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 has been implemented with the effort to minimise the number of states remaining across cycles. As a by-product, the resulting code is less than 350 lines long, complete with power pill effect and comments. It has been made to follow 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. However, in the end there are still a few details up to different interpretation. First,
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.

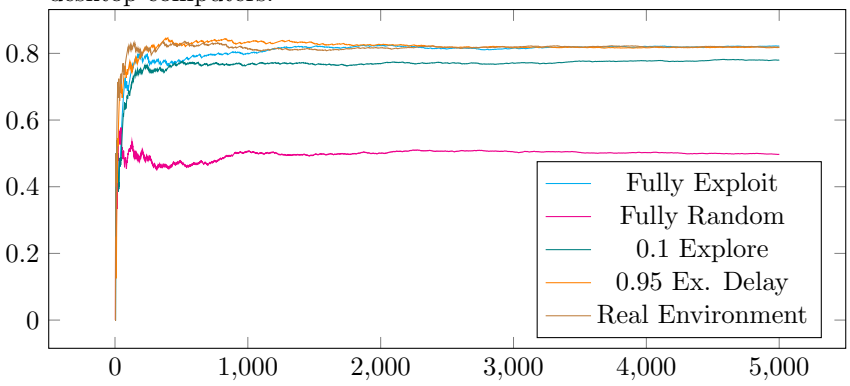
3.2 Test harness

A minimalist test harness, complete with parallel execution, is 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.



References

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