Internship Report

Generalizing and improving Artificial Intelligence of a Computer-agent game

Location : London, United-Kingdom
Company : City University London
Department : Computer Science
Supervisor : Dr. Christopher Child
Dates : 4th June 2014 to 11th September 2014
Studies : MSc. Computer Science

CITY UNIVERSITY LONDON

ensiie
Paris-Évry | Strasbourg

Preface

Currently in 2nd year of IT engineering near Paris (ENSIIE – Evry), I have the opportunity to do a 3 month internship at London City University.

I chose London because I intend to work there after graduating, and of course to improve my English level, since living in the UK means practicing everyday. Furthermore, I wanted to work on a research subject, because it allows some freedom regarding the direction of a project, sometimes leading to new exciting challenges.

In this internship report, I will describe the tasks I have been working on, as well as the personal & professionnal skills I have aquired during the internship.

For this opportunity, I thank :

Christopher Child (Dr.), who supervised my internship. He is a games technology lecturer at London City University since 2008, and course director of the Computer Games BSc (since 2012) & MSc (since 2008).
After working at Empire and Logica CMG, he now runs his own computer game company « Childish Things Ltd. ».
He led the project and gave me day-to-day feedback and analysis, despite a busy schedule.

I would also like to thank **Beyrem Jebri** & **Moad Mellah**, ENSIIE students that shared a flat with me during our stay.

Finally, all the friends I made in London, who helped making this adventure unforgettable.

Virgile Barbieux Christopher Child

Contents

List of Figures & Tables

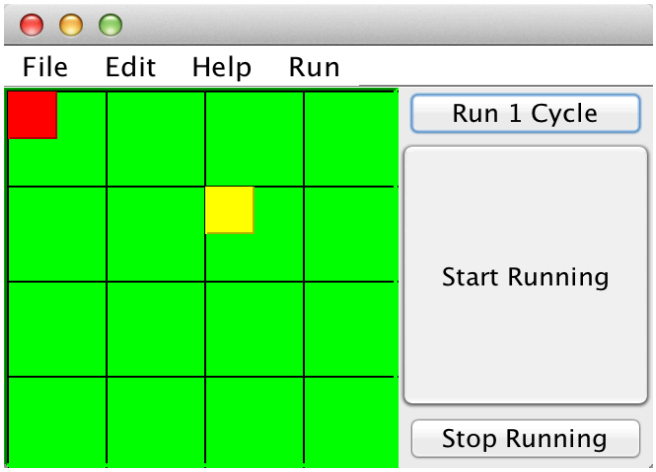
1. Existing Project

1.1 Description

The existing project is the result of a 3 years research conducted by C. Child during his PhD Thesis : « *Approximate Dynamic Programming with Parallel Stochastic Planning Operators* » (Nov. 2011).

It is a JAVA project containing 3 different games applications, and several stochastic algorithms used to build AI models considering the perception of an « Agent » is its environnement.

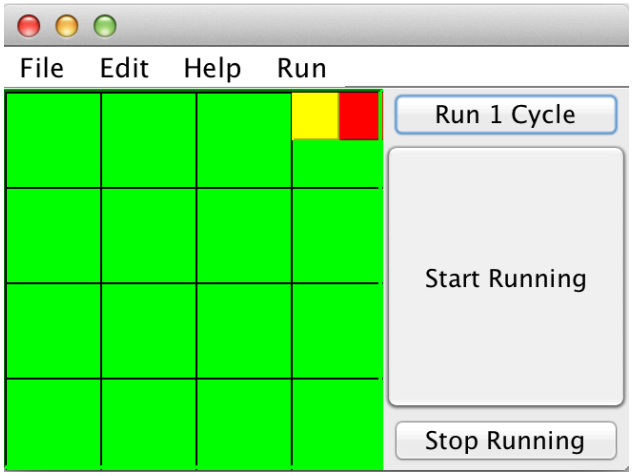
The main game I have been working with is a simple Predator/Prey Model in a 4x4 grid, like below :



Here, the Predator (red) is in the top left corner.

The Prey (yellow) is in the middle.

Figure 1.1 : Representation of the Predator/Prey game grid.



The Predator catches the Prey when it lands on the same cell.

Figure 1.2 : Representation of the Predator catching a Prey.

1.2 Agent & Environment

The environment here contains two Agents (a Predator and a Prey).

These Agents can perceive (identify what is « around » them), deliberate (think about an action outcome) and act (exerce an action).

In this game, we only work with the Predator, which means the Prey is « randomly » going from cell to cell, while the Predator tries to catch it.

In order to build a model to « guide » the Predator, an Agent is able to perceive what is around him through Perceps (Perceptions).

However, the Agent can't see the whole grid, which would of course be convenient, but not interesting enough considering action choices.

As a result, the game works using perceptions (Percep) of what the Agent sees North, West, South, East and Under (In the same cell).

To illustrate, in both figures shown above, the Agent Percep is :

Figure 1.1 : [WALL, EMPTY, EMPTY, WALL, EMPTY]
Figure 1.2 : [WALL, WALL, EMPTY, EMPTY, AGENT]

1.3 MSDD & ASDD algorithms

In order to select the best action for every step, two algorithms were used :

- ***MSDD*** (Multi-Stream Dependancy Detection)
- ***ASDD*** (Apriori Stochastic Dependancy Detection)

As they are both part of my internship work, I will describe and explain them more precisely further in this report.

By detecting dependencies between different states through these algorithms, our goal is to determine which action should be taken in a certain state, in order to have the greatest probability of catching the Prey in the future.

1.4 Motivations

Considering this, my goal will be to create a separate generalized version of MSDD & ASDD, in order to study Rules from different games.

The current system uses specific Percep & Actions build by the game, which means it would be very hard to use the existing algorithms to study a completely new game.

The idea of this project is to get a generalized version, which would allow to take an input file containing States from a game, and to build a model by learning Rules from it.

Theoretically, we could then be able to build AI for any game, as long as we have a big enough database to work with. The Agent could play a wide variety of games (basically every game we can implement in Java) like Chess, MineSweeper or 2048.

To create such an Agent, we need things to be as general as possible.

For instance, the perceptions from Figure 1.1 and Figure 1.2 above could be written like this :

Figure 1.1 : WEEWE
Figure 1.2 : WWEEA

Where W would be a Wall, A an Agent (The Prey), and E an Empty cell.

2. Building a general system

Since the existing project was really complex, we decided to start from scratch, in order to rebuild the project as best as possible (simple, efficient, polyvalent…).

2.1 Output LogFiles System

The existing project already had an Output File System, but it was not adapted to what we planed to do. That's why the first step has been to build a new Output System (LogFiles) using the Singleton class.

The previous version was creating a new link to the files (a new instance of FileWriter and PrintWriter) every time something was printed out, which resulted in huge memory access and time execution problems.

To prevent this, we used the Singleton class. With the Singleton design pattern, we can ensure that only one instance of the class is created, while providing a clear access point to the object, resulting in a cleaner system.

In other words, it allows us to create all the necessary files once at the beginning, and then access them whenever it is necessary.

For instance, when first priting « Hello » and then « world ! » in the file Output.txt :

Old version	**New version**
LogFile logfile = new LogFile (« filename ») ; *Logfile.println(« Hello ») ;* ***...*** *LogFile logfile = new LogFile (« filename ») ;* *Logfile.println(« world ! ») ;*	*LogFiles logfiles = getInstance() ;* *Logfile.println(« Hello ») ;* ***...*** *LogFiles logfiles = getInstance() ;* *Logfile.println(« world ! ») ;*
200+ instances	*1 instance, 200+ calls*
Execution Time : 40s	*Execution Time : 15s*

Table 2.1 : Differences between the old and new Ouput Log systems.

2.2 Tokens & Sensors

When reading the state *« WEEWES »*, we need to identify every character separately, since each one describes an aspect of the Agent's perception.

This state is a **Sensor** composed of 6 **Tokens**. The first 5 ones are the observations of the cells around the Agent, and the last one is the Action taken.

We can describe explicitely the above Sensor as :

Sensor « WEEWES »					
NORTH	**EAST**	**SOUTH**	**WEST**	**SAME CELL**	**ACTION**
Wall	Empty	Empty	Wall	Empty	South

Table 2.2 : Description of a Sensor.

In other words, the Predator would be alone in the top left corner, taking the action South.

A « special » Token we will need is the Wildcard. A Wildcard is represented by the char *« ***** »*, and matches every other Token.

The term « matching » is really important, for instance *« E »* matches *« ***** »*, but doesn't match *« A »*.

We can match Sensors like this :

[A, E, E, W, E, N]		[*, *, *, *, *, *] (root)
matches	doesn't match	matches every Sensor
[A, E, *, W, *, *]	[A, E, *, E, *, *]	

Table 2.3 : Sensor Matching examples.

2.3 TokenMap & Database

A) TokenMap

Since reading and comparing strings is expensive, we needed a better way to operate. Indeed, when setting, getting or comparing Tokens, we wanted to avoid checking if it's char was corresponding to *'X'*, that is why we created TokenMaps.

A TokenMap is obtained by scanning the input file, to see what Token can be seen at which position of the Sensors.

For instance :

TokenMap created : (8000 lines)
[[W, E, A], [E, A, W], [E, W, A], [W, E, A], [E, A], [N, E, S, W]]

Table 2.4 : Generated TokenMap for 8K input.

Accroding to the TokenMap, the first Token of a Sensor can either take the values W, E or A, i.e. be a Wall, an Agent, or be Empty.

B) SensorList, SensorMap & StateMap

At first, we built our database by listing all the Sensors occuring in the input file, as SensorLists.

In order to lower the execution time, we decided to implement SensorMaps.

Through SensorMaps, instead of listing every Sensor, we only stored the same Sensor once, and kept its indexes and occurencies separately.

Finally, as we needed to separate Sensors from their actions, in order to be able to choose the better one, we created StateMaps.

STATE 1 [W, E, E, E, E, *]	
Actions : [N, E, S, W]	**Occurencies** : [2281, 2354, 2319, 2253]
North leading to States : State 1 (1984 times) … State 34 (74 times) East leading to States : … South leading to States : … West leading to States : …	

Table 2.5 : StateMap entry for State « WEEEE ».*

2.4 Rules

Our goal is to study the Input by setting Rules, and determine which one are useful and relevant.

Now that we have our database (SensorList), we can build Rules defined by :

- A unique ID
- A Precursor (Precondition Sensor)
- A Successor (Postcondition Sensor)
- A Probability :

$$Prob = \frac{(Precursor\ Occurrencies\ directly\ followed\ by\ a\ Successor\ Occurrency)}{(Precursor\ Occurrencies)}$$

For instance :

RULE : [W, *, *, *, *, *] --> [*, *, E, *, *, *]		
ID : 532	**Precursor** : [W, *, *, *, *, *]	**Successor** : [*, *, E, *, *, *]
Prob : 0.938	**Precursor Occurencies** : 24282	**Rule Occurencies** : 22785

Table 2.6 : RuleList entry for Rule 532.

In other words, when there is a Wall to the North of the Agent, whatever the action is, there is **93.8%** chances to observ nothing (Empty cell) to the South of the Agent at the next state.

But what happens in the **6.2%** left ?

RULE : [W, *, *, *, *, *] --> [*, *, A, *, *, *]		
ID : 4701	**Precursor** : [W, *, *, *, *, *]	**Successor** : [*, *, A, *, *, *]
Prob : 0.062	**Precursor Occurencies** : 24282	**Rule Occurencies** : 1497

Table 2.7 : RuleList entry for Rule 4701.

We have **6.2%** chances to see an Agent to the South at the next state.

2.5 RuleSets

In the example above, we see it is interesting to « complete » a Rule by other Rules with the same Precursor, and similar Successors.

This process will be called « consolidating RuleSets ».

A RuleSet is basically a list of similar Rules. The process of consolidating RuleSets is very expensive, since we need to go through every possible Rule of the same type, determine if it occurs at least one time in the database, and add it to the right RuleSet.

In this particular case, the Rule 532 was generated while scanning the database. Then, the consolidation algorithm created both Rules :

- Rule (ID 4701) : [W, *, *, *, *, *] -> [*, *, A, *, *, *]
- Rule (ID 4702) : [W, *, *, *, *, *] -> [*, *, W, *, *, *]

The second Rule was pruned (not added) because it wasn't found in the database. It is perfectly normal, since if we have a Wall to the North, we need at least 3 steps before expecting a Wall to the South (3 South Actions).

RULESET 47	**Total Prob** : 1.0
RULE 532 : [W, *, *, *, *, *] --> [*, *, E, *, *, *]	**Prob** : 0.938
RULE 4701 : [W, *, *, *, *, *] --> [*, *, A, *, *, *]	**Prob** : 0.062

Table 2.8 : RuleSetList entry for RuleSet 47 (after consolidation).

3. MSDD Algorithm

3.1 Description

The Multi-Stream Dependancy Detection algorithm works on Rules to detect dependancies. This allows to determine which Rule is relevant, and if some Rules are covered by others.

Formal statements of the MSDD algorithm and its node expansion routine are given in *Appendix A*.

3.2 Expand

Before implementing MSDD, we need the expand function. The action of « expanding » a Sensor means replacing a WildCard by every possible Token, according to the TokenMap.

For instance, expanding [W, E, E, E, E, *] at the last index would give us the following children :

[W, E, E, E, E, N]	[W, E, E, E, E, E]	[W, E, E, E, E, S]	[W, E, E, E, E, W]

*Table 3.1 : Childrens of expand ([W, E, E, E, E, *] , 5) .*

The expand function called by MSDD is more complete. Indeed, it takes one Rule and return all of its children Rules, i.e. every possible combination obtained by expanding ONE of the available indexes in the Precursor or the Successor.

An example of a Rule expand process is given in *Appendix B*.

3.3 ClosedList & OpenList

The MSDD algorithm expands the most important Rules first, in order to get good results even if the MAXNODES parameter (set by user), is low.

To increase the performance of the algorithm, we work with two lists called OpenList & ClosedList.

The ClosedList contains the MSDD learned Rules, and the OpenList contains the Rule « waiting » to be picked.

By constantly sorting the OpenList by precursor occurrencies, we ensure that we pick the most general Rules at first. That way, even if MAXNODES is low, we are able to quickly learn what to do in most cases.

3.4 Algorithm (PseudoCode)

- We start with an empty ClosedList, and the root Rule in the OpenList

- While (MAXNODES not reached) AND (OpenList not Empty)
  - ✓ Sort OpenList
  - ✓ Expand first entry of the OpenList, adding its children to the OpenList
  - ✓ Remove this entry from OpenList, add it to the ClosedList

- If (OpenList not Empty)
  - ✓ Add remaining entries to ClosedList

- Return ClosedList

After obtaining a beautiful ClosedList, we need to make it even more beautiful by « cleaning » it, that is where filtering begins.

3.5 Filtering

A) RootNodes

The first step of filtering is to get rid of the RootNodes, i.e. Rules with a « root » Successor.

These Rules don't bring any information, since « anything can happen ».

B) Freeloaders

Freeloaders are Rules covered by another Rule, giving no more information that the more generalized version.

For instance :

Freeloader	
From Rule	[E, A, E, *, E, *] -> [*, *, E, *, *, *]
Covered by Rule	[E, A, E, *, *, *] -> [*, *, E, *, *, *]
G-Statistic Value : 0.0	

Table 3.2 : Freeloader example.

When a Predator has nothing to the North & South, and a Prey to his Right, finding out it has nothing under him is irrelevant. Of course, since there only is one Agent (Prey) in the game, if the Predator has one to his right, he can't have one under him.

To detect Freeloaders, we use the G-Statistic (or G-test). It is a maximum likelihood significance test performed in situations where the Chi-Square test is also recommended, allowing us to determine if a Rule brings something new compared to another. The G-Statistic algortihm is given in *Appendix C*.

The MSDD algorithm is using a parameter to prune Rules when G-Stat is less than a certain value. This parameter can be set to :

Significance Level	95 %	90 %	50 %
G-Statistic Value	3.841	2.706	0.455

Table 3.3 : G-Statistic Values corresponding to usual significance levels.

4. ASDD Algorithm

4.1 Description & Algorithm

Like MSDD, The Apriori-Stochastic Dependancy Detection algorithm works on Rules to detect dependancies.

It is based on the Apriori Algorithm for mining association Rules, and MSDD for finding dependencies.

ASDD works with Rules by levels. The Level of a Rule is simply its number of non-wildcarded indexes :

- [E, *, *, *, E, *] -> [*, *, E, *, *, *] is a level 3 Rule
- [E, A, E, E, E, S] -> [*, *, E, *, *, *] is a level 7 Rule

The Algorithm we used is based on the version in this paper, with some changes regarding support, and body :

http://openaccess.city.ac.uk/3002/1/ASDD.pdf

Here is the ASDD Algorithm in PseudoCode :

- We start by determining the Level 1 Rules : $\boldsymbol{L}\ [\ \mathbf{1}\ ]$
- While $\boldsymbol{L}\ [\boldsymbol{k}-\mathbf{1}]\ \neq\ \emptyset$,
  - ✓ Generate Level k Candidates : $C[k]$
  - ✓ Filter Subsets
  - ✓ Run Occurrencies Pruning
- Return $L[1]\ U\ L[2]$ … $U\ L\ [k]$

ASDD results are the same as MSDD if the MAXNODES parameter is set high enough to get a « complete » Rule generation.

4.2 Generating Candidates

The Apriori function generates the level k candidates from a level k – 1 RuleSet by merging all of its Rules.

The version used only considers one non-wildcarded index in the Successor, that means two level k-1 Rules won't be combined if they both have an effect (a non-wildcarded index in the Successor).

4.3 Inside Filtering

ASDD theoretically creates every possible Rule, which takes a lot of time (like MSDD). One of the differences is that ASDD filters progressively irrelevant Rules, instead of only relying on the Post Filtering done in MSDD.

The Post Filtering (including RootNodes & Freeloaders) for ASDD remains the same as MSDD.

A) Subsets

The Subsets of a level k Rule are all the k-1 Rules it could have been generated from.

For instance, the Subsets of [E, A, E, *, *, *] -> [*, *, E, *, *, *] are :

- ✓ [*, A, E, *, *, *] -> [*, *, E, *, *, *]
- ✓ [E, *, E, *, *, *] -> [*, *, E, *, *, *]
- ✓ [E, A, *, *, *, *] -> [*, *, E, *, *, *]
- ✓ [E, A, E, *, *, *] -> [*, *, *, *, *, *]

A Rule is pruned if at least one of its Subsets is not present in $\boldsymbol{L}\ [\boldsymbol{k}-\mathbf{1}]$

B) Occurrencies Pruning

If the Rule doesn't occur at least x (set by user) times, it is pruned as well.

5. State Generator

5.1 Description

In order to be able to « guide » the Predator in its environment, we need to be able to build a model. One way to do it is to let the Predator act randomly long enough to learn what can happen in every scenario.

A State Generator allows us to generate a State (Sensor) from another State, considering a certain Action (Token). Of course, if the database is big enough, we can use the database to predict the outcome of a certain State & Action. This is the Maps State Generator.

However, if we don't have a lot of samples, the Rules can bring us information faster, and « deduce » some frequent behaviors. The Rules State Generator also has the ability to manage « unseen » perceptions.

Nevertheless, using Maps will in most cases be more efficient, particularly in our Predator/Prey game, as we dispose of unlimited source data.

5.2 Maps State Generator

With the StateMaps, we can quickly determine what can be the outcome of a State associated with an Action.

For Instance, WEEEES can lead to :

Source : [W, E, E, E, E, S]
SENSOR 1 [E, E, E, E, E, *] Prob : 0.843 SENSOR 2 [E, E, A, E, E, *] Prob : 0.051 SENSOR 3 [E, E, E, E, A, *] Prob : 0.032 SENSOR 4 [E, E, E, A, E, *] Prob : 0.037 SENSOR 5 [E, A, E, E, E, *] Prob : 0.037 Total Prob : 1.0

Table 5.1 : Maps State Generator example for [W, E, E, E, E, S].

5.3 Rules State Generator

The Rules State Generator builds a new State from the current State in a different way.

At first, it starts from the Root State, and progressively fills its indexes with the best available RuleSets that match the current State.

Here's an example from the State [W, E, E, A, E, *] and the Action South :

Source State & Action : [W, E, E, A, E, S]		
New State : [*, *, *, *, *, *]		
RuleSet 1061	[W, *, *, A, *, S] > [A, *, *, *, *, *]	0.276
	[W, *, *, A, *, S] > [E, *, *, *, *, *]	**> 0.724**
New State : [E, *, *, *, *, *]		
RuleSet 863	[W, *, *, A, *, S] > [*, *, *, *, E, *]	**> 1.0**
New State : [E, *, *, *, E, *]		
RuleSet 704	[*, E, *, A, *, S] > [*, E, *, *, *, *]	**> 1.0**
New State : [E, E, *, *, E, *]		
RuleSet 862	[W, *, *, A, *, S] > [*, *, E, *, *, *]	**> 1.0**
New State : [E, E, E, *, E, *]		
RuleSet 657	[*, *, *, A, *, S] > [*, *, *, E, *, *]	**> 0.752**
	[*, *, *, A, *, S] > [*, *, *, A, *, *]	0.248
New State : [E, E, E, E, E, *]		

Table 5.2 : Rules State Generator example for [W, E, E, A, E, S].

In this example, we obtained [E, E, E, E, E, *], but we could also have (with less probability) obtained [A, E, E, E, E, *] or [E, E, E, A, E, *].

Once a RuleSet has been chosen, we select a Rule from it pseudo-randomly, i.e. according to the probabilities given by the RuleSet.

5.4 Precedence Algorithm

In order to get the best possible State Generator, we need to choose the RuleSets wisely. At first, we considered using Precursor occurrencies, but the most general case is not always the more relevant.

As a consequence, we established a way to choose between two RuleSets when they conflict with Precedences.

A) Conflicts

Two RuleSets conflict if they both match a certain State, and have at least one common output variable. For Instance, the State below is matched by two conflicting RuleSets :

State : [W, E, E, E, E, S]		
RuleSet 862	[W, *, *, E, *, S] > [*, *, **E**, *, *, *]	**0.92**
	[W, *, *, E, *, S] > [*, *, **A**, *, *, *]	**0.08**
RuleSet 464	[*, *, *, E, E, S] > [*, *, **W**, *, *, *]	**0.22**
	[*, *, *, E, E, S] > [*, *, **E**, *, *, *]	**0.72**
	[*, *, *, E, E, S] > [*, *, **A**, *, *, *]	**0.06**

Table 5.3 : Conflicting RuleSets.

B) Intersection

We only need to set precedence between two RuleSets if they both match a certain State. By checking for the precursor indexes in our datatbase, we are able to avoid unnecessary comparisons.

C) Combined RuleSet

If RuleSets conflict and intersect, we create a combined RuleSet from them by :

- ✓ Merging Preconditions of RS1 & RS2 to get the RS(1&2) precondition
- ✓ Identifying common non wilcarded spots and expanding to get the potential successors
- ✓ Getting Rule Probabilities

RuleSet (862 & 464)	[W, *, *, E, E, S] > [*, *, **W**, *, *, *]	**0.00**
	[W, *, *, E, E, S] > [*, *, **E**, *, *, *]	**0.88**
	[W, *, *, E, E, S] > [*, *, **A**, *, *, *]	**0.12**

Table 5.4 : Combined RuleSet example.

As we can see, RuleSet 862 brings more information, since we can't have a Wall to the South from the Input State ([W, E, E, E, E, S]).

Since this new RuleSet contains the information of the two source RuleSets, we assume it is more accurate.

As a consequence, we should pick the closest one from the combined RuleSet.

D) Error Measure

To determine which RuleSet is the better one once we have the combined RuleSet, we used a custom error measure system.

If a generated Rule is present in a source RuleSet but doesn't occur in the database, the error is increased by **0.5**, in order to penalize RuleSets creating wrong behaviours.

Otherwise, the error is increased by the absolute value of the probabilities difference.

With the above example :

- **RuleSet 862** :

$$\boldsymbol{Error} = (\mathbf{0.92} - \mathbf{0.88}) + (\mathbf{0.12} - \mathbf{0.08}) = \mathbf{0.08}$$

- **RuleSet 464** :

$$\boldsymbol{Error} = (\mathbf{0.88} - \mathbf{0.72}) + (\mathbf{0.12} - \mathbf{0.06}) + \mathbf{0.5} = \mathbf{0.72}$$

RuleSet 862 should precede over RuleSet 464, since it is closer to the combined one.

E) Using Precedences

Once all Precedences have been set, we still need to determine which RuleSet to pick for every generation step.

If a certain State is matched by several RuleSets, their is a high probability that we can't determine the better one instantly.

For instance, a State matched by four RuleSets A, B, C and D, with A preceding over B and D, B preceding over C, and C preceding over A.

In that case, we choose the one preceding the most over the other candidates, which is here A.

6. Reinforcement Learning

The goal of Reinforcement Learning is to learn a model considering transitions between stochastic Rules by setting rewards. It is an exploration mechanism acting within the environment, and « noticing » good and bad behaviours.

The Reinforcement Learning generates between 30.000 (For Rules) and 2.000.000 (For Maps) States with actions randomly taken The process result is a Table, which will help us to determine which action should be taken given a certain State.

Among the numerous Reinforcement Learning existing techniques, we chose the Q-Learning algorithm since it fits our problem the best.

6.1 Q-Learning

The Q-Learning algorithm can be used to find an optimal action-selection policy. A policy is a rule that the agent follows in selecting actions, given the state it is in.

When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state.

A) Variables

Variable	Symbol	Description
Learning Rate	*LR*	The Learning rate determines how fast the new information overrides the old one. Setting it to 0 means the Agent won't learn anything, and 1 means only the recent information is considered.
Discount Factor	*DF*	The Discount Factor determines the importance of future rewards. Setting it to 0 means the Agent will only consider short-term rewards, while 1 means the Agent will look for long-term behaviour.
Reward	*R*	The Reward is awarded when a State and action combination leads to a winning State (a State where the Predator catches the Prey). This Value can also be negative in other applications.

Table 6.1 : Q-Learning Variables.

B) Algorithm

From a State, the Q-Learning algorithm takes a random action, generates a State using the State Generator, and looks at the result State to determine if it should be rewarded or not.

Every State (State Value Table) and sometimes every Action (State Action Value Table) has a Value. This Value is updated every time the algorithm sees the corresponding State, following the formula :

$$\boldsymbol{V}\ (\boldsymbol{State}) = V(State) + LR * [R + DF * OF\ (Next\ State) - V(State)]$$

Table 6.2 : Q-Learning Update Formula.

- ✓ *OF is the Optimal Future Estimation. When a Next State is generated, we look at the States that can be reached from it. This allows us to increase the long-term behaviour performance.*

6.2 State Value Table

The State Value Table allocates a Value to each State through the Reinforcement Learning Process.

Each time an action is taken, the Q-Learning updates the Value of the source State considering the ouput State (Reward, Optimal Future etc..).

STATE 1	[W, E, E, W, A, *]	Value : 6.17
STATE 2	[A, E, E, W, E, *]	Value : 5.83
STATE 3	[E, E, W, W, A, *]	Value : 5.80
STATE 4	[E, W, W, E, A, *]	Value : 5.73
STATE 5	[E, E, A, W, E, *]	Value : 5.65

Table 6.3 : State Value Table example.

6.3 State Action Value Table

This time, each State & Action combination is allocated a Value. This changes a bit the way the Reinforcement Learner works, but allows us to try Dynamic Programming (See 6.4).

This method uses the ε-Greedy algorithm, with a value of 0.2.

The ε-Greedy algorithm tries to pick the best action instead of trying each one randomly. ε set to 0.2 means a random action will be taken with a probability of 0.2, to avoid picking always the same action without trying other possibilities.

STATE 1 [W, W, E, E, E, *]	Action : East	Value : 2.72
	Action : North	Value : 2.73
	Action : South	Value : 2.85
	Action : West	Value : 2.82

*Table 6.4 : State Action Value entry for State [W, W, E, E, E, *] .*

6.4 Dynamic Programming

The ε-Greedy algorithm generates a list of possible States for every Action, but then picks the best one to update the values. As a consequence, a lot of information is lost.

Dynamic Programming fixes this, by updating the Value of a State with all the possible actions. This means the algorithm becomes much faster, and needs less steps to be efficient.

6.5 Decision Table

In order to read both Value and Action Value Tables, we created Decision Tables. Since we already have build the model, we are now able to translate these Tables to know which action should be taken for a certain State.

For Action Value Tables, it is pretty simple since we just have to choose the maximum value taken by one of the actions.

For Value Tables, we need to apply every possible action, look at the potential output States, and then determine which action should be taken.

STATE 1	[A, E, E, W, E, *]	Action : N
STATE 2	[E, E, W, W, A, *]	Action : S
STATE 3	[E, W, W, E, A, *]	Action : S
STATE 4	[E, E, A, W, E, *]	Action : S
STATE 5	[W, W, E, E, A, *]	Action : N

Table 6.5 : Decision Table example.

7. Results (In Progress)

7.1 Execution Time

Table 7.1 shows that for 100K Input Data, ASDD is *7 times faster* than MSDD.

An important part (regarding execution time) of the Rule learning is detecting conflicts between RuleSets and setting precedences. Since this is part of the Post-Filtering process, it takes the same time for both algorithms.

For low Input Data, setting precedences only takes 4% of the algorithm time, but the more data there is, the more important becomes the precedence algorithm. For 100K Input, it reaches 89% of total time for ASDD.

Another interesting point is that even with 10K Input (which is a quite large database), we only get a quarter of the Rules we could get with 100K.

This is caused by the Freeloader filtering, since the G-Statistic need a high number of occurrencies to consider a Rule « relevant ». Lowering the G-Statistic test value would allow to consider more Rules.

Input Data	ASDD	MSDD	Precedence part
100	**7s**	**7s**	**0.2s**
	149 Rules		*3% of total time*
	114 RuleSets		
1.000	**1m40s**	**2m06s**	**4s**
	529 Rules		*4% of total time*
	418 RuleSets		
10.000	**2m29s**	**6m38s**	**17s**
	700 Rules		*4% - 10% of total time*
	384 RuleSets		
100.000	**3m45s**	**22m22s**	**31m40s**
	2479 Rules		*58% - 89% of total time*
	1141 RuleSets		

Table 7.1 : Execution Time and number of Rules & RuleSets of ASDD and MSDD considering Input Data size.

Table 7.2 shows that Maps are *30 times faster* than Rules. As a consequence, we can consider huge numbers while doing Reinforcement Learning if we use Maps, but we are quickly limited if we use Rules.

RL steps	**Maps**			**Rules**		
	Val.	**Act. Val.**	**Act. Val. Dyn.**	**Val.**	**Act. Val.**	**Act. Val. Dyn.**
20.000	18s	16s	22s	11m43s	5m18s	12m10s
2.000.000	1m39s	25s	1m38s			

Table 7.2 : Execution Time of Reinforcement Learning using Value, Action Value and Action Value with Dynamic Programming Tables for Maps and Rules considering the number of Reinforcement Learning Steps.

7.2 State Generation

Table 7.3 shows the progress made by both state generators considering the size of the Input Data.

Maps		
1000	**10.000**	**25.000**
8.19 (12 missed)	**2.64** (4 missed)	**0.83** (0 missed)

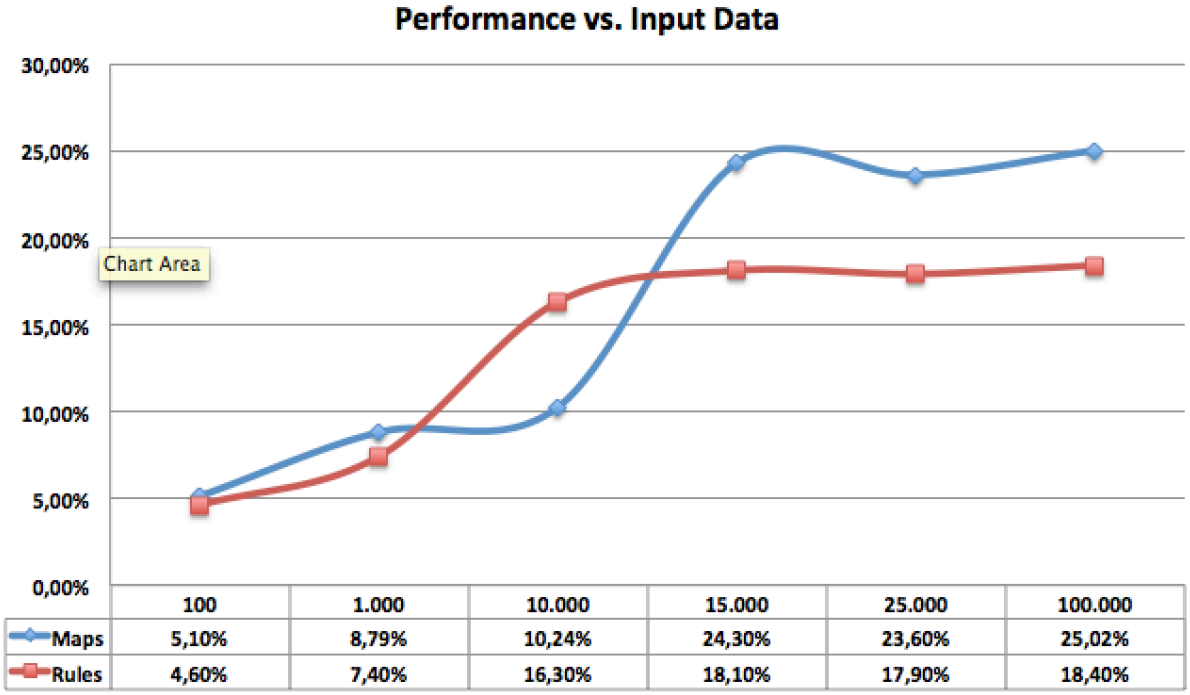
7.3 Performance

Figure 7.3 measures performance of Maps and Rules in the same conditions, considering the number of Input Data.

For a highly reduced database (100 entries), both algorithms are less effective than a « random » behaviour, which would give a performance of *6.25%*.

Between 2K and 12K Input, Rules perform better than Maps. However, when we go over 12K entries, Maps perform a lot better until we reach the optimal behaviour, which gives a performance around ***25%***.

The Rules model doesn't perform as well, giving ***18%*** at his best.
This might be explained by the number of steps used for Reinforcement Learning. Maps were trained over 1 Million iterations, while Rules only were trained over 20K. The same things were observed in the original project.



	100	1.000	10.000	15.000	25.000	100.000
Maps	5,10%	8,79%	10,24%	24,30%	23,60%	25,02%
Rules	4,60%	7,40%	16,30%	18,10%	17,90%	18,40%

Figure 7.3 : Performance vs. Input Data for Maps (Blue) and Rules (Red).

Appendix

Appendix A *: Formal statements of the MSDD Algorithm.*

```
MSDD (H, f, maxnodes)
1. expanded = 0
2. nodes  = ROOT-NODE()
3. while NOT-EMPTY(nodes) and expanded < maxnodes
do
    a. remove from nodes the node n maximising
f(H,n)
    b. EXPAND(n), adding its children to nodes
    c. increment expanded by the number of
       children generated in (b)

EXPAND (n)
1. for i from m down to 1 do
   a. if n.preconditions[i] ≠ '*' then
      return children
   b. for t ∈ Ti do
      i.   child = COPY-NODE(n)
      ii.  child.preconditions[i] = t
      iii. push child onto children
2. repeat (1) for the effects of n
3. return children
```

Source : *http://vega.soi.city.ac.uk/~eu779/publications/SMART-AAMAS03.pdf*

Appendix B *: Rule expanding process used by MSDD.*

Expanding [W, E, E, E, *, *] -> [*, *, *, *, *, *]

PRINTING RULELIST (SIZE:20)
Rule 1 (ID1946) [W, E, E, E, *, *][*, *, *, *, E, *]
Rule 2 (ID1947) [W, E, E, E, *, *][*, *, *, *, A, *]
Rule 3 (ID1948) [W, E, E, E, *, *][*, *, *, W, *, *]
Rule 4 (ID1949) [W, E, E, E, *, *][*, *, *, E, *, *]
Rule 5 (ID1950) [W, E, E, E, *, *][*, *, *, A, *, *]
Rule 6 (ID1951) [W, E, E, E, *, *][*, *, E, *, *, *]
Rule 7 (ID1952) [W, E, E, E, *, *][*, *, W, *, *, *]
Rule 8 (ID1953) [W, E, E, E, *, *][*, *, A, *, *, *]
Rule 9 (ID1954) [W, E, E, E, *, *][*, E, *, *, *, *]
Rule 10 (ID1955) [W, E, E, E, *, *][*, A, *, *, *, *]
Rule 11 (ID1956) [W, E, E, E, *, *][*, W, *, *, *, *]
Rule 12 (ID1957) [W, E, E, E, *, *][W, *, *, *, *, *]
Rule 13 (ID1958) [W, E, E, E, *, *][E, *, *, *, *, *]
Rule 14 (ID1959) [W, E, E, E, *, *][A, *, *, *, *, *]
Rule 15 (ID1960) [W, E, E, E, *, N][*, *, *, *, *, *]
Rule 16 (ID1961) [W, E, E, E, *, E][*, *, *, *, *, *]
Rule 17 (ID1962) [W, E, E, E, *, S][*, *, *, *, *, *]
Rule 18 (ID1963) [W, E, E, E, *, W][*, *, *, *, *, *]
Rule 19 (ID1964) [W, E, E, E, E, *][*, *, *, *, *, *]
Rule 20 (ID1965) [W, E, E, E, A, *][*, *, *, *, *, *]

Starting from the Successor's right end, expand creates new Rules by replacing Wildcards.

Since the Action taken at the next turn is not relevant to this RuleSet, we ignore the Successor's last index.

At first, we tried to work with a version not limited to 1 Non-Wildcard Token in the Successor, e.g. [A, *, W, *, E, *].

Because the execution time of the project exploded, we went back to the non-exhaustive original version, considering 1 Non-Wildcard Token.

Appendix C *: G-Stat (TODO).*