

Cristina Hohan Yu Chang

*Relational Reinforcement Learning with the  
Algorithm TG*

Milan

2010

Cristina Hohan Yu Chang

*Relational Reinforcement Learning with the  
Algorithm TG*

Milan

2010

# *Abstract*

Intelligenza Artificiale (IA) è un settore che mira a comprendere e creare dei metodi per emulare l'intelligenza umana e la sua capacità di risolvere i problemi. Tra i diversi campi di IA, l'apprendimento automatico (Machine Learning (ML)) è quello che si occupa di come sviluppare programmi per computer che migliorano autonomamente con l'esperienza. Questo lavoro presenta i concetti correlati all' apprendimento per rinforzo relazionale (Relational Reinforcement Learning (RRL)), una tecnica di apprendimento che combina l'apprendimento per rinforzo (Reinforcement Learning (RL)) standard con il paradigma di Inductive Logic Programming (ILP) per consentire al sistema di imparare a sfruttare la conoscenza strutturale sul dominio di applicazione. Il paradigma RL definisce su come un agente dovrebbe agire in un ambiente in modo da massimizzare una nozione di ricompensa a lungo termine; ILP, a sua volta, fornisce un sistema di linguaggio logico con cui le conoscenze acquisite vengono memorizzate e utilizzate per azioni future. Il focus principale di questo progetto è l'algoritmo TG, un algoritmo completamente incrementale che si basa su un albero di decisione del primo ordine, e l'integrazione di questo algoritmo nel sistema RRL. Descriviamo l'algoritmo TG e il sistema RRL-TG. Poi esaminiamo il lavoro di pianificazione nel mondo dei blocchi. All'interno di questo semplice dominio dimostriamo che RRL-TG ci permette di impiegare le rappresentazioni strutturali, di astrarre da obiettivi specifici e di sfruttare i risultati delle precedenti fasi di apprendimento al momento di affrontare nuove situazioni.

# *Abstract*

Artificial Intelligence (AI) is an area that aims at understanding and creating methods to emulate human intelligence and its ability to solve problems. Among the various fields of AI, Machine Learning is the one concerned with the question of how to develop computer programs that automatically improve with experience. This work presents the concepts related to Relational Reinforcement Learning (RRL), a learning technique that combines standard Reinforcement Learning (RL) with Inductive Logic Programming (ILP) to enable the learning system to exploit structural knowledge about the application domain. RL is concerned with how an agent ought to take actions in an environment so as to maximize some notion of long-term reward; ILP, in turn, provides a logical language system with which the acquired knowledge is stored and used for future actions. The main focus of this project relies on the TG algorithm, a fully incremental first order decision tree learning algorithm, and the integration of this algorithm in the RRL system. We describe the TG algorithm and the RRL-TG system. We investigate the task of planning in the blocks world. Within this simple domain we demonstrate that RRL-TG allows us to employ structural representations, to abstract from specific goals pursued and to exploit the results of previous learning phases when addressing new situations.

# *List of Figures*

2.1	Standard Model for Reinforcement Learning (SILVA, 2009). . . . .	p. 15
2.2	Example of a learning episode in the Blocks World with four blocks and having on (2.3), indicating the block 2 on block 3, as the goal, with $\gamma=0,9$ .	p. 18
2.3	General Architecture of the TILDE algorithm, where the arrows indicate the direction of the flow of information. Figure taken from (BLOCHEEL; RAEDT, 1998) . . . . .	p. 23
2.4	A relational regression tree . . . . .	p. 25
3.1	Class diagram of the proposed system. . . . .	p. 29
3.2	Interface of the RRL-TG program, showing that the blocks 0 and 1 are on the ground and the block 2 is on block 1. . . . .	p. 30
3.3	General System Architecture. . . . .	p. 31
4.1	Learning curves for the fixed state spaces of 3, 4 and 5 blocks. . . . .	p. 34
4.2	Learning curve for the variable state spaces with MSS values of 30, 50, 100, 200 and 500. . . . .	p. 34
4.3	tree generated for MSS = 200. . . . .	p. 35
4.4	Learning curve for the fixed state space of 3 blocks. . . . .	p. 37
4.5	Respective tree generated after 1000 training episodes for the fixed state space of 3 blocks. . . . .	p. 37
4.6	Learning curve for the fixed state space of 4 blocks. . . . .	p. 38
4.7	Respective tree generated after 1000 training episodes for the fixed state space of 4 blocks. . . . .	p. 38
4.8	Learning curve for the fixed state space of 5 blocks. . . . .	p. 39
4.9	Respective tree generated after 1000 training episodes for the state space of 5 blocks. . . . .	p. 39

4.10	Learning curve for the variable state space with $MSS = 30$ . . . . .	p.40
4.11	Respective tree generated after 1000 training episodes for the variable state space with $MSS = 30$ . . . . .	p.41
4.12	Learning curve for the variable state space with $MSS = 50$ . . . . .	p.41
4.13	Respective tree generated after 1000 training episodes for the variable state space with $MSS = 50$ . . . . .	p.42
4.14	Learning curve for the variable state space with $MSS = 100$ . . . . .	p.42
4.15	Respective tree generated after 1000 training episodes for the variable state space with $MSS = 100$ . . . . .	p.43
4.16	Learning curve for the variable state space with $MSS = 200$ . . . . .	p.43
4.17	Respective tree generated after 1000 training episodes for the variable state space with $MSS = 200$ . . . . .	p.44
4.18	Learning curve for the variable state space with $MSS = 500$ . . . . .	p.44
4.19	Respective tree generated after 1000 training episodes for the variable state space with $MSS = 500$ . . . . .	p.45
5.1	Learning curve for the fixed state space of 3 blocks and $MSS = 200$ . . . .	p.47
5.2	Respective generated tree for the fixed state space of 3 blocks and $MSS = 200$ . . . . .	p.47

# *List of Abbreviations*

<b>AI</b>	Artificial Intelligence
<b>FOLDT</b>	First Order Logical Decision Tree
<b>ILP</b>	Inductive Logic Programming
<b>Java EE</b>	Java Enterprise Edition
<b>Java SE</b>	Java Standard Edition
<b>ML</b>	Machine Learning
<b>MSS</b>	Minimal Sample Size
<b>RL</b>	Reinforcement Learning
<b>RRL</b>	Relational Reinforcement Learning
<b>TDIDT</b>	Top-Down Induction of Decision Tree

# *List of Algorithms*

2.1	Q-Learning . . . . .	p. 17
2.2	Q-RRL . . . . .	p. 22
2.3	TG . . . . .	p. 25



# *Contents*

<b>1</b>	<b>Introduction</b>	p. 10
1.1	Motivation . . . . .	p. 12
1.2	Goal . . . . .	p. 12
1.3	Overview of the Thesis . . . . .	p. 13
<b>2</b>	<b>Basic Concepts</b>	p. 14
2.1	Reinforcement Learning . . . . .	p. 14
2.1.1	Algorithm Q-Learning . . . . .	p. 15
2.2	Inductive Logic Programming . . . . .	p. 18
2.3	Relational Reinforcement Learning . . . . .	p. 20
2.3.1	Top-down Induction of Logical Decision Trees . . . . .	p. 22
2.3.2	TG . . . . .	p. 24
<b>3</b>	<b>Project Development</b>	p. 27
3.1	Development Environment . . . . .	p. 27
3.2	Requirements for the Implementation . . . . .	p. 27
3.3	Project Domain . . . . .	p. 27
3.4	Class Diagram . . . . .	p. 28
3.5	The Program . . . . .	p. 29
<b>4</b>	<b>Tests and analysis</b>	p. 32
4.1	Experiment Configuration . . . . .	p. 32
4.1.1	Fixed Number of Blocks . . . . .	p. 32

4.1.2	Variable Number of Blocks . . . . .	p. 33
4.2	The Results . . . . .	p. 33
4.2.1	Fixed Number of Blocks . . . . .	p. 35
4.2.2	Variable Number of Blocks . . . . .	p. 40
<b>5</b>	<b>Conclusions and Future Work</b>	p. 46
	<b>Bibliography</b>	p. 49

# 1 *Introduction*

Relational Reinforcement Learning (RRL) is a learning technique that was originated from an effort to improve the conventional Reinforcement Learning (RL) and eliminate some of its limitations. It combines RL with Inductive Logic Programming which allows the learning system to use a more expressive representation language to store knowledge. This thesis goal is to study and analyze the RRL algorithm, a concept from the Machine Learning paradigm, an Artificial Intelligence sub-field.

The concept of Artificial Intelligence has always been part of mankind's imagination. The first references to the concept of intelligent machines can be traced back to the Greek mythology, where the god Hephaestus (or Vulcan in Roman mythology) lived in a palace served by mechanical robots (WIKIPÉDIA, a). The area arouses many ethical issues and has already led to several fictions addressing these discussions and weaving the diverse possibilities of the future, such as the movies *AI: ARTIFICIAL INTELLIGENCE* (2001), *I, ROBOT* (book : 1950, movie: 2004), and *MATRIX*(1999), and many others.

The AI term itself was coined only in 1956 by John McCarthy, Marvin Minsky, Claude Shannon, in the first conference devoted to the subject (WIKIPÉDIA, b). But what is AI? Every artificial entity that somehow manages to reproduce the ability or the reasoning process of human beings to solve problems is an AI. According to Russel and Norvig (2003), to determine if a machine is intelligent or not, it should have the following capabilities (RUSSELL; NORVIG, 2003):

- **Natural Language Processing** to enable it to communicate successfully.
- **Knowledge Representation** to store the knowledge that it knows or believes to know.
- **Automatic Reasoning** to make use of the information stored to answer questions or to reach new conclusions.
- **Learning** to adapt to new circumstances and identify and explore patterns.

Currently the area of AI is fragmented into several sub-fields, each approaching the subject from different perspectives and developing their own methods, experimenting and improving, merging or incorporating them with other methods to complement faults, improve performance, discover new paths. This work focuses on the sub-area of Machine Learning (ML), more specifically, the concept of Reinforcement Learning (RL) and one of its variants, the Relational Reinforcement Learning (RRL) with the TG algorithm (DRIESSENS; RAMON; BLOCKEEL, 2001).

In general, the area of ML is dedicated to developing algorithms and techniques that enable machines and programs to learn and improve their performance in tasks through experience. Depending on the methodology used, these algorithms are divided into different paradigms (SILVA, 2009).

One of these learning paradigms is the RL one, as mentioned above, which results from a combination of a number of areas, including: statistics, computational methods, psychology, theory of optimal control. Its basic idea is to reward an agent's action when it results in a desirable state, or charge a penalty if the action takes the agent to an undesirable state. The goal of the agent is to learn which policy (i.e., which mapping from states to actions) will result in greater cumulative reward at the end of the task.

From this concept appeared a new line of thought, the RRL, which integrates the concepts of RL with the concepts of Inductive Logical Programming (ILP), another paradigm of ML. The basic principle of ILP is to generate descriptive predicates of states from examples and prior knowledge of the environment, worrying more about learning concepts and ignoring the rest of ML.

The RRL combines RL techniques, specifically Q-learning (SUTTON; BARTO, 1998), with the ILP techniques, providing a more expressive language to represent states, actions and Q functions. The RRL uses a relational regression algorithm that has as input a description of a state and an action, returning the corresponding Q value as a result and thus learning the Q function (DZEROSKI; RAEDT; BLOCKEEL, 1998).

The application domain where the RRL agent is observed is the Blocks World. In the Blocks World, the states are all the possible combinations of the blocks position (one block over another, or on the floor). The available actions are to move a block from one place to another, taking into account if it can be moved (if there is another block over it).

## 1.1 Motivation

The RL paradigm has some limitations that often make it impractical for real-world problems. The algorithms take a long time to reach an optimal solution, it does not reuse the acquired knowledge to perform similar tasks, that must be solved from scratch (LAZARIC, 2008). Another drawback of RL is the use of tables (State x Actions) that store the respective Q values, resulting in scalability problems for very large spaces. The use of ILP techniques allows to generalize the states and specific goals by using variables and describing them relationally (DZEROSKI; RAEDT; BLOCKEEL, 1998).

From the combination of ILP methods with RL ones came the RRL, the purpose of which stems from an attempt to overcome the limitations that RL presents. The first version of the RRL paradigm was based on the algorithm TILDE (BLOCKEEL; RAEDT, 1998), a system that generates a First-Order Logic Decision Tree (FOLDT). Later this algorithm was combined with the incremental algorithm G (CHAPMAN; KAELBLING, 1991) from where came the TG algorithm (DRIESENS; RAMON; BLOCKEEL, 2001), in an attempt to boost up the RRL performance.

In this thesis a research and a study on the subject is presented, followed by the implementation and testing of the TG algorithm and an analysis of the obtained results in order to evaluate its performance.

## 1.2 Goal

This work aims to describe the concepts related to the RRL paradigm and the TG algorithm, from the reasons that led to its conception to its current stage of development, implement the RRL-TG system and apply it to the Blocks World, observe its operation and performance, and then analyze the obtained results. The project consists of the following steps:

- Study and understanding of the related concepts;
- Implementation of TG algorithm in Java language;
- Tests in the Blocks World, in order to repeat and analyze the results of pioneer work (DRIESENS, 2004; DRIESENS; RAMON; BLOCKEEL, 2001) and better understand the RRL-TG system.

## 1.3 Overview of the Thesis

This thesis is organized as follows: Chapter 2 presents the fundamental concepts related to the proposed theme, such as Reinforcement Learning and the Q-Learning algorithm, having in mind the Blocks World. In Chapter 3 is documented the development of the project, its requirements and development environment, as well as the description of the application domain where the TG algorithm is applied, and a brief description of the implemented program. Chapter 4 describes the tests applied to the algorithm implemented and the results obtained, including an analysis of the results. Finally, Chapter 5 presents the reached conclusions from the analysis, and includes some suggestions for future work.

## 2 *Basic Concepts*

This chapter describes the concepts related to this project, in order to provide a theoretical basis necessary to understand the objective of this project and the obtained results.

### 2.1 Reinforcement Learning

The basic idea governing the Reinforcement Learning (RL) concept has as reference the human understanding of the learning process through interaction with the environment, through the perception of the environment, the trial, the experiment and the comprehension of the consequences of these actions. The exercise of this connection to the environment produces a wealth of information about what to do in order to achieve goals. No doubt these interactions are the major source of knowledge about the environment for a living being (SUTTON; BARTO, 1998).

RL is a computational approach to that learning process, and its algorithms are strongly associated with the techniques of Dynamic Programming, a method that solves complex problems by dividing them into smaller steps (BELLMAN, 1957). An agent is placed in a world over which he/she has no priori information, and is given the responsibility for achieving a particular goal. To the agent no information on what actions should be taken is given, as is done in most approaches to ML, but it is the agent's responsibility to explore the environment with its sensors and find out what actions, or sequence of actions, grant greater reward, through trial and error. Figure 2.1 shows the interaction of the agent with the environment. At each step  $t$  of the interaction, the agent receives an input that describes the current state  $s$  of the environment, and then chooses an action. The action modifies the state according to the dynamics of the environment and the goodness of the transition to the new state is communicated to the agent through a reinforce  $r$  (LAZARIC, 2008).

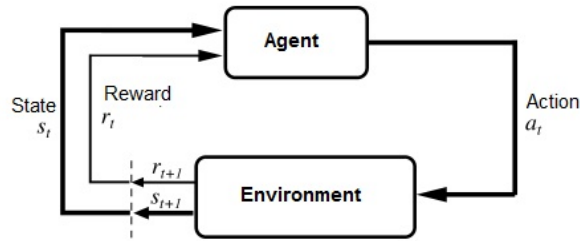


Figure 2.1: Standard Model for Reinforcement Learning (SILVA, 2009).

In addition to the agent and the environment, there are four other important elements that define an RL system:

- a policy  $\pi$ : function that maps states to actions and defines the behavior of the agent;
- an evaluation function  $V$ : for each state, while the reward function indicates the best action to be taken in that state, the function  $V$  specifies what the best course of actions over the problem  $V : S \rightarrow \mathfrak{R}$ ;
- a model of the environment, defined by:
  - a set of states  $S$
  - a set of actions  $A$
  - a transition function  $\delta(s, a) : S \times A \rightarrow S$
  - a reward function  $r : S \times A \rightarrow \mathfrak{R}$

At each point in time, the environment can be in one of the states  $s_t$  of  $S$  observed by the agent, which selects an action  $a_t = \pi(s_t) \in A$  to be executed in accordance with the policy  $\pi$ . The action results into a new state  $s_{t+1} = \delta(s_t, a_t)$  observed by the agent, that also receives a reward  $r_t = r(s_t, a_t)$  as a result of the executed action. The goal of the agent is to find a policy  $\pi^* : S \rightarrow A$  that maximizes the value of the function  $V^\pi(s)$  for every state  $s \in S$ .

### 2.1.1 Algorithm Q-Learning

One of the most popular approaches to the RL paradigm is known as Q-Learning, an iterative algorithm developed by Watkins in 1989 (WATKINS; DAYAN, 1992) to learn



the optimal policy  $\pi^*$  without a model of the environment (e.g., the transition function  $\delta$ ). This algorithm has an action-value function that assigns a value  $Q$  for each pair state-action allowing an approach to an optimal policy  $\pi^*$ , where  $r(s,a)$  is the reward (or reinforcement) received when performing the action  $a$  in state  $s$ ,  $\gamma$  is the discount factor ( $0 \leq \gamma < 1$ ) and  $V^{\pi^*}(\delta(s,a))$  is the value of the next state reached.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}(Q^{\pi^*}(s,a)) \quad (2.1)$$

With

$$Q^{\pi^*}(s,a) = r(s,a) + \gamma V^{\pi^*}(\delta(s,a)) \quad (2.2)$$

At the beginning, the  $Q$  values are initialized to a fixed value. Each time the agent receives a reward, the  $Q$  values are recalculated and adjusted based on new information from each combination of state  $s$  of  $S$ , and action  $a$  of  $A$ . The goodness of taking an action from a given state is implicit into the  $Q$  value.

The  $Q$ -function can be approximated using a lookup  $Q$  table  $\widehat{Q}$ :

$$\widehat{Q}(s_t, a_t) \leftarrow r_t + \gamma \max_{a'} \widehat{Q}(s_{t+1}, a_{t+1}). \quad (2.3)$$

There are several exploration policies that the agent can use to explore and learn about the environment. These policies guides the agent's actions toward those that are most likely to achieve a desired outcome, by mapping states of the environment to the actions the agent ought to take in those states. The most famous exploration policies are the  $\epsilon$ -greedy policy and the Softmax policy (SUTTON; BARTO, 1998). In the  $\epsilon$ -greedy policy, the agent behaves greedily (e.g., the agent exploits current knowledge to maximize immediate reward, spending no time at all to explore alternatives that might be better) most of the time (at  $1-\epsilon$  times), but can, every once in a while (at  $\epsilon$  times), select equally among all actions, allowing the environment exploration. One drawback of the  $\epsilon$ -greedy action selection is that during exploration time, it chooses equally among all action, e.g., the chances of choosing the worst-appearing action is as likely as to choose the next-to-best-action. The Softmax action selection solves this problem by assign different weights to the actions, according to their estimated value. The most common Softmax method uses a Boltzmann distribution, where the choice of an action  $a_i$  in a state  $s$  is based in the following probability distribution (DZEROSKI; RAEDT; BLOCKEEL, 1998):

$$Pr(a_i|s) = \frac{T^{-\widehat{Q}(s,a_i)}}{\sum_j T^{-\widehat{Q}(s,a_j)}}. \quad (2.4)$$

The temperature  $T$  controls the exploration of the environment by the agent. Low values of  $T$  makes the agent give preference to actions with high  $Q$  values, while a higher value causes the agent to further explore other options.

---

**Algorithm 2.1** Q-Learning
 

---

```

//inputs: information about the current state
//output: action to be performed
for each  $s, a$  do
  initializes table  $Q(s, a) = 0$ 
end for
generates an initial state  $s_0$ 
repeat
  select an action  $a_t$  and runs
  receives a reward  $r_{t+1}$ 
  observes the new state  $s_{t+1}$ 
  updates the table  $Q(s, a)$  as follows:
   $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t [r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$ 
   $s_t \leftarrow s_{t+1}$ 
until  $s$  is the goal state

```

---

The Q-learning algorithm, described in Algorithm 2.1, takes as input information about the current state and provides as output the action to be taken. With the information about the current state, the algorithm selects an action based on the probability distribution function 2.4 and executes it. When executing the chosen action, the agent receives an immediate reward, observes the resulting state, and then updates the values from the  $Q$  table according to the function 2.3.

The fact that the Q-Learning algorithm stores all the  $Q$  values in a table makes it impractical for all but the smallest state-spaces, due to the excessive memory usage to store it; in addition, whenever the goal of the problem is changed, the agent must relearn the function  $Q$  from scratch even if the goals were similar, which leads to a certain waste of prior knowledge.

This way of representing knowledge in tables is inadequate for planning problems. For example, in the Blocks World, the states are representations of all the possible combinations of the blocks positions. Consider a world of three blocks (A, B and C), the possible states and their actions are: (1) all the blocks on the floor - move (A, B), move (A, C), move (B, A), move (B, C), move (C, A), move (C, B), (2) block A on block B, block B on

the floor, block C on the floor - move (A , C), move (C, A), move (A, Floor), (3) block A on block C, block C on the floor, block B on the floor - move (A, B), move (B , A), move (A, Floor), (4) block B on block A, block A on the floor, block C on the floor - move (B, C), move (C, B), move (B , Floor), and so on. The goal of the agent within this domain is to put a certain block on another block, specified a priori. The biggest problem with this form of representation is the mandatory presence of any state and possible action, which makes it impractical for situations where the number of blocks is too large.

Figure 2.2 illustrates an example of Q-learning applied to the Blocks World. The illustrated episode consists of three state-action pairs that run from left to right. When the goal (on (2.3)) is reached after the execution of the last action, the Q values are calculated from right to left, as shown in the figure. More information about the World of blocks are provided in Chapter 3.

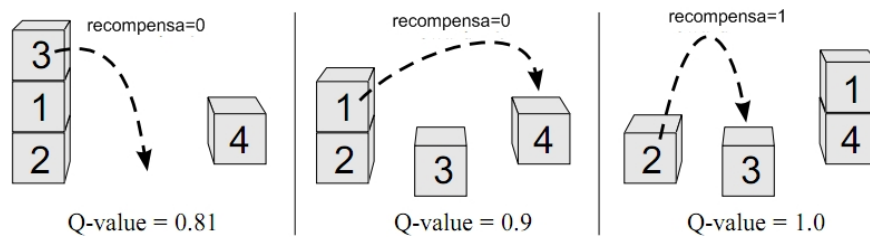


Figure 2.2: Example of a learning episode in the Blocks World with four blocks and having on (2.3), indicating the block 2 on block 3, as the goal, with  $\gamma = 0,9$ .

## 2.2 Inductive Logic Programming

The importance of the Inductive Logic Programming (ILP) lies in its ability to generalize and represent the knowledge. The ILP system consists of a learner to whom are given some training examples, from which he/she draws new relations and rules by using known relationships from its background knowledge. Knowledge is represented by a logic programming language, a computational formalism that uses first-order logic to express knowledge (as predicates), and the inference to manipulate it and draw new hypotheses (LAVRAC; DZEROSKI, 1994).

The use of an ILP system solves the Q-learning scalability problem to represent the Blocks World, by using First Order Logic (FOL) that allows a relational description of the domain. For example:

"The block A is on block B"

The objects are the blocks A and B, while the relationship between both blocks is "is on". The states and actions are represented by predicates and following some rules one can reach a new state. In this representation, only the necessary states and actions appear. In addition, one can reuse the acquired knowledge to achieve similar goals.

Systems that use this type of representation language, are called relational learning system, and consist of (SILVA, 2009):

- a **formal language** - that specifies the sentences that can be used to express knowledge;
- a **semantic** - that specifies the meaning of the sentence in the language, both as entry and exit;
- a **reasoning theory** - which is a possible non-deterministic specification of how an answer can be derived from a knowledge base.

The formal language defines the symbols and how they should be organized. And these are divided into variables, terms, atoms, predicates, facts and rules.

A **variable** is a sequence of characters that indicates a **term**, and can be a variable or a constant. The **predicates** are words that refer to relations and the **atoms** are found in the form  $p$  or  $p(t_1, \dots, t_n)$  where  $p$  is a predicate and  $t_i$  denotes a term. Some examples of atoms are: Happy(John) and PlayBasket(Assis, France) (SILVA, 2009).

The **fact** is considered as an atom clause. The **rules** are clauses with two literals using the form  $A \rightarrow B$ , where A is known as the body, also known as a premise and B is the head of the rule, referring to the conclusion.

For example, assuming a prior knowledge of the atoms  $father(Y, X)$ , where Y is the father of X, and  $female(X)$ , where X is female, one can correctly infer the hypothesis  $daughter(X, Y)$ , X is the daughter of Y, given the following rule (implication).

$$daughter(X, Y) \leftarrow female(X), father(Y, X).$$

It is extremely important to take special care when specifying the knowledge base, since its predicates specify generally valid knowledge for all the training samples. Given

an appropriate basis, the learning will be effective and if not, the learning will be difficult if not impossible. This emphasizes the importance of correctly specify the important predicates in the knowledge base.

The focus of this project is in the First Order Logical Decision Tree (FOLDT) as means of the environment representation. Blockeel and De Raedt (1998), has the following definition for FOLDT:

**Definition 2.1 (FOLDT)** *A first-order logic decision tree is a binary decision tree in which*

1. *the nodes of the tree contain a set of literals, and*
2. *different nodes may share variables, under the following restriction: a variable introduced in a node should not appear in the right branch of that node. This follows from the fact that this variable is existentially quantified in the conjunction of that node. The sub-tree to the right is only relevant when the conjunction fails, from which situation any reference to that variable becomes meaningless.*

## 2.3 Relational Reinforcement Learning

The Relational Reinforcement Learning (RRL) is a learning technique that combines RL with ILP. The goal of RRL is to abstract the specific identities of states, actions, and even objects in the environment, and identify them only by referring the objects through their properties and relations between them. When defining objects by their properties (shape, size, color) and not by their identity (apple, box, cup), any object that has these properties is handled correctly by the learning agent.

A definition of RRL is given as (DRIESSENS, 2004):

**Definition 2.3 (Relational Reinforcement Learning)** *The Relational Reinforcement Learning can be defined as follows:*

**Data:**

1. *a set of states  $S$ , represented in relational format,*
2. *a set of actions  $A$ , also represented in relational format,*
3. *a transition function  $\delta : S \times A \rightarrow S$ ,*

4. a reward function  $r : S \times A \rightarrow \mathfrak{R}$ ,
5. a generally valid prior knowledge about the environment

**Find** an optimal policy  $\pi^* : S \rightarrow A$  that maximizes the value function  $V^\pi(s_t)$  for all  $s_t \in S$ .

Each state  $s_i$  is represented by the relationship between the blocks, and has a set of possible actions that can be performed by the system. The precondition for an action to be performed is that the block to be moved and the block, or space on the ground, where it is being moved are both unobstructed. The postcondition of each action is that the moved block is on the block (or floor space) to where he was moved. The goal of the system (set of goal states) is embedded in the reward function, whereas in this project, the transition function  $\delta$  was considered as a deterministic function. Prior knowledge may include information such as the partial knowledge about the effect of actions, similarity between different states, among others.

RRL was developed to solve the problems that RL presents, as mentioned before. Instead of using a table for storing Q values, RRL system uses the information collected on Q values of different state-action pairs in a relational regression algorithm that builds a generalized Q function. Its quality is in the fact of using a relational representation of the environment and of the available actions, and by the fact that it uses a relational regression algorithm to build the Q function. This algorithm avoids the use of specific identities for states, actions and objects in its function, counting with structures and present relationships in the environment to define similarities between the state-action pairs and predict their corresponding Q values.

The Q-RRL algorithm (algorithm 2.3), which is a *Q-Learning* algorithm adapted to RRL, begins by initializing the Q function and then learns episodes like any other Q-Learning algorithm. To choose an action, the algorithm uses the same probability function used by the standard Q-Learning. During each learning episode, all the found states and the respective taken actions are stored, along with the rewards related to each pair state-action. At the end of each episode, when the system reaches its goal, the system approaches the Q value for each pair state-action using backward propagation of the reward function and the approximation of the actual Q function.

The algorithm then provides a set of triplets (*state, action, Q value*) for the relational regression mechanism that will then use this set of examples to update the estimates of the Q function, and so proceeding with the next episode.

---

**Algorithm 2.2** Q-RRL

---

```

//inputs: information about the current state
//output: action to be performed
initializes the hypothesis  $\widehat{Q}_0$  function Q
 $e \leftarrow 0$ 
repeat
  textit Examples  $\leftarrow \emptyset$ 
  generates an initial state  $s_0$ 
   $i \leftarrow 0$ 
  repeat
    select an action  $a_i$  for the state  $s_i$  using the current hypothesis for  $\widehat{Q}_e$  and executes it
    receives a reward  $r_i$ 
    observes the new state  $s_{i+1}$ 
     $i \leftarrow i + 1$ 
  until  $s_i$  is the goal state
  for  $j = i - 1$  to 0 do
    generates example  $x = (s_j, a_j, \widehat{q}_j)$  where  $\widehat{q}_j \leftarrow r_j + \gamma \max_a \widehat{Q}_e(s_{j+1}, a)$ 
    Examples  $\leftarrow$  Examples  $\cup x$ 
  end for
  updates  $\widehat{Q}_e$  using Examples and a relational regression algorithm to produce  $\widehat{Q}_{e+1}$ 
   $e \leftarrow e + 1$ 
until there is more episodes

```

---

As in algorithm 2.1, the Q-RRL algorithm receives as input the current state and provides as output the action to be performed.

Initially the relational regression mechanism used was TILDE, a logic decision tree, which afterwards was extended with an incremental algorithm to form the TG algorithm, both described below.

### 2.3.1 Top-down Induction of Logical Decision Trees

Top-down Induction of Logical Decision Trees (TILDE) is an algorithm that generates a first-order logic decision tree to perform classification, regression or assembly of data, using facts instead of attributes. Figure 2.3 shows the overall architecture of the system.

The system consists of a main core that contains a genetic Top-Down Induction of Decision Trees (TDIDT) algorithm and three auxiliary modules (BLOCKKEEL; RAEDT, 1998):

- a module that implements the ILP system part (ILP *utilities*) contains the clause refinement operator (to generate the set of tests that will be implemented on the

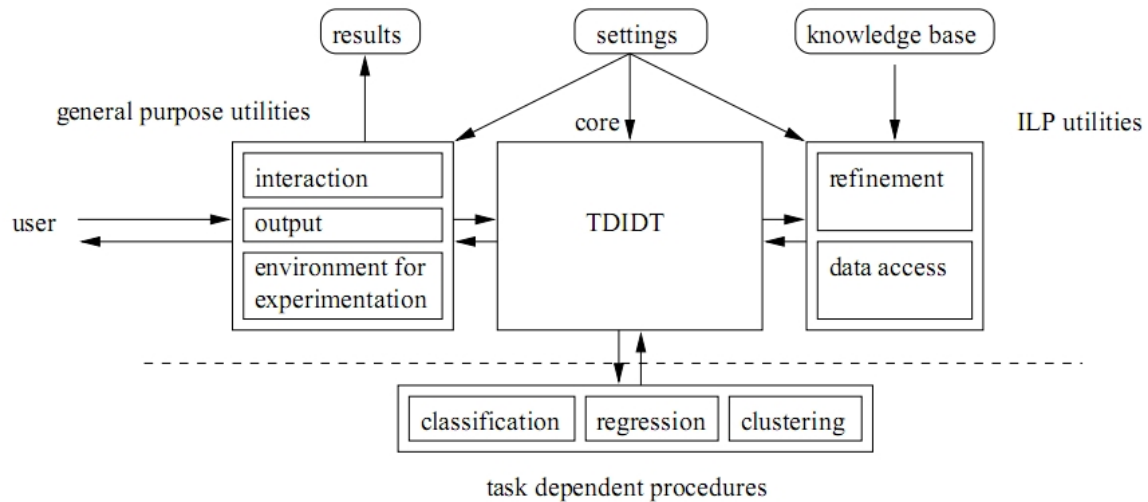


Figure 2.3: General Architecture of the TILDE algorithm, where the arrows indicate the direction of the flow of information. Figure taken from (BLOCKEEL; RAEDT, 1998)

node), as well as all the functions for data access ;

- a module that implements all the procedures related to the three options of operation: classification, regression and clustering of data (*TDIDT task dependent procedures*);
- a module that implements everything related to the user interface (*General purpose utilities*).

The system receives two types of input: a configuration file (*settings*), specifying the system parameters, and a background knowledge with examples and prior knowledge, generating one or two output files containing the results of the inductive process (*results*). The arrows indicate the flow of information within the system.

Within the context of RRL (shown in algorithm 2.2), the TILDE system is one of relational regression algorithms that can be used to process the examples generated by the RRL. Each example received by TILDE is tested through the tree until it reaches a leaf. Because it is not incremental, the TILDE system stores all the found state-action pairs (not just those generated by the episode  $e$ ) and the most recent  $\hat{q}$  value for each pair. A relational regression tree  $\hat{Q}_e$  is induced from the examples  $(s, a, q)$  after each episode  $e$ . This tree is then used by the RRL to select actions in the episode  $e + 1$ . Further details about the TILDE algorithm can be found in (BLOCKEEL; RAEDT, 1998).

The use of the TILDE algorithm in the RRL presents four identified problems (DRIESENS;



RAMON; BLOCKEEL, 2001):

- the system must keep track of an ever increasing number of examples: for each state-action pair found, the corresponding Q value is maintained.
- when a state-action pair is found for the second time, the new Q value needs to replace the previously stored value, which means that the former example must be found in the background knowledge and be replaced.
- the tree is reconstructed from the starting point for each episode. This, as well as the replacement of redundant examples are procedures that take more and more processing time as the set of examples grows.
- leafs of the tree should identify groups of similar state-action pairs in the sense that they have the same Q value. When the Q value of a state-action pair is updated on the leaf, the Q value of all other pairs found on the same leaf should be automatically updated, which does not happen.

To solve these problems, it was necessary an incremental induction algorithm. The next section deals with this algorithm.

### 2.3.2 TG

The TG algorithm (DRIESSENS; RAMON; BLOCKEEL, 2001) described in algorithm 2.4 is a combination of the TILDE algorithm (BLOCKEEL; RAEDT, 1998), that constructs a first order of classification and regression tree, and the algorithm G (CHAPMAN; KAELBLING, 1991), that uses a number of statistical values related to the performance of each possible extension for each leaf of the tree to build it incrementally.

As the TILDE, TG uses a relational representation language to describe the examples and tests that can be used in the regression tree.

The RRL sends all the generated examples in one episode to the TG, which, one by one, tests the examples through the tree until it reaches a leaf. The Q value of the reached leaf is updated with the new value (the Q value maintained at each leaf is the average of the Q values of the examples that reach it). The statistics for each leaf consist of the number of examples on which each possible test succeeds or fails as well as the sum of the Q values and the sum of the squared Q values for each of the two subsets created by the test. These statistics can be calculated incrementally and are sufficient to compute

---

**Algorithm 2.3** TG

---

```

//inputs: examples provided by the RRL
//output: TG tree
starts by creating a tree with a single leaf and empty statistics
for each training example that becomes available do
  sort the example down the tree using the tests of the internal nodes until it reaches
  a leaf
  updates the statistics in the leaf according to the new example
  if the statistics in the leaf indicate that a new split is needed then
    generate an internal node using the indicated test
    grow two new leaves with empty statistics
  end if
end for

```

---

whether some test is significant, i.e., whether the variance of the Q values of the examples would be reduced sufficiently by splitting the node using that particular test. A standard F-test (SNEDECOR; COCHRAN, 1991) with a significance level of 0.001 is used to make this decision. If the test is relevant with a high confidence, then the leaf is split into two new leaves, and each of them receives a Q value based on statistics obtained from the test used to split its parent node. Later, this value is updated as new examples are sorted in the leaf. The TG algorithm receives as inputs the examples provided by the RRL, and provides as output a tree with the knowledge stored.

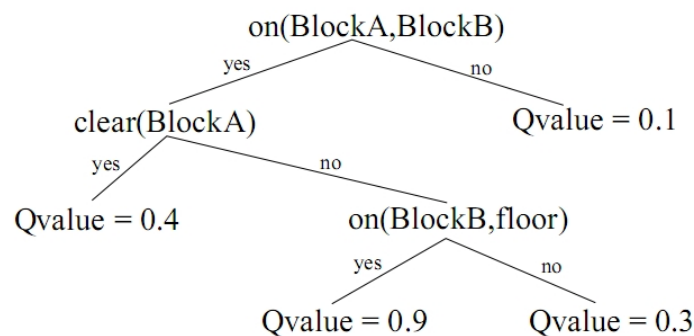


Figure 2.4: A relational regression tree

For example, take a newly created leaf, with zero statistics. At each iteration from the system, new examples are generated and delivered to the tree. Each sample is sorted through the tree until it reaches a leaf, where it is stored and the leaf's Q value is updated with the Q value of the example (the Q value of the leaf is the average value of the Q values from the samples stored in the leaf). After a minimum sample size (MSS) of examples is

stored in a leaf, the TG algorithm applies all possible tests to this set of examples, and verifies which one offers the best split of the examples into two subsets. The one offering the best split of the examples in the leaf is chosen as the leaf's test and two new leaves with zero statistics are created from it.

The tests can be of the following types:

- $\text{equal}(X,Y)$  the block X is equal to the block Y in the sense that block X is the block Y
- $\text{on}(X,Y)$  X is the block directly on the block Y
- $\text{above}(X,Y)$  block X is on the stack of blocks that are on the block Y
- $\text{clear}(X)$  block X is free

The figure 2.4 provides an example of a first logic regression tree.

Once built after the learning stage, the tree is then used as a background knowledge to aid the agent to choose future actions.

Being incremental, the TG algorithm does not need to rebuild the tree every episode, and does not need to keep all the generated examples during the various episodes.

## 3 *Project Development*

This chapter presents how the project was implemented, its development environment and the minimum requirements for its implementation.

### 3.1 Development Environment

The project was entirely developed using the software *open source* ECLIPSE, an integrated development environments (IDE) that support Java technology. In addition, the calculation of the statistics of each leaf is aided by the use of the library *Commons Math* that can be found at: <http://commons.apache.org/math/>. The rendering of the program's interface uses an adapted version of the class *GraphicsUtil* that can be obtained from: <http://www.win.tue.nl/~sthite/sp/GraphicsUtil.java>. For the graphs renderization, it was used a Java package called *Plot*, obtained from: <http://homepage.mac.com/jhuwaldt/java/Packages/Plot/PlotPackage.html>.

### 3.2 Requirements for the Implementation

To run the project, it is necessary to install the Java Runtime Environment (JRE) version 6 and the Eclipse IDE, as well as a minimum configuration of 1 GB of RAM.

### 3.3 Project Domain

The RRL-TG algorithm in this project is applied to the Blocks World. The Blocks World is a classic planning problem often used in AI, for its simple and clear characteristics. The Blocks World consists of:

- a smooth surface upon which the blocks are placed. In this case, the smooth surface will be identified as the "floor" and is represented in the program by "F".

- a set of identical blocks identified by letters or numbers.
- blocks can be moved to the floor or on top of another block, if the moving block is free to move, and the space to which it is being moved is free.
- just one block can be moved at a time.

The states are represented by facts or predicates, as follows:

- $\text{on}(X,Y)$  describes that the block  $X$  is on  $Y$ , and  $Y$  can take on the role of a block or floor;
- $\text{clear}(X)$ , indicating that on block  $X$  there is no other block.

The goal of the robot is to calculate an optimal policy  $\pi^*$  that will allow it to achieve a certain goal (for example, on (A, B)) by the quickest means. Each configuration of the Blocks World is a different state  $s$ , and each act of moving a block from one place to another is an action  $a$ , which changes the environment. Each action performed changes the state of the environment and so the robot proceeds until it reaches the goal state. Upon reaching the goal state, it receives a reward  $r$  of value 1. For the actions that result in a state other than the goal state, are given the value 0. All steps taken by the robot from the initial state to the goal state and the set of triplets state-action-reward ( $s, a, r$ ) generated by these steps constitute one episode.

The test nodes of the tree (from the TG algorithm) can be of the following types:

- $\text{on}(X,Y)$  tests whether the block  $X$  is on block  $Y$ ;
- $\text{clear}(X)$  tests whether the block  $X$  is free;
- $\text{above}(X,Y)$  tests whether the block  $X$  is in the stack of blocks that is on  $Y$ ;
- $\text{equal}(X,Y)$  tests if  $X$  and  $Y$  are the same block.

### 3.4 Class Diagram

The figure 3.1 provides the overall class diagram of the project. Class *MainUI* handles the graphical user interface and the communication between the classes; class *QLearning* generates the examples and delivers them to the TG class, where they are processed and

sorted through the generated tree; class *Grid* handles the animation of the environment; *QLearning* process the RRL part of the system, while the remaining classes, *ClearPrepos*, *Block*, *OnPrepos*, *Goal*, *State*, *QSA* and *Action* offer the structure that describes the environment and the application domain; *GraphicsUtil* provides functions to render the blocks in the graphical interface.

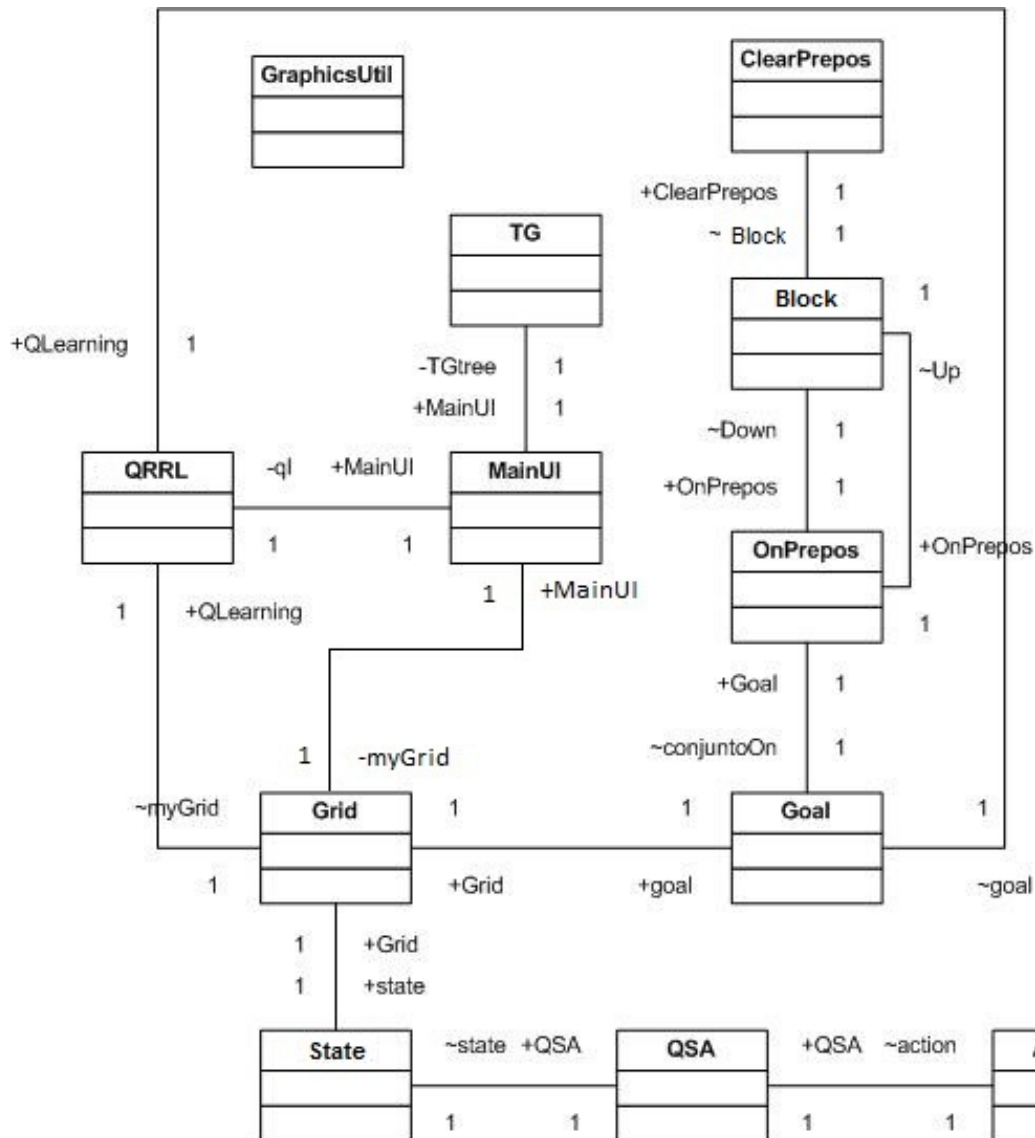


Figure 3.1: Class diagram of the proposed system.

### 3.5 The Program

In this project was developed a program implementing the RRL-TG system, having the Blocks World as the application domain. In this program you can configure the number of blocks in the Blocks World (identified by numbers), set the goal, the discount

factor  $\gamma$ , the temperature  $T$  (remembering that high values favour exploration whereas low values favour exploitation), the reward (value provided when achieving the goal) and the minimum sample size (MSS), as well as an option to set the time between an action and another, for better visualization of the operation of the program.

It is possible to execute the program in three ways: step by step, to observe the actions taken by the system; per episode, to run an entire episode; and per  $n$  cycles, to execute the algorithm  $n$  times. Figure 3.2 illustrates the interface of the program.

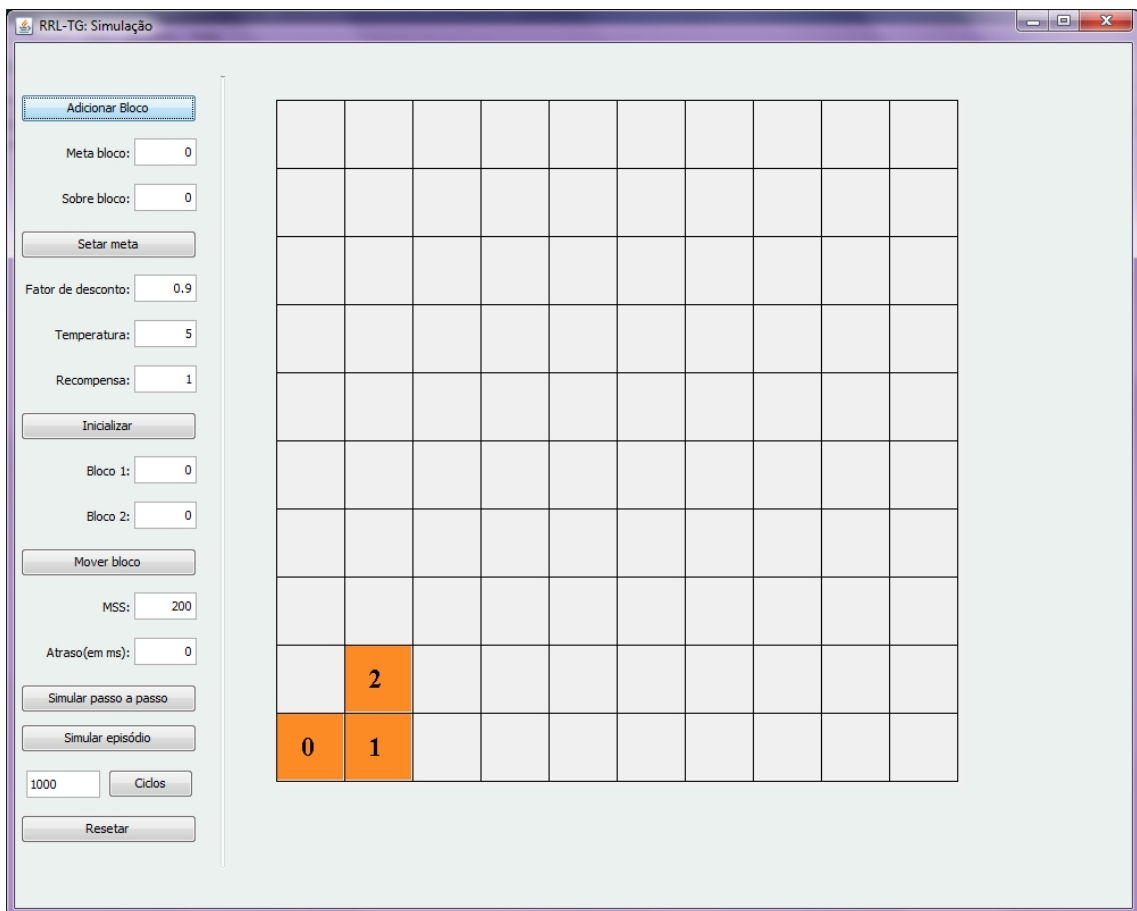


Figure 3.2: Interface of the RRL-TG program, showing that the blocks 0 and 1 are on the ground and the block 2 is on block 1.

Figure 3.3 shows the overall architecture of the system. The application domain is the environment to be explored by the agent, in this case, the Blocks World. The RRL-TG interacts with the environment and exploits its prior knowledge to choose its actions. The tree generated by the TG algorithm is stored in the background knowledge. In it are also the pre and post conditions of actions, which were described in section 2.3.

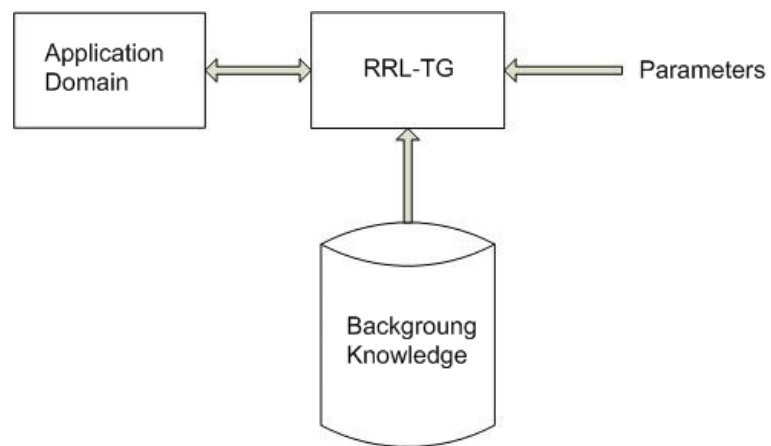


Figure 3.3: General System Architecture.



## 4 *Tests and analysis*

In this chapter are described the realized tests, the used parameters and the analysis of the obtained results by the implemented RRL-TG system.

### 4.1 Experiment Configuration

For all the tests, with the purpose to observe the performance of the algorithm under different situations, we used the following parameters: discount factor  $\gamma = 0.9$  (eq. 2.3), temperature  $T = 5$  with a decay factor of 0.95 (eq. 2.4)<sup>1</sup> and reward  $r = 1$  (eq. 2.3) if the agent reaches the goal, 0 otherwise.

Two series of tests are performed, as described below.

#### 4.1.1 Fixed Number of Blocks

The first series of tests compares the learning curve of the TG algorithm in a fixed state space (DRIESENS; RAMON; BLOCKEEL, 2001). There are three state spaces tested: with 3 blocks, with 4 blocks and with 5 blocks.

For the fixed-space tests 1000 training episodes were carried out. For every training episode the Q function built by the RRL system is translated into a deterministic policy and tested on 10 000 randomly generated test episodes that followed a greedy policy - this removes the influence of the exploration strategy from the test results. A reward of 1 was given if the goal state was reached in the minimal number of steps, 0 otherwise. In this work, the minimal number of steps was set as the minimal number of steps to achieve the goal state in the worst case (in the original test (DRIESENS; RAMON; BLOCKEEL, 2001), it is unclear how this value is set). Parameter MSS (p. 26) was set to a value of 200.

---

<sup>1</sup>The choice of an initial high temperature is used to ensure the exploration of the environment by the algorithm

### 4.1.2 Variable Number of Blocks

In the second series of tests, the number of blocks is varied between 3 and 5 blocks while learning, that is, each learning episode of 3 blocks is followed by a learning episode of 4 blocks, which in turn is followed by one of 5 blocks, and so on (DRIESSENS, 2004). Every 100 training episodes, the generated Q functions were tested on 200 randomly generated test episodes, which also applied a greedy policy. There will be a total of 1000 training episodes. The tests will be repeated for different values of MSS, as follows: 30, 50, 100, 200 and 500. The allocation of reward follows the same strategy as the previous series of tests.

## 4.2 The Results

With respect to the original test results (DRIESSENS, 2004; DRIESSENS; RAMON; BLOCKEEL, 2001), the obtained results from the tests showed some differences. Such variations can be attributed to several factors, since some parameters used in the original tests are unknown. In the first test, the MSS value is unknown, as well as the discount factor  $\gamma$ , uncertain in both tests. The first influences the test selection in the leaves, while the second one influences over the convergence of the algorithm. The graph in Figure 4.1 shows the original test result (DRIESSENS; RAMON; BLOCKEEL, 2001) for the fixed state spaces of 3, 4 and 5 blocks. The corresponding generated tree was not available in it. The graph in Figure 4.2 shows the results for the variable state spaces with MSS values of 30, 50, 100, 200 and 500, while Figure 4.3 shows the generated tree corresponding to  $MSS = 200$ . These figures represent the results expected by the project.

The charts in figures 4.4, 4.6, 4.8, 4.10, 4.12, 4.14, 4.16, 4.18 show the average reward received in each set of tests. The Y ordinate shows the average reward obtained per episode as the tree is being built.

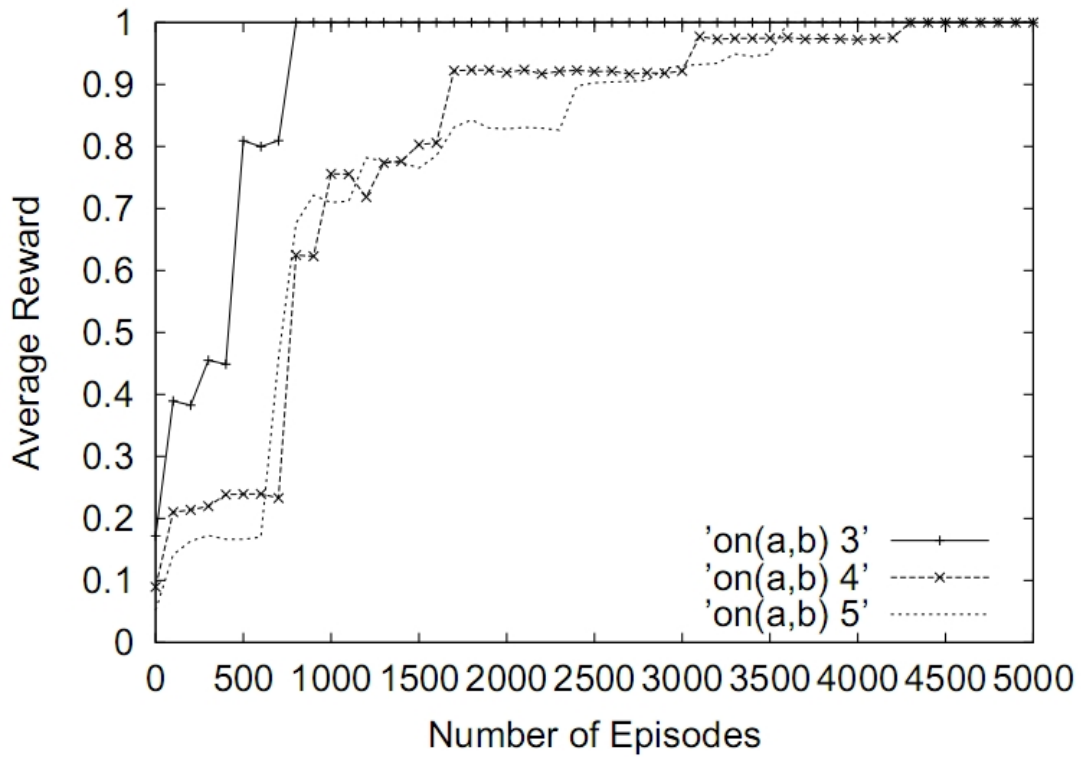


Figure 4.1: Learning curves for the fixed state spaces of 3, 4 and 5 blocks.

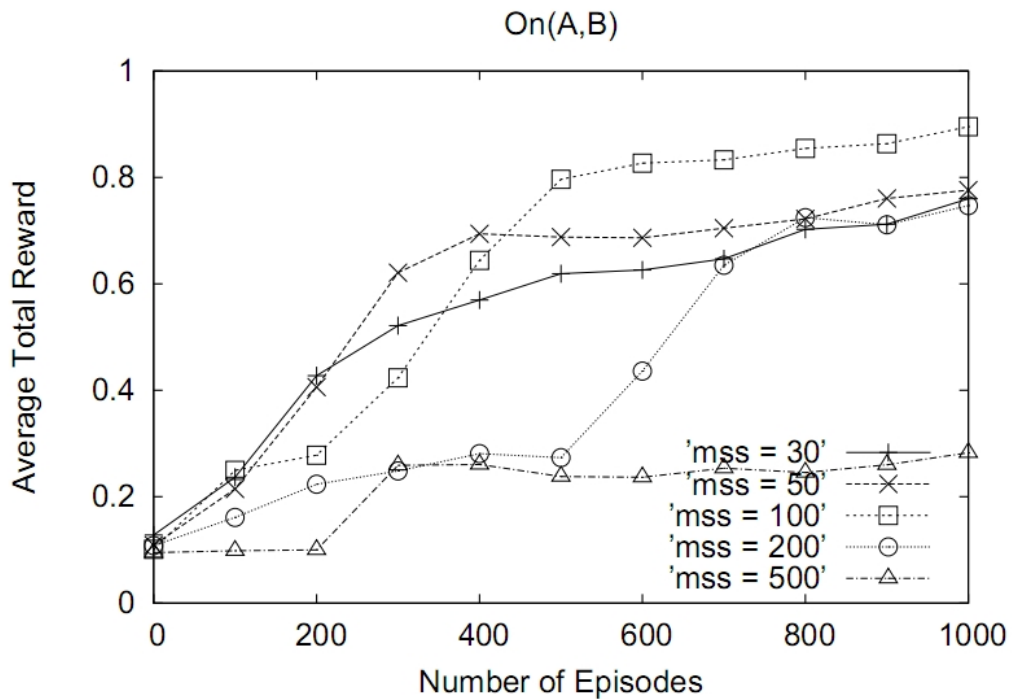
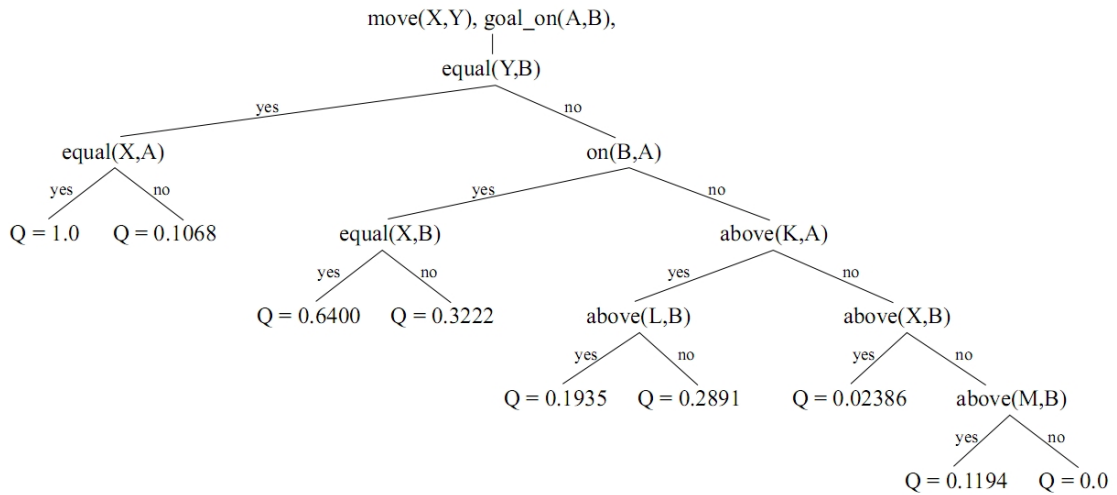


Figure 4.2: Learning curve for the variable state spaces with MSS values of 30, 50, 100, 200 and 500.

Figure 4.3: tree generated for  $MSS = 200$ .

### 4.2.1 Fixed Number of Blocks

The following charts were obtained in the first series of tests. It can be observed that a larger number of blocks in the environment implies a slower convergence of the algorithm, since it increases the amount of possible states and hence the level of difficulty to reach the goal. Changes to the average reward values can be attributed to two factors: the creation of a test in the tree, or the update of the  $Q$  values of the tree that favour the choice of a particular action.

The fact that a test was created in the tree does not always imply an improvement of the learning curve. Depending on the test chosen, the system remains at the same level of the learning curve. This is because sometimes the chosen test is not logically relevant. For example, the first test of figure 4.13 ( $equal(X,A)$ ), when it was created, it was logically irrelevant to the agent. That's because it tests whether the block  $X$  that is being moved (from  $move(X,Y)$ ) is the block  $A$  (from the goal( $on(A,B)$ )). If it is the block  $A$ , then it means that block  $A$  is being moved over any other block, including block  $B$ . If not, the test does not say much about how good is the action of moving other blocks but  $A$ . In short, the test alone is not relevant for the training of the system. Now, in figure 4.5, the first test is  $equal(Y,B)$ . This test checks whether the block  $Y$  (where the block  $X$  is being moved to) is the block  $B$  (from the goal). This test affects the learning curve because it provides a better criterion for the choice of appropriate actions, since moving any block over  $Y = B$  has two possible effects: reaching the goal, or worsening the situation.

Another factor that affects the learning curve of the system is the Q value in the leaves of the tree. As mentioned above, the Q value of each leaf is the average of the Q values associated to the examples that reach that leaf, and it is such Q values of the leaves that dictates the choice of an action in a given state. Q values are updated after each training episode. It might be that, at the time of the creation of two new leaves, the initial Q value attributed to one leaf is higher than the one attributed to its sister. Taking the test  $on(X,A)$  in Figure 4.7 as an example. After checking if the block Y is not the block B (with the first test,  $equal(Y,B)$ ), the agent checks whether the block X that is being moved is the block B ( $equal(X,B)$ ). The Q value of the left branch (0.4458) is lower than the one of the right branch (0.46283).

That makes sense, since it tells the agent that it is a bad idea to move block B. If it moves block B to the top of block A, for example, at least two more actions must be done to reach the goal. Assuming that initially the value of the left branch is greater than the value of the right branch; it causes a stronger preference for moving block B (which is bad) from its place and, therefore, the agent does not learn the full potential of that test (and the learning curve is not affected). But as the Q values of the branches are updated after each training episode, the Q value of the left branch tends to decrease, while the one of the right branch tends to increase (since moving block B can result in a larger number of actions and therefore a smaller average reward). Once the Q value of the right branch exceeds the left branch Q value, the agent learns, correctly, that the action of moving the block B is not good, and shifts to other actions. The oscillation observed in the figure 4.7 is caused by the alternance in the values of the right and left branches of the test  $equal(X,B)$ , where these values were not yet stabilized.

Looking at the tree of Figure 4.5, we can observe that it is logically correct. The first test together with the first test of the left branch of the tree tells the agent that moving the block  $X = A$  over the block  $Y = B$  results in the highest Q value ( $Q = 1$ ) and, therefore, is the best action to be held, while moving any other block than block A to the top of block B is an action that results in a smaller reward, and therefore not as attractive. On the right, the test  $on(X,A)$  together with the test  $equal(Y,B)$  tells the agent that moving block X, which is on block A, to block Y, which is not block B (i.e., clear block A and keep block B clear, both from the target) is also an interesting action to be performed by having the second largest Q value of the tree. On the other hand, if X is not over A, then the action will not reduce or increase the distance between the agent and its goal, but still, it is preferable to any action that moves any block, other than A, over block Y (test  $equal(X,A)$ ).

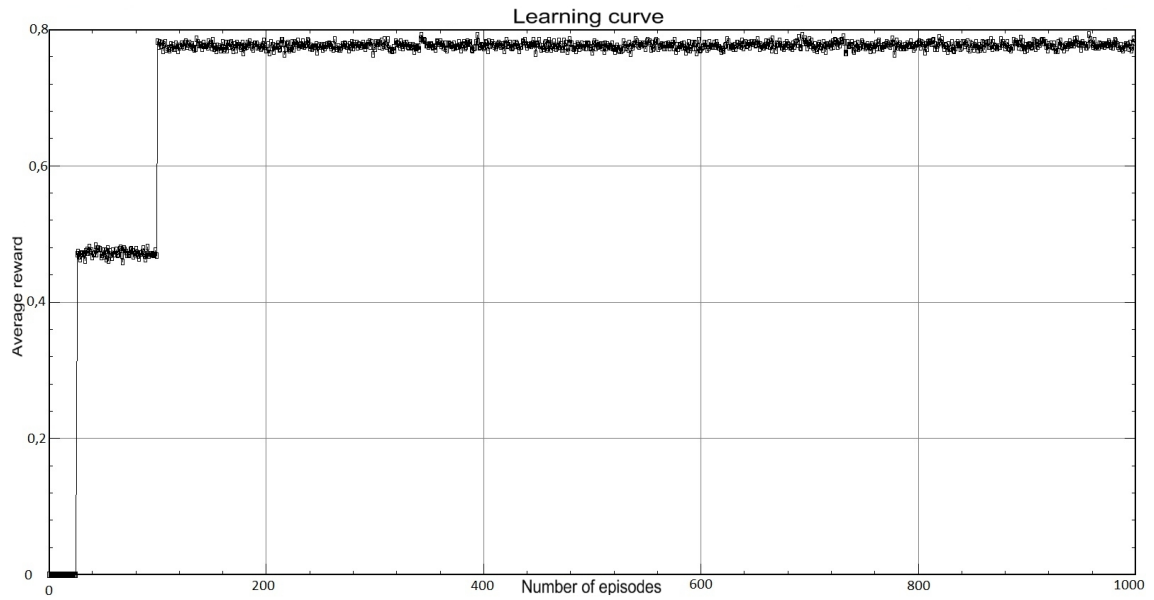


Figure 4.4: Learning curve for the fixed state space of 3 blocks.

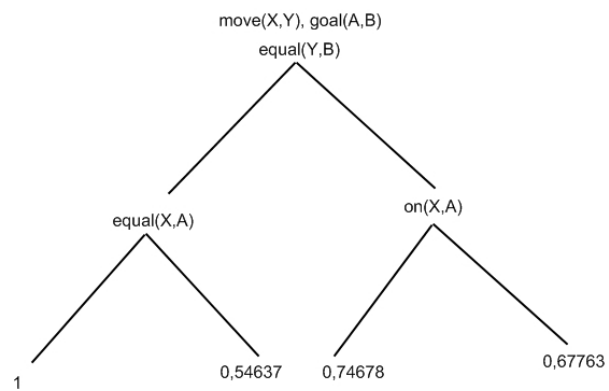


Figure 4.5: Respective tree generated after 1000 training episodes for the fixed state space of 3 blocks.

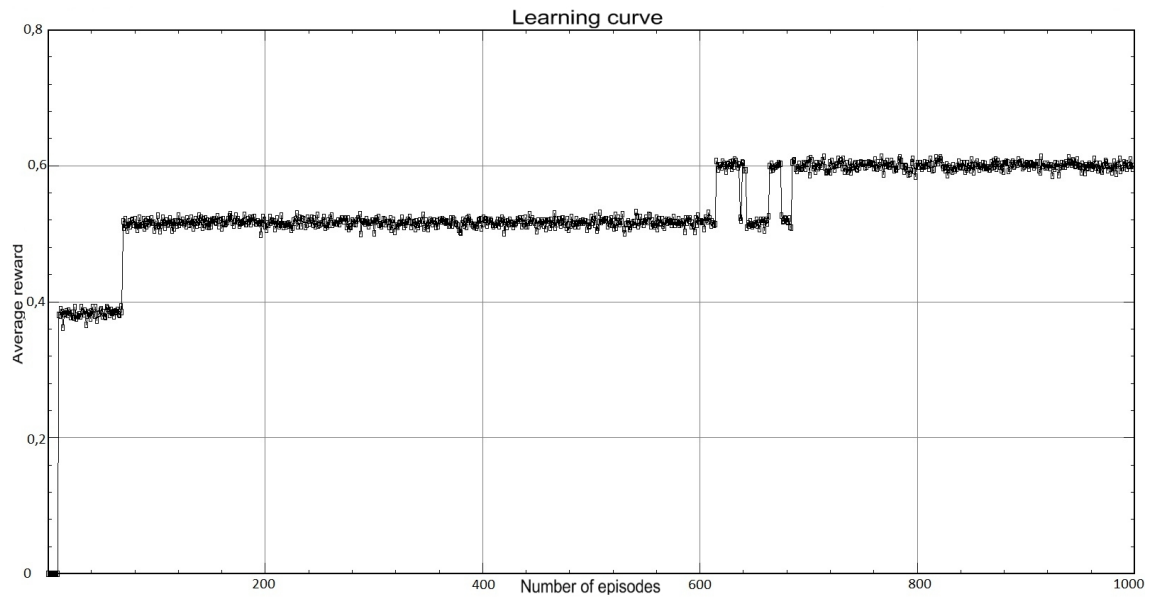


Figure 4.6: Learning curve for the fixed state space of 4 blocks.

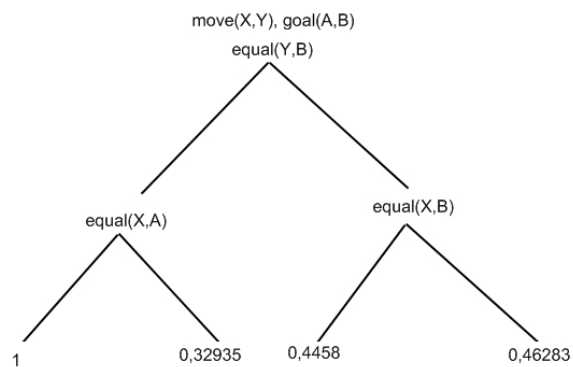


Figure 4.7: Respective tree generated after 1000 training episodes for the fixed state space of 4 blocks.

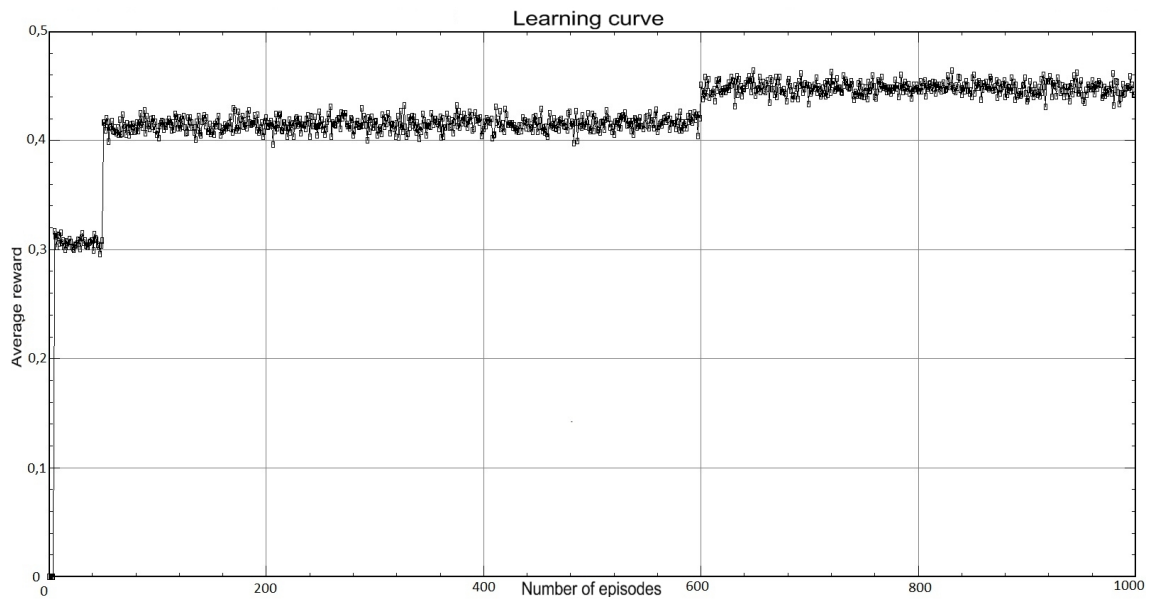


Figure 4.8: Learning curve for the fixed state space of 5 blocks.

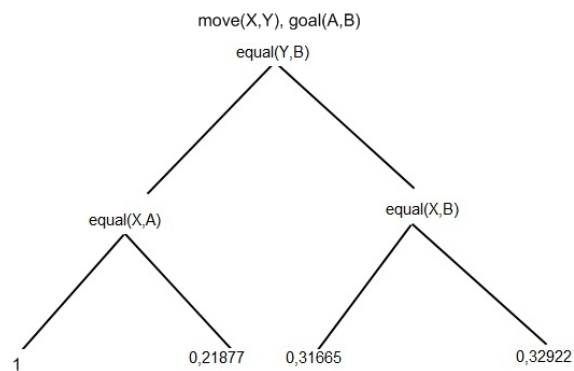


Figure 4.9: Respective tree generated after 1000 training episodes for the state space of 5 blocks.



## 4.2.2 Variable Number of Blocks

Considering that the MSS controls the speed at which the tests are created, low values make it possible for the nodes to be created quicker, and thus, the resulting tree is larger. Higher values of MSS decrease the chance of choosing a bad test in the early nodes, but also slow down the learning curve. It is to be expected that at first, the learning process of the system for low values of MSS will be faster than for high values, but it will decrease as the learning proceeds, whereas for high values of MSS, the learning will be increasing over time until it converges to a certain value.

The charts in Figures 4.10, 4.12, 4.14, 4.16, 4.18, show that for high values of MSS, the learning process of the system is relatively slow compared to the learning process for smaller MSS value. Regarding the generated trees, the one that is inconsistent is the one from Figure 4.19, with the test  $equal(X,B)$ . The Q value of the left branch should be smaller than the one from the right branch, so as to discourage the agent from moving block B, but it is greater. The tendency of the system is to correct that value with more training episodes, then it is assumed that the tree will be corrected in the subsequent steps.

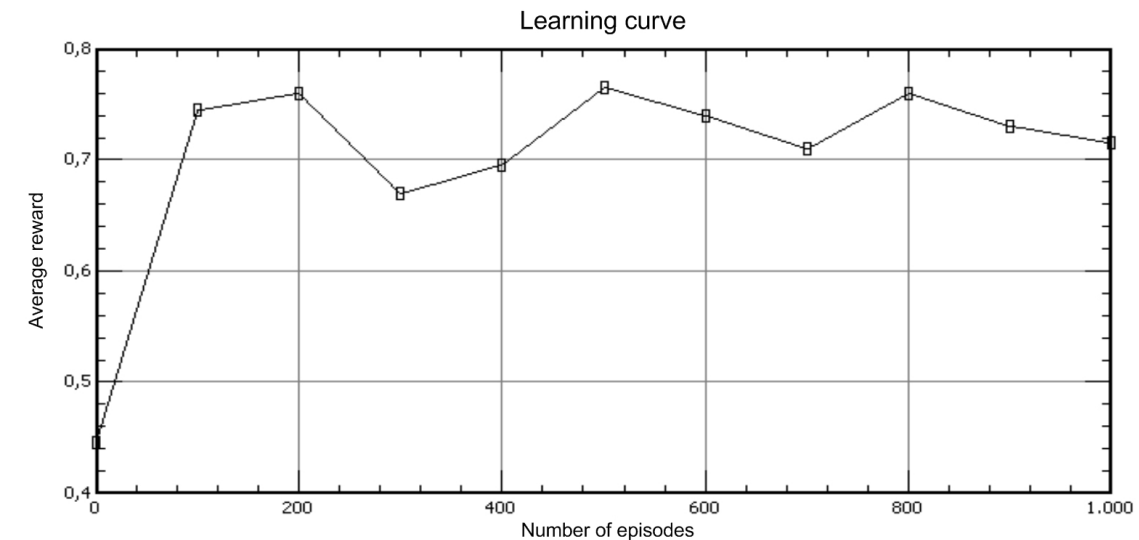


Figure 4.10: Learning curve for the variable state space with MSS = 30.

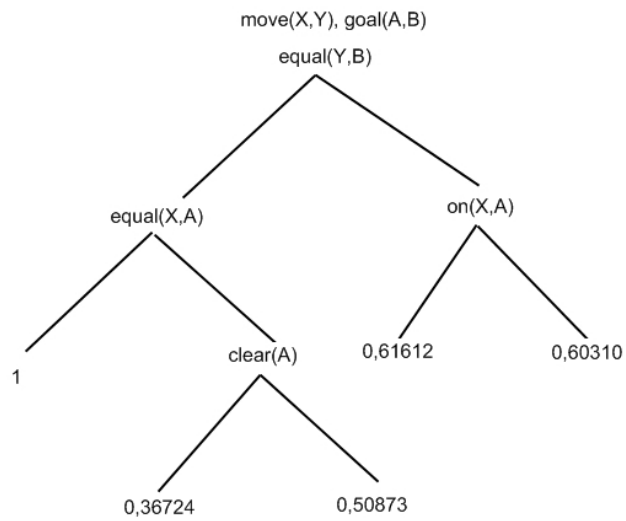


Figure 4.11: Respective tree generated after 1000 training episodes for the variable state space with  $MSS = 30$ .

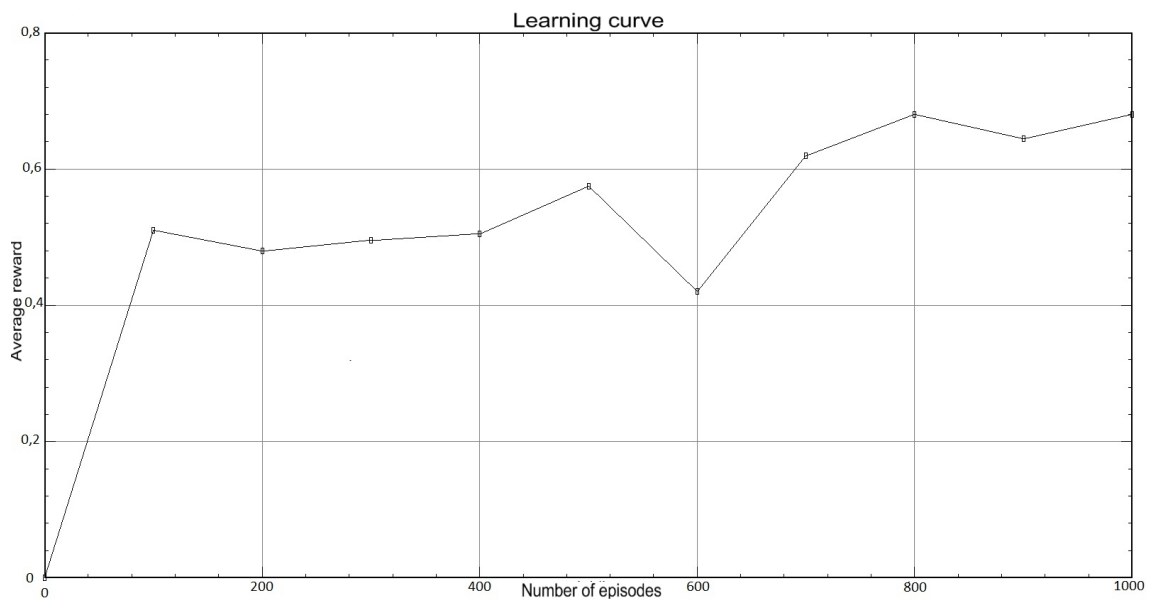


Figure 4.12: Learning curve for the variable state space with  $MSS = 50$ .

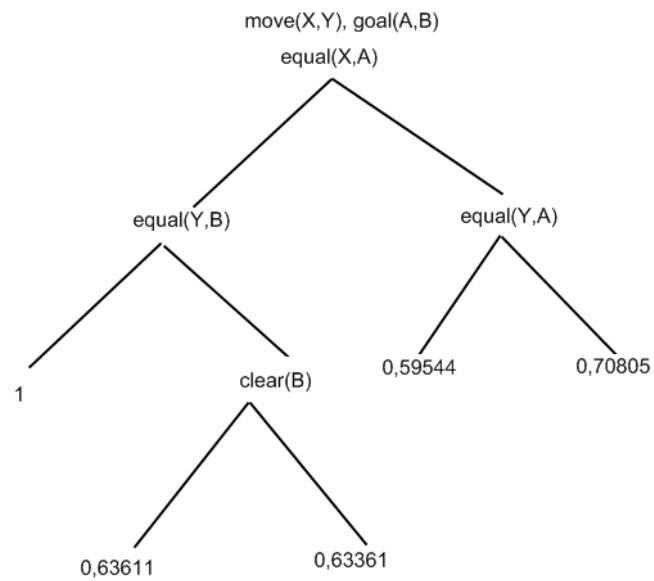


Figure 4.13: Respective tree generated after 1000 training episodes for the variable state space with  $MSS = 50$ .

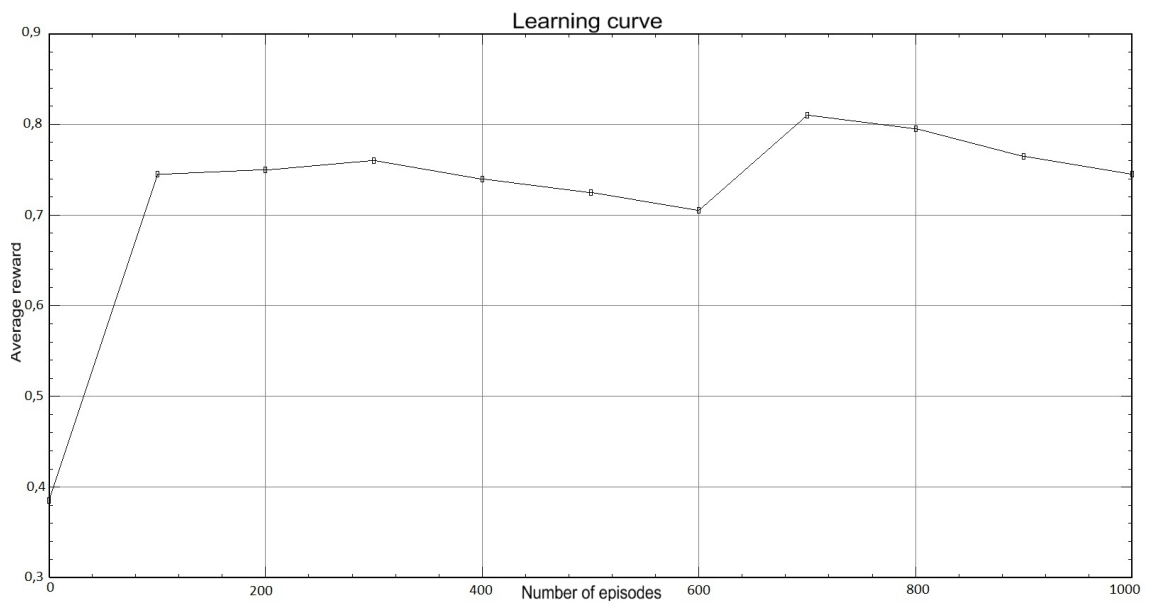


Figure 4.14: Learning curve for the variable state space with  $MSS = 100$ .

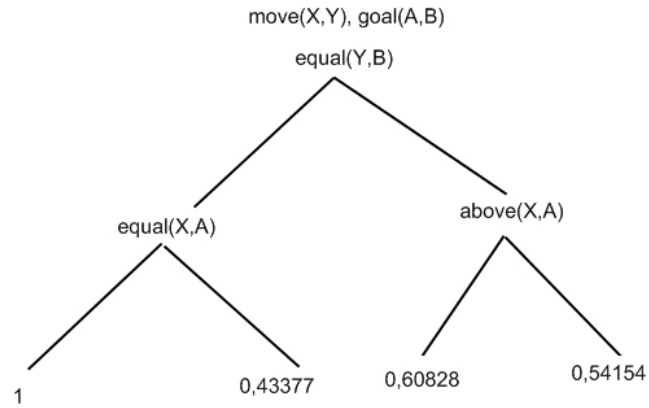


Figure 4.15: Respective tree generated after 1000 training episodes for the variable state space with  $MSS = 100$ .

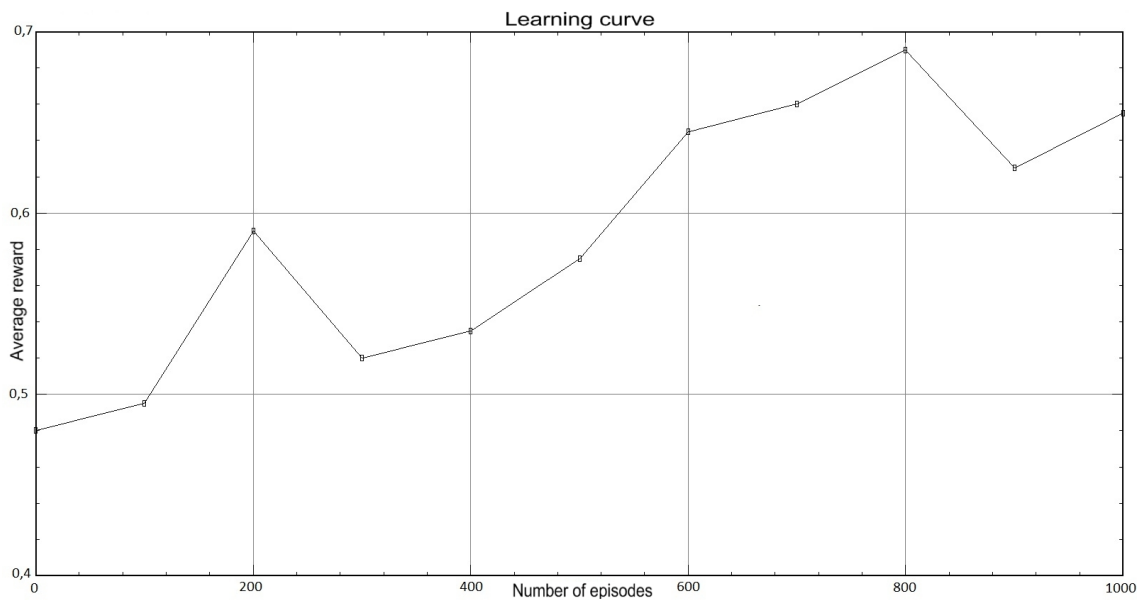


Figure 4.16: Learning curve for the variable state space with  $MSS = 200$ .

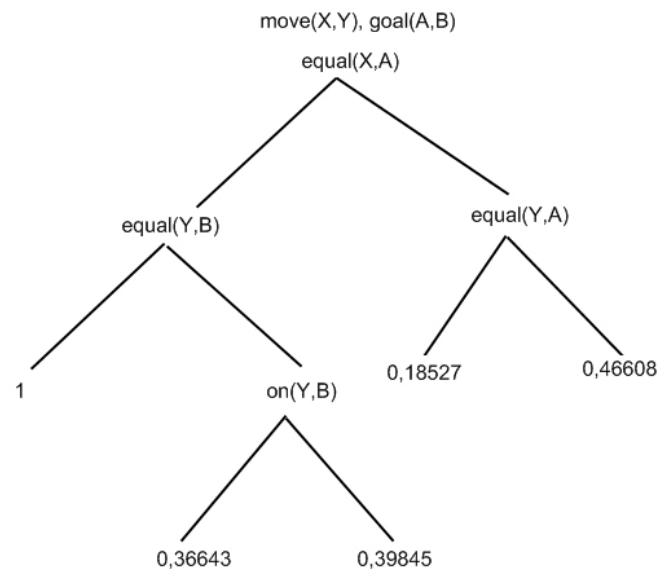


Figure 4.17: Respective tree generated after 1000 training episodes for the variable state space with  $MSS = 200$ .

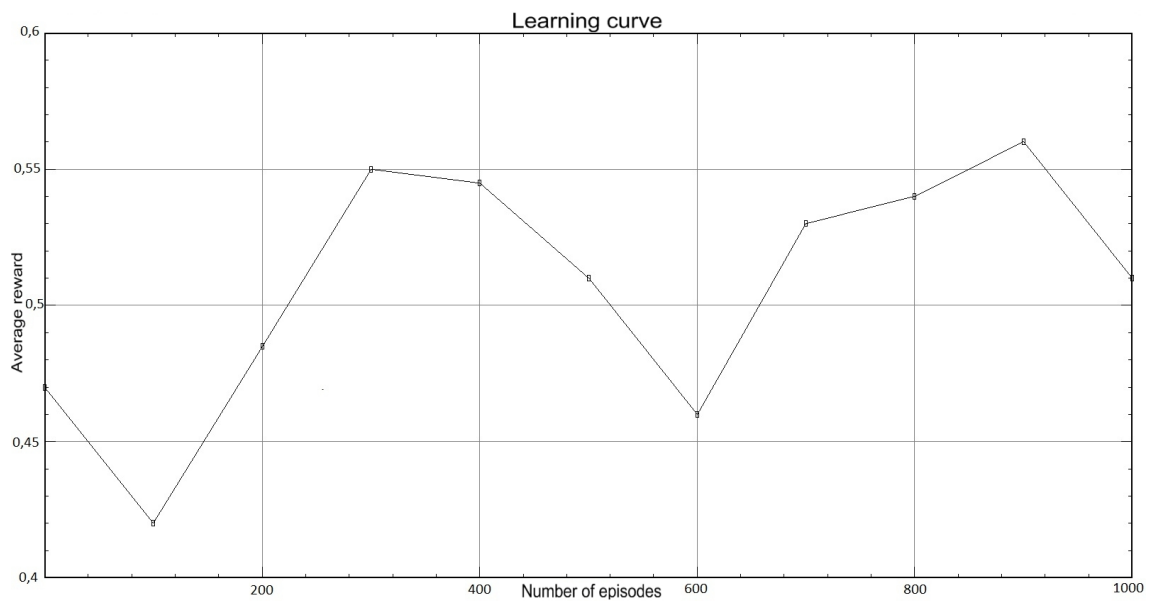


Figure 4.18: Learning curve for the variable state space with  $MSS = 500$ .

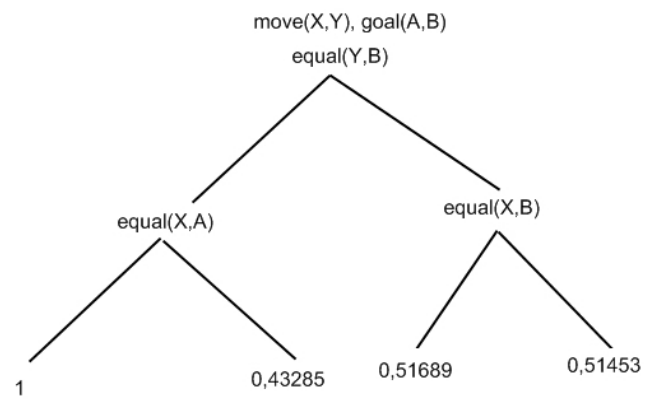


Figure 4.19: Respective tree generated after 1000 training episodes for the variable state space with  $MSS = 500$ .

## 5 *Conclusions and Future Work*

The choice of the tests of the tree is highly influenced by the examples provided by the RRL. Due to the high exploration at the beginning of the algorithm, there is a high probability that the provided examples will lead to the choice of the wrong tests for the tree. Figure 5.1 shows the graph of a disastrous learning example, and the corresponding generated tree is in Figure 5.2. From graph in Figure 5.1 it can be observed that starting from the 400th episode the average reward received decays to zero. This is because the generated tree is inconsistent, by favouring actions that in practice do not minimize the distance between the agent and the goal state (the tests  $equal(Y,B)$  and  $equal(X,A)$  choose actions that interact with the blocks that are neither A nor B). Another sequence of tests that are inconsistent are  $equal(Y,B)$  and  $on(X,B)$ . The first test verifies that Y, destiny location of block X, is the block B of the goal. The second test verifies that the block X, which is being moved, is over B. That sequence of tests basically tests if X is over B **and** is being moved over the block Y, which is the block B, which does not make much sense. On the other hand, verifying that X is not over B is correct, and moving X over the block B is a valid action, but due to the low corresponding Q value, it is not attractive. This ends up making the system prefer to move blocks that are not related to the goal, thus causing the low average reward received.

A problem found in the TG algorithm is its inability to correct errors generated at the beginning of the tree (when bad tests are chosen). The algorithm attempts to correct these errors by constructing a larger tree, which ultimately delays the learning process of the system. The choice of the tests is influenced mainly by the MSS. This parameter can be an effective means of reducing the probability of selecting bad tests at the top of the tree, but it can also introduce a number of undesirable side effects. Higher values of MSS reduce the learning speed of the TG algorithm, and consequently, the learning process of the RRL system, especially in the later stages of the learning process, due to the amount of data collected for each leaf. Lower values speed up the learning process but, as a result, the generated tree is larger due to the attempt of the system to correct early errors by

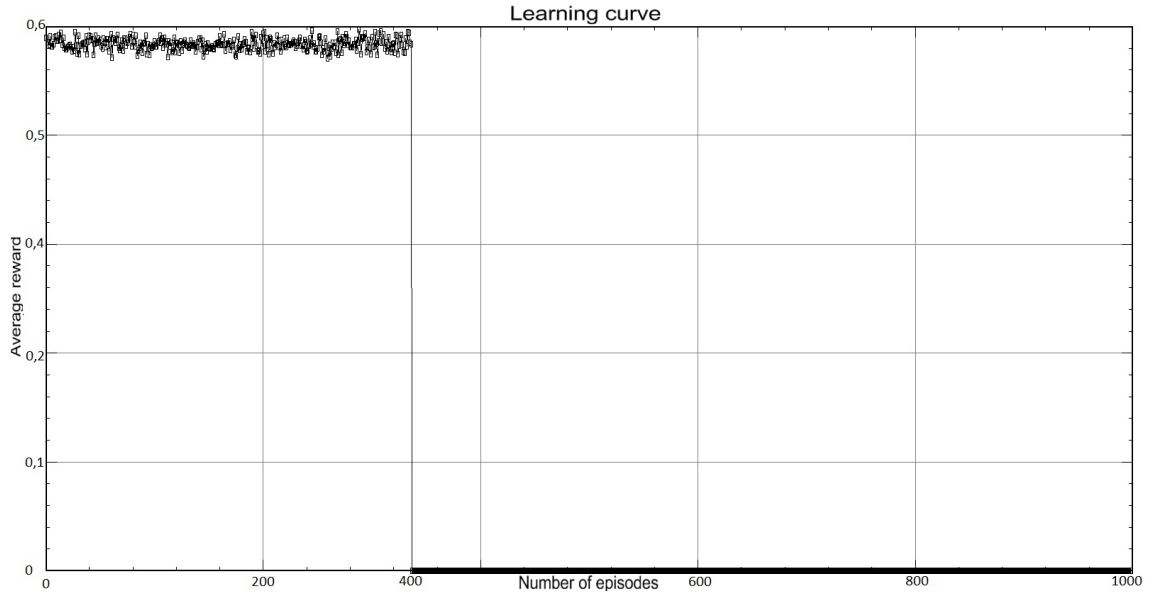


Figure 5.1: Learning curve for the fixed state space of 3 blocks and  $MSS = 200$ .

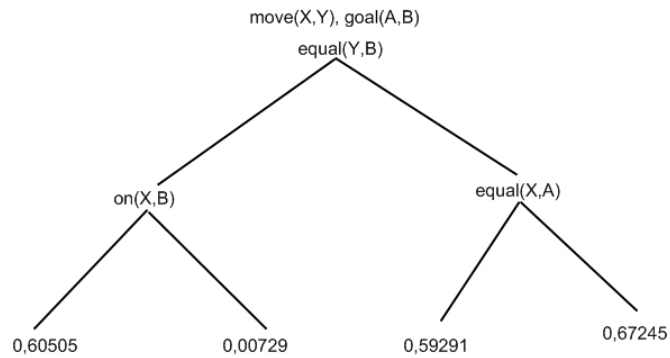


Figure 5.2: Respective generated tree for the fixed state space of 3 blocks and  $MSS = 200$ .



creating more test nodes.

A proposal for future work would be to implement a dynamic MSS, which is reduced at each level of the tree. Another proposal would be to store statistics about the tests in each internal node, not just on the leaves, and allow the TG algorithm to change a test once it is clear that a mistake was made.

# *Bibliography*

- BELLMAN, R. E. *Dynamic Programming*. New Jersey, USA: Princeton University Press, 1957.
- BLOCKEEL, H.; RAEDT, L. D. Top-down induction of logical decision trees. In: *Artificial Intelligence*. [S.l.: s.n.], 1998.
- CHAPMAN, D.; KAELBLING, L. P. Input generalization in delayed reinforcement learning: an algorithm and performance comparisons. *Proceedings of the 17<sup>th</sup> International Joint Conference on Artificial Intelligence*, p. 726–731, 1991.
- DRIESSENS, K. Relational reinforcement learning. In: *Multi-Agent Systems and Applications, 9th ECCAI Advanced Course ACAI 2001 and Agent LinkŠs 3rd European Agent Systems Summer School (EASSS 2001), volume 2086 of Lecture Notes in Computer Science*. [S.l.: s.n.], 2004.
- DRIESSENS, K.; RAMON, J.; BLOCKEEL, H. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In: *Proceedings of the 13th European Conference on Machine Learning*. [S.l.]: Springer-Verlag, 2001. p. 97–108.
- DZEROSKI, S.; RAEDT, L. D.; BLOCKEEL, H. Relational reinforcement learning. In: *Machine Learning*. [S.l.]: Morgan Kaufmann, 1998. p. 7–52.
- LAVRAC, N.; DZEROSKI, S. *Inductive Logic Programming: Techniques and applications*. New York: Ellis Horwood, 1994.
- LAZARIC, A. Knowledge transfer in reinforcement learning. *Tese de Doutorado*, Politecnico di Milano, Milão, p. 18, 2008.
- RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A modern approach*. 2. ed. New Jersey: Prentice Hall, 2003.
- SILVA, R. R. da. Aprendizado por reforço relacional para o controle de robôs sociáveis. *Tese de Mestrado*, ICMC-USP, São Carlos, 2009.
- SNEDECOR, G.; COCHRAN, W. *Statistical Methods*. 8. ed. [S.l.]: Wiley-Blackwell, 1991.
- SUTTON, R.; BARTO, A. *Reinforcement Learning: An introduction*. Cambridge: The MIT Press, 1998.
- WATKINS, C. J. C. H.; DAYAN, P. Q-learning. *Machine Learning*, v. 8, n. 3, p. 279–292, May 1992.

WIKIPÉDIA. *Hefesto* — *Wikipédia*. Online; acessado em 16 nov. 2009. Disponível em: <<http://pt.wikipedia.org/wiki/Hefesto>>.

WIKIPÉDIA. *Timeline* — *Wikipédia*: A brief history of artificial intelligence. Online; acessado em 16 nov. 2009. Disponível em: <<http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/BriefHistory>>.