# A Java Platform for Reinforcement Learning Experiments

**Francesco De Comité**

*Laboratoire d'Informatique Fondamentale de Lille (CNRS - UNSA)*
*Bat m3 info*
*Cité Scientifique*
*F-59655 Villeneuve d'Ascq Cedex*
*decomite@lifl.fr*

ABSTRACT. *This paper describes some families of Java classes implementing the main reinforcement learning algorithms, and several problems on which one can apply them. This Java implementation enables the user to easily conduct experiments, in order to test either new algorithms or new problems. The genericity of the class hierarchy allows one to add either new algorithms, new versions or variants of known algorithms, or new problems, new codings for existing problems. Having written this platform in Java makes it easy to run on any computer, as it allows one to interface it with existing machine learning software like Weka.*

RÉSUMÉ. *Nous présentons une implémentation en Java des principaux algorithmes d'apprentissage par renforcement, et de plusieurs problèmes sur lesquels appliquer ces algorithmes. Ces familles de classes Java permettent de conduire aisément des expériences, que ce soit dans le but d'étudier un algorithme ou bien un problème donné. La conception générique de cette bibliothèque permet d'ajouter facilement soit de nouveaux algorithmes, soit de nouveaux problèmes (ou des codages différents de problèmes). L'écriture en Java permet une portabilité idéale, et rend possible un interfaçage avec d'autres suites logicielles d'apprentissage, comme Weka, par exemple.*

KEYWORDS: *reinforcement learning, java*

MOTS-CLÉS : *apprentissage par renforcement, java*

PDMIA 2005, pages 1 à 8

## 1. Introduction

Besides its strong theoretical foundations, reinforcement learning gives rise to a lot of different effective algorithms, each of them conducted by several user defineable parameters. On the other hand, while describing their experiments, authors are introducing new problems to apply those algorithms on.

The interested reader has often the will to try to reproduce experiments, or to see what are the relative importance of the algorithm's settings. This interested reader could also wish to see what happens when applying a certain (novel) algorithm to another problem.

He would also like to look at the relative efficiency of different codings for the same problem, and perhaps would like to save what has been learned, in order to use it in another context, or to study more precisely what has been actually learned, even perhaps use machine learning techniques to generalize the learned policy.

In order to be able to run or reproduce experiments with various algorithms on different problems, we decided to write a program as generic as possible, which will allow the user to simply pass from one algorithm to another one, or from one problem to another one.

Studying the way reinforcement learning is described, it appears that a lot of aspects of this theory could be factorized:

– Problems are dealing with the abstract notions of agents, states, actions and rewards: an agent, in one particular state, can apply some specific actions, puting it into another states, where it can perceive a reward.

– Algorithms, knowing the current state, and the current available actions, have to choose the next action. Knowing the preceeding state, the new state, the action they have chosen, and the reward, they can store them and update their beliefs.

Nowhere in this schematic description of the behavior of reinforcement learning, was the name of the game, nor the exact algorithm useful.

Taking into account those considerations, we planned to implement a generic reinforcement learning platform, where the genericity will concern problems, agents, and algorithms.

The Java language was then chosen for several reasons: it is powerful enough for our needs for genericity, it is relatively platform independant, the data structures it defined let us concentrate on the core of algorithms. It also allows us to interface our platform with existing strong machine learning implementations, like Weka (Witten *et al.,* 2000).

At this stage of the platform's development, code is defined for fully observable discrete Markov decision processes, and continuous problems, like *Mountain car* (see below) are discretized. Future developments might raise those limitations.

The rest of this document is organized as follows: section 2 describes the architecture of the system, its organisation in three mains domains and enumerates implemented problems and algorithms. Section 3 illustrates the use of the platform, by demonstrating the power of its genericity. Section 4 briefly discusses the link with another machine learning set of programs. Finally we will conclude and give some perspectives.

## 2. General organisation

The architecture of the system consists of three main packages, plus one for each problem to code, and some little technical tool packages.

### 2.1. *Main packages*

The three main packages are:

**agents** : agents can *act* and *learn*.

**environnement** : abstract classes to define states, actions, and univers's rules.

**algorithmes** : contains the learning algorithms.

#### 2.1.1. *The agents package*

An Agent instance mainly consists of a class that:

– Knows it current state, and the state it was in just before.

– Asks its *guru* for the next action to perform.

– Once the action is performed, and the reward is perceived, it gives back this information to the *guru*.

Auxiliary methods exist to save and load agents, to read internal information stored in the agent, for monitoring and debugging.

Agents are either acting alone, for solving one-player puzzles, or competing against another agent. In this last case, they are also informed of their opponent's last action. While not still implemented, we could easily implements non concurrent or cooperative agents...

#### 2.1.2. *The environnement package*

This package defines the abstract notions of states and actions. They must be instantiated for each problem one wants to code.

Coding an action mainly consists in describing it in such a way that one can discriminate two different actions. The class **Action** is also the place when one must define how to code an Action for different outputs (c4.5, Arff, neural network...)

The class **State** is similar to the class Action, as it mainly contains the equals method, and different output codings.

The class **Contraintes** is where one has to define how to modify the current state according to the current action, how to list the actions available when in a given state. In other words, this class contains the game's rule.

#### 2.1.3. *The algorithmes package*

The abstract class implementing an algorithm is called Selectionneur. A Selectionneur has two main methods:

– Given a state and a list of available actions, returns what it thinks to be the best action to perform.

– Once the action is done, and the reward known, it uses this information to learn (whatever learning means for it...)

In this package are defined all the standard reinforcement learning algorithms:

– Standard Q-Learning algorithm (Sutton *et al.,* 1998) p149, memorizing $Q(s, a)$.

– Peng's $Q(\lambda)$ algorithm as described in (Peng *et al.,* 1994).

– Watkin's $Q(\lambda)$ algorithm as described in (Watkins, 1989), and detailed in (Sutton *et al.,* 1998).

Variants of those algorithms are also provided, which replace the $Q(s, a)$ memorization by different kinds of neural networks, in case of big states spaces. We also defined random choice algorithms, to be used as *sparring partners*. Note also that this package can handle any kind of learning algorithm, and is not restricted to reinforcement learning ones: the general framework just asks an algorithm to be able to choose an action, and enable it to learn, using the reward and/or the state where the last action led the agent in.

### 2.2. *Implemented problems*

We already implemented the following puzzles or games:

**Random Walk** : a toy example from (Sutton *et al.,* 1998), page 139 : served mainly to compare Q-Learning and Value Iteration.

**Gambler** : described in (Sutton *et al.,* 1998), page 101.

**Jack's Car Rental** : described in (Sutton *et al.,* 1998), page 98.

**Mountain-Car Task** : described in (Sutton *et al.,* 1998), page 214.

**Mazes** : a classic definition of mazes, with walls and treasure(s).

**Cannibals and Missionaries** : an old easy to solve puzzle.

**Jenga** : a two players game with a winning strategy ( For an in-depth analysis, see (Zwick, 2002)).

**Tic-tac-Toe** : a standard benchmark for learning algorithms.

**Memory** : (Zwick *et al.,* 1993) shows that there exists a winning strategy for one of the players.

**Othello** : while the game is implemented properly, no agent was able to improve its strategy in this game...

Except for Othello, each puzzle or game is solved by our learning algorithms. In the case of (Sutton *et al.,* 1998) examples, we were able to reproduce exactly all their experiments and graphs.

## 3. Genericity

Instead of describing in depth all classes and methods, it might be more interesting to illustrate on a short example how the same program frame can experiment different games and different algorithms. Figure 1 shows a program which asks an agent to learn to win at a two-players game.

The program is meant to train an agent using Q-learning to learn Tic-Tac-Toe. By uncommenting another line preceeded with an (a), one can ask the agent to learn another game! Similarily, choosing to uncomment another line preceeded with (b), one can use another algorithm. Nothing else is needed to change either the problem or the algorithm.

## 4. Interfacing with Weka

When conducting experiments in machine learning, people not only want to see their algorithm learn, but also wish to understand *what* has been learned. In other words, one needs to generalize the state-value function. An natural way to achieve this generalization can be to use machine learning algorithms like decision trees, regression, set of rules...

As a lot of those algorithms are available in Weka, and hence written in Java, we started recently to enhance our platform, by enabling the call to Weka classifiers. By tuning the coding of states,actions and values, we were able to obtain non-trivial generalizations of state/action value function, leading to a more compact and understandable representation of this function.

```
public class Example{
  public static void main(String argv[]){
      // The Game
 (a)     TicTacToeBoard p=new TicTacToeBoard();
 (a)     // JengaTower p=new JengaTower();
 (a)     //PlateauOthello p=new PlateauOthello();
 (a)     //MemoryBoard p=new MemoryBoard();

      /* The player we want to train */
(b)     Selectionneur sql=new SelectionneurQL();
(b)     // Selectionneur sql=new SelectionneurWatkins(0.9);
(b)     //Selectionneur sql=new SelectionneurPeng(0.9);
(b)     //Selectionneur sql =new SelectionneurNN();

      /* The sparring partner*/
      Selectionneur sal=new SelectionneurAleatoire();

      /* Defining both players */
      Agent2Joueurs j1=new Agent2Joueurs(p,sql);
      Agent2Joueurs j2=new Agent2Joueurs(p,sal);

      Partie2Joueurs arbitre=new Partie2Joueurs(j1,j2);

      /* Store the performance :
         resu[0] : number of won games
         resu[1] : number of ties
         resu[2] ; number of lost games
      */
      int resu[]=new int[3];
      /* Start the games */
      for(int i=1;i<1000000;i++)  resu[arbitre.episode()+1]++;
  }
}
```

**Figure 1.** *Genericity illustrated.*

## 5. Conclusion and perspectives

We believe to have defined a coherent framework for reinforcement learning experiments, letting the user concentrate himself on the coding of his problem and the analysis and interpretation of the algorithm's results.

A lot of work is still to be done, for example to code new problems and new algorithms, or interface with other programs. But those tasks can be planned incrementally, as one will just have to add bricks to the structure, without the need to redefine everything from scratch.

The source code of the classes, together with examples and a tiny tutorial are available at the following url : `www.lifl.fr/~decomite`.

## 6. References

Peng J., Williams R. J., « Incremental Multi-Step Q-Learning », *International Conference on Machine Learning*, p. 226-232, 1994.

Sutton R. S., Barto A. G., *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998. A Bradford Book.

Watkins C. J., Learning from delayed rewards, PhD thesis, Cambridge university, 1989.

Witten H., Frank E., *Data Mining: Practical machine learning tools with Java implementations*, Morgan Kaufmann, 2000.

Software available at `http://www.cs.waikato.ac.nz/ml/weka/`

Zwick U., « Jenga », *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, p. 243-246, 2002.

Zwick U., Paterson M. S., « The memory game », *Theor. Comput. Sci.*, vol. 110, n° 1, p. 169-196, 1993.