STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVIII**, Number 2, 2003

A NEW INTERFACE FOR REINFORCEMENT LEARNING SOFTWARE

GABRIELA ŞERBAN

Abstract. The field of Reinforcement Learning, a sub-field of machine learning, represents an important direction for research in Artificial Intelligence, the way for improving an agent's behavior, given a certain feed-back about its performance. In this paper we propose an original interface for programming reinforcement learning simulations in known environments. Using this interface, there are possible simulations both for reinforcement learning based on the states' utilities and learning based on actions' values (Q-learning).
Keywords: Reinforcement Learning, Agents.

1. Introduction

The interface is realized in JDK 1.4, and is meant to facilitate to develop software for reinforcement learning in known environments.

There are three basic objects:agents, environments and simulations.

The agent is the learning agent and the environment is the task that it interacts with. The simulation manages the interaction between the agent and the environment, collects data and manages the display, if any.

Generally, the inputs of the agent are perceptions about the environment (in our case states from the environment), the outputs are actions, and the environment offers rewards after interacting with it.

Figure 1 illustrates the interaction between the agent and the environment in a reinforcement learning task.

The reward is a number; the environment, the actions and perceptions are instances of classes derived from the *IEnvironment*, *IAction* and *IState* interfaces respectively. The implementation of actions and perception can be arbitrary as long as they are understood properly by the agent and the environment. It is obvious that the agent and the environment has to be chosen to be compatible with each other in this way.

Received by the editors: December, 10, 2002.

2000 *Mathematics Subject Classification.* 68T05.

1998 *CR Categories and Descriptors.* I.2.6 [**Computing Methodologies**]: Artificial Intelligence – *Learning*.

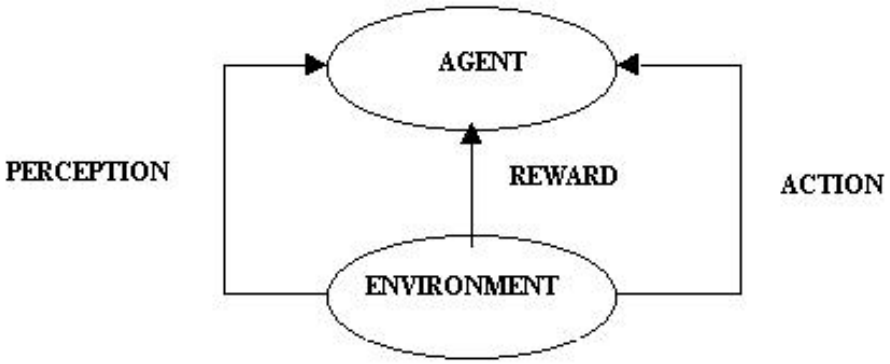

FIGURE 1. The interaction between the agent and the environment

The interaction between the agent and the environment is handled in discrete time. We assume we are working with simulations. In other words there are no real-time constraints enforced by the interface: the environment waits for the agent while the agent is selecting its action and the agent waits for the environment while the environment is computing its next state.

We assume that the agent's environment is a finite Markov Decision Process.

For using the interface, the user has to define the specialized object classes *HisState*, *HisEnvironment* and *HisAgent*, by creating instances for each. The agent and the environment are then passed to a simulation object (*CSimulation*), that initializes and interconnects them. Then, CSimulation::init() will initialize and execute the simulation.

If the agent learns the states' utilities, it has to be derived from the *AgentUtility* class, otherwise, if it learns the actions' values (Q-learning) it has to be derived from the *AgentQValues* class.

In the followings we present a prototypical example for a concrete agent.

(1) First, the user defines the class corresponding to a concrete state of the environment.
public class *HisState* **implements** *IState*
{...}
(2) Second, the user defines the class corresponding to the concrete environment in which the agent acts.
public class *HisEnvironment* **implements** *IEnvironment*
{...}
(3) The user defines the class corresponding to the concrete agent.
public class *HisAgent* **implements** *AgentUtility*
{

```
    public void actions(){...}
}
```

if the agent learns the states' utilities, respectively

```
public class HisAgent implements AgentQValues
{
    public void actions(){...}
}
```

if the agent learns the actions' values.

Using the method *actions()*, the agent perceives the actions that can be executed. In our approach, the agent's actions are numbered (starting from 1).

(4) Finally, the user defines the application class which initializes the simulation of learning process for the concrete agent in the concrete environment.

```
class Application {
    public static void main(String args[]){
            IEnvironment m=new HisEnvironment();
            RLAgent ag=new HisAgent();
//the agent perceives its actions
            ag.actions();
//instantiation for the object that realizes the simulation
            CSimulation s=new CSimulation(ag, m);
//on initialize the simulation - α, γ, ϵ, number of episodes
            s.init(0.01, 0.3, 0.1, 10);
//on display the policy
            s.policy(System.out);
}
```

We have to mention that the learning algorithms used for implementing the agents' behavior are the URU algorithm [4] for learning the states' utilities (values), respectively the SARSA algorithm [1] for Q-learning.

2. The Design of the Interface

The classes used for realizing the interface are the following:

- *IList* **INTERFACE**
  Defines the structure of a list of objects, having operations for managing the list: adding an element on a given position, removing an element from a given position, returning the number of elements from the list, returning an element from a given position.
- *IState* **INTERFACE**
  Defines the structure of a state from the environment (could have an explicit representation or an implicit one if the environment is unknown

and the agent has to retain a model of the environment). The methods of this class are for: returning a String with the member data of the class, testing the equality of two states.

- *IAction* **INTERFACE**
  Defines the structure of an action that the agent could execute. The methods of this class are for: returning a String with the member data of the class, testing the equality of two states.
- *Element* **ABSTRACT CLASS**
  Defines a generic element represented as a triplet (*IState*, *IAction*, *value*) needed for realizing the learning. Depending on the learning agent (learns the states' or the actions' values), *value* will represent the utility of the state *IState*, respectively the Q-value of the pair (*IState*, *IAction*).
- *Utility* **SUBCLASS of** *Element*
  Defines the class corresponding to an element (defined above) used in learning the states' utilities.
- *QValues* **SUBCLASS of** *Element*
  Defines the class corresponding to an element (defined above) used in learning the actions' values.

AGENT

The agent is the entity that interacts with the environment, receives perceptions (states) from it and selects actions. The agent learns by reinforcement and could have or not a model of the environment.

- *RLAgent* **ABSTRACT CLASS**
  Is the basic class for all the agents. The specific agents will be instances of subclasses derived from *RLAgent*. The methods of this class are:
  (1) **void** *actions*() **ABSTRACT METHOD**
  This function is given by the user for the specialized agent class and defines the list of actions that the agent could execute.
  (2) **Element** *choose*(**Element** e**, Integer** r**, double** *epsilon*, **IEnvironment** m) **ABSTRACT METHOD**
  This function is used in learning and allows the choice of the next element (having the type *QValues* or *Utility* , depending of the chosen learning type) to which the agent moves, starting from the current element e, in the environment m and choosing as a selection mechanism the ϵ-*Greedy* selection (*epsilon* is given as parameter). After this choice, the parameter r will contain the reward obtained by the agent.
  This method will have specific definition according to the learning method (Q-value, utility).

(3) **QValues** *next*(**IState** *s*, **IEnvironment** *m*) **ABSTRACT METHOD**
This function gives the agent's policy for moving after learning. If the object having the type *QValues* returned by the method contains the state *snext* and the action *a*, it means that the agent's policy is the following: from the state *s*, the agent will choose the action *a* and will move to the state *surm*.
This function has specific definition according to the agent's type, too.

(4) **Element** *initial*(**IState** *s*) **ABSTRACT METHOD**
The state *s* being the initial state of the environment, the method returns the initial element (*QValues* or *Utility*, corresponding to the learning's type) which will starts the learning. This method has specific definition according to the agent's type.

(5) **void** *learning*(**double** *alpha*, **double** *gamma*, **double** *epsilon*, **int** *episodes*, **IEnvironment** *m*)
Is the basic method which implements the learning algorithm of the agent in the environment *m*. There are given: the learning rate (*alpha*), the reward factor (*gamma*), the value for *epsilon* for the ϵ-Greedy selection mechanism, the number of training episodes (*episodes*).
This method is not abstract, is concretely defined in the class *RLAgent* (indifferent what is the learning's type, the learning method is the same).

- *AgentUtility* **ABSTRACT CLASS**
Is a subclass of the class *RLAgent*, being the entity which defines the behavior of an agent that learns by reinforcement based on the states' utilities. This class specializes the methods (2), (3) and (4) (defined in the superclass) according to learning based on states' utilities.
The method *actions()* is not defined in this class (that is why the class is abstract), but will be defined in the class corresponding to the specialized agent created by the user (and who can be an instance of a class derived from *AgentUtility*).
- *AgentQValues* **ABSTRACT CLASS**
Is a subclass of the class *RLAgent*, being the entity which defines the behavior of an agent that learns by reinforcement based on the actions' values. This class specializes the methods (2), (3) and (4) (defined in the superclass) according to the Q-learning method.
The method *actions()* is not defined in this class (that is why the class is abstract), but will be defined in the class corresponding to the specialized agent created by the user (and who can be an instance of a class derived from *AgentQValues*).

ENVIRONMENT

The environment basically defines the problem to solve. It determines the dynamic of the environment, the rewards and controls, the ending of the training process. In our approach, the environment will have an implicit representation as a space of sates (*IState*).

- *IEnvironment* **INTERFACE**

  Is the basic class for all environments. The specific environments will be instances of subclasses derived from *IEnvironment*. The environment classes defined by the user (subclasses of *IEnvironment*) will give specialized definitions for the following functions:

  (1) **boolean** *isValid*(**IState** *s*) **ABSTRACT METHOD**
  Is the function that verifies if a state s (represented explicitly or implicitly) is valid in its environment.

  (2) **IState** *initial*() **ABSTRACT METHOD**
  Is the method that returns the initial state of the environment (the state that will be used for initializing the learning).

  (3) **boolean** *isFinal*(**IState** *s*) **ABSTRACT METHOD**
  Is the method that returns the final state of the environment (the state that will be used by the agent for ending the training process). The final state could be given explicitly, or given implicitly by certain conditions.

  (4) **IState** *next*(**IState** *s*, **IAction** *a*, **Integer** *r*) **ABSTRACT METHOD**
  Is the main method of the interface *IEnvironment*. This method will be called by an instance of the class that simulates the learning (*CSimulation*), at each step of the simulation.
  This function determines the environment to make a transition from the current state s to a next state *surm*, after executing the specific action a. The state *surm* will be returned, the function supplying in the same time the reward r obtained after the transition.
  In the case that the action a could not be applied in the state s, the method returns *null*.

  (5) **double** *value*(**IState** *s*) **ABSTRACT METHOD**
  Is the method that gives the value of a state in the environment (the initial utility of the state and the initial Q-values). We considered that this value depends only on the current state, not on the selected action (in the case of Q-learning).
  This method will be used for initializing the learning process.

SIMULATION

- *CSimulation* **INTERFACE**

Is the basic object of the interface, that manages the interaction between the agent and the environment. Defines the *heart* of the interface, the uniform usage that all agents and environments are meant to conform to.

An instance of the simulation class is associated with an instance of an agent and an environment at the creation moment. This is made in the constructor of the class *CSimulation*. The methods of this class are for:

(1) **void** *init*(**double** *alpha*, **double** *gamma*, **double** *epsilon*, **int** *episodes*)
Is the method that initializes the simulation with the given parameters (is the function that starts the learning process of the agent).

(2) **void** *policy*(**PrintStream** *ps*)
Is the method that gives the moving policy for the agent, obtained at the end of the simulation (after the training process).
The class *CSimulation* keeps references to the instances of the agent and the environment. This facilitates cross-references of instances in case it is need.

```
public class CSimulation
{
    private RLAgent a; //reference to the agent's instance
    private IEnvironment m; //reference to the environment's instance
    ...
}
```

3. Experiment

In this section we illustrate the use of the interface on a concrete example. Let us consider the problem of a path-finding robot, whose goal is to learn (by reinforcement) to come out from a maze (moving from an initial to a final state).

We assume that:

- the maze has a rectangular form; in some positions there are obstacles; the agent starts in a given state and tries to reach a final (goal) state, avoiding the obstacles;
- from a certain position on the maze the agent could move in four directions: north, south, east, west (there are four possible actions);

For example, let us consider the environment from Figure 2. The state marked with 1 represents the initial state of the agent, the state marked with 2 represents the final state and the states filled with black contain obstacles (which the agent should avoid).

For using the interface, we defined the specialized classes for which we executed the simulation (*HisState*, *HisEnvironment* and *HisAgent*) (For lack of space

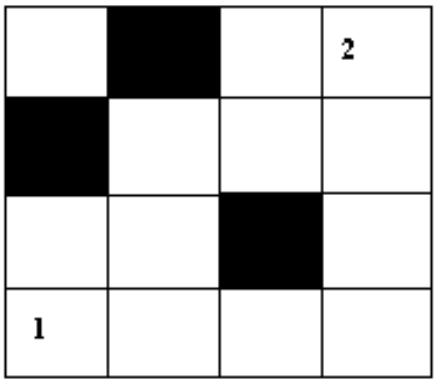

FIGURE 2. The agent's environment

the complete description of the classes may be found at the following URL: http://www.cs.ubbcluj.ro/∼gabis/agent.zip).

The moving policy learned by the agent after the training and reported by the *CSimulation* object is the same in both learning cases (states' utilities or Q-values). The learned path is: (4,1)-(3,1)-(2,2)-(2,3)-(1,3)-(1,4), respectively the sequence of actions that the agent should execute is: *North*, *East*, *North*, *East*, *North*, *East*.

4. Further Work

A further work for generalizing the interface would be the study of the case in which the agent's environment is a Hidden Markov Model [3].

References

[1] Sutton, R., Barto, A., G., Reinforcement learning, The MIT Press, Cambridge, England, 1998

[2] Serban, G., A Reinforcement Learning Intelligent Agent, Studia Universitatis "Babes-Bolyai", Informatica, XLVI (2), 2001, pp. 9–18

[3] Serban, G., Training Hidden Markov Models – a Method for Training Intelligent Agents, Proceedings of the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems, Krakow, Poland, 2001, pp. 267–276

[4] Serban, G., A New Reinforcement Learning Algorithm, Studia Universitatis "Babes-Bolyai", Informatica, XLVIII (1), 2003, pp. 3–14

"BABEŞ-BOLYAI" UNIVERSITY, CLUJ-NAPOCA, ROMANIA
E-mail address: gabis@cs.ubbcluj.ro