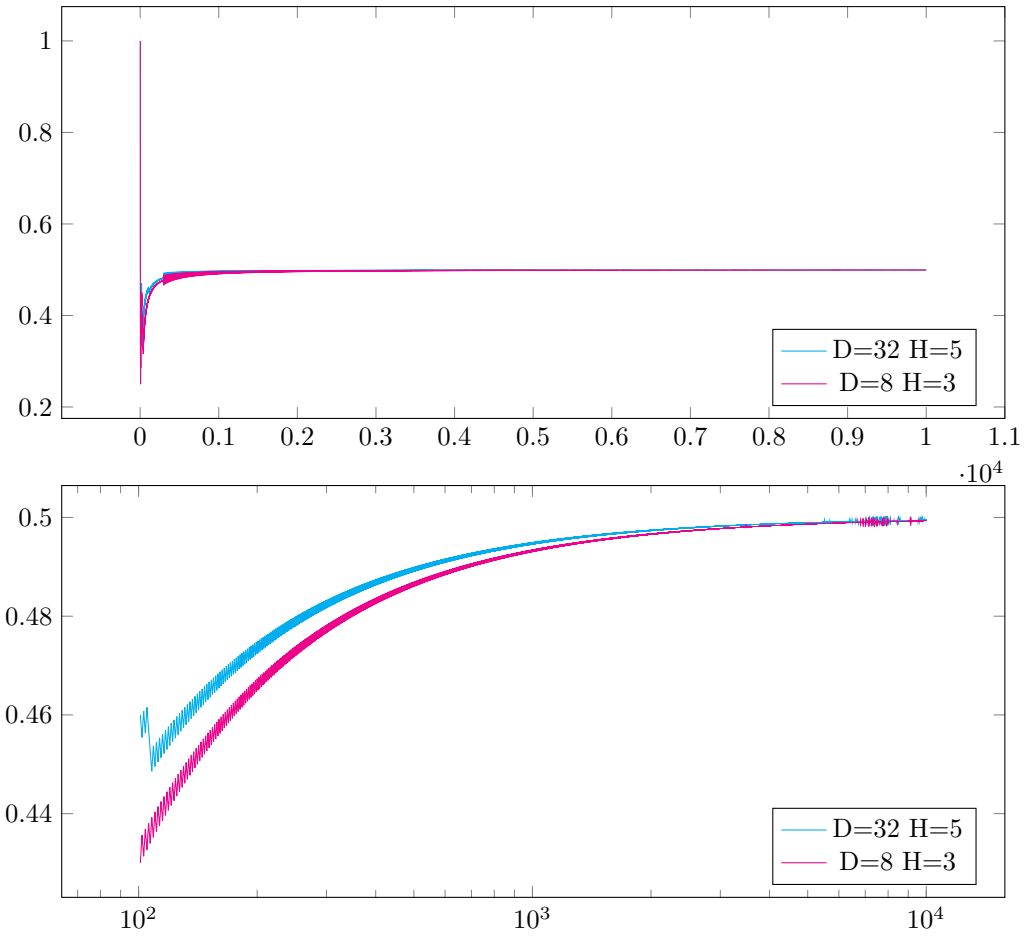


Figure 4: Average reward vs. time for 1d-Maze

4.3 Cheese Maze

As usual we tried the five standard tests in the Cheese Maze environment, results shown in Figure 5 We can see that a fully random strategy converges to an average reward of around 4.2. The advantages optimized agents have over a purely random agent is marginal, suggesting that we increase the tree depth and search horizon. Although the optimal strategy of this game is simply moving away from and then back to the cheese, The agent has to decide whether it is in the second column from history due to the perceptual aliasing.

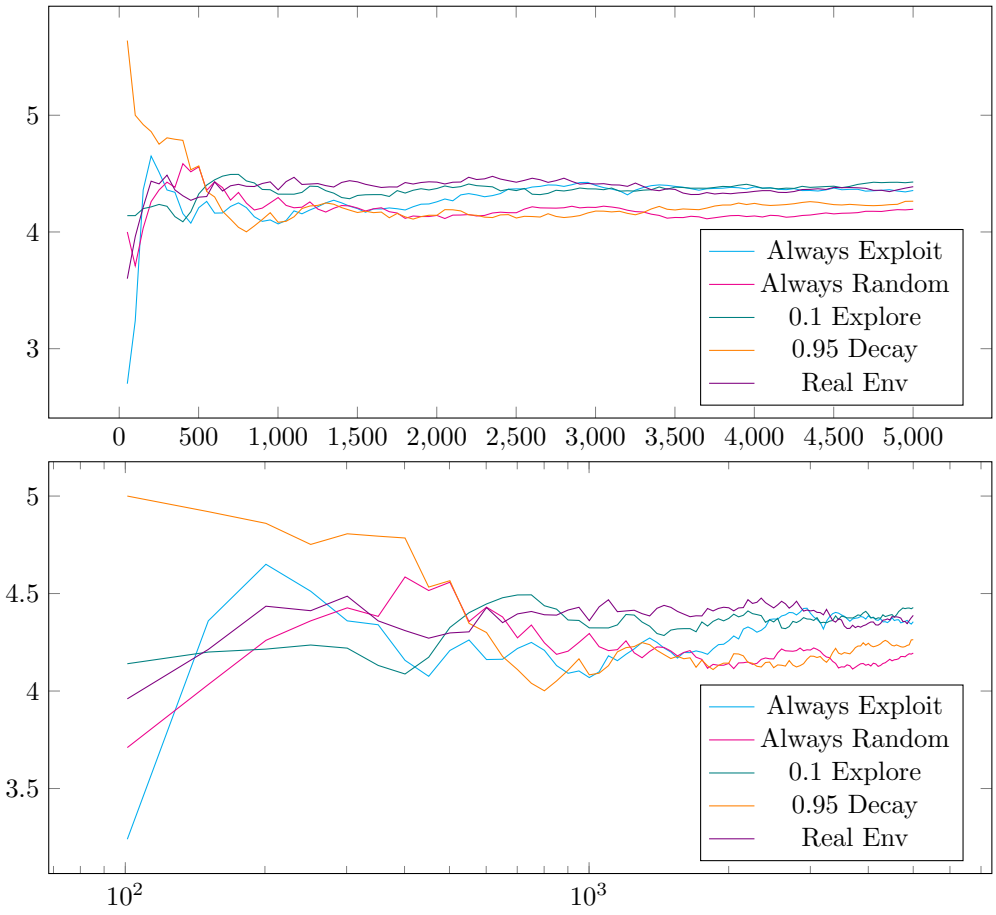


Figure 5: Average reward vs. time for Cheese Maze

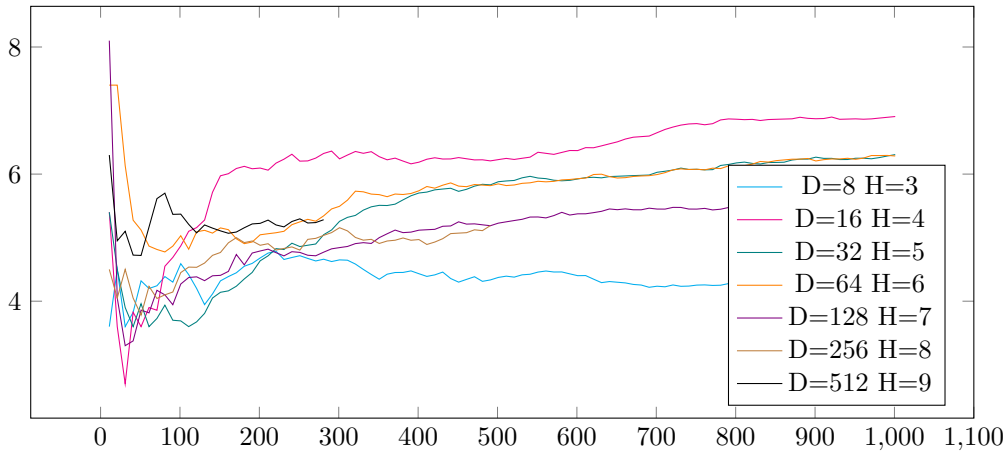


Figure 6: Average reward vs. time for Cheese Maze

Then we tested several configuration sizes, with horizon and tree depth scaling together, as shown in Figure 6.
It seems like the models are not the larger the better. More extensive tests resulted in what is displayed in Figure 7.
While we did not run the experiments until the average rewards visually moved to the optimal value 14.5 by moving back and forth onto the cheese position and receiving (shifted) rewards of 9 and 20 alternatively, manual inspection of the logfiles revealed that agents with the first four configurations did converge to this behaviour. The case "D=16 H=4 R" was configured to use the real environment for prediction, which did not show good results in this case. In the corner case with zero agent horizon, the agent ended up keep running into walls.
The conclusion we draw is that there is a hard minimum of tree depth, greater than 16 and less or equal to 32, with which the agent can converge to an optimal behaviour, although we didn't actually pin down this limit due to limited computational resource. More than that, deeper trees might make learning faster (in terms of cycles). The search horizon must be at least 1, best to be 2, but past that point, larger horizons quickly deteriorate learning rate.

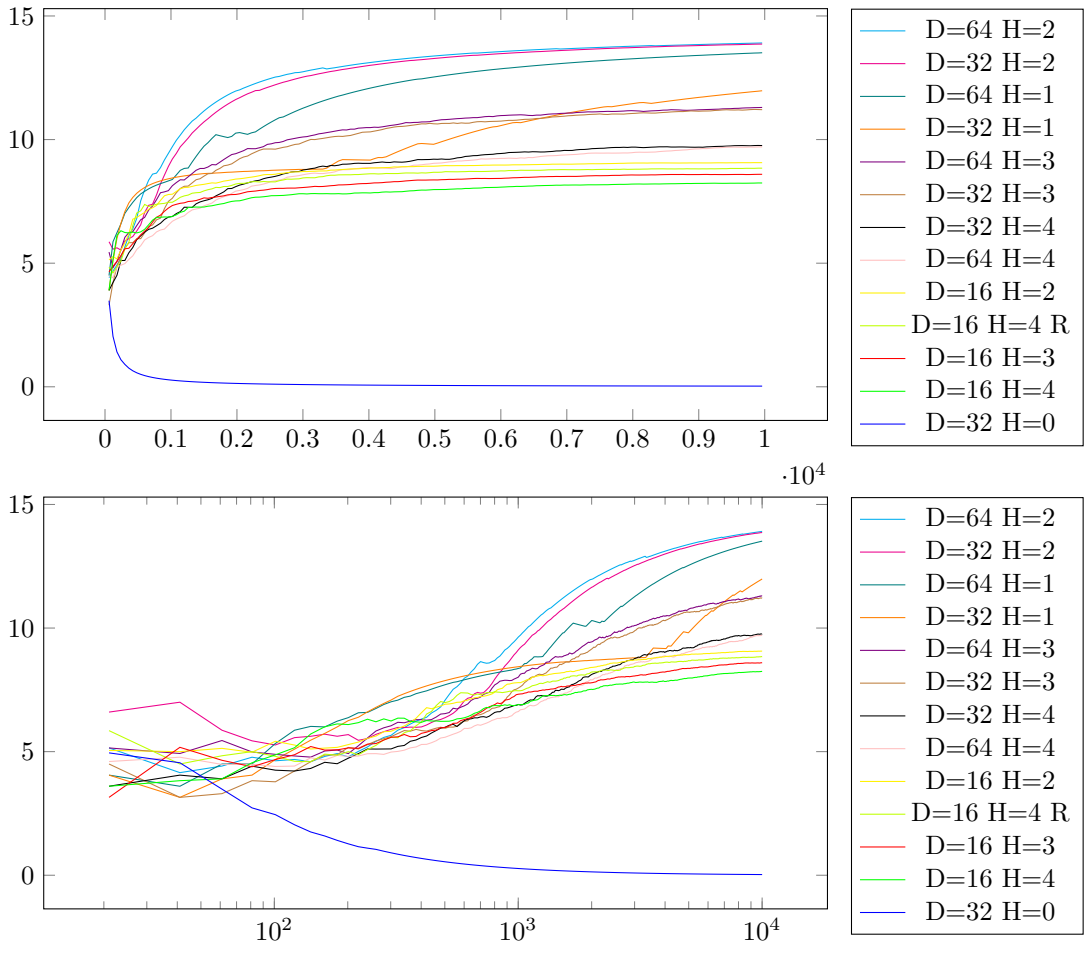


Figure 7: Average reward vs. time for Cheese Maze

4.4 Biased Rock-Paper-Scissors

As usual we did the five standard tests to get a basic idea from Figure 8. Again we found the default model too small and the advantages marginal. Luckily, this game was quick to simulate and did not require a complex model to play since there are only 3 possible actions and it is a 1D Markov process. So we were able to do extensive tests on this game, results shown in Figure 8. Only with tree depth of 16 and agent horizon of 2 did the agent converge to the optimal value of 1.2. It looks like with D=12 H=1 or D16 H1 it could converge to 1.2 as well, but we anticipated the simulation to be of too many cycles to to carry.
In general, learning the Biasd Rock-Paper-Scissors seems to be a slow task for hour agent, while the game it self being simple. The optimal agent horizon appears to vary with the tree depth, unlike that in the Cheese Maze problem.

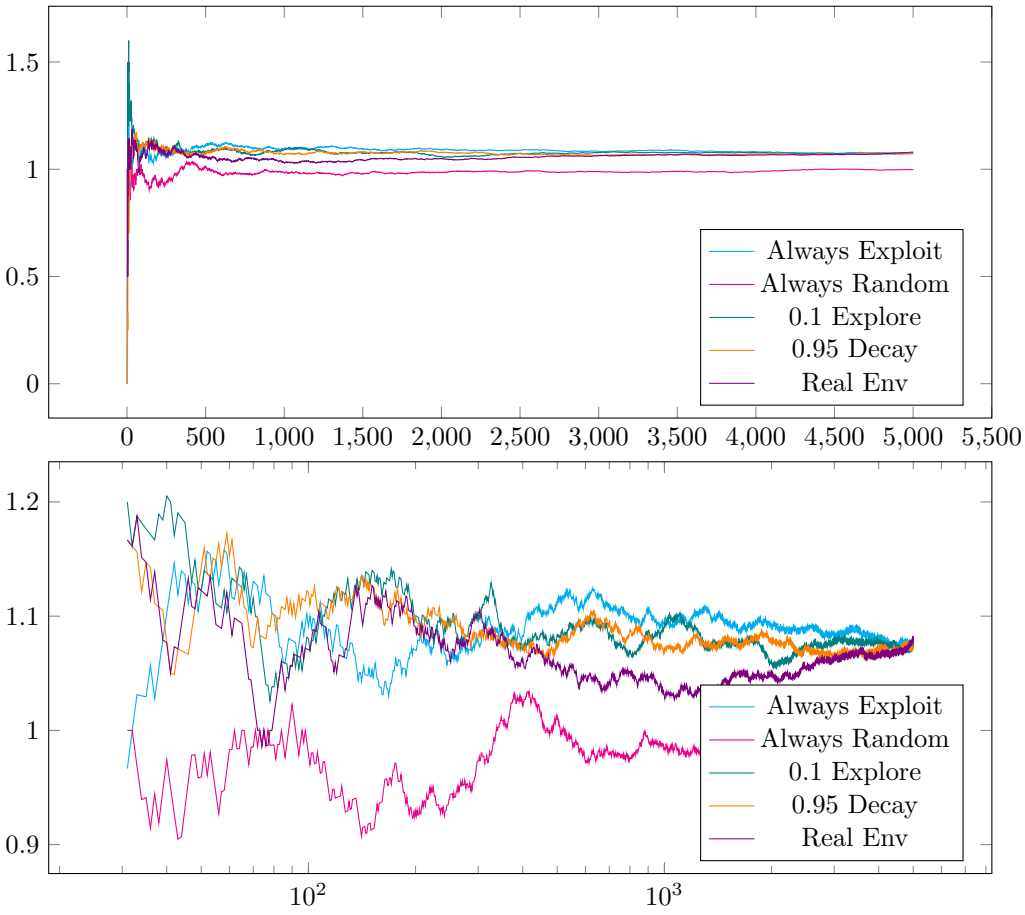


Figure 8: Average reward vs. time for Biased Rock-Paper-Scissors

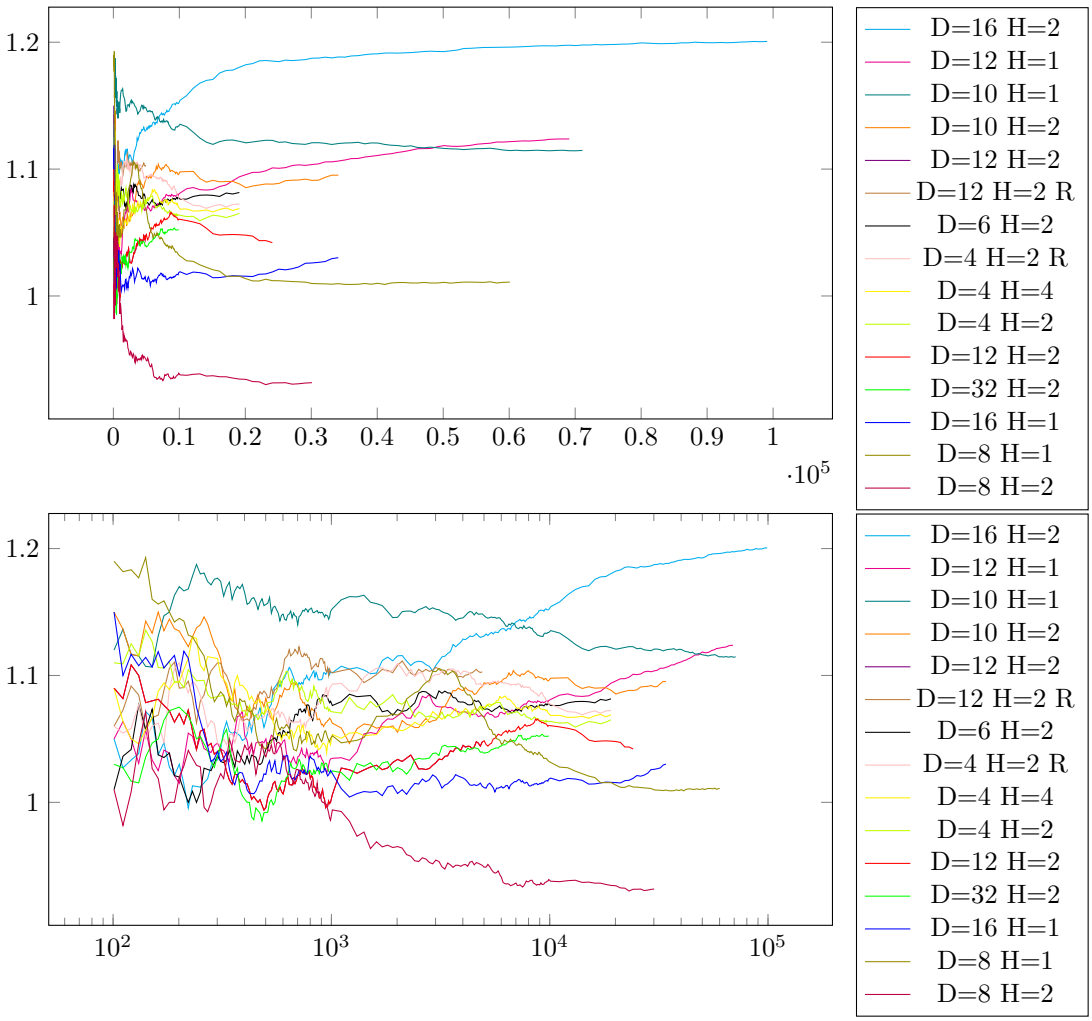


Figure 9: Average reward vs. time for Biased Rock-Paper-Scissors

4.5 TicTacToe

Again, we tried the five standard tests first and saw marginal advantage over random play as in Figure 10. The TicTacToe game is computationally expensive every cycle as there are 9 possible actions, while being a game that requires the most look ahead to play well among the games we have implemented.
Thus the two sets of hyperparameters sent to further tests shown in Figure 11 (D=32, H=5) and (D=128, H=7) were chosen somewhat arbitrarily. A few other configurations, mostly with smaller H were informally tested as well, only to be pruned early on due to computational constraints. Two versions with real environment oracle enabled were tested as well.
As can be seen from 11, all the agents showed gradual improvement without sign of convergence, and had to be stopped because of limited computational resources.
Only in TicTacToe, the oracle gave a very clear advantage, which is probably a nature of board games. The agents with oracle also ran three times as fast as the normal versions, resulting three times as many datapoints. Because of this a tuncated plot is provided in additionn to linear-time and log-time ones in Figure 11 to highlight the operation of the normal agents.

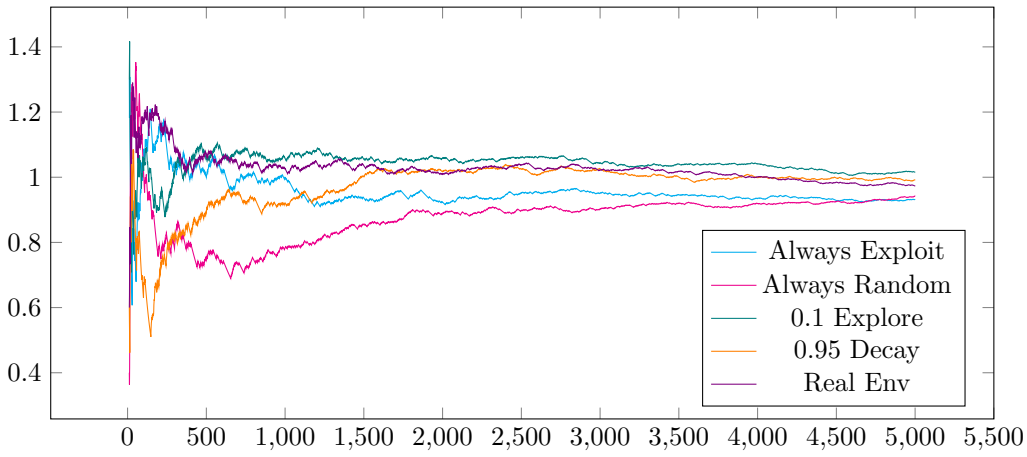


Figure 10: Average reward vs. time for TicTacToe

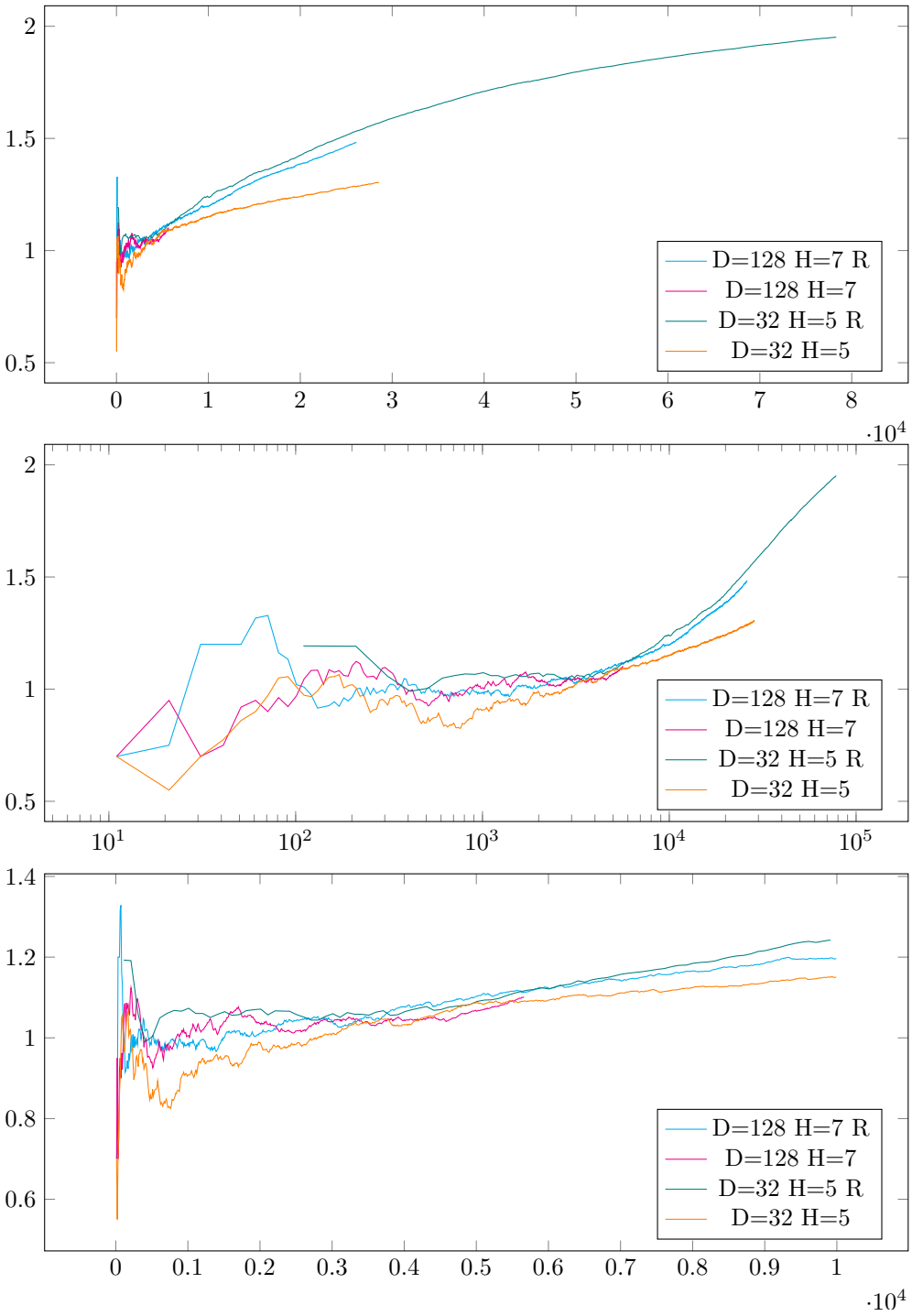


Figure 11: Average reward vs. time for TicTacToe

4.6 PacMan

For PacMan, we have tested many configurations but tried to decide which to let evolving early on, as the program is not parallelized and cannot be paused and restarted.

The observation per frame was 16, so there were not many choices for D. Then we start low with H. To be able to assess the perfermance of and prune agents early on, we opted for fixed exploration rates, although a few tests with exploration fall-off were also conducted with good (H,D) combinations we have found. Results with a representitive set of parameters are shown in Figure 13. The prefix mean reward is actually a little bit misleading in assessing agent's current performance, as the weights of the most recent actions becomes smaller and smaller as time increases. A better metric is a rolling mean over the rewards of the most recent, say, 1000 cycles. Since there were too many points to plot, we simply took mean over each 1000-cycle window, reducing total number of points at the same time. as you can see very clearly from 12, the performance of the best agents grew linearly with log-time, without sign of convergence.

Our most surprising finding was that the agent worked best with horizon of 1. More sets are placed in Figure 14 for completeness as there were too many lines to place in one plot. The best result in Figure 14 was also placed in 13 for ease of comparison.

While the original work did not suggest that it was better to use a larger H (H = 4), nor did it compare the performance under H=1 or H=2 with that under H=4, it is usually assumed that the hyperparameters presented in a work were chosen to show the algorithm performing at its best.

While we didn't run the original implementation, we could still approximately compare the performances by visually inspect the plots, that is with the Figure 10 of [1].

First of all, the Y axis in Fig. 10 of [1] was probably scaled by two. The intuition here was that if the pacman acts randomly, the dominate factor would be the penalty of running into a wall, with the expectation of (-10×0.5) per cycle for most cells, with two walls on two sides. plus the constant -1 of cost of moving. The nearest ghost has a distance of 12 to the pacman, but ghosts move initially at random, so the expectation of penalty caused by being eaten by ghosts should be much less than 50/12. Our simulation of random agent gave an average reward of around -5. A learning agent sometimes performs worse than random in the beginning, so we had an Y axis starting from -6 in our plots.

Then to determine if the performance of the agent in the original work showed a log-time growth, we first manually mesured some datapoints from Fig. 10 in [1] and plotted them on a log graph, as shown in Figure 15.

However, manual measurements are not only tedious, but also susceptible of subjectivity. To have a more accurate estimation, we rasterized the the Fig. 10 of [1], essentially taking a screenshort. Then we log-transformed the resulting image with ad-hoc Matlab script

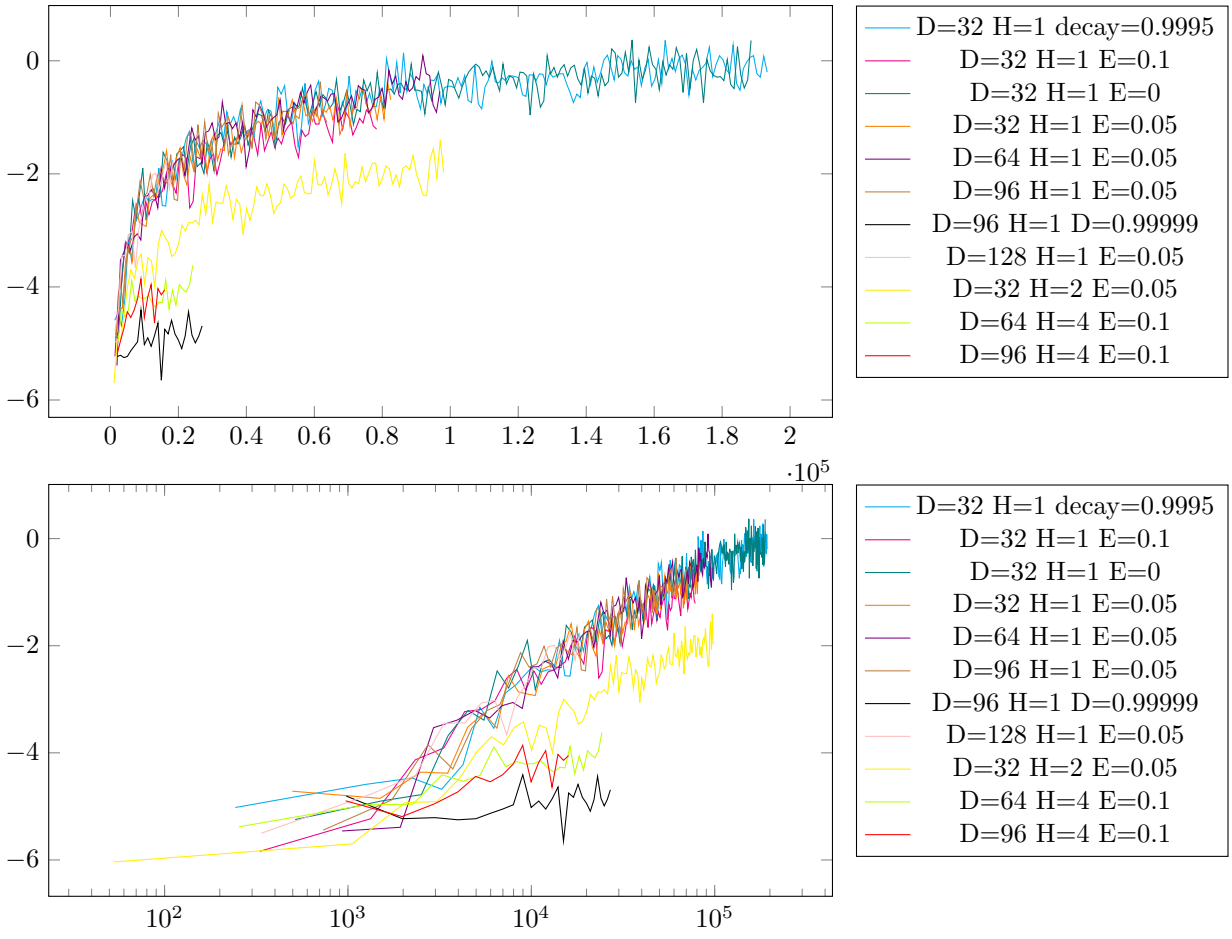


Figure 12: Average reward per 1000 cycles vs. time for PacMan

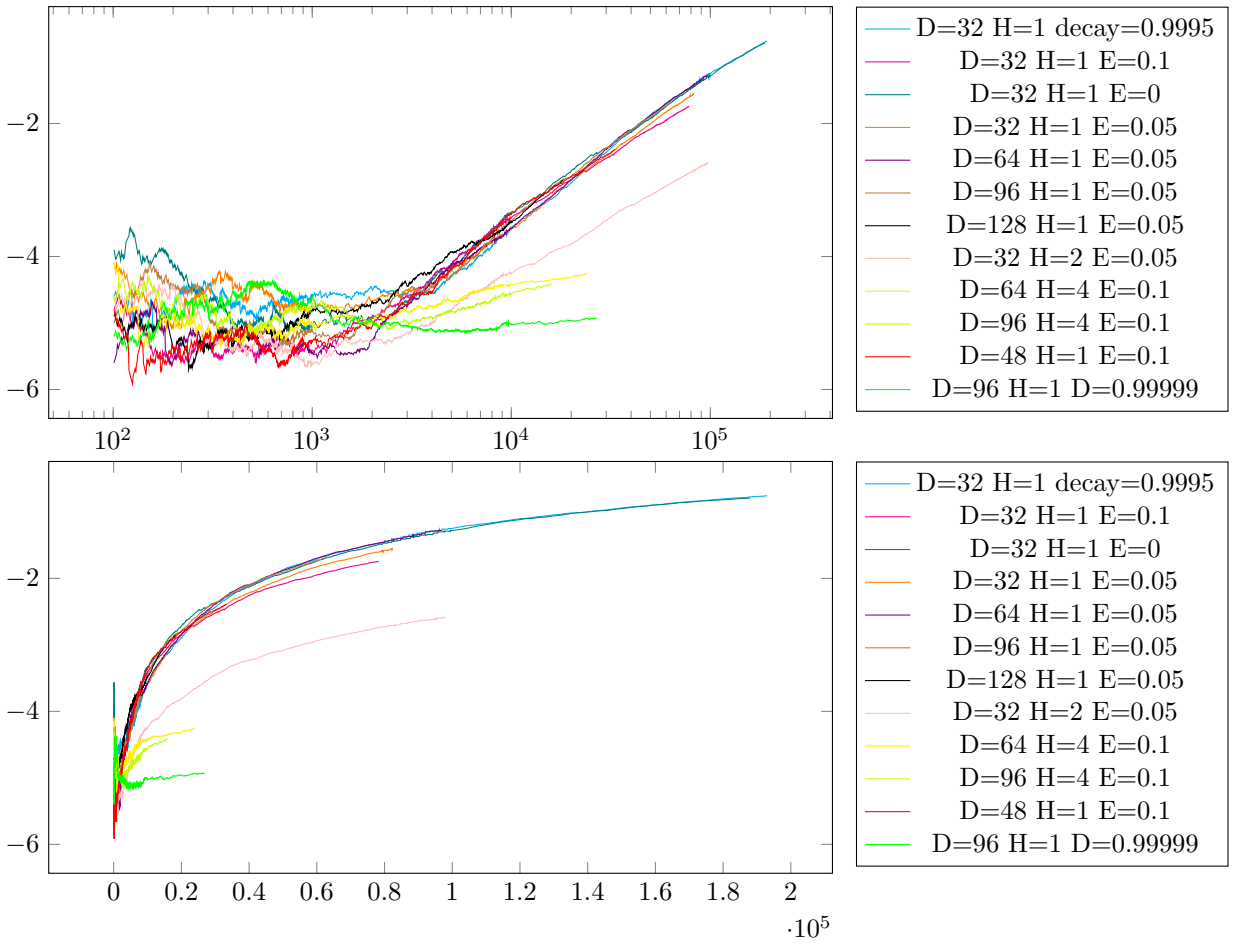


Figure 13: Average reward vs. time for PacMan

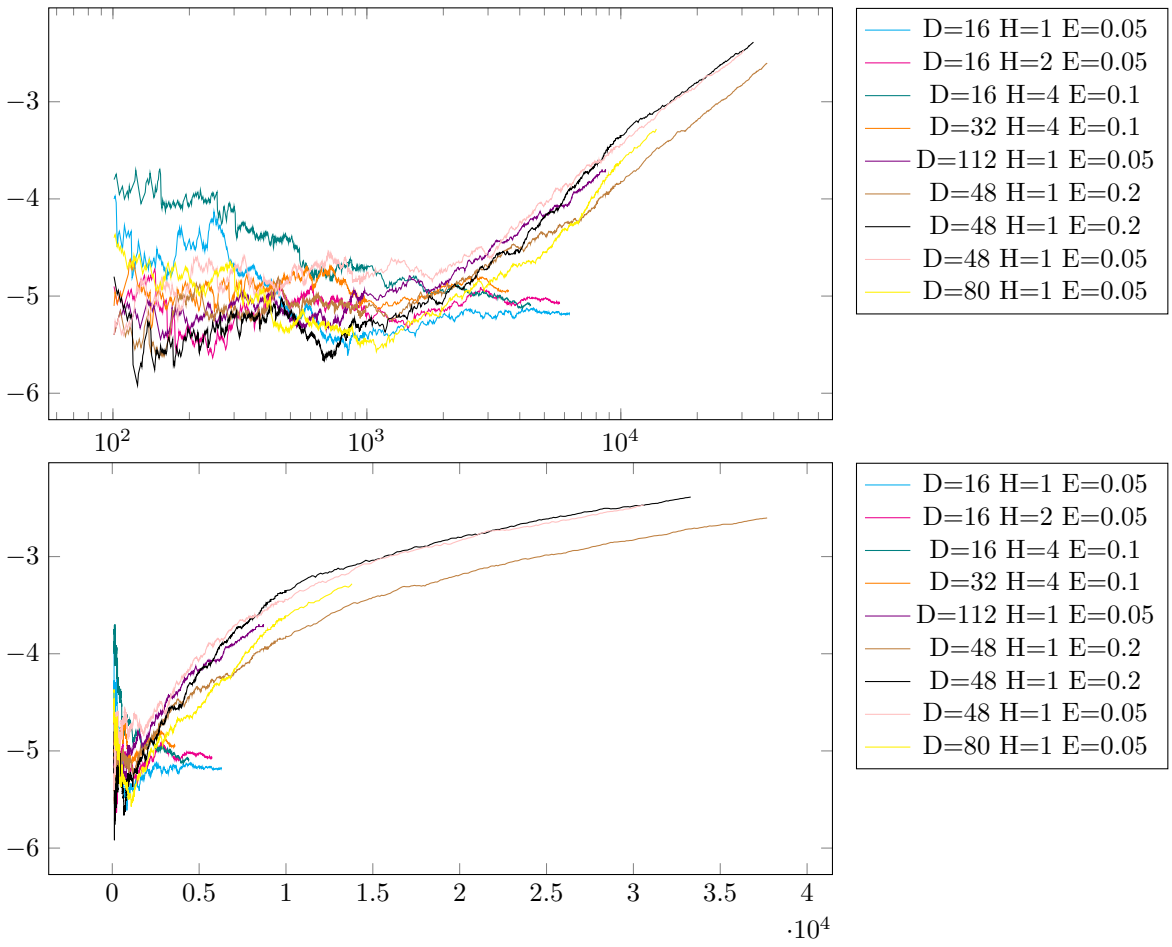


Figure 14: Average reward vs. time for PacMan (More cases)

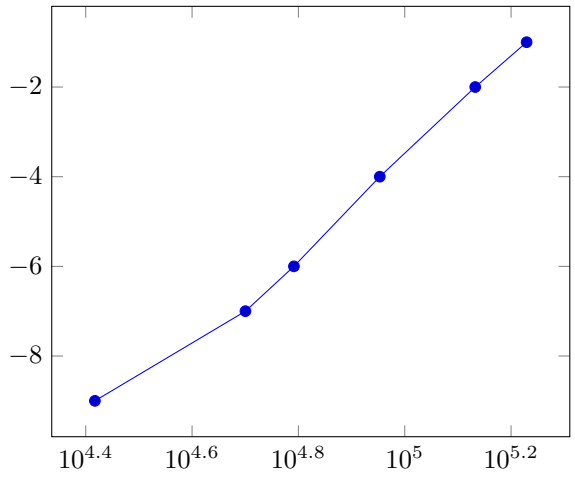


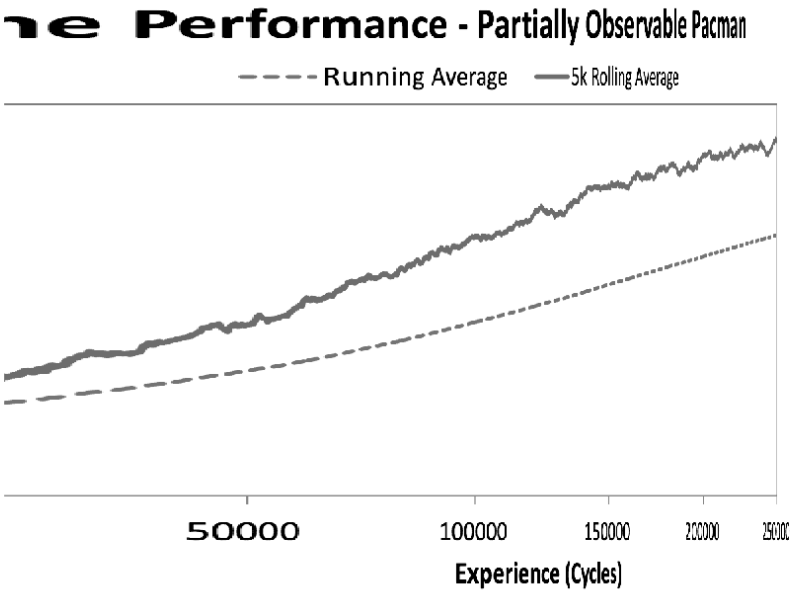
Figure 15: Manual measurements from [1]

```
g=rgb2gray(imread('jvlg.png'));
for x=1:1200; z(:,x) = g(:,ceil(1.005^x*1608/1.005^1200)); end; imshow(z);
```

where the number 1.005 is an adjustable parameter due to natural of the log transform, 1608 and 1200 being the width of original and transformed images respectively. Two transformed images with 1.005(lower) and 1.002(upper) are displayed in Figure 16, both showing clearly that the agent in the original work also entered a steady log-time growth of performance like our agents, although after about 50000 steps.
The super-log-linear growth before 50000 steps was probably due to the very slow fall-off of exploration, which was H=0.9999 and D=0.99999. Since the agent performed a steadily growing proportion of optimised steps, it looked like as if the agent was peforming better at a linear rate.
Note

$$0.99999^{10000} = 0.904837$$
$$0.99999^{50000} = 0.606529$$

Which means that before the 50000 cycle the agent would be still mostly performing random action. In our experiments in Figure 12 with same D and H, far before the 50000 cycle, the agent did give the impression of a linear growth as well, and performed very bad, just as the agent lived through 250000 cycles did.
Another thing to notice is that given such slow decay, in early stage of the simulation it was mostly performing random actions, which took negligible time each cycle compared to performing an optimised step. So although we were able to simulate 26979 cycles it got increasingly slow, to a point we decided not to continue.



10: Online performance on a challenging domain

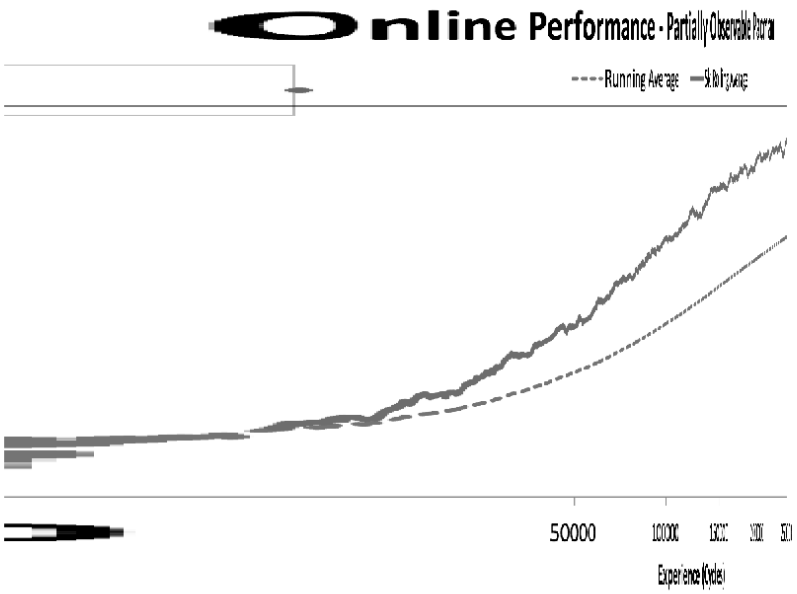


Figure 10: Online performance on a challenging domain

Figure 16: Log-transformed image of plot from [1], solid lines showing current performance.

Finally, a visual inspection of the `txt` files with ASCII graphics suggested that our agents with low horizon did learn to play pacman, in particular avoiding running into walls, evade ghosts, and eat food if nearby. To see for oneself, check

`logs/pm-*.txt`

e.g.

`logs/pm-32-1-d.txt`

You can also ask us to program a player to print the `txt` files with timing to make it animated.

5 Conclusion

We were able to let the agent play all the games as specified, in particular PacMan.
It remains to be an interesting problem how we can get the agent to play deterministically on CoinFlip. While easy to reason about, playing deterministically is not necessarily the first thing to occur to a human either. Most importantly, especially in the maze-related games, a simpler model seemed to be better, to the point of using a search horizon of 1. The size of tree, or memory capabilities, are not monotonic to game performance either. Directly giving the agent an accurate model does not necessarily result in the best performance.
In PacMan it seemed more important to be general than to be accurate. It is not hard to imagine that there could be very simple strategies just to "survive" in a maze-like environment, without actually learning an accurate model of the environment. In real world, many organisms live with basic reflexes.
Moreover, the PacMan game, due to its partial observablity, actually resembles a first-person shooter game [2], though there are no first-person shooter game as simple as PacMan, since it is a genre began in the 1990s. For a human, this type of games can be easily played "only with reflexes", at least to a leisure level(i.e. not competing or pursuing a high score), while the game environment is complex (modern FPS are huge in terms of storage and even RAM usage). At the opposite end are board games, which in our tests required us to use larger search horizons. In this games an accurate model of the world also gave clear advantage. Except for professionals and aficionados, playing Chess is usually more exhausting than playing Call of Duty, while programming a playble Chess game is a negligible effort than developing (as it far surpasses programming) Call of Duty, or even the first version of DOOM, the same to the computational resource needed to run the games.

References

[1] Joel Veness, Kee Siong Ng, Marcus Hutter, and David Silver. A monte carlo AIXI approximation. *CoRR*, abs/0909.0801, 2009.

[2] Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. Vizdoom: A doom-based AI research platform for visual reinforcement learning. *CoRR*, abs/1605.02097, 2016.