Australian National University

COMP4620/COMP8620

Advanced topics in AI / Foundations of AI

Practical Assignment 2

Maximum marks	25
Weight	25% of final grade
Submission deadline	Thursday, 22 October 2015, 23:59
Submission mode	Via Wattle or email to Tom Everitt
Questions to	Tom Everitt
Estimated time	25-30 hours per student
Late Penalty	20% per day

In the labs you have the opportunity to ask for further clarifications. So it is recommended that you start early, and figure out what problems you might encounter. Please indicate the actual time spent on the assignment, and separately the time used for preparation (reading/learning).

	Group Members	**Domains Assigned**
Group 1	TBA	Pacman
	TBA	Kuhn-Poker
	TBA	Biased Rock-Paper-Scissor
	TBA	4×4 Grid
	TBA	Tiger
Group 2	TBA	Pacman
	TBA	Tic-Tac-Toe
	TBA	Extended Tiger
	TBA	Cheese Maze
	TBA	1d Maze
Group 3	TBA	Pacman
	TBA	Tic-Tac-Toe
	TBA	Biased Rock-Paper-Scissor
	TBA	Extended Tiger
	TBA	Cheese Maze

Each group must implement the whole general MC-AIXI-CTW agent and apply it to the also to be implemented domains given in the table.

If your name does not show up in this table, but you want to participate, or if you're in this list but will not participate, please let us know asap. Rearrangement requests might be considered in the first lab session.

MC-AIXI-CTW Implementation

1 Introduction

Artificial Intelligence (AI) [RN10] has traditionally dealt with methods of optimising the behaviour of an agent in a particular environment. Recently there has been a movement towards Artificial General Intelligence (AGI) which focusses on developing agents that perform well in a wide range of environments. Universal AI (UAI) [Hut05] in particular is founded on a sound definition of rational intelligence [Leg08].

At the heart of UAI is the AIXI model, which is a mathematical but incomputable "solution" to the general AI problem. The advent of AIXI results from a unification of sequential decision theory and Solomonoff's universal induction scheme. More formally, AIXI is an agent that interacts with an environment in cycles $k = 1, 2, \ldots, m$. In cycle k, AIXI takes action a_k based on past percepts $o_1r_1 \ldots o_{k-1}r_{k-1}$ as defined below. Thereafter, the environment provides a (regular) observation o_k to AIXI and a real-valued reward r_k. Then the next cycle $k+1$ starts. Denote $\mathcal{A}$, $\mathcal{O}$ and $\mathcal{R}$ the (finite) action, observation, and rewards respectively. Also, $\mathcal{X}$ denotes the joint perception space $\mathcal{O} \times \mathcal{R}$. Given the above, AIXI is defined by

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

where $\xi(x_{1:n}|a_{a_{1:n}}) = \sum_{\nu \in \mathcal{M}} w_0^\nu \nu(x_{1:n}|a_{1:n})$ is a mixture environment model, $\mathcal{M} = \{\nu_1, \nu_2, \ldots\}$ is a model class of all chronological probability distributions [Hut05], $w_0^\nu > 0$ is a prior weight for each ν, and $\sum_{\nu \in \mathcal{M}} w_0^\nu = 1$.

The sums in the AIXI equation constitute the averaging process. Averaging and maximization have to be performed in chronological order, hence the interleaving of max and $\sum$. Students are referred to the course slides and [Hut05] for a detailed presentation and optimality properties of the model.

Although succinctly representing the notion of UAI, AIXI itself is incomputable. To be applicable in practice, approximations in one way or another are necessary to find efficient solutions to real world problems. Monte Carlo AIXI Context Tree Weighting [V+11], abbreviated MC-AIXI-CTW is the first down-scaled version of AIXI, which has been shown to work well with a wide range of problem domains. This promising approach is expected to draw considerable attention in the AI literature in coming years. In MC-AIXI-CTW, $\mathcal{M}$ is a model class of prediction suffix trees (variable-order Markov processes); each environment ν is represented by an action-conditional CTW with prior weight $w_0^\nu = 2^{-\text{CodeLength}(\text{Tree}^\nu)}$ [V+11]. The expectimax search is performed by Monte Carlo search which concentrates on the promising branches via an upper confidence bound tree algorithm.

The primary purpose of this project is to help students to get insights into theoretical concepts of algorithmic information theory, Bayesian sequence prediction, and especially the working dynamics of the AIXI model. Furthermore, through implementation exercises students have a chance of getting involved with using complex data structures and coding various sophisticated functions.

2 Background

In order to understand and get insights into the MC-AIXI-CTW model and related algorithms, students have to grasp or review various relevant concepts and methods. The core material of the project is the MC-AIXI-CTW approximation paper [V+11]. However, it is a dense article with a great deal of knowledge assumed. The following foundations are essential for understanding the MC-AIXI-CTW algorithms, and hence for correctly and succinctly implementing the MC-AIXI-CTW model and toy examples.

1. The agent-environment setup and fundamental reinforcement learning methods. Students are referred to [SB98] (Chapters 1-6).
2. Universal prior, Bayesian prediction and the AIXI model. Lecture notes of the course and the UAI textbook are key references to understand these concepts.
3. The predictive UCT algorithm developed by L. Kocsis and C. Szepesvári [KS06]. The underlying bandit problem described in [SB98] is the basis behind the UCT algorithm.
4. Information theory of data compression. Students are recommended to read chapter 5 in the standard textbook of the area [CT91].
5. Context tree weighting (CTW), the central representation for efficient model updating. The background is lucidly presented in [WST95], [WST97].

Students should start by carefully reading the MC-AIXI-CTW paper [V+11], and consult the above references and other materials in case of any difficulties.

3 Example domains

Besides implementing the whole general MC-AIXI-CTW agent, each group must implement a subset of the following domains as specified on the cover page:

1d-Maze. The 1d-maze is a simple problem from [CKL94]. The agent begins at a random, non-goal location within a 1×4 maze. There is a choice of two actions: left or right. Each action transfers the agent to the adjacent cell if it exists, otherwise it has no effect. If the agent reaches the third cell from the left, it receives a reward of 1. Otherwise it receives a reward of 0. The distinguishing feature of this problem is that the observations are uninformative; every observation is the same regardless of the agents actual location.

Cheese Maze. This well known problem is due to [McC96]. The agent is a mouse inside a two dimensional maze seeking a piece of cheese. The agent has to choose one of four actions: move up, down, left or right. If the agent bumps into a wall, it receives a penalty of -10. If the agent finds the cheese, it receives a reward of 10. Each movement into a free cell gives a penalty of -1. The problem is depicted graphically in Figure 1. The number in each cell represents the decimal equivalent of the four-bit binary observation the mouse receives in each cell. The problem exhibits perceptual aliasing in that a single observation is potentially ambiguous.

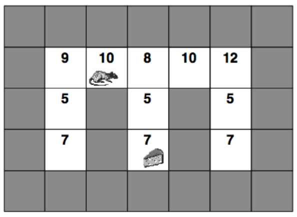

Figure 1: **The cheese maze**

Tiger. This is another familiar domain from [KLC95]. The environment dynamics are as follows: a tiger and a pot of gold are hidden behind one of two doors. Initially the agent starts facing both doors. The agent has a choice of one of three actions: listen, open the left door, or open the right door. If the agent opens the door hiding the tiger, it suffers a -100 penalty. If it opens the door with the pot of gold, it receives a reward of 10. If the agent performs the listen action, it receives a penalty of -1 and an observation that correctly describes where the tiger is with 0.85 probability.

Extended Tiger. The problem setting is similar to Tiger, except that now the agent begins sitting down on a chair. The actions available to the agent are: stand, listen, open the left door, and open the right door. Before an agent can successfully open one of the two doors, it must stand up. However, the listen action only provides information about the tigers whereabouts when the agent is sitting down. Thus it is necessary for the agent to plan a more intricate series of actions before it sees the optimal solution. Any invalid action (e.g. attempting to stand when already standing) will result in a penalty of 10.

State	**Action**	**Reward**
sitting	stand	-1
sitting	open door	-10
sitting	listen	-1
standing	stand	-10
standing	open door with tiger	-100
standing	open door with gold	30
standing	listen	-10

Table 1: **Rewards for extended tiger**

4×4 **Grid** The agent is restricted to a 4×4 grid world. It can move either up, down, right or left. If the agent moves into the bottom right corner, it receives a reward of 1, and it is randomly teleported to one of the remaining 15 cells. If it moves into any cell other than the bottom right corner cell, it receives a reward of 0. If the agent attempts to move into a non-existent cell, it remains in the same location and receives a reward of 0. Like the 1d-maze, this problem is also uninformative but on a much larger scale. Although this domain is simple, it does require some subtlety on the part of the agent. The correct action depends on what the agent has tried before at previous time steps. For example, if the agent has repeatedly moved right and not received a positive reward, then the chances of it receiving a positive reward by moving down are increased.

TicTacToe In this domain, the agent plays repeated games of TicTacToe against an opponent who moves randomly. If the agent wins the game, it receives a reward of 2. If there is a draw, the agent receives a reward of 1. A loss penalizes the agent by -2. If the agent makes an illegal move, by moving on top of an already filled square, then it receives a reward of -3. A legal move that does not end the game earns no reward.

Biased Rock-Paper-Scissor. This domain is taken from [FMRW09]. The agent repeatedly plays Rock-Paper-Scissor against an opponent that has a slight, predictable bias in its strategy. If the opponent has won a round by playing rock on the previous cycle, it will always play rock at the next time step; otherwise it will pick an action uniformly at random. The agents observation is the most recently chosen action of the opponent. It receives a reward of 1 for a win, 0 for a draw and -1 for a loss.

Kuhn Poker. The next example domain involves playing Kuhn Poker [Kuh50, HSHB05] against an opponent playing a Nash strategy. Kuhn Poker is a simplified, zero- sum, two player poker variant that uses a deck of three cards: a King, Queen and Jack. Whilst considerably less sophisticated than popular poker variants such as Texas Holdem, well-known strategic concepts such as bluffing and slow-playing remain characteristic of strong play.

In this setup, the agent acts second in a series of rounds. Two actions, pass or bet, are available to each player. A bet action requires the player to put an extra chip into play. At the beginning of each round, each player puts a chip into play. The opponent then decides whether to pass or bet; betting will win the round if the agent subsequently passes, otherwise a showdown will occur. In a showdown, the player with the highest card wins the round (i.e. King beats Queen, Queen beats Jack). If the opponent passes, the agent can either bet or pass; passing leads immediately to a showdown, whilst betting requires the opponent to either bet to force a showdown, or to pass and let the agent win the round uncontested. The winner of the round gains a reward equal to the total chips in play, the loser receives a penalty equal to the number of chips they put into play this round. At the end of the round, all chips are removed from play and another round begins.

Kuhn Poker has a known optimal solution. Against a first player playing a Nash strategy, the second player can obtain at most an average reward of $\frac{1}{18}$ per round.

PacMan. This domain is a partially observable version of the classic PacMan game. The agent must navigate a 17×17 maze and eat the food pellets that are distributed across the maze. Four ghosts roam the maze. They move initially at random, until there is a Manhattan distance of 5 between them and PacMan, whereupon they will

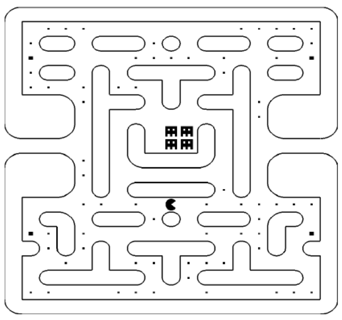

Figure 2: **A screen shot of the partially observable PacMan domain**

aggressively pursue PacMan for a short duration. The maze structure and game are the same as the original arcade game, however the PacMan agent is hampered by partial observability. PacMan is unaware of the maze structure and only receives a 4-bit observation describing the wall configuration at its current location. It also does not know the exact location of the ghosts, receiving only 4-bit observations indicating whether a ghost is visible (via direct line of sight) in each of the four cardinal directions. In addition, the location of the food pellets is unknown except for a 3-bit observation that indicates whether food can be smelt within a Manhattan distance of 2, 3 or 4 from PacMans location, and another 4-bit observation indicating whether there is food in its direct line of sight. A final single bit indicates whether PacMan is under the effects of a power pill. At the start of each episode, a food pellet is placed down with probability 0.5 at every empty location on the grid. The agent receives a penalty of 1 for each movement action, a penalty of 10 for running into a wall, a reward of 10 for each food pellet eaten, a penalty of 50 if it is caught by a ghost, and a reward of 100 for collecting all the food. If multiple such events occur, then the total reward is cumulative, i.e. running into a wall and being caught would give a penalty of 60. The episode resets if the agent is caught or if it collects all the food. Figure 2 shows a graphical representation of the partially observable PacMan domain.

Domain	A bits	O bits	R bits
1d-maze	1	1	1
Cheese Maze	2	4	5
Tiger	2	2	7
Extended Tiger	2	3	8
4×4 Grid	2	1	1
TicTacToe	4	18	3
Biased Rock-Paper Scissor	2	2	2
Kuhn Poker	1	4	3
Pacman	2	16	8

Table 2: **Binary encoding of the domain**

Notes. For simplicity, the implementation can only represent nonnegative rewards.

In order to represent some of the environments then, the rewards must be scaled and/or translated to ensure they are nonnegative. This scaling and transformation will not affect the results of the agent.

4 Implementation details

The goal of the project is to develop a simple, standalone version of the MC-AIXI-CTW agent. In particular, the agent will be single-threaded and use only light C++ and a minimal amount of the C++ standard library.

In order to emphasise implementation of the important algorithms, students will be provided with a skeleton implementation. This implementation specifies the main classes and functions and provides many of the simple or standard functions. Students may simply “fill in” the skeleton or modify it as they choose.

To be more specific, the following types of code will typically be provided:

- Reading and parsing of configuration files.
- Communication between the agent and environment including the encoding and decoding of actions and observations.
- Type definitions.
- The main structure of each class including important member variables.
- Class member variable getters/setters.
- Useful utility functions such as random number generation, logging, and parsing.
- Memory management (to some extent).

While the main implementation tasks for students include:

- The selection of an action at each time step as specified by the Monte Carlo Tree Search (MCTS) algorithm. This primarily consists of: sampling, playout, calculating expected reward, selecting actions, and updating/reverting agent state. The UCB constant C can be set to $\sqrt{2}$ or experimented with.
- The environment model as specified by the action-conditional Context Tree Weighting (CTW) algorithm and data structure. This primarily consists of: calculating probabilities, updating with observations, and reverting updates.
- High level control over predictions and the choice of actions.
- Implementation of the environment including initialisation and the generation of percepts in response to the agent’s actions.

The implementation tasks for students will be indicated by function stubs and “// TODO” comments in the source code. The structure of the library may be changed as necessary.

4.1 Agent overview

The agent is composed of several source files, each of which corresponds (roughly) to a particular component of the algorithm. Following is a summary of the files in the skeleton implementation.

agent.cpp/agent.hpp Specifies the Agent class which contains high-level code relating to action selection, model prediction, observation decoding, and action encoding. Also specifies a ModelUndo class containing information necessary to revert model updates.

main.cpp/main.hpp Reads and parses the configuration file, calls various initialisation code, and contains the main agent/environment interaction loop.

predict.cpp/predict.hpp Contains the CTNode and ContextTree classes which are used in the implementation of the action-conditional CTW algorithm. A context tree is represented by a single ContextTree instance and each node in the tree by a CTNode instance. The code in this file is used to predict the probability of a sequence, generate random future sequences, update the context tree, and revert the context tree. *Note:* The code makes use of logarithmic probabilities for numerical reasons.

search.cpp/search.hpp Contains the SearchNode class and various functions relating to MCTS.

util.cpp/util.hpp Contains useful functions such as random number generation. You may add any useful, miscellaneous functions to this file as necessary.

environment.cpp/hpp Contains the Environment class which specifies the interface for all environments. Each environment you implement should inherit from this class. For an example, see the CoinFlip class in the same file. You may wish to place more complex environments in their own (appropriately named) file.

4.2 Libraries

Except for the C++ standard library you may not use any external libraries or code. Furthermore, the use of the C++ standard library should be minimal.

4.3 Logging

You will be provided with functions for logging information about the agent and its interaction with the environment. You should use these funtions to output the information necessary for your analysis of the agents performance.

4.4 Style

The code should be consistent in terms of style and layout. We recommend following the conventions of the supplied code. In particular, the following conventions are used:

- **Classes:** upper CamelCase e.g. Agent, ContextTree, CTNode.
- **Types:** lower case followed by "_t" e.g. action_t, reward_t.
- **Functions:** lower camelCase e.g. mainLoop, genRandomSymbols, depth.
- **Variables:** lower case, with words separated by underscores e.g. m_history, terminate_lifetime, lifetime. Variables belonging to a class should begin with "m_" e.g. m_history, m_time_cycles.

4.4.1 Documentation

You should add sufficient documentation to make the code easy to understand and follow. This includes both function-level and within-function comments. You may adjust any existing comments in the code as necessary. Simple functions may be documented sparsely, if at all. The documentation should aim to make the code as independent from the paper as possible.

4.5 Configuration file

The agent's and environment's configurable options are specified by an ascii configuration file. Each line of the file corresponds to a single option/value pair of the form "option = value". The configuration parser ignores any whitespace in the file. An example configuration file will be provided.

You may add any additional options you require. For example, to set a logging level or to specify how the environment works. When specifying options for the environment we recommend you prefix the option name with the name of the environment.

4.5.1 Standard options

- **ct-depth**: maximum depth of the context tree used for prediction.
- **reward-bits**: how many bits are used to encode the reward.
- **observation-bits**: how many bits are used to encode the observation.
- **cycle-length-ms**: milliseconds after receiving a percept to choose an action.
- **agent-horizon**: the number of percept/action pairs to look forward.
- **agent-actions**: the number of distinct actions an agent can perform.
- **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.

4.6 How to run

To compile and run the code on Linux, extract the source files into a folder `~/mc-aixi-ctw` and execute in a terminal:

```
cd~/mc-aixi-ctw
g++ *.cpp *.hpp -o aixi
./aixi coinflips.conf logfile
```

The two files logfile.log and logfile.csv will contain data on the agent's actions and performance.

5 Submission, expected outcomes, collaboration and evaluation

The submission of each group should include the following:

- C/C++ source code of the MC-AIXI agent, and the selected/assigned specific examples. Your code should contain all dependencies needed and explain in a straightforward way how it should be compiled and executed.
- A report in PDF format with associated LaTeX files. It should include a description of the MC-AIXI-CTW implementation, a brief "user manual", and a detailed analysis of simulation results. The analysis should also include a description of the experimental setup, explanations of the selected parameters, and graphics showing the agent's learning progress (e.g. average reward per cycle).
- All of these files should be zipped into a single file. The report and source code should occupy different subfolders of the archive.

A good submission should have the following features:

- Correct, efficient, and general implementation of the algorithms in the core paper.
- Concise and readable code.
- Clear and concise documentation/commenting explaining the working of the code.

- Lightweight C/C++ code using no external libraries (apart from the standard library). Extraneous use of C++ features should be avoided.
- The report should be clear, succinct and complete. Any simulation provided should include detailed description of experimental setup; selected parameters of algorithms and examples; and concise interpretations of obtained simulation results.
- In the end, try to answer the following two questions. Choose two of the more simple domains, d_1 and d_2 (e.g. 1d Maze and Tiger).
  1. Train AIXI on d_1 and then continue with d_2 (without resetting the CTW and UCT). Is AIXI performing better or worse in learning d_2, after having been biased towards d_1, compared to training on d_2 from scratch?
  2. Now go back to d_1 and train AIXI on d_1 again (without resetting the CTW and UCT). Does AIXI remember d_1 and how fast does it recover it?
  3. If you are ambitious, try to repeat this experiment with other pairs or more domains.
- A platform-independent implementation. That is, one that can be successfully compiled and run with Microsoft Visual C++ 2010 Express Edition on Windows and GNU g++ 4.3 on Linux.
- Meeting the deadline.

Collaborations among students should be based on the following rules:

- Students within each group can organize themselves and split work as they wish.
- Honours students should take a leading role, ensuring smooth and timely progress and delivery, overall coordination, make prudent decisions in case no consensus can be achieved in certain matters, stay in close contact with the lab director.
- Communication between different groups should be kept to a minimum. No sharing or inspection of code or documentation or other written material is allowed.
- Team work (code-sharing, code-review, support, communication, collaboration, ...) within a group is strongly encouraged, and not limited in any way.
- Students are referred to
  http://cs.anu.edu.au/student/StudentHandbook_DCS_2008.pdf
  for general information about collaboration vs. misconduct
  (Sections 6.4 and 6.5).

The overall group outcome will be evaluated against the above expectations, with some differentiation among group members where appropriate.

6 Programming tools and references

Programming tools.

- Microsoft Visual C++ 2010 Express Edition for those working with Windows (http://www.microsoft.com/express/downloads/).
- g++ Version 4.3 for those working with Linux (http://www.gnu.org/software/gcc/).
- Each group should think about how they are going to collaborate and share information during the group project. We recommend using some tool like SourceForge (http://sourceforge.net/).

Recommended programming books for reference.

1. B. W. Kernighan and D. M. Ritchie, ***C Programming Language (2nd Edition)***, Prentice Hall, 1988. This is a very well written tutorial, and can also serve as a great reference.
2. H. Deitel and P. Deitel, ***C++ How to Program (5th Edition)***, Prentice Hall, 2005. This excellent book broadly and coherently presents concepts and techniques of C++ programming. It also offers great software development skills.

Further inquiries. Tom Everitt will be able to provide help for both the theoretical and coding parts of the project. To minimize email traffic and question repetitions, students should post inquiries in Wattle for public communication if they are of general interest. Otherwise, questions concerning the algorithm and implementation details are best directed to Tom Everitt. We will try to respond to inquiries as quickly as possible, however you should try to ask well before the due date. Contact details are given in the following table.

Name	Email	Phone
Tom Everitt	tom.everitt@anu.edu.au	02 6125 59171

References

[CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley & Son, 1991.

[FMRW09] V. F. Farias, C. C. Moallemi, B. V. Roy, and T. Weissman. Universal reinforcement learning. *IEEE Transactions on Information Theory*, 2009.

[HSHB05] B. Hoehn, F. Southey, R. C. Holte, and V. Bulitko. Effective short-term opponent exploitation in simplified poker. In *AAAI*, pages 783–788, 2005.

[Hut05] M. Hutter. *Universal Articial Intelligence: Sequential Decisions based on Algorithmic Probability.* Springer, Berlin, 2005.

[KS06] L. Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *The* 17^{th} *European Conference on Machine Learning*, pages 99–134, 2006.

[Kuh50] H. W. Kuhn. A simplified two-persion poker. In *Contributions to the Theory of Games*, pages 97–103, 1950.

[Leg08] S. Legg. *Machine Super Intelligence.* PhD thesis, IDSIA, Lugano, Switzerland, 2008.

[RN10] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, 3^{rd} edition, 2013.

[SB98] R. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, 1998.

[V+11] J. Veness, K. S. Ng, M. Hutter, W. Uther, and D. Silver. A Monte Carlo AIXI approximation. *Journal of Artificial Intelligence Research*, 40:95–142, 2011.

[WST95] F. M. J. Wilems, Y. M. Shtarkov, and T. J. Tjalkens. The context tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, 41:653–644, 1995.

[WST97] F. M. J. Wilems, Y. M. Shtarkov, and T. J. Tjalkens. Reflections on "the context tree weighting method: Basic properties". *Newsletter of the IEEE Transactions on Information Theory*, 1997.