Programmer's guide of an intelligence test source code version 1.0

This is a simple documentation/guide of how the source code of a system of realization of tests is code. This source code is part of the project [ANYNT]. This is the version 1.0 of the system, developed by Javier Insa on April 14, 2011.

This document is composed of three sections:

In the first section we can see information about the project, what the project is about, what are the purposes of this software …

In the second section there is the documentation of the most important structures of the system, a diagram class of the whole system and the explanation of the most important attributes and functions of the important classes.

In the third section there is a simple guide of how to construct a new experiment: make new exercise from scratch, launch it, get the results and export them to a csv file.

1. Information about the project

In this project we try to make a test to evaluate the intelligence of several kinds of agents, i.e. humans, animals, IA systems, etc… This version of the system only supports two kinds of agents: humans (that can interact using a graphical interface) and IA systems.

This project has been built between March 2010 and April 2011. The project has 73 classes and 7592 lines of code which 46 classes and 2993 lines of code form the subsystem that we see on this document. Other classes form the interface, design patterns and other utils of the system.

The code has been written in Java 1.6 using Eclipse Helios environment and only needs the last Java Runtime Environment (JRE) to be launched. For development purposes the Java Development Kit (JDK) is needed.

In this program there exist two basic graphic modes:

A graphic mode designed to test humans, where only the needed information about the test is shown. We can see this mode in the image below.

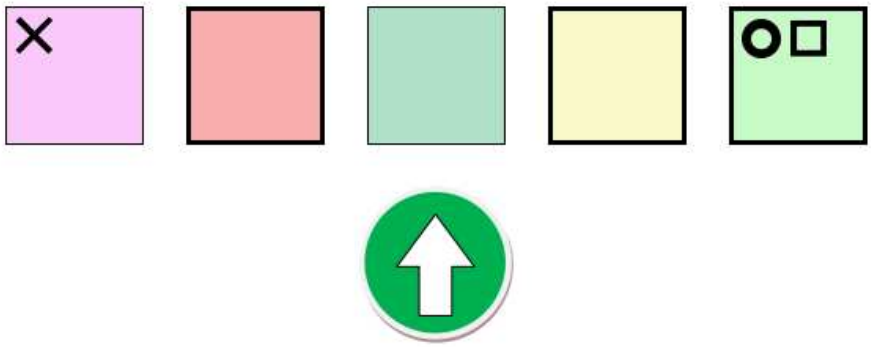

Illustration 1 - Human mode

A second mode has been created to help us to investigate and construct the system. This mode has a huge panel options from where we can indicate how we want the exercise.

This panel options has two sections.

The first one configures how the environment will be generated, where we can indicate some parameters of how the space will be generated and parameters about rewards generators.

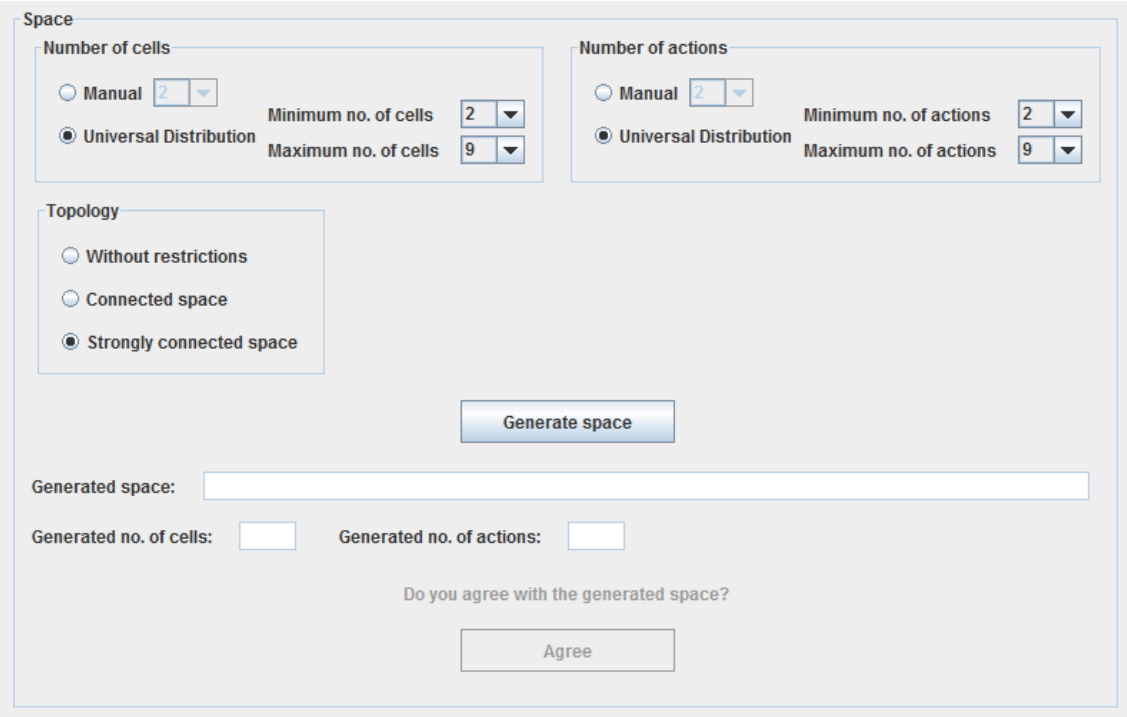

Illustration 2 - Space options

In this interface we can indicate exactly the number of cells and actions that will have the space or indicate a minimum and maximum value of these parameters and internally they will be calculated randomly using a universal distribution. Also the topology of the space can be specified so it can have no restrictions, ensure that the space generated will be connected or strongly connected.

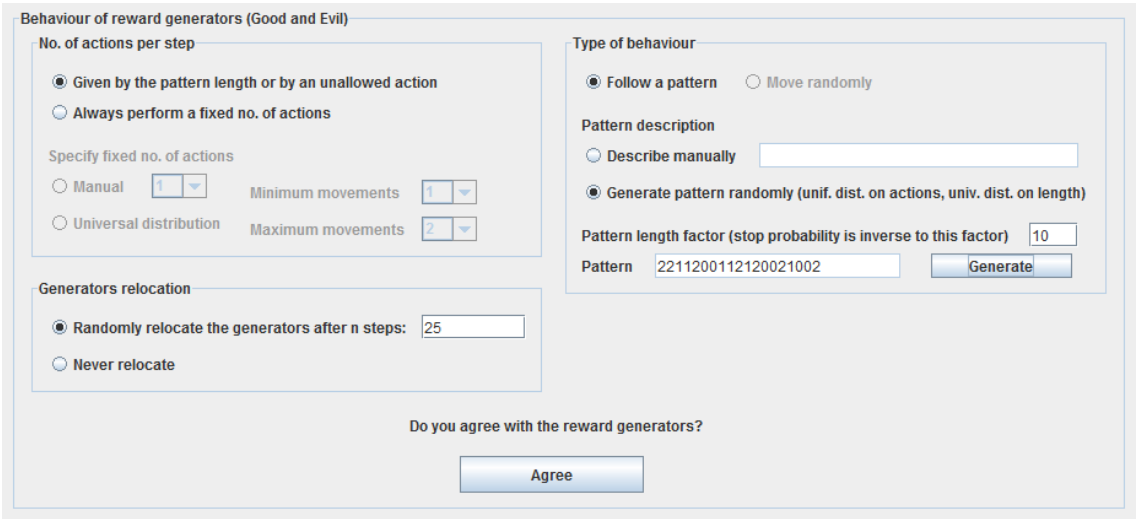

Illustration 3 - Rewards generators options

In this interface we can indicate exactly the behavior that will have rewards generators. It can be indicated how many moves they will make, if they will try to follow a pattern or move randomly, the pattern to follow if appropriated, when they will be relocated …

In this mode we see much more information in the exercise interface. We can see the entire topology of the space, rewards generators behavior, the last reward, the total rewards obtained so far, etc… We can see this mode in the image below.

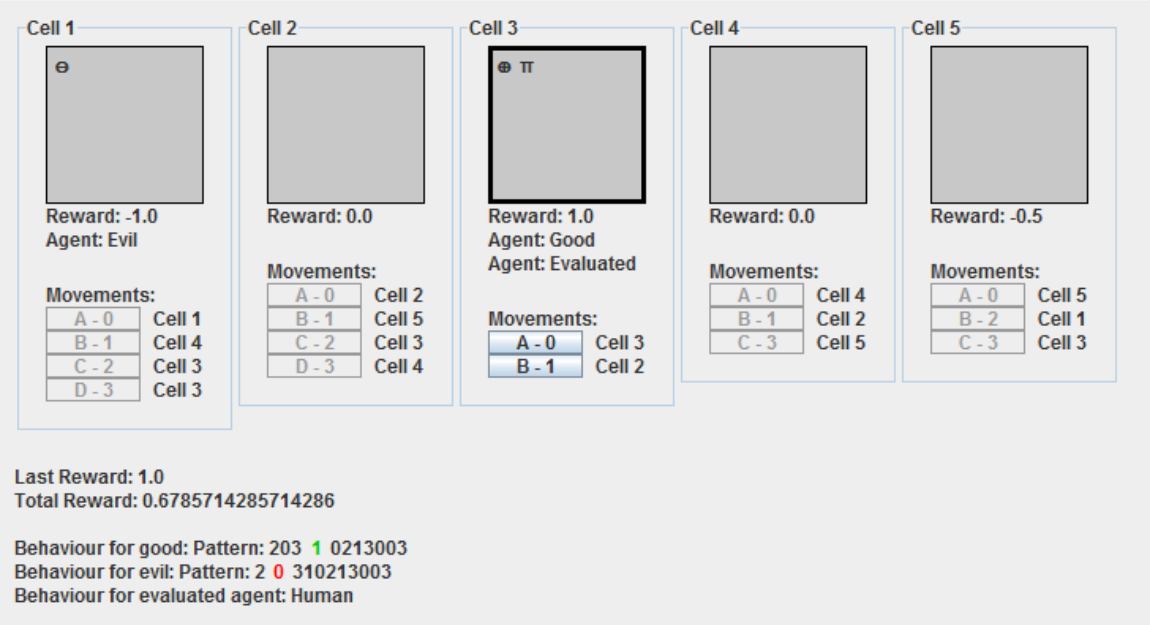

Illustration 4 - Investigation mode

Despite IA systems don't need any graphical support to interact with the system in the investigation mode we can see them in a graphical mode.

2. Documentation for the most important classes

In this section we explain some basic information about the most important classes. It's important to understand the structure of the system and the relations among all the classes in order to understand how the system is working. To make it easier here is the class diagram where we can see the structure and the relations of all the classes.

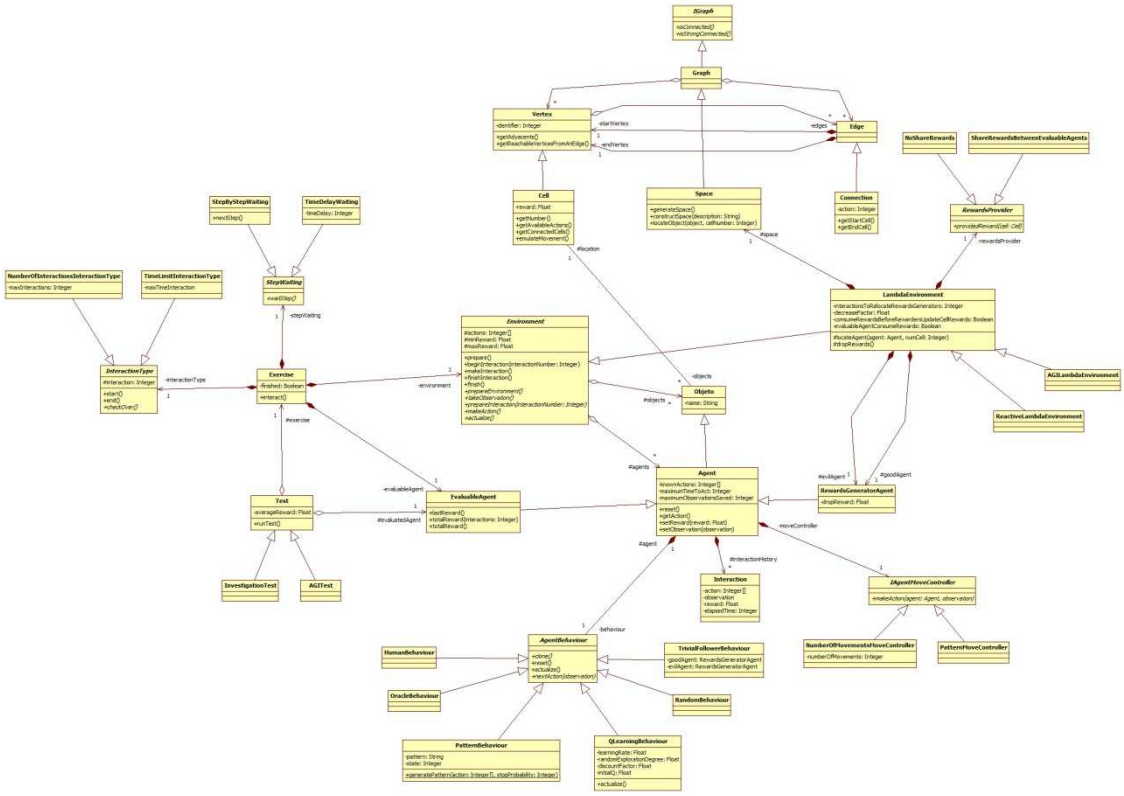

Illustration 5 - Diagram class of the system

The project is divided in several folders.

In the frontend folder there is the code of the presentation, where we can see two main packages: humanAGI (where are the classes needed to the presentation for the interface used to test humans for the AGI congress) and investigation (where are the classes needed to the presentation for the interface used to check if the system is working correctly and view how some agents interact with the environments).

In the business folder there is the code that really manages the test. It is divided in several packages, where the package anynt represent the test that is implemented in this project, so we can say that it is the most important package of the system and therefore it's the part that we explain in this document. This package is also divided in some packages: environment (there are the classes that represent all the kind of environments that can be managed), exercise (there is where an exercise is defined), object (these represent all kind of objects that

can be introduced in the environment), space (this is the space used in the lambda environment) and test (where are represented all kind of tests that can be made).

And finally in folder persistence are the classes that work directly with the files on the computer, i. e. images, sounds and write on a file the results of an experiment.

There are 6 important classes where all the system lies.

These are Test, Exercise, Space, Objeto, Agent and LambdaEnvironment, being these two last the most important of the system.

In the following pages these and other classes are explained for what they serve and their structure (Attributes and functions).

Test

This is the test that will be launched, it should have a list of exercises that will be generated when needed, but at the moment there is no need to save the last exercises, so only is saved the actual exercise.

Here we can see the Test with their relations.

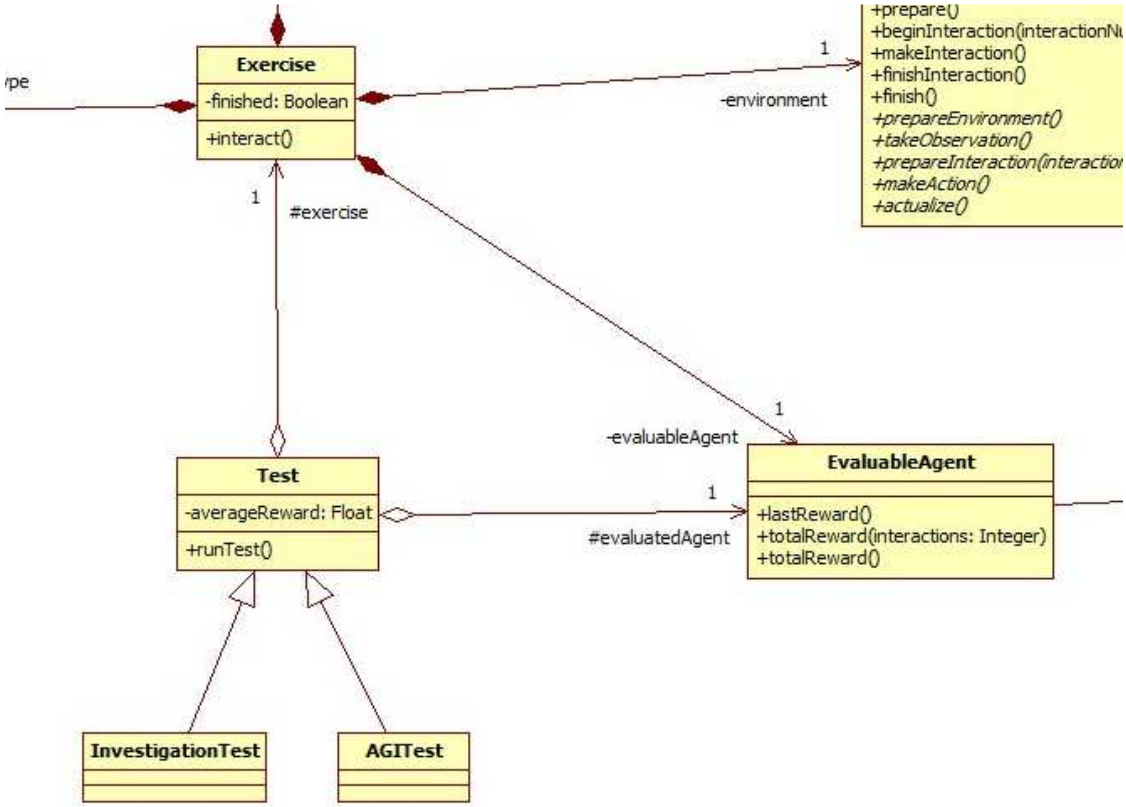

Illustration 6 - Test diagram class

Semantics of relevant attributes

Exercise – Actual exercise that it's launched in the test.

Evaluable agent – This is the agent that is evaluated in the test.

Average reward – This is the average reward of the agent between all the executed exercises.

Semantics of relevant functions

Start – The exercise is executed in a thread, this method starts the interaction.

Run Test – It starts the test

There is a simple hierarchy of functions to prepare, launch, finish ... the test and the exercises. To modify this hierarchy ensure first that it’s clear how is it mounted.

Default when creating the object

Nothing is set by default when the object is created, all the parameters must be defined before running the interaction.

Exercise

This represents one exercise of the test. It's the responsible to make the interaction between the environment and the agents. To make the interaction with the agents easier the environment is communicated directly with the agents, so the exercise delegates this functionality directly to the environment.

Here we can see the Exercise with their relations.

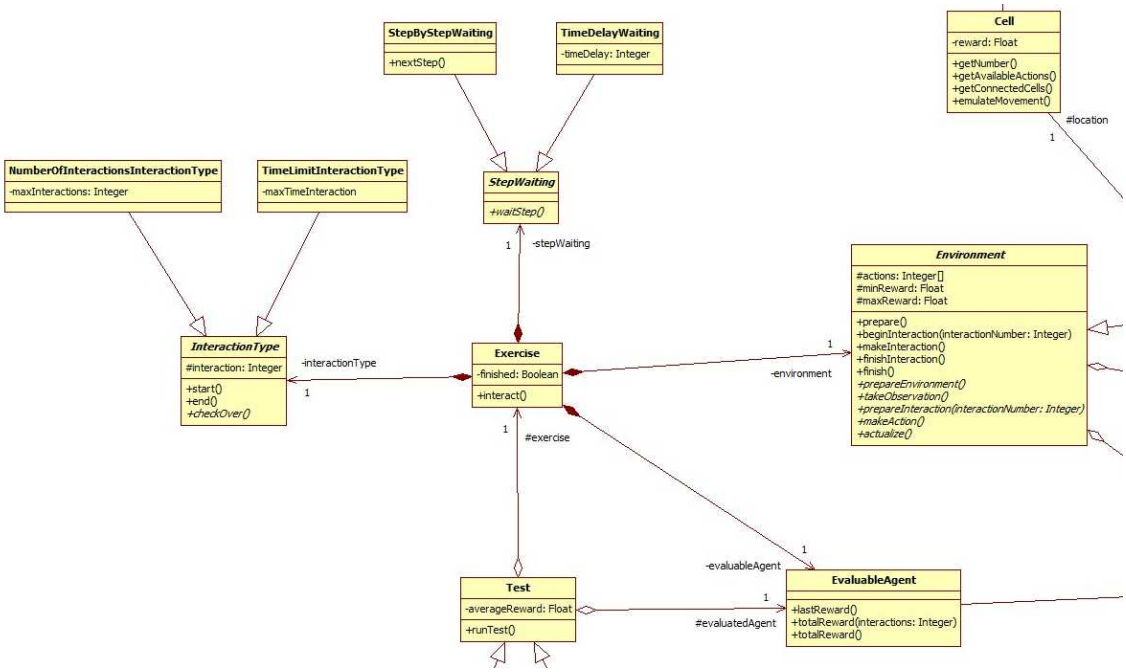

Illustration 7 - Exercise diagram class

Semantics of relevant attributes

Environment – This is the environment where will be done the exercise.

Evaluable agent – This is the agent that will be evaluated in the exercise.

Interaction type – This indicates when the exercise will be over. If it is set to NumberOfInteractionsInteractionType class then the exercise will be over when it is reached the number of interactions that is set in this class. If it is set to TimeLimitInteractionType class then the exercise will be over when it is reached the number of seconds set in this class.

Step waiting – When the evaluated agent is not a human, it could be useful to see in the interface how the interactions are happening. This class controls if the environment will wait or not between steps. If it is set to StepByStepWaiting class then the system will be waiting until you tells him to continue (with a button in the interface for example). If it is set to TimeDelayWaiting class then the system will wait the number of seconds indicated in the class (for no waiting between steps put this last class with time delay set to 0).

Semantics of relevant functions

Start – The exercise is executed in a thread, this method starts the interaction.

Interact – Starts the interaction between the environment and the agents.

Default when creating the object

Nothing is set by default when the object is created, all the parameters must be defined before running the interaction.

Lambda Environment

This kind of environment is an approximation of the environment defined in [Hernández-Orallo, 2010].
Summarized in this environment there is a space where agents will interact, there are defined 2 special agents named Good and Evil that are responsible to drop the rewards in the environment (more precisely in the space). Also objects can be placed into the environment. In this environment Good and Evil must have exactly the same behavior and cannot collide in the same cell.
Here we can see the Lambda Environment with their relations.

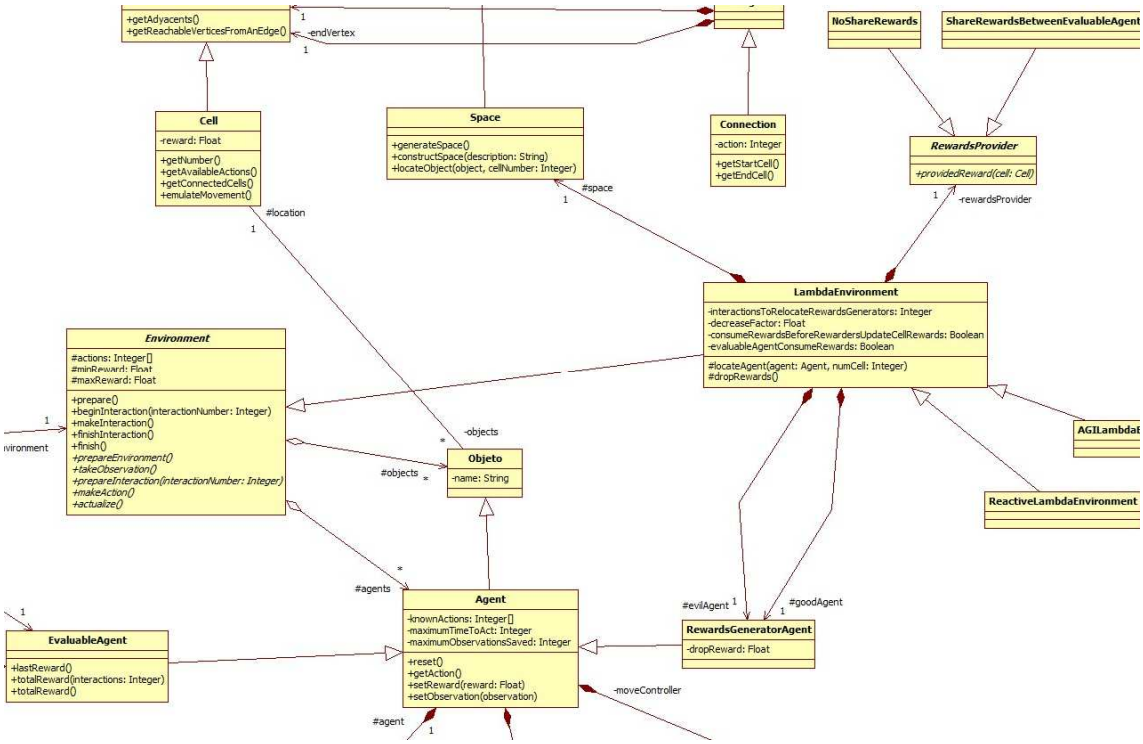

Illustration 8 - Lambda Environment diagram class

Semantics of relevant attributes

Actions – This are the actions that can be made in the environment, this actions automatically set when defining what is the space that will have the environment.

Agents – This are the agents that will interact in the environment. This agents can be reward generators, evaluable agents or simple agents.

Object – The system is prepared to work with objects (in fact all the agents are also objects) and, if needed, there can be implemented the required methods to work with them.

Space – In this space is where all the agents and objects will be located and will move. Here are located all the agents: GoodAgent, EvilAgent the EvaluableAgent…

Good Agent – This is the agent that will leave good rewards in the space. This agent is partially auto generated when the environment is created.

Evil Agent – This is the agent that will leave bad rewards in the space. This agent is partially auto generated when the environment is created.

Interactions to relocate rewards generators - After a fixed number of steps good and evil are randomly relocated on the space. If this value is set to 0 they are never relocated.

Decrease factor - This number indicates the number for which will divided the rewards after each step, if this number is set to 1 they will no decrease and if it is set to Integer.MAX_VALUE they will be set into 0 on next step.

Consume rewards before rewarders update cell rewards - This indicates when the rewards are consumed. If it's set to TRUE (before) then rewards are consumed by the agents and after all the rewards in the space are divided by the decrease factor and rewards generators leave their new rewards. If it's set to FALSE (after) then all the rewards are divided by the decrease factor and rewards generators leave their rewards and after rewards are consumed by the agents.

Evaluable agent consumes rewards - This indicates if the rewards are deleted from the space when an agent consumes it.

Reward Provider - This divides the rewards of a cell between all the agents that are in the cell. If it is set with a NoShareRewards class then rewards are not share/divided between the agents, so the entire reward is sent to all the agents that are in the cell. If it is set with a ShareRewardsBetweenEvaluableAgents class then rewards are shared/divided between the evaluable agents that are in the cell.

Semantics of relevant functions

Prepare – Prepares the environment to the interactions

Begin Interaction – It makes the first step of an interaction, such send the observation and rewards to the agents, and also some other preparations of the interaction like relocating the reward generators.

Make Interaction – It makes the second step of an interaction, all the agents say what their respective actions are.

Finish Interaction – The agent actions are made, there are calculated their rewards and the environment is actualized.

Finish – All the interactions between the agent and the environment are over. There are sent the last reward the agents because they had not received it yet.

There is a little complex hierarchy of functions to perform all the functionality mentioned above. To modify this hierarchy it's necessary to understand it well. When launching the environment, each class performs the operations to manage the attributes of his class, so if a class is the responsible of doing some operations it is who will make them and any child class will make this operations for him.

Default when creating the object

Space is not defined, it must be defined before starting the interaction.
Good and evil agents are partially created, by default with their creation they are not ready to interact with the environment.
Interactions to relocate rewarders is set to 0.
Decrease factor is set to 2.
Consume rewards are set to TRUE (before).
Rewards are deleted from the space on each step.
Rewards provider is not defined, it must be defined before starting the interaction.

Space

This is the space where all the agents and the objects will be located. The space is modeled by using a directed graph, where we have the cells (vertices) and the connections (edges). Each cell contains agents and objects, and connections (that contains the action that allows you to cross over it) allows the agents to move between cells. The agents will be able to move between cells by using the connections between them.
Here we can see the Space with their relations.

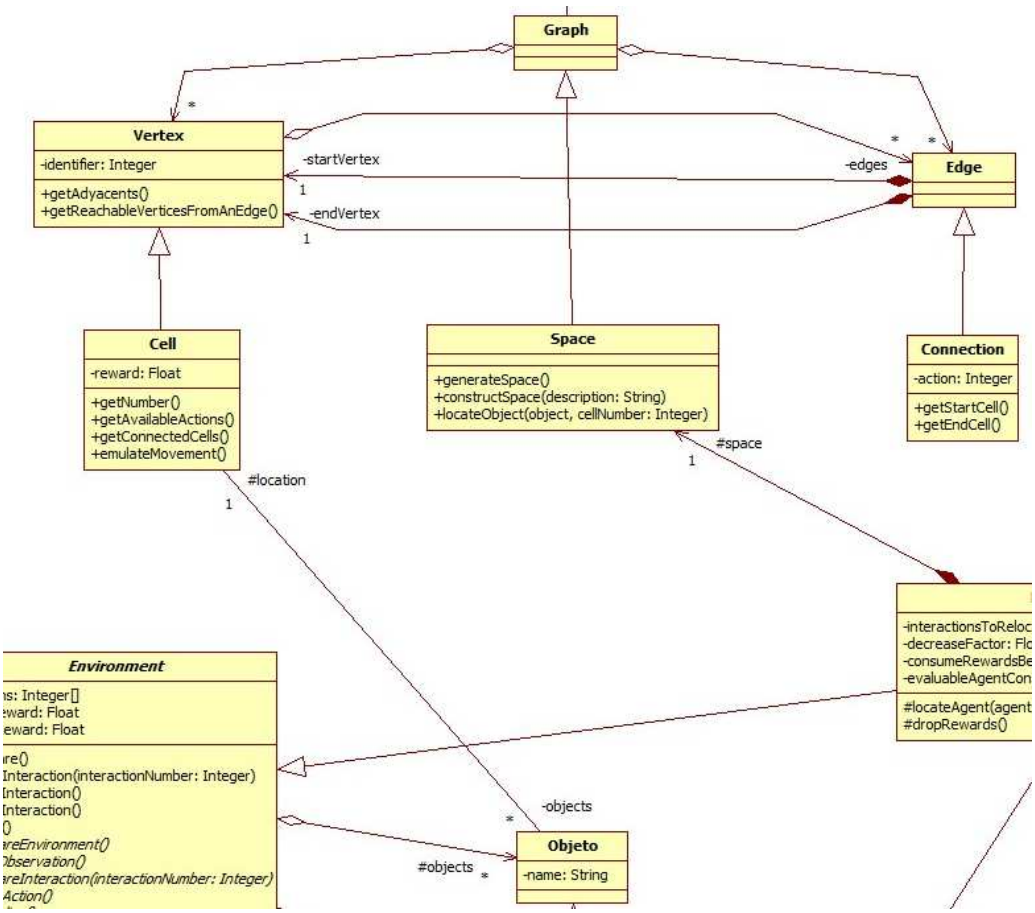

Illustration 9 - Space diagram class

Semantics of relevant attributes

Description – This is the description of the space in textual mode.

This is a little example of a description: (1++2-|1++2- -|1-2) The character | divides the space in cells, so here we have a space with 3 cells where cell 1 is defined by (1++2-), cell 2 by (1++2--) and cell 3 by (1-2). On each cell description we can see the actions available on the space (in this case there are only 2 actions) and the signs + and - indicates the displacement of the action. For the description of cell 1 we can see that the action 1 has a displacement of +2 and action 2 has a displacement of -1. The cell target of one action is calculated adding the number

of the cell with the displacement, for cell 1 and action 1 is 1+2= cell 3, and action 2 is 1-1= cell 3 (if the action tries to go out of the bounds continue cyclically).

Actually there is always an implicit action, action 0, this action always connects a cell with itself, and it's not defined in the space description. If an agent tries to perform an action that the cell doesn't have this is the action that is performed. Besides action 0 is always allowed to perform if desired.

Minimum/Maximum cells – Before creating a new space, this numbers indicates the bounds of the cells that will be created.

Minimum/Maximum actions – Before creating a new space, this numbers indicates the bounds of the actions that will be created.

Semantics of relevant functions

Generate Space – Generates a new space from scratch, generating its description and after creating the space following this space description.

Construct Space – It construct a new space following the given description.

Locate Object – It locates an object in the given cell, if the object was in another cell it is removed from the previous cell.

Is Connected – Indicates if the space generated is connected. This means that all cells are connected, so any cell is disconnected from the others.

Is Strong Connected – Indicates if the space generated is strongly connected. This means that every cell is reachable from another cell.

Default when creating the object

There is not any default space constructed by default when creating the object, it must be defined before the bounds of the cells and actions and after generate the space.
Minimum/Maximum cells are set to 2/9.
Minimum/Maximum actions are set to 2/9.

Objeto

This represents an object that can be placed on an environment (more precisely on a cell in the space).
Here we can see the Objeto with their relations.

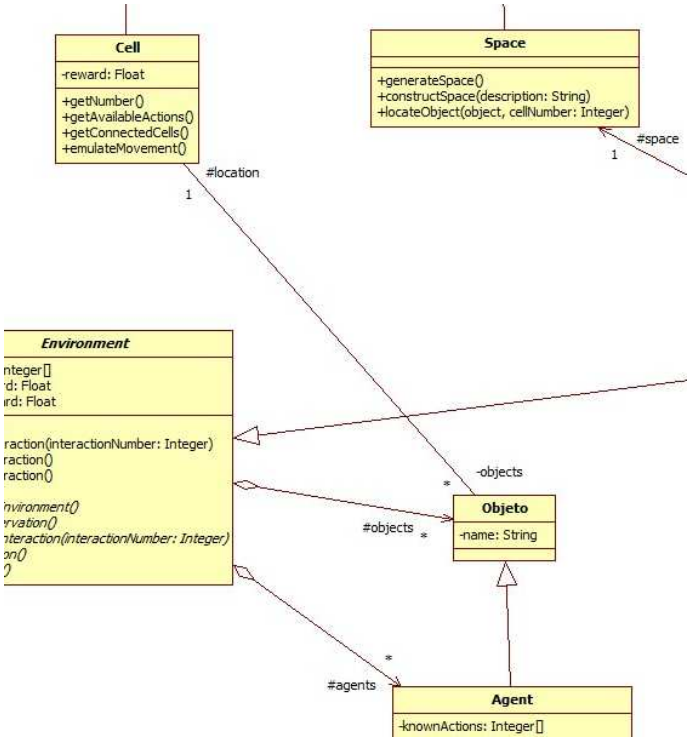

Illustration 10 - Objeto diagram class

Semantics of relevant attributes

Name – Name that will have the object

Location – Cell where the object is located in the space of the lambda environment

Semantics of relevant functions

Set Location – It changes the location of an object in the space

Default when creating the object

You indicate the name that the object will have.

The object is not set in any location.

Agent

Extends Objeto

This represents an agent, agents can move in each step in the environment (more precisely in the space). All agents have a behavior that tells him how they have to interact with the environment.
Here we can see the Agent with their relations.

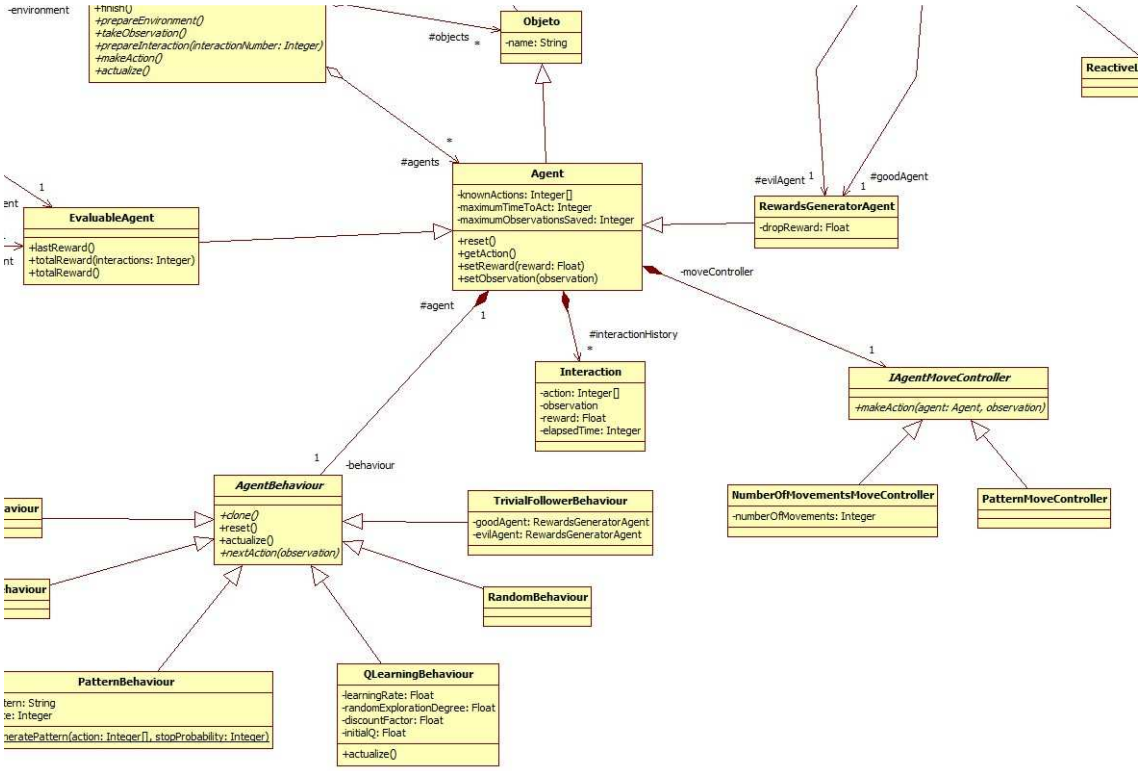

Illustration 11 - Agent diagram class

Semantics of relevant attributes

Move Controller – When an agent is interacting with an environment, it could want to make a fixed number of actions (NumberOfMovementsMoveController class), try to follow a pattern (PatternMoveController class), make always X actions which will always be available actions in the cells that it is passing through, make 5 actions and skip 1 action, …

Behaviour – This is the behavior that will follow the agent, it could be a random behavior (RandomBehavior class), try to follow always the Good agent (TrivialFollowerBehavior class), behave like a QLearning algorithm (QLearningBehavior class), have a pattern of actions to make (PatternBehaviour class), always move to the cell where the best reward is meant to be (it's not always possible to know what will be this cell) (OracleBehavior class), leave a human to behaves (HumanBehavior), …

Properly combining this two objects (MoveController and Behavior) is set the real behavior of an agent, an agent could want to make X random movements (NumberOfMovementsMoveController with RandomBehavior) or X movements of a pattern (NumberOfMovementsMoveController with PatternBehavior), it could also want to try to accomplish the entire pattern (PatternMoveController with PatternBehavior), … Obviously there are combinations that have no sense and will not work correctly.

To generating a pattern of actions that have sense on one environment, it needs the actions that are available on the environment. PatternBehavior has a static function (generatePattern(actions, stopFactor)) that takes these actions and a Stop Factor and creates a new pattern. This stop factor sets the probability to stop generating the pattern when a new action of the pattern is created.

Some behaviors need to know who are good and evil agents or what is the environment where they are interacting, simply pass it as parameters when creating.

All behaviors have the clone operation for easy copy.

Known actions – List of actions that the agent know to do.

Interaction History– This is the memory of the agent, here it can be seen all the interactions that the agent has seen in the whole exercise. The observation of the environment on a step is too big to store in memory, so it is somehow limited the number of observations that the agent will memorize (MaximumObservationsSaved attribute).

Maximum Time to Act – This is the maximum time that the agent has to act for each step. (This functionality is not fully implemented).

Maximum Observations Saved – This number indicates how many observations will be saved in the memory of the agent. If this number is set to 5, the agent will only memorize the last 5 interactions. If this number is set to 0 then the agent will memorize the whole exercise.

Semantics of relevant functions

Set Reward – Sends the reward of the last interaction to the agent

Set Observation – Sends the observation of the last interaction to the agent

Get Action – Takes the action that the agent will make in this interaction

Default when creating the object

Move Controller is not defined, it must be defined before starting the interaction.
Behavior is not defined, it must be defined before starting the interaction.
Known actions are not defined, they must be defined before starting the interaction.

The interaction history is automatically controlled by the agent, it don’t have to be managed by the programmer.
The agent will memorize the whole exercise.

Evaluable Agent

Extends Agent

This represents the agent that will be evaluated. If an agent will be evaluated it must be an instance of this class.
Here we can see the Evaluable Agent with their relations.

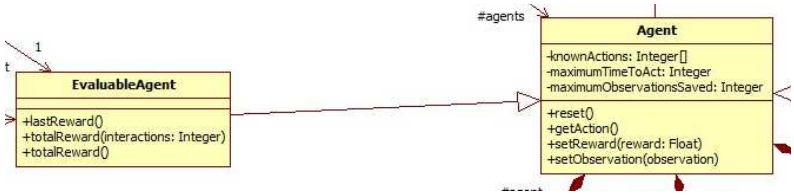

Illustration 12 - Evaluable Agent diagram class

Semantics of relevant functions

Last Reward – It returns the last reward given to the agent

Total Reward – It returns the average of rewards that the agent has had on the exercise so far.

Rewards Generator Agent

Extends Agent

This represents a special agent that drops rewards on the environment (more precisely on the space), these agents form part of the Lambda Environment and must be always on each exercise.
Here we can see the Rewards Generator Agent with their relations.

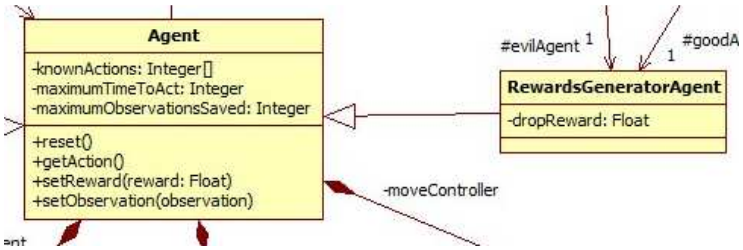

Illustration 13 - Rewards Generator diagram class

Semantics of relevant attributes

Drop reward – This is the rewards that the reward generator will drop into the cell where he is located on each step.

Semantics of relevant functions

Get Drop Reward – It returns the reward that the reward generator drops

Default when creating the object

You indicate the rewards that the agent will drop.

3. Construction and save an experiment

So far there is no interface to make experiments, so they are defined manually using code.

In this section is described a simple guide of how to construct, launch and save an exercise.

First of all it's necessary to create a Test, an Exercise, a Lambda Environment, a Space, and an Evaluable Agent. It is needed to relation all these objects between them. A space is in a environment, an environment is in an exercise, an exercise is in a test, the evaluable agent is an agent in the environment (environment.addAgent()), the evaluated agent is in the exercise and in the test (test/exercise.setEvaluableAgent()).

Because the environment has created by default Good and Evil agents you can get them from the environment (getGoodAgent() and getEvilAgent())

The three objects InteractionType, StepWaiting and RewardsProvider must be created and set to their respective objects. Exercise: Interaction type, StepWaiting. Environment: RewardProvider.

From here all the objects must be well relationed for a correct interaction. Now it's time to configure all the parameters.

There are two ways to create the space. It can be generated from scratch (generateSpace()) or created with a specified description (constructSpace(description)). When a space is generated from scratch it's frequent that it doesn't accomplish certain requirements, such not being strongly connected. To ensure these requirements, make a loop while they are not accomplished.

For example: do { space.generateSpace(); } while (!space.isStrongConnected());

All the agents must define their move controller and their behavior. Notice than even good and evil don't have these by default. Also agents must have their known actions defined, if they must know all the actions of the system simply set with environment.getActions() so they know the same actions that the environment.

Good and evil agents must have exactly the same behavior (MoveController and Behavior). So when constructing them the same object of MoveController must be set to either. Because the behavior of an agent must be exclusive to one agent, the function clone of behavior will give another object with the same behavior and this new object is set to the other reward generator.

It is strongly recommended to reassign the default parameters of all the objects to have the environment how you really want. For example: not infinity memory for agents, consume rewards after the update of the space rewards…

Before launching an exercise you can call the checkConsistency function in order to check if the exercise is consistent to be launched or, on the contrary, some relations or parameters have not been well instantiated. (exercise.checkConsistency()). If the exercise is not well built a RuntimeExecption will be launched showing where the problem was.

This is an example of a creation of a test

```
// The test
Test test = InvestigationTest.getTest();
// The exercise
Exercise exercise = new Exercise();
// The environment where we will make the interactions
LambdaEnvironment environment = new LambdaEnvironment();
// The agent that will do the interactions
EvaluableAgent agent = new EvaluableAgent("Agente");
// The space where agents will interact
Space space = new Space();
// Take the partially generated good and evil agents to configure them
RewardsGeneratorAgent good = environment.getGoodAgent();
RewardsGeneratorAgent evil = environment.getEvilAgent();

// Relate all the objects properly
test.setEvaluatedAgent(agent);
exercise.setEvaluableAgent(agent);
test.setExercise(exercise);
exercise.setEnvironment(environment);
environment.addAgent(agent);

// Set the interaction type, the step waiting and the rewards provider
// and relate them with their respective objects
InteractionType interactionType = new
NumberOfInteractionsInteractionType(1000000);
StepWaiting stepWaiting = new TimeDelayWaiting();
RewardsProvider rewardsProvider = new NoShareRewards();
exercise.setInteractionType(interactionType);
exercise.setStepWaiting(stepWaiting);
environment.setRewardsProvider(rewardsProvider);

// Generate a new space and ensure it has the desired properties
do { space.generateSpace(); } while (!space.isStrongConnected());
// Only after you have the desired space you can set it to the
// environment
environment.setSpace(space);

// Configure properly the three agents. Notice that they are already
// properly related with the environment, the exercise and the test.
// Here it's the configuration of the evaluable agent
IAgentMoveController moveController = new
NumberOfMovementsMoveController();
AgentBehaviour behaviour = new OracleBehaviour(environment);
agent.setKnownActions(environment.getActions());
```

```
agent.setMoveController(moveController);
agent.setBehaviour(behaviour);
// Here is the configuration of good and evil
String pattern =
PatternBehaviour.generatePattern(environment.getActions(), 10);
PatternBehaviour goodBehaviour = new PatternBehaviour(pattern);
good.setKnownActions(environment.getActions());
good.setMoveController(moveController);
good.setBehaviour(goodBehaviour);
// Notice that evil agent must have exactly the same behavior than
// good, so set the same move controller and clone the behavior.
PatternBehaviour evilBehaviour = goodBehaviour.clone();
evilBehaviour.setAgent(evil);
evil.setKnownActions(environment.getActions());
evil.setMoveController(moveController);
evil.setBehaviour(evilBehaviour);
// You can also configure all the parameters as you like
agent.setMaximumObservationsSaved(10);
good.setMaximumObservationsSaved(10);
evil.setMaximumObservationsSaved(10);
environment.setRewardsConsumedBeforeRewardersUpdateCellRewards(true);

// To start the test simply launch the start function.
test.start();

// You can also launch an exercise without being in a test, simply
// don’t create the test object and launch the correct method of the
// exercise.
// You can launch an exercise using the function “interact()”
exercise.interact()
// If you want to launch it using a thread use the function “start()”
exercise.start()


// When the interaction is over you can get the total rewards of the
// evaluated agent. Simply get the agent that it’s being evaluated (if
// you don’t already have it) and take the total reward.
EvaluatedAgent agent = exercise.getEvaluatedAgent()
agent.totalReward()



// Finally if you want to export the results of the execution of an
// exercise to a “csv” file, you can call the static function
// WriteExperiment located in the class ExperimentsFile from package
// files by passing as argument the exercise and the destination
// system file, if the file is not already created it will be created.
files.ExperimentsFile.WriteExperiment(exercise, filepath);

// This function only works if the environment of the exercise is a
// Lamba Environment, if there is another type of environment that
// want to be saved, feel free to create your own function.
```

This is a list of all the columns of the csv file by order of appearance

1. Number of space sells
2. Number of space actions
3. Space Description
4. Rewarders Actions per Step
5. Rewarders Fixed Actions
6. Rewarders Behaviour
7. Pattern Description
8. Pattern Length
9. Number of interactions when rewarders where relocated
10. What kind of evaluable agent was doing the exercise
11. Number of interactions made in the session
12. Consumption order of the rewards (before or after update)
13. Session Reward Decreasing Factor (2147483647 = Math.MAX_VALUE)
14. Evaluable agents consume rewards
15. Kind of rewards sharing
16. The total reward of the agent
17. All the rewards taken on each interaction, from first interaction to last interaction

Here we can see an example of the generated file.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	3	3	12-|1-2--|1++2--	N Movements	1	Pattern	p2210	4	0	Human	20	After	2147483647	Yes	No	0	-1	0	-1	1	0	-1	1	1	0	-1
2	3	3	12-|1-2--|1++2--	N Movements	1	Pattern	p2210	4	0	QLearning	20	After	2147483647	Yes	No	0,05	-1	1	1	0	0	-1	1	1	0	1
3	4	3	1-2+++|1++2--|1---2-|1+++2---	N Movements	1	Pattern	p0	1	0	Human	30	After	2147483647	Yes	No	0,933333333	0	0	1	1	1	1	1	1	1	1
4	4	3	1-2+++|1++2--|1---2-|1+++2---	N Movements	1	Pattern	p0	1	0	QLearning	30	After	2147483647	Yes	No	0,533333333	0	0	0	0	-1	0	0	1	0	0
5	5	3	1-2-|1++2-|1--2+|1+2++++|1++2	N Movements	1	Pattern	p1221101010222002	16	0	Human	40	After	2147483647	Yes	No	0,475	-1	0	0	0	0	1	0	1	0	1
6	5	3	1-2-|1++2-|1--2+|1+2++++|1++2	N Movements	1	Pattern	p1221101010222002	16	0	QLearning	40	After	2147483647	Yes	No	0,2	0	1	1	0	0	0	-1	0	0	0
7	6	6	12---3++++4--5+|1-2----3++++4+	N Movements	1	Pattern	p5	1	0	Human	50	After	2147483647	Yes	No	0,96	0	1	0	1	1	1	1	1	1	1
8	6	6	12---3++++4--5+|1-2----3++++4+	N Movements	1	Pattern	p5	1	0	QLearning	50	After	2147483647	Yes	No	0,7	-1	1	0	0	0	-1	-1	-1	-1	-1
9	7	7	1------23+++4----5--6|1--2------3+	N Movements	1	Pattern	p233441145	9	0	Human	60	After	2147483647	Yes	No	-0,2	0	0	0	0	-1	0	-1	0	0	0
10	7	7	1------23+++4----5--6|1--2------3+	N Movements	1	Pattern	p233441145	9	0	QLearning	60	After	2147483647	Yes	No	-0,066666667	1	0	0	-1	-1	-1	0	0	0	0
11	8	4	1+++23+|1-2-3+++++++|1-------2+	N Movements	1	Pattern	p102212	6	0	Human	70	After	2147483647	Yes	No	0,828571429	0	0	-1	1	0	1	1	0	1	1
12	8	4	1+++23+|1-2-3+++++++|1-------2+	N Movements	1	Pattern	p102212	6	0	QLearning	70	After	2147483647	Yes	No	0,4	0	0	-1	1	1	0	0	-1	-1	1
13	9	3	1++++++++2----|1++2+|1--------2-	N Movements	1	Pattern	p212101	6	0	Human	80	After	2147483647	Yes	No	0,1375	0	0	0	-1	1	1	0	0	0	0
14	9	3	1++++++++2----|1++2+|1--------2-	N Movements	1	Pattern	p212101	6	0	QLearning	80	After	2147483647	Yes	No	0,2	0	0	0	1	1	1	1	0	0	0
15																										

Illustration 14 - Experiment file

References

[ANYNT] http://users.dsic.upv.es/proy/anynt/

[Hernández-Orallo, 2010] A (hopefully) non-biased universal environment class for measuring intelligence of biological and artificial systems. In M. Hutter et al., editor, Artificial General Intelligence, 3rd Intl Conf, pages 182–183. Atlantis Press, 2010