UTBM - UNIVERSITE TECHNOLOGIQUE DE BELFORT-MONTBELIARD

LEMMIGEEK REVOLUTION
Agent oriented with Q-Learning

VI51 Artificial Intelligence for Games and Virtual Environments - P2016

Collaborators: JARDINO Thibaud GI04
BLEIN Thibaud GI04
BRETEGNIER Paul-Emile GI04

Teachers: GALLAND Stéphane
LAURI Fabrice
GECHTER Frank

Summary

Introduction

Continuously in virtual environments, new technologies and tools are discovered and created. This project is the realization of 3 students of the French University of Applied Sciences of Belfort-Montbéliard, studying computer Sciences in their 4rd year (out of 5 to become engineers). This project was realized within the VI51 course about artificial intelligence in video games and virtual environments.

More precisely, this project's goal was to train what we learned in the courses, here, agent oriented programming and machine learning. This game is inspired bythe original Lemmings game produced in the 90s and include an artificial intelligence on the lemmings.

We will first present the game rules, then discuss the type of artificial intelligence and the searched behavior of the lemmings. After this we will focus on the technical realization and architecture of the code. Finally we will give a brief analyze of the difficulties encountered and the application of Q Learning on this game.

Game rules

This game will take place on a rectangle map, where left and right are linked to make a sort of cylinder. Each piece of the map can initially have 5 states (first 4 initially, last 1 only realizable in game :

Entrance (only one), Exit (only one), Empty or Land Halfed Land

The goal of the lemmings is to go from the entrance to the exit without dying, to achieve this he got 6 possible moves: GoForward (moving one forward in direction of orientation if possible), GoBackward (change orientation and GoForward), Parachute (while falling, lower the falling speed by one with minimum fall speed equals to 1), Dig (if the bottom box is Land the launch a 2 turn action turning Land to Halfed Land then Halfed Land to Empty), ClimbForward (if there is a wall in front of him climbs up and if on top of the wall goes diagonal, if there is Land over him make him stay same place), ClimbBackward (change orientation and Climb). You can't climb over the map as it's like Land.

The global objective is like in the old game: bring as much lemmings to exit as possible, in this game they are 3 ways to lose lemmings: or they fall without parachute so the fall speed increases and if they arrive on land with speed superior then 3 they die, or bydestroying the ways to access to exit or they just fall under the map (dying in lava in an horrible agony).

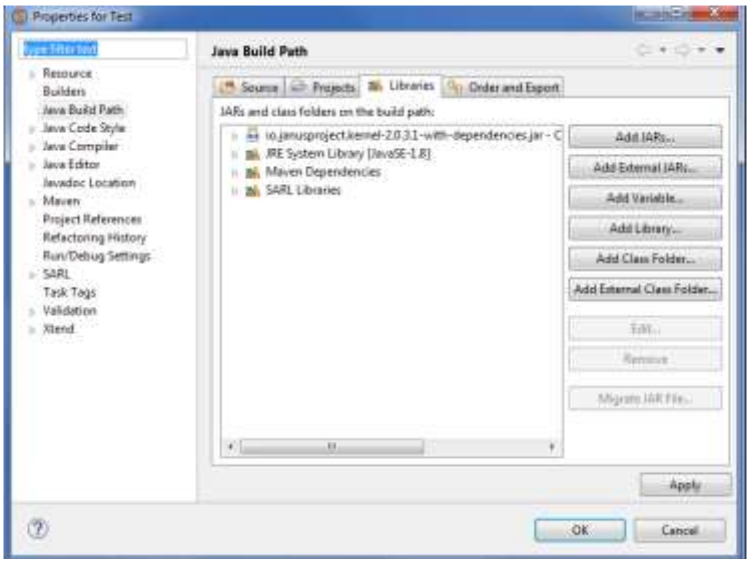
User Guide

How to install the game

To use this project on your computer you have to download SARL. SARL is a framework that facilitates the use of Janus. The main goal of this overlay of JAVA is to easily create a program with multi agents. You will find the SARL executable on the website SARL.io (here the link : http://www.sarl.io/download/index.html).

After that, you need to clone our project on Github (https://github.com/iSeehz/VI51). Now you are ready to set up SARL. As you can see our project is a maven repository, you just have to build the project Test with the pom.xml.

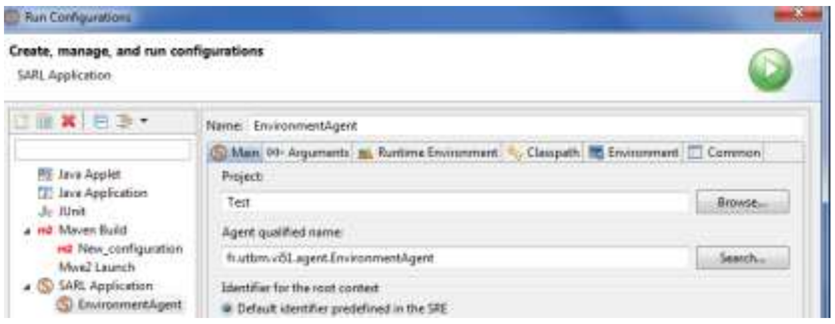
To launch Test, you need to specify in the Java Build Path the jar " io.janusproject.kernel-2.0.3.1-with-dependencies.jar" (contained at the root of our project).



JARs and class folders on the build path:

- io.janusproject.kernel-2.0.3.1-with-dependencies.jar - C
- JRE System Library [JavaSE-1.8]
- Maven Dependencies
- SARL Libraries

Now, you can create a Runnable SARL Application using Run Configuration. The class to call is EnvironmentAgent. Be careful, you have to check if the Runtime Environment use the Janus Kernel.



Name: EnvironmentAgent

Project: Test

Agent qualified name: fr.utbm.vi51.agent.EnvironmentAgent

Identifier for the root context

● Default identifier predefined in the SRE

And that's it, the simulator will start !

If you want more lemmings, modify the var numberChoice in EnvironmentAgent.

Type and game logic of the AI

Quick introduction on Q Learning

Firstly, machine learning is studying of past experiences and data to try to make predictions. Q Learning is a learning reinforcement algorithm.That's mean that a the actions used in the game are improving the AI. In Q Learning you got an evaluation matrix made by the user and a probability matrix. Each of those matrix have for lines the different possible states and for columns the possible actions.

The values in the probability matrix are initially all the same then after each action, a new value is computed for the state/action combination given by this:

Value (state/action) = Evaluation + Chose coefficient * max (state/all action)

Ex:

Evaluation Matrix

$$\begin{array}{c|c}
 & \textbf{Action} \\
\textbf{State} & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array}
\end{array}$$

$$\begin{array}{c}0\\1\\2\\3\\4\\5\end{array}\begin{bmatrix}
-1 & -1 & -1 & -1 & 0 & -1 \\
-1 & -1 & -1 & 0 & -1 & 100 \\
-1 & -1 & -1 & 0 & -1 & -1 \\
-1 & 0 & 0 & -1 & 0 & -1 \\
0 & -1 & -1 & 0 & -1 & 100 \\
-1 & 0 & -1 & -1 & 0 & 100
\end{bmatrix}$$

Initial Probability Matrix

$$\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 100 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & \boxed{80} & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

The coefficient value is: 0.2

If you are in state 3 and do action 1 and this brings u in state 3 again.

New Value (3/1) = 0 + 0.2 * 80 = 16

Searched behavior of our Lemmings

The goal of the Lemming is to reach the exit, and in our game there is no possibility to put new elements to access again a higher platform you left. So we want our Lemming to walk over all the platform, then if he can't find the exit, he should try to go down ensuring he got something below him to land on, if finally he found nothing he should jump down or dig hoping he got something below him. To get the technical realization go to the next part of the report.

Implementation Overview

UML Class Diagram

Veuillez trouver le digramme uml en Annexe.

Model aspect

The Environment

The environment java-class manage the world, its objects and the interactions. It also give the perception to the agents. It contains a lots of information: it counts the number of bodies, the dead lemmings, how much of them get out, a list of the cells that changed.

The world

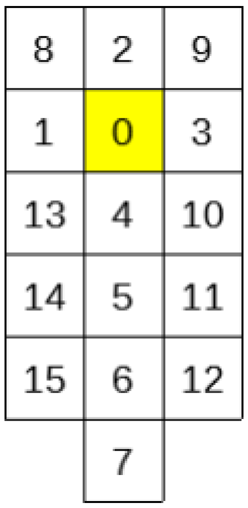
The world is a grid, managed with a double list of cells. It is generated via a text file. We've defined 6 types of cell:

- Entry: It's the spawn cell for every lemmings
- Exit: it's the objective, where lemmings must go in order to win
- Land: it's a solid cell, on which a body can walk
- Half: it's a cell getting dug, intermediate state before empty
- Empty: It's an "air" cell.
- Wall: it's what is returned in perception over the top of the map

All lemmings' body, when created, are stored in a list of lemmingBodies. References to them are stored in a list inside the cell where they are. At the beginning, all of them are stored in the list of the entry cell. The two edges of the map are connected, it means that if, the body try to go outside by the left-side of the grid, he'll arrive on the right-side.

The perception

The perception of the lemming depends on its line of sight. By default, it can see 1 case on each 8 cardinal directions, and 3-4-3 cells below him. Its line of sight isn't blocked by walls, it can see through. The global perception of the lemming is stored as an array of Percepts (description of a cell: position and type). If a Percept is out of the map, it returns an "Empty" Cell if it's under the map, a "Wall" if it's over the map. The function getPerception will receive an id. It will use the function searchBody, to find its coordinates. Then, it will add all Percepts in the array in this order: left, top, right, bottom left-top corn, right-top corner, right-bottom corner, left-bottom corner (see illustration)



The moves

Possible moves in our simulator, the different moves depend on the orientation of the Lemming. It can move backward and forward (left or right, depending on the orientation), dig, parachute or climb (left or right, depending on the orientation) if a Lemming is using a backward move or traversing the limits of the map the fatigue is increased. If the Lemming is falling while having no land left and right fatigue is reset.

MoveBody

This function receive a lemming's id, and a move and return a boolean: true if the movement is done, false otherwise. First, if the body is falling, its fatigue is reset. Then, if it's digging, (the cell under is half at this moment), it continue to dig, so, the cell becomes empty and every lemmings staying on cell will fall. If the lemming isn't digging but is on a (half) land, it can move (backward or forward) or dig., if he move backward, its fatigue increases. If the body uses the move "parachute", the parachute is activated and the body continues to fall (if it can) and falling speed decrease. This function calls, when it's necessary, other function such as moveRight, moveLeft, or climbingBody. Theses functions will check if the movement is possible (depending on the cells around the lemmings) and do the movement. Else, if it's not possible, the body doesn't move. Like in PacMan.

StatusBody

After a movement, the statusBody function, checks if the body "arrived in the air": for example, the lemmings was on the edge of a platform and move out of the platform, there isn't land below it, it is in the air. Then, it's going to fall. This function, also increase the acceleration of a body, and if this acceleration is too high on landing (3 in this simulation) the function kill the body. This function is also in charge of moving out the lemmings in the exit cell.

The lemmings body

The lemmingBody Java class contains all the characteristics of a lemmingBody: its position, its line of sight, its orientation, its action (parachute, climbing, digging, suicide…) and its fatigue. When a lemming spawn, it is in the entry cell of the map, he's oriented on the right. By default, its fatigue is set at 1 and can be increased by 1 when it goes backward and by 0,8 when it goes on the other side of the map. The Java class contains all the procedures to activate or deactivate actions. When a lemming is falling, its orientation is "down", when it lands, it orientation depends on its fatigue, if it is odd, the lemmings turn right, if not he will turn left.

Agent aspect

The main class is EnvironmentAgent. This class contains all the object's access used in our project.
EnvironmentAgent launches the GUI, creates the level by default with the EnvironmentModel. It spawns all the LemmingAgent needed for the simulator using numberChoice. Moreover, EnvironmentAgent manages all the event received in the EventSpace and modify with function's call when it's needed the objects in the simulation.

We have two variables rather useful in this class :
var mapOfGUID : HashMap<Address,Integer > and **var** moveInfluences:HashMap<Integer, PossibleMove>

The first links the Address of LemmingAgent and the Body ID to know which bodies belong to one agent.

The second recovers the influence sent by the LemmingAgent and links it with the body ID to identify when all the agent have sent their perceptions.

We have the function sendPerceptionsToAgents() which send the perception to each agent using its address. We choose to use this solution even if it is the most expensive because that is a question of security an agent must have access just to its own perception.

Thanks to the class Controller, all the interaction with the gui is sent in the event space using event.

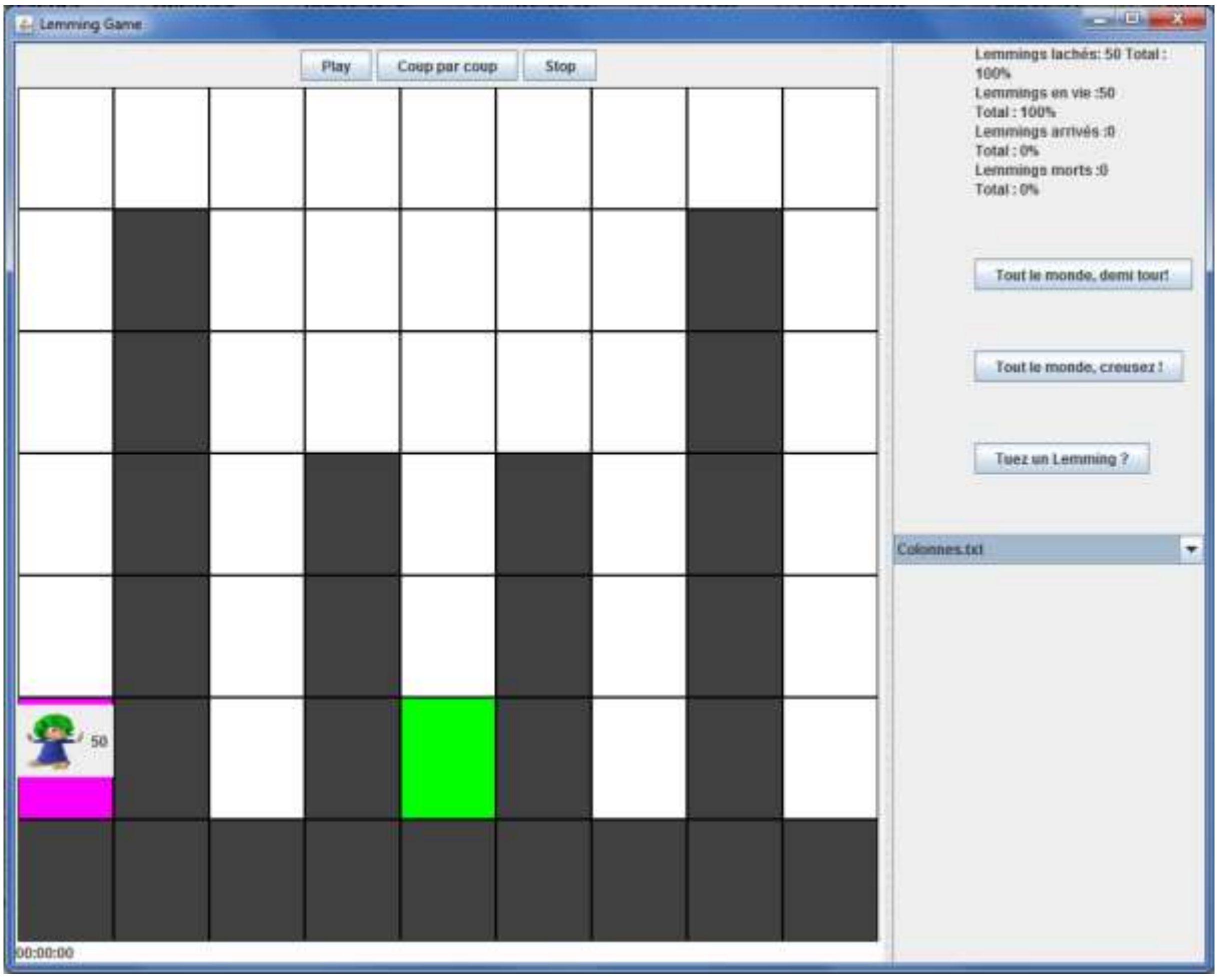
GUI aspect

Here, this is our GUI.

You can play in auto the simulator, step by step each turn and stop the current simulation.

The user can interact with the world using the 3 buttons "Tout le monde, demi tour!", "Tout le monde creusez" and "Tuez un Lemming".

The CheckBox "Colonnes.txt" represents the level displayed on the example. You can make your own level in the folder ("VI51\Test\src\main\resources\fr\utbm\vi51\level") using the json format.



Lemmings lachés: 50 Total : 100%
Lemmings en vie :50
Total : 100%
Lemmings arrivés :0
Total : 0%
Lemmings morts :0
Total : 0%

`0 represents the spawn of lemmings (only one 0), 1 represents a empty's cell, 2 represents a land and 3 the exit searched by lemming.`

One little example:

```
{
   "Height": "6",
   "Width": "5",
   "Content": [
          [0,1,1,1,1],
          [2,2,2,2,1],
          [1,1,1,1,1],
          [1,2,2,2,1],
          [1,1,3,1,1],
          [2,2,2,2,2]
   ]
}
```

You have to add your file in Option Panel in this line :

"`String level[] = {"Etages.txt","Colonnes.txt","Escalier.txt” ,…};`"

AI aspect

Firstly we searched how we could ensure that the Lemming will walk through all the platform so we added him a feature called by default “exhaust level” which increase by 1 each time the lemming turns, and adding 0,8 each time he traverse the map limit to ensure he won’t walk straight infinitely. Initially and after each time we got him falling without having a wall left or right side exhaust level is set to 1. When exhaust level reaches 3 the Lemming begin to search another platform to land on. When exhaust levels gets on 5 the Lemming should try soon to go down without any other conditions.

Making the Evaluation map required us to consider all combinations possible on the Lemmings perception. Then we grouped those for which we wanted the same actions to occur (ex: Going Forward, Climbing Backward or Digging) to get a matrix easier to understand. We also grouped the piece of perception below the lemming because we only needed to see if there was a door or a platform not to know how much land was there or where it exactly is.

The full decision making process occurs like this: the Lemming computes the state in which he is, he gives the current states to a function and get back a vector of probability of all actions to occur. Then he take a “random” number between 0 and the sum of all positives probabilities and get the correspondent action.

The probability and evaluation matrixes are stored in .txt files, they are both load when launching the program. Probability matrix is edited after each move and saved after each turn. The probability matrix initially only contains 100 valus

To train our Lemmings we realised some maps to try to englobe a huge number of cases, we let a big majority of them and added some bigger maps, anyway, u can add maps as said before.

The explanations of the 0 present in our matrix is easy: the action were performed when they were still 100 values in it and so the probability of the action became:
-10 + 0.1 * 100 = 0 and this probability is now null so they won’t do the action again.

Conclusion

This project was a good experience for us, it made us apply agent oriented programming on a funny game. It also made us improve our knowledge in learning algorithms and especially in Q Learning.

This project is also a good overview of a part of our capacities and what we can achieve in a small team and a limited time with other projects nearby.

We can objectively criticize the way we used Q Learning on this project, having no real stabilization of the probability matrix, due to the fact that we were using an evaluation method that would only stabilize if an action on a definite state would always raise the same next state. To make this happen we could add a last matrix that count the number of time we got the combination state/action and next state and then not add q(next state/max) but the average value of q(each next states/max/probability). But I think that was a good approach for a first real work on it.

The things caused us the most trouble were the lack of examples on SARL and the necessity to forget no cases on the lemming state evaluation.

Finally, ameliorations that could be done on this game could be:

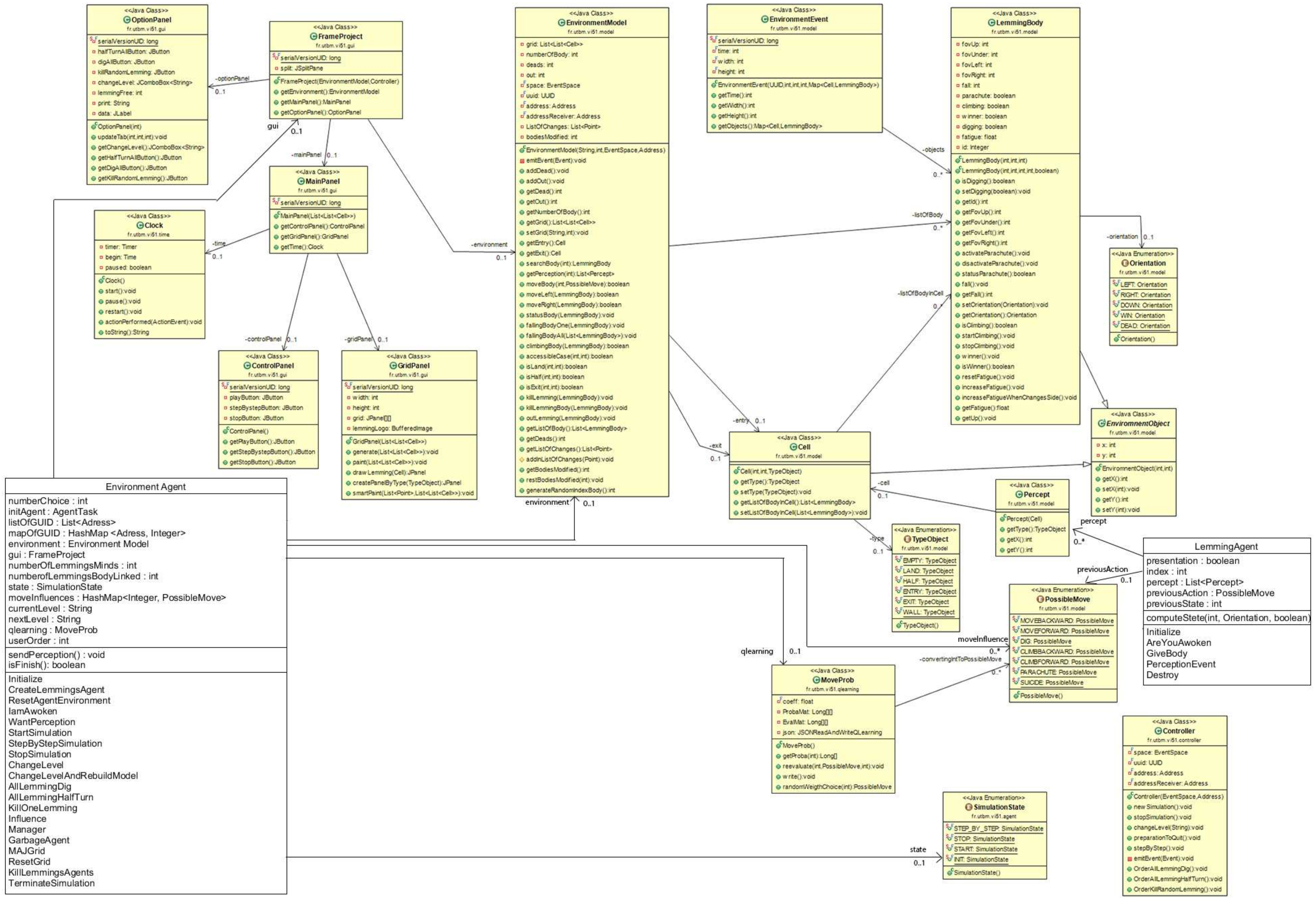
- Possibility to click on map pieces to change from void to land and land to void
- Possibility to choose in game the number of lemmings for the next games
- More actions like making stairs, jumping …
- …

Bibliography

- **r-bloggers.com/**a-quick-introduction-to-machine-learning-in-r-with-caret/
- **mnemstudio.org/**path-finding-q-learning-tutorial.htm
- http://people.revoledu.com/kardi/tutorial/ReinforcementLearning/Q-Learning-Example.htm

- **http://sarl.io/**

Annexe: Class Diagram



<<Java Class>> OptionPanel fr.utbm.vi51.gui
serialVersionUID: long
halfTurnAllButton: JButton
digAllButton: JButton
killRandomLemming: JButton
changeLevel: JComboBox<String>
lemmingFree: int
print: String
data: JLabel
OptionPanel(int)
updateTab(int,int,int):void
getChangeLevel():JComboBox<String>
getHalfTurnAllButton():JButton
getDigAllButton():JButton
getKillRandomLemming():JButton
<<Java Class>> FrameProject fr.utbm.vi51.gui
serialVersionUID: long
split: JSplitPane
FrameProject(EnvironmentModel,Controller)
getEnvironment():EnvironmentModel
getMainPanel():MainPanel
getOptionPanel():OptionPanel
-optionPanel 0..1
gui 0..1
-mainPanel 0..1
<<Java Class>> MainPanel fr.utbm.vi51.gui
serialVersionUID: long
MainPanel(List<List<Cell>>)
getControlPanel():ControlPanel
getGridPanel():GridPanel
getTime():Clock
<<Java Class>> Clock fr.utbm.vi51.time
timer: Timer
begin: Time
paused: boolean
Clock()
start():void
pause():void
restart():void
actionPerformed(ActionEvent):void
toString():String
-time 0..1
-controlPanel 0..1
-gridPanel 0..1
<<Java Class>> ControlPanel fr.utbm.vi51.gui
serialVersionUID: long
playButton: JButton
stepBystepButton: JButton
stopButton: JButton
ControlPanel()
getPlayButton():JButton
getStepBystepButton():JButton
getStopButton():JButton
<<Java Class>> GridPanel fr.utbm.vi51.gui
serialVersionUID: long
width: int
height: int
grid: JPanel[][]
lemmingLogo: BufferedImage
GridPanel(List<List<Cell>>)
generate(List<List<Cell>>):void
paint(List<List<Cell>>):void
drawLemming(Cell):JPanel
createPanelByType(TypeObject):JPanel
smartPaint(List<Point>,List<List<Cell>>):void
-environment 0..1
<<Java Class>> EnvironmentModel fr.utbm.vi51.model
grid: List<List<Cell>>
numberOfBody: int
deads: int
out: int
space: EventSpace
uuid: UUID
address: Address
addressReceiver: Address
ListOfChanges: List<Point>
bodiesModified: int
EnvironmentModel(String,int,EventSpace,Address)
emitEvent(Event):void
addDead():void
addOut():void
getDead():int
getOut():int
getNumberOfBody():int
getGrid():List<List<Cell>>
setGrid(String,int):void
getEntry():Cell
getExit():Cell
searchBody(int):LemmingBody
getPerception(int):List<Percept>
moveBody(int,PossibleMove):boolean
moveLeft(LemmingBody):boolean
moveRight(LemmingBody):boolean
statusBody(LemmingBody):void
fallingBodyOne(LemmingBody):void
fallingBodyAll(List<LemmingBody>):void
climbingBody(LemmingBody):boolean
accessibleCase(int,int):boolean
isLand(int,int):boolean
isHalf(int,int):boolean
isExit(int,int):boolean
killLemming(LemmingBody):void
killLemmingBody(LemmingBody):void
outLemming(LemmingBody):void
getListOfBody():List<LemmingBody>
getDeads():int
getListOfChanges():List<Point>
addInListOfChanges(Point):void
getBodiesModified():int
restBodiesModified(int):void
generateRandomIndexBody():int
environment 0..1
<<Java Class>> EnvironmentEvent fr.utbm.vi51.model
serialVersionUID: long
time: int
width: int
height: int
EnvironmentEvent(UUID,int,int,int,Map<Cell,LemmingBody>)
getTime():int
getWidth():int
getHeight():int
getObjects():Map<Cell,LemmingBody>
-objects 0..*
-listOfBody 0..*
-listOfBodyInCell 0..*
<<Java Class>> LemmingBody fr.utbm.vi51.model
fovUp: int
fovUnder: int
fovLeft: int
fovRight: int
fall: int
parachute: boolean
climbing: boolean
winner: boolean
digging: boolean
fatigue: float
id: Integer
LemmingBody(int,int,int)
LemmingBody(int,int,int,int,boolean)
isDigging():boolean
setDigging(boolean):void
getId():int
getFovUp():int
getFovUnder():int
getFovLeft():int
getFovRight():int
activateParachute():void
disactivateParachute():void
statusParachute():boolean
fall():void
getFall():int
setOrientation(Orientation):void
getOrientation():Orientation
isClimbing():boolean
startClimbing():void
stopClimbing():void
winner():void
isWinner():boolean
resetFatigue():void
increaseFatigue():void
increaseFatigueWhenChangesSide():void
getFatigue():float
getUp():void
-orientation 0..1
<<Java Enumeration>> Orientation fr.utbm.vi51.model
LEFT: Orientation
RIGHT: Orientation
DOWN: Orientation
WIN: Orientation
DEAD: Orientation
Orientation()
-entry 0..1
-exit 0..1
<<Java Class>> Cell fr.utbm.vi51.model
Cell(int,int,TypeObject)
getType():TypeObject
setType(TypeObject):void
getListOfBodyInCell():List<LemmingBody>
setListOfBodyInCell(List<LemmingBody>):void
<<Java Class>> EnvironmentObject fr.utbm.vi51.model
x: int
y: int
EnvironmentObject(int,int)
getX():int
setX(int):void
getY():int
setY(int):void
-cell 0..1
<<Java Class>> Percept fr.utbm.vi51.model
Percept(Cell)
getType():TypeObject
getX():int
getY():int
percept 0..*
-type 0..1
<<Java Enumeration>> TypeObject fr.utbm.vi51.model
EMPTY: TypeObject
LAND: TypeObject
HALF: TypeObject
ENTRY: TypeObject
EXIT: TypeObject
WALL: TypeObject
TypeObject()
Environment Agent
numberChoice : int
initAgent : AgentTask
listOfGUID : List<Adress>
mapOfGUID : HashMap <Adress, Integer>
environment : Environment Model
gui : FrameProject
numberOfLemmingsMinds : int
numberofLemmingsBodyLinked : int
state : SimulationState
moveInfluences : HashMap<Integer, PossibleMove>
currentLevel : String
nextLevel : String
qlearning : MoveProb
userOrder : int
sendPerception() : void
isFinish(): boolean
Initialize
CreateLemmingsAgent
ResetAgentEnvironment
IamAwoken
WantPerception
StartSimulation
StepByStepSimulation
StopSimulation
ChangeLevel
ChangeLevelAndRebuildModel
AllLemmingDig
AllLemmingHalfTurn
KillOneLemming
Influence
Manager
GarbageAgent
MAJGrid
ResetGrid
KillLemmingsAgents
TerminateSimulation
LemmingAgent
presentation : boolean
index : int
percept : List<Percept>
previousAction : PossibleMove
previousState : int
computeState(int, Orientation, boolean)
Initialize
AreYouAwoken
GiveBody
PerceptionEvent
Destroy
previousAction 0..1
<<Java Enumeration>> PossibleMove fr.utbm.vi51.model
MOVEBACKWARD: PossibleMove
MOVEFORWARD: PossibleMove
DIG: PossibleMove
CLIMBBACKWARD: PossibleMove
CLIMBFORWARD: PossibleMove
PARACHUTE: PossibleMove
SUICIDE: PossibleMove
PossibleMove()
moveInfluence 0..*
-convertingIntToPossibleMove 0..*
qlearning 0..1
<<Java Class>> MoveProb fr.utbm.vi51.qlearning
coeff: float
ProbaMat: Long[][]
EvalMat: Long[][]
json: JSONReadAndWriteQLearning
MoveProb()
getProba(int):Long[]
reevaluate(int,PossibleMove,int):void
write():void
randomWeigthChoice(int):PossibleMove
<<Java Class>> Controller fr.utbm.vi51.controller
space: EventSpace
uuid: UUID
address: Address
addressReceiver: Address
Controller(EventSpace,Address)
newSimulation():void
stopSimulation():void
changeLevel(String):void
preparationToQuit():void
stepByStep():void
emitEvent(Event):void
OrderAllLemmingDig():void
OrderAllLemmingHalfTurn():void
OrderKillRandomLemming():void
state 0..1
<<Java Enumeration>> SimulationState fr.utbm.vi51.agent
STEP_BY_STEP: SimulationState
STOP: SimulationState
START: SimulationState
INIT: SimulationState
SimulationState()