PIQLE
A Platform for Implementation of Q-Learning Experiments

Francesco De Comité

December 15, 2006

Chapter 1

Introduction

PIQLE was primarily designed for implementing and testing the algorithms and problems described in Sutton's and Barto's *reinforcement Learning, An Introduction* ([SB98]).
As it is already visible into this book, reinforcement learning algorithms are relatively generic, as based on a quite abstract notion of state, action, reward.
We tried to respect the genericity and transcript it in Java, a language very well suited for this.
Defining and designing separately the notions of agent, algorithms, environment, and how those objects communicate with each other made it possible to obtain a very general platform for reinforcement learning experiments , easy to understand, easy to use, and also easy to extend, by mean of adding new algorithms, new problems, new agents...
A very little time ago , it also appeared that Piqle can be connected quite simply with other existing frameworks, like RL-Framework developed at the university of Alberta, or the problem simulators at McGill University.
This document will try to be a reference for the most recent version of Piqle (beginning with version 2), describing the general architecture, the details of each package and classes, as well as some examples of how to correctly design problems to be solved.
Any feedback on any level will be welcome : program bugs, documentation typos,awful english, need for precisions...
On the other hand, any developer wanting to add his/her skills to the development of Piqle is warmly welcome.

Chapter 2

Quick Start

All the files needed to run Piqle on some examples are available at sourceforge :

`http://sourceforge.net/projects/piqle`

You can choose to download the latest release (download piqle), or the latest source revision in the subversion repository.

2.1 Downloading the latest release

- Download piqle-core, which contains all the definitions you need to launch an experiment.
- Uncompress piqleMA.tgz(`tar -xvzf piqleMA.tgz` under Linux) : you will obtain two directories, `sources` and `doc`, the last one containing the Piqle javadoc.
- At this stage, you can already launch the maabac example :
  - Create a new directory `classes` at the same level as `doc` and `sources`
  - Compile :

```
javac -d classes -sourcepath sources sources/TestMaabac.java
```

  - Run the program :

```
java -cp classes TestMaabac
```

- If you didn't met any problem at this stage, you can come back to sourceforge and load the examples package : download piqle-examples and uncompress it *into* the `sources` directory.
- You then obtain a new package for each problem, and test programs in the `sources` default package.
- Compile one of those test programs and run it (see above).
- Try editing the test program, to see how easy it is to modify the learning algorithm, and how a given algorithm reacts to parameters tuning.

- Compare two test programs to see how few are the differences between two experiments designs : very often, changing the problem you want to solve can be done by only changing one line into the test program code.

2.2 Downloading the latest source

The Subversion repository contains the latest source version, but may not be fully usable, if we are in the process of modifying it, for example. This latest version of `piqle-core` and `piqle-examples` are in the `Branches` part of the repository, while the documentation is is the `Trunk` part of the repository (named `piqleV2doc`).
Download all the files in piqle-core, and the package you want from piqle-examples, together with a test program (names of test programs are related to those of the packages), and put them into the same directory organization as described in the case when you download the latest releases. Next steps are similar as those described in section 2.1.

2.3 The documentation

There are two types of documentations with Piqle :

- The javadoc : even if it is given with the downloadable release, it is always a good idea to rebuild it once you uncompressed the release : thus you will be sure to always have an up-to-date information.

- This document : the pdf version is always corresponding to the latest release, while the files from the subversion repository concern the version of Piqle under construction, and might be incomplete. The files available through subversion can be compiled using latex.

Chapter 3

General Organisation

Piqle is organized around three main entities :

Environments Representing the universe of the problem, i.e. the physics (or the rules for a game), the differents methods to describe states and actions.

Algorithms Software elements able to choose the next action to perform. Within the framework, algorithms are called `Selectors`.Interesting algorithms are those which are able to learn.

Agents They make the interface (not in the Java sense !) between Environments and Algorithms.

On top of those three main features, the `referees` are scheduling the succession of phases, as illustrating here after :

- The Environment tells the Agent in which state it is at this time, and provides the list of possible actions (figure 3.1).
- The Agent communicates this information to its Algorithm (figure 3.2).
- The Algorithm chooses the action to perform, and gives its answer back to the Agent (figure 3.3).
- The Agent tells the Environment what action it wants to perform (figure 3.4).
- The Environment compute the new state, the reward, and the new list of possible actions, and send those information to the Agent (figure 3.4).
- The Agent transmits everything to the Algorithm, which can now learn from experience, as it received a reward for its former choice,choose the next action to perform, and the cycle begins again (figure 3.5).

At this stage, some remarks will make the organization of Piqle quite easy to understand :

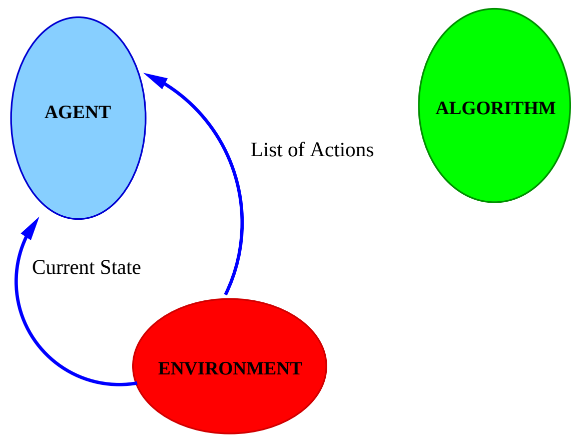

Figure 3.1: Phase 1 : the agent learns its state

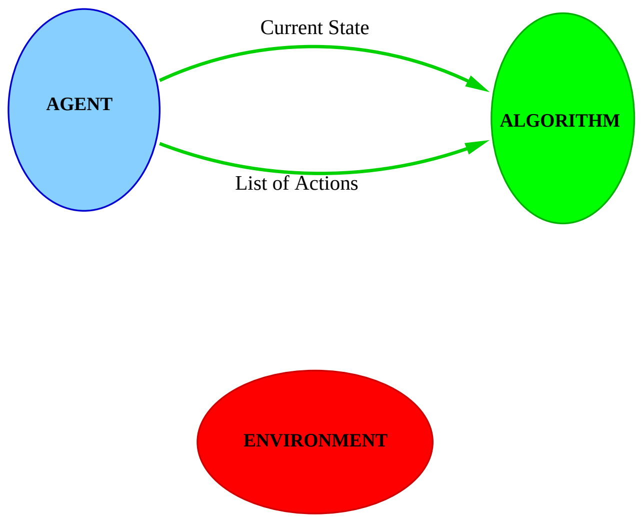

Figure 3.2: Phase 2 : the algorithm knows its current state

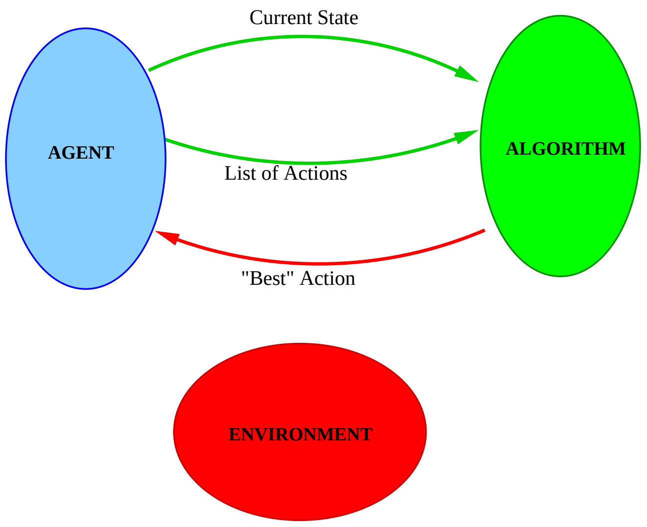

Figure 3.3: Phase 3 : the algorithm chooses the action

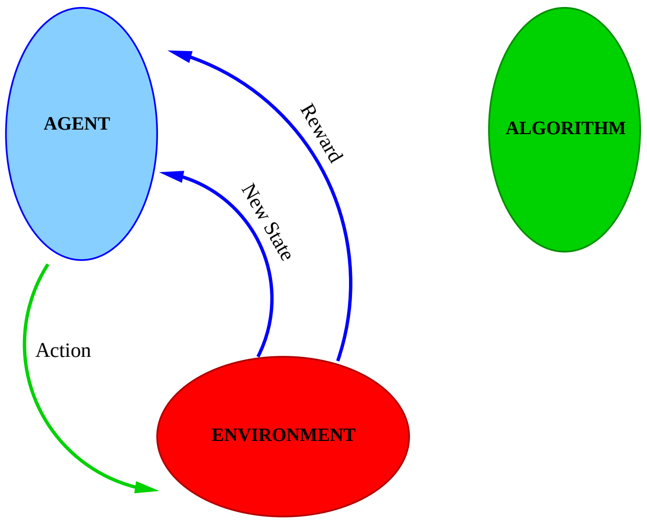

Figure 3.4: Phase 4-5 : the environment computes reward and new state

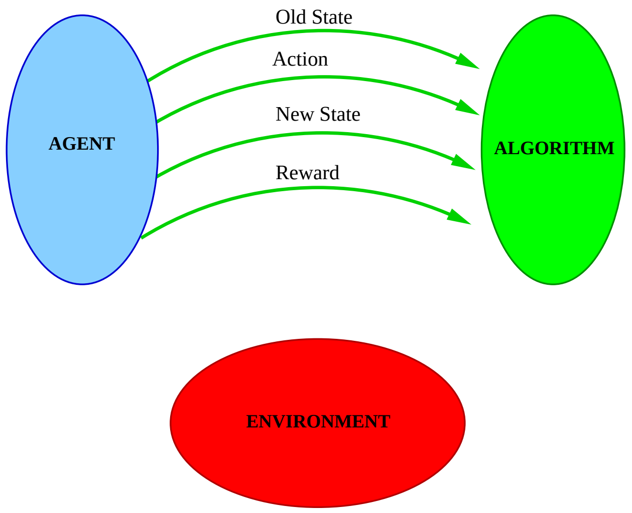

Figure 3.5: Phase 6 : the algorithm can learn

- Agents are just interfaces between Environments and Algorithms : they just receive and transmit information : they do not need to know exactly in which universe they are *moving*.
- Algorithms are just asked to choose an action into a given list of actions, and receive a reward for their last choice. The thing they have to remember (or learn) is roughly speaking : *What was (or what will be) my next reward if I choose this action ?* . This reasoning scheme can be made independent of the Environment, as the Algorithm mainly has to store (and retrieve) its past experience as triples (state,action,reward).
- Environments can be described as generic and abstract entities.
- Hence the only place where one will have to really describe the problem one wants to solve is the instantiation of the generic classes of the `environment` package.

Those remarks directly lead to the Java organization of Piqle :

- Three main packages independent from the problem to solve : `agents`, `referees`, `algorithms`.
- One generic `environment` package for the common behaviour of any depictable environment.
- For each problem, a new package to instantiate the `environment` package.
- Around those main packages, some utilities will be gathered in secondary packages. Some future refactoring might move those packages to more accurate places, making them subpackages of the main ones.

Having outlined this package's architecture, we are now ready to describe more precisely each of them.

Chapter 4

Agents

Agents have two closely related roles :

- Allowing the environment and the algorithm to communicate with each other.
- Communicate itself with the referee, in order to schedule each episode.

A general overview of the hierarchy of package `agents` is shown in figure 4.1

4.1 The basic interface

Interface `IAgent` is described in figure 4.2

- As the agent provides an interface between an algorithm and an environment, it must contain two fields for those two entities. The first two methods : `getAlgorithm` and `getEnvironment` allow access to each of those fields.
- The next two methods, `enableLearning` and `freezeLearning`, are controling the learning behaviour of the agent : one can ask the agent to stop learning at a certain stage, and see whether this agent behaves cleverly enough.
- Method `getLastReward` asks the algorithm (in cascade, through the current state) for the reward corresponding to the last action of the agent. This is used by referees to compute the total reward for an episode.
- Method `act` is the core of the agent behaviour : here the agent will ask its algorithm for the best action to perform, and return the chosen action to the referee (which will then communicate this choice to the environment).
- Method `explainValues` outputs the learned $Q(s,a)$ values : it is up to the experimenter to find out, from those data, what the agent has really learned (maybe not an easy task !).

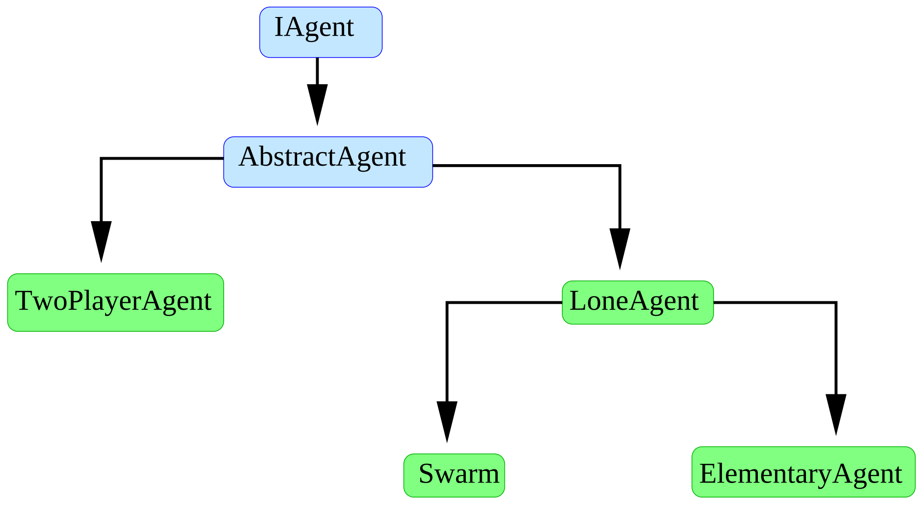

Figure 4.1: The agent's hierarchy

```
public interface IAgent {
public IStrategy getAlgorithm();
public IEnvironment getEnvironment();

public void enableLearning();
public void freezeLearning();

public IState getCurrentState();
public void setInitialState(IState s);

public double getLastReward();

public IAction act();

public void explainValues();

public Dataset extractDataset();

public void saveAgent();
public void saveAgent(String fileName);

public void newEpisode();
}
```

Figure 4.2: What a state must be able to do

- Method `extractDataset` is much like `explainValues` : it outputs $Q(s,a)$ values in a format suitable for analysis by Piqle's neural networks.

- Both methods `saveAgent` allow for saving an agent into a file. Each non-abstract agent classes (`LoneAgent` and `TwoPlayerAgent`) also implement `readAgent` methods. As those methods are static, they can not appear into the definition of an interface.

- In case there are some settings to be done at the beginning of an episode : resetting reward, resetting the algorithm. . . , the interface provides the method `newEpisode`.

4.2 The Abstract Generic Agent

Directly above the `IAgent` interface, the abstract class `AbstractAgent` defines the code of quite all the methods defined in the interface.
Static methods `readAgent` can still not be written here, as we need to know whether we are speaking of a `LoneAgent` or of a `TwoPlayerAgent`.
At an advanced level of code reading, one may remark that the method `choose` is declared as public (while it seems that protected or even private would have made it): this was necessary for the correct implementation of the Octopus Arm problem : as we shall see later (see chapter 13), the implementation of the octopus arm learner is not *pure* Piqle : some concessions had to be made in order for both codes to cooperate.

4.3 The One Player Case

The basic class for one player game agent is `LoneAgent`. It is simply an `AbstractAgent` with the only constraint of having an `IEnvironmentSingle` associated environment. It thus manipulates states of class `AbstractState`.
Note that now, we can write the code for the `readAgent` methods.

4.3.1 Subclasses of `LoneAgent`

Two classes are derived from the `LoneAgent` class : both are related to the multi-agent extension of Piqle. See chapter 12 for a more detailed view of the multi-agent implementation.
Roughly speaking, a multi-agent system is a gathering of limited (in terms of perception and of behaviour) agents : the `ElementaryAgent`s. Those agents are grouped into a `Swarm`, which is also an `IAgent`.
This `Swarm` communicates with the environment in both directions, as any `IAgent` does:

- The `Swarm` receives the description of the current environment's state.

- The `Swarm` dispatches this information to the `ElementaryAgents` it is composed of.
- Each `ElementaryAgent` sees this current state through its limited perception : a `Filter` extracts from the original state's information the only part of it visible for this `ElementaryAgent`.
- Each `ElementaryAgent` chooses the action it will perform.
- The `Swarm` collects all those individual actions into a `ComposedAction`, and send it to the environment.
- The environment uses this `ComposedAction` to compute its next state, and send back to the `Swarm` the reward for this episode's step.
- Finally, the `Swarm` informs all its components of this reward.

One can already remark that those two agents are sharing the tasks usually devoted to an agent : one communicates only (with the environment, and with its subjects), while the other only acts. This means that each of those two subclasses, while still being considered as `LoneAgents`, will only implement a part of the `IAgent` methods.

4.3.1.1 Swarm

A `Swarm` has no associated `ISelector` (algorithm) : each of its components has its own one.
All `ElementaryAgents` being part of a `Swarm` are stored into an `ArrayList` : methods are written to add an `ElementaryAgent` to a `Swarm`, to extract an `ElementaryAgent` from the `Swarm`.
Most of the *important* behaviours of a `Swarm` have the same skeleton :

Setting the initial state : the `Swarm` asks each of its components to set its own initial state as a filtered version of the global state.

Resetting for a new episode : the `Swarm` asks each of its components to reset : thus each `ElementaryAgent` will in turn ask its associated algorithm to perform all the necessary initialisations for a new episode to begin.

Choose and apply an action those two methods are at the basis of the `IAgent` method `act()`.

Choose The `Swarm` asks each `ElementaryAgent` to choose the best action according to its associated algorithm, collects all those elementary actions into a `ComposedAction` : this is method `choose()`.

Apply The `ComposedAction` is passed to the environment, which computes the new state, and returns the reward for this step.The `Swarm` then informs its subjects of the new state, whom then becomes their current state (through *filtering*), and of the reward, whom they use to learn.

4.3.1.2 Elementary Agent

An `ElementaryAgent` :

- Only implements a part of the `IAgent` methods.
- Needs a `Filter` to extract from a *general* state only the information it is allowed to perceive.

The methods of `IAgent` an `ElementaryAgent` must implement are :

setting the current state initially or at each episode's step.

choosing the action to perform by asking its algorithm.

Methods of `IAgent` that are meaningless for an `ElementaryAgent` are :

ApplyAction this is done by the `Swarm` the `ElementaryAgent` belongs to.

readAgent this is the `Swarm`'s job also.

Other methods specific to this class of agent are :

Filter related defining the filter (in the constructor), retrieving it.

Internal reward In addition to the reward that will be communicated by the `Swarm`, one can define an internal reward specific to each `ElementaryAgent`. Imagine for example that each `ElementaryAgent` is a muscle (the `Swarm` could then be considered as a body member, or a body part) : a muscle action could be a contraction, and some contraction could costs more energy than others. The internal reward could then be used while trying to learn an upper level task with the minimum effort from each muscle (in fact, that is how the maabac example is defined).

4.3.1.3 Remarks for the multi-agent case

- Trying to explain how `Swarm` and `ElementaryAgent` are defined, we entered into a relatively precise description of the complete multi-agent Piqle's definition : parts of this chapter will perhaps be repeated into chapter 12.
- As the multi-agent framework was originally defined for one player games, `Swarm` and `ElementaryAgent` are defined as subclasses of `LoneAgent`. There might not be too much work to generalize their definition in order to fit with the two players games case.

Chapter 5

Environment

Inside the `environment` package, there are three main entities :

Environments All that is needed to compute new state and reward, indicating when the game is over and eventually, who won it.

Actions Able to generate any action for a given problem, tools for comparing, copying and coding actions.

States Code, compare, copy states. Auxiliary methods to access the associated environment.

5.1 Environments

5.1.1 The basic definition

All environments must implement the `IEnvironment` interface, shown in figure 5.1. This interface defines the expected behaviour of any instantiated environment.

```
public interface IEnvironment extends Serializable{
public ActionList getActionList(IState s);
public IState successorState(IState s,IAction a);
public double getReward(IState s1,IState s2,IAction a);
public boolean isFinal(IState s);
public int whoWins(IState s);
```

Figure 5.1: The minimal environment interface

That is to say :

- Given a state of the environment, telling what actions are possible.

- Given a state and an action, compute the next state. This computation can be non deterministic or probabilistic.
- Computing a reward from the previous state, the current state, the action taken (as usual in reinforcement learning).
- Indicating whether the game is over or not, and who won : the real winner for a two players game, only telling if the player won or lost in the one player case.

5.1.2 Separating one and two players games

Directly above this interface, are defined two other ones :

IEnvironmentSingle for one player games environments.

IEnvironmentTwoPlayers for two players games.

The need to differentiate those two cases comes from the fact that we want to define a method returning the default initial state of an environment, and states are of differents classes whether we implement a one player or a two players problem.

5.1.3 Two more refinements for one player environments

5.1.3.1 Graphical tools

The abstract class `AbstractEnvironmentSingle` provides a minimal graphical interface to one player environments : a window opens showing the current state's value (as obtain by the state's `toString` method).
For a given problem, defining a *design area*, putting it into the java JFrame named `graf`, and using the method `sendState` to update it will provide the user with a graphical representation of the evolution of its problem.
Good examples of implementation of graphical views on problems are given in the *Acrobot,Mountain Car, Mazes, Mobile Robot* environments.

5.1.3.2 Multi-Agents environment

This environment modelizes the local environment as perceived by an elementary agent in a multi-agents frame. In this model, elementary agents just ask the environment to give them the list of possible actions. All other computations (new state, reward) are done into a global *standard* environment, of class `IEnvironmentSingle`, locally defined for each problem (Multi-agent is at this time only defined for one-player games).

```
public interface IAction extends Cloneable,Serializable{
public Object copy();
public int nnCodingSize();
public double[] nnCoding();
public boolean isNullAction();
public int hashCode();
public boolean equals(Object o);
```

Figure 5.2: What an action must do

5.1.4 Environment for Tiling

The interface `TileAbleEnvironment` provides environments with a method which returns the list of tiles corresponding to a (state,action) pair. This list of tiles will then be exploited by a `TDFASelector` algorithm. More details on tile coding will be provided in chapter 8

5.2 Actions

Actions are probably the simplest classes in Piqle. The only things we must be able to do while dealing with Actions is create them, compare them for equality, code them for learning or storing algorithms.
The basic definition of interface `IAction` is described in figure 5.2.

copy create a new Action which is the copy of the previous one : used when storying (state,action) pairs.

nnCoding,nnCodingSize For defining a coding of an action as a real-valued vector, to be used in algorithms based on neural networks.

hashCode,equals It is up to the programmer to define when two actions are declared equal. Equality is tested in Java HashMaps, using those two methods. To avoid inexpected experiments behaviours, be sure to always redefine both methods when you instantiate a new problem !

isNullAction For some games, a player may have no possible action, and must pass, while the game is not finished yet : this can be the case when playing Othello, for example. Null Action must be a special action, or an action's property. The way the program is written forced us to define this method. Note that this method is of no use for the examples provided with Piqle at this time.

5.2.1 Composed Action

For the multi-agent case, the global action performed by the swarm of elementary agents is made of elementary actions coming from each individual agent.

```
public interface IState {
public void setEnvironment(IEnvironment c);
public IEnvironment getEnvironment();

public ActionList getActionList();
public IState modify(IAction a);
public double getReward(IState old, IAction a);
public boolean isFinal();

public IState copy();
public int nnCodingSize();
public double[] nnCoding();
public int hashCode();
public boolean equals(Object o);
}
```

Figure 5.3: What a state must be able to do

This class, derived from the `IAgent` interface, gathers all those individual agents, allows to iterate on those actions, to add a new action, and to retrieve a particular action. More details will be given in chapter 12

5.2.2 ActionList

This is an auxiliary class to store the list of possible actions from a given state. This list is given to the algorithm which makes it choice among the elements (`Actions`) of this list.

5.3 States

A state must define accurately a given configuration of the environment : one must be able to decide whether two states of the environment are equal, or equivalent.
The basic definition of interface `IState` is described in figure 5.3.

setEnvironment getEnvironment : connects the state with the environment it belongs to.

getActionList,modify,getReward,isFinal just call the corresponding methods in the Environment class.

copy to clone a state, mainly used when storing the state into a HashMap.

nnCodingSize,nnCoding define the format of the state's representation for use in algorithms based on neural networks.

hashCode,equals the same as described here above for `IAction`

5.3.1 AbstractState

This abstract class only defines the code of the first four methods seen above, the ones connecting the state with its environment. It also set the field corresponding to that environment.
Remark that agents, and thus algorithms, may not be able to perceive all the details of a state. This is reflected in the fact that states may contain fields which are used by the environment to compute a new state, while those fields are used neither for the computation of `hashCode` and `equal` method, nor for the definition of `nnCoding`.

5.3.2 Interface and abstract class for two-players game state

In addition to the general state behaviour, a two-player game's state knows :

- Who is the winner (in case the game is over).
- Whom is the turn to play.

This adds four methods to this interface, compared with the basic `IState` interface.
In exactly the same manner as for interface `IState`, we define an abstract class above this interface : `AbstracttwoPlayerState`, in order to write the code for all methods but the ones coding the state for a particular problem (`nnCoding()`,`nnCodingSize()`,`hashCode()`,`equals()`) : this is the abstract class `AbstractTwoPlayerState`.

5.3.3 Filter

Filters are mainly used in the multi-agent case (see chapter 12), while there is a priori no problem for using them in a more conventional case. The principle is easy to explain and to understand :
The environment in which a `Swarm` (see chapter 12) is placed might be very complex, while each elementary agent can perceive only a part of this environment. A filter takes as input a (maybe sophisticated) state's description, and extract only the information the elementary agent is allowed to perceive, defining this way a new state of a different (and less sophisticated) class.
Examples of `Filter`'s definition and use can be found in the `maabac` and `ICMLOctopus` environments.

Chapter 6

Algorithms

This chapter details the general organization of package `algorithms`, and the implementation of each algorithm. It gives no explanation concerning the theoretic background of those learning algorithms, especially reinforcement learning algorithms.
For a complete vision of reinforcement learning, please refer to [SB98]
For some algorithms, reference papers will be cited. Those references are also present into the javadoc of the corresponding files.

An algorithm is defined for each `IAgent` : its primary role is to choose an action to perform, given a list of possible actions.
Good algorithms will choose the best action. Learning algorithms will use their past experience to improve their choice, hopefully choosing more often good (best ?) actions.
Thus algorithms in Piqle will mainly implement :

- A method for choosing an action within a list of actions.

- A method for learning, given the starting state, the chosen action, the new state after applying the action, and the reward given by the environment for this action's choice : this is the standard paradigm of reinforcement learning.

Some algorithms may not learn, some others may choose their action at random, depending of those two methods' implementations. Algorithms have several ways to store their past experiences, and also different ways to use this experience.
All that results in a package containing a lot of algorithms, organized in a multiple levels hierarchy we are now going to explain. Understanding this hierarchy may help you define variations of existing algorithms as new classes, or to find where to add a new algorithm of yours.

6.1 A first approach

What we ask an algorithm to do is :

Choose an action This part of its behaviour is contained in the basic interface `IStrategy`.

Learn This is the main method of interface `IStrategyLearner`. We also placed here the method `newEpisode` which should perform all the necessary initialization at the begining of an episode.

Those three methods are called by the `IAgent` whom the algorithm belongs to. Not so important, but maybe useful, is the capacity for someone to understand or analyse what an algorithm actually learned. For those algorithms which store their experiences, this means one must be able to retrieve and analyse those values. Two methods can help :

- By customizing the `toString()` method.
- By extracting a preformatted dataset to be exploited by a neural network : this is why the interface `ISelector` defines the method `extractDataset()`[1]

Having stated all those remarks, we can now describe easily the top of the algorithms' hierarchy :

- A basic interface defining the `getChoice` method : `IStrategy`
- Another basic interface defining `learn` and `newEpisode` : `IStrategyLearner`
- A third interface gathering the two previous ones, and addind method `extractDataset` : `ISelector`

All algorithms defined so far in Piqle are implementations of interface `ISelector`, some directly, others through a cascade of intermediate classes, forming the sub-hierarchy of reinforcement learning algorithms.
We will now describe each of those algorithms, begining with the direct children of interface `ISelector`, and then spending more time on the reinforcement learning algorithms hierarchy.
A complete view of Piqle algorithms hierarchy is shown in figure 6.1.

6.2 RandomSelector

This algorithm returns an action picked at random into the list of available actions. It learns nothing, does not need to reinitialize anything at the beginning of an episode, and is unable to restitute what it might have learned.

[1]A refactoring could perhaps put this method somewhere else in the hierarchy.

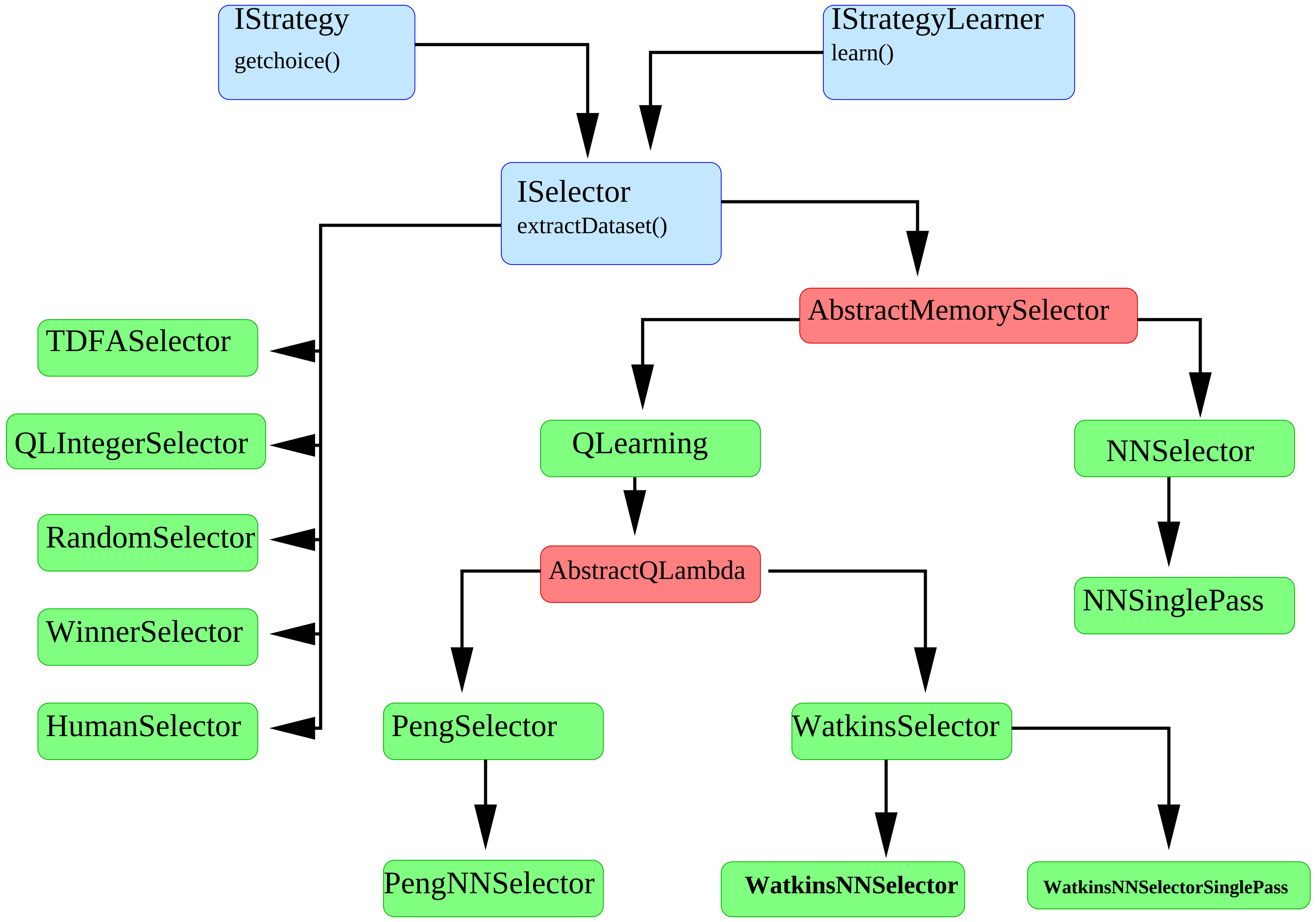

Figure 6.1: Algorithms Java Hierarchy

`RandomSelector` is nevertheless useful :

- For testing new algorithms, or newly coded problems : bugs can only come from learning algorithm or problem coding.
- As a lower bound benchmark : a learning algorithm performing worst than `RandomSelector` is in big trouble...
- As a sparring partner in two players game.

6.3 WinnerSelector

Very similar to `RandomSelector`, except that if there is an action that makes the agent win the problem, this action is chosen.
A not too difficult to write and a little bit more challenging sparring partner.

6.4 HumanSelector

Having an agent which presumabily has learned to play a certain game, it might be interesting to test it yourself, by playing against it. This what the `HumanSelector` is for. This object is quite as simple as `RandomSelector` and `WinnerSelector` : it learns nothing, and only has a `getChoice()` method.
This `getChoice()` method prints a list of available actions onto the screen, and asks the (human) player to choose his movement among this list.
Note that this *algorithm* is not good for learning, as it slows down dramatically all the process (humans are not very fast at reading and typing !)

6.5 TDFASelector

This algorithm is for applying the tile coding paradigm, as depicted in [SB98], chapter 8.3, more precisely the algorithm in figure 8.9, page 213. A more detailed explanation of Piqle's implementation of tile coding will be given in chapter 8.
Roughly speaking, the environment is divided in several *tilings*, and for each of those *tilings*, the pair (state,action) is associated with one `tile`. The $Q(s, a)$ value of a (state,action) pair is computed from the individual value of each tile.
`TDFASelector` uses the list of tiles associated with a (state,action) pair to retrieve and update the value of this pair : thus it differs from *standard* reinforcement learning algorithms in the definition of `getChoice` and `learn`, everything else defined in this class is a copy of what we will see into the reinforcement learning algorithms : maybe some refactoring could replace the `TDFASelector` into the reinforcement learning hierarchy ?

6.6 The Reinforcement Learning Hierarchy

As far as Piqle is concerned, reinforcement learning can be described roughly as follows :

- The algorithm receives from its master (the `agent`) a state, and a list of possible actions.
- It estimates the value of each possible action, according to the $Q(s,a)$ value it has stored or computed.
- It returns the action that it believes will produce the greater reward.
- After the action has been taken, and the state of the environment has been updated, the algorithm updates its $Q(s,a)$ estimation.

This leads to the following preliminary remarks :

1. We need a class which can store and retrieve values indexed with a pair (state,action).
2. We need to know when two states (and two actions) are equal (if we want to be able to apply point 1).
3. We do not need to estimate values for (state,action) pairs we never encountered.

Everything we need for point 2 to be true was already done when we defined `IState`'s and `IAction`'s `equals` and `hashCode` methods, in chapter 5.
Point 3 states that we will avoid to use big (quite) empty arrays to store $Q(s,a)$ values.
We are now ready to detail the reinforcement learning algorithms part of the class hierarchy, beginning with the abstract class that gathers the most important part of the structure and code.

6.6.1 AbstractMemorySelector

This abstract class defines :

A Memory : the place where the $Q(s,a)$ will be stored (whatever the meaning of *store*)

A method for learning : the standard method described in [SB98], page 149.

A method for choosing the next action to perform. Actually three methods, depending of the strategy used to keep a little bit of randomness in the choice.

Parameters for tuning the algorithm.

Each item is now described more precisely, some classes from the package `qlearning` will be just overviewed here, letting our precise description waits until chapter 7.

6.6.1.1 The memory

In this class, the exact kind of memory we are using is left abstract, and will be instantiated differently in each of its implementations.
This memory must implements the `IRewardStore` interface, from the `qlearning` package, which means it is asked to be able to :

- retrieve a value associated with a (state,action) pair : method `put()`.
- store a value associated with a (space,action) value : method `get()`.

6.6.1.2 The learning method

The algorithm used for learning is exactly the one describe in [SB98], page 149 : it will be used without modification in both direct daughter classes, `QlearningSelector` and `NNSelector`.

6.6.1.3 The choice of next action

Roughly speaking, method `getchoice()` returns the best action, as far as $Q(s,a)$ are concerned. But, in reinforcement learning, algorithms must preserve a balance between *exploration* (visiting new states) and *exploitation* (choosing the more rewarding action). This is done by introducing a little bit of randomness in the choice of the next action to perform. Several ways for implementing and controlling randomness have been proposed. Piqle defines three implementations :

ϵ-greedy A random action is choosen with probability ϵ instead of the best action.

Roulette wheel Each action can be chosen with a probability proportional to its $Q(s,a)$ associated value.

Boltzmann selection As in the roulette wheel procedure, each action can be chosen with a probability related to its $Q(s,a)$ associated value, but using a different function than the linear one.

The default behaviour is ϵ-greedy.
Hence the `getChoice()` method is just a branching to another choice function, depending of the chosen implementation.

6.6.1.4 Parameters

Some numbers are necessary to control and tune the behaviour of reinforcement learning algorithms, some others to define precisely the randomness functions. Those values are given with their setters and getters. Some of them also need to change over time, and the class indicates how and how much they must change between two successive episodes. Variables names are as far as possible similar to those used by [SB98].

α **(alpha)** Step-size parameter. Must theoretically decrease to ensure convergence. Two decrease method are proposed, exponential and geometrical. Only the first ensures convergence, but both give good practical results.

γ **(gamma)** Discount rate parameter.

τ **(tau)** *Temperature* for the Boltzmann selection implementation of randomness.

epsilon **(epsilon)** The probability of choosing an action at random in the ϵ-greedy case. Note that ϵ is supposed to decrease over time : we have chosen to let this control to the main program, as could be seen in the examples coming with the Piqle distribution, instead of implementing decreasing schemes as for α. Thus, when you implement an experiment using ϵ-greedy reinforcement learning, do not forget to explicitly decrease and set (with method `setEpsilon`) the value of ϵ.

6.6.2 QLearningSelector

This class is a direct implementation of class `AbstractMemorySelector`, setting its memory to a Java implementation of hash maps. The keys for this `HashMap` will be computed from the values of states and actions, as will be described in chapter 7.
If the `QLearningSelector` directly implements the standard Q-Learning algorithm as depicted in [SB98], Piqle also provides an experimental version of Q-Learning, called `QLearningIntegerSelector`,which only allows to store integers (more precisely bytes) as $Q(s, a)$ values ; this was studied in [AS04], and was intended to implement Q-learning algorithms in resources limited micro-robots. *Note that this class is not a daughter of* `QLearningSelector`*, due to practical considerations.*

6.6.3 Neural networks based algorithms

In this category of classes, the memory is implemented as a neural network.
The two classes implementing the $Q(s, a)$ memory are `NNSelector` and `NNSelectorSinglePass`. The differences between them is not very important, as one will see.
We have roughly speaking the following sequence of actions :

- for each possible action, the algorithm puts in a neural network's input level some coding for the pair (state,action), reads a value as output, which it considers to be the $Q(s, a)$ value.
- Action producing the best value is selected.
- When informed of the reward, the algorithm updates its neural network.

This depicts the behaviour of `NNSelectorSinglePass` *;* `NNSelector` *actually waits to have a certain number of examples to update its neural network.*

Piqle's implementation of neural network is standard feed-forward neural network using the backpropagation algorithm.
Those algorithms are often slower than *normal* Q-Learning, but still effective in learning, while avoiding the space limitation problem for domains with a very big number of states and/or actions.
It is up to the programmer to find the good state and action coding in terms of vectors of real values (see examples in chapter 9), in order to capture the essential features which will allow learning.

6.6.4 $Q(\lambda)$

[SB98] presents two implementations of this reinforcement learning algorithm, which combines Q-Learning and the use of *eligibility traces*. Refer to the book for a complete description and study.
Piqle proposes an implementation for both algorithms : as they share a lot of common characteristics, an abstract class gathering those common parts of code and those common variables is first defined, then instantiated in each case.

6.6.4.1 AbstractQLambdaSelector

$Q(\lambda)$ algorithms introduce two new features, relatively to Q-Learning algorithms :

Eligibility traces to remember and reward states encountered on the way toward the resolution of the problem.

λ **(lambda)** the decay-rate parameter for eligibility traces, making far in the past encountered states less important than freshly met ones.

The class `AbstractQLambdaSelector` introduces those two new features, and basic methods to handle them. The learning and action choosing methods will be defined quite differently in both daughter classes.

6.6.4.2 Watkins's $Q(\lambda)$

A direct implementation of the above abstract class, defining `learn()` and `choice()`, according to the algorithm detailed in [SB98], page 184.
The two direct daughter classes of this class only differ in that they use a neural network (normal for `WatkinsNNSelector`, with no dataset, learning at each step for `WatkinsNNSelectorSinglePass`) as the memory for reward values.

6.6.4.3 Peng's $Q(\lambda)$

Algorithm cited in [SB98] and detailed at :
`ftp://ftp.ccs.neu.edu/pub/people/rjw/qlambda-ml-96.ps`
`PengNNSelector` uses a neural network as memory.

Chapter 7

The Q-Learning utilities package

This package gathers some useful classes for implementing Q-Learning algorithms.

- Grouping a state an an action into the same object, use it as a key into HashMaps.
- Defining different types of objects to store $Q(s, a)$
- Defining Value Iteration.
- Defining an object to handle eligibility traces.
- Defining different ways to return a default $Q(, s, a)$ when none is still available.

7.1 ActionStatePair

When storing $Q(s, a)$ values into HashMaps, one need to define a key. In the Q-Learning settings, the natural key is the pair (state,action) associated with this $Q(s, a)$ value. Hence we define a new object, `ActionStatePair` which group into a single object an `IAction` and a `IState`.
Methods `hashCode()` and `equals()` are redefined for this class, from the definition of similar methods in the corresponding state and action.
Nothing has to be done in this class, but remember that it is important to carefully define `hashCode()` and `equals()` when instantiating a problem : the notion of equality you will define for your states and actions is the one that will be used to store and retrieve your $Q(s, a)$ values !

7.2 Remembering the rewards

Every class designed to store of compute $Q(s, a)$ values must implement the `IRewardStore` interface, which defines three methods :

- `get()` : when a value is already stored, returns this value. Otherwise, return a default value. The kind of default value returned can be controlled, see forward in section 7.3.
- `put()` Enter a reward into the storing mechanism, indexed with state and action.
- `extractDataset()` Transform the $Q(s, a)$ values into a dataset suitable for use with Piqle's neural networks. This method is not always implemented, and may return NULL.

There are four ways to store expected reward values in Piqle at this time :

- Using HashMaps (for real numbers or integers).
- Using neural networks (with or without a learning data set).

7.2.1 Using HashMaps

The most natural way to store $Q(s, a)$ values is in arrays indexed by states and actions, or by some coding derived from those two objects. The problem is that it might need a lot of space, and a lot of cells can be void and useless.
Java's HashMaps make it possible to store only values for (state,action) pairs actually encountered during an episode : values are stored into the HashMap, the (state,action) pair serves as the key to retrieve them.
Two classes are implementing `IRewardStore` in terms of Java's HashMaps :

- `RewardMemorizer` for standard real-valued $Q(s, a)$ values.
- `RewardMemorizerInteger` for integer (byte) valued approximation of $Q(s, a)$ values, as described in [AS04], and used in `QLearningIntegerSelector`.

7.2.2 Using neural networks

7.2.2.1 General Principles

When state and/or action space become too big, even `HashMap`s are unable to store all the experience an agent met while performing successive episodes. The idea is then to try to learn the $Q(s, a)$ value as a function of the (state,action) pair. Any machine learning algorithm could be used, but at this time, only feed-forward neural network with backpropagation are used[1].
The general learning scheme is as follows :

[1] Actually, a previous version of Piqle implemented a more general scheme, using Weka's implementations of machine learning algorithms : this extension had to be rewritten off, and may appear in next versions of Piqle.

- The algorithm stores a fixed number of (state,action,reward) 3-uples.*During this storing phase, the algorithm chooses an action at random.*
- When this number of 3-uples is reached for the first time, a neural network is build, and successive decisions will be taken by returning very often (there is still some exploration !) the action for which the output of the network is maximal.
- While returning the best action at each step, the algorithm continues to store the new 3-uples the agent sends to it.
- After a new stock of 3-uples are memorized, the network updates its weights by launching a learning phase.

This behaviour is implemented in class `RewardMemorizerNN` : a variant, where the neural network is build first, and learns at each step, without memorizing any dataset is implemented in class `RewardMemorizerNNSinglePass`.
Those neural networks can be used in Q-Learning algorithms without changing those algorithms, just telling Java that the `IRewardStore` will be of class `RewardMemorizerNN` : look for example at the source code for `NNSelector`, which is said to inherit from `AbstractMemorySelector`, and where the code only define methods and fields linked to the neural network use.
Classes `PengNNSelector` and `WatkinsNNSelector` also illustrate this simple definition.

7.2.2.2 Coding States and Actions for Neural Networks

Piqle's neural networks need as input a vector of real values, and gives as output a single real number.
The input is build from the (state,action) pair, which means that we must provide way to transform a (state,action) pair into a vector of real value.
This is done in three steps, situated at different places into Piqle's class hierarchy :

- Code a state as a vector of real numbers : this is defined in interface `IState` from package `environment` by method `nnCoding()`. For each problem, it is up to the programmer to find the good coding scheme for the states, in term of a vector of real values.
- Code an action as a vector of real numbers : defined in method `nnCoding()`, from interface `IAction`, in package `environment`. Here again, the programmer must find, for a given problem to code, the best way to code an action into a vector of real numbers, in order to capture all the relevant features which will allow the network to actually learn.
- Group those two vectors into a single one, which will be fed into the network : this is done in methods `get` and `put` from class `RewardMemorizerNN`, in package `qlearning`.

A standard way to code state or action into a vector of real numbers, which seems to give reasonably good results :

- For each state or action feature that can be expressed as a real number : use directly this value, perhaps after having normalized it.
- For each discrete field, which can take k distinct values (k not too big): code it with k input cells, from which only one will be equal to 1, the others being 0 (one among k).

7.3 Choosing a default value

The first time an algorithm looks into either its HashMap or its neural network for a $Q(s, a)$ value, it will off course find nothing, but must return a value ! In Piqle, the return value can be chosen by the programmer, when defining the algorithm.
The interface defining the default value to be returned in case no actual value is available is called `IDefaultValueChooser` and is part of package `qlearning`. This interface contains only the method `getValue()`.
The classes implementing this interface are :

NullValueChooser which always return zero.

ConstantValueChooser which returns a constant value, passed as parameter.

IntervalValueChooser returns a value uniformly distributed between two bounds (parameters)

Of course, any other strategy for choosing a default return value could be added to this list.
The default value chooser is the `NullValueCHooser`. One can change this default setting be using the algorithm's Constructor asking for a value chooser as parameter : this information is relayed from the algorithm down to the `RewardMemorizer` included in this algorithm [2].

7.4 Eligibility traces

For $TD(\lambda)$ algorithms, we need to remember all the (state,action) pairs the agent encounters during the current episode. Each of those pairs is associated with a trace : a real number. All those *eligibility traces* are stored in a set, actually an HashMap.
Available operations are :

- Adding an eligibility trace to the set.

[2]At this time (15/12/2006), this scheme has still to be implemented for `RewardMemorizerInteger`

- Retrieving a value.
- Iterating on the set of eligibility traces.

All operations that will be called by $TD(\lambda)$ algorithms.

7.5 Value iteration

When states and actions are easily enumerable, one may wish to apply the Value Iteration algorithm to compute $Q(,s,a)$, without applying Q-Learning algorithms.
See [SB98],page 100 for more details. Example `Jack's Car Rental` illustrates the use of this interface.
This allows to compare algorithms with the result of an iterative algorithm. Interface `ValueIteration` defines the methods needed to implement this algorithm in Piqle :

computeVStar : compute iteratively $V^\star(s)$

displayVStar : print the computed value on the output : you might have to reformat it to your needs.

extractPolicy : derive an optimal policy from $V^\star$

Chapter 8

Tile Coding

8.1 Introduction

Tile coding, also known as CMAC, offers the possibility to treat problems with large state spaces within the frame of reinforcement learning, avoiding the curse of memory overflow. Together with adapted algorithms, it enables users to approximate state or (state,action) values for otherwise intractable problems.
For more details on tile coding, and related algorithms, see [SB98], chapter 8. At this time, we implemented rectangular tile coding, but very few work will have to be done to add implementation of irregular tiling, log stripes, diagonal stripes, as illustrated in [SB98] figure 8.6 or rectangular rotated tiling, for example.
Later in this chapter we will show where to create the Java classes for those implementations, without modifying the rest of the library structure.
Note that this implementation of tile coding is suitable only for problems where states are expressed in term of continuous components, and actions are discrete.
Concerning the learning algorithms which can use this notion of tile coding, we implemented the linear, gradient-descent version of Watkin's $Q(\lambda)$, as describe in [SB98], page 213, more precisely, the corrected version which can be found on the book's website : `http://www.cs.ualberta.ca/~sutton/book/QFA.pdf`
The rest of this chapter is organized as follows : section 2 will describe the hierarchy of classes and the link between them and some Java tricks and considerations, section 3 will explain how those classes are used for two problems : the classical mountain car and acrobot, and show some results obtained by using tile coding.

8.2 Implementation

Two aspects of the implementation have to be explained :

- How the tile coding paradigm is implemented in Java, and how it finds its place into Piqle.

- How to call tile coding to solve a particular problem.

8.2.1 Java and Piqle Implementation

8.2.1.1 Individual tiles

The basic brick of all this construction is a `Tile`. A `Tile` is some portion of space, belonging to a unique `Tiling`, which is a set of `Tiles`. A `Tile` only has to remember the `Tiling` it belongs to, and its own θ coefficient. We also need a method enabling the program to read and update θ (methods `setTheta` and `getTheta`).
Others methods are left for the sake of monitoring and debugging.
Note that due to the way Java treats equality among objects, we do not need to redefine `hashcode` nor `equals` for `Tiles` : we will just have to ensure that a given `Tile` will not be created twice : this will be done in the `Tiling` class.

8.2.1.2 Tiling

`Tiling` is an abstract class, whose only method returns the `Tile` associated with a certain (state,action) pair. By extending this class, one can define different partitions of the space covered by the `Tiling`. At this time (15/12/2006), we only implemented a generalization of hypercubic tiling of space (`HyperRectangularTiling`) and a more general version, allowing to ignore certain dimensions : `HyperRectangularSparseTiling`.

Implemented tilings

`RectangularTiling` : This `Tiling` instantiation is left as an example, and could be used with two-dimensional state definitions (like the Mountain Car problem). In addition to the basic features of the `Tile` class, one can access to the coordinates of the tile inside the associated `Tiling`. Within the `RectangularTiling` class, method `computeTile(double x,double y)` is the place where the program associates a `Tile` to a point (x, y) of the states space. Note that the end of this method ensures that no tile will be generated twice.

`HyperRectangularTiling` : This is a straightforward generalization of the class `RectangularTiling` to any number of dimensions.

`HyperRectangularSparseTiling` : this extension of the class `HyperRectangularTiling` enables the user to ignore certain dimensions when defining a `Tiling`. This permits to reproduce quite exactly the *standard* settings of Sutton's experiments with Mountain Car and Acrobot problems. `MountainCarTilingH` and `AcrobotCLS2Tiling` environments illustrate how one can define those original settings (see below).

How to define new tilings

A `Tiling` is just a partition of a n-dimensional space, eventually permitting to ignore some dimensions. In order to define a new `Tiling`, one just has to clearly define this partition, write a method `computeTile(IState s,IAction a)`, and ensure that no `Tile` while be created twice. The best places to understand how to do that might be the classes `RectangularTile` and `RectangularTiling`.

8.2.1.3 Some auxiliary classes

Inside the `tiling` package, three auxiliary classes are written, to allow clear and easy manipulation of `Tile`s and `Tiling`s. As those classes are intended to help the manipulation of `Tile`s and `Tiling`s, there might be no need for the user to modify, extends of look inside them.

`ListOfTiles` : this class collects all the `Tile`s, from different `Tiling`s, associated with a particular (state,action) couple. It also contains a method to compute the sum of θs.

`SetOfTilings` : in the CMAC scheme, each action is associated with a set of several `Tiling`s. This class gathers all the `Tiling`s corresponding to a specific action. When defining an environment designed to be solved by applying tile coding, one will have to create and define as many `SetOfTilings` as the number of possible actions. See Mountain Car and Acrobot examples.

`EligibleTiles` : the learning algorithms using tilings are maintaining a list of `Tile`s met during the exploration of an episode. This class is meant to memorize those `Tile`s, allowing to manipulate them ; mainly modify their θ, or erase the complete list of `Tile`s.

8.2.1.4 Algorithms using tilings

Sutton and Barto describe two algorithms using tilings : a linear gradient-descent Sarsa(λ), and a linear, gradient-descent Watkins's Q(λ). At this time (15/12/2006), we only implemented the second one, from the pseudo-code available at `http://www.cs.ualberta.ca/~sutton/book/QFA.pdf`. The corresponding class, located in package `algorithms`, is called `TDFASelector`.

8.2.1.5 Connecting tilings and problems

Tilings are strongly dependent of the problem to solve, so, it seems quite difficult to define an independent tiling scheme. The solution we decided to use is the following : each problem we want to solve using tilings must be defined by an environment implementing the `TileAbleEnvironment` interface, which forces an environment to have a method which can return a list of tiles associated with a (state,action) couple. In addition this environment must define precisely the tilings. Have a look at `MountainCarTiling` and AcrobotCLS2Tiling : their constructors instantiate the tilings, and the method `getTiles`, from interface `TileAbleEnvironment`, is defined.

8.2.1.6 Where to tune ?

There are several places where parameters can be modified, and there are also several ways to do that.

Size of Tiles : the number of `Tile` following each `Tiling` dimension must be filled into the forth parameter of the tiling constructor (at least for `HyperRectangularSparseTiling` class), named nbDiv in the environment class constructor.

Ignored dimensions : this is defined by the `pvalid` boolean array in the environment class constructor.

Number and Structure of `Tilings` : again, define them into the environment class constructor. Have a look at `AcrobotCLS2Tiling` and `MountainCarTilingH`.

Algorithm parameters : the class `TDFASelector` provides all the methods to set, read, modify the algorithm parameters $\alpha, \gamma, \lambda, \epsilon$

8.2.2 Implemented problems

Two problems are implemented :

Mountain Car : the program `MountainTiling` illustrates the implementation of tiling for this problem, using the `MountainCarTilingH` environment definition. Figure 8.1 compares the behavior of `TDFASelector`, `QLearningSelector` and `WatkinsSelector`, in terms of average reward, on this problem. The only parameter that had to be changed beetween each experiment is α.

Acrobot : `TestAcrobotCLS2Tiling` illustrates the seek of a solution using the tiling paradigm. Tiling is the only algorithm which learns how to move the acrobot, and the solution seems to be very sensitive to parameters settings. Figure 8.2 shows an example of learning curve, in term of episode length before to reach the goal.

8.3 Remarks

8.3.1 RLFramework

To define a tiling scheme, one needs to know the upper and lower bounds of values in each dimension, and those informations are provided by the RL-Framework defined by the people at the University of Alberta (`http://rlai.cs.ualberta.ca/RLBB/RL-Framework-concepts.html`)

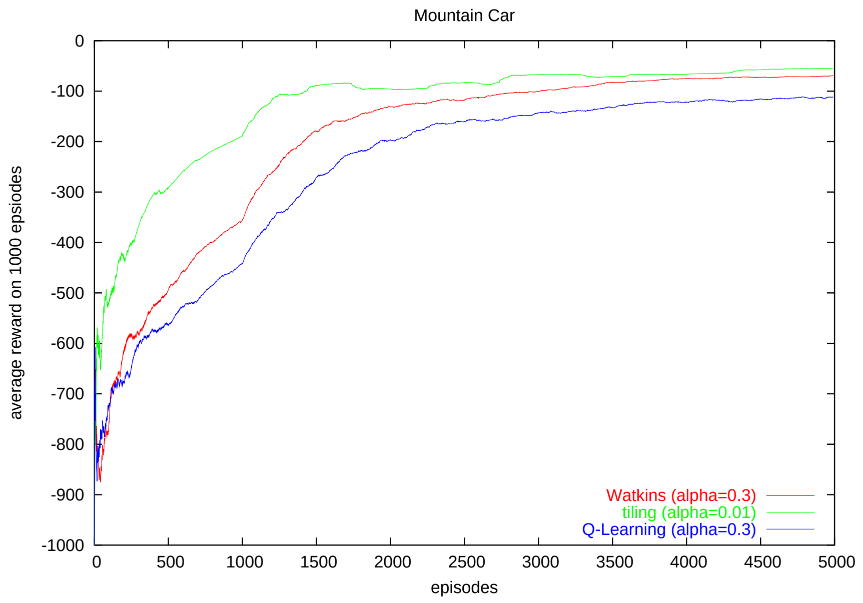

Figure 8.1: Comparing tiling, QL, and Watkins on Mountain Car

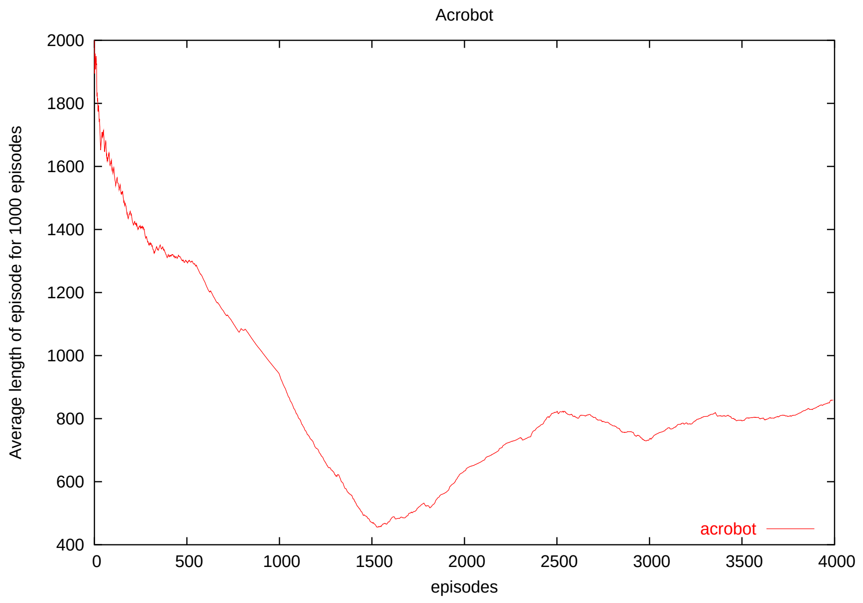

Figure 8.2: Learning behaviour of Acrobot

Defining a tiling for problem expressed in this syntax might be straightforward. Next versions of Piqle will include wrappers to connect it with the RL-Framework.

8.3.2 Tuning

Finding the good learning parameters seems to be very difficult, as the algorithm looks very sensible to parameter variations.

Chapter 9

Two Short Illustrating Examples

This chapter will present two examples, one for the One Player case, and the other for the Two Players case. We will try to show the successive steps leading from the definition of the problem we want to solve, to the final Piqle implementation. Sources for those two examples can be found into the distribution.

9.1 Cannibals and Missionaries

Cannibals and Missionaries is an old puzzle : three cannibals and three missionaries have to cross a river, using a boat that can contain only two people. There must always be someone to ride the boat, and if, by chance, the number of cannibals exceeds the number of missionaries on any bank, the cannibals eat the missionnary men, and the game is lost. The final goal is to carry all the people on the other bank.
One can generalize the game, changing the number of cannibals and missionaries, or the size of the boat : Martin Gardner analyzed those variants in a Scientific American column :

- Boat with one seat only : no solution.
- Boat with two seats :
  - One cannibal and one missionary : one solution in one step.
  - Two of each kind : at least one solution in five steps.
  - Three people of each kind : four solutions in eleven steps.
  - More than three people : no solution.
- Boat with three seats :
  - Four people of each kind : at least one solution in nine steps.

  - Five people of each kind : at least one solution in eleven steps.
  - More people : no solution.

- Boat with four seats or more : there is always a solution (always leave one cannibal and one missionary in the boat, and, at each step, make one other people of each kind cross the river)

The problem is not difficult, and can easily be solved *by hand*, but it is a good way to show how to code it.
We will first explain how to code the original problem in order for Piqle to find the solution, and then we will show how easy it is to code the more general settings, and still make Piqle find the solution.

9.1.1 A Brief Analysis

9.1.2 Actions

Depending on the size of the boat, k, the number of possible actions must reflect all the possible boat loads, i.e : k_1 missionaries and k_2 cannibals, k_1, k_2 verifying $k_1 + k_2 \leq k$ and $k_1 + k_2 \geq 1$.
Thus the class `MissionaryAction` will quite only consists of two integer fields : one for k_1 (`missionariesOnBoat`), the other for k_2 (`cannibalsOnBoat`). Note that in this case, the `MissionaryAction` object does not know the size of the boat.
The verification of the validity of k_1 and k_2 values is left to the environment, which will be asked to produce the list of all possible actions.
We now need to define equality between two actions : the natural way to define it is to say that two actions are equals if both of their fields are equals.
Hash code is meant to speed the search of a given action into the storage structure, we choose there a simple function ($k_1 + 3 \times k_2$). This choice is not critical there, as the problem is not a big one, the number of actions and states remains reasonable, and a hash code avoiding collisions between objects does not need to be efficient. For more heavy problems, one might consider spending time to analyze and design a good hashing function !
The last part of class `MissionaryAction` to be defined is a coding for use with neural networks. We must code two integers, and the natural possible choices are :

1. Two real numbers , $\frac{k_1}{k}$ and $\frac{k_2}{k}$
2. Two concatenated vectors of $k+1$ *booleans* each, where only one boolean in each vector is true (the ones at position k_1 in the first vector, and at position k_2 in the second one). In practice, we define a vector of size $2 \times k + 2$ (there can be between 0 and k missionaries or cannibals), and set to one the elements at index k_1 and $k + k_2 + 1$. This completely defines methods `nnCoding()` and `nnCodingSize()` for class `MissionaryAction`.

We have chosen the second solution, and example program `Missionaries.java` shows that those settings make learning effective. Note that this is not very economic, as the neural network will have a lot of entries, while only a few will be active. Another solution would have been to limit the size of the vector, let's say to 10, and apply a modulo operation to find the non null index:

```
public double[] nnCoding(){
double code[]=new double[20];
code[this.missionariesOnBoat%10]=1.0;
code[10 +this.cannibalsOnBoat%10]=1.0;
return code;
    }
```

9.1.3 States

A brief analysis of the problem shows that the only things someone has to know to solve the problem are :

- The initial number of missionaries (which is the same as the initial number of cannibals in all the *normal* variants).
- The number of missionaries on the left bank (starting bank) of the river.
- The number of cannibals on the left bank (starting bank) of the river.
- The side where the boat is.

As it will soon become clear, the initial sizes of the populations can be memorized by the environment, leaving it to the state to memorize the other three informations.
Thus a `MissionaryState` will consists in three fields :

MaLeft number of missionaries on the starting bank.

CALeft number of cannibals on the starting bank.

isAtLeft true if the boat is at the starting bank.

Equality of two states is the equality of each field, hash code function is similar to this defined for `MissionaryAction`, and neural network coding is also similar to the one designed for `MissionaryAction` (in the *modulo* version, with two added inputs to code the bank where one can find the boat).

9.1.4 Environment

The environment will memorize game's constants, like the sizes of the initial populations and eventually the boat's capacity.
It will also code the rules of the game by :

- Building the list of available actions starting from a given state.

- Telling when the game is over.
- Computing the next state, after a move is chosen and performed.

Computing the next state is independent of the game's rules (if we suppose all meals are taken on either bank, and not into the boat), while the two other points can give way to discussion :

- Shall we code the rules as told before ? This means that the only available moves are *safe* moves, where no missionary will be eaten ?
- Shall we let the program learn the rules, by letting it trying loosing moves, and then giving to them a negative reward ?

Reinforcement learning, and hence Piqle, let the programmer choose its own conception of the game. Two environments are provided :

`Congo` Where the rules are observed, and only legal and non loosing actions are allowed.

`Brazzaville` a subclass of `Congo` allows loosing moves, by implementing differently both `getActionList()` and `getReward()` methods.

9.1.4.1 Only non loosing moves

In this configuration, the only final move is when everybody has landed safely on the right bank of the river. In this case, a reward of $+1$ is given, otherwise, the reward is null for any other step of the episode.
In this implementation, episodes ends either because the maximum number of episode's steps is reached, or a solution is found.
This suffices for the program to find the shortest solution, because the decreasing factor γ forces the program to find the shortest solution in order to maximize its cumulative discounted reward.
See `Congo.java` for more precise information of the settings, and `Missionaries.java` for the sample application program.

9.1.4.2 Any move

Learning is there more difficult, as the program has not only to learn the sequence of winning moves, but also, and perhaps at first, the legal moves !
Note that in this case, an episode can ends in a very few number of steps, while granted with a bad reward.
In order for this implementation to allow learning, some indications have to be given, by means of the reward definition : a negative reward is given for each safe but non-terminating move, a huge negative reward in case of faulty move, and of course a positive reward in case of success.
By properly tuning the algorihtm and episode parameter, learning in this case is also possible, and might be very fast. This is left as an exercice…

9.2 Jenga

Jenga is a two players game which gathers several interesting properties relatively to Piqle implementing and reinforcement learning :

1. The rules are simple, and thus simple to implement.
2. Actions are limited (three different ones).
3. States can be described simply.
4. The number of moves in one play is finite.
5. There is a winning strategy for the second player.

This makes this game a good example to implement and test Piqle's learning algorithms.

9.2.1 The Game

Jenga is a game where each player, on his turn, must take a rectangular block out of a tower or regularly disposed blocks, avoiding to make the tower collapse. The tower is made of 18 layers of three blocks each, each block of a layer making a 90 degrees angle with the upper and the lower layers.
Zwick ([Zwi02]) shows that there is a winning strategy for the second player, for the commercial version of the game, and that there is always a winning strategy when one changes the height of the tower.
Thus, if we succeed in coding the problem, we should be able to build an agent which plays Jenga perfectly.
[Zwi02] will also help us defining actions and states, as Zwick carefully defined them. Please refer to his paper for the justification of the state and actions coding.

9.2.1.1 Actions

[Zwi02] shows that there exists only three different actions :

- Taking the central block out of a three blocks layer.
- Taking a side block from a complete three blocks layer.
- Taking the lateral block out of a two contiguous blocks layer.

Hence the class `ActionJenga` is quite the simpliest action class one could define : it build three static actions, each one defined by an integer, and equality is just Java's default `equals()` method [1].
Because of this limited number of possible actions, neural network coding can be expressed in a *one among k* scheme.

[1]One can find a similar construction when a problem has a finite (little) number of different actions, see the mazes examples, where the robot can only go forward, backward, turn left or right.

9.2.1.2 States

[Zwi02] shows that the following quantities completely define a state :

- The initial size of the tower.
- The number of blocks on the top layer (0,1, or 2) :Z.
- The number of three blocks layers : X.
- The number of two adjacent blocks layers Y.

The last two items are standing for the number of layers out of which a player can take a block.
In fact, [Zwi02] shows that the values of $X \bmod 3$ and $Y \bmod 3$ are already informative enough : we could have used this enhanced representation without problem (left as an exercice).
Thus a state will be a class with those four integer fields, equality is defined simply by equality of each field, standard hashing function is easy to define. There is neither no difficulty to understand the neural network coding (which uses the simplification remark made above about the modulo 3 coding).

9.2.1.3 Environment

The main methods of class `JengaTower` are easily written if we recall the rules of the game.

`getActionList()` Two actions are possible for a full three blocks level, while only one action is possible for a two adjacent blocks level. More, it is not important to know which actual level will be transformed, we just need to know if at least one of the levels of one of those two kinds is still existing into the tower. Thus, the list of possible actions will contain three,two,one, or zero possible moves. In case the list is void, the player looses the game.

`successorState` Knowing the action taken, and the tower's configuration, there is no problem for writing or understanding this method.

9.2.1.4 Testing

Program `JengaTest.java` demonstrates that a Q-Learning algorithm learns quickly how to win : after 9000 episodes, it never loose not even a game. By using agent's `explainValues()` method, one might be able to retrieve the conclusions of [Zwi02] (left as an exercice).

9.2.2 General remarks about two players games

The programmer must in this case be careful as the notion of `turn`, indicating which player must play becomes important for a lot of reasons :

- List of possible moves.

- Telling who won.
- Telling whether the game is finished or not.
- Computing the reward.

Note also that Jenga is quite simple, as both players are using the same blocks. For other games, like checkers, each player has his own pawns.

Chapter 10

The One Player Case

This chapter will present in some details, but less precisely than for the `Cannibals and Missionaries` problem, each one player example available in the sourceforge release of Piqle applications.
We will explain the design and coding of actions, states, and environments, and give some clues about learning experiments.

10.1 Acrobot

The acrobot problem is described in [SB98], page 270 : a two-link underactuated robot must reach the horizontal position by applying a torque to its central joint : it is similar to a gymnast trying to reach the horizontal position by only moving his legs. A lot of simplifications are made in the initial model, but the Piqle's implementation coresponds strictly to this model (no further simplifications).

10.1.1 Actions

There are only three possible actions, we applied here the same implementation than in the `Cannibals and Missionaries` problem from section 9.1.2. In fact, if one compares the two files `ActionAcrobot.java` and `ActionJenga.java`, one will see that, except for the package and import directives, the rest of the code only differs in the variables and fields names.

10.1.2 States

A configuration of the acrobot can be defined by four continuous variables :

- The angles of the two poles, θ_1 and θ_2
- The angular velocity of each pole $\ddot{\theta}_1$ and $\ddot{\theta}_2$

Hence an `AcrobotState` will contain those four fields. Having to deal with continuous values will pose some problems in defining equality and hash code

function, and to ensure that those two functions are compatible (i.e two objects said to be equal must have the same hash code !).
This is not the case in our implementation, but we believe that the number of problem remains negligible.
The hash code function for a state is build upon the integer part of its four fields, combining those four integers into a single one, using standard hash code constructions with prime numbers.
Two states are said to be equal if they are close enough in each of their dimensions.
There might be possible, on the border, for two states to be equal, but having different hash code functions.
Nevertheless, we can show that for certain settings, our test programs can learn with such approximative functions. Defining others more precise `equal()` and `hashCode()` methods might lead to better results.

10.1.3 Environment

A simple hierarchy of environment has been defined :

- `Acrobot` implements a first version of the system's physics. As there were some problems with the equations, we soon defined an upper class :
- `AcrobotCLS2` implements a more robust definition of the `successorState()` method, strictly copied from the CLSquare implementation: `http://www.ni.uos.de/`.
- Both classes `AcrobotCLS2Tiling` and `AcrobotCLS2TilingSimple` inherit from `AcrobotCLS2`, each one adding a different tiling frame.

10.1.4 Possible experiments

Several files are given to test those different environments :

- `TestAcrobotCLS2.java` uses a Watkins's $TD(\lambda)$ algorithm together with a neural network to store/learn $Q(s, a)$ values. It seems that neural network based algorithms can learn to solve this problem, while other more classical reinforcement learning algorithms are unable to find one. Maybe because the way neural network based algorithms explore the environment is different ?
  A default $Q(s, a)$ value of 2 forces the algorithm to consider new states as interesting ones, and thus forces exploration.
- `TestAcrobotCLS2Tiling.java` uses one of the two tiling schemes available for the acrobot. The frame of the episode is a little bit tricky : in a first time, the agent explores a lot (epsilon is kept fixed at 0.3), with episodes of length 1000. After it succeeded in finding a solution 1000 times, the length of an episode is reduced to 500, and epsilon begins to decrease. You can change the tiling scheme by changing the declaration at line 15.

Those two files are given as basic examples, a lot of things can be done by either tuning parameters, or choosing other algorithms, and trying new tiling schemes.

10.2 Mountain Car

The mountain car problem is described in [SB98], page 214. A car must climb onto the top of a hill. It can turn its motor in order to go full throttle forward, full throttle backward, or turn it out. The car does not have enough strength to climb the hill by simply going forward : it must first go backward.

10.2.1 Actions

Nothing new in this section : as for the two environments already described in this chapter, there are only three actions, the class `ActionMountainCar` differs from the two above classes just by renaming variables !

10.2.2 States

The state of the car is completely defined by its position and its speed (two real numbers) : this kind of state is very similar to the acrobot state's definition, hence class `MountainCarState` is very similar to `AcrobotState`.

10.2.3 Environments

Three environments are given :

MountainCar The basic version of the environment : defines position and speed bounds, and compute the successor state according to the physics described in [SB98].

MountainCarTiling defines a tiling : ten rectangular two dimensional (position $\times$ speed) randomly shifted tilings for each (three) action, as defined in [SB98].

MountainCatTilingH : same as above, but written with the generalized tiling definition, which allows to ignore certain dimensions for a tiling.

Program `MountainGraphic.java` illustrates the resolution of this problem, the graphical interface is designed by José Antonio Martin.

10.3 Gambler

The Gambler's Problem is described in [SB98], page 101, and serves as an example for Value Iteration. The Piqle implementation uses the Value Iteration Piqle's interface to produces figures allowing to redraw figure 4.6 page 103 of [SB98].

10.3.1 Actions

The gambler's actions are the amount of the bet, which is an integer between 1 and the gambler's capital. Hence, an Action while be an object with a unique integer field : the value of this bet.
There is no particular of difficult to understand topic in the definitions of the other methods.

10.3.2 States

Similarly, a state is completely defined by the amount of the gambler's capital. This class will also consist of a unique integer field, thus making the methods very easy to write and to read.

10.3.3 Environment

The Gambler's environment must implement the rules of the game, that is to say :

- You can bet any value between 1 and your capital : thus leads to a list of possible actions of size *capital*.
- The next state has a value of your capital plus or minus your bet, depending if you won or not.

As this environment is also required to implement `ValueIteration` interface, we must add methods `computeVStar` and `extractPolicy`. Please remark that those two methods strongly rely on the definition of states and actions, in the way they enumerate them, hence it might have been difficult to code them outside the `GamblerGame` class ; making `computeVStar` more generic seems out of reach. . .
Program `GamblerTest.java` shows how to experiment with this problem. Have a look at the comments inside this program.

10.4 Mazes

Mazes and finding your way inside them is a current topic in reinforcement learning. Package `mazes` tries to implement differents versions of mazes problems :

- What kind of actions are possible ?
- What can one perceive from its environment into the maze ?
- What is the task one is asked to perform ?

The mazes we consider in this section are composed of discrete squares, the agent can move from a square to an adjacent one, if possible.
Thus, different versions of actions, states, and environments are proposed. A more synthetic view might allow to gather and rewrite everything in a more generic manner…
Combining those different definitions leads to a number of different environments, illustrated in several example programs.
We will first describe the different kinds of actions, the different kinds of states, then some environments based on those actions and states : we guess it will be easy for the programmer to extend the definitions in order to cast them to his needs.

10.4.1 Actions

Having in mind the problem of a robot asked to find its way through a maze, we defined the possible actions of an agent as either :

- The different directions it can move towards (four or eight directions).
- Moving (forward) or turning (which angle ?)
- Eventually actionning tools (not implemented in the actual version).

The first two behaviours are implemented in two independent classes we are going to describe now :

10.4.1.1 MazeAction

Nine moves are possible : moving in any of the eight directions, or stay still. Those moves are coded using two integers x, y which values are between -1 and +1

10.4.1.2 AliceAction

Inspired by the coding used in [AS04], the agents are here allowed to :

- Move forward.
- Move backward.
- Turn left 90°.
- Turn right 90°.

Figure 10.1 depicts the basic maze actions hierarchy.

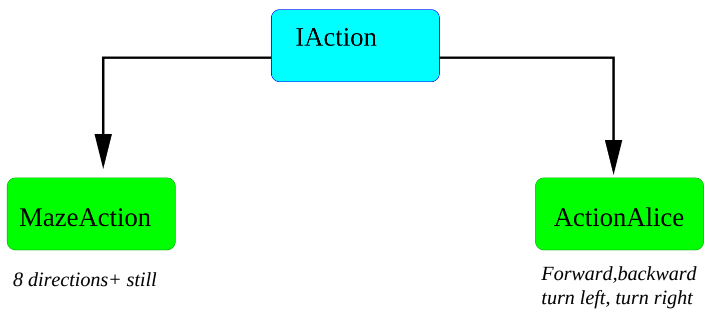

Figure 10.1: Maze actions hierarchy and characteristics

10.4.2 States

A state must contains enough information for the environment to locate it into the maze. But this information can be hidden to the agent : the agent can convey information about its position (namely, the coordinates of its position), without being allowed to use it for learning.
Thus, in some cases, the state definition will contain information not available for the learning algorihtm, and this will be seen because those information will not be used when coding hashCode, equals, or neural network coding of the state.
The hierarchy of states classes is depicted on figure 10.2

10.4.2.1 The basic class : `MazeState`

This basic class consists of only two integer fields x, y denoting the coordinates of the agent into the maze.
As the size of the maze is not limited, neural network coding can not be of type *one among* k.
Note that in this case methods `equals` and `hashCode` are defined from the x, y fields, hence learning is based upon those values, and then the agent learns according to its coordinates inside the maze : the agent can have at last a perfect geographical knowledge of its environment.
The associated environment is class `Maze`.
Learning in this case is quite easy, as illustrated in example program `SimpleMazeExample.java`, where `QlearningSelector` and `NNSelectorSinglePass` both give good and quick results. At first sight, `NNSelector` seems to have much more problems, at least with the default settings.

10.4.2.2 LocalMazeState

In this case, the agent only knows what surrounds it in the eight directions, (that is to say, the value of the eight squares around it), and its distance from the treasure.

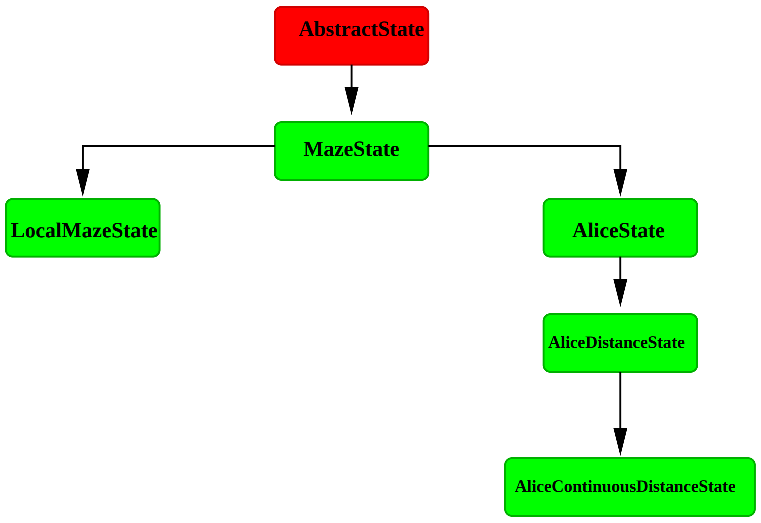

Figure 10.2: Maze states hierarchy

Looking at method `equals` shows that only those informations are taken into account to decide whether two states are equivalent.
While comparing the example program `LocalVisionMazeExample.java` with `SimpleMazeExample.java`, one will notice that the only change is the class the maze belongs to, which changes from `Maze` to `LocalVisionMaze`.

10.4.2.3 AliceState

The agent has information about what it sees forward, backard, on its right, and on its left, but has no ideas of its orientation. Its coordinates in the maze, and its orientation are fields of `AliceState`, but are not used neither in `hashCode()`, `equals()` nor for neural network coding, hence they are not used for learning. There is no treasure in that setting, the agent's goal will be to stay as long as possible into the maze, moving forward. Negative rewards will be given if the agent goes backward, or turn (this will be defined in the `AliceMaze` environment).
Example program `AliceMazeExample.java` shows an effective learning for this task, and displays the behaviour of the agent after learning for 100.000 episodes (reward for an episode is already positive after 20.000 episodes).

10.4.2.4 AliceDistanceState

The agent is very similar to the one defined by `AliceState` and `AliceMaze` environment, except that it can see at a distance of two squares in each of the four directions. Uncomment line 18 in program `AliceMazeExample.java` to see how to use it and how the agent can learn in a `AliceMazeDistance` environment, compared to an agent defined in an `AliceMaze` environment.

10.4.2.5 AliceContinuousDistanceState

This class is a further generalization of class `AliceDistanceState`, where the agent has information about the distance between it and the walls in all four directions.
It is used within the `AliceMazeContinuousDistance` environment. Uncomment line 20 of example program `AliceMazeExample.java` to explore learning in this settings.

10.4.3 Environments

There is not much to say about the maze environments, as each of them is associated with a previously seen type of state and action. Let us just recall the relations between environments, states and actions:

Environment	State	Action
Maze	MazeState	MazeAction
LocalVisionMaze	LocalMazeState	MazeAction
AliceMaze	AliceState	AliceAction
AliceMazeDistance	AliceDistanceState	AliceAction
AliceMazeContinuousDistance	AliceContinuousDistanceState	AliceAction

Figure 10.3 shows the relations between the different classes, actions and states representing mazes in Piqle.

10.5 Mobile robot simulation

This environment is meant to simulate in a little bit more realistic way the problems involving autonomous mobile robots like Khepera or Alice, and study their behaviour. Robots and sensors descriptions are closer from the real ones, and the physics of the environment are more concretely designed.
Let us give some idea on how the environment is conceived :
A mobile robot is made of *actuators* and *sensors*, and wanders into an *environment* :

Actuators : all the parts that make the robot move or act. At this time, *actuators* are limited to one motor and a steering wheel. Hence actions are braking, accelerating, turning. One can imagine to add a gripping arm, a rotating camera turret...

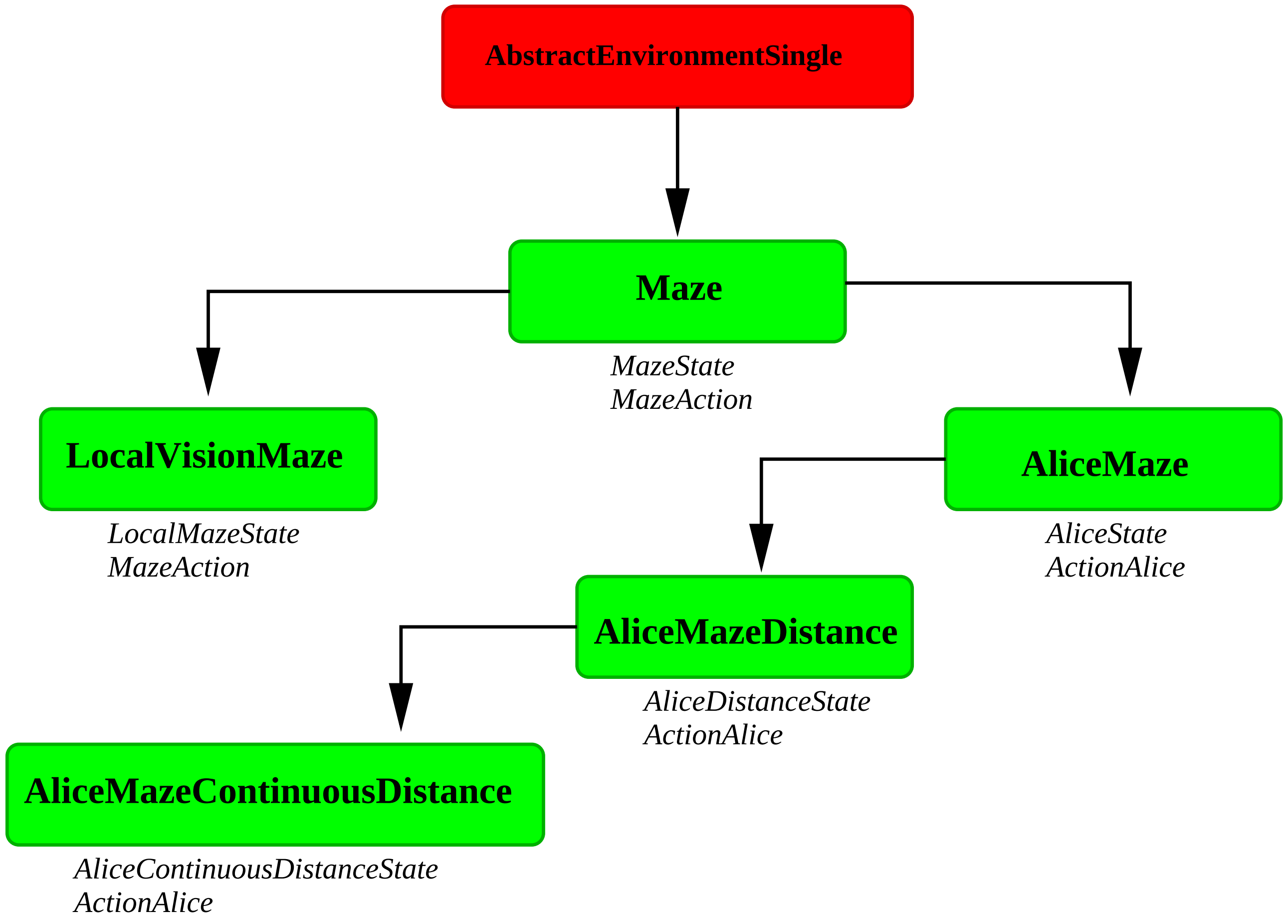

Figure 10.3: Maze environment hierarchy

Sensors Any kind of sensors can be defined, their outputs make the robot's perception and are to be considered as parts of the state description.Calculating the output of a sensor is a task left to the environment. This output can be continuous or discrete (mini-robots like Alice work with bytes).

Environment Only the environment can use the coordinates and the angle the robot makes with one of the coordinates axes to compute sensors outputs and the new position of the robot.

10.5.1 Actions

As in our examples so far, the only actuators are the motor and the steering wheel, a `MobileAction` has two important fields which define the commands applied to the mobile robot :

Speed Indicating whether to brake, accelerate or keep a constant speed. hence field `speed` is defined as an three valued integer.

Steering wheel We defined a discretized description of the steering wheel action, as a seven valued integer. Note that Piqle does not allow continuous valued actions.

Both definitions could have been generalized, to allow more choices for the speed behaviour, for example. Enhancements might not be difficult to program.
Note that adding an actuator mainly consists in adding an integer field, taking care of using it while defining the physics of the environment, and the description of the state of the robot...

10.5.2 States

A `MobileState` contains two different kinds of information :

Position and direction Those informations are necessary for the environment to compute the next state of the robot, but are not visible for the robot itself : those are the fields XCoordinate, YCoordinate and angle.

Speed and vision Informations provided by the robot's sensors : its eyes and the speed counter.

10.5.2.1 Speed

Speed is a positive or null integer, with a maximum value of `maxSpeed`. `maxSpeed` can be tuned using normal getters and setters.

10.5.2.2 Robot's vision

In this robot simulation, the only sensors are distance sensors. The number of sensors, their directions of vision, are defined into the associated environment (see below), but we can also give an overview here :

A robot has n distance sensors, uniformly spaced, with an angle of α, and symetrically placed with respect to the front of the robot.
n is called `retinaSize`, defined in abstract class `Circuit`, and its default value is 3.
α is called `angle` in abstract class `Circuit`, with a default value of 10 degrees. Both parameters can be changed and inspected through abstract class `Circuit`'s getters and setters.
What defined a state, from the point of view of the mobile robot, are the values of its sensors. Those values are stored into arrays of dimension `retinaSize`:

distances which stores the distances in each directions as real numbers.

discreteDistances which discretizes those values, into *far*, *medium* and *near* values.

Those distances are computed when a new state is created. The discretization method which transforms a `distance` into a `discreteDistance` is hard coded at this time (`MobileState` ligne 78): one may think on a more flexible definition...

10.5.3 Environments

Basically, the environment associated with a mobile robot is a closed arena with obstacles. The primary goal when conceiving this example was to make a robot learn how to turn round into the circuit as far as possible : the difficulty is to see when the robot passes the arrival line...
Hence, as a first approach, robots are asked to learn to stay as long as possible inside the arena without hiting obstacles, while running as far as possible.
The arena is defined as a couple made of :

External wall named `outside`

An internal wall named `inside` : this internal wall can also be seen as a set of obstacles (eventually just one !)

The walls and obstacles are of class `Polyline`, defining a (set of) closed curves. Class `Polyline` only contains methods to draw it, to see whether a given robot (state) is inside or outside it, and methods to compute the minimal distance between a point and the `Polyline`. It is up to the programmer to define functions computing the distance between a mobile robot and a wall in a given direction.
Two different arenas are defined, both are implementations of abstract class `Circuit`:

`CircularCircuit` A circular road, made of two concentric circles.

`Pinball` A circular arena with circular obstacles (*Bumpers*) inside.

Quite all methods are defined in the abstract class `Circuit`, while `CircularCircuit` and `Polyline` mainly define the computation of distances as seen by the sensors.

Chapter 11

The Two Players Case

Two players games differs from one player games for two reasons :

1. The state obtained by one player after applying an action is not always the same, as it depends on the opponent's move : a new referee must be defined (`TwoPlayerReferee`), which informs each player of its opponent's move. When informed the agent can apply its learning algorithm.
2. More technically, the generic notion of State is a little bit different, as now we must know whom is the turn to play, and who is the winner.

11.1 Memory

Memory is a popular two-players game : cards are disposed face hidden on the table. At his turn, each player must flip two cards : if they match, he take them, otherwise he put them again, face hidden, on the board.
[ZP93] analysed this game, proving that there is a winning strategy.
The game is given in two flavors : one that try to maximize the number of cards won, the other trying to maximize the probability of winning : they only differ on the reward method.

11.1.1 ActionMemory

[ZP93] analysis shows that there are only three possible moves :

- Choosing two known cards, and thus give no new information to the opponent.
- Flip an unknown card, then a known one.
- Flip two unknown cards (which gives the maximum information to the opponent)

Thus `ActionMemory` is a standard example of limited choice action we met several times in other problems (jenga, acrobot,. . .)

11.1.2 MemoryState

A `memoryState` represents the actual state of the game, i.e :

- The list of known cards and their position on the board.
- The position of unknown cards.
- The actual score of each player.
- A boolean indicating who has to play next.

Equality of two `MemoryState` is based on the equality of each of those fields, while neural network coding only uses the number of known and unknown cards. Some refinement could surely be done there . . .

11.1.3 Game boards

Two implementations are given : `MemoryBoard`, which gives a positive (+1) reward in case the player wins, negative (-1) if it looses, and `MemoryBoardZP` where the reward is proportional of the number of pairs the player found. This shows how different variants of a game can be implemented, by solely changing the `getReward()` method.
Program `Memory.java` gives an example of game, where one can experiment the learning ability of the agents : the actual settings seem to make the agent prefer ties.

11.2 Tic-Tac-Toe

Implementing and understanding the implementation of tic-tac-toe is quite straightforward, and can serves as a simple example on how to implement a two players game. It is also useful for first tests of new algorithms. As the (state,action) space is still of reasonable size, one can also explore the $Q(s,a)$ values in order to see and understand what has been learned.

11.2.1 TicTacToeAction

A `TicTacToeAction` contains the coordinates (row and column) where to put the pawn. No tricks nor difficulties in this class. Note that a *one among nine* coding might also have been chosen.

11.2.2 TicTacToeState

The state memorizes the actual position on the board, i.e. the value of each square (void, white pawn or black pawn).
Those informations are stored in a two dimensional array. Of course, as it is a `AbstractTwoPlayerState`, it also has a field telling who must play.
For simplifying computation, we also memorize the number of turns already taken, even if this information can be computed from the position array.

11.2.3 TicTacToeBoard

No difficulty either in this class : possible actions are placing the right colored pawn into a void square, computing a new state is just adding last move to the board's configuration, and rewards are tuned to avoid learning ties, by giving penalties in this case.
Example program `TestTicTacToe.java` illustrates an agent which, when playing first, learns to win in presence of non-learning opponents, and to not loose in case its opponent has also learning capabilities.

Chapter 12

The Multi-Agent Case

12.1 Introduction

This chapter is a short presentation about the way we implement multi-agents learning schemes in Piqle. The next section will enumerate the basic ideas, the third section will detail the new general classes, the forth section will present an example of use. Finally, ection five introduces the more general framework of communicating or cooperating agents.

12.2 Basic ideas

The structure of Piqle is left quite unchanged by this extension towards multi-agent learning. Some new classes were added, none were changed. In the next section, we will describe more precisely each new class, but here, we only give the basic ideas behind this implementation.

- A `Swarm` is a new implementation of the class `LoneAgent` : it has a different kind of choosing/learning algorithm, and a different vision of the actual state of the environment.

  The basic behaviour of Piqle is maintained : in presence of a `IState`, the `IAgent` asks its algorithm to give it the `IAction` to perform. The `IEnvironment` computes the new `IState` and the reward. After this is done, the `IAgent` informs back its algorithm about the new `IState` and the reward.

  The differences are :

  - The `Swarm` asks each of its composing `ElementaryAgent`s to ask its own algorithm which elementary action it should perform.
  - All those elementary actions are collected all together into a Java collection named `ComposedAction`.

  - The `Swarm`, which builds this `ComposedAction`, asks the environment to compute the next state and the new reward. Remark that, even if it is not the case in the two short examples presented here, each `ElementaryAgent` could proposed an action of a different kind.
  - This reward is sent back to each `ElementaryAgent`, which in turn will give those informations to its algorithm, which then will be able to learn.

- As it can be inferred from the above behaviour description, the universe associated with a `Swarm` no longer has to return a list of possible `Actions`. There are then two kind of environments :
  - A general environment, visible from the `Swarm`, which defines initial state, computes the next state based on a state and a `ComposedAction`, computes rewards, and knows when an episode ends. This environment does no longer give the list of possible `Actions`.
  - Elementary environments (`ElementaryMultiAgentEnvironment`), tied to individual `ElementaryAgents`, which will be able to compute the list of possible `Actions`.
  - For practical Java programming reasons, and (perhaps) more theoretical questions about the way `ElementaryAgents` perceive their environment, `ElementaryAgents` are a little bit more than just `Agents` : they can *filter* the representation of a state to extract just what they can perceive of it.

12.3 The new classes

12.3.1 Package `agents`

This package contains two new classes :

- The class `Swarm`, which represents a group of `LoneAgents`.
- The class `ElementaryAgent`, which represents an element of a `Swarm`.

Those two new classes, as usual in Piqle, are independent of both the learning algorithm(s) and of the problem to solve. The only limitation, for the moment, is that `Swarms` can only (try to) solve one player puzzles : a `Swarm` is an extension of the class `LoneAgent`, but not of the class `TwoPlayerAgent`.

12.3.1.1 The class `Swarm`

As `Swarm` extends `LoneAgent`, only a few methods and fields had to be added or modified :

- The field `ListOfAgents` is a collection aimed to contain all the `ElementaryAgents` composing a `Swarm`. We have chosen an `ArrayList`, but other collections would work as well. A new method (`add`) helps adding an `ElementaryAgent` to this `Arraylist`.

- Situating an `Agent` into its environment, i.e. setting its initial state, requires now to set the same initial state to all the `ElementaryAgents` members of the `Swarm` : method `setInitialState` is then rewritten.
- Similarily, (eventually) resetting the internal states of an `Agent` at the beginning of an episode requires now a loop over the `ElementaryAgents`.
- Less anecdotic, asking a `Swarm` to choose the `Action` to perform is now a two steps procedure :
  - Ask each `ElementaryAgent` to choose its elementary `Action`.
  - Collect all those `Actions` into a new kind of `Action` containing all those elementary `Actions` : an instance of the class `ComposedAction`. Note that for the moment, we rely on the structure of `ArrayList` to make the correspondance between the i^{th} `Action` and the i^{th} `ElementayAgent` : this does not appear to be very safe, and a more sophisticated version of `ComposedAction` might be thought of.

  That is why the method `choose` is redefined in this class.
- Asking a `Swarm` to apply the `Action` (namely method `applyAction`) previously chosen is also rewritten into a two times process :
  - Asking the environment to modify the current state, according to the previous state and the `ComposedAction`, receiving the reward back from the environment.
  - Inform each `ElementaryAgent` about the new state and reward, asking them to learn. Note that for learning from old state and new state, `ElementaryAgent` have to *filter* the state, in order to extract from them the perceptions they are allowed to use.
- Extracting a dataset, useful for external neural network treatment, has not be thought about, at least for the moment. It is quite understandable, as learning does not take place in the `Swarm`, but into the sub-`Agents`.

12.3.1.2 The class `ElementaryAgent`

An `ElementaryAgent` is a kind of `Agent` which always apply a `Filter` to the state it considers : the complete state is perhaps not visible from the set of its sensors.
The other reason to apply a `Filter` before to consider a state, is that, from the point of view of an `ElementaryAgent`, the state must be connected with a local version of the environment (otherwise, the method `getListeActions` would be the one from the general environment, which returns nothing).

12.3.2 The package `environment`

Three new classes are added in this package :

- The class `ComposedAction` : to collect all the individual action choices of `ElementaryAgent`s.
- The abstract class `ElementaryMultiAgentEnvironment`, voiding some useless methods in the multi-agents framework, and defining the method `getActionList`.
- The abstract class `Filter`, to indicate how to filter a state.

12.3.2.1 The class `ComposedAction`

In this version, a `ComposedAction` is just an extension of an `ArrayList`, inheriting all methods to add, access, eventually remove elements from this collection. As said before, a more sophisticated and secure definition might be thought of. For code security reasons, we also voided the methods one normally will not have to call in the context of multi-agents learning.
In fact, the only rewritten method is `copy`, to ensure that the content of a `ComposedAction` will be copied. The `clone` method might do that, perhaps...

12.3.2.2 `ElementaryMultiAgentEnvironment`

The only role of this class is to provide to an `ElementaryAgent` the list of the possible actions from a given state.
Typically, this class contains only the (re)definition of method `getActionList` : please have a look at the provided examples to see this basic definition.

12.3.2.3 The abstract class `Filter`

The agent might see exactly the same state that the one provided by the `Swarm` : in this case, applying a `Filter` to a state is just copying the state, only changing the referred `Environment` (see class `MountainCarFilter`).
In other cases, what the `Agent` sees is just a part of the complete state.

12.4 Example

This example is an implementation of MAABAC, whose details can be found in [PDD04]. MAABAC is a modelisation of body members as a *swarm* of muscles. The goal is for the member to reach a target. In Piqle's implementation, each muscle is an agent, knowing only its contraction and the distance between the extremity of the member and the target.
Two example programs are given :

- `TestMaabac.java` : A four segments arm (i.e eight muscles) tries to reach a target at (3.0,0.5), while the arm's extremity is initially at (4.0,0.0). Each muscle is using a standard Q-learning algorithm to learn, and the graphical interface appears when the arm reaches the target 100 times in a row.

- `TestMaabacTiling.java` : a two segments arms tries to reach a target at (1.8,0.5), while its extremity is originally at (2.0,0.0). This time, the four muscles are governed by linear approximators using tile coding. Learning as to be tuned, as it can take a very long time ...

In both examples, the code might be clear enough to allow modifications (number of segments, target's position, learning scheme...) for different kinds of experiments.
Note also on this example that each muscle could have been linked to different kind of learning algorithm, and different local vision of a state (through definitions of `MuscleFilter`).

12.5 Communicating/Cooperating agents

For agents to be able to communicate/cooperate, they need to be aware of the other agents actions[1].
We define a new class of agents : `ElementaryCooperativeAgent`s, which :

- Maintains a list of *neighbours* agents.

- Have a perception of the environment composed of :
  - Their filter vision of the global environment (local state as before).
  - The last action of each of their neighbours.

The rest of Piqle is left quite unchanged by this new framework, and we mainly implemented two new *generic* classes :

`ElementaryCooperativeAgent` An extension of `ElementaryAgent`, which maintain a list of neighbours.

`ExtendedStateWithAction` that will represent the `ElementaryCooperativeAgent`'s vision of the environment : its perception of the current environment's state, and it's neighbours' actions.

We will now describe more precisely those two classes, and then explain how to instantiate and use them.

[1]Another approach would have been to define environments as objects containing agents, we did not follow this path.

12.5.1 `ExtendedStateWithActions`

An object of this class is composed of :

- A (filtered) state, corresponding to the agent's perception of the global environment.
- A list of actions : the last actions taken by the agent's neighbours. This list is implemented as an ArrayList.

Since, at the beginning of an episode, no neighbours actions are available, we must introduce the concept of *null action*, telling us that no action has been taken : hence a new abstract class, `NullableAction` is added to package `environment`. Thus we can assure that the *null action* will be instantiate : see `MuscleNullableAction` in package `maabacVersion2`.
A lot of methods can be described at this level, even before instantiation, provided we know how to use `IState` and `NullableAction` methods. Thus `ExtendedStateWithActions` is declared abstract and generic, depending on two generic classes :

- F extending `IState` for the agent's local vision of the environment.
- E extending `NullableAction` : to represent all the agent's neighbours actions.*Note that in this definition, all neighbours or elementary agents must have the same definition of actions : this might be a little bit restrictive and could probably be generalized to heterogen sets of actions...*

By looking inside `ExtendedStateWithActions` code, it might be clear that constructors and methods `hashCode,equals(),nnCoding(), nnCodingSize()` can already be written at this level, lightening the code for inheriting classes.
A little trick, defining a fake method `buildState()` that must be overridden by inheriting classes allows even more generalization, leaving only constructors and `buildState()` to be written : see how`ExtendedLocalVisionMaabacState` in package `maabacVersion2` extends `ExtendedStateWithActions`.

12.5.2 `ElementaryCooperativeAgent`

In a similar way as the one used above, we can define a generic class, depending on the classes of both the `ExtendedStateWithActions` class, and the `NullableAction` class, to define quite all the methods of an `ElementaryCooperativeAgent` , letting only parametric class instantiations, constructors and `buildState()` method to be written in the inheriting classes. Here is how this class is build :

- An `ElementaryCooperativeAgent` maintains a list of its neighbours : the main program has to add those neighbours one by one, using method `addNeighbour()`.

- Once all neighbours have been added, the user must inform the agent about it, so it can build its initial state, by combining its perception of the global environment state and the last actions of its neighbours.
- If one tries to launch the experiment while having forgotten to *close* the list of neighbours (using method `buildInitialComposedState()`), the program will stop, displaying an error message :
```
Can't set state before list of neighbours completed
        Must call buidInitialComposedState
```
- Of course, at the beginning of the experiment, no agent has already moved, so their first action is a non-action : we then had to define a new class of actions : `NullableAction` which always defines an instance representing a *Null* action.

12.5.3 Example of use

Package `maabacVersion2` and sample program `TestMaabacV2.java` are illustrating how to use this new framework. Compare `TestMaabacV2.java` with `TestMaabac.java` to see the differences of use. All maabac's package classes were rewritten in package `maabacVersion2`,in order to avoid confusion, but few are really different.
Elementary cooperative agents are of class
`MaabacElementaryCooperativeAgent`.
Global states (`MaabacStateV2`) are identical to `MaabacState`.
Extended states (states + neighbour actions) are of class
`ExtendedLocalVisionMaabacState`.

12.6 Conclusions

We believe those approaches of implementing multi-agents learning schemes in Piqle are quite natural and easy to use.The way we face the general multi-agents problem seems very general and flexible to embed a lot of standard multi-agents situations, and to prepare to introduce other schemes.

Chapter 13

Taming the Octopus

13.1 Introduction

We describe in this chapter the successive steps we performed in order to connect Piqle with the octopus arm simulator used in ICML 2006 "Reinforcement Learning Competitition and Benchmarking Event" workshop.
We would like to show that Piqle allows fast testing of standard learning schemes, as well as quick and easy prototyping and evaluation of new learning schemes. On the other hand, we show that the communication protocol used in the Octopus Arm Simulator, through the use of sockets, is very easy to understand and to connect to Piqle, giving access to an often useful graphical interface.
Separating agents, environments and algorithms is the primary concept of Piqle, it is also this philosophy that is at work in the simulator : that's perhaps why the connection between those two programs is so easy to set up.
The rest of the chapter is divided as follows : the second section describes the Octopus Arm task. In the third section, we explain our multi-agent approach of the problem, as far as the Octopus Arm problem is concerned[1],more precisely how the octopus arm is divided into a number of individual muscles, working all together in order to move (hopefully correctly) the octopus arm. Finally, the fourth section will deal with conclusions and perspectives.

13.2 The Octopus Arm Problem

The Octopus Arm environment is described at the following url :
`http://www.cim.mcgill.ca/~dcasto4/octopus/`
where one can also found a configurable simulator, and a more precise description.
Roughly speaking, the octopus arm is divided into n segments, each segment's state is defined with eight real numbers. Two more real numbers are linked

[1]the in-depth presentation of how Piqle implements multi-agent problem can be found in chapter 12

to the base of the arm,which is anchored at a fixed location : those numbers express the rotational motion of this base.
One can command the octopus arm by :

- Send orders to a particular segment : this type of actions is defined with three real numbers in $[0, 1]$
- Apply forces on both ends of the arm's anchorage, which uses two real numbers.

One can play with a number of parameters, for example the number of segments, but also define the type of task one wants the system to perform or learn :

- The task can be a *Target Task*, where the arm must touch (or avoid) a serie of targets.
- The task can be a *food task*, where the goal is to push food particles into a *mouth*. For this type of task, the simulator uses four real numbers to describe the position and speed of each food particle.

So, a state vector for a N segments octopus arm in a food task with P food pills has a size of $2 + 8 \times N + 4 \times P$, while the action vector is of size $3 \times N + 2$.
The simulator is implemented using a system of sockets, so that agent and environment communicate by mean of exchanging vectors of numbers :

- The size of the state vector and of the action vector are sent to the agent at the beginning of the experiment (from an xml file).
- At each step of an episode :
  - The simulator sends the current state description, and the reward.
  - After that, the agent sends to the environment the action vector.
  - With this action vector, the simulator computes the new state, and the loop continues.

13.3 Connecting Octopus with Piqle

13.3.1 Octopus

An agent wanting to communicate with the Octopus Arm Simulator must implement four main methods, constituing the definition of an `ExtendedAgent` interface (defined in the Octupus Arm Simulator release):

init : where the agent receives basic information about the problem to solve (size of the octopus, task to perform).

start : to call at the beginning of each episode : returns the first action of the episode.

step : Receiving the current state description and the reward for the last step, computes the next action to perform, and learn (from state, penultimate action, and reward)

end : Instructions to be executed at the end of an episode.

endWithState is a variation of **end**, where the simulator indicates which was the last reached state.

We defined three classes extending this `ExtendedAgent` interface :

ICMLAgent A basic learning agent, which saves itself ones every 100 episodes.

ICMLExplore An agent that explores its environment by turning round, before starting to learn.

ICMLread An agent initialized by reading a previously saved agent : this allows to stop the program and start it again later.

13.3.2 Piqle

The basic element in Piqle is an`agent`: we then decided to incorporate one of those `agents` into the above `ExtendedAgent`, making the `ExtendedAgent` wrap our Piqle `agent`. We are then able to use some useful standard Piqle's features, like saving and reading an already trained agent, for example.
A number of points need still to be defined :

- Octopus actions are vectors of continuous values, but Piqle can only handle discrete actions.
- The state space is so huge that no single agent will be able to learn from Piqle.

To try to make the problem solvable, we decided to implement the `agent` by mean of a `Swarm` of `ElementaryAgents` : the `Swarm` will communicate with the environment in both ways, playing the role of an interface between the octopus arm and the group of `ElementaryAgents`.

13.3.2.1 Actions

Defining an action as a vector of real numbers poses two problems :

- The space of actions is too huge to be explored fruitfully by reinforcement learning algorithms.
- Piqle can only handle discrete finite sets of actions.

We then split the problem into more elementary problems :

- Instead of having a big powerful agent manipulating action vectors of size $3 \times N + 2$, we define $3 \times N + 2$ elementary agents, each one being in charge of one real number.

- Each real number is discretized : the discretization step can be fixed inside the `OctopusAction` class.

13.3.2.2 Elementary agents

Each elementary agent is in charge of one component of the original action vector. It can only see a part of the global environment. The part of the environment it can see is defined by its `filter`. A `filter` takes as parameter the global environment, and extracts the data the elementary agent is allowed to know.
To allow a minimal communication between agents, our implementation of the `octopusFilter` not only extract some informations from the environment (i.e. the state vector), but also from the previous actions of neighbouring agents (i.e the action vector).
As, in the case of the Octopus Arm Simulator, contiguous values inside both state and action vectors correspond to contiguous segments and agents, the `OctopusFilter` is set to cut slices into those two vectors.
In addition, in the case of a *food task*, information concerning the pills positions and speeds are given to the agents controlling the segment at the free extremity of the octopus arm.

13.3.2.3 Swarm

In *pure* Piqle, agents are dealing with state and sction objects which can be any kind of Java objects. When interfacing with the Octopus Arm Simulator, states and actions are only real vectors.
We thus define the new class `SwarmICML`, which is meant to return a real vector when asked for an action, and uses directly state and action vectors when asked to learn. Learning itself, for the `SwarmICML` is just asking its elementary agents to learn. `SwarmICML` plays here the role of an interface.

13.3.3 Putting things together

`ICMLAgent` is an agent in the sense of the Octopus Arm Simulator : it implements the methods needed for communicating with the simulator (start, init, step...). It contains itself an agent in the Piqle's meaning, and this agent is itself a `Swarm`, composed of $3 \times N + 2$ agents.
Let's describe more precisely each part of the `ICMLAgent`.

13.3.3.1 Declarations

We define a number of fields to :

- Monitor the experiment (size of episode, reward per episode, index of the current episode...)
- Store actions and states values.

- Pre-allocate room for learning algorithms.

13.3.3.2 Init

- As this method has for parameters the size of both state and action vectors, we can allocate memory for those vectors.

- We can compute the number of arm segments, as well as the number of eventual food pills.

- Thus we know whether the task is a *target task* or a *food task* : in the second case, we will have to communicate the positions and speed of the food pills to the last segment : this can memorized in the `upperLimit` field.

- We indicate in how many intervals the action values will be discretized : `OctopusAction.init`

- The rest of the method defines all the elementary agents :

  – their local environment, which role is only to return all the possible actions.

  – Their learning algorithms, which do not need to be all identical.

  – Their range of vision (`OctopusFilter`).

13.3.3.3 Start

Returns the first action, memorizes it, in order for agents to use the last action of their neighbours to learn at next steps. Each time the experiment accomplishes 100 episodes, the agent is saved on disk.

13.3.3.4 step

- Receives and dispatches state and reward information.

- Asks the agents to learn.

- Returns the next action composed of each elementary agent action.

13.3.3.5 end and endWithState

Like `step`, but prints statistics for the episode, instead of computing a new action.

13.4 Practical set up

- Obtain and install the Octopus Arm Simulator :

  `http://www.cim.mcgill.ca/~dcasto4/octopus/`
- Obtain ICMLPiqleVersion2.tgz, at sourceforge :

  `http://www.sourveforge.net/projects/piqle`

  copy it into the Octopus Arm Simulator `agent/java` directory, uncompress it.
- Compile ICMLAgent.java :
  ```
  javac -cp .:java-agent-handler.jar ICMLAgent.java
  ```
- Launch the environment and the agent, as described in the Octopus Arm Simulator documentation.

You are now ready to experiment, modify parameters, learning algorithms, learning strategies. . . .
Two others agents can also help you :

- `ICMLread` shows how to load a previously saved agent, and read back some of its characteristics.
- `ICMLExplore` is a non-learning agent that only explore its surroundings. We were hoping that this strategy could help initializing the learning schemes : this is not the case at this point.

13.5 Conclusion

The most obvious goal of this chapter is to explain how to use Piqle together with the Octopus Arm Simulator, and how to design experiments which can be monitored through the simulator graphical interface.
The tasks that were used in the competition were too hard to learn for our agent : easier tasks (target tasks were the targets are not too far from the initial position of the arm) are more interesting, even if learning doesn't seems to be straightforward.
Our agent can surely not pretend to be a good example of intelligent agent, but the way we defined it, and how it connects easily with the Octopus Arm Simulator illustrates, in our opinion, the power of Piqle, as far as design and test of learning schemes are concerned.

Chapter 14

Conclusion and Perspectives

14.1 Conclusion

We believe we provided a robust framework for reinforcement learning experiments design : not only from the point of view of implementing new problems, but also from the point of view of experimenting and comparing reinforcement learning algorithms. The object-oriented approach led to a modular and hierarchical conception, which allows quickly written modifications, enhancements, tests.
As shown in chapter 13 and at NIPS 2005 workshop, interfacing Piqle with other learning frameworks is easy.
We hope that Piqle users will enrich it by adding new problems and new algorithms.

14.2 Perspectives

A lot of roads are open :

- Add new conception of multi-agent learning schemes.
- Add the latest published algorithms.
- Add new problems (chess finals, backgammon, othello. . .)
- Integrate Piqle into a user-friendly graphical interface, where one could tune his experiment : choice of the algorithm, tuning of the parameters. . . and see graphically the learning curve.

Any idea will be welcome and could surely find its place inside Piqle.

Bibliography

[AS04] Masoud Asadpour and Roland Siegwart. Compact -learning optimized for micro-robots with processing and memory constraints. *Robotics and Autonomous Systems*, 48(1):49–61, 2004.

[PDD04] Ph. Preux, S. Delepoulle, and J-C. Darcheville. A generic architecture for adaptive agents based on reinforcement learning. *Information Sciences Journal*, 161:37–55, 2004.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. A Bradford Book.

[ZP93] Uri Zwick and Michael S. Paterson. The memory game. *Theor. Comput. Sci.*, 110(1):169–196, 1993.

[Zwi02] Uri Zwick. Jenga. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 243–246. Society for Industrial and Applied Mathematics, 2002.

Contents