A Monte-Carlo AIXI Approximation: Tutorial

Daniel Visentin

May 4, 2011

Contents

Abstract

MC-AIXI-CTW is an intelligent agent which learns through experience how to perform well in some environment. This includes, but is not limited to, games such as Tic Tac Toe, Pacman, or Kuhn Poker. MC-AIXI-CTW is an approximation of the universally optimal AIXI agent by **?**. However, whereas the AIXI agent is incomputable (i.e. it cannot be run on a computer), the MC-AIXI-CTW provides a graceful tradeoff between optimality and computational resources. That is, the more time and memory given to MC-AIXI-CTW, the better it performs.

The MC-AIXI-CTW software package provides a simple implementation of the MC-AIXI-CTW agent, several environments, and associated documentation with the goal of being simple to set up, use, and extend. This tutorial gives an introduction to the various components of the package, instructions on how to run the agent on the included environments, how to adjust the parameters that affect the agents learning performance, and how to add new environments/extend the agent.

1 Introduction

This document is designed to give a practical introduction to using the MC-AIXI-CTW software package. The package implements a simple version of the MC-AIXI-CTW agent from **?**. Since the paper covers the agent in depth, we shall restrict ourselves to a brief non-technical overview of the agent before moving onto a tutorial of how to obtain/compile/run the agent. Following this will be more details on some of the aspects of running the agent as well as pointers on how to extend the code. For those who wish to get straight into running the agent, please see section 2.

1.1 MC-AIXI-CTW

The MC-AIXI-CTW agent seeks to interact intelligently within a particular environment. The environment can be just about anything, from a chess game to the entire universe. The agent interacts with the environment by performing actions (such as moving a piece on a chess board) and the environment interacts with the agent by providing observations (such as an image from a camera) and rewards for the agents actions. It is the goal of the agent to use its past interaction history to learn how to choose the actions which will lead to the greatest long-term reward.

There are two main components which combine to give the agent's action selection policy. The first is a model of the environment in which the agent attempts to model how the environment works and hence to predict how likely any given outcome is. Second is an algorithm for estimating the expected reward of each possible action by using the probabilities associated with the environment model. In particular, the MC-AIXI-CTW agent uses context-tree weighting for the environment model and the ρUCT search algorithm for estimating the expected rewards. More details of these algorithms can be found in the original paper by **?**.

2 Getting started

This section briefly outlines the process of getting the source code, compiling the code, running the agent on an environment, visualising the results, and generating the documentation. More details on the individual steps can be found in later sections.

2.1 Step 1: Prerequisites

2.1.1 Required

The only thing required to use the MC-AIXI-CTW software is a C++ compiler. We will assume that you are using either g++ on Linux (`http://gcc.gnu.org`) or visual studio on windows (`http://microsoft.com/visualstudio/`).

2.1.2 Optional

The following utilities are not required for the use of the core code.

- **Make:** Having the make utility (`http://gnu.org/software/make/`) on Linux will enable easier compilation.
- **Graphing:** The MC-AIXI-CTW package also comes with a python script for graphing the data, use of which is optional. In order to use the graphing utility, you must have a relatively recent version of python (`http://python.org`), numpy (`http://numpy.scipy.org`), and matplotlib (`http://matplotlib.sourceforge.net`).
- **Documentation:** In order to generate the source code documentation, you must have a utility called doxygen (`http://doxygen.org`) and a suitable LaTeXinstallation (`http://latex-project.org`). Doxygen extracts specially-formatted comments from within the source code in order to make the documentation. LaTeXis used to typeset the formulas and PDF documentation.

2.2 Step 2: Getting the code

2.2.1 Linux

Download the archive. Extract it into a directory called `mc-aixi-ctw` using the following command

```
tar -xf mc-aixi-ctw.tar.gz
```

For the duration of this tutorial we shall specify all file paths relative to the `mc-aixi-ctw` directory.

2.2.2 Windows

Download the archive. Extract the zip folder and open the solution file `mc-aixi-ctw.sln` in Visual Studio.

2.3 Step 3: Compiling the code

2.3.1 Linux

Compiling the code on Linux requires a C++ compiler and (optionally) the "make" utility. Assuming you have both of these, the code can be compiled simply by running the command:

```
make
```

from within the `mc-aixi-ctw` directory. This will result in the creation of the `aixi` executable within the `mc-aixi-ctw` directory. Because the make command avoids recompiling code that has not changed, it is necessary to clean the results of a compilation in order to recompile everything from scratch:

```
make clean
make
```

While this is not always necessary, some changes to the source code might require a complete recompilation.

Alternatively, in the absence of the make command, one can use a compiler directly. The appropriate command for g++ is:

```
g++ -O3 -Wall -o aixi src/*.cpp
```

2.3.2 Windows

To compile the code, simply press F7. This is not a necessary step as running the code from within Visual Studio will automatically compile it.

2.4 Step 4: Running the code

In order to run the code and have it do something useful, it is necessary to specify a few configuration options. Most importantly, the environment for the agent to interact with and the agent's search parameters. Several example configuration files are available from the `conf` directory, one for each environment. Details of all the configuration parameters and their possible values can be found in section 3. For now, we will run the agent on a maze environment using one of the example configuration files. The environment consists of a 4×4 empty maze, the bottom-right corner of which is the "goal state." The agent is able to move around the maze using the four cardinal directions (up, down, left, right). Reaching the goal state gives the agent a reward and transports it to another square at random.

2.4.1 Linux

Running the agent is as simple as invoking the following command from within the `mc-aixi-ctw` directory:

```
./aixi conf/4x4-maze.conf log/4x4-maze.log
```

In this example, `./aixi` refers to the executable we compiled in the previous step, `conf/4x4-maze.conf` is configuration file for the 4×4 maze environment, and `log/4x4-maze.log` is the file to which the agent will log its progress. The agent-environment interaction can be stopped at any time by pressing Ctrl-C (or otherwise stopping the process) or at a particular time using certain configuration options (see section 3).

2.4.2 Windows

There are two steps in running the code from visual studio. First, to specify command-line arguments, select the mc-aixi-ctw project in the solution explorer, right-click on it and select the properties option (figure 1). Then, under Configuration Properties select Debugging. Edit the Command Arguments option (figure 2) so that it reads

```
conf/maze-4x4.conf log/maze-4x4.log
```

The first argument gives the configuration file for the environment while the second gives the file to which the agent will log its progress. Exit from the properties dialog by clicking OK. Now to run the code simply press F5 or click on the play button in the toolbar.

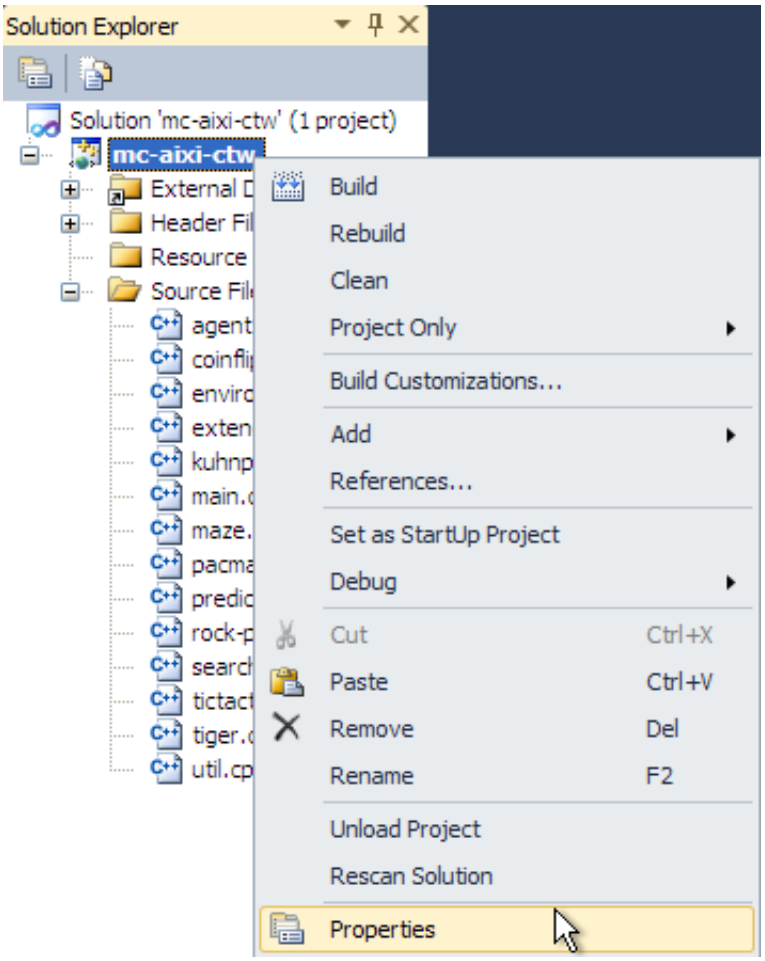

Figure 1: Project properties

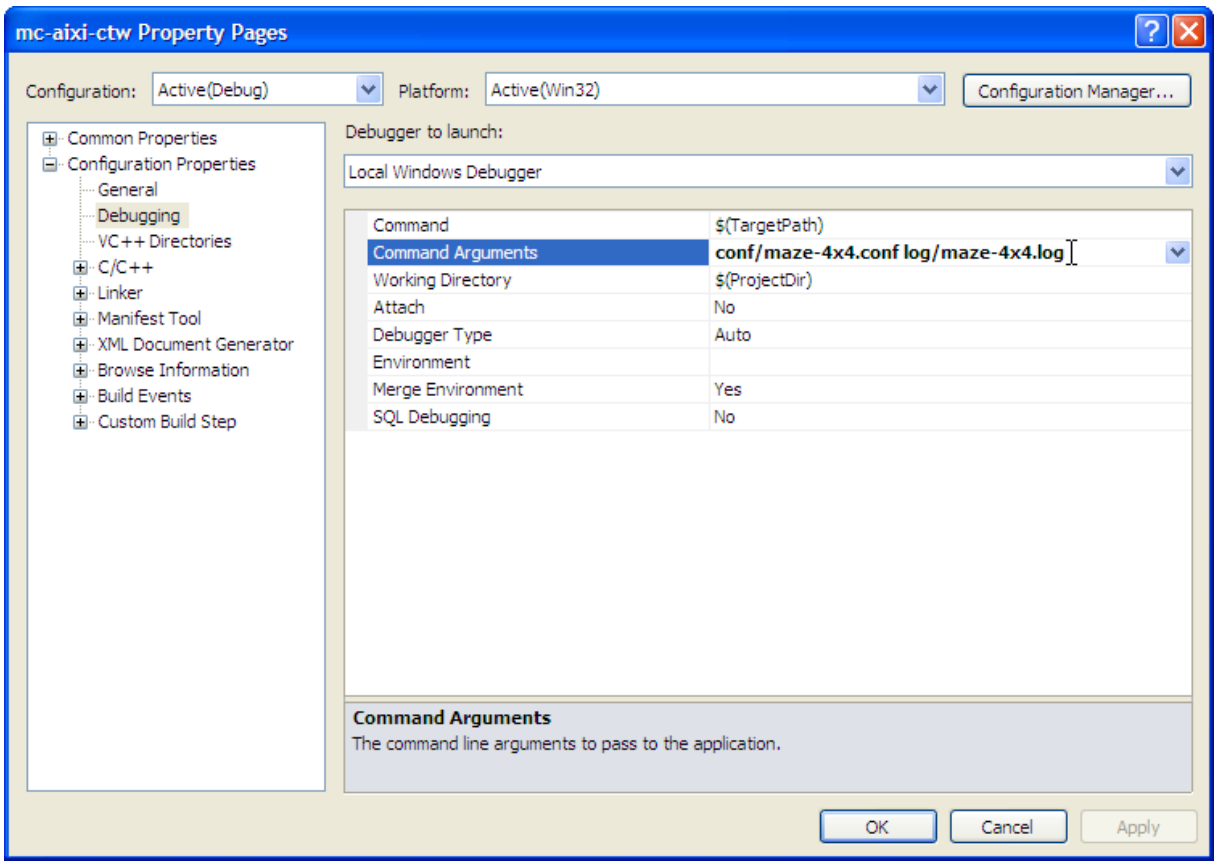

Figure 2: Command-line arguments

2.5 Step 5: Visualising the results (Optional)

Having run the agent, the results of the agent's interaction are available in the log file `log/4x4-maze.log`. The log file is a comma-separated-value (CSV) file with one line per cycle of interaction plus an initial header line. The header line gives the names of the quantities logged to the file while the all the other lines give the corresponding values for a particular cycle.

Provided with the source code is a graphing utility that enables you to visualise the results. Alternatively, you can use your own selection of tools (e.g. excel or matlab) to analyse the data. The graphing utility has certain prerequisites in addition to those of the agent, these are detailed in section 2.1. Running the graphing utility is simply a matter of invoking the following command

```
python graph.py
```

from within the mc-aixi-ctw folder. The graphing utility will generate a set of graphs for each log file in the `log` directory, each of which will be placed in a sub-directory of the `graph` directory.

2.6 Step 6: Generating documentation (Optional)

Generating the HTML and PDF documentation requires the use of doxygen and LaTeX as detailed in section 2.1. These utilities extract specially-formatted comments from the code and compile them into a set of web pages or a PDF for easy viewing. To generate the documentation, simply run the command

```
doxygen mc-aixi-ctw
```

from within the `doc` directory. The HTML documentation can be viewed by opening the file `doc/html/index.html` in a web browser. The LaTeX documentation requires an initial compilation step:

```
pdflatex refman.tex
```

run from within the `doc/latex` directory. The PDF documentation can then be accessed by viewing the file `doc/latex/refman.pdf`.

3 Configuration options

The program is configured through the use of key/value pairs in a configuration file. Each line of the file may contain at most one assignment of the form

```
key = value
```

with any character following a # being ignored as a comment. In order to tell the program which configuration file to use, the path of the configuration file is passed as the first command-line argument to the executable. For example, to use a file `conf.conf` we would invoke

```
./aixi conf.conf
```

Configuration options can be separated into a number of categories which we shall address below. Example configuration files can be found in the `conf` directory.

3.1 Agent configuration

These options configure the agent, particularly the effort it puts into searching for the optimal action.

- **agent-horizon:** The depth of the agent's search horizon. When the agent considers choosing a particular action, it estimates the action's consequences a certain number of cycles into the future. The search horizon specifies the maximum number of cycles to look ahead. *Default value:* 5. *Valid values:* positive integers.
- **ct-depth:** The maximum depth of the context tree used by the agent. Larger values enable the agent to more accurately model complex environments but require increased computation and memory resources. *Default value:* 30. *Valid values:* positive integers.
- **exploration:** The probability that the agent chooses an action at random instead of using the ρUCT search. *Default value:* 0.0 (i.e. no exploration). *Valid values:* decimal values between 0.0 and 1.0 inclusive.
- **explore-decay:** The rate at which the exploration probability decreases each cycle. In particular, if e is the initial exploration probability and c is the explore-decay then the exploration rate after cycle t is $c^t e$. *Default value:* 1.0 (i.e. no decay). *Valid values:* decimal values between 0.0 and 1.0 inclusive.
- **mc-simulations:** The number of Monte-Carlo simulations to perform when choosing an action. More simulations are more likely to give accurate estimates of each actions expected utility but require increased computation and memory resource usage. *Default value:* 300. *Valid values:* positive integers.
- **terminate-age:** The number of cycles of interaction between the agent and environment. When this number is reached, the program terminates. A value of 0 will cause the agent and environment to interact indefinitely. *Default value:* 0. *Valid values:* nonnegative integers.

3.2 Environment configuration

These options specify general environment properties. See the subsections for options which apply to specific environments.

- **environment:** Determines which environment the agent will interact with. *Valid values:* coin-flip, extended-tiger, kuhn-poker, maze, pacman, rock-paper-scissors, tictactoe, tiger.
- **action-bits/observation-bits/percept-bits/reward-bits:** Specifies the maximum number of bits needed to encode an action/observation/percept/reward. These are set by the environment when it is created.
- **max-action/max-observation/max-reward:** Actions, observations, and rewards are passed between the agent and environment as nonnegative integers. These options specify the maximum action that can be sent to the environment and the maximum observation/reward that can be received from the environment. These are set by the environment when it is created.

3.2.1 Coin-flip:

The coin-flip environment involves the environment flipping a biased coin and the agent attempting to predict which side it will land on.

- **coin-flip-p:** The probability of the biased coin landing on heads. *Default value:* 0.7. *Valid values:* Decimal values between 0.0 and 1.0 inclusive.

3.2.2 Extended tiger:

- **tiger-listen-accuracy:** The probability that listening for the tiger while seated will correctly identify which door the tiger is behind. *Default value:* 0.85. *Valid values:* Decimal values between 0.0 and 1.0 inclusive.

3.2.3 Maze:

The maze environment represents a maze through which the agent must navigate. Configuration involves specifying the structure of the maze, the rewards the agent receives from each square, as well as the type of observations given to the agent. Examples configuration files include `conf/cheesemaze.conf` and `conf/maze-4x4.conf`.

- **maze-num-rows:** The number of rows in the maze. *Valid values:* Nonnegative integers.
- **maze-num-cols:** The number of columns in the maze. *Valid values:* Nonnegative integers.
- **maze-layout**n**:** The structure of row n of the maze ($1 \leq n \leq$ maze-num-rows). The value should be a contiguous sequence containing maze-num-cols symbols as follows:
  - **@** Represents an impassable square in the maze.
  - ***** Represents an empty square which can be teleported to.
  - **&** Represents an empty square which cannot be teleported to.
  - **!** Represents an empty square which can be teleported from.

  If the agent moves into a **!** square it will teleport at random to a ***** square. The maze must contain at least one square capable of being teleported to.
- **maze-rewards**n**:** Comma-separated list of rewards for each square in row n ($1 \leq n \leq$ maze-num-rows). If the agent enters (or attempts to enter) a particular square, it receives the corresponding reward, regardless of whether it can actually enter the square or not. Rewards are translated by the program so that the minimum reward in the maze has value 0. *Valid values:* Integers.
- **maze-observation-encoding:** The type and encoding of observations received by the agent. *Valid values:*
  - **uninformative:** The agent receives the same observation each cycle.
  - **walls:** The agent receives an observation specifying whether there are walls (**@**) above, below, left, or right of its current position.
  - **coordinates:** The observation specifies the coordinates of the agent in the maze.

3.2.4 Tiger:

- **tiger-listen-accuracy:** The probability that listening for the tiger will correctly identify which door the tiger is behind. *Default value:* 0.85. *Valid values:* Decimal values between 0.0 and 1.0 inclusive.

3.3 Miscellaneous

These options do not apply directly to either the agent or environment.

- **random-seed:** Used to set the random seed of the program. Repeatedly using the same value across different runs of the program should (assuming no other changes) result in the same sequence of generated random numbers and hence the same sequence of interactions between the agent and environment. *Default value:* 0. *Valid values:* nonnegative integers.
- **verbose:** Determines whether the program logs interaction information to the standard output as well as to the log file. When debugging, it is useful to set this option to true so as to see what is happening between the agent and environment. *Default value:* false. *Valid values:* true, false.

4 Log format

The agent-environment interaction is logged to a file in comma-separated value (CSV) format. Each line in the log file corresponds to an interaction cycle except for the first line which is a list of the names of the values being logged. The field names are as follows:

- **cycle:** The interaction cycle which the current line refers to. Numbered beginning from 1.
- **observation:** The observation received by the agent represented as an integer.
- **reward:** The reward received by the agent.
- **action:** The action taken by the agent represented as an integer.
- **explored:** 1 if the agent explored (i.e. chose an action at random) and 0 otherwise (i.e. the agent searched for an action).
- **explore rate:** The current exploration rate (probability that the agent will explore this cycle).
- **total reward:** The total reward accumulated by the agent up to and including this cycle.
- **average reward:** The average reward received by the agent from all cycles up to and including the current cycle.
- **time:** The time (in seconds) elapsed over the cycle.
- **model size:** The number of nodes in the agent's context-tree model.

To direct the program to log at a particular location (e.g. `log/mylog.log`), provide the path as the second command-line argument to the executable:

```
./aixi conf.conf log/mylog.log
```

5 Extending the code

This section will give some brief pointers about where to start if you wish to add something to/change the code.

5.1 Adding a new environment

We will detail the steps required to create and integrate a new environment into the code. It might be helpful to look at the coin flip environment which is a relatively simple example. For illustration purposes, we will assume the environment is called "jumping castle."

1. **Create files:** Create a new code (.cpp) and header (.hpp) file for your environment in the `src` folder. For example, `src/jumping-castle.cpp` and `src/jumping-castle.hpp`.

2. **Update makefile (optional):** If you use the make command to compile your code, it is necessary to update the makefile so it knows to compile your new environment. To do this, simply append "`src/jumping-castle.o`" to line in `Makefile` beginning with "aixi:".

3. **Include the environment:** Open the file `src/main.cpp` and locate the #include section at the top of the file. You should see a block of #include statements relating to environments. Add the line

   ```
   #include "jumping-castle.hpp"
   ```

   to this block. You should also add this line to the top of the `src/jumping-castle.cpp` file you previously created.

4. **Inherit from the Environment class:** All environments must inherit from the Environment class (`src/environment.hpp`). This is done by adding the line

   ```
   #include "environment.hpp"
   ```

   to the top of `src/jumping-castle.hpp` and creating a new class to represent the new environment:

   ```
   class JumpingCastle : public Environment {};
   ```

   Within the newly created class you must define a number of methods (in `src/jumping-castle.hpp`) and implement them (in `src/jumping-castle.cpp`). In particular:

   - **Constructor:** Create the constructor with prototype

     ```
     JumpingCastle(options_t &options);
     ```

     The constructor will be passed the dictionary of configuration options. It will then set-up the environment and assign an initial observation and reward to the variables m_observation and m_reward (inherited from Environment).

   - **Action/Observation/Reward range:** The new environment must specify the range of possible actions, observations, and rewards. Since the code can only handle interactions represented by nonnegative integers, it is simply a matter of specifying the maximum value of each type of interactions. This is done by inheriting the methods maxAction(), maxObservation(), and maxReward() from the Environment class and having them return the appropriate values. As an example, assume the

jumping castle environment has two possible actions: jump and bounce. We might assign these actions to the values 0 and 1 respectively, in which case we would have maxAction() return 1.

- **performAction():** The performAction() method is perhaps the most important thing to implement. The method takes an agents action and calculates the next percept (observation and reward), placing the values in the variables m_observation and m_reward. The function is inherited from the Environment class and has prototype

```
virtual void performAction(action_t action);
```

- **Define a print() function (optional):** The Environment class provides a default print function which returns a string representation of the current state of the class. Overriding this method enables the environment to return a more tailored representation but is not required.

5. **Enable the environment:** The final step is to allow the environment to be selected. Open the `src/main.cpp` file and navigate to the main() function. Within this function is a large if statement where each condition checks the value of the environment_name variable against a particular environment. You need to add a test for your new environment of the form

```
else if (environment_name == "jumping-castle") {
    env = new JumpingCastle(options);
}
```

6. **Compile and run:** Compile the code with the commands

```
make clean
make
```

At this stage you will have to fix any compilation errors due to your code before you continue.

Create a new configuration file `conf/jumping-castle.conf` that includes the line

```
environment = jumping-castle
```

and an appropriate selection of other options.

Run the executable with

```
./aixi conf/jumping-castle.conf log/jumping-castle.log
```

5.2 Changing the environment model

The MC-AIXI-CTW agent models the environment using a context-tree. The code related to the model is contained in the CTNode and ContextTree classes within the `src/predict.cpp` and `src/predict.hpp` files. In changing/extending the model, it is necessary to keep in mind that:

- The model will be updated each cycle with the most recent action and percept. Thus, each update should be efficient and the model size should not grow without bound. Furthermore, it should be possible to efficiently revert the model to a previous state.
- The model should be able to calculate the probability of future interaction sequences and to sample from this distribution.

5.3 Changing the action selection policy

The MC-AIXI-CTW agent uses the ρUCT search algorithm to choose an action each cycle. Each sample is performed using the SearchNode class from the `src/search.cpp` and `src/search.hpp` files. The number of samples and ultimate action selection is handled by the search() method of the Agent class from the `src/agent.cpp` and `src/agent.hpp` files. Finally, the playout() method of the Agent class is used to give an initial reward estimate for future interaction sequences. It is used the first time a particular interaction sequence is sampled by the ρUCT algorithm. Ideally, the playout policy should be computationally efficient. For example, the default playout policy chooses actions at random for its sample. In changing how action selection is performed it is possible to change some but not all of these components.

5.4 Logging additional data

Logging is done in the inner loop of the mainLoop() function (`src/main.cpp`) using the logger stream. To change what is being logged, simply locate the statements involving logger and make the necessary changes.

5.5 Efficiency

This implementation of MC-AIXI-CTW values simplicity over efficiency. For this reason there are several possible extensions/changes to the codebase that will result in a quicker agent. For example, the algorithm creates and deletes a lot of nodes in the context tree during the search phase. This can be made more efficient by using a preallocation of nodes, or modifying the context tree to support copy-on-write. A more substantial improvement would involve parallelising the search code.

6 About

This project developed out of an assignment during the COMP4620/8620 advanced topics in artificial intelligence course at the Australian National University (`http://cs.anu.edu.au/courses/COMP4620/2010.html`). The goal was to develop a simplified version of the MC-AIXI-CTW agent using C++.

6.1 Acknowledgements

The MC-AIXI-CTW algorithm was developed by **?**. The code in this project is based on the more heavy-weight implementation by Joel Veness (`http://jveness.info/software/default.html`). Thanks go to the following students who participated in the coding of the algorithm: Chirag Chatbar, Mayank Daswani, Aaron Defazio, Sotirios Diamand, Nitin Gupta, Jonathon Hunklinger, Qiaochu Li, Xiang Li, Joseph Noel, Alexander Fergus O'Neill, Karunanithi Prabhu, Fatemeh Rajabiyazdi, Sam Rathmanner, Paul Rivera, Wen Shao. Phuong Nguyen supervised the students. Marcus Hutter provided many useful comments on the code, documentation, and tutorial.