

Multi-agents with PIQLE

Francesco De Comit 

June 9, 2006

1 Introduction

This text 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, while the fourth section will present an example of use.

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 **Agent** : it is designed to contain a number of individual **ElementaryAgents**, each one with its own choosing/learning algorithm, and with its own vision of the actual state of the environment.

The basic behaviour of **PIQLE** is maintained : in presence of a **State**, the **Agent** asks its algorithm to give it the **Action** to perform. After this is done, the **Agent** informs back its algorithm about the new **State**, and the reward.

The differences here are that :

- The **Swarm** asks each of its composing **ElementaryAgents** to ask its own algorithm which elementary **Action** it should perform.
- All those elementary **Actions** are collected all together in a 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 propose 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 (`ElementaryMultiAgentContraintes`), 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.

3 The new classes

3.1 Package agents

This package contains two new classes :

- The class `Swarm`, which represents a group of `Agents`.
- 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 `Agent`, but not of the class `Agent2Joueurs`.

3.1.1 The class `Swarm`

As `Swarm` extends `Agent`, 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.

- Similarly, (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 `choix` is redefined in this class.

- Asking a `Swarm` to apply the `Action` (namely method `applyAction`) previously chosen is also rewritten in 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`.
- While not rewritten for the moment, there might be some problems saving and reading a `Swarm`.

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).

Briefly speaking, `Filters` are necessary from the point of view of Java code writing.

As for the class `Swarm`, we have, at this time, no ideas on how to save a `Swarm` or its components, and we left the `saveAgent` and `readAgent` methods void.

3.2 The package environnement

Three new classes are added in this package :

- The class `ComposedAction` : to collect all the individual `Action` choices of `ElementaryAgents`.
- The abstract class `ElementaryMultiAgentContraintes`, voiding some useless methods in the multi-agents framework, and defining the method `getListeActions`.
- The abstract class `Filter`, to indicate how to filter a `State`.

3.2.1 The class `ComposedAction`

In this preliminary 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 my 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...

3.2.2 The abstract class `ElementaryMultiAgentContraintes`

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 `getListeActions` : please have a look at the provided examples to see this basic definition.

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 `Contraintes` (see class `MountainCarFilter` or `LedFilter`, for example).

In other cases, what the `Agent` sees is just a part of the complete `State` : see for example the LED example below, where an `Agent` can only see a diameter limited part of the bar of LEDs. In this situation, an `Agent` has to transform the current `State` into a more restricted `State`, according to its sensors (and also changing the global `Contraintes` associated with the global `State` to a local version of `ElementaryMultiAgentContraintes`).

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`).

5 Conclusions

We believe this approach of implementing multi-agents learning schemes in **PIQLE** is 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.

References

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