Applying Relational Reinforcement Learning to Multi-Agent Environments

Group d536a

January 13, 2005

Department of Computer Science Aalborg University

Department of Computer Science

TITLE:
Applying Relational Reinforcement Learning to Multi-Agent Environments

PROJECT PERIOD:
September 3, 2004 – January 13, 2005

PROJECT GROUP:
d536a

GROUP MEMBERS:
Carl Christian Sloth Andersen
Tim Boesen
Thomas Pryds Lauritsen
Dennis Kjærulff Pedersen

SUPERVISOR:
Uffe Kjærulff

ABSTRACT:

This report documents an investigation into the viability of relational reinforcement learning when faced with non-deterministic multi-agent environments. In such environments, both the non-determinism introduced by the roll of the dice, and non-determinism introduced by the presence of other agents exist.
To conduct the investigation, a toolbox for generating transition examples and inducing logical decision trees is described and implemented. The toolbox is used to conduct experiments on both single- and multi-agent environments, thereby illustrating the principle difference of the two. It is shown that learning a reasonable policy in multi-agent environments is difficult when using relational reinforcement learning.
The two main examples used throughout the report are the well known *Blocks World* as well as a simplified version of the board game *Risk* from Hasbro.

Preface

This project was done by group d536a at Aalborg University, Department of Computer Science. We would like to thank Uffe Kjærulff for supervising the entire project.

Aalborg, January 13, 2005

Carl Christian Sloth Andersen

Tim Boesen

Thomas Pryds Lauritsen

Dennis Kjærulff Pedersen

Contents

Chapter 1

Introduction

Machine learning is a fast growing field in computer science. Its influence can be seen in many aspects of our daily lives, from computer games to checking out groceries at the local supermarket.

Within the field of machine learning is Reinforcement Learning (RL), a technique for letting agents learn optimal behaviour in unknown environments by reinforcing their actions with positive and negative rewards. In its most simple form, RL can only be applied to very simple problems due to poor scaling ability. Two research branches are aimed to solve this problem. The first is Hierarchical RL [10], which uses a hierarchical decomposition of larger problems to obtain simple and reusable solutions. The second, and the topic of this report, is Relational RL [15] (RRL).

The basic principle of RRL is to describe the state of an environment using relational properties instead of absolute properties. Often, this makes it possible to introduce state generalisations that reduce the number of essentially different states—thereby creating the opportunity to learn more complex problems. Instead of representing the behaviour function[1] in a table based manner (as is normally the case in traditional reinforcement learning), RRL makes use of first order logic and so-called logical classification trees. This enables a more compact representation.

The scaling of RRL, compared to traditional RL, has previously been tested on the Blocks World problem in [15], however the experiments presented in that paper was mostly conducted to highlight the principal advantages of RRL. They do conclude though, that in the Blocks World scenario, using a rich representation, like first order logic, to represent the behaviour function seems to make RRL scale better than traditional RL.

This report will extend their work, and further test the scaling of RRL. The experiments are conducted using a simple variant of the board game Risk called Simple Risk. This small game introduces the complexity of two types of non-determinism. One type is the non-determinism created by the roll of a set of dice, while the other is the practically unpredictable behaviour of other teams

[1] In Q learning, this is known as the Q function.

(humans or other agents). It is also discussed whether any reasonable behaviour function can be learnt given this scenario and the applied techniques. The experiments are conducted using an RRL toolbox, which has been implemented as part of the work put into this report.

1.1 Outline of the Report

The report is organised as follows: In the rest of this chapter, we review the history of RL and related work. In Chapter 2, we describe traditional RL and the scaling problems associated with the technique. Chapter 3 explains how RRL tries to solve this problem by using Blocks World as the general example. The process of implementing an RRL toolbox and the considerations going into this subject is presented in Chapter 4, and in Chapter 5 the toolbox is used to express the non-deterministic Simple Risk problem. Finally, Chapter 6 discusses the results of the experiments conducted with Simple Risk and the toolbox, while Chapter 7 concludes on the work presented in this report.

1.2 History of Reinforcement Learning

The history of reinforcement learning has two main threads which together have become modern reinforcement learning [29].

One thread concerns the problem of optimal control and its solution using value functions and dynamic programming. The focus on optimal control dates back to the late 1950s, and was used to describe the problem of designing a controller to minimise a measure of a dynamical system's behaviour over time. One of the approaches to this problem was developed in the mid-1950s by Richard Bellman and colleagues [6]. This approach uses the concept of a dynamical system's state and of a value function, to define a functional equation, now often called the Bellman equation.

The other thread is concerned with learning through trial and error and began in the psychology of animal training where "reinforcement" theories are common. The essence is that actions followed by good or bad outcomes have a tendency to be picked more or less frequently accordingly. Thorndike called this the "Law of Effect" [30] because it describes the effect of reinforcing events on the tendency to select actions. Harry Klopf [20, 21] introduced the trial and error to artificial intelligence as reinforcement learning. He recognised that an adaptive behaviour was missing, as researchers had almost only been focusing on supervised learning.

1.3 Related Work

The concept of Q learning has been widely covered by several researchers. Some of them are Christopher J.C.H. Watkins and Peter Dayan in [31].

The problem of scaling using reinforcement learning has been addressed by, among others, Thomas G. Dietterich. In [10], Dietterich explains hierarchical reinforcement learning, which uses hierarchical subdivision to lower the complexity of a given problem. This is done by using subtasks, that combined describe the entire problem. Following, an optimal policy for the entire problem is the collections of policies from the subtask. Some of the subtasks can hopefully be reused by different supertasks, without the need for more learning in the subtask. Each subtask is normally not dependent on all the variables in the problem domain. State abstraction is the technique of abstracting over these irrelevant variables. With the reuse of subtask and by applying state abstraction a reasonable scaling is achieved.

The scalability of relational reinforcement learning has been investigated in [15] by Sašo Džeroski, Luc De Raedt and Kurt Driessens. Their article describes how to use RRL to find solutions to the Blocks World problem with an increasing number of blocks.

Related to our work on creating a toolbox for relational reinforcement learning is ACE [8], which was developed in 2001 by among others Hendrik Blockeel, Luc Deshaspe, Jan Ramon, Luc De Raedt, Wim Van Laer and Jan Struyf. ACE is a data mining system that provides a common interface to a number of relational data mining algorithms including TILDE and TG. The system is based on an underlying Prolog engine to handle the relational input and generate corresponding Prolog programs.

Another toolbox for reinforcement learning is the "Markov Decision Process Toolbox for Matlab" by Kevin Murphy [24]. This toolbox supports value and policy iteration for discrete MDPs.

1.4 Tools Used in this Report

The following tools were used during the creation of this report. Java JDK 1.5 [3]. A prolog engine called SWI-Prolog 5.4.2 [5]. For the inter java-prolog communtiaction, JPL 3.0.3 [4] and InterProlog 2.1 [1] was used. Furthermore C++ using the C compiler of Visual Studio .Net 2003 [2] was used.

Chapter 2

Reinforcement Learning

One of the most commonly used paradigms in machine learning is supervised learning. Supervised learning is a general method for approximating functions, and can e.g. be used to train neural networks. Training data in supervised learning consists of input/output pairs, some of which are supplied for training, while the rest are saved for testing the approximated function.

Supervised learning is very good for solving problems such as classification and problems where the desired behaviour is known. Many problems fit nicely within the supervised training paradigm. For instance, imagine that you would like an agent to learn when the weather is ideal for playing tennis [22]. Then you simply need to organise a set of examples in which playing tennis is a good idea, e.g. `[(sunny, weak wind),(overcast, weak wind)]`, and a set in which it is *not* a good idea, e.g. `[(rainy, weak wind),(rainy, strong wind)]`, and feed them to the supervised learning algorithm.

For some problems it can be very difficult (or even impossible) to define the optimal behaviour of an agent in advance. It is often easier to pick out specific elements from a problem and say: "if the agent is in state s and chooses action a, then it should be rewarded (or penalised)". This is the essence of reinforcement learning, where the agent acts on reinforcing stimuli from the environment in which it exists. Agents can start out with no knowledge of the environment, and learn the optimal strategy for reaching an unknown goal as illustrated in Figure 2.1 [23]. The agent learns in a trial and error manner by exploring the environment and by, to some extent, exploiting what it has already learnt. Reinforcement learning is in some of the literature also referred to as reward/penalty learning, or consequence learning.

This chapter reviews the formalism behind traditional Tabular Reinforcement Learning (TRL), where the behaviour function being learnt is approximated in a table based manner. The first part of the chapter concentrates on deterministic environments whereafter non-determinism is discussed. The environment is traditionally represented as a Markov Decision Process. At the end of the chapter, two different exploration/exploitation policies are presented, and finally the poor scaling ability of TRL is shown.

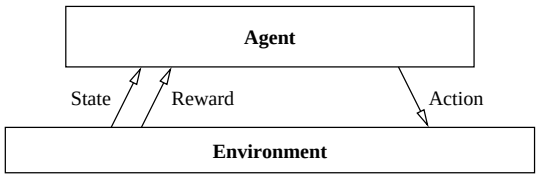


Figure 2.1: Agent interaction with the environment.

2.1 Markov Decision Process

A Markov Decision Process (MDP) [23] is a four-tuple $\langle S, A, r, \delta \rangle$, where S is a set of states, A is a set of actions, $r : S \times A \rightarrow \mathcal{R}$ is a reward function, and $\delta : S \times A \rightarrow S$ is a transition function. The process models the ability of being in a certain state, carrying out an action, and thus ending up in a new state. Having a state and an action as input, the transition function will provide the resulting state, and the reward function will provide an immediate reward. The immediate reward may be high if the concerned transition is of desirable character for the goal of the agent. If the result of the transition is not desirable, the reward may be negative, or there may be no reward at all for that transition. Thus, for a given discrete time step t (the time it takes to complete one transition), the reward r_t and the resulting state s_{t+1} are defined as follows:

$$\begin{aligned} r_t &= r\left(s_t, a_t\right) \\ s_{t+1} &= \delta\left(s_t, a_t\right) \end{aligned}$$

where $s_t \in S$ is a state and $a_t \in A$ is an action, both for time step t.

An MDP is a model of the environment in which an agent exists. An agent does not necessarily have full access to its environment—for instance the reward and/or the resulting state of an action in a certain state may be unknown until the agent tries the action in question. The fact that the next state depends only on the current state and the chosen action, is an important feature of the model which is called the Markov property.

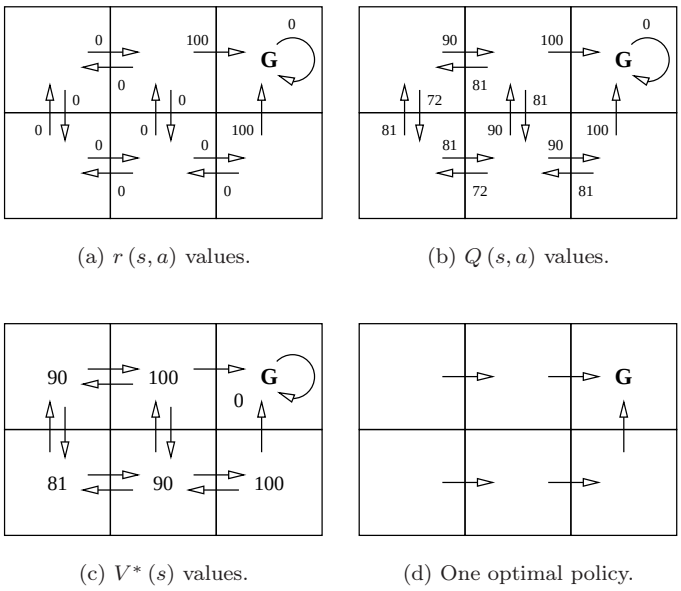
A problem for the agent is the fact that most actions in most states will probably give no immediate reward. For instance, an agent in a labyrinth may stray around unrewarded until it finally finds the exit and is rewarded. This is called *delayed rewards*. In order to be able to find an optimal policy it must thus itself assign values to the actions presented to it in a given state, and then select the most lucrative action in any situation. By defining a policy $\pi\left(s_t\right)$ as a function $\pi : S \rightarrow A$ which gives an action from an observed state s_t, one can define the *discounted cumulative value* over time $V^{\pi}\left(s_t\right)$ for a given policy such that:

$$\begin{aligned} V^{\pi}\left(s_t\right) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \\ &\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned} \tag{2.1}$$

The discounted cumulative value[1] introduces the constant $0 \leq \gamma < 1$ which discounts the relative value of future rewards more than that of immediate rewards. Thus it will be more attractive for the agent in the labyrinth to take the shortest route to the exit rather than to detour.

An optimal policy π^* must be one that for every step selects an action that gives the highest cumulative value for the state in question. Notice, that often there will be several optimal policies since two actions for a given state may yield the same highest cumulative value. For an example of r values and an optimal policy, refer to Figure 2.2. An optimal policy is defined as:

$$\pi^* \equiv \arg\max_{\pi} V^{\pi}(s), (\forall s)$$



(a) $r(s,a)$ values. (b) $Q(s,a)$ values.

(c) $V^*(s)$ values. (d) One optimal policy.

Figure 2.2: A simple deterministic world to illustrate the basic concepts of Q learning. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, $r(s,a)$ gives reward 100 for actions entering the goal state **G**, and zero otherwise. Values of $V^*(s)$ and $Q(s,a)$ follow from $r(s,a)$, and the discount factor $\gamma = 0.9$. An optimal policy, corresponding to actions with maximal Q values, is also shown. This example is from [23].

2.2 Q Learning

To make a qualified choice between actions in a given state we need to learn the function $\pi^* : S \rightarrow A$. It would be obvious to choose the action (or one of the

[1] Additionally there are the *finite horizon reward* and the *average reward*. These are explained in [23].

actions) that has the highest cumulative value for a given state, i.e. V^{π^*}, or in short notation V^* (for an example of V^* values, see Figure 2.2).

The optimal action $\pi^*(s)$ for a given state s can be defined as

$$\pi^*(s) = \arg\max_a \left[r\left(s,a\right) + \gamma V^*\left(\delta\left(s,a\right)\right)\right] \tag{2.2}$$

I.e. the action that maximises the immediate reward plus the discounted maximal cumulative value of the state resulting from the δ transition function.

Unfortunately, this requires access to the reward function r and the transition function δ, which are often inaccessible to the agent. An example is the aforementioned agent in the labyrinth which initially does not know anything about the layout of its mazy environment. Therefore the agent will have to make use of trial and error and remember how things went so that it can use the experience in the next pass.

One way to do this, is to use the *Q learning algorithm*. It works by iteratively approximating the sum of future rewards from a given state. The more iterations performed, the more precise the approximation will become—provided that the environment does not change. In TRL, the approximated values are stored in a table which links states and actions together in pairs. For an example of such a table, refer to Figure 2.1. The value of the function $Q\left(s,a\right)$ is defined as the maximal discounted cumulative reward obtainable from a given state s by performing a given action a, as follows:

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) \tag{2.3}$$

Using $V^*(s) = \max_{a'} Q(s,a')$ a recursive version of Equation 2.3 can be defined as

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a),a') \tag{2.4}$$

Notice that the righthand side of Equation 2.3 is a part of Equation 2.2. Therefore it is possible to redefine π^* as depicted in Equation 2.5 using Equation 2.3 and thereby become independent of the r and δ functions:

$$\pi^*\left(s\right) = \arg\max_a Q\left(s,a\right) \tag{2.5}$$

We use the symbol $\hat{Q}$ to describe the agent's current approximation of Q. Initially all $\hat{Q}$ values are set to random values[2] or simply zero whereafter the current state s is observed. Subsequently, the following is repeated infinitely: An action a is chosen and performed, the resulting immediate reward r and state s' are observed, and $\hat{Q}\left(s,a\right)$ is updated (for s, the previous state) using the current $\hat{Q}$ value for the new state. The algorithm is described in further detail in Table 2.2.

The Q learning algorithm, except the initialisation of $\hat{Q}$, represents one *episode* of learning. For each episode, the agent's initial state is randomly chosen. An

[2] [15] explains why initialising to random values causes faster converging to Q.

State / Action	a_{up}	a_{down}	a_{right}	a_{left}
s_1	0	72	90	0
s_2	0	81	100	81
s_3	0	0	0	0
s_4	81	0	81	0
s_5	90	0	90	72
s_6	100	0	0	81

Table 2.1: An example of a Q table showing the tabular representation of the Q values of Figure 2.2(b). The states have been given the names s_1 through s_6 consecutively from the upper left state to the lower right state.

$\forall s, a$ **repeat**
  $\hat{Q}(s,a) \leftarrow 0$ `(or a random value)`
Repeat `forever`
  $s \leftarrow$ `an initial state`
  Repeat `until` s `is a goal state`
    $a \leftarrow$ `a chosen action`
    `Perform` a
    $r \leftarrow r(s,a)$
    $s' \leftarrow \delta(s,a)$
    $\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$
    $s \leftarrow s'$

Table 2.2: The Q learning algorithm for deterministic actions and rewards.

episode can end, for instance, when the agent enters an *absorbing state*—a state where, no matter which action the agent performs, it will stay in that state. In the example in Figure 2.2, the state **G** is absorbing. $\hat{Q}$ values will propagate out backwards from the goal state. This is a consequence of the fact that the $\hat{Q}$ update of the current state/action pair is calculated from the $\hat{Q}$ value of the following state/action pair. The proof that the $\hat{Q}$ values converge to Q as the number of updates approaches infinity is presented in [23], pages 378-379.

2.3 Non-determinism

An environment like the labyrinth used in Section 2.1 is deterministic. If we are instead dealing with a non-deterministic environment, where the transition function $\delta(s, a)$ and the reward function $r(s, a)$ may not always yield the same results given the same input, we will need to extend the Q learning algorithm. Such an environment is the Simple Risk game which will be explained later in this report, but also other games have transition and reward functions with probabilistic outcomes. In order to allow non-determinism, the following changes to the deterministic definitions are made.

V^π is redefined to be the expected value (E) over its non-deterministic outcomes of the discounted cumulative reward received by applying policy π. Hence Equation 2.1 becomes:

$$V^\pi(s_t) \equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

Equation 2.3 is rewritten to express the expected Q value as:

$$\begin{aligned} Q(s,a) &\equiv E\left[r(s,a) + \gamma V^*(\delta(s,a))\right] \\ &= E\left[r(s,a)\right] + \gamma E\left[V^*(\delta(s,a))\right] \\ &= E\left[r(s,a)\right] + \gamma \sum_{s'} P(s'|s,a) V^*(s') \end{aligned}$$

where $P(s'|s, a)$ is the probability that action a in state s will result in state s'. The recursive definition of Q for non-determinism (analogous to Equation 2.4) is:

$$Q(s,a) = E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a') \tag{2.6}$$

If e.g. the reward function returns a non-deterministic result, our stored $\hat{Q}$ value for a given s and a would change every time our deterministic training rule is applied, even if $\hat{Q}$ is already (close to) Q. In other words; it would not converge. Therefore, the line

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

in Table 2.2 is rewritten into

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n \left[r + \max_{a'} \hat{Q}_{n-1}(s',a') \right]$$

where α_n is the learning factor for the nth iteration of the algorithm. This is an estimate of $\hat{Q}_n(s,a)$ in the nth iteration of the algorithm. The learning factor can be defined as

$$\alpha_n = \frac{1}{1 + visits_n(s,a)}$$

where s and a are the state and action updated during the nth iteration, and where $visits_n(s,a)$ is the total number of times this state/action pair has been visited up to and including the nth iteration. Using the learning factor has the effect of putting more emphasis on the previous approximation of $\hat{Q}$, and less on new observations. As α_n approaches zero, the $\hat{Q}$ approximation stabilises.

The fact that $\hat{Q}$ converges to Q as n approaches infinity in a non-deterministic environment is proven in [31].

2.4 Exploitation and Exploration

While executing the Q learning algorithm from Table 2.2, an agent must repeatedly choose the next action to execute. This action can be chosen randomly, but that will most likely lead to slow learning, since an agent might search non-profitable paths again and again. That is, the agent will keep exploring the environment and not exploit what is already learnt. On the other hand, if an agent continuously chooses the action with the highest payoff, it risks converging to a sub-optimal policy—simply because there might exist better undiscovered paths (with a higher payoff) that are never explored.

In [15] the authors solve this tradeoff by calculating a prior probability distribution over the possible actions from the current estimation of the $\hat{Q}$ function. Actions with a high payoff get a higher probability of being chosen than actions with a low payoff. However, all actions get a probability greater than zero. The probability distribution is defined as follows:

$$P(a_i|s) = \frac{k^{-\hat{Q}(s,a_i)}}{\sum_j k^{-\hat{Q}(s,a_j)}} \tag{2.7}$$

where $P(a_i|s)$ is the probability of selecting action a_i given state s. The constant $k > 0$ determines how strongly the selection favours actions with high $\hat{Q}$ values. Lower values of k encourage exploiting while higher values increase the probability of exploring. Before calculating the probability distribution, the range of Q values should be normalised to a number between zero and one. High Q values might otherwise result in very large numbers. Figure 2.3 shows the effect of setting the constant to different values.

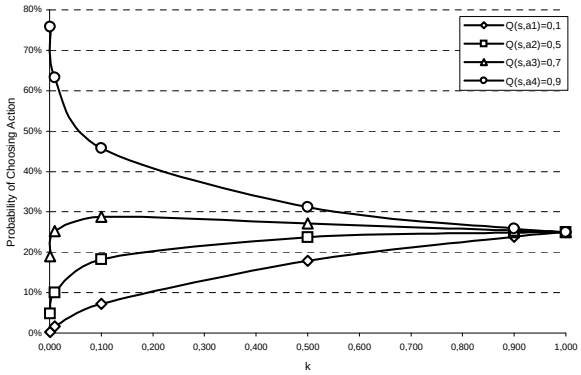


Figure 2.3: The effects of constants.

Another technique is combining a greedy strategy with an *optimism in the face of uncertainty* heuristic [19]. A greedy strategy will always choose the action with the highest payoff. To avoid converging to a sub-optimal policy, the agent will have strong prior beliefs on the payoffs of untested actions. Only strong negative evidence will remove an action from consideration.

For more information on exploitation and exploration, refer to the survey in [19], which has a list of formally justified techniques as well as a list of more ad-hoc techniques (including the greedy strategy).

2.5 Scaling with Tabular Q Learning

The Navigation problem from Figure 2.2 is a toy example. It only has six different states and four different actions. The table needed to represent the $\hat{Q}$ function for that problem therefore only needs to contain 24 values. The number of values n needed is calculated as

$$n = actions \cdot \left(gridsize^{agents} \right)$$

where *actions* is the number of actions available for the learning agent, *gridsize* is the number of squares in the grid, and *agents* is the number of existing agents. If we expand the grid to 100 squares and add one more agent, which also has a location in the grid, then the $\hat{Q}$ table needs to contain $40,000$ Q values. For real life problems the number of state/action pairs will quickly explode even further, thereby making the problems impossible to solve with tabular Q learning. Table 2.3 shows the table sizes needed to represent the $\hat{Q}$ function as the number of states increases. Obviously it is impossible to work with this kind of increasing space requirement.

The research effort to make Q learning usable on larger problems has mainly focused on exploiting specific problem properties. To use the above example, instead of worrying about the exact position of multiple agents, it might be

Grid Size	Actions	Agents	$\hat{Q}$ Table Size
6	4	1	24
100	4	2	$40,000$
200	4	3	$32 \cdot 10^6$
500	4	5	$125 \cdot 10^{12}$

Table 2.3: Scaling of tabular Q learning in the Navigation problem.

enough to know a generalised direction vector between them. We are then exploiting the relations that exist between the agents. This enables the possibility of using information in one setting that was learnt in another setting, where the same relations exist. Using relations with reinforcement learning is the topic of Chapter 3.

Similarly some problems have a hierarchical structure by nature. Imagine that the Navigation problem was a smaller part of driving a taxi. First the agent has to navigate to a customer, and then it has to navigate to the drop off destination. The task of navigating does not depend on whether we have a customer in the taxi or not—it only depends on the destination. The task of navigating can therefore be learnt once, and then used from each parent task. This is known as hierarchical reinforcement learning. For more information on this subject, refer to [10].

2.6 Summary

This chapter has reviewed the basic concepts of reinforcement learning. It has been shown how agents can learn optimal policies in unknown environments by reinforcing their actions with positive or negative rewards. The Q function was defined to, basically, encapsulate the unknown transition and reward function. Training an agent becomes a matter of storing and updating Q values for each state/action pair in the environment.

The convergence of Q learning has earlier been proven for both deterministic and non-deterministic environments. Unfortunately, the scalability of traditional Q learning is very poor. This is due to the combinatorial explosion of state/action pairs, that occurs when new information is added to a state description.

The next chapter describes Relational Reinforcement Learning. This is a technique that utilises relations between objects in an environment in order to reduce the state space—thereby making it possible to solve more complex problems than with traditional reinforcement learning.

Chapter 3

Relational Reinforcement Learning

A solution aimed at easing the combinatorial state/action pair explosion in reinforcement learning, is allowing generalisations over states and objects present in a given problem domain. One such technique is called Relational Reinforcement Learning (RRL), and it combines traditional reinforcement learning with inductive logic programming.

In the following sections, the application of inductive logic programming in RRL is described using a family tree scenario. In the rest of the chapter, the toy domain of *Blocks World* is used to illustrate the ideas and practical use of RRL.

3.1 First Order Inductive Logic Programming

In the rest of this report we will make extensive use of first order logic. For this reason, this section will give a very short introduction to first order logic and inductive logic engines such as Prolog.

Although most Prolog implementations have a slightly different syntax, the basic building blocks are normally the same. These are

- **Constants**: `'block', block, 7, a, ...`
- **Variables**: `Block, A, X, ...` (capitalised)
- **Functions**: `f, cos, +, ...`
- **Logical Symbols**: $\vee$ `(or)`, $\wedge$ `(and)`, $\neg$ `(negation)`, `...`
- **Predicates**: $=$, $<$, `mother, father, ...`

Because the logical symbols shown in the list above are difficult to include in a programming language, Prolog implementations most often use , (comma) as $\wedge$

and ; (semi-colon) as $\vee$. For the same reason, negation is sometimes represented \ (backslash) or as the predicate `not` (or both).

A simple Prolog *knowledge base* consists of a list of *facts*. For instance, the following example describes the facts known about an imaginary person `john`.

```
male(john).
parent(john,mona).
parent(john,leonardo).
female(mona).
male(leonardo).
```

The facts describe that `john` is a male, and that he has two parents, `mona` and `leonardo`. We are also told that `mona` is female and that `leonardo` is male. Using Prolog we can query on existing facts. For instance, the query `male(john)` will return `yes`, while the query `female(john)` will return `no`. Each fact ends with a period telling Prolog that a new fact or *rule* will start.

Rules are predicates which are useful when the combination of existing facts contain "hidden" information. Given the stated facts, we, as humans, can easily infer that `mona` must be the mother of `john` and that `leonardo` must be the father. Prolog does not have this ability, so instead we create the rules:

```
mother(X,Mother) :- parent(X,Mother), female(Mother).
father(X,Father) :- parent(X,Father), male(Father).
```

The rules state that the mother of `X` is both the parent of `X` and a female, and vice versa for the father of `X`. Notice that `X`, `Mother` and `Father` are variables. We can now create the query `mother(john,mona)` which will return `yes`.

Imagine now that we do not know the mother of `john`, but would like to ask Prolog. We construct and execute the query `mother(john,Mother)`. Prolog must now infer whether or not `john` has a mother and, if so, instantiate `Mother` to her name. Such a query is solved using syntactic inference as illustrated below:

```
parent(john,Mother), female(Mother)   →
parent(john,mona), female(Mother)     →
parent(john,mona), female(mona)       →   yes
```

In this case, the `mother` predicate consists of two other predicates, `parent` and `female`. To begin with, Prolog searches for an *instantiation* of the *uninstantiated* variable `Mother` that will make `parent(john,Mother)` true. It finds `Mother=mona`. It then uses this instantiation on the next predicate yielding `female(mona)`, which is also true. Thus, it is inferred that `john` does indeed have a mother and her name is `mona`. Prolog returns `Mother=mona` and `yes`.

Sometimes a query has more than one solution. For instance, the query

```
parent(john,Parent)
```

first returns `yes` with the solution `Parent=mona`. Prolog can then be instructed to search for other solutions by backtracking over the query yielding `Parent= leonardo`. All solutions to a query can be returned as a list by using the built-in predicate

```
findall(+Template, +Goal, -Bag)
```

which creates a list of the instantiations that `Template` gets successively on backtracking over `Goal`. The list is returned in `Bag`. In this context, the operator `+` means that a variable must be instantiated in a query, and `-` means that it must *not* be instantiated. The query

```
findall(Parent, parent(john,Parent), Bag).
```

returns the solution `Bag=[mona, leonardo]`. Prolog lists are represented using square brackets `[]`, and can contain all sorts of elements. For instance, we can represent a number of facts in a list:

```
[male(john), parent(john,mona), parent(john,leonardo),
female(mona), male(leonardo)]
```

A list of facts holding at a specific point in time can be viewed as a state. A state should only contain facts that cannot be automatically inferred using other facts. A predicate like `mother(X,Mother)` should therefore never be inserted into a state, but can instead be used to generalise over sets of states as described later in this section.

In the context of reinforcement learning, the interesting application of inductive logic is its ability to represent state generalisations. Remember that a state is a list of non-inferable facts. The primary idea is, that if a predicate p succeeds when applied to the states s_1 and s_2, but fails when applied to state s_3, then we can group the states together accordingly. E.g. applying the predicate `male(X)` on all states, we effectively create two sub-groups, namely a group of states with at least one `male` fact, and a group of all other states. More predicates can be applied to create even more sub-groups.

For each sub-group we can assign a quality value of performing a specific action (e.g. a Q value). This allows us to create a policy specifying the best action for a group of states—instead of specifying the same for each single state as in traditional reinforcement learning.

Throughout the report, we make use of the built-in Prolog predicate `member`. This predicate performs a member check on a specified list, e.g.

```
member(a,[a,b,c])
```

returns `yes`, while `member(d,[a,b,c])` returns `no`. We also use the anonymous parameter represented with an underscore `_`, to create queries where a parameter can be anything (like a wildcard).

This section has described the basics of inductive logic needed to read the rest of this report. The next section will clarify the benefits of applying inductive logic to traditional reinforcement learning.

3.2 Relational Problems

Many problem domains exhibit relational properties. The family domain used in the previous section is one example, and the Blocks World domain presented later in this chapter is another.

The fact that a domain has relational properties, means that it has entities that can be described by the same relations. For instance, imagine two families, f_1 and f_2, both with two parents and a child. The only difference between such two families is the name of respective family members (at least using our state descriptions), and as mentioned in the previous section, those can be generalised away. using family f_1, we can therefore learn which actions are best to perform at any given state. The result of this is a policy, which can be reused on family f_2.

Combined, inductive logic and the use of relations results in a significant state reduction for relational problem domains. There is one more important benefit though. Since the goal state of an agent can also be generalised, it becomes possible to change the goal to some extent without being forced to learn a new policy. This also makes it possible to use bootstrapping of policies as explained in Section 3.9. Neither of these benefits seem to be possible in traditional reinforcement learning.

While the use of relations is natural in many problem domains, some domains does not exhibit any relational properties at all. For instance, it would be difficult to find usable relations for describing a large labyrinth.

3.3 The RRL Environment

The environment in RRL can be represented as an extended MDP. The extended MDP is defined as a nine-tuple:

$$\langle S, F, A, r, \delta, s_{goal}, pre, BK, DB \rangle$$

where

- S is the set of states represented in a relational language,
- F is the set of non-inferable facts usable in any state $s \in S$,
- A is the set of actions represented in a relational language,
- $r : S \times A \rightarrow \mathcal{R}$ is the reward function, which is *unknown* to the agent,
- $\delta : S \times A \rightarrow S$ is the transition function, which is *unknown* to the agent,
- $s_{goal} \in S$ is a goal state generalisation represented in a relational language.
- $pre : S \times A \rightarrow \{tt, ff\}$ is the action precondition function,
- BK is background knowledge generally valid in S represented as rules in a relational language, and
- DB is a declarative bias which restricts the language in which a policy is represented.

Each element in the tuple is described further in the following. A state $s \in S$ is a list of facts that hold true in that state (without generalisations). An example of a possible state is

$s =$ `[male(john), parent(john,mona), parent(john,leonardo)`
`female(mona), male(leonardo)]`

which describes a small family. A state is constructed by combining elements from the set of non-inferable facts F.

The goal state generalisation s_{goal} denotes the relations that must exist, if the agent is in a goal state. A goal state generalisation, which specifies that the goal of a family, is to have at least one child, can be expressed like `Parent(X, Parent)`.

In each state s, a subset of actions $A_s \subset A$ is available for the agent to choose from. The subset is determined by the action precondition function *pre*, which is defined as:

if *action a is allowed in state s* **then** $pre(s, a) = tt$ **else** $pre(s, a) = ff$

The background knowledge BK consists of rules that are generally valid in the set of states S, and that can be inferred by the facts in a state. For instance, the `mother` and `father` predicates described in Section 3.1 are considered background knowledge.

The non-inferable facts F and background knowledge specifies how agent policies are represented, but a so-called declarative bias is needed to apply type and mode restrictions. Consider again the rule `mother(X,Mother)`. We expect both `X` and `Mother` to be of the type *person* and not e.g. *blocks*. The latter would not make any sense and should not be allowed. To understand mode restrictions, consider again constants and variables, where variables can be either instantiated or uninstantiated. We might not allow constants in our policy representation, which means that a predicate like `mother(john,mona)` is not allowed. Similarly, we can choose not to allow the predicate `mother(X,Mother)` if not both `X` and `Mother` have previously been instantiated by another predicate.

The task of RRL is the same as for traditional reinforcement learning: to find a policy for selecting actions $\pi : S \rightarrow A$ that maximise the expected discounted cumulative reward. A policy can be represented as either the standard Q function, or as a policy function $P : S \times A \rightarrow \{tt, ff\}$. The P function encodes the optimality of choosing action a in state s, and it has been shown that convergence to an optimal policy is faster obtained using this function [15].

Both functions can be approximated as *logical decision trees*, which make use of non-inferable facts and background knowledge as node *tests*. The approximation of the two functions is the topic of Section 3.7.1.

In the next section, the relational toy domain Blocks World is introduced.

3.4 Blocks World

The Blocks World problem is represented by a simple scenario, where each of a number of blocks can either sit on the floor or on top of another block. Each block is labelled with a letter, and a letter can only be the label of one block. Otherwise the blocks have the same properties.

The goal of a Blocks World problem is to end up in a certain state, i.e. a certain configuration of (some of) the blocks, for instance to have block `b` situated on top of block `a` or to have all blocks on the floor. To be able to end up in the goal state, a move action is available. This can be used to move the blocks, one at a time, onto another block or onto the floor—given there are no other blocks on top of the moved one.

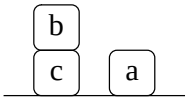


Figure 3.1: An example of a Blocks World state.

3.5 Representation of States and Actions

A Blocks World state can be described using a Prolog list which includes the following facts: `on(a,b)` expresses that block `a` is on top of block `b` and `clear(c)` says that there are no blocks on top of block `c`. The example of a Blocks World state in Figure 3.1 can be described as a series of relational facts: `clear(b), on(b,c), on(c,floor), clear(a), on(a,floor)`. This series of facts constitutes a Prolog program in itself on which tests can be run.

Actions can also be described using Prolog predicates. In a Blocks World scenario with three blocks, `a`, `b`, and `c`, the available actions are `move`(x, y) where $x \neq y, x \in \{\texttt{a}, \texttt{b}, \texttt{c}\}, y \in \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{floor}\}$. If block `b` is to be moved on top of block `a` in the above-mentioned example, this would be expressed as `move(b,a)`.

An implementation of Blocks World in Prolog is shown in Table 3.1 [14, 15]. Here, a state is represented by a variable `S` which is a concatenation of the facts that describe the state. The auxiliary predicate `holds` checks whether all facts in a list are in `S`. The implementation also models the precondition function as the predicates `pre` which specifies criteria for the validity of an action in a given state. E.g. the lower `pre` predicate says that the preconditions for a `move` action that moves a block `X` to the floor, is that `X` is clear and that `X` is not already on the floor. The `pre` predicates utilise `holds` to check whether these two criteria are met by the state `S`.

The transition function is modelled as the predicates `delta`. These predicates take in a state `S` and an action as parameters, and yield a new modified state `NextS`. First, a `delta` checks whether the action is valid in the given state. For that it uses the tests that are also contained in the `pre` relations.

Notice that the first two `pre` predicates are alike, except that in one of the predicates, one of the predicates (`on(X,floor)`) is negated. This is also the case in the first two `delta` predicates since they utilise the same `holds` tests as `pre`. The reason that these two relations cannot be collected into one is that for one `delta`, block `X` may not be on the floor, while for the other, it must. This difference is also the reason that `pre` is not used directly in `delta`; it is not irrelevant *which* `pre` is tested.

The first definition of `delta` is for `move(X,Y)` where block `X` is on a third block (i.e. not on `Y` and not on the floor). A new temporary state `S1` is created where `Y` is not clear anymore—since we moved `X` on top of it—and `X` is not on the other block anymore. This is done by simply removing the `clear(Y)` and `on(X,Z)` relations from `S`. At the same time a new variable `Z` is instantiated to that other block from which `X` was moved.

Finally, two new relations are added to `S1` yielding `NextS` so that `Z` is now clear and `X` is now on `Y`. Thereby we have successfully altered the state `S` to reflect the changes necessary for a block `X` to be moved onto `Y`.

3.5.1 Generalisation

A key feature of RRL is that states are described relationally. This allows for a decrease in the size of the state space since it is possible to generalise over states. For instance,

```
pre(S,move(X,Y)) :-
   holds(S,[clear(X), clear(Y), not X=Y, not on(X,floor)]).
pre(S,move(X,Y)) :-
   holds(S,[clear(X), clear(Y), not X=Y, on(X,floor)]).
pre(S,move(X,floor)) :-
   holds(S,[clear(X), not on(X,floor)]).

holds(S,[]).
holds(S,[not X=Y | R]) :-
   not X=Y, !, holds(S,R).
holds(S,[not A | R]) :-
   not member(A,S), holds(S,R).
holds(S,[A | R]) :-
   member(A,S), holds(S,R).

delta(S,move(X,Y),NextS) :-
   holds(S, [clear(X), clear(Y), not X=Y, not on(X,floor)]),
   delete([clear(Y),on(X,Z)], S, S1),
   add([clear(Z),on(X,Y)], S1, NextS).
delta(S, move(X,Y), NextS) :-
   holds(S, [clear(X), clear(Y), not X=Y, on(X,floor)]),
   delete([clear(Y),on(X,floor)], S, S1),
   add([on(X,Y)], S1, NextS).
delta(S, move(X,floor), NextS) :-
   holds(S, [clear(X),not on(X,floor)]),
   delete([on(X,Z)], S, S1),
   add([clear(Z),on(X,floor)], S1, NextS).

goal(S) :-
   member(on(a,b), S).
```

Table 3.1: Implementation of `pre`, `holds`, and `delta` for Blocks World in Prolog.

learning the results of moving block `a` to block `b` is the same as learning the results of moving block `b` to block `a`.

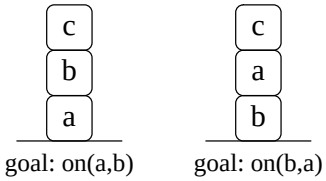


Figure 3.2: Two Blocks World states with goals. These can be generalised.

The fact that Prolog variables are used instead of constants is important since this allows us to generalise over many states: As illustrated in Figure 3.2, if e.g. the goal is `on(a,b)` and the state has block `c` on top of `b` which is on top of `a` (the wrong order of `a` and `b` according to the goal), then that situation would be the same as one where the goal is `on(b,a)` and the state has block `c` on `a` on `b`. Also, it does not matter whether the upper block is in fact `c` or `d` or any other block. This is of course the case since the same actions have to be learnt regardless of the names of the blocks, assuming that the blocks are otherwise similar and have the same properties. A solution for both cases is shown in Figure 3.3. In effect, blocks not represented in the goal state are interchangeable.

Further generalisation can be achieved by specifying background information relations, such as `above(A,B)` which recursively checks whether block `A` is above block `B`, possibly with other blocks between them.

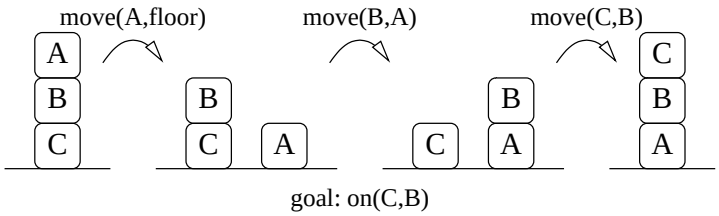


Figure 3.3: A general solution that works for both scenarios in Figure 3.2.

3.6 Relational Q Learning Algorithm

The relational Q learning algorithm (Q-RRL) expands the algorithm for TRL shown in Table 2.2. The purpose of the Q-RRL algorithm is to learn the Q function by generating a set of *examples* that can later be used for the induction of Q *trees* (see Section 3.7).

Each example is a state, an action, a goal state, and a corresponding Q value. Figure 3.4 (from [15]) illustrates an episode with four such examples. States are represented graphically while actions and Q values are specified below. Notice that the third action leads to a goal state (it gets Q value 1) while the last action gets Q value 0. This is because after entering an absorbing goal state, no further rewards can be expected (refer to Figure 2.2). Precise descriptions of the four examples (Q values, actions, and state descriptions) are given in Table 3.2 (also from [15]).

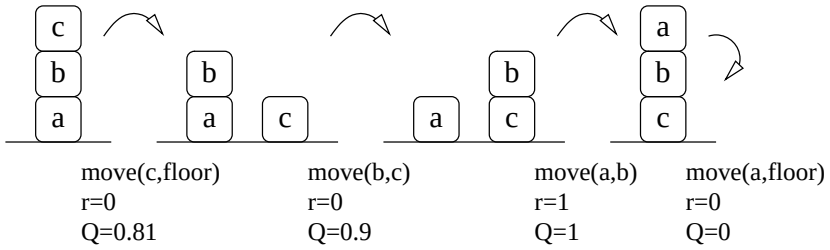


Figure 3.4: An episode with four examples for Blocks World with graphically represented states. Discount factor $\gamma = 0.9$ and $goal(on(a, b))$.

Example 1	Example 2	Example 3	Example 4
`qvalue(0.81).`	`qvalue(0.9).`	`qvalue(1.0).`	`qvalue(0.0).`
`move(c,floor).`	`move(b,c).`	`move(a,b).`	`move(a,floor).`
`goal(on(a,b)).`	`goal(on(a,b)).`	`goal(on(a,b)).`	`goal(on(a,b)).`
`clear(c).`	`clear(b).`	`clear(a).`	`clear(a).`
`on(c,b).`	`clear(c).`	`clear(b).`	`on(a,b).`
`on(b,a).`	`on(b,a).`	`on(b,c).`	`on(b,c).`
`on(a,floor).`	`on(a,floor).`	`on(a,floor).`	`on(c,floor).`
	`on(c,floor).`	`on(c,floor).`	

Table 3.2: Four examples generated from the episode in Figure 3.4.

The Q-RRL algorithm is presented in Table 3.3. As mentioned, it is based on the algorithm for TRL (in Table 2.2) and thus they are similar. We initially set all $\hat{Q}$ values to 0, and generate a random initial state s_0. Examples are stored in a set *Examples* which is initially empty. The algorithm then enters a loop where in each iteration, an action is chosen according to an exploration policy (see Section 2.4). The action is performed on the current state s_i, whereafter the resulting state s_{i+1} and the corresponding reward are observed.

An example is created using a state s_i, the action a_i performed in that state, the goal state, and a $\hat{Q}$ value which is calculated on basis of the reward r_i. If a generated example is already in *Examples*, the old example is replaced, otherwise the new one is inserted. Finally, the $\hat{Q}$ function is updated, which is the topic of the next section.

3.7 Approximating Q and P with Decision Trees

Using the Q-RRL algorithm, we can generate a number of examples. The question now is how to use these examples to generate approximations of the Q and P functions without having the same form of increasing space complexity as in traditional reinforcement learning. One answer to this question is to combine the generalisation obtained by using inductive logic programming with decision trees.

3.7.1 Logical Decision Trees

The term *decision tree* describes two types of trees used to approximate functions, namely *classification trees* and *regression trees*. The primary difference is that classi-

---

$\forall s, a$ **repeat**
  $\hat{Q}_0(s, a) \leftarrow 0$
`Set` *Examples* `to the empty set`
$e \leftarrow 0$
Repeat `forever`
  $e \leftarrow e + 1$
  $i \leftarrow 0$
  $s_0 \leftarrow$ `a random state`
  While not $goal(s_i)$ `do`
    $a_i \leftarrow$ `stochastically chosen action`
      `using the` Q `exploration strategy from Equation 2.7`
      `using the current approximation of` $\hat{Q}_e$
    `Perform` a_i
    $r_i \leftarrow r(s_i, a_i)$
    $s_{i+1} \leftarrow \delta(s_i, a_i)$
    $i \leftarrow i + 1$
  For $j \leftarrow i - 1$ **to** 0 **do**
    `Generate example` $x = (s_j, a_j, \hat{q}_j)$`,`
      `where` $\hat{q}_j \leftarrow r_j + \gamma \max_{a'} \hat{Q}_e(s_{j+1}, a')$
    If `an example` $(s_j, a_j, \hat{q}_{old})$ `exists in` *Examples*`,`
      `replace it with` x`,`
    else `add` x `to` *Examples*
  `Update` $\hat{Q}_e$ `to produce` $\hat{Q}_{e+1}$ `using` *Examples*

Table 3.3: The Q-RRL algorithm [15] for relational reinforcement learning.

fication trees approximate functions that output discrete values, and regression trees approximate functions that output continuous values. More general information on decision trees can be found in [22].

A special kind of decision tree, which uses first order logic, is formally introduced in [9], and has since been used to approximate the Q and P functions in RRL [15]:

Definition 3.7.1 (FOLDT). *A first order logical decision tree (FOLDT) is a binary decision tree in which*

1. *the nodes of the tree contain a conjunction of literals, and*
2. *different nodes may share variables, under the following restriction: a variable that is introduced in a node (which means that it does not occur in higher nodes) must not occur in the right branch of that node.*

Each node in a FOLDT is a test, which splits a list of examples in two using a logical query (we use Prolog queries). The result of a node test is either `yes` (left branch) or `no` (right branch)—thus a FOLDT is a binary decision tree. Using variables makes it possible to express node tests such as `on(A,B)` which tests if any block `A` is on top of any block `B`. Combining this with variable sharing between nodes allows the expression of conjunctions like `on(A,B), on(B,floor)`, where `B` references the same block in both parts of the conjuction. Figure 3.5 shows an FOLDT example illustrating the basic idea of variable sharing.

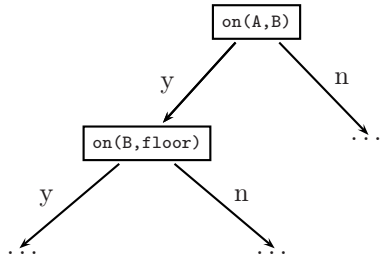


Figure 3.5: A FOLDT illustrating variable sharing.

An extra *root* is often attached to a tree. This root contains tests that always succeed—thereby creating relevant variable instantiations that the node tests can use. In Blocks World, a root might be

```
goal(on(A,B)), action(move(C,D))
```

where `A` and `B` will be instantiated with the values of the actual goal state, e.g. `A` = `a` and `B` = `b` for `goal(on(a,b))`. The variables `C` and `D` will be instantiated with the values of the action being tested, e.g. `C` = `a` and `D` = `c` for the action `move(a,c)`.

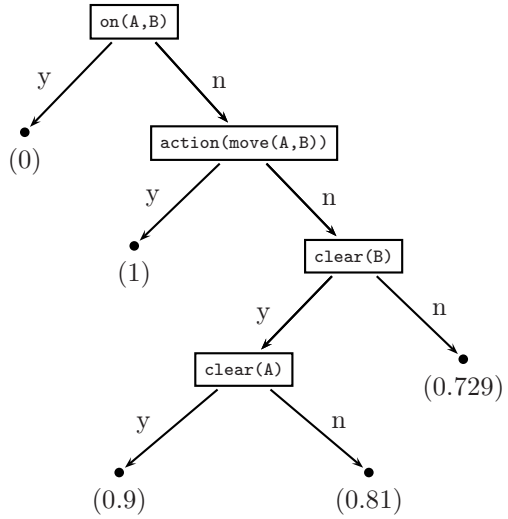
Like other decision trees, their logical counterparts are created by feeding a set of examples to a tree induction algorithm. The induction of logical decision trees is described in detail in Chapter 4.

3.7.2 Q Trees

An example of a logical regression tree representing an approximation $\hat{Q}$ is illustrated in Figure 3.6. The example is constructed from the Blocks World domain. Besides

the graphical representation of the Q tree, an equivalent Prolog program is shown. To represent the tree in Prolog, a rule based if-then-else construction is used. The cut operator (`!`) denotes that if the test in question succeeds, then no other tests are performed. This makes the order of the tests an important part of the construction.

```
root:  goal(on(A,B)), action(move(C,D))
```




```
q(0)    :- goal(on(A,B)), action(move(C,D)), on(A,B), !.
q(1)    :- goal(on(A,B)), action(move(C,D)), action_move(A,B), !.
q(0.9)  :- goal(on(A,B)), action(move(C,D)), clear(B),
           clear(A), !.
q(0.81) :- goal(on(A,B)), action(move(C,D)), clear(B), !.
q(0.729).
```

Figure 3.6: Q tree from Blocks World and its equivalent Prolog program.

Using Blocks World, it is easy to see why the second restriction of Definition 3.7.1 is needed. When testing if any block `A` is on top of another block `B`, and the answer is `no`, it makes no sense to reference `A` again, since there is no such block. Notice, however, that it *is* possible to reuse variable names when going down a `no` branch in a tree represented as Prolog rules. This is possible because of basic programming language scope rules.

Example: Choosing the Best Action in Blocks World

Figure 3.1 shows an example scenario from Blocks World. In the following, we assume that the Q tree from Figure 3.6 has previously been learnt. There are three possible actions that the agent can perform: `move(B,floor)`, `move(B,A)` and `move(A,B)`. To check which action gives the highest cumulative reward, three examples are generated in which the scenario and the goal state is combined with the respective actions. The generated examples are shown in Table 3.4.

Example 1	Example 2	Example 3
goal(on(a,b)).	goal(on(a,b)).	goal(on(a,b)).
move(b,floor).	move(b,a).	move(a,b).
on(a,floor).	on(a,floor).	on(a,floor).
on(b,c).	on(b,c).	on(b,c).
on(c,floor).	on(c,floor).	on(c,floor).
clear(a).	clear(a).	clear(a).
clear(b).	clear(b).	clear(b).

Table 3.4: Blocks World examples.

Using the Q tree from Figure 3.6, the Prolog tests of Table 3.5 are performed on all three examples. Recall that the predicates `goal(on(A,B))` and `action( move(C,D))` always succeed and just instantiate the relevant variables `A`, `B`, `C` and `D` for the following predicates. For instance, in the first test, when applied to Example 3, `A` = `a`, `B` = `b`, `C` = `a` and `D` = `b`.

q(0)	:-	goal(on(A,B)), action(move(C,D)), on(A,B), !.
	→	Example 1 fails, Example 2 fails, Example 3 fails
q(1)	:-	goal(on(A,B)), action(move(C,D)), action(move(A,B)), !.
	→	Example 1 fails, Example 2 fails, Example 3 succeeds
q(0.9)	:-	goal(on(A,B)), action(move(C,D)), clear(B), clear(A), !.
	→	Example 1 succeeds, Example 2 succeeds

Table 3.5: Example Classification with Prolog tests.

Furthermore, notice that after Example 3 succeeds in the second test, it is not tested in following tests because of the cut operator. This assures that an example is classified with one, and only one, Q value.

As a result of the tests, Example 1 and 2 are classified with a Q value of 0.9, while Example 3 is classified with a value of 1. If the agent were playing to exploit what it had already learnt, it should therefore choose the action `move(a,b)`.

3.7.3 P Trees

The $Q(s,a)$ function encodes a quality value of performing action a in state s. This is useful while an agent is still learning, where each quality value can be updated individually according to experience. But when an agent is done learning, only optimal actions are interesting, and thus the Q function encodes more information than necessary.

In essence, we would like to make a discretisation over the range of Q values for a state and its possible actions, such that the pair that returns the highest Q value is classified as optimal and all others as non-optimal. In [15] the authors define the policy function P for this purpose:

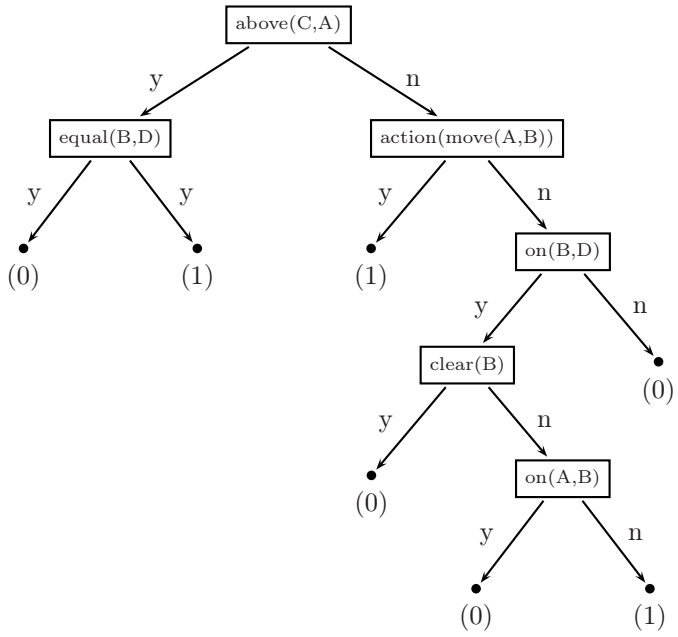
$$\textbf{If } a = \arg\max_a Q(s,a) \textbf{ then } P(s,a) = 1 \textbf{ else } P(s,a) = 0 \tag{3.1}$$

where 1 indicates an optimal and 0 indicates a non-optimal action.

Besides being more compact, the results of the experiments performed in that paper suggest that the optimal P function, for a given problem, is learnt faster than the optimal Q function—even if the two functions are learnt from the same set of examples. The theoretical explanation for this is given in Section 3.8.

We will use the notation $\hat{P}$ for the current approximation of the P function. An approximation $\hat{P}$ is modelled as a logical classification tree (see Definition 3.7.1), because we only need to classify a state/action pair as either optimal or non-optimal. For an example of a P tree, refer to Figure 3.7 which shows the optimal P tree for Blocks World using three blocks.

```
root:  goal(on(A,B)), action(move(C,D))
```




```
p(nonoptimal) :- goal(on(A,B)), action_move(C,D), above(C,A),
                 equal(B,D), !.
p(optimal)    :- goal(on(A,B)), action_move(C,D), above(C,A), !.
p(optimal)    :- goal(on(A,B)), action_move(C,D),
                 action_move(A,B), !.
p(nonoptimal) :- goal(on(A,B)), action_move(C,D), on(B,D),
                 clear(B), !.
p(nonoptimal) :- goal(on(A,B)), action_move(C,D), on(B,D),
                 on(A,B), !.
p(optimal)    :- goal(on(A,B)), action_move(C,D), on(B,D), !.
p(nonoptimal).
```

Figure 3.7: Optimal P tree, and its equivalent Prolog program, for Blocks World using three blocks.

To understand the illustrated P tree, consider the two scenarios in Figure 3.8. The

right scenario is classified as non-optimal, because it satisfies the conjunction

```
above(C,A),equal(B,D)
```

On the other hand, the left scenario is classified as optimal, because it only satisfies `above(C,A)`. In other words, it is considered optimal to move away any block sitting on top of `A`, as long as the block is not moved on top of `B`.

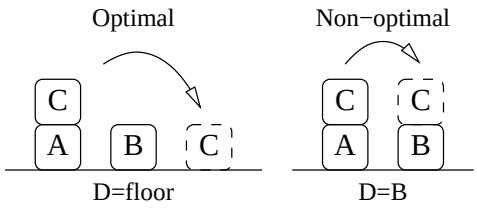


Figure 3.8: Classification of optimal and non-optimal actions using the P tree from Figure 3.7.

The P function is defined in terms of the Q function, which means that for every possible approximation $\hat{Q}$ of Q, a corresponding approximation $\hat{P}$ of P exists. Since we have a new approximation $\hat{Q}$ at the end of each iteration of the Q-RRL algorithm, we can extend the algorithm, such that it also learns a new approximation $\hat{P}$. To achieve this, the pseudo-code in Table 3.6 [15] should be added to the end of the Q-RRL algorithm. The added code builds a new $\hat{P}$ every time a new $\hat{Q}$ is ready at the end of an episode. It runs through every possible action for each visited state in the episode. For each state/action pair, it creates an `optimal` example if the pair maximises the value of $\hat{Q}$. Otherwise, a `nonoptimal` example is created. Finally, the examples are fed to a tree inducing algorithm in order to update the $\hat{P}$ tree.

```
...
for j = 1 to 0 do
   for all actions a_k possible in state s_j do
      if state action pair (s_j; a_k) is optimal according to Q̂_{e+1}
         then generate example (s_j; a_k; c) where c = 1
         else generate example (s_j; a_k; c) where c = 0
update P̂_e to produce P̂_{e+1} using these examples (s_j; a_k; c)
```

Table 3.6: Learning P-trees from Q-trees (P-RRL).

3.8 Convergence of Q and P

It is a well known fact that $\hat{Q}$ converges to Q given an infinite number of episodes when using a tabular representation of $\hat{Q}$. By changing the representation of $\hat{Q}$ from a table to a decision tree, the convergence becomes dependent on the set of generalising tests T_Q, where $t_Q \in T_Q$ is a test that can be performed in an internal tree node. In essence, the total expressiveness of T_Q must be able to separate any two state/action pairs, if they do not have the same quality value.

The above is easily achieved by exhausting combinations of state predicates using all possible constants—e.g. `on(a,b)`, `on(a,c)`, `on(b,a)`, `on(b,c)` and so on. However, using constants opposes the principle of object generalisation, and the resulting Q tree would be practically unusable in any other setting, besides the one in which it was learnt.

In practise, the total convergence of $\hat{Q}$ is not that important. What *is* important, however, is that, for any given state s, the action that maximises Q also maximises $\hat{Q}$. This statement also translates into saying that we are mostly interested in the convergence of $\hat{P}$. Since the convergence of $\hat{P}$ is a relaxation of the convergence of $\hat{Q}$, two properties should be noted:

- The optimal $\hat{P}$ is learnt faster than the optimal $\hat{Q}$, and
- less expressiveness is required of the set of tests T_P to represent the optimal $\hat{P}$ tree, than of T_Q to represent the optimal $\hat{Q}$ tree.

It is, however, unlikely that P can be learnt without using a state/action quality measure such as the Q function. This makes the second property less important, because even though T_P is easier to specify, we still need T_Q to create $\hat{Q}$.

3.9 Guidance

In large domains, even the state generalisations possible with RRL will not make it possible to learn anything sensible. In such domains, the rewards may be distributed so sparsely in the state space that they are almost never encountered with random exploration, which is often used in the beginning of the learning process. In such domains RRL will not be able to produce a policy without using a very long time. E.g. try to imagine a Blocks World start state with 50 blocks, all placed on the floor, and with the goal state that they all should be stacked on top of each other. Using random exploration, it would probably take a very long time to reach a goal state even for a powerful machine.

This problem can be solved with guidance. An agent can for instance be guided by providing it with a "reasonable" policy in the beginning of the learning process. One way to create this policy is by logging the behaviour of a human expert and present this to the Q tree induction algorithm. The agent can then use the built Q tree to explore the state space, and thus be directed towards more rewards. In [12] the authors compare RRL with guided RRL, and they show that guidance improves the learning rate of RRL.

Another strategy is to supply guidance interleaved with the normal exploration policy of the agent. One benefit of the interleaving strategy is that the agent can consult the expert when a state is encountered in which it has insufficient experience to choose an action. This allows the guidance to focus on areas of the state space which have not yet been explored.

A third strategy is to use bootstrapping. In [15] the authors use bootstrapping to learn policies in Blocks World. They argue that optimal policies are learnt faster when starting with a small number of blocks and gradually increasing this number. The policy learnt in a world with four blocks and the goal `on(A,B)` after two episodes will produce non-optimal behaviour in 12 out of the 73 states. If the optimal Q tree from a world with three blocks and the goal `on(A,B)` is used at the beginning, the policy will only produce non-optimal behaviour in 8 of the 73 states. Another advantage is

that the generated tree is much smaller. Without bootstrapping it consist of 44 nodes while it only has 27 nodes with bootstrapping.

Try to imagine a Blocks World start state with 50 blocks, all placed on the floor, and with the goal state that they all should be stacked on top of each other. Using random exploration, it would probably take a very long time to reach a goal state even for a powerful machine.

3.10 Summary

In this chapter, the combination of traditional reinforcement learning and inductive logic programming, known as relational reinforcement learning, has been described. The technique reduces the state space of a problem domain by using logical generalisations.

A highly relational domain, Blocks World, has been used to show the advantages of generalisation. The number of states that essentially differ in Blocks World with three blocks (using the goal `on(a,b)`) can be reduced from 13 to 11. For four blocks, the reduction is from 73 to 66. This ratio increases combinatorially since all blocks that do not occur in the goal state are interchangeable. Depending on the existing background information, the state reduction can become much greater.

The Q function, denoting the real-valued quality of a state/action pair, and the policy function P, are approximated using logical decision trees with the ability of sharing variables between node tests.

Domains can, however, also get too large for relational reinforcement learning alone. Rewards may be distributed so sparsely, that they are unlikely to ever be encountered. This problem can be solved with various kinds of guidance, such as bootstrapping or human interference.

In the next chapter, the described techniques, together with various algorithms for inducing logical decision trees, are implemented in a toolbox. The toolbox is later used for conducting experiments on the performance of relational reinforcement learning.

Chapter 4

RRL Toolbox

The previous chapter presented a formal description of RRL, and used the Blocks World as an example. Of course, the concept of RRL can be used on a wide variety of problems, and most of the involved processes can be generalised and automated. To help us conduct our experiments, we have therefore implemented an RRL toolbox for generating examples and inducing Q trees. This chapter describes both the functionality and interface of the toolbox, but also theoretical considerations about possible future extensions. We will again use Blocks World as an ongoing example.

Section 4.1 describes the specification and architecture of the toolbox. The task of using the toolbox to specify a problem such as Blocks World is explained in Section 4.2. Section 4.3 and Section 4.4 describe how the toolbox uses such a specification to generate examples and create an agent policy represented as a Q or P tree, respectively. Finally, in Section 4.5 we review experimental results constructed using the toolbox for the Blocks World domain.

4.1 Toolbox Specification

The main purpose of the toolbox is to test the performance of RRL in a non - deterministic multi-agent environment. In this chapter, however, we will only consider deterministic single-agent environments such as Blocks World. This will allow a more smooth transition from the formal descriptions (as presented in the previous chapter) to the actual implementation. It will also allow us to validate the toolbox implementation by comparing experimental results with other experiments performed using the Blocks World domain [15]. In the next chapter, the toolbox is extended to handle non-deterministic environments with multiple competing and cooperating agents.

A secondary goal of the toolbox is that it should be as general as possible. That is, it should be able to handle any given deterministic single-agent problem, not just Blocks World. Since all experiments can be conducted using only Q trees, the toolbox will not support P trees.

4.1.1 Architecture

Q learning can be divided into problem specification, example generation and decision tree induction. As a consequence, the toolbox has been decomposed into three such

components. Figure 4.1, showing the toolbox architecture, illustrates this decomposition. It also illustrates how the toolbox utilises SWI Prolog/JPL and InterProlog.

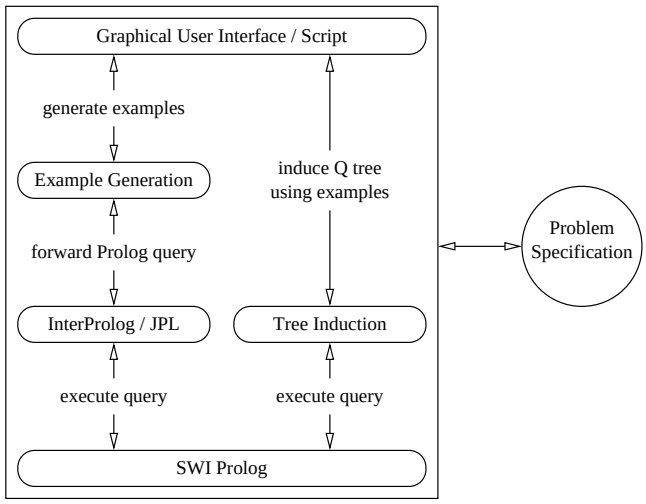


Figure 4.1: Toolbox architecture.

As illustrated, a graphical user interface (GUI) or script is used to combine the example generation and tree induction components. The two components communicate with Prolog, which consults the problem specification (that is mostly written in Prolog code as described later in this chapter). The problem specifcation is also consulted by other components to look up various constants.

During testing of the toolbox, it became apparent that InterProlog was a bottleneck, because it handles the communication between Java and SWI Prolog through sockets. We therefore tested the Java Prolog Library (JPL) which is part of the SWI Prolog package. The library connects to the foreign interface of SWI Prolog through the Java Native Interface (JNI), and this resulted in an about 2 to 1 optimisation of Prolog queries. To optimise further, we decided to implement the tree induction algorithm in C++, connecting directly to the foreign interface of SWI Prolog.

4.2 Problem Specification

Observed from the outside, the toolbox accepts a problem specification and returns an agent policy. Using the toolbox, the specification is entered in a GUI, which is specifically designed to handle single-agent problems like Blocks World. Alternatively, the specification can be enterted directly in the toolbox file structure. The specification is divided into five sub-specifications:

- Initial states specification
- Transition specification
- Goal state specification
- Auxiliary predicates specification
- Decision tree tests specification

- Declarative bias specification

Each sub-specification is explained in more detail in the following sections. When compiled, the combination of all the specifications results in a Prolog knowledge base, a file representing the declarative bias, and a file representing the initial states. The knowledge base includes both the `pre` and `delta` predicates as well as the implementation of decision tree tests and any auxiliary predicates. It also includes a goal state predicate.

The declarative bias file encodes information about type and mode restrictions on the decision tree tests. It also contains user specified root tests, which are tests that always succeed. This is explained in Section 4.2.5. Finally, all possible initial states are listed together in their own file.

Screenshots of the GUI can be found in Section 4.2.6 and in Appendix A. A complete specification of Blocks World can be found in Appendix B.

4.2.1 Initial States

An initial state $s_0 \in S_0$ is a state from which an episode is started (refer to the Q-RRL algorithm in Table 3.3). Many problems always start out from the same initial state. For instance, a game of chess always begins from a certain board configuration. This can be a problem for a learning algorithm, since it might take a very long time before less probable states are visited. Instead, all problems can be assigned a set of possible initial states S_0, and then it can be left to the toolbox to pick a random one in the beginning of each episode. Fruthermore, this makes it possible to guide the learning agent in the right direction, if it is discovered that it does not perform well in certain states.

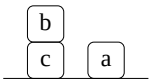


Figure 4.2: A possible initial state in Blocks World with three blocks.

An initial state is specified as a list of non-inferable facts. For instance, the Blocks World scenario illustrated in Figure 4.2 is specified as

```
clear(a),clear(c), on(c,floor),on(b,c),on(a,floor)
```

All initial states are stored in a file using this format, and is fetched by the toolbox when learning is initialised. For domains where the set of initial states is too large to specify directly, a user should create a state generator for the particular domain.

Another solution would be to let the toolbox pick an initial state from the entire set of possible states S (leaving out goal states). This solution would require a specification of all non-inferable facts and the existing constraints on their parameters, because the toolbox would have to auto-generate the initial state[1].

[1] For very small domains, the set of states S could be specified directly

4.2.2 Transitions and the Goal State

State transitions are performed by executing actions, where the available actions in a state are defined by the precondition function `pre`. The precondition function have the interface `pre(S, A)` where `S` is a state, and `A` is an action. It succeeds if action `A` is allowed in state `S`, otherwise it fails. It must be specified in Prolog code along with the `delta` transition function.

In Blocks World, only one action type exists, namely `move(X,Y)`, but the blocks that can be moved varies across different states. Below is a recap of the precondition function for Blocks World[2]:

```
pre(S,move(X,Y)) :-
   holds(S,[clear(X), clear(Y), not X=Y, not on(X,floor)]).
pre(S,move(X,Y)) :-
   holds(S,[clear(X), clear(Y), not X=Y, on(X,floor)]).
pre(S,move(X,floor)) :-
   holds(S,[clear(X), not on(X,floor)]).
```

During example generation, the toolbox needs to find all available actions for any given state. By using the predicate `findall`, Prolog can backtrack over all possible solutions to a query, and return the solutions in a list. For instance, the query

```
S=[on(a,floor),on(b,floor),on(c,b),clear(a),clear(c)],
findall(A, pre(S, A), Bag).
```

where `S` is a state and `A` is an action, would return

```
Bag = [move(c,floor),move(c,a),move(a,c)].
```

The actual transition is performed by the `delta` function. It has the interface `delta(S, A, NextS)`, where `NextS` is the state resulting from performing action `A` in state `S`. As an example, the query

```
S=[on(a,floor),on(b,floor),on(c,b),clear(a),clear(c)],
delta(S, move(a,c), NextS).
```

would return `NextS = [on(a,c),on(b,floor),on(c,b),clear(a)]`.

The toolbox expects all problems to have an absorbing goal state. The goal state, which can be a generalisation, must be specified with a `goal` predicate. For instance, the Blocks World goal of stacking block `a` on top of block `b` is specified as

```
goal(S) :- holds(S, [on(a,b)]).
```

which succeeds if the predicate `on(a,b)` holds in the state `S`.

In its current version, the toolbox only allows goal state rewards. This is no problem in Blocks World, where no more is necessary. However, some problems might be better specified with a more extended use of rewards. Using state abstraction, the toolbox can easily be extended with a reward rule such as

[2]Refer to Section 3.5 for an explanation of the similarity of the two topmost precondition rules in Blocks World

```
r(0.5,E) :- action(move(A,floor),E)), above(A,B,E).
```

where `E` is an example combining a state, an action and a goal state (as described in Chapter 3). The rule gives a reward of 0.5 for moving block `A` to the floor if it was previously above block `B`. The rule should be invoked before an action is executed, and could furthermore be made non-deterministic if needed. This can be achieved by the use of Prologs `random` predicate. In the rest of the report, only goal state rewards will be considered.

Notice that the action in an example, e.g. move(a,b), is encapsulated in an `action` predicate yielding `action(move(a,b))`. To test if the action performed in an example `E` is `move(a,b)`, the query `action(move(a,b),E)` is executed. Similar, the goal state is encapsulated in a `goal` predicate. The encapsulation also makes it easier to parse examples, especially when several action types exist (unlike Blocks World where the only action type is `move`).

4.2.3 Tests and Background Knowledge

The test assigned to a decision tree node can either be a generalised non-inferable facts or background knowledge. It might not always be desirable, however, to allow all non-inferable facts as tests. For instance, to construct a Blocks World policy that never uses the test `on(A,B)` directly, we could specify that `above(A,B)` should be used instead. For some problems, this might result in a more general policy.

The toolbox therefore initially assumes that no non-inferable facts can be used as tests. To be considered, they must be specified directly and are therefore no different from background knowledge. A test is specified by its name, its (possible empty) list of parameters and its Prolog implementation. The toolbox automatically adds an extra parameter `E` to each test, which is a Prolog list representing the example to be tested. The following rules specifies various Blocks World tests:

```
          on(X,Y,E)   :-  member(on(X,Y),E).
action(move(X,Y,E))   :-  member(action(move(X,Y)),E).
    goal(on(X,Y,E))   :-  member(goal(on(X,Y)),E).
         clear(X,E)   :-  member(clear(X),E).
       above(X,Y,E)   :-  on(X,Y,E).
       above(X,Y,E)   :-  on(X,Z,E), above(Z,Y,E).
       equal(X,X,E)       .
```

As illustrated, the implementation of a non-inferable test (like `on(X,Y,E)`) is just a member check. For instance, the test `on(X,Y,E)` checks if `on(X,Y)` exists in the example `E`. Background knowledge, such as `above(X,Y,E)`, is implemented as a conjunction or disjunction of non-inferable facts and/or other background knowledge.

Remember from Chapter 3 that some tests never fail and only has the purpose of instantiating variables. With the relatively few tests necessary in Blocks World, such tests can be located in the root of a Q or P tree. For larger problems, this might not, however, be the best solution. This topic is described further in Section 4.4.1.

While this takes care of the specification of tests, we still need to specify a declarative bias, so that the toolbox continuously can generate the actual set of possible tests for any given node in a tree.

4.2.4 Auxiliary Predicates

In the previous sections, we have used the predicate `holds`, which is not a standard Prolog predicate. The predicate makes it possible to test if a list of other predicates holds in a given state, e.g. `holds(S,[on(b,a), clear(b)])` tests if `on(b,a)` and `clear(b)` holds in the state `S`. The same could be achieved by using a conjunction of `member` predicates, but this would make the Prolog code less intuitive to read and more difficult to alter. The toolbox therefore supports these kinds of auxiliary predicates, which must be specified in the Prolog knowledge base like any other predicate.

4.2.5 Declarative Bias

A declarative bias assigns type and mode restrictions for the set of tests. Type checking is handled by allowing a user to construct a list of object types in a particular problem domain. Each parameter in a test must be assigned exactly one of these types. In Blocks World, only one object type exists, namely `Block`. In other problems, however, several object types, which must not be mixed, might exist. The list of object types is specified in the declarative bias file as follows:

```
begin(types)
   Block
   Person
   Country
end(types)
```

Mode checking is specified by denoting, for each parameter in a test, if the particular parameter can be replaced by an uninstantiated and/or an instantiated variable. The current version of the toolbox does not allow constants in a test[3]. In the following specification, a plus(+) denotes that a parameter can be replaced by an instantiated variable, and a minus(-) denotes that it can be replaced by an uninstantiated variable. The specification combines type and mode restrictions for each parameter.

```
begin(tests)
   on(+-Block,+-Block)
   clear(+-Block)
   above(+-Block,+-Block)
   equal(+Block,+Block)
end(tests)
```

Parameters of always succeeding tests assigned to a root, can furthermore be assigned an identification number, so that the same variables can be referenced in several root tests. The root used so far in Blocks World does not need this functionality and can be specified as

```
begin(root)
   goal_on(Block-0,Block-1)
   action_move(Block-2,Block-3)
end(root)
```

[3]To test if a block is on the floor in Blocks World, a test like `onfloor(A)` can be specified. It turns out, however, that this is not necessary to represent an optimal policy for the goal `on(a,b)`.

However, to allow the specification of a goal test such as `goal(on(A,B),on(B, C))`, denoting that block `A` must be on top of block `B` and that the same block `B` must be on top of block `C`, the toolbox must have this ability.

4.2.6 Graphical User Interface

The GUI of the toolbox contains a screen for each sub-specification described in the previous sections. The screens can be accessed via tabs as illustrated in Figure 4.3 and Figure 4.4. Basically, the GUI allows a user to enter a problem specification as Prolog code and to set relevant constants, such as the discount and exploration factor. For platform independency, it is implemented using the Java 2 Platform Standard Edition 5.0.

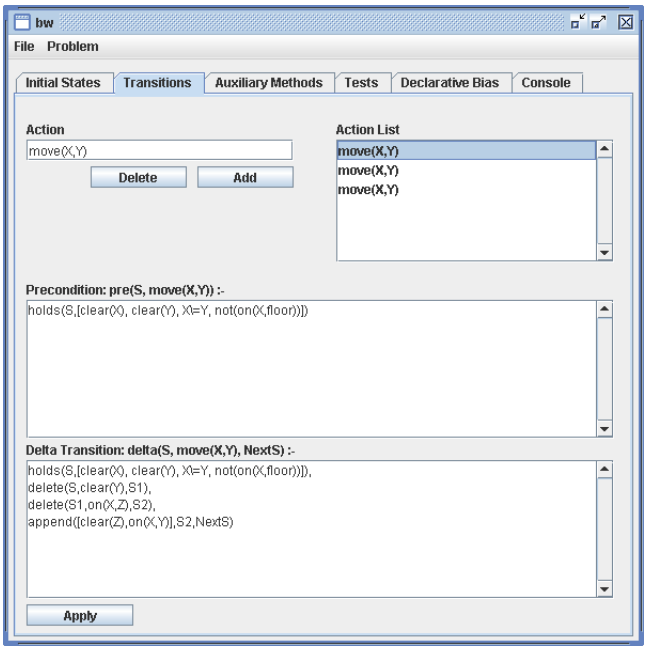


Figure 4.3: The GUI for specifying transitons.

The GUI is not necessary for using the toolbox, since all specifications can be constructed as described in the previous sections. The benefits of using the GUI anyway, are that it does not allow any illegal specifications to be compiled, and that it clearly illustrates the capability of the toolbox. More screenshots of the toolbox GUI can be found in Appendix A.

4.3 Example Generation

To generate examples, we utilise the Q-RRL algorithm from Table 3.3. Table 4.1 illustrates how the specified `pre` and `delta` functions are used to progress through any problem state space. Since the interface of the two functions are the same for all problems, the illustrated algorithm combined with the Q-RRL algorithm can be used to generate examples for any problem.

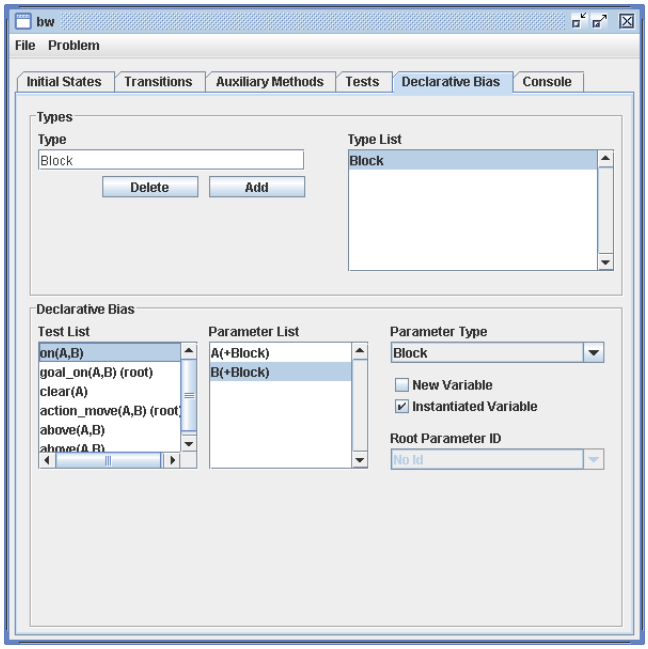


Figure 4.4: The GUI for specifying a declarative bias.

```
S ← chooseRandomInitialState()
While not goal(S) do
   Actions ← pre(S)
   A ← chooseActionUsingExplorationPolicy(Actions)
   S ← delta(S,A)
```

Table 4.1: Using `pre` and `delta` to progress through a problem state space.

Each example in a generated episode is assigned a Q value. The example leading to an absorbing goal state is assigned the value 1, and any example generated while already being in a goal state is assigned the value 0. Any other example E is assigned a value using the formula

$$\hat{q}_E \leftarrow \gamma \max_{a'} \hat{Q}(s_{E+1}, a') \tag{4.1}$$

where s_{E+1} is the state in the example following example E in the episode.

For optimisation reasons, we only update $\hat{Q}$ when we have assigned new Q values to an entire episode. This means that we actually have two Q values for the example $E+1$, namely the value denoted by $\hat{Q}$, but also the value assigned to example $E+1$ by the formula. In a deterministic environment, we can safely choose the largest of these values since we have observed both. In a non-deterministic environment, using the latter value might result in a slower convergence, because examples are assigned much too low Q values in the early stages of learning. These faulty values continues to effect the Q average until enough new values have been observed.

When a specified number of episodes is performed, all generated examples are stored in a file, which can be used for tree induction.

4.4 Inducing Decision trees

Having specified a problem and generated examples, we are now ready to construct an agent policy represented as a Q tree, which, as mentioned, is a logical decision tree. Most algorithms developed to induce decision trees use top-down induction and are variations of ID3 [26] and its successor C4.5 [28]. Briefly stated, ID3 builds a decision tree from a fixed set of examples. The resulting tree is used to classify future samples. An overview of ID3 is given in Table 4.2.

```
while a test which improves the information gain in a leaf exists
   Create decision node with best test and two leaves
   Split examples with the test
```

Table 4.2: The basic of the ID3 algorithm.

The algorithm uses a greedy search, that is, it picks the best test and never looks back to reconsider earlier choices.

C4.5 is an extension of the basic ID3 algorithm designed by Quinlan to address some issues not dealt with by ID3, e.g. avoiding overfitting the data, reduced error pruning and handling continuous attributes. When a decision tree is built based on some example set, normally referred to as training data, overfitting should be avoided. If the example set contained n examples the tree could do $n-1$ splits and classify each example correctly. New examples will however often be wrongly classified. *Pre-pruning* and *Post-pruning* are two techniques to avoid such overfitting. Pre-pruning is the task of not creating overly specific nodes, while Post-pruning is the task of removing the overly specific nodes after the tree is induced [16].

Using pre-pruning there are some stopping rules to prevent the tree from growing branches that do not improve the accuracy of the tree. Some rules are:

1. The number of observations in a node is lower than a certain threshold.
2. The gain from splitting a node using a test is too low.

In Post-pruning the tree is grown as far as possible. Superfluous subtrees are then replaced with leaves. One strategy for this is *Reduced Error Pruning* proposed by Quinlan [27]. Given a tree, each subtree is in turn pruned and compared with the unpruned tree. If the simplified tree yields improvement the subtree is kept pruned. The simplified tree is then further pruned and so on. Another strategy is *Error-Based Pruning* which is implemented in C4.5. It uses a bottom-up approach. The new thing in Error-Based Pruning is that it can replace a subtree in a tree with a branch of that subtree.

The toolbox induces Q trees using an ID3-like algorithm, but does not yet have support for pruning. Since Q trees are logical regression trees, the quality of a test is calculated using the variance of Q values. The variance in a node is calculated as the sum of squares of differences between the Q values and the mean as shown in Equation 4.2.

$$node_{variance} = \sum_{i=0}^{|E|} (q_{E_i} - q_{mean})^2 \tag{4.2}$$

where $|E|$ is the number of examples in the node and q_{mean} is the mean value of q in the node:

$$q_{mean} = \frac{\sum_{i=0}^{|E|} q_{E_i}}{|E|}$$

The quality of a test is the relative improvement of the summed variance of the two child nodes, that would be created using the test, and the variance of the parent node, as shown in Equation 4.3. There are however some tests with a quality of zero that could be useful in a decision node. These tests are explained in the next section.

$$Quality = \frac{variance(Childnode1) + variance(Childnode2)}{variance(parentnode)} \tag{4.3}$$

4.4.1 Refinements

When refining a conjunction of tests in inductive logic programming by adding new predicates to it, one of these predicates may not improve the clause much, although it would do so in conjuction with another predicate. In Blocks World, we can define a predicate `elevation(A,X)` that returns the elevation `X` of block `A` (i.e. the number of blocks that are below `A` plus one). Consider the following two imaginary predicates:

```
elevation(A,X)
X>=2
```

Neither of the two predicates above would be an improvement to the empty predicate since both `elevation(A,X)` and `X>=2` are meaningless without the other. Every block would succeed on the former test since all blocks have an elevation. The test actually just serves the purpose of instantiating `X` to the elevation of block `A`. The latter test does not make sense since without the former test, `X` would not be instantiated. However, an addition of a conjunction of the two tests yields a considerable improvement to the empty test since we are then generalising over all blocks that have an elevation of at least two:

```
elevation(A,X), X>=2
```

There are at least a few solutions to this problem. One solution, supported by the toolbox, is to add tests that always succeed to the root of the tree so that those tests are always included in the predicates of the corresponding Prolog program. If we add `elevation(A,X)` from the example above to the set of root tests, `X` will always be instantiated to the elevation of the block pointed to by `A`. Thereby only one test, namely `X>=2`, has to be added in order to test whether the elevation of block `A` is greater than or equal to two.

However, this solution has the disadvantage that if there are many such tests that can be applied with different parameters, the number of tests added to the root becomes very large. For instance, it may be desirable to have variables instantiated to the elevations of both `A`, `B`, `C`, and `D`.

Another solution is to let the learner *look ahead* in the refinements [7, 8], i.e. if `elevation(A,X)` is a refinement candidate to a conjuction of test predicates, then the learner may look ahead at which other refinements are available after `elevation(A,X)`. This way the combination of two tests will be tried without one of them being rejected first. This is computationally very heavy, though. The resulting number of tests to check t_c is

$$t_c = t_{nla} + \prod_{i=0}^{l} t_{la} - i$$

where t_{nla} is the number of tests that do not have look ahead enabled, t_{la} is the number of look ahead-enabled tests, and l is the level of look ahead. So if a two-level look ahead is implemented and there are 15 tests whereof 5 have look ahead enabled, $t_c = 10 + (5-0) \cdot (5-1) \cdot (5-2) = 70$.

There are several ways to incorporate look ahead. One way is to, for every refinement, look ahead at the next possible refinements. Another way is to incorporate two-step-refinements into one's refinement operator. This is described in more detail in [7].

4.4.2 Discretisation

While inducing a decision tree, one continuously searches through sets of test candidates to find the best tests. In the worst case scenario, this requires searching through all the possible candidates, which is very time consuming when the number of candidates is high. As an example, take the test `elevation(A,X)` proposed earlier. If the declarative bias allowed the variables to be constants it would result in 2500 tests in a world with 50 blocks (number of blocks times the number of possible elavations). However, it can be argued whether or not there is any significant difference between the tests `elevation(a,40)` and `elevation(a,45)`, testing if block `a` is elevated by 40 or 45 blocks. If such similar tests could be merged into one test, a lot of time would be saved.

Discretisation consists of converting a continuous domain into a discrete one or a large into a smaller [7]. This reduces the number of tests, which speeds up the process of inducing Q trees. As a side-effect, using discretisation also makes it less likely that a tree becomes overfit from its training examples. When discretising, it is important to find some reasonable thresholds to test upon, so the data will be distributed into useful sets. Otherwise, useful information might be lost and the agent will not be able to learn an optimal policy.

Fayyad and Irani [17] have developed an algorithm for discretisation based on entropy. It first finds a threshold that divides a set of examples into two subsets in such a way that their average class entropy is as small as possible. This is repeated recursively until some stopping criterion is reached. Some simple stopping criteria are to stop splitting when the data is coherent, i.e. ,its variance is below a given threshold or stop splitting when the number of examples in a node is below a given threshold. An example of discretisation on integers is given in Figure 4.5. Here the stopping criterion is that no new node with less than two elements may be created.

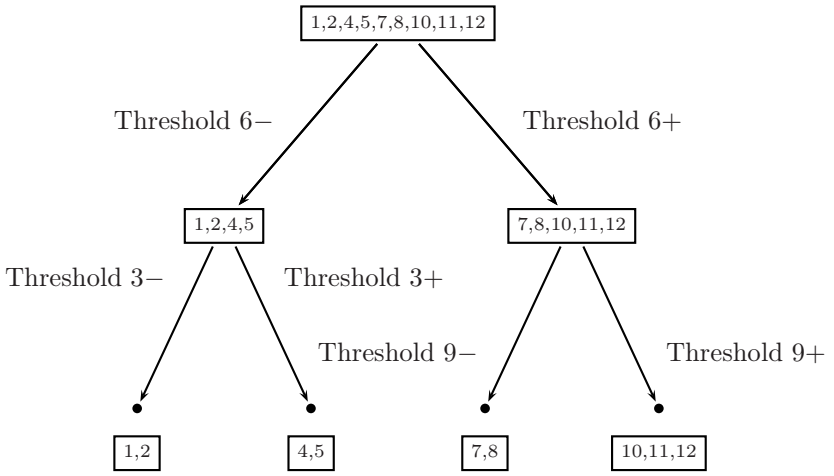


Figure 4.5: Discretisation on integers.

Often a certain discretisation is meaningful because of some theoretical assumptions or previous studies about a problem domain. These things cannot easily be inferred automatically. In such situations one might want to discretise the data manually to get a meaningful discretisation. This is the only kind of discretisation currently supported by the toolbox. An example of this is given in Section 5.3.3.

4.4.3 Incremental Tree Induction

There are three problems with non-incremental tree induction, as used by the toolbox, with respect to performance. First, it has to keep track of an increasing amount of examples when inducing the Q trees, making the tree building more and more time consuming. Secondly, when an example is encountered for the second time, its possible new classification value must replace the old one in the set of examples. Since the set of examples increases with the number of episodes performed, this process takes an increasingly amount of time. Last, the tree has to be rebuilt from scratch after each episode, and this also takes an increasing amount of time.

To avoid these problems, an incremental algorithm can be used. In reinforcement learning, the Q values learnt in initial episodes are usually far from correct. This means that an incremental algorithm would have to take this into account. Also, when old examples are not stored for later reference, the algorithm cannot directly use old information to generalise from.

The TG algorithm was introduced in [13] to solve these problems. TG stands for Tree Generalisation and it is a fully incremental induction algorithm and a first order extension of the G algorithm introduced in [25]. The pseudo code for TG is shown in Table 4.3.

```
Create an empty leaf
while data available
   Sort data down to a leaf
   Update statistics in the leaf
   if split needed then
      Create two empty leaves
```

Table 4.3: The TG algorithm.

Before a split can occur, the number of examples in a node has to exceed a predefined minimal example set size, and the information stored about the examples in the node should show that the used test is appropriate. The minimal example set size is used to avoid a split based on too few examples. The statistics in the leaf contains information about all the tests that could be used to split the tree further. Among this information is the number of examples on which the a test succeeds and fails, the sum of their Q values and the sum of their squared Q values, respectively. The Q value to predict is also stored in the statistics. This value is obtained from a parent node's statistics when a new leaf node is created, and it is further adjusted as examples are sorted down to the leaf.

By using the TG algorithm, trees are not rebuilt after each episode but are incremented. This means that the algorithm is not as computationally heavy as non-incremental algorithms. The TG algorithm only uses an example once, when the statistics for a node is updated. There is therefore no need to store and update the old examples, resulting in a huge save in space used. Because each leaf starts out with empty statistics, old (possibly noisy) Q value examples are deleted.

As mentioned in [11], a major drawback of the TG algorithm is that it cannot undo bad splits. One way to solve this is to try not to make bad splits. The chance of bad splits can be reduced by increasing the minimal example set size. This will however result in a slower learning rate as the tree grows, because the leaves are not as often assigned new examples. This can again be solved by using dynamic minimal sample sizes, which can be reduced for each level of the tree. Another solution is to store the statistics for all nodes in a tree, and not only for the leaves. It would then be possible to backtrack when a bad split is discovered, but this might result in using more space than a non-incremental algorithm.

The minimal sample set size and the need to compensate for bad splits increase the number of episodes needed for TG trees to converge to the right Q function. But because the algorithm is much faster than the non-incremental algorithm, the time needed to converge is, despite the larger number of examples, considerably smaller as can be seen from Table 4.4 which is a section of a table in [13].

4.5 Validating the Toolbox

To validate whether the toolbox produces correct results, we performed some experiments on a known environment, Blocks World with three blocks. The environment

		3 blocks	**4 blocks**	**5 blocks**
Original RRL	Stacking (30 episodes)	6.15 min	62.4 min	306 min
RRL-TG	Stacking (200 episodes)	19.2 sec	26.5 sec	39.3 sec

Table 4.4: Execution time of the RRL algorithm on Sun Ultra 5/270 machines.

is known in the sense that a fully converged Q tree for this environment is available. After letting our toolbox produce a Q tree that we expect to be converged, we can compare the two trees, and if they are equivalent, our tree must be fully converged as well.

The experiment was performed by letting the toolbox: First, explore the environment, second, exploit what it has learnt so far. These two steps (exploring and exploiting) were repeated a number of times, iteratively building a Q tree for every episode of exploring. In more detail, the two steps are the following:

- The first step was to let the agent learn one episode (from an initial state to the goal state) from the Blocks World environment with three blocks. The exploration constant was set to 1 and the Q tree was updated for every episode. The fact that the exploration constant was set to 1 means that all actions had the same probability of being chosen.
- The second step was to let the agent exploit the learnt information by performing a number of episodes without updating the Q tree. For each episode, the number of actions that were performed in order to reach the goal state was counted. The exploration constant was set to 0.0001 during this step. The reason that it was not set to 0 is that when we have only performed a few learning episodes and thereby have a poor Q tree, the optimal strategy might go in ring. I.e. the optimal action in a state a might lead to state b in which the optimal action might lead back to state a. Therefore we need to have a slight amount of exploration, i.e. randomness.

Common for the two steps was that we had random initial states and the goal was `goal(on(a,b))`. By performing the two steps repeatedly we expected to get a repeatedly better Q tree and thereby better and better results in the second step. In this case, "better results" refers to a lower number of actions from an initial state to the goal state as the agent is following an increasingly better strategy.

Since we have some randomness when exploiting the knowledge (in step two), we have to perform more than one episode and look at the average over a number of episodes. We performed 100 episodes and calculated the median and the arithmetic mean of the resulting action counts for each repetition of the two steps, and the two steps were repeated until the $\hat{Q}$ values in the tree had converged to the real Q values. To check this we used a converged Q tree that was presented in [15] by Džeroski et al. The results of the experiment are presented in Table 4.5 and illustrated in Figure 4.6.

Episodes	**0**	**1**	**2**	**3**	**5**	**6**	**8**	**25**
Examples	**0**	**2**	**8**	**10**	**18**	**21**	**28**	**29**
Median	9.5	8	4	3	2	2	2	2
Arith. mean	12.89	10.12	5.21	5.15	2.7	3.4	2.72	2.72

Table 4.5: The results of the experiment.

In Table 4.5, the columns each represent an iteration of the two steps in the experiment. "Episodes" refers to the number of learning episodes that were performed after that iteration. Note that for presentation purposes some iterations were left out from this table since they did not introduce new examples, although they did yield updated $\hat{Q}$ values. Thus no new examples were introduced from learning episode nine through twenty-four, when only one example was missing in order to have a complete example base for this environment. "Examples" is the number of examples that were produced so far. The median and the arithmetic mean were each calculated over 100 episodes of exploiting the learnt policy, and they express the number of actions that were performed until the goal state was reached.

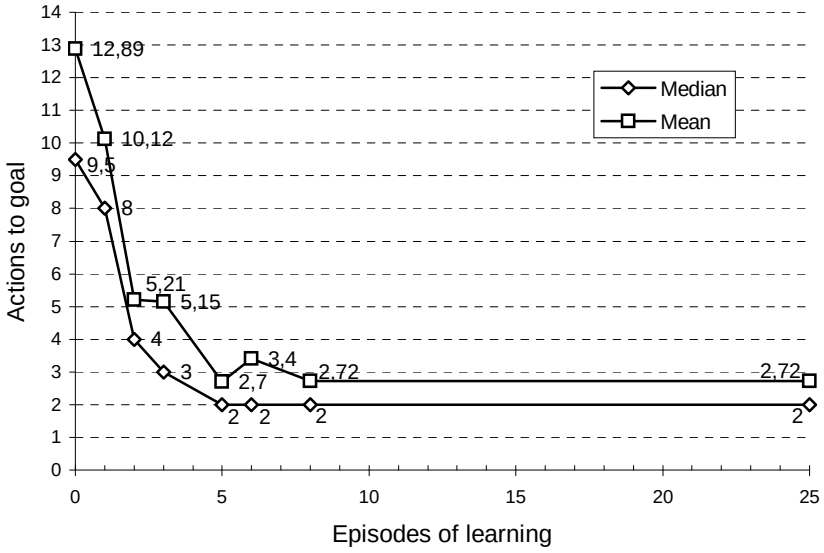


Figure 4.6: A graphical representation of the median and arithmetic mean over the number of actions in Table 4.5.

The median and mean values are graphically represented in Figure 4.6. Here we can see that from an average of 12.89 actions after 0 learning episodes (total randomness), the mean curve descends to just below 3 actions after only 8 learning episodes. Since we have a degree of randomness when exploiting the learnt knowledge, a few of these episodes are solved in a relatively high number of actions. This reflects in the deviations after especially 3 and 6 episodes. In order to disregard these peak values and thereby minimise the deviations in the curve caused by randomness, we have also calculated the median, which gives a curve that is much more smooth.

Actually the median values stabilise already after 5 episodes of learning, even though this is a point where we know that the $\hat{Q}$ values have not yet converged. This can be explained by the fact that at this point, most of the tree structures are identical to that of the converged tree, except that the $\hat{Q}$ values have not converged completely. In other words, after five learning episodes the Q tree is able to give the optimal strategy, though not from entirely correct $\hat{Q}$ values, i.e. the corresponding P tree would have converged at this point.

The resulting Q tree after 25 episodes of learning is presented in Figure 4.7 and Table 4.6. The tree is an exact match of the one in [15] down to the no-branch of the test `clear(D)`. However, this subtree differs only in the tests that were chosen; the struc-

ture and the values are the same. This is most likely caused by a different ordering in choosing a test during the building of our tree versus the tree of Džeroski et al.

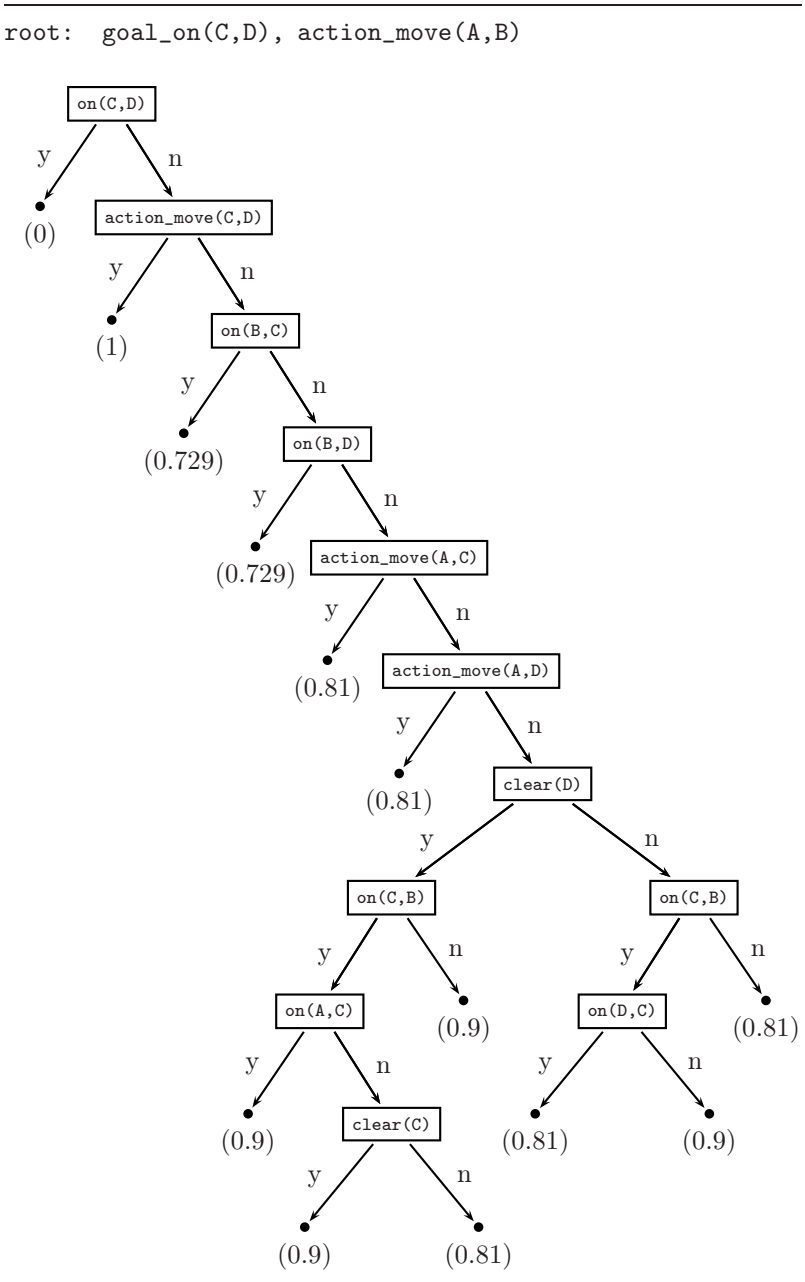


Figure 4.7: The resulting Q tree after 25 learning episodes.

```
q(0.000) :- goal_on(C,D), action_move(A,B), on(C,D), !.
q(1.000) :- goal_on(C,D), action_move(A,B), action_move(C,D), !.
q(0.729) :- goal_on(C,D), action_move(A,B), on(B,C), !.
q(0.729) :- goal_on(C,D), action_move(A,B), on(B,D), !.
q(0.810) :- goal_on(C,D), action_move(A,B), action_move(A,C), !.
q(0.810) :- goal_on(C,D), action_move(A,B), action_move(A,D), !.
q(0.900) :- goal_on(C,D), action_move(A,B), clear(D), on(C,B), on(A,C), !.
q(0.900) :- goal_on(C,D), action_move(A,B), clear(D), on(C,B), clear(C), !.
q(0.810) :- goal_on(C,D), action_move(A,B), clear(D), on(C,B), !.
q(0.900) :- goal_on(C,D), action_move(A,B), clear(D), !.
q(0.810) :- goal_on(C,D), action_move(A,B), on(C,B), on(D,C), !.
q(0.900) :- goal_on(C,D), action_move(A,B), on(C,B), !.
q(0.810) :- goal_on(C,D), action_move(A,B), !.
```

Table 4.6: The Prolog program equivalent to the Q tree in Figure 4.7.

4.6 Summary

To conduct experiments using relational reinforcement learning, we have constructed a toolbox, in which a problem can be specified. A problem specification can be entered through a GUI, or directly specified using the toolbox file structure. So far, only deterministic environments are supported. The output of the toolbox is a Q tree, which can be used as a policy.

When constructing a general toolbox, several potential problems must be considered. The discretisation of continuous variables to minimise the number of decision tree tests has been described. So has the refinement of tests, where a potential problem is tests that succeed for all examples. This can be solved by placing such tests in an extra decision tree root, or by performing lookahead.

The toolbox was tested using Blocks World with the goal `on(a,b)`. An agent was trained to reach this goal in as few steps as possible. After 25 episodes, the Q tree of the agent had converged to an optimal policy.

In the next chapter, the toolbox is extended to support non-deterministic multi-agent environments.

Chapter 5

Extending the RRL Toolbox

The previous chapter introduced the RRL toolbox, which so far only handles deterministic single agent environments such as Blocks World. In this chapter, we will extend the toolbox to also handle non-deterministic environments with both cooperating and competing agents. We will however setup certain restrictions on this type of environments, such that the toolbox can supply a general interface.

The example environment used in this chapter is a simplification of the popular board game Risk [18] called *Simple Risk*. The basics of Simple Risk is described in Section 5.1. The rest of this chapter uses Simple Risk to discuss the implementation of the necessary features that the toolbox must support to handle such environments. The effort to keep these features general is also discussed.

5.1 Simple Risk

Simple Risk is a turn based board game, played by multiple competing *teams*. A Simple Risk board consists of a number of *countries*, where each is allied with exactly one team at any given time. The goal of a team is to possess all countries on the board. Figure 5.1 shows a Simple Risk board based on Africa. This board is divided into six regions. Each region, henceforth referred to as countries, is assigned to a team, e.g. Madagascar and Egypt are allied under the grey team, while South Africa belongs to the white team. Each country has a number of other countries as its neighbours. For instance, Madagascar is a neighbour of South Africa, while Egypt is a neighbour of neither. Notice that all countries are connected, either directly as neighbours, or indirectly through other countries. Countries alligned with different teams are enemies.

Each country has a number of armies to its disposal (a country must have at least one army at all times). It can use these armies to declare war on neighbouring enemies, and thereafter perform a number of attacks. When declaring war, some of a country's armies are moved into war. These armies are now up against the armies of the defending enemy. During a war, the attacking country can choose to perform an attack or retreat (at least one attack is required). Performing an attack results in a dice throw, where each country is assigned a number of dice according to their number of armies dedicated to the war. The number of dice is defined as

$$\text{attacker dice count} = \begin{cases} \textbf{1} \text{ if } \texttt{attacking armies} = 1 \\ \textbf{2} \text{ if } \texttt{attacking armies} = 2 \\ \textbf{3} \text{ if } \texttt{attacking armies} \geq 3 \end{cases}$$

$$\text{defender dice count} = \begin{array}{l} \mathbf{1} \text{ if } \texttt{defending armies} = 1 \\ \mathbf{2} \text{ if } \texttt{defending armies} \geq 2 \end{array}$$

When rolling the dice, the highest die of the attacker is compared to the highest die of the defender, and if both countries have at least two dice, the second highest die of each are compared as well. For each comparison, the team with the lowest die loses an army. If either the number of attacking armies or defending armies are reduced to zero, then the war is over. If the attacker wins, the defending enemy is conquered and it changes alliance to the team of the attacking country. The remaining attacking armies are transferred to the defending enemy. If the defender wins, the attacker has simly lost all its armies dedicated to the war. If an attacker at some point chooses to retreat, its attacking armies are withdrawn. However, retreating carries a penalty of the loss of one army.

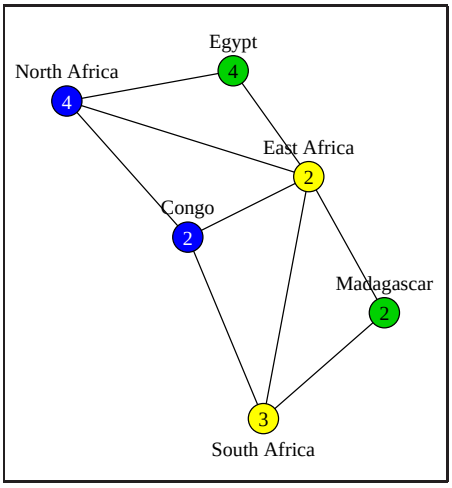
Figure 5.1 shows a Simple Risk board in four different configurations. Together these configurations show a scenario where North Africa tries to conquer East Africa. The initial state of the board, before war has been declared, is illustrated in 5.1(a). In 5.1(b), North Africa has declared war on East Africa with three attacking armies. East Africa has two armies, which will both be used to oppose the attack. Figure 5.1(c) shows the scenario where the highest dice roll comparison of the battle results in a draw, reducing the attacking armies to 2. The second highest comparison favours North Africa, thereby reducing the defending armies to 1 as seen in 5.1(c). North Africa decides to attack once again, and a new battle is initiated. Figure 5.1(d) shows the scenario where East Africa lost the battle, which means that the defending armies is reduced to 0, and East Africa no longer has any armies. Because of this, North Africa transfers alls its attacking armies unto East Africa, thus making this the new army count of East Africa. When a country has been conquered it changes alliance, in this case East Africa changes from white to black. The result of the conquest is shown in 5.1(d).

If all countries on a team have decided that it is no longer feasible to attack, the countries now get an opportunity to move armies to neighbouring allied countries. The precondition for moving an army to another allied country is that the country must be a neighbour, and that the country moving armies must always leave at least one army behind. In Simple Risk, there must always be at least one army on all countries at any given time. After a country has started moving armies, declaring war and attacking is disallowed.

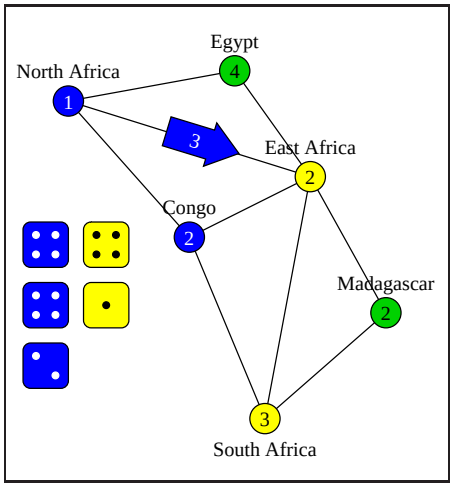
At some point, all countries on a team have decided that there is no more to gain by moving armies. When this happens, the initiative is passed on to the next team. This process is explained in greater detail in Section 5.4. Every time a new team gets the initiative, all of its countries are reinforced with one new army.

5.1.1 Relational Properties of Simple Risk

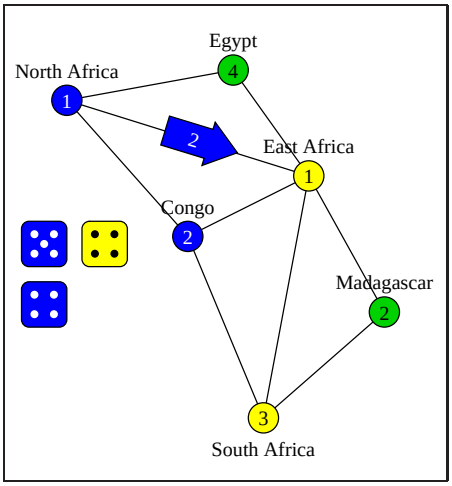
Given the environment of Simple Risk, and the task of conquering the world, we must search for relational properties that will allow an agent to learn this task, but at the same time avoid a combinatorial explosion of states. In effect, we must find a small part of the game, and show that learning only this part will allow an agent to play Simple Risk with a reasonable skill. There are two obvious possibilities, namely playing the game from the perspective of a team, or playing the game from the perspective of a country.



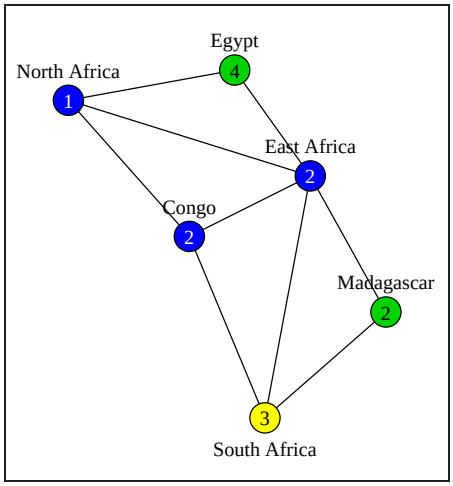
(a) Initial state.



(b) North Africa attacks East Africa with three armies.



(c) Both the attacker and defender have lost an army.

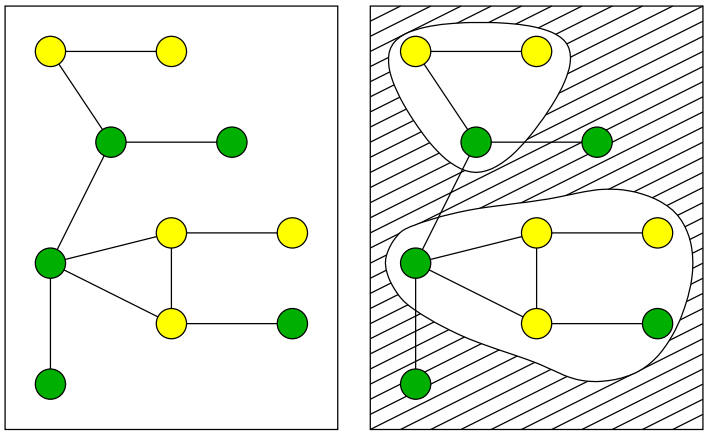


(d) North Africa successfully invades East Africa.

Figure 5.1: A Simple Risk scenario.

Teams

A human player controlling a team in Simple Risk has the possibility of observing the entire board and afterwards choosing actions accordingly. Giving our agent the same possibility means that each agent state must represent the entire board. This is shown in Figure 5.2(a).



(a) A state represents an entire board.

(b) A state represents clusters of the board.

Figure 5.2: Graphically representation of team agent states.

Since even small boards have many possible configurations, such a solution will yield a rapid state explosion as a board is extended by adding new countries. The most dominant factor in this explosion is the number of combinations in which the countries can be connected. The number of combinations grows exponentially as illustrated in Figure 5.3.

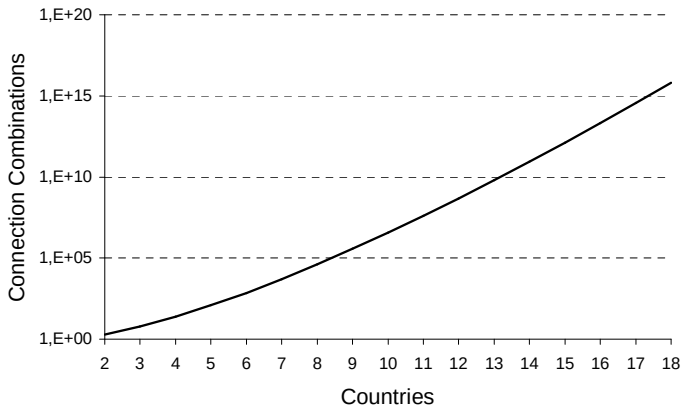


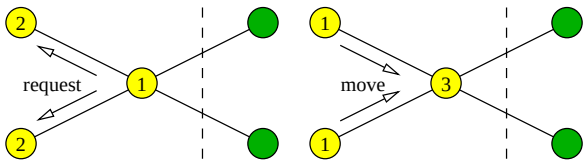
Figure 5.3: Combinations of connected countries (logarithmic scale).

Another more relational solution, while still approaching the game from the perspective of a team, is to restrict the knowledge of a team agent to its own countries and their immediate neighbours. That is, an agent will see a board in pieces determined by its own countries, and not have a full overview of the entire board. This is illustrated in Figure 5.2(b), where the shaded area is the part of the board of which the agent has no knowledge. A policy learnt using such a solution could subsequently be used by any team to play Simple Risk. In the worst case, this solution also suffers from the mentioned state explosion, although to a lesser degree. This is because the clusters most likely do not cover the entire board before an agent is about to win.

Countries

We now narrow the perspective and observe the game from the viewpoint of a country, which only has knowledge of itself and its immediate neighbours. A country can potentially be a neighbour to all other existing countries. Notice that this does not create a state explosion in the same scale as before, because the number of combinations, in which the country can be connected to other countries, only increases by one, when a new country is added to the board.

Obviously, this very limited view of the entire board makes it hard to play Simple Risk with a reasonable skill. Each country is an autonomous agent, and acts in what it thinks is its own best interest. To handle this problem, the state space can be extended by introducing predicates that act as messages—thereby allowing agents to communicate by message parsing. One of the main challenges in Simple Risk is to move troops to the front, since they are of less use in countries without enemies. To enable an agent to learn this behaviour, the predicates `front` and `requestarmy` can be used. For each allied neighbour `C` located on the front, an agent has the fact `front(C)` in its state. If a country needs reinforcements (e.g. if it is located at the front), it can perform the action `requestarmy`, which will append the predicate `requestarmy(C)` to the state of all allied neighbours. The neighbours can then choose to act on this request or not. The idea of moving troops to the front is illustrated in Figure 5.4.



(a) Requesting armies from allied neighbours.

(b) Responding to the request.

Figure 5.4: Moving troops to the front.

Message parsing adds to the overall state count, and for very small boards this solution has a larger state space than for team based agents. This quickly changes as more countries are added though. Section 5.2.2 describes the size of the state space using this solution, both for a traditional Q learner and a relational Q learner. A policy learnt for one country can be used by all other contries, although the number of neighbours should be varied during training.

In the rest of the report, we assume that agents are countries. We made this choice

both because of the smaller state space, but also because of the challenge of getting the agents to cooperate in an RRL setting.

5.2 State Description

A Simple Risk state is a number of predicates, each describing a fact that holds true for the agent, or a property of the surroundings that the agent is currently in. As in the deterministic setting described in Section 3.1, a state is simply a Prolog list of facts each separated by a comma. For instance, the following is part of a state for the agent `northafrica` with an army count of `2`.

```
[a(northafrica), armies(2), ...]
```

A complete description of a team is a list of the agent states aligned with that team:

```
[...],[...],[...]
```

A complete board description is a list of all agent states:

```
[ [...],[...],[...]  ]
```

5.2.1 A State in Simple Risk

An agent state in Simple Risk consists of facts describing the agent itself, as well as facts describing its neighbours. The facts that always appear in a state are

- the name of the agent `a(name)`,
- the name of the agent's team `t(name)`,
- the army size of the agent `armies(armysize)`,
- the name and army size of neighbouring enemies `enemy(name,armysize)`, and
- the name and army size of neighbouring allies `ally(name,armysize)`.

For each enemy and ally that a country has, a fact is added, representing the name and army size of that particular neighbour. A state description of Egypt, as seen in Figure 5.1(a), is represented as

```
[a(egypt), t(grey), armies(4), enemy(northafrica,4),
                  enemies(eastafrica,2)]
```

To avoid very long games, the maximum number of armies allowed on a country has been set to 15. As mentioned in Section 5.1.1, agents must use message passing to cooperate. The only action that an agent can manually use to send a message is `requestarmy`, which lets allied neighbours know that the agent needs reinforcements. This is achieved by adding the fact `requestarmy(C)` to the state of allied neighbours, where `C` is the country requesting reinforcements.

Furthermore, the message of letting agents know which allied neighbours are on the front is automatically updated as part of the transition function. Every time a country

changes alliance, the front lines are updated, allowing the agents to observe this using the `front(C)` predicate, where `C` is an allied neighbour on the front.

When a country goes to war, a `war(C,A)` predicate is added to its state, where `C` is the defending country, and `A` is the number of attacking armies. The presence of this predicate is part of the precondition for performing the actions `attack` and `retreat`. It is removed when either the defending country has been conquered, or the attacking armies have retreated or been beaten.

If a country is conquered, the predicate `invaded` is added to its state. When the country, now in alliance with a new team, performs its next action, the predicate is removed. To be invaded is an negative goal state for an agent, and as such, no further rewards can be expected. This means that all actions performed from such a state must yield a Q value of zero. The `invaded` predicate thereby gives the tree induction algorithm the possibility of grouping all `invaded` state/action pairs in one leaf.

In Simple Risk, offensive actions are always performed before defensive actions in a round (i.e. attack before move). An agent cannot move armies around before all allied agents have finished attacking. This is controlled by the introduction of *modes*, which are further described in Section 5.3.1.

As an example of a complete state of an agent in Simple Risk, the following represents the state of North Africa in the scenario illustrated in Figure 5.1(c), where North Africa is at war with East Africa:

```
[a(northafrica),t(black),armies(1),enemy(egypt,4),enemy(eastafrica,1),
ally(congo,2),front(congo),mode(war),war(eastafrica,2)]
```

5.2.2 Size of the State Space

As described in Section 5.1.1, the size of the state space increases with the number of countries on the board. But since, in practice, a country has at most seven or eight neighbours (normally), countries added to the board beyond this number will not add more states. Another dominant factor is predicates which use the continuous value of army count. For instance, a predicate like `armies(A)`, where $0 < \texttt{A} \leq 15$, increases the state space by a factor of 15. The total state space of Simple Risk (without generalisations) can be calculated as the product of all the combinations of state facts and their parameters. Figure 5.5 shows how this number increases with the number of countries on the board.

To get an idea of the size of the state space for a relational Q learner using generalisations, we start by discretising the army count parameter from 15 to 5. These five new values indicate whether an army count is one, two, three, between four and 10 (including), or above ten. This discretising is done when inducing Q trees as described in Section 4.4.2. Secondly, observe that, given the goal of having no enemies, all enemies are interchangeable—meaning that it is the same to conquer either enemy `A` or enemy `B` (the same principle can be applied to allies). Likewise, an agent can disregard the names of its neighbours. Generalisations like this are achieved by e.g. constructing a Q tree rule with the body

```
ally(C0,A0), armyrequest(C0), action(movearmy(C0)), !.
```

where `C0` is an allied country, and `A0` is its army count. This generalises over all states where any ally requesting armies should receive reinforcements. It is difficult to make a precise calculation of the actual reduction in states. However in Chapter 6, we create

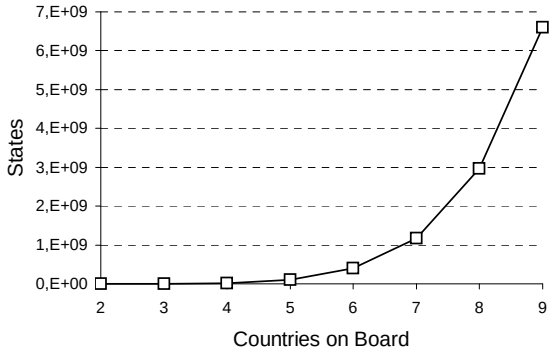


Figure 5.5: Size of state space using traditional Q learning.

an agent with a reasonable skill at playing Simple Risk, and this agent is described using only eleven Q tree rules. This strongly suggests a very large reduction of the state space.

5.3 Problem Specification

As for deterministic single-agent environments, the toolbox must accept a problem specification and return an agent policy. The toolbox does not yet supply a graphical user interface for multi-agent environments, so the specification is made directly to a Prolog knowledge base as well as a declarative bias and initial state file. These files, as well as other properties, are tied together using a configuration file. Most of the sub-specifications explained in Section 4.2 are unchanged, while a new specification of so-called `modes` is needed. The specification of transitions is also slightly changed.

A complete specification of Simple Risk can be found in Appendix C.

5.3.1 Modes

By defining agents as countries, we are relying on the agents to learn to cooperate and act as a team. Given a correct problem specification, it seems likely that this can be achieved. Unfortunately, since the agents do not know the meaning of the various messages initially present in their states, the task of learning cooperation can take a very long time.

To minimise this problem, the toolbox uses modes, which is a way of restricting the actions available to an agent. At any given time, a team is in a specific mode. Since there are potentially several agents on a team, this means that all agents on that team is in that specific mode. To progress to the next mode, all agents in a team must perform the action `pass`. The mode of an agent is represented with the predicate `mode` in the agent's state. Simple Risk has two modes, war-mode and move-mode, and they restrict the possible actions so that an agent can only declare war and attack while in war-mode, and only move armies and request for backup while in move-mode. This makes the agents, which are learning to cooperate, more likely to combine related actions, thereby getting more experience using such combinations.

Modes also open up for the possibility of allowing state transitions whenever a mode ends. This is achieved through a specific callback predicate that must be specified. Every time a mode ends, the toolbox calls the predicate `endmode` with the interface

```
endmode(States, Mode, Team, NextTeam, NextStates)
```

where `States` is the states of existing agents before the mode `Mode` ends, and `NextStates` is the states after the next mode is initialised. To allow transitions based on the team ending mode, as well as on the team which gets the turn next, these are also supplied as parameters. For instance, in Simple Risk, each country on a team is reinforced with one army when the team gets its turn. The `endmode` predicate for ending move-mode is therefore implemented as

```
endmode(States, move, TeamId, NextTeamId, NextStates) :-
   deletemode(States, mode(move), States1),
   reinforce(States1, NextTeamId, States2),
   beginturn(States2, NextTeamId, NextStates).
```

Move-mode is the last mode of a team, before the turn is passed on to the next team. The `endmode` predicate shown therefore first deletes `mode(move)` from all agents, and ends by forwarding the turn by calling `beginturn` with `NextTeamId`. In between, `reinforce` is called to reinforce the next team with one army.

5.3.2 Transitions

The action precondition function is no longer sufficiently defined using only the state of one agent. Instead, it must take the states of all existing agents into account. Furthermore, the precondition of an action may depends on which team an agent belongs to. The new Prolog interface of the precondition predicate is defined as

```
pre(States, Id, TeamId, Action)
```

where `States` is the states of the agents in the environment and `Id` is the name of the active agent. `TeamId` is the team of the active agent, and `Action` is the action which is tested.

Similarly, the `delta` function is redefined as the Prolog predicate

```
delta(States, Id, TeamId, Action, NextStates)
```

where `NextStates` is the updated agent states. Notice that using Simple Risk with both competing and cooperating agents as an example, we have defined that each agent in the environment belongs to a specific team. This definition can easily be applied to environments where only competing agents exist—namely by only having one agent per team.

The actions of Simple Risk are divided between war-mode and move-mode. In war-mode the existing actions are

- `declarewar(C,A)`, where `C` is the country declared war upon, and `A` is the armies delegated to the war,

- `retreat`, which performs the retreat of all armies at war, and
- `attack`, which makes the armies at war perform an attack on the defending country.

In move-mode they are

- `movearmy(C)`, which moves one army to the allied country `C`, and
- `requestarmy`, which requests reinforcements from all allied neighbouring countries.

As an example, the precondition of the action `declarewar` is that the active agent is in war-mode and that the number of attacking armies is between 1 and 14. Furthermore, no other wars can be in progress. Of course, the agent declared war against must in fact be an enemy, and finally, the number of attacking armies must be less than the number of armies present in the active agent's country. The precondition predicate for `declarewar` is implemented as

```
pre(States, Id, TeamId, declarewar(C, A)) :-
   member(A,[1,2,3,4,5,6,7,8,9,10,11,12,13,14]),
   not(member(war(_,_), States)),
   state(Id, States, S),
   member(mode(war), S),
   member(enemy(C,_), S),
   member(armies(AHome), S), AHome>A.
```

where the `state` predicate uses the performing agent's `Id` to find its state `S` in `States`. As described in the previous chapter, the toolbox can use such `pre` implementations to query for the actions that are available to an agent. For instance, the following query

```
States=
   [[a(egypt),t(black),armies(2),enemy(congo,1),mode(war)],
   [a(congo),t(grey),armies(1),enemy(egypt,2),]],

findall(Action, pre(States, egypt, green, Action), Bag)
```

would return `Bag=[declarewar(congo,1), pass]`. The actual transition made by executing a `declarewar` action should update the performing agent's state with a `war` fact, and move the specified armies into war. The `delta` predicate for `declarewar` is implemented as

```
delta(States, Id, TeamId, declarewar(C,A), NextStates) :-
   pre(States, Id, TeamId, declarewar(C,A)),
   state(Id, States, S),
   delete(S, armies(AHome), S1),
   NewAHome is AHome - A,
   append(S1, [war(C,A), armies(NewAHome)], S2),
   delete(S2, invaded(_), S3),
   replace(S, S3, States, States1),
   updateArmiesInNeighborStates(Id, NewAHome, States1, States2),
   delta(States2,Id,TeamId,attack,NextStates).
```

This predicate first uses `pre` to make sure that the action is legal. It then finds the state of the active agent, and updates the armies attached to its country by first deleting the previous armies from the agent's state, calculating a new army count, and then appending this new count back into the state. If a country has previously been invaded, the `invaded` predicate is removed from its state, because the agent is now performing actions on behalf of its new team. Since the number of armies in the active agent's country has changed, an update is performed on the states of the country's neighbours. Finally, the implicit `attack` action performed after declaring war is executed.

The action `attack` is non-deterministic due to the roll of the dice. This kind of non-determinism is implemented by the use of the Prolog predicate `random` as illustrated by the `rolldie` predicate

```
rolldie(Die) :- Die is random(6)+1.
```

where `Die` represents the die being rolled. Another kind of non-determinism is introduced by the presence of other agents, and is described further in Section 5.4.

Besides the transitions existing in the actual environment, a `pass` action must also be specified. The toolbox uses this action to pass control between the existing teams and agents. Normally, the `pre` and `delta` predicates describing this action can be left empty.

5.3.3 Discretisation

As explained in Section 4.4.2, the toolbox only supports manual discretisation of continuous values. In Simple Risk, the number of armies present in various facts causes unnessesary many states and tests. The allowed number of armies on a country is 1 to 15, but, in essence, we can split this into 5 intervals: 1, 2, 3, $4-10$ and >10. This is achieved by defining the predicate `armyconvert`:

```
armyconvert(A,1) :- A=1.
armyconvert(A,2) :- A=2.
armyconvert(A,3) :- A=3.
armyconvert(A,4) :- A>=4, A=<10.
armyconvert(A,5) :- A>10.
```

where `A` is the actual number of armies, and the following parameter is the identification of the interval in which the number belongs. This predicate can then be used in various tests, such as testing if one army count is greater than another:

```
agt(X,Y) :- armyconvert(X,X1), armyconvert(Y,Y1), X1>Y1.
```

5.3.4 Configuration

Since there currently is no GUI available for multi-agent environments, the glue that ties the complete problem specification together is a configuration file. This file contains properties such as

- the Prolog executable,

- the agent and team predicate names,
- the name of the *example generating* agent,
- the number of modes and their names,
- mode iteration restrictions,
- the exploration policy (and exploration constant if needed) and the attached Q tree for each agent,
- the discount factor,
- the example file, and
- the tree induction executable.

The agent and team predicate names can be specified, so that they do not necessarily have to be `t` and `a`. The name of the example generating agent tells the toolbox which agent that generates examples (see Section 5.4). Modes were described in Section 5.3.1, and they are specified in the configuration file with their number and names. To make sure that a team does not continue forever without ever passing control, a restriction on the number of iterations for each mode can be set. For instance, in Simple Risk one can specify that each agent on a team is only given the initiative 4 times while in war-mode. If this number if reached, then the toolbox forces a mode change.

The exploration policy of each agent can be specified to be either random, greedy or using the Q exploration strategy explained in Section 2.4. If the latter is chosen, the exploration constant must also be set. The discount factor denotes the extent to which the rewards of previous actions are discounted, and the example file is the file in which the toolbox stores examples. Finally, the path to the tree induction executable program must be specified, so the toolbox can generate Q trees over generated examples.

5.4 Example Generation and Tree Induction

Even though the complete state of a multi-agent environment consists of several agent states, we are still interested in letting an agent learn to act solely on the contents of its own state. This means that an example is still a single state combined with an action, a goal, and a Q value. When the environment consists of several agents, they are all potentially generating examples for the Q tree. However, to keep things simple, we have chosen to have only one example generating agent in each episode. Table 5.1 shows the algorithm used by the toolbox to progress through the state space of a multi-agent environment[1].

Episodes are started by choosing a specified initial state and a random starting team. The mode of the starting team is set to zero (the names of the various modes are not important to the toolbox, only their orderings). Until a goal state is reached, each agent belonging to the active team can choose to perform an action or pass control. If all agents on a team pass control, then the team progresses to the next mode. When leaving the last mode, control is passed to the next team. Figure 5.6 illustrates the layout of an episode graphically.

When an agent performs an action, the resulting state is defined by the `delta` function. But when generating episodes as described, an agent almost never gets to perform a new action on the resulting state of its previous action—simply because this state has been altered by other agents in between. In effect, even otherwise deterministic

[1] The algorithm is represented in pseudo-code giving previously described logical predicates an imperative look to increase readability.

```
States ← generateStartState()
TeamId ← chooseRandomTeam()
Mode ← 0
While not goal(States) do
   For Id = Agent_first To Agent_last on team TeamId
      Actions ← findall(Action, pre(States, Id, TeamId, Action))
      A ← chooseActionUsingExplorationPolicy(Actions)
      States ← delta(States, Id, TeamId, A)
   If all agents passed then
      If Mode is last mode then
         States ← endmode(States, Mode, TeamId, NextTeamId)
         TeamId ← NextTeamId
         Mode ← 0
      Else
         Mode ← Mode + 1
```

Table 5.1: Using `pre` and `delta` to progress through the state space of a multi-agent environment.

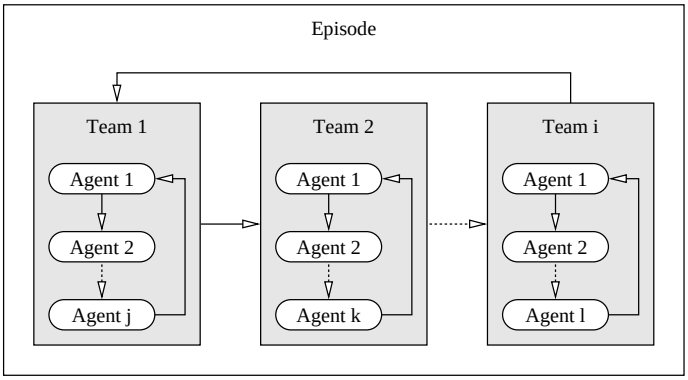


Figure 5.6: The three layers of an episode. i is the number of teams, and j, k and l are the numbers of agents of the respective teams.

actions become non-deterministic because of the multi-agent environment. The idea is that this non-determinism will lead to the agent learning the *actual* consequence of its actions over time, instead of just observing the immediate transition.

Given the identically constructed examples, the induction of Q trees is conducted in a fashion very similar to that of single-agent environments. However, one new heuristic is introduced, namely the possibility to specify the minimum number of examples that must be present in a node to generate another split. For instance, by setting this value to 50, the induction algorithm will not try to split nodes with less than 50 examples. This heuristic both speeds up the actual tree induction, but also avoids over-fitting trees to some extent. Moreoever, it helps in not making trees too large. Very large trees affect the performance of learning greatly, and the underlying Prolog engine quickly becomes a bottleneck because of the the large amount of tests that must be performed.

5.5 Summary

This chapter has described the changes made to the RRL toolbox in order to support non-deterministic multi-agent environments. These changes are primarily the introduction of modes and the redefinition of the `pre` and `delta` predicates.

The real difference between non-deterministic multi-agent environments and deterministic single-agent environments becomes very clear in the next chapter, where we experiment with various setups to let an agent learn how to play Simple Risk.

Chapter 6

Experiments

In the previous chapter, a general approach for non-deterministic multi-agent environments was introduced. In this chapter, this approach was tested by performing various experiments using Simple Risk. The correctness of the Simple Risk environment was tested by the implementation of a hand-made reasonable agent. Section 6.1 describes this agent. From Section 6.2 and forward, the setups and results of the conducted experiments are described.

6.1 A Reasonable Agent

In order to test whether the learnt data in our Q tree is useful, we have constructed a Q tree for a "reasonable" agent against which we can test the Q tree agent and see which one wins the most games. It would have been best to construct an optimal agent, of course, but it seems quite impossible to manually construct rules for every state in the state space.

Instead we have set some rules, the effect of which we would expect a learning agent of at least some experience to have learnt. These rules are presented in Table 6.1 and each rule contains the root, an action, and some conditions that must hold in order for that action to be the potentionally most reasonable action. Each rule `q(`q`, 1, E) :-` r`,` a`,` c`.` should be read as: "In the example `E`, action a is potentionally reasonable if conditions c are satisfied given the root r". The root is `goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E)`.

Actually a P tree which classifies the actions as reasonable or unreasonable would have been enough, but since our toolbox does not support P trees, the reasonable agent is represented as a Q tree. After all, everything that can be read from a P tree can also be read from its corresponding Q tree. So when the actions are tested one by one, the action(s) yielding the highest Q values are considered reasonable—the rest are considered unreasonable. Note that the cut operators do not mean that only one action can come into consideration of being reasonable since the tree is consulted several times; once per legal action.

Hence the first rule in Table 6.1 says that it is potentionally reasonable[1] to declare war on a country `C0` with an attack count of `A0`, if `C0` is an enemy and `A0` is greater than the army count `A1` of that `C0`, and `A0` is the largest possible attack army. Or in

[1] Only potentionally reasonable since other actions might have a higher Q value.

```
q(0.1, 1, E) :- (1)
   goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E),
   action(declarewar(C0,A0,E)), enemy(C0,A1,E), max_attack(A0,E),
   fgt(A0,A1,E), !
.
q(-0.1, 1, E) :- (2)
   goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E),
   action(movearmy(C0,E)), enemy(C1,A0,E), !
.
q(0.2, 1, E) :- (3)
   goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E),
   action(movearmy(C0,E)), equal(I0,I2,E), armyrequest(C0,E), !
.
q(0.1, 1, E) :- (4)
   goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E),
   action(movearmy(C0,E)), equal(I0,I2,E), !
.
q(0.09, 1, E) :- (5)
   goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E),
   action(requestarmy(E)), gt(I2,I0,E), !
.
q(0.09, 1, E) :- (6)
   goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E),
   action(requestarmy(E)), armyrequest(C0,E), !
.
q(0.1, 1, E) :- (7)
   goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E),
   action(attack(E)), war(C0,A0,E), enemy(C0,A1,E), fgt(A0,A1,E), !
.
q(0.1, 1, E) :- (8)
   goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E),
   action(attack(E)), war(C0,A0,E), enemy(C0,A1,E), fequal(A0,A1,E), !
.
q(0.05, 1, E) :- (9)
   goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E),
   action(retreat(E)), war(C0,A0,E), enemy(C0,A1,E), fgt(A1,A0,E), !
.
q(0.01, 1, E) :- (10)
   goal(enemies(I0,E)), neighbours(I1,E), enemies(I2,E), allies(I3,E),
   action(pass(E)), !
.
q(0.0, 1, E).
```

Table 6.1: The Prolog tree for a reasonable agent.

plain English: It is potentionally reasonable to attack an enemy country with one's full army if one's army count exceeds that of the enemy country.

The second rule says it is potentionally *un*reasonable (note the negative Q value) to move an army to a country `C0` if the agent has an enemy neighbour `C1`. The third rule says that it is potentionally reasonable to move an army to a country `C0` if the number of enemy countries of the goal state `I0` is equal to the number of actual enemy countries `I2` (most likely none) and the agent has an army request from `C0`. The fourth rule says the same as the third one does, except that the agent does not need to have an army request. This rule has a lower Q value and is thereby lower prioritised, though.

The fifth rule says that it is potentionally reasonable to request an army if `I2` is greater than `I0`, i.e. one or more of the agent's neighbours are enemies (given `I0` is zero). The sixth rule says that it is potentionally reasonable to request an army if the agent has an army request from a neighbour.

According to the seventh and eighth rules it is potentionally reasonable to attack[2] if war was declared on a country `C0` with an attack army count of `A0` and `C0` is an enemy with an army count of `A1` and `A0` is greater than (rule seven) or equal to (rule eight) `A1`.

Rule nine says that it is potentionally reasonable to retreat if the same conditions as in rules seven and eight are met, except that the enemy's army count `A1` is greater than the attack army count of the agent `A0`.

The tenth rule (pass) has no conditions since pass is always a potentionally reasonable action (when it is a legal action, that is). However, its Q value is very low so it will only be considered actually reasonable if no other actions are potentially reasonable.

Of course, since we set up static rules for the reasonable agent to follow, it will not exactly be a hard-to-beat opponent for a human player. It will be easy to figure out that the agent *always* attacks if it has the superior power and otherwise is reluctant; it never takes any chances. However, we believe that it is useful to test a learning agent against it since it produces a stable performance.

6.2 General Setup

For all the conducted experiments, we first used the toolbox to generate policies over an increasing number of episodes. During this activity, the learning agent was a single country, which was part of a team playing against various other agents. An episode was ended if this country was invaded, or if the team of the country conquered the entire board. To speed up the experiments, an episode was also ended if the teams reached a predefined maximum number of turns. This number was set to 30 while training. Since Simple Risk is about conquering countries, and since there is no difference between conquering a country at the beginning of a game or at the end of a game, no knowledge was lost. When an episode ended, the learning agent received a reward proportional to the remaining number of countries belonging to its team. The discount factor was set to 0.9, while the exploration constant varied across the different experiments.

The Q trees induced by the toolbox used the following heuristics: A node was only considered for a split if its variance divided by the number of examples in the node was below 0.002. This number was reached by continuously observing the induced

[2]Recall that it is only possible to attack the country on which war has been declared, and it is only possible to have declared war on one country at a time. Therefore it is not necessary to specify which country to attack.

trees. Furthermore, a node was not split if its number of examples was below 2% of the total amount of examples in the Q tree.

After training, each generated policy was tested by playing against enemy teams with agents choosing their actions randomly. Actions in the tested policy was chosen using a greedy strategy. The maximum number of rounds was increased to 60 to give somewhat defensive policies a chance to finish their games. To allow the consideration of unfinished games, a score represented as the number of remaining countries was used. Each policy was tested in multiple games to ensure the validity of the results, and thereafter compared to the results of the hand-made reasonable policy, as well as a totally random policy.

Rewards

The reward given for conquering the entire board was set to the highest possible positive reward, `1`. The penalty for being invaded was set to `-1`. If an episode ended without reaching a goal state, the given reward was a value greater than `-1` and less than `1`. The value of the reward was, as mentioned, based on the number of countries in the possession of the learning agent's team.

6.3 Experiment One: Africa

The board setup for this experiment was as illustrated in Figure 5.1, which is a small board over Africa. The initial state, both when training and testing, was randomly generated such that each state consisted of three teams, each with at least one country. The other countries were randomly assigned a team. The number of armies assigned to each country was set to 5. Using this initial state distribution, the number of countries allied with the learning agent varied from 0 to 3 at the beginning of an episode.

The exploration constant k was set to 0.5 for all agents on the team of the learning agent. This meant that actions with a higher Q value had a slightly higher probability of being chosen (see Figure 2.3). The enemy teams played using the hand-made reasonable policy, also with an exploration constant of 0.5.

To counteract looping behaviour, e.g. two allied countries moving armies between each other forever, each mode was only repeated for a fixed number of steps during each team turn. For both training and testing, war-mode was set to at most three iterations, and move-mode to at most four iterations.

The result of the experiment is illustrated as a graph in Figure 6.1. The graph shows the average score (number of remaining countries when an episode ends) of each policy generated over the increasing amount of episodes. A policy was generated for each tenth episode. The random policy performed as expected with an average of two remaining countries (out of six). The hand-made reasonable policy did not perform as well as expected, only achieving a score of 3.44. The learnt policies generally performed better than a random policy, and a couple were even close to the reasonable policy with a score of 3.26. Over time, their performance seemed to degrade though. They stabilised somewhat after 500 episodes, whereafter they only fluctuated by approximately one remaining country.

The number of unfinished games (ending as draws) increased a great deal as more episodes were used to train the agent. This is illustrated in Figure 6.2. The high number of unfinished games suggest that the agent learns to play very defensively, not willing to attack. Since the agent was trained by playing against the hand-made

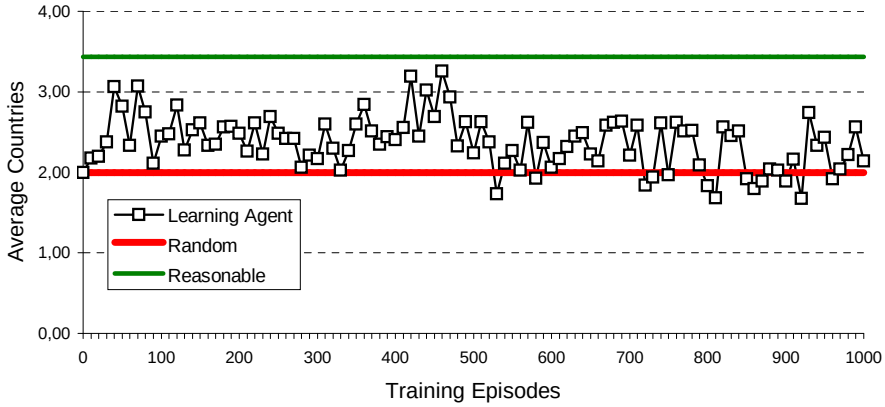


Figure 6.1: The score of the trained policies.

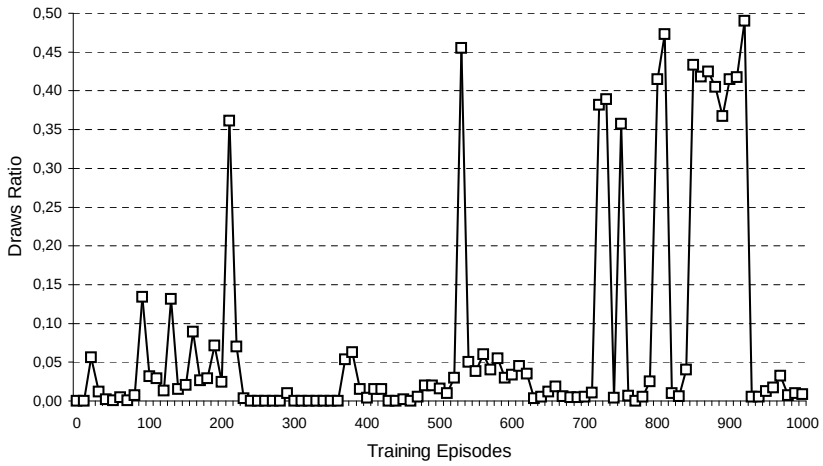


Figure 6.2: The number of draws during testing.

reasonable policy, it might have learnt to prolong a likely defeat, thus discounting the negative reward for as long as possible. Another possibility is that examples with `pass` actions are assigned too high Q values, because they are almost always present as the last action before a win. This assumption is further tested in Section 6.6.

The graph fluctuates a great deal. This is most likely caused by close-lying Q values, where only a couple of examples can change the optimal action of a given state. Another factor is, that even if the learning agent performs the worst action in a state, the opponent might do something even worse, thus, the bad action might end up resulting in a positive reward. Similarly, choosing the best action in a state might result in a defeat, giving the good action a penalty. Last, it might be that 1000 episodes is not nearly enough to actually learn a good policy for playing Simple Risk.

The poor performance of the hand-made reasonable policy suggests that this setup introduces too much randomness. The small board does not give good agents the opportunity to move armies to the front line, since all countries are on the front most of the time. Furthermore, acting randomly when the number of actions is limited is often not that bad. In the next experiment, we increase the size of the board to remove much of this randomness.

6.4 Experiment Two: Increasing the Size of the Board

For this experiment, the board was changed from Africa to the board illustrated in Figure 6.3. In the initial state of each episode, each team was assigned six countries (illustrated by different shades in the figure), giving a total of 18 countries on the board. Each of these countries were given five armies. This board layout was designed such that some countries started out on the front, while some were further away.

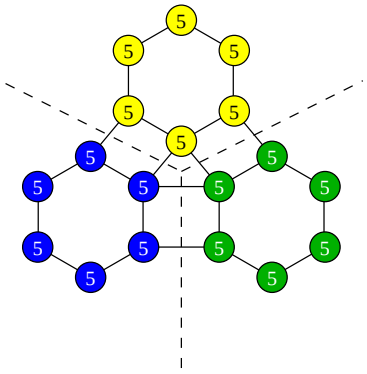


Figure 6.3: The new larger board.

During training, the exploration constant for the learning agent and its team was set to 0.5. The two enemy teams also used the policy being learnt, meaning that the agent was playing against itself. The exploration constant of enemy teams was set to 0.5.

The result of the experiment is illustrated in Figure 6.4. Again the random policy performed as expected with an average of six remaining countries. The reasonable policy also performed as expected with an average score of 15.26. This is because the

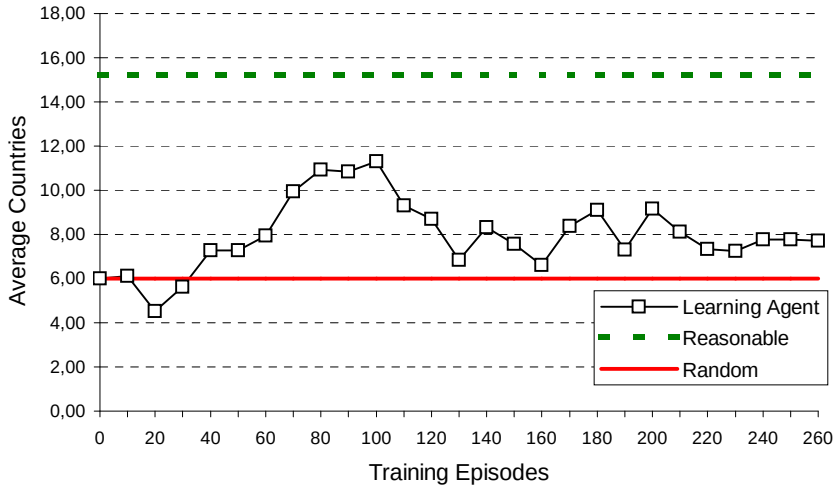


Figure 6.4: The score of trained policies on the large board

ability to move armies to the front means much more on this larger board. The learnt policy begins by performing worse than the random policy, but quickly recovers and reaches a peak of 11.31. Following, its score drops to below 10 countries and stays there.

Although the learnt policy never gets as high a score as reasonable, it is encouraging to see that it, except for the first episodes, clearly outperforms the random policy. However, it seems that the agent actually learns a great deal, but then stabilises on a score much worse than its best performance. This is most likely caused by the same problems as were mentioned in the previous experiment.

6.5 Experiment Three: Bootstrapping

In this experiment, we used the policy that peaked in the previous experiment to bootstrap a new policy. Instead of letting the agent play against itself, we tried to let it play against a random policy. It was the hope that the agent would win most games and be reinforced in its actions. For this reason, the exploration constant for the team of the agent was set to 0.1.

Figure 6.5 shows the results of this experiment. The graph evolves pretty much identical to the results of the previous experiment. This suggests that the agent is not learning to play defensively by playing against a better opponent. Instead, it seems more likely that examples with `pass` actions are receiving too high Q values because of their location in an episode. The `pass` action is most often the last action in a victory, since the probability of the learning agent performing the last attack is only 1/17. Being the last action, it receives a higher Q value than e.g. a possible previous `declarewar` action—thereby making the agent more passive. The next experiment is aimed at clarifying this problem.

It is worth mentioning, that out of the 7100 games played for this and the previous experiment, all the learnt policies together only lost a total of 20 games. 12 of these games were lost by policies created over 10 and 20 episodes, respectively. The Q tree

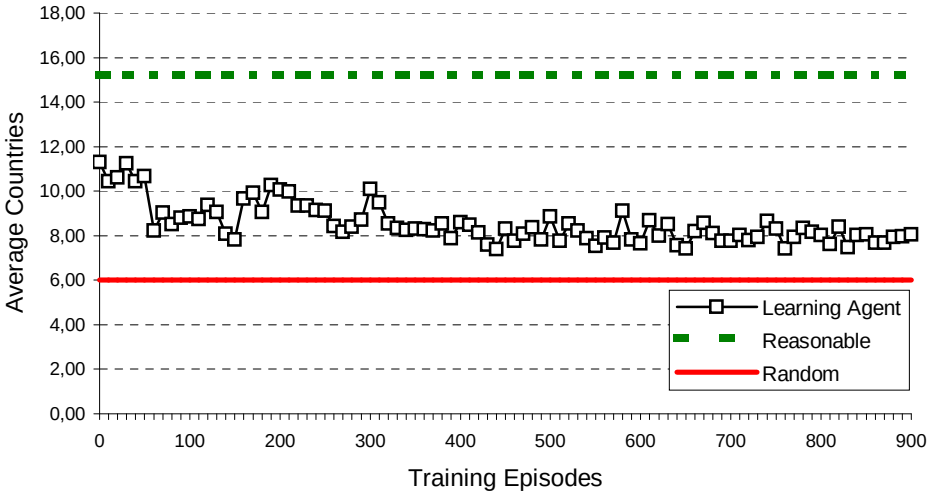


Figure 6.5: The score of trained policies on the big board using bootstrapping

that performed best (generated over 100 episodes) is illustrated as Prolog code in Appendix D.

6.6 Experiment Four: Defensive Policies

The performance of the hand-made reasonable policy was tested in Section 6.4, and it showed that it performs well. In this experiment, we used yet another board, which favoured agents able to move armies to the front to an even greater extent. The board is shown in Figure 6.6. To perform well on this board, armies must be moved from the outer countries towards the front.

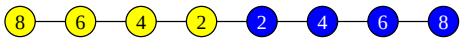


Figure 6.6: A board favouring agents able to move armies to the front.

Using this board, we generated 1000 episodes with the hand-made reasonable policy, which performed extremely well and won every single game. These examples were used to generate policies for every 100th episode. The performance of these policies should approach the performance of the reasonable policy. Figure 6.7 shows the draw ratio of the induced policies.

As was suspected, even though the policies were generated over episodes in which the reasonable agent won every game by playing aggressively, these policies still played defensively, thereby getting a lot of unfinished games. Two possible solutions to this problem were tested. The first solution was to remove all examples with a `pass` action from the set of examples before a Q tree was induced. However, this of course resulted in a poor performing agent, which kept attacking no matter the odds.

The second solution was to reduce the Q value of examples with a `pass` action by a factor of 0.5—thus easing the problem of assigning these examples too high a value. Figure 6.8 shows the results of this.

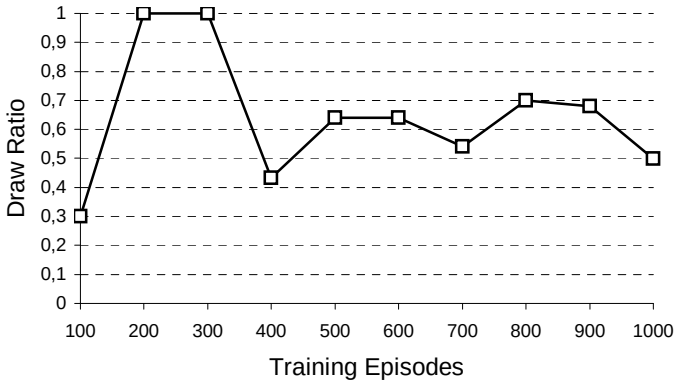


Figure 6.7: Policies created over examples generated by the reasonable policy.

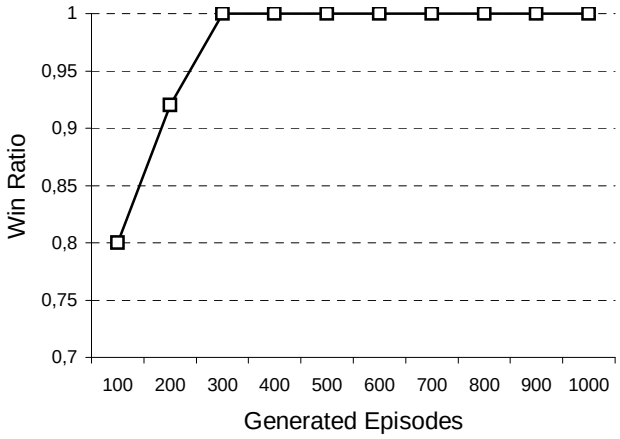


Figure 6.8: Win ratio of policies generated with reduced Q values for examples with `pass` actions.

As illustrated, the generated policies quickly reached a win ratio of one, and did not play defensively. It therefore seems likely, that this was the primary factor that prevented our learnt policies from performing even better. Due to the time constraints of this project, this assumption was not further tested.

6.7 Summary

This chapter described the experiments performed on the non-deterministic single-agent environment of Simple Risk. The environment was tested by the implementation of a hand-made reasonable policy, which was shown to perform quite well.

The first experiment was conducted using a board inspired by Africa, but the small size of this board introduced too much randomness. In the two next experiments, the board was therefore enlarged. This resulted in an agent with some skill, although not as much as expected. The last experiment illustrated that this was most likely caused by `pass` actions being assigned too great Q values.

Together, the experiments performed in this chapter does, however, show that it is possible to apply relational reinforcement learning to non-deterministic multi-agent environments. In spite of the mentioned problem, the agent rarely lost any games. Unfortunately, it is unclear how much an agent could have learnt using the generated number of episodes, had this problem not existed.

Chapter 7

Conclusion

Even in the simplest and smallest domains, teaching a machine the skills needed to complete a task is not easy. As the state space becomes larger, so does the amount of computation power needed, in fact adding just a little complexity to a domain can cause a massive state explosion.

A technique used to minimise the effects of state explosion is called Relational Reinforcement Learning. Relational Reinforcement Learning builds upon the principles introduced in Reinforcement Learning. So far the main effort within the field of Relational Reinforcement Learning has concerned deterministic environments, where only a single agent is acting.

7.1 Contributions

To explore Relational Reinforcement Learning, a toolbox was implemented in which relational problems, such as Blocks World, could be specified. This toolbox allowed a user to specify a problem, whereafter the toolbox performed example generation and Q tree induction based on the specified problem. To test the validity of the toolbox, Blocks World was implemented. The results generated by the toolbox matched those found in [14].

With this in mind, the report continued by describing, and subsequently implementing, a general method for learning and generating policies when faced with a turn based non-deterministic multi-agent environment. Existing agents cooperated through message passing, and competed against each other. Turns were controlled by using modes and letting agents pass control among each other. As mentioned, this method can be applied to a wide range of relational problems.

7.2 Results

The possibility of learning a reasonable policy in a non-deterministic multi-agent environment using the described technique was shown to be difficult. Although the learnt policies did show that the agent had indeed learnt some aspects of playing Simple Risk, they did not perform as well as expected. The main problem seemed to be that the quality of passing control was given too high Q values. This assumption was

justified by creating policies using examples generated by an agent playing with the hand-made reasonable policy. Even though the reasonable agent won every training game by being aggressive, the policies generated using these examples resulted in a very passive agent. Similarly generated policies, except the Q values of passing control were decreased by a factor of two, did not show signs of this problem.

Generally, multiple agents introduce so much non-determinism, that it is unclear how successfully reinforcement learning can be without generating a very large number of episodes. In Simple Risk, the resulting state of performing an action in a state can be virtually anything due to the interference of other agents. This results in Q values converging extremely slowly.

The implemented toolbox can be applied to both deterministic and non-deterministic environments, although not all specific settings can be specified yet. Some work still remains in making the multi-agent part of the toolbox completely general, as well as keeping the graphical user interface up to date with the underlying functionality.

7.3 Future Work

First and foremost, further testing in multi-agent environments should be conducted using variations of the applied method. For instance, the resulting state of an action performed by an agent could be the immediate following state, and not the state resulting from the interference of other agents. This reduces the non-determinism, but it is unclear if agents can learn the consequence of their actions in such a setup.

The maximal number of episodes generated for training an agent was 1000. The performance of policies based on more than the current number of training episodes should of course be tested. Before attempting such a task, the performance of generated Q trees should be increased by using a different tree construction. Specifically, the current construction (from [14]) forces the Prolog engine to re-test the same queries again and again. The performance can be improved even further by using an incremental algorithm for inducing Q trees.

Further appliance of guidance might also be interesting.

Appendix A

Toolbox GUI

This appendix illustrates screenshots of the RRL toolbox GUI.

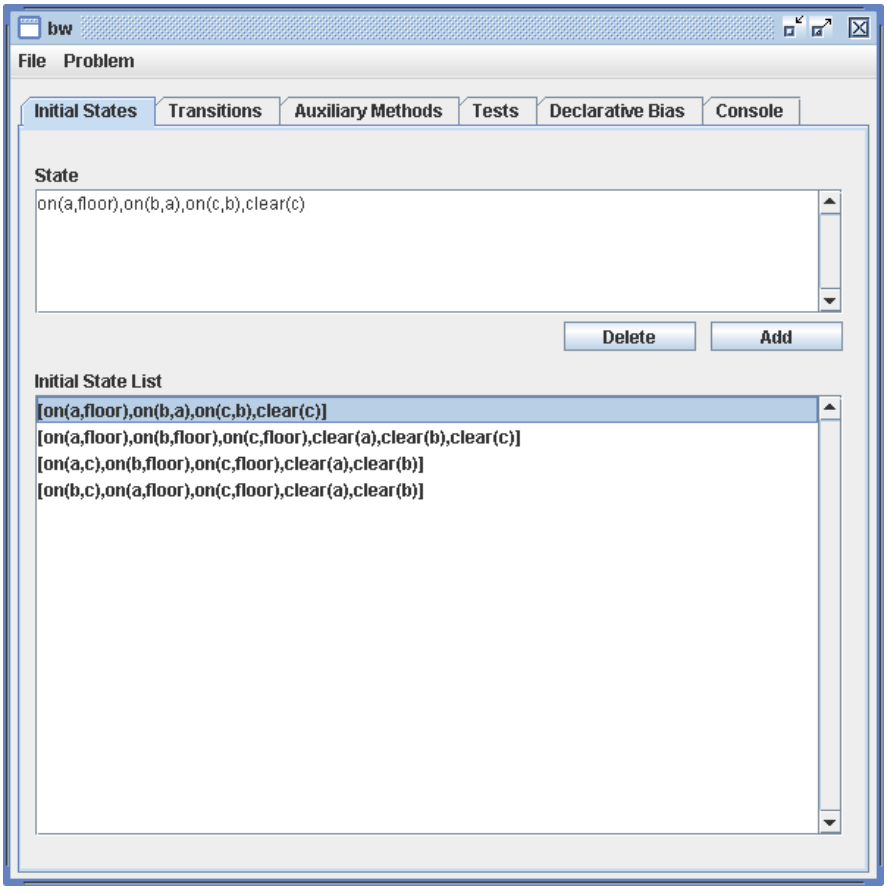


Figure A.1: Specifying initial states in the toolbox.

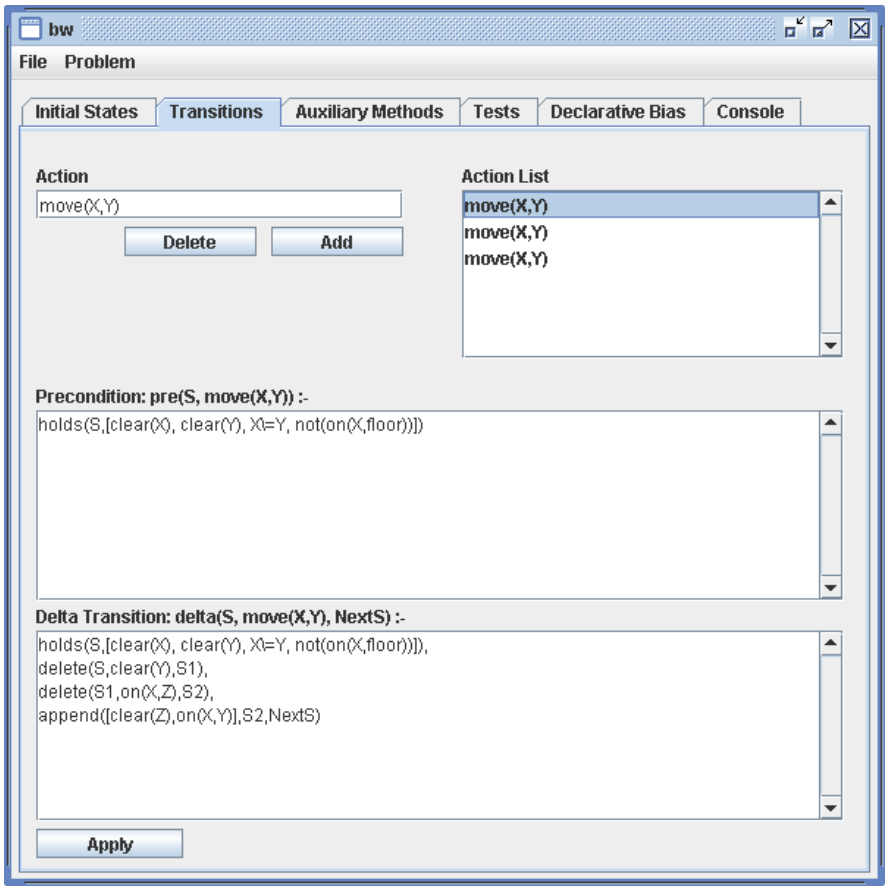


Figure A.2: Specifying transitions in the toolbox.

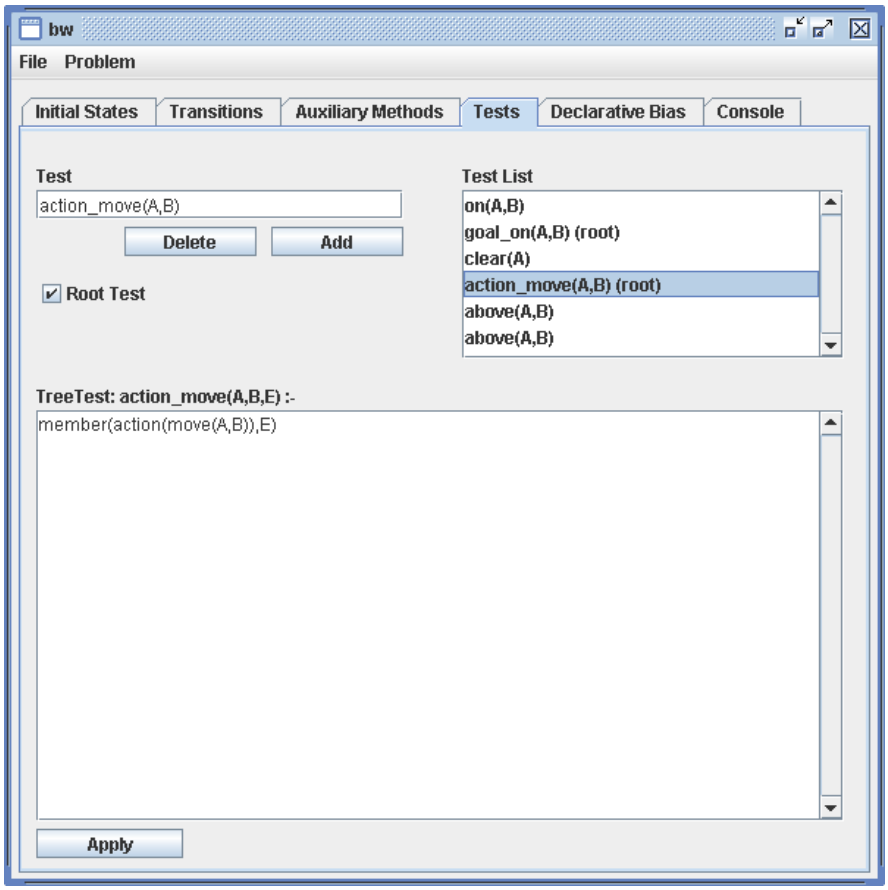


Figure A.3: Specifying tests and background knowledge in the toolbox.

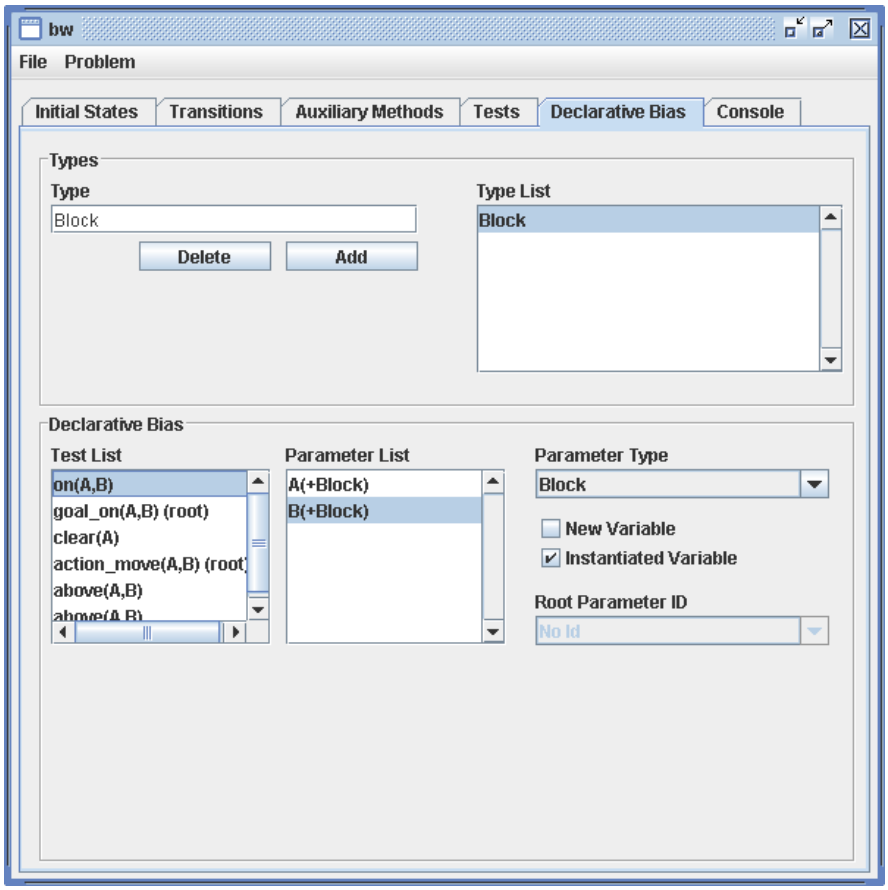


Figure A.4: Specifying a declarative bias in the toolbox.

Appendix B

Blocks World Specification

This appendix shows the specification of Blocks World using the RRL toolbox.

B.1 Prolog Knowledge Base

```
%File autogenerated by RRL Toolbox
%Generated Prolog code is testet on SWI-Prolog 5.4.2
%Problem name: bw

%Auxiliary Predicates
holds(S,[]).
holds(S,[ XY | R ]) :-
        XY, !, holds(S,R).
holds(S,[ not(A) | R ]) :-
        not(member(A,S)), holds(S,R).
holds(S,[A | R]) :-
        member(A,S), holds(S,R).

%Transitions
pre(S, move(X,Y)) :-
        holds(S,[clear(X), clear(Y), XY, not(on(X,floor))]).
pre(S, move(X,Y)) :-
        holds(S,[clear(X), clear(Y), XY, on(X,floor)]).
pre(S, move(X,floor)) :-
        holds(S,[clear(X), not(on(X,floor))]).

delta(S, move(X,Y), NextS) :-
        holds(S,[clear(X), clear(Y), XY, not(on(X,floor))]),
        delete(S,clear(Y),S1),
        delete(S1,on(X,Z),S2),
        append([clear(Z),on(X,Y)],S2,NextS)
.

delta(S, move(X,Y), NextS) :-
        holds(S,[clear(X), clear(Y), XY, on(X,floor)]),
        delete(S,clear(Y),S1),
```

```
            delete(S1,on(X,floor),S2),
            append([on(X,Y)],S2,NextS)
.

delta(S, move(X,floor), NextS) :-
            holds(S,[clear(X), not(on(X,floor))]),
            delete(S,on(X,Z),S1),
            append([clear(Z),on(X,floor)],S1,NextS)
.

%Tests
on(X,Y,E) :- member(on(X,Y),E).
action(move(X,Y,E)) :- member(action(move(X,Y)),E).
clear(X,E) :- member(clear(X),E).
above(X,Y,E) :- on(X,Y,E).
above(X,Y,E) :- on(X,Z,E), above(Z,Y,E).
equal(X,X,E).
%Goal
goal(S,X) :- call(X,S).
goal(on(X,Y,E)) :- member(goal(on(X,Y)),E).
```

B.2 Declarative Bias File

```
begin(types)
    Block
end(types)

begin(tests)
    on(+Block,+Block)
    action(move(+Block,+Block))
    equal(+Block,+Block)
    clear(+Block)
end(tests)

begin(root)
    goal(on(Block-0,Block-1))
    action(move(Block-2,Block-3))
end(root)
```

B.3 Initial State File

```
clear(b),on(c,floor),on(a,floor),on(b,a),clear(c)
on(b,c),clear(a),clear(b),on(c,floor),on(a,floor)
clear(a),on(b,floor),clear(c),clear(b),on(c,floor),on(a,floor)
clear(c),on(b,floor),on(c,a),clear(b),on(a,floor)
clear(a),on(b,c),clear(b),on(c,floor),on(a,floor)
clear(a),on(b,floor),clear(b),on(c,floor),on(a,floor),clear(c)
on(c,a),on(b,floor),clear(c),clear(b),on(a,floor)
clear(b),on(c,floor),on(a,floor),on(b,a),clear(c)
clear(c),on(b,floor),clear(a),clear(b),on(c,floor),on(a,floor)
clear(b),on(c,a),clear(c),on(b,floor),on(a,floor)
```

clear(c),on(b,floor),clear(a),clear(b),on(c,floor),on(a,floor)
clear(a),on(b,c),clear(b),on(c,floor),on(a,floor)
clear(a),on(c,b),clear(c),on(b,floor),on(a,floor)
on(b,c),on(c,a),clear(b),on(a,floor)
clear(a),on(b,floor),clear(b),on(c,floor),on(a,floor),clear(c)
clear(a),on(b,floor),clear(b),on(c,floor),on(a,floor),clear(c)
on(c,b),clear(c),on(b,floor),clear(a),on(a,floor)
on(a,c),clear(a),on(b,floor),clear(b),on(c,floor)
clear(c),on(a,floor),clear(a),on(c,b),on(b,floor)
clear(b),on(c,floor),on(a,floor),on(b,a),clear(c)
on(a,c),clear(a),on(c,b),on(b,floor)
on(a,floor),on(b,a),on(c,b),clear(c)
on(b,a),on(a,c),clear(b),on(c,floor)
on(a,c),clear(a),on(b,floor),clear(b),on(c,floor)

Appendix C

Simple Risk Specification

C.1 Prolog Knowledge Base

```
%Auxiliare Predicates
countries([],_,0).
countries([State|R], TeamId, Count) :-
    member(t(TeamId),State) ->
        countries(R, TeamId, Count1),
        Count is Count1 + 1 ;
        countries(R, TeamId, Count).

delete_elements(S, [], S).
delete_elements(S, [H|T], NextS) :-
    (member(H,S) -> delete(S, H, S1) ; S1 = S),
    delete_elements(S1, T, NextS).

notmember(X,L) :- not(member(X,L)).

setNArmies(Country, Force, S, NextS) :-
    (replace(ally(Country,_), ally(Country,Force), S, S1) -> true
    ; S1=S),
    (replace(enemy(Country,_), enemy(Country,Force), S1, NextS) -> true
    ; NextS=S1).
list_setNArmies(Country, Force, States, NextStates) :-
    maplist(setNArmies(Country,Force), States, NextStates).

setNArmyRequest(From, S, NextS) :-
    (not(member(armyrequest(From),S)),
    member(ally(From,_), S),
    append(S, [armyrequest(From)], NextS))-> true
    ; NextS=S.
list_setNArmyRequest(From, States, NextStates) :-
    maplist(setNArmyRequest(From), States, NextStates).

convertAllyToEnemy(ally(N,Armies), enemy(N,Armies)) :- !.
convertAllyToEnemy(X, NewFact) :- NewFact = X.
list_convertAllyToEnemy(S, NextS) :-
```

```
    maplist(convertAllyToEnemy,S, NextS).

convertEnemy(Player, States, enemy(N,Armies), ally(N,Armies)) :-
    state(N, States, SN),
    member(t(Player),SN), !.
convertEnemy(Player, States, enemy(N,Armies), enemy(N,Armies)).
convertEnemy(Player, States, X, Y) :- X=Y.
list_convertEnemy(States, S, NextS) :-
    member(t(Player), S), maplist(convertEnemy(Player, States),S, NextS).

convertAllyNToEnemy(N, ally(N,Armies), enemy(N,Armies)) :- !.
convertAllyNToEnemy(N, X, NewFact) :- NewFact = X.
list_convertAllyNToEnemy2(N, S, NextS) :-
    maplist(convertAllyNToEnemy(N),S, NextS).
list_convertAllyNToEnemy(N, States, NextStates) :-
    maplist(list_convertAllyNToEnemy2(N),States, NextStates).

convertEnemyNToAlly(Player, N, S, NextS) :-
    (member(enemy(N,_),S),
    member(t(Player), S),
    replace(enemy(N,A),ally(N,A),S,NextS)) -> true
    ; NextS=S.
list_convertEnemyNToAlly(Player, N, States, NextStates) :-
    maplist(convertEnemyNToAlly(Player,N),States, NextStates).

trimArmyRequestFromN(N, S, NextS) :-
    delete(S, armyrequest(N), NextS).
list_trimArmyRequestFromN(N, States, NextStates) :-
    maplist(trimArmyRequestFromN(N),States, NextStates).

updateFrontLine(States, S, NextS) :-
    findall(front(N),
    (member(ally(N,_),S), state(N,States,SN), member(enemy(_,_),SN)),Bag),
    list_to_set(Bag, Set),
    delete(S, front(_), S1),
    append(S1,Set,NextS).
list_updateFrontLine(States, NextStates) :-
    maplist(updateFrontLine(States),States, NextStates).

update(Element, Replacement, Pattern, S, NextS) :-
    findall(Replacement, (member(Element,S),call(Pattern)), Bag),
    list_to_set(Bag, Set),
    findall(Element, (member(Element,S),call(Pattern),
    delete(S, Element,XS)), XBag), list_to_set(XBag,XSet),
    delete_elements(S, XSet, S1), append(S1,Set,NextS).
update(Element, Replacement, S, NextS) :-
    findall(Replacement, member(Element,S), Bag), list_to_set(Bag, Set),
    findall(Element, (member(Element,S),delete(S, Element,XS)), XBag),
    list_to_set(XBag,XSet),
    delete_elements(S, XSet, S1), append(S1,Set,NextS).

replaceChar(X,Y,Z,R) :-
    X=Z -> R=Y ; R=Z.
replace(X,Y,L,NextL) :-
```

```
    maplist(replaceChar(X,Y),L,NextL).

list_member(Fact, States) :- not(checklist(notmember(Fact),States)).

attack(Country,[X],[Y], L, NextL) :-
    X=<Y -> attacker_lost(Country, L, NextL)
    ; defender_lost(Country, L, NextL).

attack(Att,[X|XT],[Y], L, NextL) :-
    XT[], attack(Att,[X],[Y],L,NextL).
attack(Att,[X],[Y|YT], L, NextL) :-
    YT[], attack(Att,[X],[Y],L,NextL).
attack(Att,[X|XT],[Y|YT], L, NextL) :-
    XT[], YT[],
    attack(Att,[X],[Y],L,L2), attack(Att,XT,YT,L2,NextL).

attacker_lost(Country, States, NextStates) :-
    state(Country, States, S),
    delete(S, war(Def, AAtt), S1),
    NewAAtt is AAtt - 1,
    (NewAAtt=0 -> S2=S1 ; append(S1, [war(Def, NewAAtt)], S2)),
    replace(S, S2, States, NextStates).

defender_lost(Country, States, NextStates) :-
    state(Country, States, S),
    member(t(Team),S), member(war(Def,AAtt),S),
    state(Def, States, SDef),delete(SDef, armies(ADef),SDef1),
    replace(SDef,SDef1, States, States1),
    NewADef is ADef - 1,
    (NewADef=0 ->
        replace(enemy(Def,_), ally(Def, AAtt), S, S1),
        delete(S1, war(_,_), S2),
        replace(S,S2,States1,States2),
        replace(t(_), t(Team), SDef1, SDef2),
        update(ally(X,XA), enemy(X,XA), SDef2,SDef4),
        update(enemy(Y,YA), ally(Y,YA), (state(Y,States,YS),
        member(t(Team),YS)), SDef4, SDef5),
        delete(SDef5, armyrequest, SDef6), delete(SDef6, armyrequest(_), SDef7),
        append(SDef7, [armies(AAtt), mode(war), invaded(Country)], SDef8),
        replace(enemy(Country,A), ally(Country,A),SDef8,SDef9),
        replace(SDef1,SDef9,States2,States3),
        list_convertAllyNToEnemy(Def, States3, States4),
        list_convertEnemyNToAlly(Team, Def, States4, States5),
        list_trimArmyRequestFromN(Def, States5, States6),
        list_updateFrontLine(States6, NextStates)
        ;
        append(SDef1, [armies(NewADef)], SDef2),
        replace(SDef1, SDef2, States1, States2),
        list_setNArmies(Def, NewADef, States2, NextStates).

roll(1, 1, [AttDie], [DefDie]) :-
    AttDie is random(6)+1, DefDie is random(6)+1.
roll(2, 1, AttDice, [DefDie]) :-
    AttDie1 is random(6)+1, AttDie2 is random(6)+1,
```

```
    msort([AttDie1, AttDie2], AttDice),
    DefDie is random(6)+1.
roll(3, 1, AttDice, [DefDie]) :-
    AttDie1 is random(6)+1, AttDie2 is random(6)+1, AttDie3 is random(6)+1,
    msort([AttDie1, AttDie2, AttDie3], AttDice),
    DefDie is random(6)+1.
roll(1, 2, [AttDie], DefDice) :-
    AttDie is random(6)+1,
    DefDie1 is random(6)+1, DefDie2 is random(6)+1,
    msort([DefDie1, DefDie2], DefDice).
roll(2, 2, AttDice, DefDice) :-
    AttDie1 is random(6)+1, AttDie2 is random(6)+1,
    sort([AttDie1, AttDie2], AttDice),
    DefDie1 is random(6)+1, DefDie2 is random(6)+1,
    msort([DefDie1, DefDie2], DefDice).
roll(3, 2, AttDice, DefDice) :-
    AttDie1 is random(6)+1, AttDie2 is random(6)+1, AttDie3 is random(6)+1,
    sort([AttDie1, AttDie2, AttDie3], AttDice),
    DefDie1 is random(6)+1, DefDie2 is random(6)+1,
    msort([DefDie1, DefDie2], DefDice).
rolldice(AttForce,DefForce,AttDice, DefDice) :-
    (AttForce>3 -> X=3 ; X=AttForce),
    (DefForce>2 -> Y=2 ; Y=DefForce),
    roll(X,Y,AttDice1, DefDice1),
    reverse(AttDice1,AttDice), reverse(DefDice1,DefDice).

state(Country, L, S) :- member(S, L), member(a(Country),S).

del(X,S,NextS) :- delete(S,X,NextS).

goalreached([],TeamId).
goalreached([S|R],TeamId):-
    member(t(TeamId),S) ->
        goalreached(R,TeamId) ; fail.

goalfailed([],TeamId).
goalfailed([S|R],TeamId):-
    not(member(t(TeamId),S)) ->
        goalfailed(R,TeamId) ; fail.

beginturn(TeamId, S, NextS) :-
    member(p(TeamId),S) -> append(S,[mode(war)],NextS)
    ; NextS = S.

reinforce(TeamId, S, NextS) :-
    member(t(TeamId),S) ->
        delete(S,armies(A),S1),
        ((A < 15) ->
                NewA is A+1,
                append(S1,[armies(NewA)],NextS) ;
                append(S1,[armies(A)],NextS))
    ; NextS = S.

armyConvert(Army,1) :- Army=1.
```

```
armyConvert(Army,2) :- Army=2.
armyConvert(Army,3) :- Army=3.
armyConvert(Army,4) :- Army>=4, Force=<10.
armyConvert(Army,5) :- Army>10.

%Transitions
pre(States, Id, TeamId, declarewar(C,A)) :-
    not(list_member(war(_,_),States)),
    member(A,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]),
    state(Id, States, S),
    member(mode(war), S),
    member(enemy(C,_),S),
    member(armies(AHome), S), AHome>A.

pre(States, Id, TeamId, attack) :-
    state(Id, States, S), member(war(_,_),S).
pre(States, Id, TeamId, retreat) :-
    state(Id, States, S), member(war(_,_),S).
pre(States, Id, TeamId, movearmy(C)) :-
    state(Id, States, S), member(requestarmy(C),S),
    member(mode(move),S),
    member(armies(AHome), S), AHome>1.
pre(States, Id, TeamId, requestarmy) :-
    state(Id, States, S), member(mode(move), S),
    not(member(armyrequest,S)).
pre(States, Id, TeamId, pass):-
    state(Id, States, S), not(member(war(_,_),S)).

delta(States, Id, TeamId, declarewar(C,A), NextStates) :-
    pre(States, Id, TeamId, declarewar(C,A)),
    state(Id, States, S),
    delete(S, armies(AHome), S1),
    NewAHome is AHome - A,
    append(S1, [war(C,A), armies(NewAHome)], S2),
    delete(S2, invaded(_), S3),
    replace(S, S3, States, States1),
    list_setNArmies(Id, NewAHome, States1, States2),
    delta(States2,Id,TeamId,attack,NextStates).
delta(States, Id, TeamId, retreat, NextStates) :-
    pre(States, Id, TeamId, retreat),
    state(Id, States, S),
    delete(S, armies(AHome), S1), delete(S1, war(_,AAtt), S2),
    NewAHome is AHome + AAtt - 1,
    append(S2, [armies(NewAHome)], S3),
    delete(S3, invaded(_), S4),
    replace(S, S4, States, States1),
    list_setNArmies(Id, NewAHome, States1, NextStates).
delta(States, Id, TeamId, movearmy(C), NextStates) :-
    pre(States, Id, TeamId, movearmy(C)),
    state(Id, States, S), state(C, States, SC),
    delete(S, armies(AHome), S1), delete(SC, armies(ACHome), SC1),
    NewAHome is AHome - 1, NewACHome is ACHome +1,
    append(S1, [armies(NewAHome)], S2),
    delete(S2, invaded(_), S3),
```

```
    append(SC1,[armies(NewACHome)], SC2),
    delete(SC2, armyrequest, SC3),
    replace(S, S3, States, States1), replace(SC, SC3, States1, States2),
    list_setNArmies(Id, NewAHome, States2, States3),
    list_setNArmies(C, NewACHome, States3, States4),
    list_trimArmyRequestFromN(C, States4, NextStates).
delta(States, Id, TeamId, requestarmy, NextStates) :-
    pre(States, Id, TeamId, requestarmy),
    state(Id, States, S), append(S, [armyrequest], S1),
    delete(S1, invaded(_), S2), replace(S, S2, States, States1),
    list_setNArmyRequest(Id, States1, NextStates).
delta(States, Id, TeamId, pass, NextStates)
    pre(States, Id, TeamId, pass).
delta(States, Id, TeamId, attack, NextStates) :-
    pre(States, Id, TeamId, attack),
    state(Id, States, S), member(war(Def, AAtt),S),
    state(Def, States, SDef), member(armies(ADef), SDef),
    rolldice(AAtt, ADef, AttDice, DefDice),
    attack(Id, AttDice, DefDice, States, NextStates).

%Mode Predicates endmode(States, attack, TeamId, NextTeamId, NextStates) :-
    maplist(replace(mode(war),mode(move)), States, NextStates).

endmode(States, move, TeamId, NextTeamId, NextStates) :-
    maplist(del(mode(move)), States, States1),
    maplist(reinforce(NextTeamId), States1, States2),
    maplist(beginturn(NextTeamId), States2, NextStates).

%Tests war(Country,Armies,E) :- member(war(Country, X),E), armyConvert(X,Armies).
ally(Country, Armies,E) :- member(ally(Country,X),E), armyConvert(X,Armies).
enemy(Country, Armies,E) :- member(enemy(Country,X),E), armyConvert(X,Armies).
armies(Armies,E) :- member(armies(X),E), armyConvert(X,Armies).
attackmode(E) :- member(mode(war),E).
movemode(E) :- member(mode(move),E).
armyrequest(E) :- member(armyrequest,E).
armyrequest(N,E) :- member(armyrequest(N),E).
invaded(C,E) :- member(invaded(C), E).
invaded(E) :- member(invaded(_),E).
front(C,E) :- member(front(C),E).

action_declarewar(Country,Armies,E) :-
    member(action(declarewar(Country,X)),E), armyConvert(X,Armies).
action_attack(E) :- member(action(war),E).
action_retreat(E) :- member(action(retreat),E).
action_movearmy(Country,E) :- member(action(movearmy(Country)),E).
action_requestarmy(E) :- member(action(requestarmy),E).
action_pass(E) :- member(action(pass),E).
equal(X,Y,_) :- X=Y.
notequal(X,Y,_) :- not(X=Y).
gt(X,Y,_) :- X>Y.
fequal(X,Y,_) :- armyConvert(X,X1), armyConvert(Y,Y1), X1=Y1.
fgt(X,Y,_) :- armyConvert(X,X1), armyConvert(Y,Y1), X1>Y1.

max_attack(A,E) :- member(armies(X),E), X1 is X-1, armyConvert(X1,A).
```

```
neighborsOfAlignment(0,A,[]).
neighborsOfAlignment(X,A,[F|R]) :-
    functor(F,Name,2),A=Name,
    neighborsOfAlignment(Y,A,R), X is Y + 1.
neighborsOfAlignment(X,A,[F|R]) :-
    neighborsOfAlignment(X,A,R).

invaded(States,TeamId,Id):-
    state(Id,States,S), not(member(p(TeamId),S)).

enemies(X,E) :- neighborsOfAlignment(X,enemy,E).
allies(X,E) :- neighborsOfAlignment(X,ally,E).
neighbors(X,E) :- enemies(X1,E), allies(X2,E),X is X1+X2.

%Goal
goal(enemies(X,E)) :- member(goal(enemies(X)),E).
```

C.2 Declarative Bias File

```
begin(types)
    C
    A
    I
end(types)

begin(tests)
    action(declarewar(+-C,+-A))
    action(attack)
    action(retreat)
    action(movearmy(+-C))
    action(requestarmy)
    invaded
    war(+-C,+-A)
    ally(+-C,+-A)
    enemy(+-C,+-A)
    attackmode
    movemode
    armyrequest
    armyrequest(+-C)
    front(+-C)
    equal(+C,+C)
    gt(+I,+I)
    equal(+I,+I)
    aequal(+A,+A)
    agt(+A,+A)
    max_attack(+-A)
end(tests)

begin(root)
    goal(enemies(I-0))
    enemies(I-1)
    armies(A-0)
```

```
end(root)
```

Appendix D

Q Tree

This appendix shows a Q tree for the Simple Risk environment generated by the toolbox after 100 episodes of learning.

```
q(-0.00860,4,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A1,E), armyrequest(E),
action_declarewar(C1,A1,E), ally(C2,A0,E), !.
q(-0.11105,51,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A1,E), armyrequest(E),
action_declarewar(C1,A1,E), !.
q(-0.00331,54,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A1,E), armyrequest(E), attackmode(E), !.
q(-0.07895,13,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A1,E), armyrequest(E), !.
q(-0.06856,1,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A1,E), action_requestarmy(E), !.
q(-0.00571,13,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A1,E), action_declarewar(C0,A0,E), !.
q(-0.00376,12,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A1,E), action_declarewar(C1,A2,E),
ally(C2,A2,E), !.
q(0.00032,26,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A1,E), action_declarewar(C1,A2,E),
ally(C2,A0,E), !.
q(-0.00169,7,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A1,E), action_declarewar(C1,A2,E), !.
q(0.00140,31,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A1,E), !.
q(-0.15023,4,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), movemode(E), enemy(C1,A0,E), !.
q(0.00693,8,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), movemode(E), max_attack(A0,E),
ally(C1,A0,E), armyrequest(C0,E), ally(C0,A0,E), !.
q(-0.00339,125,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), movemode(E), max_attack(A0,E),
ally(C1,A0,E), armyrequest(C0,E), armyrequest(E), !.
q(-0.00058,21,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
```

```
max_attack(A1,E), front(C0,E), movemode(E), max_attack(A0,E),
ally(C1,A0,E), armyrequest(C0,E), !.
q(-0.00954,48,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), movemode(E), max_attack(A0,E),
ally(C1,A0,E), !.
q(-0.04652,58,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), movemode(E), max_attack(A0,E),
armyrequest(C0,E), !.
q(0.01380,3,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), movemode(E), max_attack(A0,E), !.
q(-0.00884,70,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), movemode(E), ally(C1,A1,E), !.
q(0.01389,62,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), movemode(E), armyrequest(C0,E), !.
q(0.04285,37,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), movemode(E), !.
q(-0.04110,7,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), action_declarewar(C1,A1,E),
armyrequest(E), !.
q(-0.00709,4,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), action_declarewar(C1,A1,E), ally(C0,A0,E),
!.
q(-0.00177,33,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), action_declarewar(C1,A1,E), !.
q(0.00472,34,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), ally(C1,A1,E), ally(C0,A0,E),
armyrequest(C2,E), ally(C2,A1,E), !.
q(0.02951,2,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), ally(C1,A1,E), ally(C0,A0,E),
armyrequest(C2,E), !.
q(-0.00065,23,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), ally(C1,A1,E), ally(C0,A0,E), !.
q(-0.00172,132,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), ally(C1,A1,E), ally(C2,A0,E),
armyrequest(C1,E), armyrequest(C0,E), attackmode(E), !.
q(0.00000,4,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), ally(C1,A1,E), ally(C2,A0,E),
armyrequest(C1,E), armyrequest(C0,E), !.
q(-0.00319,19,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), ally(C1,A1,E), ally(C2,A0,E),
armyrequest(C1,E), !.
q(-0.00038,66,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), ally(C1,A1,E), ally(C2,A0,E), !.
q(-0.00179,37,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), ally(C1,A1,E), armyrequest(C1,E), !.
q(-0.01167,50,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), ally(C1,A1,E), !.
q(-0.00522,23,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), enemy(C1,A2,E), armyrequest(C0,E), !.
q(0.00428,5,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), enemy(C1,A2,E),
action_declarewar(C1,A2,E), !.
q(0.00283,15,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), enemy(C1,A2,E), war(C1,A3,E), !.
```

```
q(0.00101,17,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), enemy(C1,A2,E), enemy(C1,A0,E), !.
q(-0.00168,17,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), enemy(C1,A2,E), !.
q(-0.00090,110,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), max_attack(A0,E), armyrequest(C0,E),
attackmode(E), !.
q(0.00000,4,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), max_attack(A0,E), armyrequest(C0,E), !.
q(0.04328,4,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), max_attack(A0,E), !.
q(0.00922,69,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), ally(C1,A0,E), !.
q(0.02461,97,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), front(C0,E), !.
q(0.00253,3,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), movemode(E),
armyrequest(E), armyrequest(C0,E), enemy(C1,A2,E), !.
q(0.00682,158,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), movemode(E),
armyrequest(E), armyrequest(C0,E), !.
q(-0.00156,12,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), movemode(E),
armyrequest(E), !.
q(-0.00086,39,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), movemode(E), !.
q(0.00360,39,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), enemy(C1,A2,E),
action_declarewar(C1,A0,E), !.
q(0.00244,14,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), enemy(C1,A2,E),
action_declarewar(C1,A2,E), !.
q(0.00028,34,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), enemy(C1,A2,E), !.
q(-0.00146,87,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), armyrequest(E),
attackmode(E), armyrequest(C0,E), !.
q(0.00000,1,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), armyrequest(E),
attackmode(E), !.
q(0.00000,3,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), armyrequest(E), !.
q(-0.00027,61,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), ally(C0,A0,E), !.
q(0.03137,30,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), movemode(E), !.
q(0.00159,27,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), ally(C0,A1,E), !.
q(0.00500,27,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A2,E), attackmode(E), fgt(A0,A2,E),
ally(C1,A0,E), !.
q(-0.00079,33,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A2,E), attackmode(E), fgt(A0,A2,E),
action_declarewar(C1,A3,E), !.
```

```
q(-0.01511,13,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A2,E), attackmode(E), fgt(A0,A2,E), !.
q(0.06750,2,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A2,E), attackmode(E), !.
q(-0.09301,5,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), enemy(C0,A2,E), !.
q(0.05413,1,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), fequal(A0,A1,E), movemode(E), max_attack(A0,E),
action_requestarmy(E), !.
q(0.03715,115,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), fequal(A0,A1,E), movemode(E), max_attack(A0,E), !.
q(0.00000,1,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), fequal(A0,A1,E), movemode(E), !.
q(0.03395,107,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), fequal(A0,A1,E), attackmode(E), armyrequest(E),
max_attack(A0,E), !.
q(0.03958,8,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), fequal(A0,A1,E), attackmode(E), armyrequest(E), !.
q(0.05413,1,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), fequal(A0,A1,E), attackmode(E), !.
q(0.00000,7,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), fequal(A0,A1,E), !.
q(0.07896,88,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
max_attack(A1,E), !.
q(-0.08520,45,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
movemode(E), enemy(C0,A1,E), front(C1,E), ally(C1,A1,E), !.
q(-0.23222,53,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
movemode(E), enemy(C0,A1,E), front(C1,E), armyrequest(C2,E),
armyrequest(E), !.
q(-0.15376,30,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
movemode(E), enemy(C0,A1,E), front(C1,E), armyrequest(C2,E), !.
q(-0.10223,33,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
movemode(E), enemy(C0,A1,E), front(C1,E), !.
q(-0.34752,34,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
movemode(E), enemy(C0,A1,E), !.
q(-0.02814,49,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
movemode(E), ally(C0,A0,E), armyrequest(C0,E), front(C0,E),
armyrequest(E), !.
q(-0.02132,7,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
movemode(E), ally(C0,A0,E), armyrequest(C0,E), front(C0,E), !.
q(-0.11069,5,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
movemode(E), ally(C0,A0,E), armyrequest(C0,E), !.
q(-0.01115,40,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
movemode(E), ally(C0,A0,E), !.
q(0.01211,11,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
movemode(E), !.
q(-0.04193,18,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
enemy(C0,A0,E), armyrequest(E), war(C0,A1,E), !.
q(-0.20993,33,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
enemy(C0,A0,E), armyrequest(E), attackmode(E), front(C1,E), !.
q(-0.13297,68,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
enemy(C0,A0,E), armyrequest(E), attackmode(E), !.
q(0.00000,2,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
enemy(C0,A0,E), armyrequest(E), !.
```

```
q(0.00000,15,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
enemy(C0,A0,E), invaded(E), !.
q(-0.00939,20,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
enemy(C0,A0,E), !.
q(-0.03446,26,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
ally(C0,A0,E), armyrequest(C0,E), !.
q(0.00000,6,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
ally(C0,A0,E), invaded(E), !.
q(-0.00589,15,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
ally(C0,A0,E), war(C1,A1,E), !.
q(-0.02315,50,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
ally(C0,A0,E), enemy(C1,A1,E), !.
q(-0.01035,52,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
ally(C0,A0,E), !.
q(-0.00136,11,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
front(C0,E), war(C1,A1,E), !.
q(-0.00274,26,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
front(C0,E), armyrequest(E), !.
q(-0.00667,23,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E),
front(C0,E), !.
q(0.00005,14,E) :- goal_enemies(I0,E), enemies(I1,E), armies(A0,E), !.
```

Bibliography

[1] *InterProlog 2.1*. http://www.declarativa.com/InterProlog/default.htm.

[2] *IVisual Studio .Net 2003*. http://msdn.microsoft.com/vstudio/.

[3] *JDK 1.5*. http://java.sun.com/.

[4] *JPL 3.0.3*. http://www.swi-prolog.org/packages/jpl/.

[5] *SWI Prolog*. http://www.swi-prolog.org/.

[6] Richard Bellman. *Dynamic Programming*. Princeton University, 1957.

[7] Hendrik Blockeel. *Top-Down Induction of First Order Logical Decision Trees*. PhD thesis, Katholieke Universiteit Leuven, Belgium, December 1998. http://www.cs.kuleuven.ac.be/~dtai/publications/files/20977.ps.gz.

[8] Hendrik Blockeel, Luc Deshaspe, Jan Ramon, and Jan Struyf. *The ACE Data Mining System - User's Manual*. http://www.cs.kuleuven.ac.be/~dtai/ACE/.

[9] Hendrik Blockeel and Luc De Raedt. Top-Down Induction of First-Order Logical Decision Trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

[10] Thomas G. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *J. Artif. Intell. Res. (JAIR)*, 13:227–303, 2000.

[11] Kurt Driessens. *Relational Reinforcement Learning*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2004. http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41286.

[12] Kurt Driessens and Sašo Džeroski. Integrating Guidance into Relational Reinforcement Learning. *Machine Learning*, 57:271–304, December 2004.

[13] Kurt Driessens, Jan Ramon, and Hendrik Blockeel. Speeding up Relational Reinforcement Learning Through the Use of an Incremental First Order Decision Tree Learner. In *ECML*, volume 2167 of *Lecture Notes in Computer Science*. Springer, 2001. ISBN 3-540-42536-5.

[14] Sašo Džeroski, Luc De Raedt, and Hendrik Blockeel. Relational Reinforcement Learning. In *International Workshop on Inductive Logic Programming*, pages 11–22, 1998.

[15] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational Reinforcement Learning. *Machine Learning*, 43(1/2):7–52, 2001.

[16] Floriana Esposito, Donato Malerba, and Giovanni Semeraro. A Comparative Analysis of Methods for Pruning Decision Trees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19(5):476–491, 1997.

[17] Usama M. Fayyad and Keki B. Irani. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. In *IJCAI*, pages 1022–1029, 1993.

[18] Hasbro. *Risk Homepage*. http://www.hasbro.com/risk.

[19] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement Learning: A Survey. *J. Artif. Intell. Res. (JAIR)*, 4:237–285, 1996.

[20] Harry Klopf. Brain Function and Adaptive Systems—A Heterostatic Theory. Technical report, Air Force Cambridge Research Laboratories, 1972. AFCRL-72-0164.

[21] Harry Klopf. *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence*. Hemisphere, Washington, D.C., December 1982. ISBN 089116202X.

[22] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 0-07-115467-1.

[23] Tom M. Mitchell. *Machine Learning*, chapter 13 – Reinforcement Learning. McGraw-Hill, New York, 1997.

[24] Kevin Murhpy. Markov Decision Process Toolbox for Matlab. `http://www.cs.ubc.ca/~murphyk/Software/MDP/mdp.html`.

[25] Armand Prieditis. Machine Discovery of Effective Admissible Heuristics. In *IJCAI*, pages 720–725, 1991.

[26] J. Ross Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986.

[27] J. Ross Quinlan. Simplifying Decision Trees. *International Journal of Man-Machine Studies*, 27(3):221–234, 1987.

[28] J. Ross Quinlan. *C4.5: Programs for Machine Learning Morgan Kaufmann*. Morgan Kaufmann, 1993. ISBN 1-558-60238-0.

[29] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Bradford Books, 1998. ISBN 0-262-19398-1.

[30] Edward L. Thorndike. *Animal Intelligence*. Thoemmes Continuum, 1911. ISBN 185506698X.

[31] Christopher J.C.H. Watkins and Peter Dayan. Technical Note: Q-Learning. *Machine Learning*, 8(3-4):279–292, May 1992.

References containing URLs are valid as of January 13, 2005.