Faculty of Engineering and Science
University of Aalborg

Department of Computer Science

TITLE:
Relational Reinforcement Learning

PROJEKT PERIOD:
DAT6,
February 2004 - July 2004

PROJECT GROUP:
d633a

GROUP MEMBERS:
Klaus Jensen, santakj@cs.auc.dk

SUPERVISOR:
Uffe Kjærulff, uk@cs.auc.dk

NUMBER OF COPIES: 4

NUMBER OF PAGES: 66

SYNOPSIS:

Many machine learning problems are hard to solve due to the size of the state space used in the application. In such a case, finding the optimal solution requires a lot of computation. This report is part of a project, where focus lies on finding ways to decrease the size of state spaces used in small computer games. A commonly used machine learning technique known as Reinforcement Learning has a hard time dealing with large state spaces, because of the table-based Q-learning used to learn a given environment. Relational interpretation is then used to extend conventional Reinforcement Learning with relational representations through first order logic, yielding the Relational Reinforcement Learning technique. The Blocks World is used as example for showing the strengths and weaknesses of Relational Reinforcement Learning. Finally, Tetris$_{LTD}$, a reduced version of the well-known puzzle game of Tetris is presented and implemented using Reinforcement Learning.

Summary

Many machine learning problems prove hard to solve due to the size of their state space. Having to visit every single state present simply is not a feasible solution. Instead, attention should be drawn towards representation schemes capable of decreasing the size of a given state space.

Such schemes are concerned with finding patterns within the state space that makes it possible to derive a generalized subspace. This subspace then works as a representation of the total state space, which again can be used in conjunction with a machine learning technique.

In this report, focus lies on finding a proper representation approach for small computer games to be learnt by means of the machine learning technique known as Reinforcement Learning. Here, table-based Q-learning is used to learn the optimal policy of a given environment.

However, the table-based approach of storing state information is only convenient for applications with a small state space. Also, in many cases more expressive ways than simple state enumeration are needed to represent a given state.

It has therefore been necessary to come up with a better representation method for states and actions as they appear in the environment. Different approaches such as propositional and Deictic representations are discussed through examples, along with structural representations such as relational interpretations and labelled directed graphs.

Relational interpretation is then used to extend the conventional reinforcement learning technique with relational representations through the use of first order logic, yielding the Relational Reinforcement Learning technique.

Relational reinforcement learning is capable of decreasing the size of a given state space through means of generalization. Using the Blocks World as example, the concept of relational reinforcement learning is introduced along with its advantages and disadvantages.

Among the advantages are the fact that relational reinforcement learning now is capable of learning a much wider array of applications, which do suffer from a large state space and the problems that follow. And to do it using a much more expressive representation.

However, there are disadvantages. Some domains are hard to describe relationally, or do not work well with first order logic results. Also, some state spaces are likely to have less frequent patterns, making it hard for relational reinforcement learning to learn the optimal policy of the environment effectively.

The well-known puzzle game of Tetris is then discussed with special emphasis on how to solve the Tetris problem with reinforcement learning techniques. Realizing the high complexity of the standard game of Tetris with seven different pieces, a reduced version, Tetris$_{LTD}$, is then implemented.

The implementation, though not fully operational, made it clear that even simple domains can be hard to learn. Currently, the application suffers from being able to select the next best optimal policy from a given state. This unfortunate feature makes the agent more willing to stack the tetrominoes than try to complete a bottom row.

Conclusively, reinforcement learning is a good choice for learning small computer games. On the other hand, relational reinforcement learning is capable of handling more complex domains. Still, more live examples of applications outside the area of games need to be developed and evaluated.

Contents

1 Introduction

Parents are often marbled by the learning abilities of their infants when growing up. Humans, as well as animals to a certain extent, have an extraordinary talent for learning by interacting with their environment and using this knowledge to handle new, unforeseen situations. A term invented by humans for this exact process is called *intelligence* which is one of the key elements in the basic foundation of all most lifeforms.

1.1 Intelligence

By exploring the natural context of cause and effect, understanding the consequences of our actions as well as the chronology of the world, we are able to use this understanding in order to benefit ourselves. This allows us to achieve a higher goal such as learning to drive a car, walk on stilts, or even learn a new language. In fact, often we do not even need an explicit teacher to learn about new things. Instead we rely on our senses to observe, feel, smell, taste and listen to the environment in order to better understand it.

Throughout our lives, we are greatly aware of how our actions effect our surroundings and as such, we often exercise a concrete behaviour that seeks to influence what happens around us. However, now and then we stumble across new unforseen situations that challenge, trick and amaze us, because we have no knowledge of, nor any experience that applies to them.

Being faced with such new situations does not always have a positive outcome. We tend to make a fool of ourselves, end up hurt, shocked, fascinated or simply intrigued by them. It is not by coincidence that sayings such as *"you cannot make an omelet without breaking a few eggs"* or *"you must crawl before you can walk"* exist, implying that it often takes a few bad attempts before getting it right. It is only human to make mistakes, because making mistakes is one way of finding the right path or solution to a given problem.

It is said that knowledge is power. And knowledge itself may be provided througout life in many different ways and through many different sources: listening to a school teacher, reading a book, observing the behaviour of people and animals, touching an electric fence, taking apart a taperecorder to study its inside, and so forth. Here, the natural curiosity of humans is often what drives us.

1.2 Machine Learning

An exciting field in the science of computers deals with learning on a machine level. For decades attempts have been made to come up with techniques for developing computer systems capable of learning and improving themselves from experience. Influenced by areas involving mathematics, philosophy, statistics, artificial intelligence, biology and many others, some attempts have been succesful, others have not.

The field of machine learning is of special interest to us humans, because the requirements to have computers help us in our daily life has grown from simple, mathematical calculating tasks to more complex ones. These could include data mining tools to help find consumer purchase patterns in large databases, learning a car to drive on its own, to classify and filter out unwanted email, and so forth.

Recently, the computer game industry has shown a lot of interest in machine learning with the sole purpose of providing challenging and exciting new games and simulators for the always demanding game playing consumers. Often, the problem with a given computer game is that once it has been played a few times, the game becomes predictable and boring. In such a case, applying machine learning techniques could help in building a computer-based opponent that adapts itself to the moves made by the human player and the current state of the game.

As with any application, being an intelligent game or a data mining tool, the problem is to find a suitable learning method. This method should be able to learn all, or at least a representative part, of a given domain, enabling the computer to handle certain amounts of input, possibly with distorted data, and produce a good end result. And more importantly, the learning method should be capable of doing it all within an acceptable time limit.

Obviously, this is almost never the case. Problem domains are seldom small, rarely simple nor noncomplicated. And coming up with a succesful solution that performs learning in an optimal way is not an easy thing to accomplish. Often, the amounts of data to be handled proves itself too big, and it can be hard to find ways of covering the entire domain space, or even representative parts of it.

1.3 Project Outline

Many machine learning problems are difficult to solve because of the size of the state space of the given application. In such a case, finding the optimal solution requires a lot of computation, because the learning system in theory has to visit every state at least once. This project is concerned with finding a proper represen-

tation method, which has enough expressive power to describe the states of a small computer game related domain. At the same time the method in question, should be able to decrease the state space without losing any state information.

1.4 Contents of the Report

This report concerns a study in the area of a machine learning technique known as *"Reinforcement Learning"*. A detailed description of this learning technique is provided in Chapter 2, which explains the basic concept, elements and terminology used. Realising a weakness of reinforcement learning in the table-based way of representing state-action information, a variety of more expressive representation approaches is presented in Chapter 3.

In Chapter 4, the standard reinforcement learning technique is extended with a relational representation of states and actions. This new, enhanced form of reinforcement learning referred to as "relational reinforcement learning" is then described using the example of the Blocks World. Also, the ACE data mining tool is presented and discussed with particular emphasis on constructing relational reinforcement learning applications.

Later, in Chapter 5, a presentation of the game of $Tetris$ is given, and Chapter 6 includes the implementation of a limited version of the game called $Tetris_{LTD}$ is described. This is done in the context of standard reinforcement learning. Chapter 6 concludes on the project work and provides ideas for further work.

2 Reinforcement Learning

Reinforcement learning (RL) is a computational approach to learning by interaction with the environment. This way of learning is a foundational idea of many theories concerning intelligence and learning inspired by human nature. The main difference between RL and other approaches to machine learning is that focus in RL lies mainly on goal-directed learning achieved through a *trial-and-error* process.

The history of reinforcement learning has its beginning within the psychological studies of animal learning[SB98a]. Combined with the 1950s research in optimal control, these studies lead to some of the earliest work in artifical intelligence in the 1980s. This work has evolved over the years into what we today refer to as "reinforcement learning".

Unlike most forms of machine learning, the learning entity in reinforcement learning is not told directly what to do. That is, to execute the optimal behaviour with respect to a given domain and a given decision problem. The learner must discover the optimal behaviour itself by exploring the world and interacting with it. At first, the learner behaves irrationally due to a lack of knowledge, but after a while it becomes smarter and better suited for selecting an optimal approach in solving the decision problem.

The optimal behaviour corresponds to a sequence of moves, or steps that leads to the answer or goal of the decision problem. A goal is reachable, because a system based on RL is rewarded whenever it has made a "good" choice and punished if not. However, the system must find its own way to achieving a reward by trying out different possibilities. Hence, RL is different from traditional supervised learning, where learning is performed by means of examples provided by an experienced supervisor.

In order to illustrate the main difference between reinforcement learning and supervised learning, the following examples can be used:

- A student watches a teacher solve exercises on the blackboard, and learns how to imitate this behavior for his own home-work (Supervised Learning)
- When picking up and handing over the newspaper from the driveway, the dog receives a treat from its owner. The dog quickly learns to repeat this behaviour (Reinforcement Learning)

The argument for not using the supervised approach is that it can be difficult to provide enough examples that represent all the different interactive situations an agent can end up in. In such cases it proves very useful if the learning entity in a reinforcement learning problem is able to learn from its own experience.

2.1 Terminology

A specific terminology is used when describing the elements of a reinforcement learning problem scenario. Here, an **agent** constitutes the learner, e.g. the abstract user that is trying to solve the reinforcement learning problem within the boundaries of a given domain. In this domain, the agent can carry out actions that affect and changes the state of it. The **environment** is an abstraction over the domain in which the agent exists, can perceive, and act in.

In the context of this report, the concept of RL involves building an agent capable of sensing as well as interacting with the environment in which it is placed. The agent must find the optimal approach, called a policy to a given problem by selecting a sequence of actions which leads to states with the greatest cumulative reward.

A **policy** defines the learning agent's way of behaving by mapping from perceived states of the environment to actions to be taken when in those states. An **action** is a well-defined atomic step or move performed by an agent, and a **state** is merely a snapshot of the state of the environment along with the current state of the agent at a given time t.

For every action an agent performs, a reward function assigns an immediate reward to the agent. A **reward** is a scalar value which represents the degree to which a state is desirable. In the simplest form using a game as example, the function could provide a positive reward when a game was won, a negative reward (punishment) when the game was lost, and provide a neutral reward of zero in any other game state.

Commonly, states as well as actions are represented using a rather simple approach such as enumeration or similar. However, some reinforcement learning problems demand the use of a better representation form. In Chapter 3 a few widely used approaches to describing information about states and actions are explained.

1	**Environment:**	You are in state 65
2		You have 4 possible actions
3	**Agent:**	I'll take action 2
4	**Environment:**	You received a reward of 7
5		You are now in state 15
6		You have 2 possible actions
7	**Agent:**	I'll take action 1
8	**Environment:**	You received a reward of -4
9		You are now in state 65
10		You have 4 possible actions
11	...	...

Table 2.1: An example of the dialog between an agent and its environment

Imagine for a second the well-known game of Tic-Tac-Toe as seen in Figure 2.1. Here the goal of the reinforcement learning agent is to fill up three adjacent cells, connected either vertically, horizontally or diagonally with identical symbols of either O's or X's. If this goal is reached, the agent has won the game.

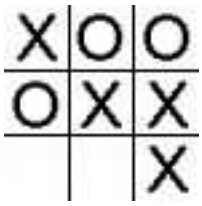

Figure 2.1: The game of Tic-Tac-Toe

Here, the enviroment would correspond to the 3 x 3 board including the 0's and X's. The two players would constitute the agents, an action could be to "put a mark on the board at position x, y", and a state would be the state of the environment after any move by an agent. In the simplest form the reward could be 100 points for filling up three horisontally, vertically or diagonally connected cells and winning, -100 for letting the opposing agent win and a reinforcement of 0 in any other state. The policy to be learned could be to find the sequence of moves that in the end, results in most winning situations.

An example of a natural language dialog between a given agent and the environment in which it exists can be seen in Table 2.1. Here, the environment presents to the agent a list of possible actions to choose between along with the current state of the agent. The agent then selects an action from the list and the environment

rewards the agent with respect to the new state.

Reinforcement learning has been used in a large variety of software solutions. It has succesfully been implemented in applications ranging from path finding systems and computer-based opponents in simple boardgames such as Tic-Tac-Toe[SB98b] and Backgammon[SB98c], to larger, industrial applications including control systems for elevators[SB98d], robotics[SB98e] and even electronic warfare for the military[KHI94].

2.2 Markov Decision Processes

A sequential decision problem can be formalized using Markov Decision Processes (MDPs). An MDP is a model of a decision problem, where an agent can perceive a given set of states in its environment. Also, the agent has access to a set of actions, from which it can select and perform its next action at each time step t. If the set S of states and set A of actions are finite, the MDP is also called finite.

The current state and action of the agent determines a probability distribution on future states. If the resulting next state only depends on the current state as well as the action of the agent, the decision process obeys what is referred to as the Markov property.

Definition 2.1 *A Markov Decision Process* **(MDP)** *is a 4-tuple, (S, A, r, δ), where S and A are finite sets and*

1. *S is the set of states,*
2. *A is the set of actions,*
3. *$r : S \times A \to R$ is the reward function, and*
4. *$\delta : S \times A \to S$ is the transistion function*

An MDP consists of a set S of states of an environment, which the agent can assume, and a set A of actions, from which the agent can choose its next move. At any time step t, the agent uses its sensory system to retrieve the state s_t of the environment, selects its next action a_t from A and executes it.

An immediate reward $r_t = r(s_t, a_t)$ is returned from the environment, letting the agent know whether the chosen action was "good" or "bad". At the same time, the environment deterministically selects and puts the agent in a new state using the transition function $s_{t+1} = \delta(s_t, a_t)$. This means that the new state solely depends on the current state and the action of the agent.

The job of the agent is now to learn an optimal policy, $\pi : S \rightarrow A$. Based on its current state s_t, it must select its next action, $\pi(s_t) = a_t$, and find the policy that produces the greatest cumulative reward $V\pi$, shown in equation 2.1 with an infinite horizon.

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{2.1}$$

The **infinite horizon** reward solution considers the long-run reward of the agent influenced by the discount factor γ, a constant weight between 0 and 1. Mathematically, using the infinite horizon model, as opposed to the finite one, is more tractable in most applications. This is because an application, in theory, can run forever. Or at least, because the lifetime length of an agent is unknown.

A **finite horizon** reward solution, $\sum_{i=0}^{h} r_{t+i}$, is an alternative to the infinite one that considers the undiscounted sum of rewards over a finite number of steps denoted by h. The finite horizon solution is applicable when the lifetime of the agent is known. The only thing the agent needs to think about is that at a given time it should optimize its expected reward for the next h steps. It does not need to worry about what will happen afterwards. However, the problem here is that the actual lifetime length of the agent may not always be known in advance.

As an alternative to finite and infinite horizon rewards, the **average reward** model can be used. The expression $\lim_{h\rightarrow\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$ considers the average reward per time step t over the lifetime of the agent. Again, the precise length of the lifetime of the agent has to be known in advance, before the average reward solution can be taken into consideration.

In this report, a restriction to using the infinite horizon is made, where the reward is discounted by the factor γ. Future rewards are often discounted more than immediate ones, since it is only natural to seek to be rewarded sooner as opposed to later. The more the future matters, the higher the value of the discount factor.

If $\gamma = 0$ only the immediate reward is taken into consideration. Otherwise, rewards are discounted exponentially by the factor γ^i where i denotes a given time step into the future. If $\gamma < 1$ the greatest cumulative reward $V\pi$ is known as the "discounted cumulative reward".

The optimal policy that maximizes $V^{\pi}(s)$ for all states s is denoted by π^* :

$$\pi^* \equiv \arg\max_{\pi} V^{\pi}(s), (\forall s) \tag{2.2}$$

To simplify notation the value function $V^{\pi^*}(s)$ of an optimal policy is often denoted $V^*(s)$. In other words, this is the reward an agent will receive if following the optimal policy beginning at a given state s. The requirement here is, however, that the transistion function as well as the reward function is known to the agent. This is not always the case. Fortunately, Q-learning is helpful here.

2.3 Q-learning

Q-learning is a reinforcement learning algorithm for learning how to estimate long-term expected reward for any given state-action pair, where the reward function r and the transistion function δ are unknown to the agent. One of its advantages is that it does not need a model of the environment and as such, it is applicable for on-line learning situations. It is, however, dependent on the size of the state space since concentional Q-learning uses a look-up table as representation of it. Table-based Q-learning is therefore mainly feasible for problems of a smaller scale.

An evaluation function $Q(s, a)$ is used to retrieve the values of the state-action pairs. Using this function, it is possible to find the maximum discounted cumulative reward achievable by beginning in state s and selecting a as the first action. Here, the value returned from Q is the reward given when executing the action, together with the value of following the optimal policy afterwards, discounted by γ:

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

The equation is rewritable with respect to the similarities to part of the equation from earlier. This means that a given agent is capable of selecting a global, optimal action a despite lack of knowledge of the reward and transistion function, and despite using only the local values of Q. All it has to do is to learn the Q function, and use it to select the maximum-valued a in any given state s. $Q(s, a)$ can be rewritten into the following:

$$\pi^*(s) = \arg\max_a Q(s, a) \tag{2.4}$$

Figures 2.2, 2.3, 2.4 and 2.5 illustrates an example of solving a deterministic MDP using Q learning. Here, the environment used is a small grid world, where each cell in the grid depicts a state. The arrows pointing from one state to another represents the actions, which the agent can select in order to change between the states of the world. The cells in the grids marked G corresponds to the goal state of the world. In this example the goal state is referred to as "absorbing", since it does not have any transistion arrows leading away from it.

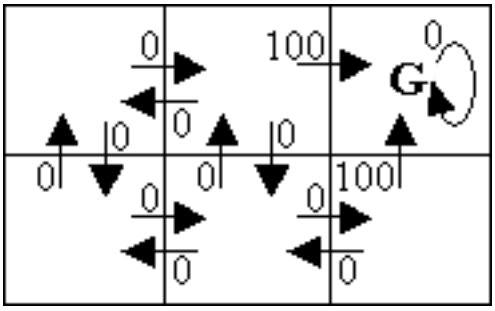

Figure 2.2: $r(s, a)$ values (immediate reward)

In Figure 2.2 each arrow is associated with a number. This represents the immediate reward $r(s, a)$ an agent would receive when changing between the two states in the direction of the arrow, executing the action stated by the arrow. In this world, the only reward given is when the agent selects an action that leads directly to the goal state. Every other transition between states is rewarded, or "punished", with a value of zero.

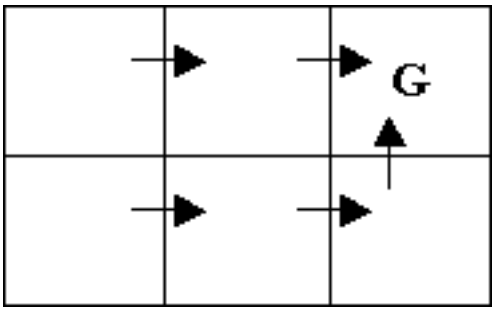

Figure 2.3: An optimal policy

Given a discount factor $\gamma = 0.9$, the optimal policy π^* as well as its corresponding value function $V^*(s)$ can be determined. Figure 2.3 shows what an optimal policy could look like in this case in any given state of the grid-world. This corresponds to the agent selecting the optimal "path" that will lead it to the goal state.

The values of V^* for each state can be seen in Figure 2.4. Here, each cell now holds a discounted reward according to the optimal policy in that particular state. In the case of the bottom right state in the grid, the value 100 is the immediate reward received from selecting the optimal policy, which in this state is the action to "move-up". Using the bottom center state, the optimal policy to reach the goal state G is first to "move-right", receiving the immediate reward of zero, and then to "move-up", generating a reward of 100.

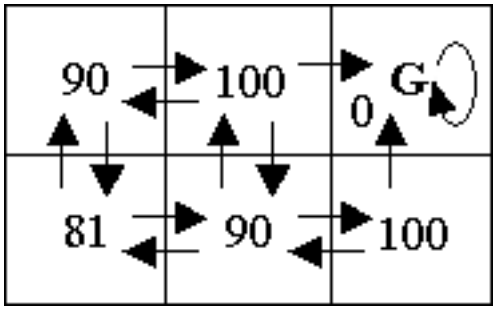

Figure 2.4: $V^*(s)$ values

The actual calculation is the sum of discounted future rewards over an infinite future is :

$$0 + \gamma 100 + \gamma^2 0 + \gamma^3 0 + ... = 90 \tag{2.5}$$

Figure 2.5 illustrates the Q values for any state-action transition in the grid-world example. This corresponds to adding the value of r for the particular transistion and the value of V^* for the resulting state together, discounting them by the factor γ. The optimal policy here is the same as selecting the actions with the maximum Q-values in any state in the grid.

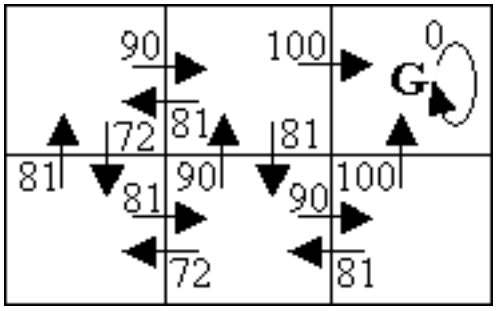

Figure 2.5: $Q(s, a)$ values

2.3.1 The Q-learning Algorithm

The algorithm for learning the Q-function uses iterative approximation in order to learn the optimal policy. Approximation is needed, since only the sequence of immediate rewards r is given. The relationship between Q and V^* is, however, very useful in finding a reliable way to estimate the training values for Q. Using :

$$V^*(s) = \max_{a'} Q(s, a') \tag{2.6}$$

yields a rewriting to the following, recursive definition of Q:

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a') \tag{2.7}$$

With this equation it is possible to use iterative approximation in learning the Q-function. The actual algorithm using a pseudo code based notation, can be seen in Table 2.2. The symbol $\hat{Q}$ refers to the approximation of the actual Q-function and is represented as a look-up table. Each entry in the table corresponds to a state-action pair $\langle s, a \rangle$, in which the value for $\hat{Q}(s, a)$ is stored. The look-up table therefore holds the current hypothesis for each actual, yet unknown value of $Q(s, a)$.

1	**for** each state-action pair (s, a) **do**
2	set current table entry $\hat{Q}(s, a) = 0$
3	observe current state s
4	**do** forever
5	choose an action a and execute it
6	receive immediate reward r
7	observe new state s'
8	update $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$
9	$s \leftarrow s'$

Table 2.2: The standard Q-learning algorithm

Initially, each entry in the table is reset to zero. During each iteration, the agent perceives its current state s and selects an action a. After executing a, the agent receives $r = r(s, a)$ from the reward function as well as $s' = \delta(s, a)$ from the transistion function. Next, it updates the current hypothesis $\hat{Q}(s, a)$ using the following update rule:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a') \tag{2.8}$$

2.3.2 The Action Selection Problem

An important aspect of reinforcement learning lies in selecting an experimentation approach that produces most effective learning. Here, the agent indirectly affects the effectiveness through the distribution of the training examples since the actual

sequence of actions determines the immediate succes. A problem, known as the *exploitation/exploration problem*[SB98a] deals with the dilemma of choosing between having the agent focus on two different experimentation approaches, namely that of exploitation or exploration.

The exploitation approach is important when the agent seeks to maximize its cumulative reward. This is true because the agent is forced to exploit what it has already learned. That is, to visit states and select actions that it already knows will provide a high reinforcement. On the other hand, using the exploration approach will give the agent the opportunity to explore unknown states and actions. This approach will make the agent focus on exploring the environment in order to gather new information, while hoping to find a state or action with a, so far, undiscovered high reward.

Selecting an action a in state s is commonly done probabilistically in Q-learning instead of just having the agent select the action that maximizes $\hat{Q}(s, a)$. The problem here is that the agent will begin a tendency of exploiting its current approximation $\hat{Q}$, hence favoring early states and actions that the agent already has learned will provide a reward.

On the other hand, it is not favorable for the agent to have too much focus on exploring new states and actions, seeking rewards in the form of a higher Q-value. A balance between the risk of favorizing either exploitation or exploration can be made using a probabilistic approach in solving the action selection problem:

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}} \tag{2.9}$$

A probability is assigned to actions based on the Q-values, and no action must be assigned a probability of 0. A nonzero probability ensures that the action can in fact be chosen, since an action with a probability of zero is uninteresting to the agent. In the abovementioned equation, $P(a_i|s)$ yields the probability of selecting action a_i given the current state s.

A constant denoted k where $k > 0$ is included to help determine how strongly the action selection process favors actions that have a high Q-value associated with them. If k holds a high value, the agent will tend to exploit what it has already learned, because actions with above average $\hat{Q}$ will be asigned a high probability.

Contrary to this, a small value of k will assign a high probability to other actions, causing the agent to explore the ones currently holding a small Q-value in the hope of finding a higher.

An alternative to a constant k-value, k can be adjusted with the number of itera-

tions. This will allow the agent to change its behaviour during its lifetime. In some cases it is favorable for the agent to start out using an exploration approach and later on, to focus more on exploitation.

Using this particular strategy makes sense in applications where a more natural process of learning is requested. When dealing with a new domain, it seems only logical to initially to explore the boundaries of it in order to discover and learn the overall environment. Later on, when the basic surroundings are known to the agent, it should try to explore new things and learn details about it.

In the following chapter, a series of different approaches to representing state and action related information is presented and discussed. Choosing a proper representation method is an important part of the development of an application, because it often directly affects its success. That is, how well the application in question is able to solve a given problem.

3 Representing States and Actions

When dealing with reinforcement learning problems, it is necessary to find a proper way of representing the states and actions involved. Any representation form can be as good as the next one, as long as it represents the domain in question, has acceptable performance and sufficient expressive power to help clarify and solve the problem given.

In the following, a few methods applicable to representing states and actions are explained according to their level of expressiveness, beginning with the low level ones. These representation forms[Dri04a] include :

- State Enumeration
- Propositional Representations
- Deictic Representations
- Structural Representations

3.1 State Enumeration

In traditional reinforcement learning[1] states are usually represented using simple state enumeration. This representation form has a very low level of expressive power since a state merely is represented and identified using a unique, numerical index value. Still, it is a sufficient choice in many situations where little information about states and actions and their context is kept and reused.

State enumeration yields the use of anonymity in the sense that the individual states and their unique role in solving the reinforcement learning problem is of little interest. Instead, states are stored in table-based form, where access is gained through

[1] Reinforcement learning using table-based Q-learning

the use of their index. Here, knowing the index suffices and no direct interpretation of states or actions is needed.

The main strength of using state enumeration is simplicity. At the same time, simplicity is also the main weakness. This is because it may prove too simple to use in some applications that does not settle for identifying states and actions using just simple numbers. In Table 2.1 the dialog between the environment and the agent is based on a represention form like state enumeration.

3.2 Propositional Representations

Propositional representation is a representation form that corresponds to describing a state as a so-called feature vector. This vector then holds an attribute for each possible property of the environment of the agent. An attribute here could be Boolean, an enumerable range, a continuous value, and so forth.

The game of Tic-Tac-Toe, as seen in Figure 2.1, is a good domain for making use of a propositional representation form. Each of the nine cells can either be empty or hold the X or O-symbol. This simple domain can be represented using a feature vector of length nine, where each attribute $att \in \{empty, X, O\}$.

Using Figure 2.1 the feature vector is $f = \{X, O, O, O, X, X, empty, empty, X\}$ when representing the Tic-Tac-Toe board from left to right, top to bottom. The X-agent could then use this representation form to learn that placing a X in the center of the board with $f = \{X, O, O, O, empty, X, empty, empty, X\}$ would make it win the game.

Also, the agent could use the feature vector to determine the positions of the board that makes it impossible for it to win. An example here could be a situation $f = \{?, O, ?, ?, O, ?, ?, empty, ?\}$, where ? denotes an insignificant attribute value, independent of the outcome. In any case, the X-agent would lose and receive a negative reward if it left such a board configuration to the opposing player. Obviously, the agent should here learn how to avoid ending up in such a situation.

A weakness of using a propositional representation approach is that a problem arises when trying to describe attribute properties and relations that may/may not exist between different states. An example here could be the problem of not being able to represent that not placing an X next to *any* two adjacent O's would result in the agent losing the game. Also, propositional representations fail when applied to a dynamic domain where the number of objects being represented is changed over time, or unknown at first.

Propositional representation methods can be used in situations, where a more ex-

pressive representation form than state enumeration is required. It can be used in smaller domains where state blacklisting is an acceptable way of expressing, for instance, undesirable states. Blacklisting here solely refers to the concept of keeping track of all the states in the state space that present a similar situation. Blacklisting states may work in domains with limited size state spaces. However, a more generalized approach is more convenient and scales better.

In the simple Tic-Tac-Toe example, the agent needs to learn any combination of two adjacent O's in order to have a complete blacklist of the possible threats. In the end, the blacklist would here consist of all the 16 different combinations of placing two adjacent O's on the board. The basic rule for any of these combinations is, however, exactly the same: *"Any combination of two adjacent O's should be avoided"*. This simple rule yields the use of a more general approach.

3.3 Deictic Representations

A deictic representation form deals with representing a varying number of objects in a dynamic environment. Basically, it offers a solution to the problem exposed in propositional representation methods, by providing the agent with a *focal point*. This focal point is then used to define the rest of the environment, e.g. the environment is defined in relation to the focal point.

This particular approach is very similar to what most people do in many situations in real life. A good example here is giving directions to someone, because the world here is described in relation to where the person providing directions is standing at that exact time. Providing the person who is lost with a deictic representation of the environment might not even include specific street names. It could be based on constructs such as the following:

- Two floors up
- The second crossroad
- The street on your left

Such direction constructs only make sense when applied in relation to the focal point in question. The destination point would clearly differ if the exact same set of directions were to be provided in two different locations. As such, the focal point is similar to the starting point of the agent. Or, the end point of following a previous set of directions. Deictic representations can also be used in the description of objects:

- The last person you talked to
- The glove on your right hand
- The movie you are watching

Although the deictic approach in representing states and actions may seem a natural choice for many applications, it suffers from the problem of complexity. In basic Q-learning, the agent has to explore the entire state space of the environment by means of state-action pairs. If a deictic representation of the state space was used, every possible focal point in the environment also had to be explored, hence causing a substantial increase in the complexity of the given learning problem.

3.4 Structural Representations

The main idea behind structural representation methods is that the real world is filled with relationally connected objects, each displaying certain properties. So, in order to fully describe a relational world, a relational presentation of it must be deduced, involving the available states and actions as they appear to the agent in the environment.

Role playing games are excellent applications for using structural representations. Typically, the player here controls a dynamic amount of characters with different characteristics, e.g. belong to a certain race, possess certain abilities, strengths and weaknesses, and so forth. The job of the player is to develop these different characters by, among others, leading them into battle, gather helpful objects and complete certain quests.

A role playing game presents a very complex world to the reinforcement learning task. The requirement here is to come up with a suitable representation form that is capable of describing not just basic state and action information, but also the many different objects, their characteristics and individual relationships.

The complexity of describing the battle part of a role playing game could involve:

- Dynamic character amount (some characters die while others are born during game play)
- Unique characters (characters are of different types and have a different number of abilities)
- Individual character behaviour (the behaviour of a character is dependent on the situation, current abilities etc)

- Relative character strength (a character can be stronger against certain types of enemies)
- Generic actions (a magic spell might have multiple targets)

These features are very difficult to represent using any of the representation methods mentioned so far without ending up with a lossy description of the game states. In order to reach an acceptable, lossless representation level with enough expressive power to describe a role playing game, a relational approach can be used.

3.4.1 Relational Interpretations

Using a relational interpretation approach involves representing each state-action pairs as sets of relational facts. The notation used here is very different from the ones mentioned earlier because a high-level representation language is used in the description of an environment, e.g. objects, states and actions.

Consider the small domain of the package delivery robot as seen in Figure 3.1. The task of the robot is to deliver the packages to their individual destinations as quickly as possible. The robot is capable of carrying several packages all at once, dependent on their accumulated, physical size.

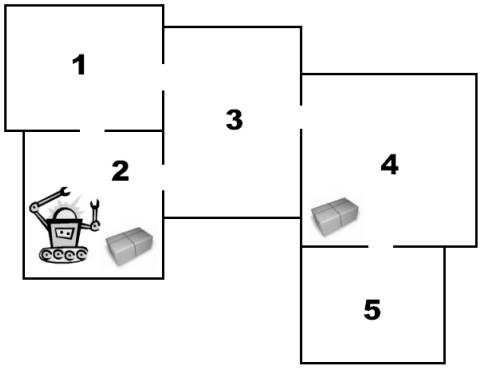

Figure 3.1: The package delivery robot domain

The delivery robot carries navigational equipment to help it find its way round the rooms of the building in question. At random intervals, a package may appear in any of the rooms for the robot to pick up and deliver elsewhere. The set of actions available to the robot consists of $A = \{move(D), pickup(P), dropoff(P)\}$, with the set of directions $D = \{North, South, East, West\}$ and the set of packages $P = \{p_1, ..., p_n\}$.

The relational facts used in the representation respect the reasoning of First-Order Logic[Mit97]. In first-order logic each statement is a construct which basically can be broken up into a predicate and a subject. The predicate defines or modifies the properties of the subject. The form $P(x)$ is a construct of the predicate P and the subject x, here represented as a variable.

Other useful expressions include $F(x)$, where F is a function instead of a predicate. In first-order logic, the difference between a predicate and a function is that a predicate can only take on values of true or false. A function, however, may take on any value.

Table 3.1 shows the relational facts about the current state of the robot as depicted in Figure 3.1. Each fact is a construct consisting of the type of relation, and one or more embedded variables. The variable r followed by a number denotes a given room, while p followed by a number denotes a certain package.

The relational interpretation shown in Table 3.1 provides enough state knowledge to describe the current location of the robot as well as that of each package, the individual size of the packages, and the maximum loading capabilities of the robot.

1	location(r2).	destination(p1,r3).
2	carrying(p2).	destination(p2,r4).
3	maximumload(5).	destination(p4,r3).
4		
5	package(p1).	size(p1,3)
6	package(p2).	size(p2,1).
7	package(p3).	size(p4,3).
9		
10	location(p1,r4).	
11	location(p2,r2).	
12	location(p4,r2).	

Table 3.1: A relational interpretation of the state of the delivery robot in Figure 3.1

In this case, the navigational equipment is aware of how the individual rooms are connected. This information, however, could easily be represented using relational facts such as $connected(r1, r2)$., $connected(r1, r3)$. and so forth. Whenever a new package arrives in a room, new facts concerning its location, destination and size is simply added to the current representation.

A big advantage of using relational interpretation is that of generalization. Using simple relational facts even a complex world of objects, states and actions can be described. Furthermore, the high level description language can easily be read and understood by humans, providing the developer with a better overview of the

reinforcement learning problem.

Also, this approach scales rather well as opposed to other representation forms. Adding new state information is simply a matter of providing new facts to the current representation. Another advantage is that it is possible to derive new facts from current ones without explicitly adding them to the representation. This is explained in more detail in the part of Section 4.2 of Chapter 4 that deals with the concept of Logic Programming.

3.4.2 Labelled Directed Graphs

Another possible representation method for describing relational information is through the use of a graph. A graph is a useful format for displaying structural information between objects in a given domain. Here, a node in the graph represents the object, while an edge between two nodes describes their relationship.

A common use of graphs is in applications involving navigational tasks, where the graph becomes a representation of a roadmap or similar. In such a case the nodes could represent location points in the world, and the edges could represent a path or road connecting the location points to one another. The agent is then able to find its way round the environment, using the graph as a directional map.

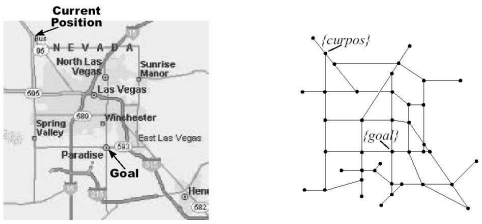

Figure 3.2: A road map and its representation as a graph

One-way streets could be represented using directional edges in the graph, and any additional travelling information such as speed limits could be supplied using a labelled graph. Figure 3.2 shows a piece of a Nevada road map along with its representation as a labelled graph. The agent is positioned at the node labelled $\{curpos\}$ and must select the optimal path which will lead it to the goal node labelled $\{goal\}$.

The graph approach is an exciting alternative to relational interpretation. Though a graph seems to be closer to the language of a computer, it is still able to provide a developer with good overview. The main differences between relational interpretation and a graph representation include maintenance and the ability to scale. Though it may seem a bit more complex to adapt a graph to a new or changing

environment, the graph does not need to be interpreted like in the relational interpretation approach.

The following chapter introduces the area of relational reinforcement learning, a variant of reinforcement learning that uses generalization of state space information through relational representations. This feature makes the learning method applicable for many applications that suffer from large state spaces.

4 Relational Reinforcement Learning

Using relations in the description of a given domain seems obvious in many situations. As an example, an intelligent vacuum cleaning robot and its environment can be used. Here, it might not be interesting for a robot just to know its exact location, i.e. its XYZ-coordinate in the three-dimensional world. More importantly could be the relations that exist between the robot and the environment in which it exists at a given time.

For instance, it might be crucial for the robot to know that it currently is operating behind a table in the middle of the room located at the left at end of the hall. Or that the charging facility of the robot is located behind it, as opposed to in front of or to the left/right of it. Being aware of absolute positions in a room where furniture is moved around may cause great confusing to the robot. Also, the robot itself might not always know its actual starting position. In reality, a hybrid solution is most likely to be used.

Relational reinforcement learning (RRL) is an alternative to conventional reinforcement learning that uses a different form of representing the Q-values than that of a simple, tabular one. Through the use of relational descriptors and generalization, RRL is able to decrease the size of the state space. This particular approach makes reinforcement learning better suited for applications that have to deal with a large state space and the problems that follow.

In the following, a description of the basic Blocks World domain is presented. This domain will serve as example throughout the report to illustrate the concept of relational reinforcement learning.

4.1 Introducing the Blocks World

The Blocks World comprises a domain of floor along with a constant number of blocks that can be either stacked or unstacked. A block in the world can either be on top of another or be on the floor. In this simplistic used, the set B of blocks available to the agent is $B = \{a, b, c\}$. It is assumed that the blocks are of similar size and shape. Also, a stack can only be neatly built, i.e. it is not possible to place a block on top of two or more neighboring blocks.

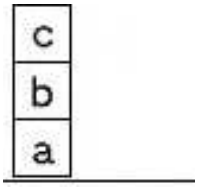

Figure 4.1: An example of a stack of blocks in the blocks world

Here, a relational representation of states becomes obvious, because a block object can be described using its position in the stack relative to its neighboring objects. Using Figure 4.1 as a reference, the stack could be described relationally (in natural language) as depicted in Table 4.1.

BLOCK	DESCRIPTION
a	"on the floor" and "below b"
b	"on top of a" and "below c"
c	"on top of b" and "below (none)"

Table 4.1: A relational description of the stack of blocks in Figure 4.1

The number of possible states available in the Blocks World with just the three blocks a, b, and c is $3! + 3! + 1! = 13$ as seen in Figure 4.2. The arrows represent transistions used to move between the different states. Important here is that duplicate and mirrored configurations such as the concrete order of blocks on the floor, or blocks moved to the floor on either the lefthand or righthand side of the stack, are excluded from the problem description.

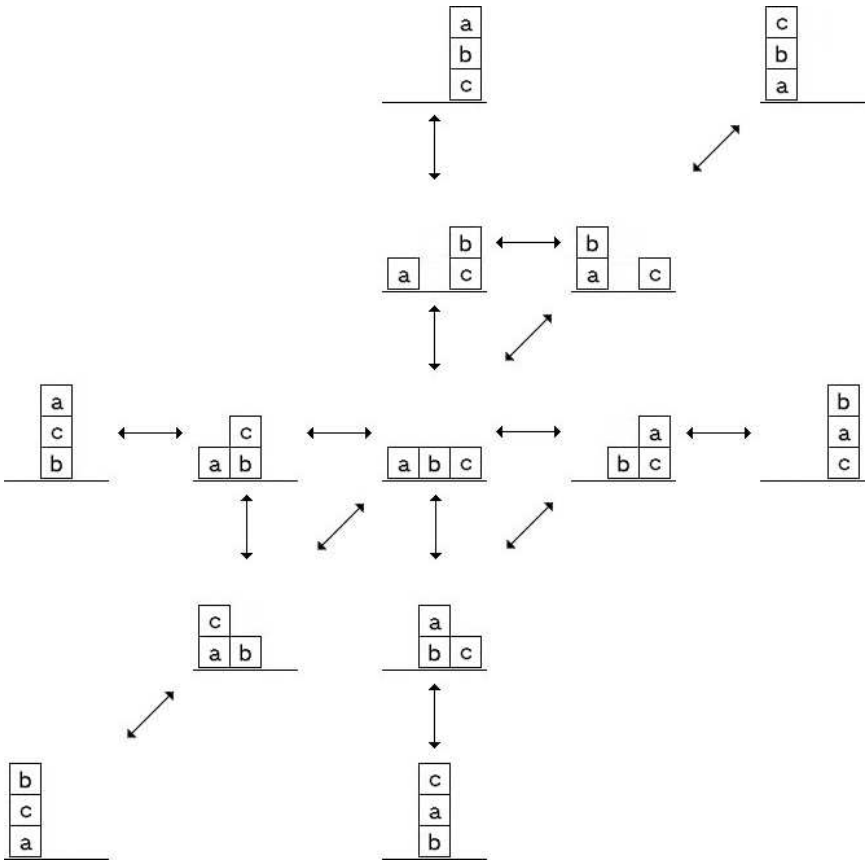

Figure 4.2: The 13 different configurations of stacking 3 blocks in the Blocks World

4.2 Generalization

The main idea in relational reinforcement learning is to decrease the size of the state space through the reuse of generalized state information. This approach, however, is only convenient when working with domains where information is in fact reusable and can be described using a relational language or similar.

For instance, the results of learning how to stack block a on top of block b, would be similar to the one of stacking block b on top of a. Also, generalization could be feasible if going from a block domain with only three blocks to a domain with four or more. Obviously, the full generalization approach is mainly usable if the different blocks used have the same properties, e.g. their size and shape is exactly the same and the basic rules that make up the world remain unchanged.

In relational reinforcement learning generalization over states is essential for build-

ing applications that are able to perform well when dealing with state spaces of a considerable size. Generalization of state and action information in a domain is possible if the state space contains patterns that can be generalized and reused.

If the states and actions share the same set of relations such as in the example of stacking block a on b and block b on a respectively, generalization is most likely to be feasible. The patterns can be exploited by describing state information relationally, as known in the field of *logic programming*.

Logic programming[Spi02] is a declarative and relational style of programming in which facts and relationships between variables can be described using boolean statements called predicates. Besides simply evaluating the predicates, these can be used to infer new facts about the variables in question. The simplistic example given below states two facts about the variables *human* and *socrates* which can be used to infer a third: Socrates is mortal.

1. Socrates is human
2. All humans are mortal

A more detailed description of the logical programming paradigm will not be provided here. Relational reinforcement learning mainly uses the predicate approach from logic programming as a way of representing information about states as well as actions.

4.3 Representing States And Actions

Relational reinforcement learning uses the *relational interpretation* approach (see Section 3.4.1) in the representation of information concerning states and actions. In RRL, a state is described and represented as a set of basic facts that hold in the state. In the case of the Blocks World example, a fact could be the predicate $on(a, b)$, hence implying that a currently is positioned on top of b. The facts (presented in Prolog[1] syntax) concerning the state s of the stack depicted in Figure 4.1 would be:

$$s = \{on(a, floor), on(b, a), on(c, b), clear(c)\}$$

[1]Prolog is a programming language based on the logical programming paradigm

Here, $on(a, floor)$ proclaims that block a currently is positioned on the floor, $on(b, a)$ that b resides on a, $on(c, b)$ that block c can be found on top of b and $clear(c)$ refers to the fact that no other block currently is placed on top of block c. Combined, this set of facts provides a relational snapshot description of the state of the stack at a given time.

The $below$ relation mentioned earlier in the natural language example in Table 4.1 is discarded here, since it does not bring forth any new information about the relationship between two blocks in a stack. Using the on and $clear$ predicates is enough to represent the configuration of a given stack. Here, as an example, the predicate $on(b, a)$ implicitly expresses that if block b is on top of block a, the predicate $below(a, b)$ stating that block a is below block b, can be derived.

The set A of actions available to the agent in the world with three blocks consists only of a single action, $A = \{move(x, y)\}$, where $x \in B$ and y is either a block or the floor. Also, $x \neq y$ in order to ensure that a block cannot be moved onto itself, e.g. $move(a, a)$.

The actions of A are also represented relationally and covers the possible actions exisiting in a given domain. Using the Block World again, an action could be $move(a, floor)$, which moves block a from a stack of blocks to the floor. When encountering a given state, only the actions currently available in it can be seen by the agent. Also, not every action may be applicable in a given state if the overall domain rules forbid it, e.g. trying to move a block which cannot be moved because it currently is placed beneath another one.

In relational reinforcement learning, a given agent can execute an action a in state s if the preconditions of executing a in s are satisfied, i.e. $pre(s, a) = true$. Definitions involving *preconditions* as well as the *effects* of actions has to be supplied along with the relational representation of state-action pair information in order to check and control the dynamics of the environment.

Table 4.2 shows a piece of Prolog code containing the domain rules of the Blocks World, where the predicate pre is used to define the preconditions for the action $move(X, Y)$ with variables $X \in \{a, b, c\}$, $Y \in \{a, b, c, floor\}$ and $Y \neq X$. The predicate $delta(S, A, S1)$ representing the relational transistion function δ defines the effect of executing the action $move(X, Y)$. If $\delta(S, A) = S1$ then $delta(S, A, S1) = true$. The transition function can be seen in Definition 4.1.

If the state of the stack is $s_1 = \{on(a, floor), on(b, a), on(c, b), clear(c)\}$ as seen in Figure 4.1 and the agent selects the $move(c, floor)$ action, the precondition $pre(s_1, move(c, floor))$ in line 3 is evaluated. This precondition checks that the statement $holds(s_1, [clear(c), not\ on(c, floor)])$ can be evaluated as true. In the case of the action $move(c, floor)$, the predicate holds because $clear(c) = true$ and $not\ on(c, floor) = true$.

```
1   pre(S,move(X,Y))                 :-  holds(S,[clear(X), clear(Y),
                                         not X=Y, not on(X,floor)]).
2   pre(S,move(X,Y))                 :-  holds(S,[clear(X), clear(Y),
                                         not X=Y, on(X,floor)]).
3   pre(S,move(X,floor))             :-  holds(S,[clear(X), not on(X,floor)]).
4
5   holds(S,[]).
6   holds(S,[ not X=Y | R ])         :-  not X=Y, !, holds(S,R).
7   holds(S,[ not A | R ])           :-  not member(A,S), holds(S,R).
8   holds(S,[A | R])                 :-  member(A,R), holds(S,R).
9
10  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).
11  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).
12  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).
```

Table 4.2: Prolog definitions of preconditions and effects of actions in the blocks world

The $delta(s_1, move(c, floor), s_2)$ in line 12 defines the state situation after the action $move(a, floor)$ has been executed from state s_1. This statements becomes true, if the predicate to delete the current predicate $on(c, b)$ from the state information is true, and if the predicate to add the new predicates $on(c, floor)$ and $clear(b)$, as well as change the state s_1 to s_{1+1}, are true.

4.4 Relational Markov Decision Processes

In regular reinforcement learning the decision learning problem could be solved using an MDP. The problem of solving a relational reinforcement learning problem is very similar, except for the two different ways of representating state-action information. Therefore, the basic MDP can be extended to a Relational Markov Decision Process (R-MDP).

Definition 4.1 *A Relational Markov Decision Process* **(R-MDP)** *is a 5-tuple, (S, A, K, r, δ), where S, A and K are finite sets and*

1. *S is the set of states represented in a relational format,*
2. *A is the set of actions represented in a relational format,*
3. *K is an optional set of background knowledge in a relational format generally valid about the environment,*
4. *$r : S \times A \rightarrow R$ is the reward function, and*
5. *$\delta : S \times A \rightarrow S$ is the transistion function*

Like a regular MDP, the R-MDP consists of a set S of states that the agent can assume, and a set A of actions, from which the agent can select its next move at a given time t. Also, the reward function r and the transistion function δ are no different from a standard MDP. The goal is still to find the optimal policy $\pi : S \rightarrow A$ that will provide the agent with the greatest cumulative reward possible.

The set K is, however, different from regular MDPs. Here, extra background knowledge helpful in solving the relational reinforcement learning task can be supplied. Every piece of background knowledge needs to be provided in relational form and could include predicates concerning the size of the stack, the number of blocks used in the domain, the number of individual stacks, and so forth.

4.5 Goal States And Rewards

In the Blocks World there are three different goal states for the agent to reach during gameplay:

1. **Stack** all blocks (using one big stack)
2. **Unstack** all blocks (move all blocks from the stack to the floor)
3. **Put** a specific block on top of another

If the agent manages to reach a goal state, the reward function $r : S \times A \rightarrow R$ will grant the agent a reward of 100. The agent will receive a reward of 0 for any other action that does not result in the agent reaching a goal state.

An *episode* constitutes the sequence of action and state changes, decided by an agent, until a given goal state is reached. In the Blocks World example, the syntax used for representing a goal state is $goal(on(x, y))$. That is, to reach a state where block x is on top of block y.

4.6 Relational Q-learning

Dealing with an infinite or very large state space creates a problem when using tabular Q-learning. Though some sort of indexing method could be used to increase performance, keeping a large table is seldomly feasible. The problem here is the size of the look-up table needed to represent the Q-values as the number of actions and states increase:

$$Q\text{-table size} = \text{number of states} \times \text{number of actions per state}$$

Relational reinforcement learning accomodates this problem by trying to minimize the state size problem through the use of *generalization*. In its basic form, tabular Q-learning is used strictly for storing Q-values. Being just a "container" of numerical values, a look-up table does not hold the expressive power needed to represent the Q-values in a more general form.

The idea of relational reinforcement learning is to use a relational approach in representing, storing, retrieving and updating the $Q(s, a)$ values. This is achieved through the use of a so-called Q-tree[DRD01], which stores general state-action information using a tree representation. The Q-tree is then used to learn a P-tree, which is an abstraction of the Q-values and only represents the policy.

4.6.1 Regression Trees

In relational reinforcement learning, the Q-tree is used to generalize over state and action information supplied in a relational format. This way, it is no longer necessary to retrain everything over from scratch if smaller changes are made to the domain, e.g. increasing the number of blocks from three to four, or changing the overall goal from $goal(on(a, b))$ to $goal(on(b, a))$. Also, using the Q-tree enables the system to cover a large state space in a more optimal way than with table-based Q-learning, because it partly reuses experience.

The Q-tree used to represent the Q-values is a so-called *relational regression tree*[Dri04b]. A regression tree is a variant of a decision tree[Jen01], which is a

common way of representing a decision making process. Despite similarities with the physical structure of a decision tree, a regression tree is designed to approximate real-valued functions instead of being used for the common decision tree purpose of classification.

Definition 4.2 *A Relational Regression Tree* **(RRT)** *is a binary, 3-tupled decision tree variant (D, T, O), where D and T are finite sets of nodes, O is a finite set of Boolean*[2] *decision test outcomes and*

1. *D is the set of decision nodes containing tests in a relational format,*
2. *T is the set of terminal nodes containing Q-values in a relational format, and*
3. *$O \rightarrow [yes, no]$*

In machine learning, regression is concerned with finding/approximating a real-valued target function that fits a given set of observations, e.g. to construct a model of a process using examples of that process. In the case of relational reinforcement learning, the model used is the relational regression tree mentioned above. The term "relational" is used, because the information stored in the tree is provided using a relational representation approach.

The structure of a regression tree is based on a hierarchy of nodes and is built using three basic components: *decision nodes*, *terminal nodes* and *decision test outcomes*. Each decision node in the tree contains a logical test with the outcome *yes* or *no*. A terminal node comprises a leaf in the tree that holds the prediction of the model, i.e. a numerical value.

The nodes of a regression tree are connected through possible outcomes of the decision nodes that connects two nodes to each other. Each decision test outcome leads to a lower level of nodes in the tree model until a terminal node is finally reached. A terminal node is a significant part of a regression tree model since the prediction of it is located there.

Any path followed from root to leaf is hence a conjunction of tests that works as representations of subareas of the overall regression surface being approximated. For each of these subareas different values of the goal (a Q-value) is predicted. The set of subareas obtained by the regression tree should be mutually exclusive, so that each training example only falls into one of these areas.

An example of a relational regression tree used in the Blocks World can be seen in Figure 4.3. Here, a decision test is represented as a square containing a test

[2]The notation 'yes/no' is used instead of 'true/false' to represent the outcome of a decision node test

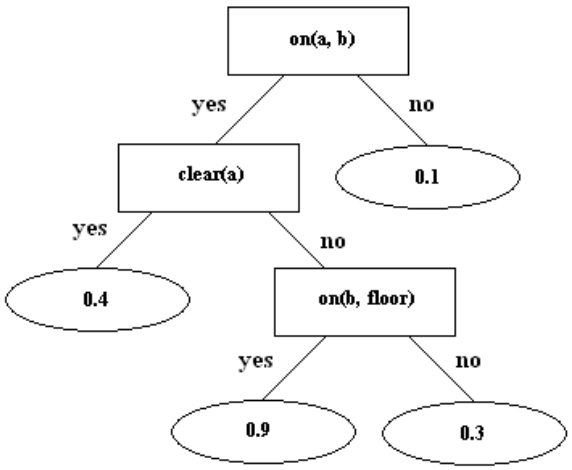

Figure 4.3: An example of a relational regression tree (a Q-tree) as used in the Blocks World

in a relational format. The connections between two nodes labeled 'yes/no' are the possible outcomes of the decision test in the topmost node of the connection. The elipses comprises the leaves of the tree and are terminal nodes that contain the prediction of the model.

4.6.2 Learning the Q-tree

In relational reinforcement learning, the values to predict using the regression tree are the Q-values in the form of a Q-tree (see Figure 4.3 again). The actual construction of the tree is achieved through the use of a regression algorithm which as input receives state-action information organized in a specific format. This input, conveniently referred to as an *"example"* covers the state of the agent, the action chosen and the related Q-value along with any background knowledge such as the goal or similar. Table 4.3 shows the contents of a random piece of input.

Q-value	Action	Goal	Facts
qvalue(0.81)	action(move(c,floor))	goal(on(a,b))	on(a,floor) on(b,a) on(c,b) clear(c)

Table 4.3: The different parts of the input to the Q-tree

Once provided to the Q-tree, the training example is sorted down through the Q-tree. Starting at the root the path through the tree is given by acting according to the result of the tests in the decision nodes encountered on the way. This process continues until a terminal node is reached and the corresponding Q-value is updated.

A test in the Q-tree is performed by running the example input containing Prolog-facts such as $on(a, b)$, $clear(b)$ and so forth against the test in the decision node encountered. The decision node test which basically is a Prolog-query then executes the test and looks at the result. If the test failed, the example is sorted down the *no*-branch of the relational regression tree. If the test was succesful the *yes*-branch is chosen instead.

Whenever a training example cannot be fully sorted, e.g. no terminal node for it exist, the Q-tree is expanded with a branch of new decision nodes and a new terminal node corresponding to the contents of the example input. The tree is considered to be complete or learned, when no new branches need to be added. That is, when the tree is capable of succesfully sorting any new example provided without making any changes to the physical structure.

The learning approach mentioned above is, obviously, a poor choice for large scale domains. Here, every new example is rather "carelessly" inserted into the tree until it ends up covering the entire state space. In other words, the result is now pretty much similar to table-based Q-learning except for the relational regression tree representation of Q-values. Keeping track of all state-action pair information is not optimal.

Instead of struggling with building a complete tree, a general tree can in many cases be constructed as an alternative. In a general tree, taking the Blocks World as example, a block is not referred to individually except through the variables stated in the goal. This is of course necessary in order to check whether a concrete goal has in fact been reached or not. Instead of referring to the individual blocks, an abstraction of them is utilized.

Using the general approach now makes it possible for the tree to represent the optimal policy for several similar goals all at once, such as the goal of reaching $on(a, b)$, $on(b, c)$ and $on(c, a)$. In the following a description of two known algorithms for learning a general Q-tree is presented and analyzed. The algorithms which will be described include $TILDE$-RT and TG.

4.6.3 The $TILDE$-RT Algorithm

The $TILDE$-RT[vO01] algorithm conceived by Hendrik Blockeel and Luc De Raedt in 1998 requires all training examples to be available at once, yielding a non-incremental learning system. Should any additional example input become available later on, the algorithm has to rebuild the entire structure of the tree all over from root to leaf. In order to do this, the algorithm has to keep track of both old and new examples.

When a given RRL-application is running, experience is collected and stored as examples of the type seen in Table 4.3. During an episode in the application, i.e. from a given start state to a given goal state, the agent selects actions according to the current policy and current Q-values. Any new state-action pair visited is kept in a new example, while the Q-value of old ones are updated directly in the tree.

```
1  Initialize Q̂_0 to assign 0 to all (s, a) state-action pairs
2  Initialize Examples to the empty set
3  e := 0
4  do forever
5     e := e + 1
6     i := 0
7     generate a random state s_0
8     while not goal(s_i) do
9        choose an action a_i and execute it
10       receive immediate reward r_i = r(s_i, a_i)
11       observe new state s_{i+1}
12       i := i + 1
13    endwhile
14    for j = i-1 to 0 do
15       generate example x = (s_j, a_j, q̂_j)
         where q̂_j := r_j + γ max_{a'} Q̂_e(s_{j+1}, a')
16       if an example (s_j, a_j, q̂_old) exists in Examples then
17          replace the example with x
18          else
19             add x to Examples
20    update Q̂_e using TILDE-RT to produce Q̂_{e+1} using Examples
```

Table 4.4: TILDE-RT based Q-learning

When an episode comes to an end, $TILDE$-RT is used to induce a new, updated Q-tree, using both old as well as new examples. During induction, the computation

of possible tests in a node may depend on variables in nodes higher in the regression tree. Tests higher in the tree must also be taken into account when determining whether an example input satisfies a test in a given node.

In the Blocks World variables are used to represent general, abstract blocks instead of concrete ones. This way, the tree becomes a general one, where concrete goals still can be used. A variable represents the same block down through the branches of the tree and as such can be used to represent any general relationships with other blocks. Important here is that different nodes can share the variables of the tree.

Once the Q-tree has been trained using the current example set, a Prolog Knowledge Base (KB) can be constructed. A KB merely contains the Q-tree along with all the relational facts in the state including the action and the goal. This KB can easily be transformed into a Prolog program, which can be used as a Q-function to retrieve the Q-values from.

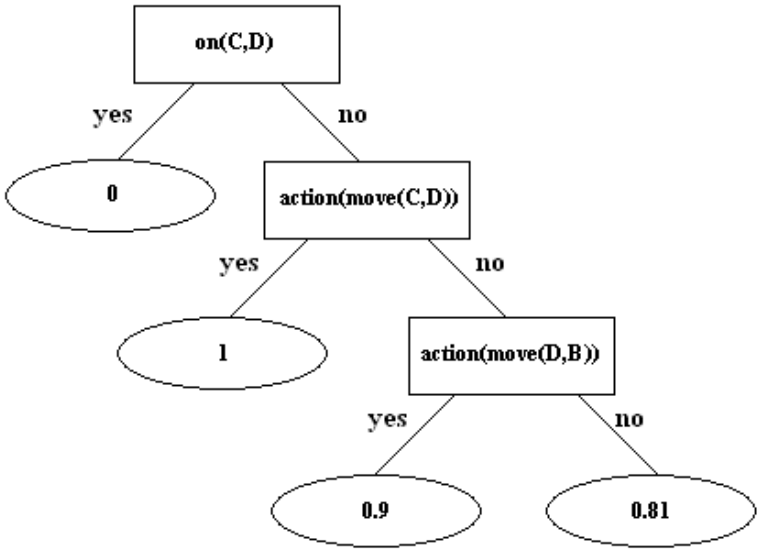

Figure 4.4: The Q-tree induced by $TILDE\text{-}RT$ from the examples in Table 4.5

The Q-learning algorithm integrated with $TILDE\text{-}RT$ can be seen in Table 4.4. Basically, the standard Q-learning algorithm from Table 2.2, Chapter 2 is reused with a small extension. The main difference between the two is in the last part of the algorithm, line 14-20, containing the for-loop and the $update$, which traverses the example set generated, and induces the actual tree.

Initially, in line 1 the Q-value of all state-action pairs in the Q-tree $\hat{Q}_0$ is set to zero like in standard table-based Q-learning. Line 2 clears the set of current examples, so it does not contain examples from a previous run. The algorithm then starts

learning by selecting actions, executing them, receiving rewards, and changing states correspondingly (line 9-11).

Whenever the new state reached is a goal state, the experience gained during that episode is stored. In line 14, this experience is traversed backwards, and in line 15 the temporary example x is generated. If the state-action pair in this new example is already existing in the Q-tree, i.e. the state-action pair has alread been visited at least once, then the Q-value of the old example currently in the tree is updated with the new one. If not, the example x will not be included in the tree until the next rebuilding of it is scheduled.

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 4.5: Examples generated for achieving $goal(on(a,b))$ as seen in Figure 4.5

When the current experience has been fully traversed, line 20 deals with inducing the new tree $\hat{Q}_e$ according to the $TILDE\text{-}RT$ regression algorithm, the current episode, and the current examples available, both the old as well as the new ones. Table 4.5 shows the example set generated after a given episode e for the goal $on(a,b)$.

This set contains the examples 1-4, where each example corresponds to each step towards the goal state as seen in Figure 4.5. Here, each example contains the current Q-value, the action chosen as well as the current goal, and the current state of the block stack. Figure 4.5 also lists the individual actions, the corresponding rewards denoted r and the Q-values, here denoted Q.

Unfortunately, the non-incremental algorithm for $TILDE\text{-}RT$ suffers from a number of rather serious problems. These include :

1. Rebuilding the entire Q-tree after each episode
2. A constant growing number of examples has to be memorized
3. Updating existing Q-values requires searching through the entire example set

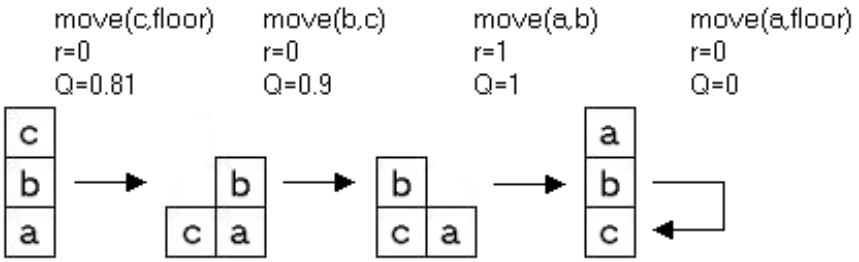

Figure 4.5: The states visited for reaching the goalstate $on(a, b)$. The corresponding examples are available in Table 4.5

It is considered a big problem that for each new episode the Q-tree has to be rebuilt all over again, in order to fully represent both old and new state-action pair information. However, the tree improves itself this way by being able to change the overall physical structure completely after each episode. Initially, the Q-tree will only cover a small fraction of the entire regression area, before reaching a certain level of experience as the tree grows.

However, rebuilding the entire tree after each and every single episode quickly becomes a problem as the number of generated examples is increased. These examples have to be stored in memory or in a flat file, searched and finally inserted into the Q-tree. No old example can be removed from the example set as it is needed in the rebuilding of the tree after each episode.

Another problem is that of updating the Q-values of already experienced state-action pairs. Here, the entire set of examples has to be searched before an optimal update can be performed. And for each new example added to the example set, this search becomes even more extensive, affecting the overall performance of the application.

As an alternative to $TILDE$-RT, the incremental TG-algorithm can be used. This algorithm deals with all the above-mentioned problems of the $TILDE$-RT algorithm.

4.6.4 The TG Algorithm

The TG-algorithm[DRB02] developed in 2001 by Driessen, Ramon and Blockeel is a first order extension of the G-algorithm[CK91] created by Chapman and Kaelbling in 1991. The G-algorithm is a learning algorithm for decision trees which is updated incrementally. That is, it is updated every time a new training example is provided as input. The TG-algorithm shares the incremental feature with the

```
1  initialize by creating a tree with a single leaf with empty statistics
2  for each learning example that becomes available do
3     sort the example down the tree using the tests of the internal nodes
      until it reaches a leaf
4     update the statistics in the leaf according to the new example
5     if the statistics in the leaf indicate that a new split is needed then
6        generate an internal node using the indicated test
7        grow 2 new leaves with empty statistics
```

Table 4.6: The TG-algorithm

G-algorithm.

The main difference between the G and the TG-algorithm is that TG uses relational interpretation in the description of example input and in the description of the tree itself. The G-algorithm, on the other hand, only works for propositional representations, e.g. problems that can be described with a less expressive representation form than a relational one. The kind of Q-trees built by $TILDE$-RT and TG are essentially the same.

Table 4.6 shows a high-level view of the TG-algorithm without the Q-learning part. The algorithm is used after each episode like with $TILDE$-RT. However, instead of rebuilding the tree over from scratch, the new examples available are simply inserted directly into the growing tree. Or updated, if the given state-action pair has already been visited.

In line 1, the algorithm creates the root node of the tree. As long as the application is running, line 2 checks to see if any new examples have come available. If so, the given example is sorted down the tree in line 3 according to the outcomes of the tests in the decision nodes the example encounters on its way through the tree. When a leaf is reached (line 4), the statistics of the leaf is updated with the results of the sorting process of the new example.

If the new statistics show that a new split is necessary for the example to find its permanent place in the tree, a new decision node is created in line 6 to hold the new test. At the same time, two new leaves with empty statistics are generated and connected to the new decision node. This takes place in line 7. The algorithm then returns to line 2 and continues until no new examples are received.

Besides keeping the Q-value to predict, each leaf in the tree stores, for each decision node test, the number of examples for which the test succeeded, the sum of their related Q-value along with the sum of squared Q-values. The same three values are kept for the set of examples for which the test failed. Keeping these six values enables TG to compute the *significance* of a test in a leaf and to decide

whether to split the leaf in question or not.

If a given test is significant, the variance of the Q-values supplied in the examples would be reduced sufficiently by splitting the node using the test in question. Splitting a node is carried out after some minimal number of examples has been collected and some test becomes significant with a high confidence.

The TG-algorithm has an obvious advantage from $TILDE\text{-}RT$: since it incrementally adds new example input, it does not have to rebuild the Q-tree over from scratch after each episode. This means that with $TILDE\text{-}RT$ it is necessary to keep track of an increasing number of examples and store them in memory, as well as replace current Q-values with new, whenever a state-action pair is encountered again. These capabilities proves TG as a much faster algorithm than $TILDE\text{-}RT$.

A problem with the incremental TG-algorithm is to select a good *minimal sample size*. The minimal sample size determines when exactly to perform a split on a given node, effecting the size of the tree and the convergence rate. Since the tree is induced incrementally, one or more bad splits performed on initial nodes will make later branches suffer from it as well. Opposed to $TILDE\text{-}RT$ the TG-algorithm cannot reverse a bad split by totally rebuilding the tree when more experience is collected.

4.7 Relational P-learning

The P-tree[DRB02] works as an abstraction of the Q-values in the Q-tree. Instead of mapping state-action pairs to their Q-values, a P-tree performs a mapping from state-action pairs to the optimal or non-optimal policies. A P-tree is hence a representation of the optimality of a given state-action pair. The P-tree itself is built after each episode once the Q-tree has been learned.

The P-learning proces is typically less complex than the Q-learning one. Basically, a P-tree is a representation of a function that returns true if a given action a is considered optimal in a given state s, and false otherwise: $if\ a \in \pi^*(s)\ then\ P(s,a) = true\ else\ P(s,a) = false$. In general, a P-function can be represented in a more compact way than a Q-function, because it does not assign different real values to state-action pairs.

Whereas Q-learning deals with the distance to, and the amount of next and later rewards, P-learning usually leads to a further improvement of the policy generated. Or leads to a faster convergence of the optimal policy. On the other hand, the Q-function implicitly knows the distance from a current state s to the goal state.

The P-function is defined using the optimal policy π^*, which again can be defined

```
1  Initialize Q̂_0 to assign 0 to all (s, a) pairs
2  Initialize Examples to the empty set
3  e := 0
4  while true
5     generate an episode consisting of states s_0 to s_i and actions a_o to a_{i-1}
      (where a_j is the action taken in state s_j) through the use of a standard
      Q-learning algorithm and the current approximation for Q̂_e
6     for j = i - 1 to 0 do
7        generate example x = (s_j, a_j, q̂_j),
         where q̂_j := r_j + γ max_{a'} Q̂_e(s_{j+1}, a')
8        if an example (s_j, a_j, q̂_old) exists in Examples then
9           replace it with x
10          else
11             add x to Examples
12    update Q̂_e using TILDE-RT to produce Q̂_{e+1} using Examples
13    for j = i-1 to 0 do
14       for all actions a_k possible in state s_j do
15          if state-action pair (s_j, a_k) is optimal according to Q̂_{e+1} then
16             generate example (s_j, a_k, c) where c = true
17             else
19                generate example (s_j, a_k, c) where c = false
20 update P̂_e using TILDE to produce P̂_{e+1} using these examples (s_j, a_k, c)
21 e := e+1
```

Table 4.7: The algorithm for learning the P-tree

as a function of the Q-function. Therefore, the P-function can also be expressed in terms of the Q-function, yielding: $if\ a \in \arg\max_a Q(s,a)\ then\ P(s,a) = true\ else\ P(s,a) = false$. This again means that any approximation $\hat{Q}$ of Q has a corresponding approximation $\hat{P}$ of P, hence the algorithm for Q-learning can be extended to include P-learning. This is achieved by adding an additional step at the end of the Q-learning algorithm. This extra step then defines the $\hat{P}$ in terms of $\hat{Q}$.

Constructing the P-tree is strongly dependent on the Q-tree and will not work if the Q-tree does not work. Though the P-tree is not vital for the relational reinforcement learning problem to be solved, it does boost the generalization process a bit, especially when, for example, the number of blocks in the Blocks World gets larger than the number used during training.

The algorithm for learning the P-tree in conjunction with standard Q-learning and

$TILDE$[3] can be seen in Table 4.7. The initial parts, lines 1-12, are similar to the algorithm used for $TILDE$-RT, where the system is initialized and episodes are carried out, and corresponding examples of the type (s, a, q) are generated, and updated immediately in the Q-tree if a given state-action pair already has been visited.

The lines 13-20 cover the actual P-learning process: For each state visited during an episode, P-learning looks at all possible actions available to the agent in that particular state as well as the Q-value predicted by the Q-tree for these actions. If the state-action pair is optimal, the example $(s, a, true)$ is generated in line 16. If not, the example $(s, a, false)$ is generated instead in line 19.

When all the examples have been covered, line 20 deals with updating the current approximation of P-tree $\hat{P}_e$ with the new examples, yielding the new approximation, $\hat{P_{e+1}}$. Instead of the non-incremental $TILDE$, a classification tree building TG-algorithm could have been used to induce an incrementally induced P-tree. The resulting trees would have little differences, if any.

The P-tree, as opposed to the Q-tree, is a relational classification tree. The only difference between the regression tree and the classification tree, is the information stored in the leaves. Instead of containing real valued numbers, the leaves of a classification tree contain classes. In the case of the P-tree, the classes used are *optimal* and *nonoptimal*. The initial P-tree induced from the example 1 generated by the Q-tree can be seen in Figure 4.6. The episode used for the Q-tree and the P-tree is the one illustrated in Figure 4.5.

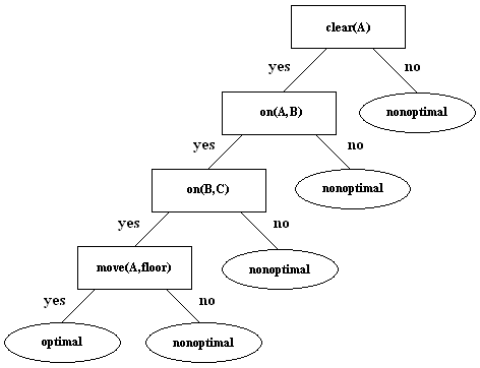

Figure 4.6: The P-tree generated from example 1 in Table 4.8 for the action $move(c, floor)$ and the goal of reaching $on(a, b)$

[3] $TILDE$ is merely the classification tree building version of $TILDE$-RT

Example 1	Example 2-1	Example 2-2	Example 2-3
optimal.	optimal.	optimal.	nonoptimal.
move(c,floor).	move(b,c).	move(b,floor).	move(c,b).
goal(on(a,b)).	goal(on(a,b)).	goal(on(a,b)).	goal(on(a,b)).
clear(c).	clear(b).	clear(b).	clear(b).
on(c,b).	clear(c).	clear(c).	clear(c).
on(b,a).	on(b,a).	on(b,a).	on(b,a).
on(a,floor).	on(a,floor).	on(a,floor).	on(a,floor).
	on(c,floor).	on(c,floor).	on(c,floor).
Example 3-1	Example 3-2	Example 3-3	Example 4
optimal.	nonoptimal.	optimal.	nonoptimal.
move(a,b).	move(b,a).	move(b,floor).	move(a,floor).
goal(on(a,b)).	goal(on(a,b)).	goal(on(a,b)).	goal(on(a,b)).
clear(a).	clear(a).	clear(a).	clear(a).
clear(b).	clear(b).	clear(b).	on(a,b).
on(b,c).	on(b,c).	on(b,c).	on(b,c).
on(a,floor).	on(a,floor).	on(a,floor).	on(c,floor).

Table 4.8: Examples for learning the P-tree by $TILDE$ generated from the examples for the Blocks World episode in Figure 4.5

4.8 ACE

ACE[BDRS04] is a data mining tool developed in 2001 covering a large variety of relational data mining algorithms, including $TILDE$-RT and TG. A large number of people has been involved in the proces of building the tool, including Hendrik Blockeel, Luc Deshaspe, Jan Ramon, Luc De Raedt, Wim Van Laer and Jan Struyf. Each contribution works as an independent part of the system and ACE is merely providing a common interface to these parts.

Over the years the ACE-tool has been improved and widely extended. The system is based on an underlying Prolog engine to handle the relational input and generate corresponding Prolog programs. Currently, ACE incorporates the algorithms depicted in Table 4.9.

4.8.1 Input Files

Depending on the system part envoked, a generel ACE session as shown in Figure 4.7 involves three input files with the file extensions *name.kb*, *name.bg* and *name.s* where *name* refers to a concrete name of the application in question, and the extension refers to the actual type of input. The contents of the individual input files

$TILDE$	An upgrade of the C4.5. decision tree learner towards relational data mining
$WARMR$	An upgrade of the APRIORI-algorithm towards relational data mining
ICL	A relational rule learner based on the rule learner $CN2$ and upgraded towards relational data mining
$RegRules$	A system for performing linear regression with relational features
KBR	A system for learning first order kernels
NLP	A system for learning neural logic programs
$RIB3$	A relational instance based learning system
TG	An incremental version of $TILDE$
RRL	A system for performing relational reinforcement learning that can use the following incremental regression systems: KBR, NLP, $RIB3$ and TG.

Table 4.9: The current algorithms covered by the ACE data mining tool

must all be Prolog-based, so the so-called ILProlog[BDD$^+$02] engine that runs inside ACE can handle the input. Each input file has a certain content as well as purpose.

The Knowledge Base File

The **.kb* file or *knowledge base* file contains examples generated by a running application such as the blocks world. The examples cover both training examples as well as examples for test purposes. In order to differentiate between the individual examples supplied in the example set, the *models* format can be used as a delimiter.

Here, each example must be described using a model delimiter, i.e. each example must begin with the the line *begin(model([name]))* and end with the line *end(model([name]))*. The *name* can be used to identify a given example, but unique names are not a requirement. In between the delimiters, all facts used to describe the properties of a single example must be provided.

The contents of the knowledge base file and the contents of the background knowledge file may seem quite similar. However, a relation or predicate should be put in the background knowledge if adding the example to the set of examples does not cause any change to the definition of that predicate. If not, it belongs in the knowledge base.

The Background Knowledge File

The file extension **.bg* reveals a *background knowledge* file. This particular file should contain information generally valid about the domain from which the examples to use in ACE are generated. If no such information exist, the file may be left blank. Making use of the background knowledge file is optional.

The Settings File

The **.s* file, also referred to as the *settings*, includes the actual settings of the ACE-tool as supplied by the user. The contents of this file will influence the way the system handles the input, computes the output and which part of it that should be run. The settings file can be rather complex to use because the ACE-tool provides a large variety of different settings to use.

Among the more important settings are: system parameters, language bias, control settings on input as well as output, output information options and pruning methods for decision and regression tree structures. Furthermore, settings involving the choice of algorithms to use, the part of the system to use on the input, which values to predict, classes to use for classification, how to perform clustering and so forth.

4.8.2 Output File

The contents of the generel output file **.out* depends on the settings as defined in the settings file. The file is basically used to display the settings as chosen by the user along with statistics and the results concerning the computation which has just taken place.

In the case of the blocks world statistics could include the amount of time used to induce the Q-tree as well as the P-tree, the number of nodes in the trees, the number of nodes present in the trees after post-pruning them and many more. The results would be the trees themselves, displayed in a special text-notation format and their derived Prolog programs.

The resulting Q-tree induced by the examples from Table 4.5 along with its equivalent Prolog program can be seen in Table 4.10 and Table 4.11 respectively. The regression algorithm used to induce this particular tree was $TILDE\text{-}RT$. The ACE-tool is capable of generating quite a lot of output besides the generel output file once specified in the settings file.

One of the main problems with the ACE data mining tool is that a lot of different

```
1  action(move(A,B) , goal(on(C,D))
2  on(C,D) ?
3  +–yes:   [0]
4  +–no:    action(move(C,D)) ?
5           +–yes:   [1]
6           +–no:    action(move(D,B)) ?
7                    +–yes:   [0.9]
8                    +–no:    [0.81]
```

Table 4.10: The Q-tree induced by $TILDE\text{-}RT$ using the examples from Table 4.5

```
1  qvalue(0)      :-  action(move(A,B)) , goal(on(C,D)) , on(C,D), !.
2  qvalue(1)      :-  action(move(A,B)) , goal(on(C,D)) , action(move(C,D)), !.
3  qvalue(0.9)    :-  action(move(A,B)) , goal(on(C,D)) , action(move(D,B)), !.
4  qvalue(0.81).
```

Table 4.11: The Prolog program derived from the Q-tree displayed in Table 4.10

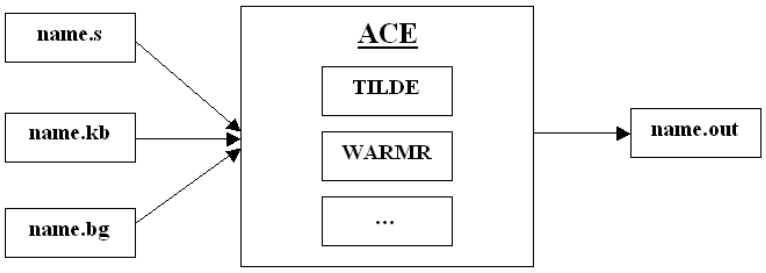

Figure 4.7: The general input/output scenario of using ACE

developers has been working on the system over a long period of time. Now, it basically works as a big black box system that is fed with certain types of input. The system then processes the input and stores the result in several output files. What happens during computation of the input inside the tool itself is, unfortunately, rather poorly documented in the user manual. Some parts, like the integrated TG-algorithm is not even covered.

Another problem is how the tool handles the many different parts of the system. It becomes a rather complex task of setting up the system for the parts one needs to actually use. Not only is it necessary to define which part you want to use and how exactly you want it to work, but you also explicitly need to tell it which parts you do not want make use of. Also, helpful error-related information is quite insufficient.

The ACE-system was supposed to be used in conjunction with this project, but unfortunately it appears that the current version does not seem to run properly. Even the enclosed examples used for regression will not produce the necessary output, i.e. a Q-tree and a P-tree. Instead the user is presented with an non-informative error and support for the tool is no longer provided.

Attempts to fix the problem are limited by the lack of documentation of the RRL, the TG and the $TILDE$-RT systems. And trying to produce a similar tool is directly related to a massive work load, which unfortunately does not comply with the time limit. In comparison, the complete ACE-tool itself took about four years to finish.

In the next chapter, the well-known puzzle game of Tetris is presented. This seemingly simple game is discussed in detail with speciel emphasis on how to solve it through the use of reinforcement learning techniques.

5 Tetris

The original video game of Tetris[TTC02] was invented by Russian mathematician Alexey Pajitnov in 1985 on an Electronica 60[1] at the Moscow Academy of Science's Computer Center. Later ported to the IBM PC, Apple II and Commodore 64, it became one of the most popular games of the late 80s and was soon running on almost every computer platform and game console available.

The popularity of Tetris, however, lead to the downfall of a number of software companies due to a legal rights dispute concerning the copyright ownership of the game. Many variations of the original game exist. In this report, however, the specification[Fah03] of Tetris is as presented in the following sections.

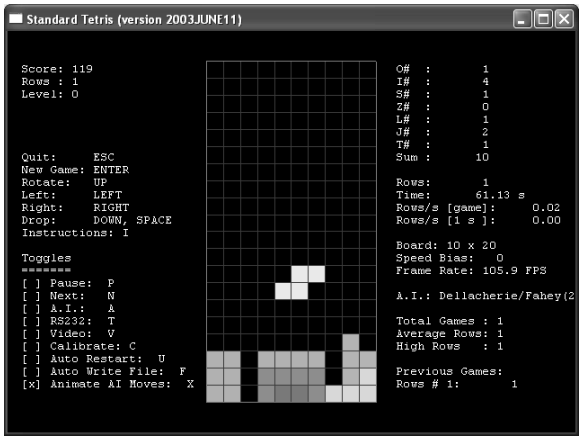

Figure 5.1: One of the many Tetris implementations available (courtesy of Colin Fahey, www.colinfahey.com)

5.1 The Board

The board of Tetris constitutes a *well* in which the player must stack dropping pieces. The well is a matrix, 10 columns wide and 20 rows deep, with a wall at the bottom, and at the right- and lefthand side of the board. The walls, or barriers of the board cannot be exceeded by any piece. This means, that movement, including

[1] A terminal computer made in the Soviet Union

rotation, of a piece must proceed according to the barriers of the well. Neither can be performed if doing so will move the piece outside the board.

The board itself is rowbased and filled from bottom to top as the game forwards in time. In order to avoid flooding the well with pieces, the player must complete the rows, which removes them from the board. Completing four rows all at once is referred to as a *Tetris*, hence the name of the game (see Figure 5.2). If the player fails to remove any rows, or the stacking of pieces is unorganized and incoherent, the well will quickly flood as the gravity of the game pulls the pieces towards the bottom of it.

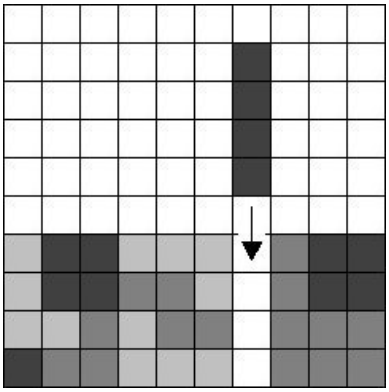

Figure 5.2: Using the vertically rotated I-piece to score a "Tetris"

5.2 The Tetromino Pieces

A piece used in the game of Tetris is called a tetromino[2], which is a geometric shape composed of four orthogonally connected squares. The name itself is taken from the Greek word for the number four, *tetra*. With respect to the connectivity, using this number as inspiration for constructing differently shaped pieces yields seven combinations as seen in figure 5.3. The pieces are denoted letters from the alphabet due to their characteristic similarities.

5.3 Rules and restrictions

Deceptively, Tetris is a rather simple puzzle game. The task of the player is to organize and stack falling puzzle pieces of different shapes into an orderly manner

[2]Tetromino is sometimes spelled tetramino or tetrimino

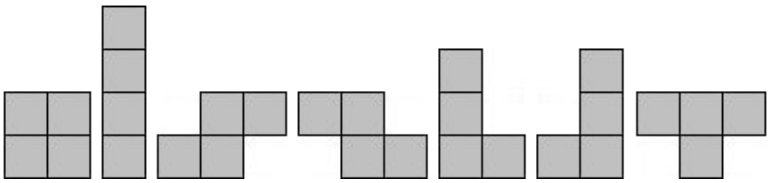

Figure 5.3: The seven tetrominoes (O, I, S, Z, L, J, T) used in Tetris

so that they fit the bottom of a board. The stacking must be carried out in such a way that holes, canyons and peaks in the heap of pieces are avoided. If not, the heap will end up too high and new pieces will quicky cram up the top of the board, ending the game. Each time a player manages to fill a row completely, it is removed from the board and the pile on top of it, if any, collapses exactly one row.

During each iteration of the game, a controlling timer will do one or more of the following, depending on the status of the game:

- **Spawn** a new piece when necessary
- **Collapse** rows that are complete
- **Drop** current piece one row
- **Check status** of the game

A random generator selects the next puzzle piece in a continuous sequence of pieces as long as the game is running. The piece is spawned at the top of the board and drops one row at a given time interval. The player can rotate and move the current piece to the right, to the left and downwards on the board with respect to an underlying coordinate system and the possible orientation of the different pieces.

When the current piece reaches and settles at the bottom or on any given row, completed or uncompleted, the player loses all control of it and it must remain there until, if ever, the row is completed later in the game. Every time a piece comes to a halt, a new piece is spawned. This event will continue to take place as long as the board still has enough empty space left on it.

5.4 Solving Tetris with Reinforcement Learning

In the following an MDP of a Tetris game is conceived and developed. In order to use reinforcement learning techniques with respect to the properties of Tetris and

MDPs, it is necessary to decompose the game and describe the different parts of the it.

The MDP for a Tetris game contains the following parts of interest:

- The size of the Tetris state space (involving the S set of states)
- The actions available (A)
- The unknown reward function ($r : S \times A \rightarrow R$)
- The unknown transistion function ($\delta : S \times A \rightarrow S$)

5.4.1 The Tetris State Space

The size of the state space of Tetris is important for developing a succesful application. If the state space proves too large, Q-learning will have great difficulties in reaching each and every single state of the game, at least once. The policy found might therefore, in fact not be the optimal one.

The factors that affect the size of the state space include:

1. The physical size of the board
2. The number of actions available
3. The number of tetrominoes and their orientations

Physical Board Size

The board of a conventional Tetris game consists of a 20 rows $\times$ 10 columns matrix, providing a total of 200 cells. Each piece in the game always covers exactly 4 of these cells, except for pieces that have been disassembled due to completed rows being removed from the board.

Available actions

Tetris is a very limited game when it comes to the number of possible actions each piece can execute in a given state. Still, this is enough to produce a rather complex and difficult game to solve regarding reinforcement learning. The limitations of the

board constituted by the walls of the well, however, prevents the use of all actions in every state of the board.

The set of actions A available to a Tetris playing agent is:

$$A = \{moveleft, moveright, movedown, rotate\}$$

On the lefthand side of the board, one cannot use the $moveleft$-action, since it will push the piece outside the board. Naturally, the same limitations exist with $moveright$ on the righthand side along with $movedown$ when the bottom of the well has been reached. Similar problems arise when trying to execute the $rotate$ action on a piece in a given state.

Also, these limitations are present if an attempt to move a piece on top of any other piece is made. In any such event, a collision will take place and prevent the action from being executed.

Tetrominoes and Orientations

There are a total of seven pieces in the standard game of Tetris. Each piece, though sharing the same physical area, have distinct differences in shape and orientation. The pieces and their shapes can be viewed in Figure 5.3.

Not all pieces have the same possibilities for rotation. The number of orientations for each piece is shown in Table 5.4.1. Since each orientation basically provides a new tetromino, the actual number of pieces used is hence $1+2+2+4+4+4 = 17$ instead of just the 7. This greatly affects the complexity as well as the size of the state space.

PIECE	ORIENTATIONS
O	1
I	2
S	2
Z	2
L	4
J	4
T	4

Figure 5.4: Orientations available to the Tetris pieces

5.4.2 The reward function

The reward function $r : S \times A \rightarrow R$ is an important part of the MDP. An agent should be rewarded when entering states that brings it closer to the goal of the game. A goal, however, in Tetris is somewhat unclear. Obviously, the overall goal of the game for the agent is to stay alive for the longest period of time possible by completing and hence removing rows from the board.

This, however, is not a satisfactory goal in the sense of reinforcement learning. A way of rewarding a player during game play is needed, since it not acceptable to wait until the agent loses or "not loses" before a reward is given. Such a goal would be infeasible to try to learn and would not make much sense anyway, In theory, the goal state of *not losing* might never be reached, since a game could continue forever.

The reward function should reward the positioning of tetromino pieces at the bottom, which do not *mess up* the pile, and punish the ones who do. The term "messed up" is, however, not a very useable condition to check in a machine learning context.

Instead it is necessary to give a more clear definition of when a pile in Tetris is messed up along with just how messed up it is before and after the agent executes an action and changes states. Then, a comparison between a snapshot of the pile before and after the positioning of the piece in questioned, can be exploited and used to reward or punish the player.

First, a proper representation of a pile needs to be conceived. This can be done in numerous ways, but a common characteristica should be the ability to represent information about:

- Canyons
- Holes
- Peaks
- Pile contour

Canyons

A canyon (see Figure 5.5) is a disruption in the surface[3] of the pile. Canyons are important in the game, since they are necessary in the building and completion of

[3]The surface constitutes the upper layer of the pile in the well, extending from the left to the right of the board

rows. Creating canyons should and cannot be avoided.

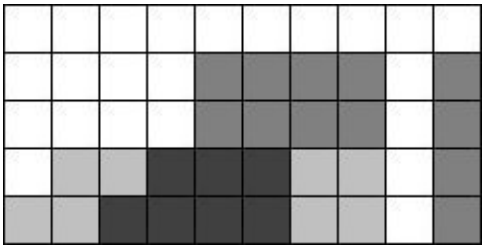

Figure 5.5: A canyon present in the pile of pieces

One should, however, limit the depth of a canyon, because the piece(s) required to fill up the canyon might not be spawned right away. This would force the player to try to fill it with incompatible pieces, which most likely would cause holes and oddly shaped subcanyons. All in all, canyons themselves should only be **punished** when of a certain depth.

Holes

A hole (illustrated in Figure 5.6) is an unreachable part of the board beneath the pile surface. Before this space can be reached and filled, thereby completing the row(s) that makes up the hole, access to the it must be achieved. This can be done by removing any blocks of pieces that cover up the hole to the left, right or top of it.

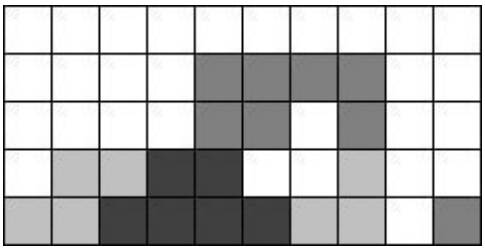

Figure 5.6: A hole in the pile prevents completion of rows

Holes are probably the main reason for messing up the pile, since they temporarily prevent rows from being completed and removed. This makes the pile grow high and the end result is a flooding of the well. Although holes at times are unavoidable, the player should seek to have them removed as soon as possible. Hence, a reward for creating a hole in the pile should be a **negative** one.

Peaks

A peak is an abrupt extension of the surface of the pile. Peaks are, like canyons, impossible to avoid since they are a natural element in the process of the stacking of Tetronimo pieces. Figure 5.7 shows an example of a peak in a Tetris game.

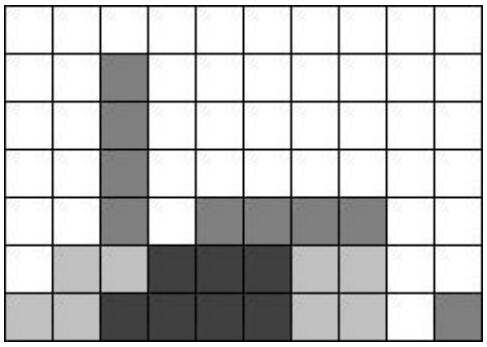

Figure 5.7: A peak extending from the surface of the pile

Tall peaks are usually the result of a non-optimal way of arranging the pieces in the pile. The taller the peak, the higher the risc of the game coming to a quick end, because the peak will reach the top of the well. In general, peaks of a certain height should be avoided, hence suggesting a **negative** reward.

Skyline

Defining the elements responsible for creating a messed up pile is not enough. A way of representing the pile as a whole is required if a comparisment using a "before-and-after" snapshot is to be possible.

One approach could be to describe the upper contour or *skyline* of the pile by counting the height of the columns and storing each value in a contour set C, starting with the leftmost column and moving across the board, ending with the rightmost one.

Using the example illustrated in Figure 5.8, the members of the contour set C would contain $C = \{1, 2, 6, 2, 3, 3, 3, 3, 1, 1\}$. The problem with this particular approach is that it lacks information about holes in the pile.

Evaluating whether the agent has made a good move or not is now possible, if the snapshot of the skyline taken before the move and the one taken afterwards, are compared. If the resulting skyline is worse than the initial one, the agent is punished. And rewarded otherwise.

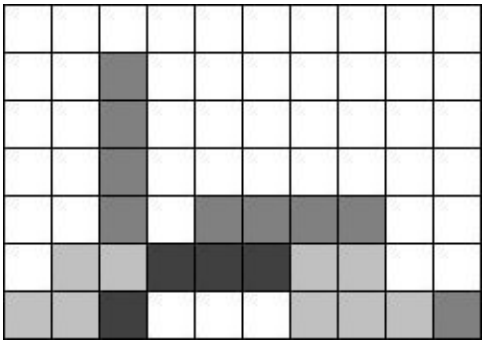

Figure 5.8: A pile with the contour $C = \{1, 2, 6, 2, 3, 3, 3, 3, 1, 1\}$

If the skyline produced reveals the presence of 1, 2, 3, or 4 complete rows, these are consideres subgoals in the game, and should be rewarded accordingly. Since the Tetris, i.e. completing four rows, removes the most rows from the board, the biggest reward should be provided here.

In the following chapter, a reinforcement learning implementation of the game of Tetris is described. The implementation is largely based on the ideas and conclusions made in the abovementioned text.

6 Implementation

The following chapter of this report is concerned with an actual implementation of the puzzle game of Tetris, based on conventional reinforcement learning. Due to the high complexity of the full size Tetris game, a limitation of it is presented and implemented.

6.1 Tetris$_{LTD}$

Tetris$_{LTD}$ is a drastically reduced version of the Tetris game described in Chapter 5. Implementation has been done using Microsoft Visual C++ 6.0. and the source code is available on the enclosed CDROM. In order to limit the complexity, the amount of tetronimoes used are cut down from seven to one. The piece which is used comprises a quadratic, single cell based tetronimo.

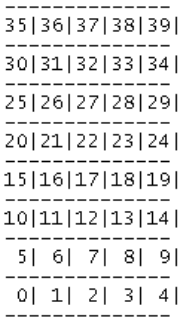

Figure 6.1: The 40-cell board used in the Tetris$_{LTD}$ implementation

The Tetris Board

The Tetris board used in the implementation is a 5 x 8 cell board based on a simple list containing the 40 elements as seen in Figure 6.1. Each element can be either **?** (unoccupied), **B** (occupied by a tetromino) or **A** (occupied by the agent tetromino, i.e. the current tetromino handled by the agent). The start state of the agent

tetronimo can be found in the cell labeled "37" in the center of the top row of the board.

The Reward Function

In Tetris$_{LTD}$, the reward function is based on the $skyline$ approach as described in Section 5.4.2. Here, a contour set C describes the height of every column on the board at any given time during game play. This representation form is particular useful in this case, because the problem with making holes using the quadratic tetronimo simply is not present.

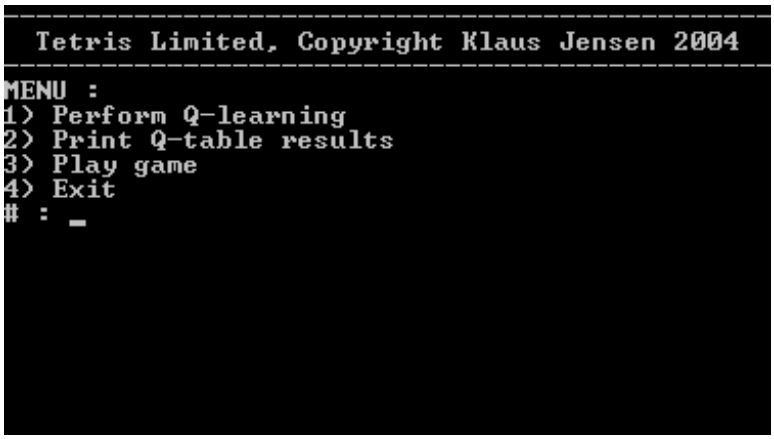

Figure 6.2: The screen menu of the Tetris$_{LTD}$ game

Also, representing canyons and peaks is extremely easy. It is just a matter of keeping track of every column height and its adjacents neighbours across the board. In Figure 6.3 the column heights can be seen right below the textual version of the Tetris$_{LTD}$ board.

Whenever a tetronimo piece is added to the pile on the board, the corresponding column is increased. If the affected column was not among the smallest columns in C, the agent will receive a negative reward of 10 for each tetromino in the column, i.e. $-10 \times columnheight$.

If the agent manages to place a piece such that the bottom row of the board is complete, a Tetris has been scored. In such a case, the agent will receive a positive reward of 100 for entering the goal state. After the reward has been given, the row collapses and any other pieces left on the board drops one row.

Output

Running the Tetris$_{LTD}$ application will generate two log-files: *Qlearning.txt* and *Gamestates.txt*. The first file contains data such as states, actions chosen, Q-values calculated, and a textual version of the current board, all gathered during the Q-learning proces. An excerpt of a single Q-learning step can be seen in Figure 6.3.

```
------------------------------------------------
Step: 197
------------------------------------------------
?????
?????
?????
?????
????B
????B
?BABB
?BBBB
-----
02224             = column heights
-----
Current state     = 7
Current column    = 2
Left wall         = 5
Right wall        = 9
Tetrises          = 0
Blocks            = 10
Games             = 1
Chosen action     = move_down
New state         = 7
Immediate reward  = -20
Gamma             = 0.9
Max reward        = 0
Q_hat value       = -20 (-20 + 0.9 * 0)
------------------------------------------------
```

Figure 6.3: Output from a single step in table-based Q-learning in Tetris$_{LTD}$

Using the application is fairly straightforward. The menu containing 4 different options appears as seen in Figure 6.2. Option 1 performs Q-learning for a given number of steps supplied by the user. Once the Q-learning process has committed, the user can select option 2 and three. Selecting 2 will print the entire contents of the Q-table on the screen, while option 3 will start a concrete Tetris$_{LTD}$ game.

Once the game is finished, the Gamestates.txt file can be viewed to see what exactly happened during the game play. And last but not least, selecting option number 4 can be used to exit the application.

Results

The game of Tetris$_{LTD}$ is, however, not yet fully operational. A current problem when playing the game involves in having the agent follow the steps as guided by the optimal policy from the start state 37. Once the first tetromino has been positioned at the bottom of the board, the next one will follow the same path.

This will make the agent end up in a position right above the previous piece and try to enter the same state of it. Consequently, the new piece ends up right on top of the first instead of selecting the action which will lead it into a state with second highest Q-value.

7 Conclusion

Learning a machine the skills to play a game or just to find its way around in a given environment is not an easy task, even for the simplest and smallest of domains. Usually, finding the optimal solution to a problem requires an immense amount of computation to be done.

This project has been concerned with a study of different representation methods feasible to use in conjunction with machine learning techniques for applications which suffer from large state spaces.

In the context of small computer games, the particular area of reinforcement learning was introduced and discussed. Reinforcement learning seems like an obvious choice for many domains, but seemingly table-based Q-learning does have a problem with learning environments that involves a lot of different states.

Different approaches for decreasing a state space was therefore looked into. The purpose here was extend conventional reinforcement learning with a better, and more expressive representation form. Here, it was found that structural representations such as relational interpretation are among the most expressive methods known today.

Extending reinforcement learning with a relational, first order logic representation yields a relational reinforcement learning technique. This improvement of the traditional MDP into a R-MDP makes it possible to describe a given state space in a much more general way.

The strengths and weaknesses of using relational reinforcement learning was then investigated through the use of the Blocks World example. Among the strengths were the possibility of generalizing over state space information, which could decrease the size of a given state space considerably.

Also, using first order logic, it is actually possible to derive new facts from existing ones, making it a very expressive learning method. Among the weaknesses it was found that not all applications could make use of relational reinforcement learning. If no patterns in the state space can be found and generalized, it cannot solve the problem.

Unfortunately, it was not possible to get a running example of a relational reinforcement learning application, since the ACE-tool necessary for training a Q-tree and a P-tree was not usable due to internal faults and lack of support from the

developers.

Instead, conventional reinforcement learning was used to implement Tetris$_{LMT}$, a reduced version of the well-known puzzle game of Tetris. The reduction was needed in order to be able to use table-based Q-learning in the game. Tetris$_{LMT}$ was, however, not fully completed due to time related issues.

Conclusively, reinforcement learning is a good choice for small applications such as board games, mobile robots, and other simplistic systems. Relational reinforcement learning on the other hand, can be applied to a much wider array of more complex domains which can be described relationally.

7.1 Future Work

Several ideas and thoughts related to the area of reinforcement learning have come to mind while working on this particular project. Some of the more interesting includes:

- Extending relational representations with the aspect of mirroring state information. In games like Tetris, many states can be mirrored vertically on the board. This particular approach could be used to decrease the state space even further
- Moving beyond the area of games. What other applications could benefit from reinforcement learning? And perhaps more importantly, which ones cannot?
- Extending the expressive power of first order logic used in relational reinforcement learing. A problem with logic is that is basically sees the world as being either black or white. Some problems cannot be solved this way, or will at least provide poor results
- Investigate the possibilities of using relational reinforcement learning in conjunction with other relational techniques. These could include working with XML, relational databases and similar. What do one gain and what are the advantages/disadvantages?

Bibliography

[BDD+02] Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Ramon, and Henk Vandecasteele. *Improving the Effeciency of Inductive Logic Programming Through the Use of Query Packs.* 2002.

[BDRS04] H. Blockeel, L. Deshaspe, J. Ramon, and J. Struyf. *The ACE Data Mining System - User's Manual*. 2004.

[CK91] David Chapman and Leslie P. Kaebling. *Input Generalization In Delayed Reinforcement Learning: An Algorithm and Performance Comparisons*. Proceedings of the International Joint Conference on Artificial Intelligence, 1991.

[DRB02] Kurt Driessens, Jan Ramon, and Hendrik Blockeel. *Speeding up Relational Reinforcement Learning Through the Use of an Incremental First Order Decision Tree Learner*. 2002.

[DRD01] Saŝo Dẑeroski, Luc De Raedt, and Kurt Driessens. *Relational Reinforcement Learning*. 2001.

[Dri] Kurt Driessens. *Relational Reinforcement Learning*.

[Dri04a] Kurt Driessens. *Relational Reinforcement Learning, Ph.D. Thesis*, chapter 3, pages 25–36. 2004.

[Dri04b] Kurt Driessens. *Relational Reinforcement Learning, Ph.D. Thesis*, chapter 5, pages 53–70. 2004.

[Fah03] Colin P. Fahey. *Tetris AI*. Colin P. Fahey, 2003.

[Jen01] Finn V. Jensen. *Bayesian Networks and Decision Graphs*. Springer-Verlag, 2001.

[KHI94] Dr. A. Harry Kloph, Lt Mance E. Harmon, and Capt Leemon C. Baird III. *Reinforcement Learning: An Alternative Approach To Machine Intelligence*. 1994.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[SB98a] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*. The MIT Press, 1998.

[SB98b] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*, chapter 1.4, pages 10–15. The MIT Press, 1998.

[SB98c] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*, chapter 11.1, pages 261–267. The MIT Press, 1998.

[SB98d] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*, chapter 11.4, pages 274–279. The MIT Press, 1998.

[SB98e] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*, chapter 11.3, pages 270–274. The MIT Press, 1998.

[Spi02] Michael Spivey. *An Introduction to Logic Programming Through Prolog*. Prentice Hall Europe, 2002.

[TTC02] LLC The Tetris Company. *Tetris*. The Tetris Company, LLC, 2002.

[vO01] Martijn van Otterlo. *Relational Representations in Reinforcement Learning: Review and Open Problems*. 2001.