Machine Learning, 43, 7–52, 2001

Relational Reinforcement Learning

SAŠO DŽEROSKI saso.dzeroski@ijs.si
Department of Intelligent Systems, Jožef Stefan Institute, Jamova 39, SI-1000 Ljubljana, Slovenia

LUC DE RAEDT deraedt@informatik.uni-freiburg.de
Institüt für Informatik, Albert-Lüdwigs-Universität Freiburg, Georges Köhler, Allee 79, D-79110 Freiburg, Germany

KURT DRIESSENS kurt.driessens@cs.kuleuven.ac.be
Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

Editor: Lorenza Saitta

Abstract. Relational reinforcement learning is presented, a learning technique that combines reinforcement learning with relational learning or inductive logic programming. Due to the use of a more expressive representation language to represent states, actions and Q-functions, relational reinforcement learning can be potentially applied to a new range of learning tasks. One such task that we investigate is planning in the blocks world, where it is assumed that the effects of the actions are unknown to the agent and the agent has to learn a policy. Within this simple domain we show that relational reinforcement learning solves some existing problems with reinforcement learning. In particular, relational reinforcement learning allows us to employ structural representations, to abstract from specific goals pursued and to exploit the results of previous learning phases when addressing new (more complex) situations.

Keywords: reinforcement learning, inductive logic programming, planning

1. Introduction

Within the field of machine learning, both reinforcement learning (Kaelbling et al., 1996) and inductive logic programming (or relational learning) (Muggleton & De Raedt, 1994; Lavrač & Džeroski, 1994) have received a lot of attention since the early nineties. It is therefore no surprise that both Leslie Pack Kaelbling and Richard Sutton (in their invited talks at IJCAI-97, Nagoya, Japan) suggested to study the combination of these two fields.

From the reinforcement learning point of view, this could significantly extend the application perspective. Most representations used in reinforcement learning are inadequate for describing planning tasks such as the simple blocks world. Even reinforcement learning work that involves generalization has largely employed an attribute-value representation. Due to the use of variables in relational representations, it is possible to abstract from specific details of the learning tasks, such as the specific goal pursued. Indeed, when learning to plan in the blocks world, one would expect that the results of learning how to stack block a onto block b would be similar to stacking c onto d. Current approaches to reinforcement

learning have to retrain from scratch if the goal is changed in this manner, while for relational reinforcement learning such retraining is unnecessary. Relational reinforcement learning also allows us to exploit and apply the results of learning in a simple domain when learning in a more complex domain (e.g., going from 3 blocks to 4 blocks in the blocks world).

From the inductive logic programming point of view, it is important to address domains such as reinforcement learning. So far, inductive logic programming has mainly studied concept-learning, and largely ignored the rest of machine learning. By demonstrating the potential of relational representations for reinforcement learning, we hope to show that the relational learning methodology does not only apply to concept-learning but to the whole field of machine learning.

With this in mind, we present an approach to relational reinforcement learning and apply it to simple planning tasks in the blocks world. The planning task involves learning a policy to select actions. Learning is necessary as the planning agent does not know the effects of its actions. Relational reinforcement learning employs the Q-learning method (Watkins & Dayan, 1992; Kaelbling et al., 1996; Mitchell, 1997) where the Q-function is learned using a relational regression tree algorithm (see (De Raedt & Blockeel, 1997; Kramer, 1996)). A state is represented relationally as a set of ground facts. A relational regression tree in this context takes as input a relational description of a state, a goal and an action, and produces the corresponding Q-value. The Q-learning method can also be adapted in order to learn the P-function, an explicit representation of the policy implicitly represented by the Q-function. The P-function, which is represented as a first order logical decision tree, takes as input a state, an action, and a goal and predicts whether the action is optimal or not.

The paper is organized as follows. In Section 2, we view planning (under uncertainty) as a reinforcement learning task. In Section 3, we briefly review Q-learning and show how Q-learning can be used to learn a P-function. Section 4 briefly reviews decision trees while focusing on logical decision trees. Section 5 introduces relational reinforcement learning that combines Q-learning and logical regression trees, as well as P-learning and logical decision trees. Section 6 presents a variety of experiments aimed at exploring the potential of relational reinforcement learning. Section 7 concludes, touches upon related work and discusses avenues for further work.

2. Problem specification

2.1. Reinforcement learning

The typical reinforcement learning task using discounted rewards can be formulated as follows:

Given

– a set of possible states S.
– a set of possible actions A.
– an **unknown** transition function $\delta : S \times A \rightarrow S$.
– an **unknown** real-valued reward function $r : S \times A \rightarrow R$.

Find a policy $\pi^* : S \rightarrow A$ that maximizes

$$V^{\pi}(s_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

for all s_t where $0 \leq \gamma < 1$.

At each point in time, the reinforcement learning agent can be in one of the states s_t of S and selects an action $a_t = \pi(s_t) \in A$ to execute according to its policy π. Executing an action a_t in a state s_t will put the agent in a new state $s_{t+1} = \delta(s_t, a_t)$. The agent also receives a reward $r_t = r(s_t, a_t)$. It will be assumed that the agent does not know the effects of the actions, i.e. δ is unknown to the agent, and that the agent does not know the reward function r. The task of learning is then to find an optimal policy, i.e., a policy that will maximize the discounted sum of the rewards.

This formulation of reinforcement learning is typical (cf. (Mitchell, 1997; Kaelbling et al., 1996)). The key contribution of *relational* reinforcement learning is that relational representations will be used to represent states, actions and policies. Also, relational learners (as offered by inductive logic programming) will be employed as generalizers.

2.2. Reinforcement learning for planning

Planning with incomplete knowledge can be be recast as an instance of the reinforcement learning task sketched above. The main differences between typical planning tasks (as e.g. considered in STRIPS (Fikes & Nilsson, 1971)) and reinforcement learning are that

- in planning, one knows the effects of one's actions, i.e., the transition function δ is known to the agent,
- in planning, a known precondition-condition function *pre*:$S \times A \rightarrow \{true, false\}$ is given, which specifies in which states it is legal to apply which actions.
- in planning, one is given a *goal* function $goal : S \rightarrow \{true, false\}$, which characterizes the target states.
- in planning, the aim is to start from a state s_1 and to find a sequence of actions $a_1, \ldots, a_n$ ($a_i \in A$) such that
  - $goal(\delta(\ldots\delta(s, a_1))\ldots, a_{n-1}), a_n) = true$, and
  - $pre(\delta(\ldots\delta(s, a_1))\ldots, a_{i-1}), a_i) = true$.

This close relation between reinforcement learning and planning can be exploited in order to define a problem of learning to plan under incomplete knowledge. The setting is essentially that of reinforcement learning where

- A policy π has to be learned.
- The function δ is unknown to the agent.
- The reward at time t is $r_t = r(s_t, a_t)$. We will assume here that $r_t = 1$ if $goal(\delta(s_t, a_t)) = true$ and $s_t \neq \delta(s_t, a_t)$; otherwise $r_t = 0$. The reward function r is unknown to the learner as it relies on the unknown δ. The reward function only gives a reward in goal states.

– The state at time $t+1$ is $s_{t+1} = \delta(s_t, a_t)$ if $goal(s_t) = false$; otherwise $s_{t+1} = s_t$. This captures the idea that goal states are absorbing states, i.e., once the agent reaches a goal state, it stays there.

The optimal policy π^* allows us to compute the shortest plan to reach a goal state. So, learning the optimal policy (or approximations thereof) will allow us to improve planning performance.

2.3. An example

The type of learning task outlined above has been also considered by Pat Langley in his book (Langley, 1996). He uses it to illustrate reinforcement learning and as an example task he employs the blocks world.

Consider the situation where we have three blocks called a, b and c, and the floor. Blocks can be on the floor or can be stacked on each other. Each state can be described by a set (list) of facts, e.g., $s_1 = \{clear(a), on(a, b), on(b, c), on(c, floor)\}$. The available actions are then $move(x, y)$ where $x \neq y$ and $x \in \{a, b, c\}$, $y \in \{a, b, c, floor\}$.

It is then possible to define the preconditions and effects of actions. The Prolog code in Table 1 defines *pre* and δ respectively. The predicate `pre` defines the preconditions for the action `move(X,Y)` while the predicate `delta` defines its effects: `delta(S,A,S1)` succeeds when $\delta(S, A) = S1$. States are represented as lists of facts and the auxiliary predicate `holds(S,Query)` succeeds when `Query` would succeed in the knowledge base containing the facts in S only. The goal is to stack a onto b, i.e., `goal(S) :- member(on(a,b),S)`. Note that names starting with capitals denote variables in Prolog: thus a in $on(a, b)$ is a constant denoting a specific block and A in $on(A, b)$ is a variable denoting any block.

3. Q-learning and P-learning

Here we summarize Q-learning, one of the most common approaches to reinforcement learning, which assigns values to state-action pairs and thus implicitly represents policies. We then introduce the approach of P-learning which in addition represents policies explicitly.

3.1. Q-learning

In the setting sketched in Section 2.1, Q-learning allows us to approximate the optimal policy. The optimal policy π^* will always select the action that maximizes the sum of the immediate reward and the value of the immediate successor state, i.e.,

$$\pi^*(s) = \arg\max_a (r(s, a) + \gamma V^{\pi^*}(\delta(s, a)))$$

With this formulation of π^* we can acquire the optimal policy by learning V^{π^*}, provided perfect knowledge of δ and r. In our setting, however, the learner does not know δ and r. Therefore, even if we learned V^{π^*}, we would not be able to obtain π^* from it.

Table 1. A Prolog definition of the functions *pre* and δ.

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

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

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

The Q-function for policy π is defined as follows:

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

Knowing Q^*, the Q-function for the optimal policy, allows us to rewrite the definition of π^* as follows:

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

This rewrite is important as it shows that if the agent can learn the function Q^* instead of the V^{π^*} function, it will still be able to act optimally. In the following, we will call Q^* simply Q or the Q-function. For a fixed *goal*, an approximation to the Q-function, $\hat{Q}$, in the form of a look-up table, is learned by the algorithm in Table 2, cf. (Mitchell, 1997). Note that one can reduce the complexity of Q-learning by using an action-penalty representation or by setting initial Q-values to be different from zero (Koenig & Simmons, 1996).

It is common in Q-learning to select action a in state s probabilistically so that $Pr(a \mid s)$ is proportional to $\hat{Q}(s,a)$, e.g.,

$$Pr(a_i \mid s) = T^{-\hat{Q}(s,a_i)} \Big/ \sum_j T^{-\hat{Q}(s,a_j)} \tag{1}$$

Table 2. The basic Q-learning algorithm.

for each s, a **do**
  initialize the table entry $\hat{Q}(s, a) = 0$
  e := 0
do forever
  e := e + 1
  i := 0
  generate a random state s_0
  while not $goal(s_i)$ **do**
    select an action a_i and execute it
    receive an immediate reward $r_i = r(s_i, a_i)$
    observe the new state s_{i+1}
    i:=i+1
  endwhile
  for j=i−1 to 0 **do**
    update $\hat{Q}(s_j, a_j) := r_j + \gamma max_{a'} \hat{Q}(s_{j+1}, a')$

Lower values of the parameter T (temperature) give stronger preference to actions with high values of $\hat{Q}$ causing the agent to exploit what it has learned, while higher values of T reduce this preference allowing the agent to explore actions that currently do not have high values of $\hat{Q}$. Selecting actions according to this scheme will be called the *Q exploration strategy*.

3.2. P-learning

The Q-function encodes the optimal policy in a complex manner as it assigns a Q-value to all the possible state-action pairs. It will turn out useful to represent the optimal policy in a simpler way. This is realized by the P-function, which we define as follows:

$$\text{if } a \in \pi^*(s) \text{ then } P(s, a) = 1 \text{ else } P(s, a) = 0$$

Instead of assigning different real values to the state-action pairs, the P-function only decides whether the state-action pair is optimal (1) or not (0). In general, P-functions can be represented more compactly than Q-functions. Indeed, the Q-function implicitly encodes knowledge about the distance (number of steps) from the current state to the goal states, whereas the P-function does not. Examples of this, in the context of planning will be given later in this paper. This point is important as both functions will be represented by logic programs within relational reinforcement learning.

As the P-function is defined in terms of the optimal policy π^*, which in turn can be defined as a function of the *Q-function*, we can also express the P-function in terms of the Q-function in a straightforward manner:

$$\text{if } a \in \arg\max_a Q(s, a) \text{ then } P(s, a) = 1 \text{ else } P(s, a) = 0$$

This also means that any approximation $\hat{Q}$ of Q has a corresponding approximation $\hat{P}$ of P. As a consequence, the above algorithm for Q-learning can be extended into an algorithm for P-learning by adding an extra step that defines $\hat{P}$ in terms of $\hat{Q}$ at the end of the algorithm.

Instead of the Q exploration strategy it is now also feasible to use the function $\hat{P}$ to select the actions to execute in given states. This is then done using the following probabilities:

$$Pr(a_i \mid s) = T^{-\hat{P}(s,a_i)} \Big/ \sum_j T^{-\hat{P}(s,a_j)} \tag{2}$$

The corresponding strategy is called the *P exploration strategy.*

4. Top-down induction of logical decision trees

4.1. Decision trees

Decision trees are among the most popular representations for learning and data mining, see e.g. (Mitchell, 1997; Quinlan, 1986; Quinlan, 1993; Breiman et al., 1984). The term decision trees refers to classification trees and regression trees, although it is often used as a synonym for classification trees. The leaves of decision trees contain a prediction, which is a discrete class value in the case of classification (decision) trees or a continuous (real) class value (or a function yielding real values) in the case of regression trees. Each internal node of a decision tree contains a test. Furthermore there will be one subtree for each possible outcome of a test in the tree. In this way, decision trees partition the whole example space and assign class values to each example. To make predictions with a decision tree one starts in the root of the tree and applies the root's test to the example. Then one takes the branch that corresponds to the outcome of the test in the example and propagates the example to the corresponding subtree. If the resulting subtree happens to be a leaf, one reads off the prediction, otherwise one applies the procedure recursively on the example and the subtree. For instance, the decision tree shown in figure 1 can be used to classify states with three blocks named a, b and c into the classes *stacked* and *unstacked*. As an illustration, consider a state in which `clear(a) = true, clear(b) = false`, and `clear(c) = true`. This example would be classified in the third leaf (class *stacked*).

Classification and regression trees are typically induced using a divide and conquer algorithm, called top-down induction of decision trees (TDIDT). To induce trees, one starts from a set of examples and considers all possible tests in the root of the tree and selects the test that performs best according to a certain heuristic (e.g. information gain in the case of classification and variance reduction for regression). One then splits the data set according to the outcome of the test in the examples and one propagates the examples to the resulting subtrees. For each subtree, one then decides whether to turn the subtree into a leaf or to recursively call the induction procedure. This process continues until the tree is completed.

Fully grown trees are sometimes pruned to increase the reliability of their predictions on unseen cases. Various implementations of tree induction exist, cf. e.g. (Quinlan, 1986;

```
clear(a) ?
+--yes: clear(b) ?
        +--yes: clear(c) ?
                +--yes: unstacked
                +--no: stacked
        +--no: stacked
+--no: stacked
```

Figure 1. A decision tree to predict whether there is a stack in the 3 blocks world.

Quinlan, 1993; Breiman et al., 1984), using different heuristics and extensions of the basic TDIDT approach.

4.2. Logical decision trees

Classical decision trees employ propositional or attribute value representations. Recently, however, these representations have been upgraded towards first order logic, resulting in the frameworks of logical classification and regression trees (Kramer, 1996; Blockeel & De Raedt, 1998). As the work on logical decision trees has been extensively published elsewhere, we introduce only the key differences with classical decision trees. For full details of these logical decision tree methods we refer to (Blockeel & De Raedt, 1998; Blockeel et al., 1998; Kramer, 1996).

One key difference between logical and classical decision trees is that classical decision trees work with examples in attribute value form. This means that each example is described by a single feature vector (or single tuple in a table). In logical decision trees, an example is essentially a relational database (or a Prolog knowledge base) described by a set of facts. As an illustration consider Table 4, where each example corresponds to a state description in the block's world. States are assumed to be fully described, i.e. the closed-world assumption applies.

The other main difference is that logical decision trees employ (Prolog)-queries as tests in the internal nodes of the decision trees, e.g. `on(A,c)` (is there any block A on block c?). Since the outcome of a query in an example is either true or false, the resulting trees are always binary. Furthermore, the queries can contain variables, and these variables may be shared among several nodes of the trees. When variables are shared among several nodes of the tree they refer to the same object. Consider the tree shown in figure 2. This tree contains two queries: `on(A,B)` and `on(B,C)`. In order to classify an example with this tree, one will first test whether `on(A,B)` succeeds in the example for some `A` and `B`. If so, one will then test whether the query `on(A,B), on(B,C)` succeeds in the example. This shows that variables shared among multiple nodes in logical trees are supposed to bind to the same objects. Due to this property it is only meaningful to propagate the variables of a node to the succeeding branches of the subtree (labeled yes). Consider the failing branch of `on(A,B)` in figure 2. Given that there is no `A` and `B` such that `on(A,B)` it does not make sense to refer to `A` or `B`

```
on(A,B) ?
+--yes : on(B,C) ?
         +--yes : stacked
         +--no : unstacked
+--no : unstacked
```

```
stacked :- on(A,B), on(B,C), ! .
unstacked :- on(A,B), ! .
unstacked.
```

Figure 2. A logical decision tree that predicts whether there is a stack in the blocks world.

in the failing subtree of `on(A,B)`. The semantics of the tree is completely characterized by the corresponding Prolog program shown in figure 2. Due to the complications that arise one needs to employ cuts (the !) in the Prolog program. Because of the cuts, different rules in the program behave as in an if-then-else program. To classify an example one tries whether the condition part of the first rule is satisfied. If it is, one uses the corresponding prediction, otherwise one tries the second rule. This process continues until a rule is found whose condition is satisfied. The use of cuts in the Prolog program closely corresponds to so-called first order decision lists introduced by Mooney and Califf (Mooney & Califf, 1995). These and other aspects of the representation of logical decision trees are explored in detail in (Blockeel & De Raedt, 1998).

We will use logical decision trees as implemented in the programs TILDE (Blockeel & De Raedt, 1998) (for classification) and TILDE-RT (Blockeel et al., 1998) (for regression). From a practical point of view, TILDE can be viewed as an extension of the C4.5 (Quinlan, 1993) system for induction of decision trees. It uses the same heuristics to select tests in internal nodes, as well as the same mechanisms for tree pruning. For our purposes, TILDE-RT should be regarded as an extension of propositional regression tree learners, such as CART (Breiman et al., 1984). Nevertheless, TILDE-RT employs different heuristics (see (Blockeel et al., 1998) for details). The basic TILDE and TILDE-RT algorithms are outlined in Appendix A. TILDE and TILDE-RT employ the typical well-known top-down induction of decision trees (TDIDT) algorithm. The only difference lies in the generation of candidate tests to be put in the nodes. This will be explained in the next subsection.

4.3. Declarative bias

Because first order representations are more expressive than attribute value representations the search space explored by inductive logic programming systems is much larger (and often infinite). To focus the search on the most relevant hypotheses and to eliminate useless hypotheses from the search space, nearly all inductive logic programming systems employ some kind of declarative bias mechanism, see (Nedellec et al., 1996) for an overview. Declarative bias is often implemented by means of so-called mode and type declarations.

Type declarations specify the types of the arguments of the predicates involved and restrict the types of queries and clauses to be type conform. Consider the predicates `on/2` and `numberofblocks/1`. The type of arguments 1 and 2 of the predicate `on/2` is object (block) and the type of the only argument of `numberofblocks/1` is integer. Under these declarations the query `?-on(X,Y), numberofblocks(X)` is not type conform as it requires that `X` is of both type object and integer.

Modes specify properties about the calling patterns of predicates in clauses, queries or conditions to be induced. For example, the mode `on(+,-)` specifies that at the time of calling the predicate `on/2`, the first argument should be bound (or instantiated, it is of type $+$) whereas the second argument should be free (not instantiated, it is of type $-$). One can also combine these modes and write for instance `on(+-,+-)` stating that all calling patterns are permitted.

Modes are useful because they can focus the hypothesis language on interesting clauses or queries by excluding useless ones. For example, it can be used to exclude clauses/queries such as `?-height(X,XH), XH < YH.` by using the mode `+ < +`. This mode specifies that the two arguments of predicate 'less than' (`</2`) should be instantiated and guarantees in this context that the numbers would be bound before testing whether one is smaller than the other. Another type of mode is # which specifies that the resulting argument should be bound in the clause/query to a constant, e.g. `on(-,#)` would require that the last argument is instantiated. The rmode formalism employed by TILDE and TILDE-RT implements and slightly extends the above notions of type and mode declarations, cf. (Blockeel & De Raedt, 1998).

The main point where TILDE and TILDE-RT differ from propositional decision tree algorithms is the generation of tests to put in the internal nodes. The tests that are considered in a node depend on 1) the declarative bias, and 2) the tests in nodes higher in the tree (on succeeding branches). Roughly speaking, TILDE collects all literals in succeeding ancestors (including the root) of the node and then applies a so-called refinement operator to generate the tests. The refinement operator employs the declarative bias specifications. As an illustration of this, consider first the root node in the tree of figure 2 and its succeeding branch. The test in the root is `on(A,B)`. Given only the mode declaration `on(+, -)`, two refinements would be generated, i.e., `on(A,B), on(A,C)` and `on(A,B), on(B,C)`. This results in two candidate tests for the succeeding branch: `on(A,C)` and `on(B,C)`. Suppose the heuristic chooses as best the latter one (as in the actual tree in figure 2) and also that the resulting node should be further split. Then the tests considered in the succeeding branch of the node `on(B,C)` in figure 2 would be `on(A,D)`, `on(B,D)` and `on(C,D)`. On the other hand, in the failing branch of the node `on(B,C)`, one would consider only `on(A,D)` and `on(B,D)` because it does not make sense to refer to the variable `C` there.

4.4. Background knowledge

Another key issue when using inductive logic programming is background knowledge. Background knowledge consists of the definitions of general predicates that can be used in the induced hypotheses. It thus influences the concepts that can be represented.

Unlike the predicates that are used to describe training examples (state/action/qvalue or state/action/optimality triples in our case), such as `on(A,B)`, background knowledge

predicates specify knowledge that is generally valid across the whole domain, i.e., for all training examples. The predicate `above(A,B)` (cf. Appendix B) defines when block `A` is above block `B` in terms of the predicate `on(A,B)`. This knowledge holds over all states in the blocks world.

It is well-known that the representation language is a crucial parameter in machine learning. Given an adequate language, learning will be effective, and given an inadequate one learning will be difficult if not impossible. Applied to inductive logic programming this means that it is important to specify the right predicates in the background knowledge.

One advantage of inductive logic programming in this context is that the combination of background knowledge and declarative bias allows the user to influence the learning process and results. For instance, as we will see in the experiments, it is sometimes necessary to employ the predicate `numberofblockson(A,N)` to learn effective policies. This predicate specifies that there are exactly `N` blocks above block `A`.

Another issue related to our experiments is that of block identities: if one knows that the absolute identities of blocks are not important as opposed to their relative ones, then one can specify this using the modes (only allowing for a combination of $+$ and $-$ and not for #). While learning will not necessarily be unsuccessful without this knowledge, it can be much slower. To illustrate this, we have performed some relational reinforcement learning experiments for the stack and unstack goals (see Section 6.2). Without the assumption that policies are independent of block identities, TILDE uses block identities in the policies learned in early episodes, but does not reference block identities in the policies learned after a larger number of episodes. However, the time needed for learning the policies was three times longer as compared to the case when block identities were not used at all.

This illustrates the flexibility of inductive logic programming. If the user has partial knowledge, intuitions or expectations about the hypotheses to be induced, they can be elegantly encoded using a combination of background theory and declarative bias. If one does not possess such knowledge, one may have to search a larger space, may require more examples and time to identify the target concept, and in the worst case, learning might be unsuccesfull.

5. Relational reinforcement learning

5.1. The need for relational representations

Given the framework for Q- and P-learning presented in Section 3, we could now learn to plan in the blocks world sketched earlier. Using the approach as it stands we could store all the state-action pairs encountered and memorize/update the corresponding Q values, having in effect an explicit look-up table for state-action pairs. This is how Pat Langley initially addresses the relational reinforcement learning task in his book (Langley, 1996), cf. also Section 7.2. The P values could then be derived from this. This approach has however a number of disadvantages:

– It is impractical for all but the smallest state-spaces. Furthermore, using look-up tables does not work for infinite state spaces which could arise when first order representations

are used (e.g., if the number of blocks in the world is unknown or infinite the above method does not work).
– Despite the use of a relational representation for states and actions, the above method is unable to capture the structural aspects of the planning task.
– Whenever the goal is changed from say $on(a, b)$ to $on(b, c)$ the above method would require retraining the whole Q function.
– Ideally, one would expect that the results of learning in a world with 3 blocks could be (partly) recycled when learning in a 4 blocks world later on. It is unclear how to achieve this with the lookup table.

The first problem can be solved by using an inductive learning algorithm (e.g., a neural network as in (Langley, 1996)) to approximate Q and P. The three other problems can only be solved by using a *relational* learning algorithm that can abstraction from the specific blocks and goals using variables. We now present such a relational learning algorithm called RRL. The main contribution of this paper is to address the *generalization* problem for reinforcement learning in a relational setting.

5.2. The task of relational reinforcement learning

We have already considered reinforcement learning (Section 2.1), its application to planning (Section 2.2), and relational learning (Section 4). We give a more precise definition of the relational reinforcement learning (RRL) task below. The RRL task is a reinforcement learning task (items 1 to 4), where states, actions and policies are represented relationally, and consequently, background knowledge and declarative bias are employed during learning (items 5 and 6). We illustrate the task formulation within the blocks world that will be used in our experiments. We want to emphasize though that RRL is a general approach and is applicable to domains other than planning in the blocks world.

The RRL task is specified as follows:

Given are:

1. *A set of possible states S, described in a relational language.* States are represented as sets of basic facts that hold in a state. The closed-world assumption is applied to state descriptions. In the blocks world, the basic facts concern the predicates $on(A, B)$ and $clear(A)$. The RRL algorithm encounters states one by one and does not see the entire set a priori.
2. *A set of possible actions A, also represented in a relational language.* In the blocks world, one can move one block onto another $move(A, B)$ or to the floor $move(A, floor)$. Not all actions are applicable in all states. The RRL algorithm sees only the actions applicable in a given state, as specified by the function $pre{:}S \times A \rightarrow \{true, false\}$. It is defined in Table 1 for the blocks world.
3. *A transition function* $\delta{:}\ S \times A \rightarrow S$. For the deterministic block world, this function is defined in Table 1. The RRL algorithm, however, does not rely on knowledge about this function. It only uses it to execute actions and move to new states. This function can in principle be nondeterministic (e.g., a move action might actually fail and not change the current state).

4. A real-valued reward function $r : S \times A \rightarrow R$. At present we use the goal function *goal*:$S \rightarrow$ {*true*, *false*} to define $r : r(S, a) = 1$ if $goal(\delta(S, a)) = true$, $r(S, a) = 0$ otherwise.
5. *Background knowledge generally valid about the domain* (*states in S*). This includes predicates that can derive new facts about a given state. In the blocks world, a predicate *above*(A, B) may define that a block A is above another block B.
6. *Declarative bias for learning relational representations of policies.* Together with the background knowledge, this specifies the language in which policies are represented. In the blocks world, e.g., we do not allow policies to refer to the exact identity of blocks.

The task is to **find** a policy for selecting actions $\pi : S \rightarrow A$ that maximizes the expected discounted reward. Policies can be either represented as real-valued Q-functions or as binary (optimal/non-optimal) classifier policies (P-functions).

5.3. The Q-RRL algorithm

The relational reinforcement learning (Q-RRL) algorithm is obtained by combining the classical Q-learning algorithm with stochastic selection of actions and a relational regression algorithm. Instead of having an explicit lookup table for the Q-function, an implicit representation of this function is learned in the form of a logical regression tree, called a Q-tree.

The Q-RRL algorithm is given in Table 3. The main point where RRL differs from the algorithm in Section 3.2 is in the for-loop where the $\hat{Q}$-function is modified.

The initial tree $\hat{Q}_0$ assigns zero value to all state-action pairs. From each goal state g encountered, an example (g,a,0) is generated for each action a whose preconditions are satisfied in g. The rationale for this is that no reward can be expected from applying an action in an absorbing goal state.

A possible initial episode ($e = 1$) in the blocks world with three blocks a, b, and c, where the goal is to stack a on b (i.e., $goal(on(a, b))$) is depicted in figure 3. The discount factor γ is 0.9 and the reward given is one on achieving a goal state, zero otherwise.

The examples generated by RRL use the actions and the Q-values listed above the arrows representing the actions. The actual format of these examples is listed in Table 4. It is exactly

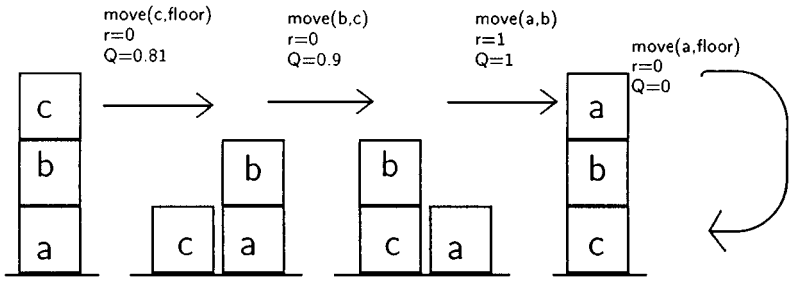

Figure 3. A blocks-world example for relational Q-learning.

Table 3. The Q-RRL algorithm for relational reinforcement learning.

Initialize $\hat{Q}_0$ to assign 0 to all (s, a) pairs
Initialize Examples to the empty set.
e := 0
do forever
  e := e + 1
  i := 0
  generate a random state s_0
  while not goal(s_i) **do**
    select an action a_i stochastically
      using the Q-exploration strategy from Equation (1)
      using the current hypothesis for $\hat{Q}_e$
    perform action a_i
    receive an immediate reward $r_i = r(s_i, a_i)$
    observe the new state s_{i+1}
    i:=i+1
  endwhile
  for j=i−1 to 0 **do**
    generate example $x = (s_j, a_j, \hat{q}_j)$,
      where $\hat{q}_j := r_j + \gamma max_{a'} \hat{Q}_e(s_{j+1}, a')$
    if an example $(s_j, a_j, \hat{q}_{old})$ exists in Examples, replace it with x,
      else add x to Examples
  update $\hat{Q}_e$ using TILDE-RT to produce $\hat{Q}_{e+1}$ using Examples

Table 4. Examples for TILDE-RT generated from the blocks-world Q-learning episode in figure 3.

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).	

this input that is used by TILDE-RT to generate the Q-tree $\hat{Q}_1$. TILDE-RT (De Raedt & Blockeel, 1997; Blockeel et al., 1998) is an algorithm for learning logical regression trees (as described in Section 4).

TILDE-RT is not incremental, so we currently simulate the update of $\hat{Q}$ by keeping all (s, a) pairs encountered[1] (not just those encountered in episode e) and the most recent $\hat{q}$ value for each pair. In non-deterministic domains, it would probably be a good idea to average the $\hat{q}$ values instead of keeping only the most recent value. A relational regression tree $\hat{Q}_e$ is induced from the (s, a, q) examples after each episode e. The tree $\hat{Q}_e$ is then used to select actions in episode $e + 1$.

```
root : goal_on(A,B) , numberofblocks(C) , action_move(D,E)
on(A,B) ?
+--yes: [0]
+--no: clear(A) ?
       +--yes: [1]
       +--no: clear(E) ?
              +--yes: [0.9]
              +--no: [0.81]
```

```
qvalue(0) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , on(A,B), !.
qvalue(1) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , clear(A), !.
qvalue(0.9) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , clear(E), !.
qvalue(0.81).
```

Figure 4. A relational regression tree and its equivalent Prolog program generated by TILDE-RT from the examples in Table 4.

In order to apply TILDE-RT to induce a Q-tree, the input for TILDE-RT is a set of state-action pairs together with the corresponding Q-values, represented as sets of facts. From these, TILDE-RT induces (using the techniques sketched in Section 4) a relational regression tree in which the predictions correspond to the real numbered Q-values.

To illustrate the above notions, consider the episode shown in figure 3. The examples for TILDE-RT generated by the RRL algorithm are given in Table 4. The corresponding relational regression tree induced by TILDE-RT from these examples, using the background knowledge listed in Appendix B, is shown in figure 4. This tree is a logical regression tree as described in Section 4. There is one slight difference with the trees introduced in Section 4 and this is the use of the root of the tree. The root of the tree in all decision trees shown below contains a query that succeeds in all examples. The reason for having a root is that this allows to bind the relevant variables (in this case the goal, possibly the numberofblocks, and the action under consideration). Because the root query succeeds for all examples it is propagated to all nodes in the decision trees. Furthermore it appears in all Prolog clauses derived from the decision trees.

To find the Q-value corresponding to a state-action pair, one has to construct a Prolog knowledge base containing the Prolog program (corresponding to the tree), all facts in the state, the action, and the goal. Running the query `?-qvalue(Q)` will then return the desired result. E.g., the Q-tree above will return a Q-value of zero for all actions if the goal is *on*(A, B) and `on(A,B)` holds in the state (goal states are absorbing). On the other hand, if the goal *on*(A, B) does not yet hold and A is clear, all actions get a value of one.

```
qvalue(0) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , on(A,B), !.
qvalue(1) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , action_move(A,B), !.
qvalue(0.729) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , height(D,F) , height(E,G) , F < G, !.
qvalue(0.81) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , eq(E,A), !.
qvalue(0.81) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , eq(E,B), !.
qvalue(0.81) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(A,B), !.
qvalue(0.9) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , height(D,F) , F = 3 , on(D,A), !.
qvalue(0.81) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , height(D,F), F = 3, !.
qvalue(0.9) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , clear(B), !.
qvalue(0.9).
```

Figure 5. An optimal Q-tree generated by Q-RRL in the three blocks world.

Figure 5 lists the Prolog rewrite of a Q-tree that is optimal for the three blocks world and has been induced by Q-RRL after 10 episodes. The tree was induced using the background knowledge listed in Appendix B. The settings used for TILDE-RT can be found in Appendix C. It is important to note that the individual blocks are not referred to in the tree itself directly, but only through the variables of the goal. This means that the tree represents the optimal policy not only for achieving the goal $on(a, b)$, but also $on(b, c)$ and $on(c, a)$. This is one of the major advantages of using a relational representation for Q-learning.

5.4. The P-RRL algorithm

In the previous sections, we showed how the Q-function could be approximated by Q-trees. In this section, we show how an approximation of the P-function, called the P-tree, can be obtained.

One approach to approximating the P-function would be to directly apply the definition of the P-function in terms of the Q-function, where the Q-function in the definition is replaced by the induced Q-tree as sketched in Section 3.3. However, as the Q-function is typically more complex than the P-function, this would lead to an unnecessarily complex and indirect definition of the P-function. As the definitions of both P and Q will be learned it may well turn out easier and more effective to learn P than to learn Q. This will only be the case when the induced P-function does not refer to the Q-function. The P-function can be represented

Table 5. Learning P-trees from Q-trees within the P-RRL algorithm.

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

as a logical decision tree, a P-tree, that predicts whether the state action pair is optimal (P-value is 1) or non-optimal (P-value is 0). The P-RRL algorithm learns P-trees in addition to Q-trees. It is identical to the Q-RRL algorithm with the following two exceptions: 1) to learn the P-tree, the code in Table 5 is added at the end of the **do forever** loop in Table 3 and 2) the P-exploration strategy as defined by Eq. (2) is used to select actions.

All state-action pairs for which the state was encountered in the last episode are classified as optimal or non-optimal according to the induced Q-tree. The resulting examples are then fed into the TILDE system that will induce a logical decision tree. The only difference between a logical decision tree and a logical regression tree is the information in the leaves. The leaves of regression trees contain real numbers, whereas those of decision trees contain classes.

The initial tree $\hat{P}_0$ assigns value one to all state-action pairs. From each goal state g encountered, an example (g,a,0) is generated for each action a whose preconditions are satisfied in g. The rationale for this is that no reward can be expected from applying an action in an absorbing goal state, hence no action in a goal state is optimal.

If we look back at the examples of figure 3, and apply the P-RRL part of the algorithm, the examples in Table 6 would be generated. These could then be fed into the TILDE system that could then induce a logical decision tree.

A P-tree in Prolog format generated by P-RRL from the examples in Table 6 is shown in figure 6. The same background knowledge is used as for inducing Q-trees. Although induced from one episode only, this tree comes close to the correct optimality tree for this domain. If the goal is `on(A,B)` and there is a block above `A` (`above(D,A)`) it is optimal to move `D` away (`action_move(D,E)`). The only exception to this is when we move `D` to `B`: this exception is provided for by the first clause of the tree in figure 7, which was induced by TILDE during the experiments described in Section 5 and is equivalent to the correct tree.

```
root : goal_on(A,B) , numberofblocks(C) , action_move(D,E)
above(D,A) ?
+--yes: optimal
+--no: action_move(A,B) ?
       +--yes: optimal
       +--no: nonoptimal
```

Figure 6. A P-tree for the three blocks world generated from the examples in Table 6.

Table 6. Examples for learning a P-tree by TILDE generated from the blocks-world Q-learning episode in figure 3.

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).

```
nonoptimal :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(D,A) , eq(E,B), !.
optimal :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(D,A), !.
optimal :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , action_move(A,B), !.
nonoptimal :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , on(B,E) , clear(B), !.
nonoptimal :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , on(B,E) , on(A,B), !.
optimal :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , on(B,E), !.
nonoptimal.
```

Figure 7. An optimal P-tree generated by P-RRL in the three blocks world.

Note that while we have chosen to use the logical decision and regression tree inducers TILDE and TILDE-RT, other relational regression (Karalic & Bratko, 1997; Kramer, 1996) and classification (Quinlan, 1990; Kramer, 1996) approaches can be used to induce relational representations of Q-functions and policies in the Q-RRL and P-RRL algorithms.

6. Experiments

6.1. Questions addressed

The experiments described in this section will attempt to answer several questions about relational reinforcement learning. We will focus on the following ones:

1. Is relational reinforcement learning effective for different goals?
2. Can P-RRL and Q-RRL learn optimal policies for state spaces with a fixed number of blocks?
3. Can P-RRL and Q-RRL learn optimal policies for state spaces with different numbers of blocks?
4. Can P-RRL and Q-RRL learn from experience in which the number of blocks is varied?
5. Is P-RRL to be preferred over Q-RRL?
6. Under which conditions does relational reinforcement learning work?

6.2. Experimental setup

We performed two different sets of experiments. In the first set of experiments, the policies were learned from state spaces in which the number of blocks was held constant, cf. Section 6.3. In the second set of experiments, discussed in Section 6.4, we varied the number of blocks while learning.

In both sets of experiments we tried out different goals such as stacking, unstacking and $on(a, b)$ (cf. Section 6.2.1 for a discussion on the goals pursued) and used the background knowledge and parameter settings discussed in Section 6.2.2, except for the experiments described in Section 6.6.

6.2.1. Setup: The tasks. The following goals in the block's world were pursued:

– stack: goal reached if all blocks are on one stack
  (`?- not (on(A,floor), on(B,floor), not A=B)` in Prolog)
– $on(a, b)$: goal reached if block a is on block b
– unstack: goal reached if all blocks are on the floor
  (`?- not (on(A,B), not B=floor)`)

Prolog code specifying the optimal policies for achieving these goals is given in Table 7. The optimal policy for unstacking is the simplest: moving any block (that is not already on the floor) to the floor is optimal. The policy for stacking is a bit more complex: moving a block to the highest stack is optimal. The policy for achieving $on(a, b)$ is the most complex: if possible, a should be moved to b; otherwise, a block above a or b should be moved away (but not to the stack where b or a are).

The above ordering of the three goals also corresponds to the number of reachable goal states, in decreasing order. A reachable goal state is a goal state (where the goal is satisfied) and which can be reached from a non-goal state in a single step (by applying one action).

Table 7. Optimal policies for three goals in the blocks world.

```
optimal(unstack,move(A,floor)) :-
    on(A,B),
    B\=floor.

optimal(stack,move(A,B)) :-
    height(B,HB),
    not (height(C,HC), HC > HB).

optimal(onab,move(A,B)) :-
    goal(on(A,B)).

optimal(onab,move(X,Y)) :-
    goal(on(A,B)),
    above(X,A), not above(Y,B).

optimal(onab,move(X,Y)) :-
    goal(on(A,B)),
    above(X,B), not above(Y,A).
```

Table 8. Number of states and number of reachable goal states for three goals and different numbers of blocks.

No. of blocks	No. of states	RGS stack	RGS on(a,b)	RGS unstack
3	13	6	2	1
4	73	24	7	1
5	501	120	34	1
6	4 051	720	209	1
7	37 633	5 040	1 546	1
8	394 353	40 320	13 327	1
9	4 596 553	362 880	130 922	1
10	58 941 091	3 628 800	1 441 729	1

For the goal `on(a,b)`, e.g., the state $s_1 = \{clear(a), on(a, b), on(b, c), on(c, floor)\}$ is a reachable goal state, while $s_2 = \{clear(c), on(c, a), on(a, b), on(b, floor)\}$ is not a reachable goal state.

For the unstack goal and a fixed number of blocks there is only a single state that satisfies the goal. For the stack goal, given n blocks there are $n!$ goal states. The number of possible states increases exponentially with the number of blocks. This is summarized in Table 8.

One point that should be clear from this table is that for some of the goals (e.g., unstacking with 10 blocks) the reinforcement learning algorithm described in Section 3.2 is inapplicable: the probability of reaching the goal state by random exploration is extremely low (given only 1 goal state in 58 941 091). Another point that this table demonstrates is the difficulty of learning policies in the blocks world. Despite the fact that the blocks world is an artificial toy domain, policy learning can become very complex due to the large number of possible states.

Note also that for unstack, the number of possible actions increases as one gets closer to the goal states: one step away from the goal state there are $(n-1)(n-2)+1$ possible

actions. For stack, on the other hand, there are only two possible actions if we are one step away from a goal state.

6.2.2. Background knowledge and parameters. The background knowledge and the settings used by TILDE-RT and TILDE are listed in Appendix B and C. It includes the predicates `above(A,B)` (block A is above block B, transitive closure of the relation `on(A,B))`, `eq(A,B)` (equality, `A=B`), `height(A,H)` (the height of block `A` is `H`), `numberofblocks(N)`, `numberofstacks(M)` and `diff(X,Y,Z)` (subtraction, `Z=X-Y`). The same background knowledge is used for both TILDE and TILDE-RT. There is a slight difference in the settings: when learning policies, TILDE is not allowed to use constants for the heights of stacks and the number of stacks. E.g., it can compare the heights of two stacks (needed for the stacking policy), but cannot check directly if there is a stack of height 4. The same background knowledge and settings are used for the three problems, with the only difference of placing the corresponding goal literal in the root of the tree (`goal_on(A,B)` for *on*(*a*, *b*), `goal_stack` for stack, `goal_unstack` for unstack).

In the following sections, we describe experiments with the P-RRL algorithm, which subsumes the Q-RRL algorithm. The P-exploration policy was used throughout. The starting temperature was set to 5 in Eq. (2).

6.3. Fixed number of blocks while learning

Our first set of experiments investigates whether relational reinforcement learning can find optimal policies for the three goals mentioned above when keeping the number of blocks fixed during learning. The learned policies can then be evaluated in two ways depending on whether the number of blocks is fixed during evaluation or not.

6.3.1. Evaluating learned policies on fixed number of blocks. Three learning experiments were conducted for each goal, one with 3, one with 4 and one with 5 blocks. Within each scenario, 5 runs of 30 episodes each of the P-RRL algorithm were performed. The quality criteria described below were recorded after each episode and averaged over the five runs, e.g., over the first episode of each run, over the second episode, etc. It is these averages that are depicted in the graphs on the figures below.

For each of the three tasks and each number of blocks, the learned policies were evaluated on the same number of blocks (same state space) they were learned on. Two different quality criteria were applied, which are feasible to calculate for small numbers of blocks.

The first is the Root Mean Square (RMS) of the error between the value function defined by the learned Q-function and the optimal value function. The second is the accuracy of the policy represented by the learned Q-function. The accuracy is defined as the percentage of state action pairs that are correctly classified as optimal or non-optimal by using the learned Q-function. For reference, the accuracy of the random policy (that selects an applicable action at random) is given in Table 9.

The learning curves for the RMS error and the policy accuracy are depicted in figure 8. For the latter, standard deviations are also given. These results clearly show 1) that optimal or close to optimal policies are rapidly learned in the case of stacking and unstacking; and 2)

Table 9. Accuracy of the random policy.

	3 Blocks	4 Blocks	5 Blocks
Stacking	42.9	37.6	32.2
Unstacking	66.7	55.1	49.0
On(a,b)	61.7	55.6	50.9

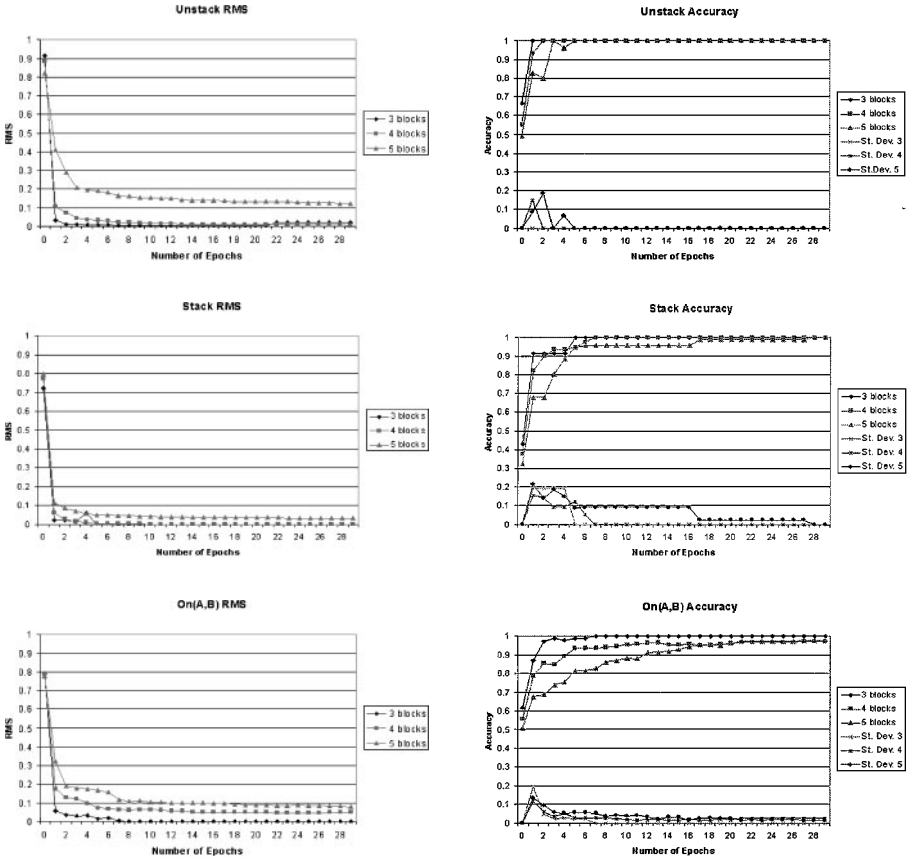

Figure 8. Learning curves for three different goals in the blocks world. The RMS of the error between the optimal and learned value function and the accuracy of the learned policy are measured. For the latter, standard deviations are also given.

that the difficulty of the learning task increases and the performance of the learner decreases with the number of blocks (e.g., for stack and especially $on(a, b)$ with 5 blocks).

One important point about relational reinforcement learning is that for the goal $on(a, b)$ the results remain exactly the same when the goal is varied to say $on(c, d)$. This is because the P- and Q-trees abstract away the name of the blocks by using variables.

6.3.2. Evaluating learned policies on varying number of blocks. As the number of blocks increases, the number of states in the blocks world increases very fast (cf. Table 8) and it becomes impractical to calculate the RMS of the value function and the policy accuracy over the entire state space. We thus take a random sample of states. Exploiting the learned policy, we start in each of the selected states and generate a plan for achieving the selected goal by choosing an optimal action proposed by the policy. A plan generated in this fashion is optimal if it has the same number of actions as a plan generated for the same starting state and goal by using the optimal policy (see Table 7). The quality measure that we consider here is optimality, defined as the percentage of states in the sample for which an optimal plan is generated.

To estimate the optimality, we randomly generated 3 samples of 156 states, one for each goal, where states could have 3 to 10 blocks. We took 3*n states with n blocks where the goal pursued was not satisfied. We thus took $3 * 3 = 9$ states with 3 blocks, 12 states with 4 blocks, . . . , and 30 states with 10 blocks, a total of 156 states.

We exploited the policies represented by the Q- and P-functions (referred to as Q-policies and P-policies) learned by the P-RRL algorithm in the previous subsection. The policies were tested on the set of 156 states appropriate for the selected goal and the accuracy was recorded as the percentage of states in which the goal was reached in the optimal number of steps. As in the subsection above, the results were averaged over the 5 runs. The learning curves for the Q-policies and P-policies are given in figure 9.

From this figure, we can conclude that:

1. Note first that both the Q-policies and the P-policies tested here perform well on the state spaces where they were learned (with fixed number of blocks—3, 4, or 5, cf. figure 8). Here we are testing them on a new, much larger state space than the one they were trained on and it is natural that they will perform worse.
2. When learning from 3 blocks, the Q-policies rapidly converge to those optimal for 3-block states and reach a plateau (between 20% and 40%) of optimality. The reason for this low performance is that the Q-values basically encode the number of steps from the goal when executing the specified action in the given state. These numbers depend—of course—on the number of blocks. When learning from 4 and 5 blocks, optimality improves as the number of episodes increases, albeit slowly, and reaches around 60%. The notable exception is learning stacking with 4 blocks, where optimality of over 90% is reached.
3. The P-policies seem to converge to optimal or close to optimal strategies when learning with a sufficiently large number of blocks (4 or 5), with convergence being fastest for unstacking and slowest for $on(a, b)$. A look at the optimal policies for each goal (listed in Table 7) makes this easier to understand: the unstacking policy is simplest and the $on(a, b)$ policy the most complicated of the three. When learning with 3 blocks, policies that are optimal for three-block states are learned, which however do not generalize to states with higher numbers of blocks (except for unstacking). When learning with higher number of blocks convergence to the optimal or close to optimal strategies slows down. Furthermore, the P-function does not get stuck on plateaus. The P-function for $on(a, b)$ does not reach optimality in the 30 episodes, but makes constant progress. We take a closer look at the $on(a, b)$ problem in Section 6.6.

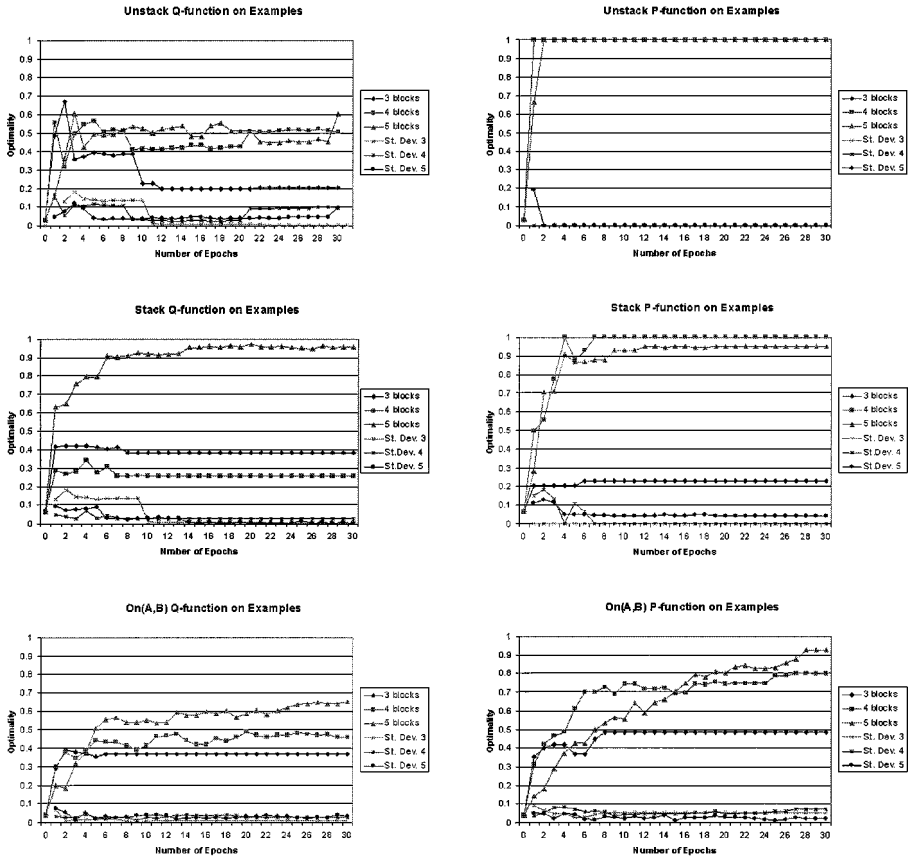

Figure 9. Learning curves for three different goals in the blocks world. The same policies as in figure 8 are evaluated. The percentage of optimal plans generated for a sample of 156 starting states with different numbers (3 to 10) of blocks is depicted.

4. The P-policies perform much better than the Q-policies on the new state space. This is not surprising, as they do not make direct reference to the number of steps to a goal state and thus depend less on the number of blocks.

For illustration, Appendix D lists the P- and Q-policies learned after the 30 episodes of the last (fifth) run of each experiment. We can see immediately that the P-policies have a much shorter representation. The P-policies for unstack and stack are recognizably optimal. The P-policy for unstack states that an action is nonoptimal if it moves a block onto another block (only blocks can be clear!), otherwise (action moves a block to the floor) the action is optimal. The P-policy for stack states that an action `move(B,C)` is nonoptimal if the stack with `C` on top is not the highest stack around (there is a stack with `E` on top that is higher).

6.4. Varying the number of blocks while learning

In a second set of experiments we varied the number of blocks while learning. Three experiments were performed, one for each goal.

The results were averaged over 10 runs and each run consisted of 45 episodes. Each run started with 5 episodes involving states with 3 blocks, followed by 15 episodes involving states with 4 blocks, followed by 25 episodes involving states with 5 blocks. The temperature was decreased by a factor 0.95 after every episode to stimulate the use of learned knowledge when the learning problem becomes harder. The learned policies were evaluated for a varying number of blocks, as described in Section 6.3.2. The results are shown in figure 10.

Let us first look again at the results of stacking and unstacking. The graphs clearly show that the learned P-trees are close to optimal even though the Q-trees are not optimal. Furthermore, when increasing the number of blocks (after episodes 5 and 20) there is a temporary decrease in performance of the learned policies (a small one for the P-trees and a more significant one for Q-trees). This is due to the changes in the Q-function that occur when the state-space changes. The Q-function depends on the number of steps to the goal state. When the number of blocks is increased the possible distance to the goal also increases and the Q-function has to be adapted. This is somewhat related to the notion of concept drift (Widmer & Kubat, 1998).

After 30 episodes, the optimal P-function for stacking is learned, which was not the case for learning from states with 3 or 5 blocks only. This seems to indicate that relational reinforcement learning can bootstrap itself. The result of learning on easier tasks can—indeed—be used to attack harder tasks. As indicated in Table 8, the probability of finding the goal state in the world with 10 blocks can be close to zero. However, using the sketched procedure, starting from simple states and gradually increasing the difficulty of the problem, the optimal policies can be learned.

There is a notable decrease of performance of the P-policy for $on(a, b)$ after switching from 4 to 5 blocks (after episode 20). The fall in performance is not reversed in the remaining 25 episodes, although the Q-policy improves slowly but steadily. In fact, the performance of the P-policies is worse than with learning from 4 or 5 blocks alone. We examine this issue in more detail in Section 6.6. First, however, we try to explain the differences between the P- and Q-trees.

6.5. Q-learning versus P-learning

The previous experiments clearly indicate that the P-trees almost consistently outperform the Q-trees for stacking and unstacking. The explanation for this relies on two observations. First, as already mentioned above, the Q-trees encode the number of steps from the goals, whereas the P-trees only encode whether a certain step is optimal or not. The optimal action in a given state typically does not rely on the number of blocks or the number of steps from the goal, but rather on the properties of the state and action. This is evident from the optimal policies listed in Table 7. Therefore, P-trees learned for states with a sufficiently large number of blocks are likely to behave nicely on problems with a different number of

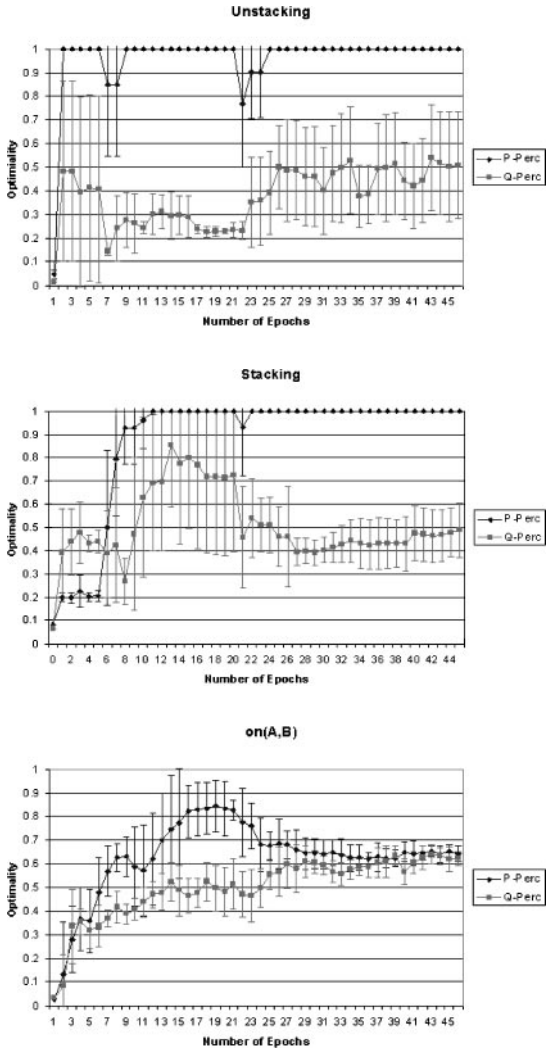

Figure 10. Learning curves for three different goals in the blocks world, where the number of blocks is varied during learning. The percentage of optimal plans generated for a sample of 156 starting states with different numbers (3 to 10) of blocks is depicted. Error bars of one standard deviation are given.

blocks. This is not the case for Q-trees. This is somewhat related to the work on generalizing numbers in explanation based learning (Mitchell et al., 1986).

Secondly, the P-trees are always simpler than the Q-trees because they only need to distinguish two classes: optimal and not optimal, whereas Q-trees distinguish among many values. Finally, the reader may wonder why the P-trees perform better than the Q-trees even though the P-trees are derived from the Q-trees? To explain this, observe that the Q-trees are close to optimal for states with the same fixed number of blocks as used in the episodes

(with the exception of $on(a, b)$). The P-trees then abstract away from this number of blocks and the number of steps from the goal. This ability is entirely due to the use of inductive logic programming and gives an indication where reinforcement learning may benefit from using relational learning.

6.6. When does relational reinforcement learning work?

The previous experiments clearly showed that relational reinforcement learning works well for stacking and unstacking but less so for achieving $on(a, b)$.

The first question that arises is how good (or bad) the results on $on(a, b)$ really are. The analysis so far has only looked at optimal plans, a very stringent criterion. If a policy takes one action longer than necessary, this has been considered a complete failure in the accuracy figures presented so far. However, a non-optimal plan might still be of good quality. To investigate the quality of the generated policies, we evaluated them along two further criteria: 1) the proportion of states where the policy loops, and 2) the ratio of the numbers of actions taken by the policy and the optimal plan, respectively.

The evaluation of the policies for $on(a, b)$ learned as described in Section 6.4. along these two criteria is depicted in figures 11 and 12. Figure 11 shows the percentage of cases where more than 10 times the optimal number of number of steps were needed: the policy was considered to loop in this case and its execution was stopped. After a few episodes, the P-policies do not loop at all, while Q-policies still loop even after 45 episodes.

Figure 12 depicts ratio of the number of actions taken by the policy and the number of actions in the optimal plan. So 100% is the best one can score on this criterion. If the policy looped, it was stopped after 10 times the optimal number of steps. Even though the P-policies are not optimal, they come very close to optimality. Once they stop looping (after

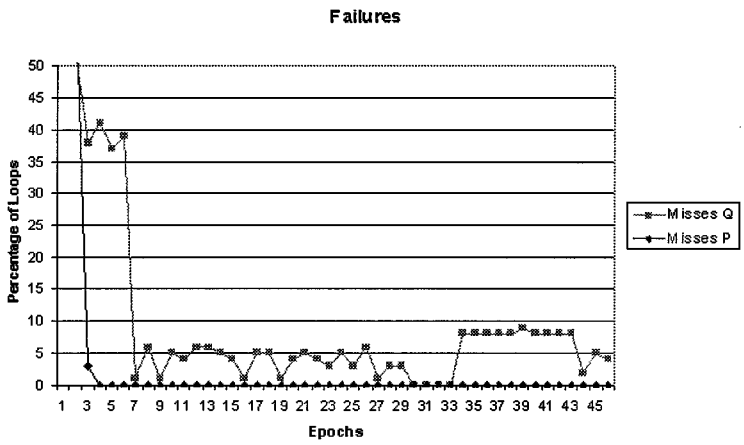

Figure 11. The percentage of states for which the learned policy for on(a,b) loops (takes more than 10 times the optimal number of steps).

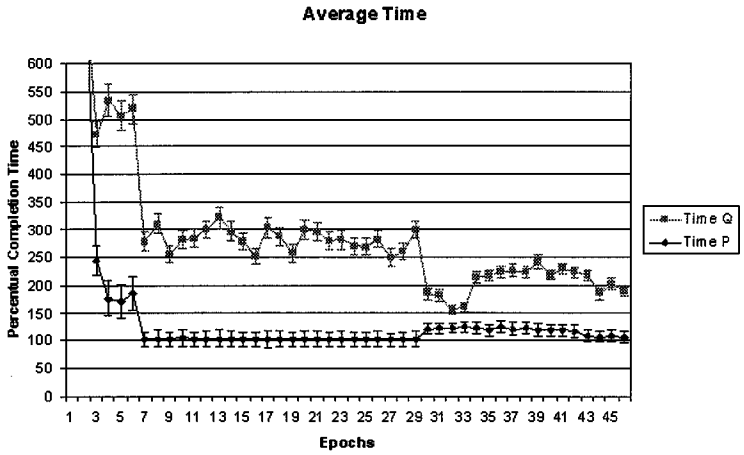

Figure 12. The ratio of the number of steps taken by the learned policy for on(a,b) and the optimal number of steps. Error bars of one standard deviation are given.

a few episodes), they take less than 1.5 times the optimal number of steps (on the average). The Q-trees perform consistently worse than the P-trees, but not really bad. The number of steps needed to reach the goal is about twice the optimal one in the late episodes.

Despite the fact that the problem with $on(a, b)$ is not as bad as it appeared at first sight, the question remains as to why relational reinforcement learning has problems learning the optimal policy. One already mentioned reason for this is that the optimal policy to achieve $on(a, b)$ is more complex (cf. Table 7). Indeed, in order to achieve $on(a, b)$ one first has to clear a and b and then to move a onto b, which is more complex than the other strategies. Though this fact might explain why learning for $on(a, b)$ is slower than for stacking or unstacking, it does not explain some other facts. In figure 10, the Q- and P-trees for $on(a, b)$ seem to perform equally well on states with a varying number of blocks. From the experiments for stacking and unstacking one would expect the P-trees to perform significantly better.

To investigate these anomalies, we performed some experiments where the goal of RRL was a subgoal of $on(a, b)$ namely $clear(a)$. The results of this test are shown in the first two curves in figure 13 (P-perc and Q-perc).

Although the P-tree performs a little better than the Q-tree, no optimal policy is learned. We then investigated the resulting Q- and P-trees. The learned Q-tree was very complicated and was clearly incorrect. The explanation for this is that—using the representation language and background knowledge available—TILDE-RT cannot represent the correct Q-tree. The reason is that the Q-tree actually implicitly encodes the number of steps from the goal state. For $clear(a)$ this means that one has to know the number of blocks that are above a. This was partly confirmed in another experiment and is illustrated in figure 14. We tried to learn the correct Q-function using a fixed number of blocks (4) and compared the generated values with the real ones. Although RRL was able to predict the correct actions as optimal or not (left part of figure 14), it was not able to represent the correct Q-values for the entire state-space (right part of figure 14, RMS greater than zero).

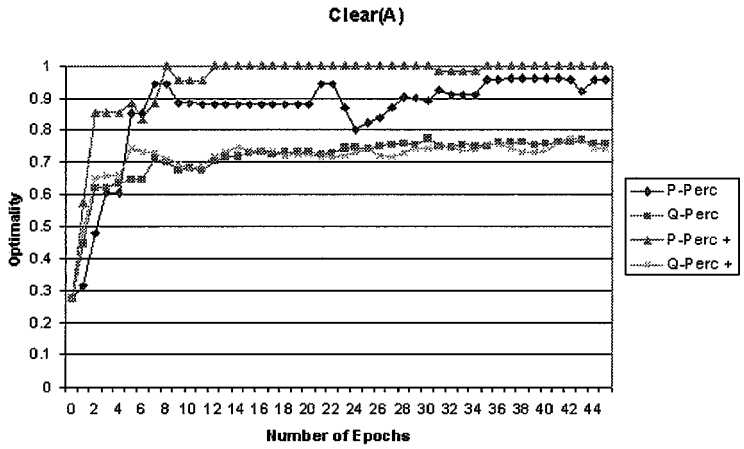

Figure 13. Learning curves for the goal clear(a) for the blocks world with a varying number of blocks during learning. The curves denoted with + are obtained using an additional background knowledge predicate.

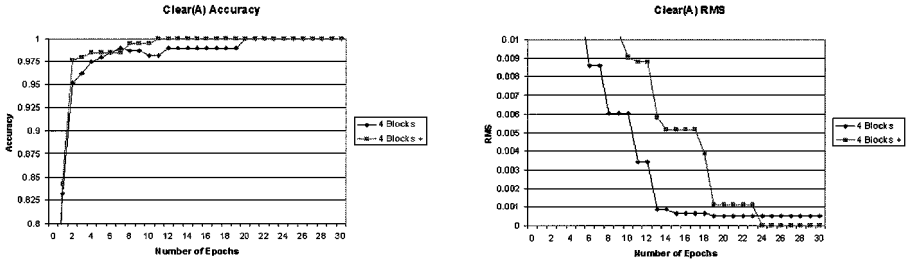

Figure 14. Learning curves for the goal clear(a) in the 4 blocks world. The curves denoted with + are obtained using an additional background knowledge predicate.

To test this hypothesis, we ran the relational reinforcement learning algorithm P-RRL again, but this time we added the predicate `numberofblockson(X,N)` (there are N blocks on top of block X) to the background theory and modified the mode and type declarations accordingly. The results are shown in figure 13 under the Q+ and P+ curves. What is surprising is that although the Q tree performs equally well as the Q+ tree, the P+ tree is optimal.

A further experiment was carried out in which a Q+ tree was learned and tested on states with a fixed number of blocks (4) (cf. figure 14). It turns out that the resulting Q+ trees outperform the Q-trees. More specifically, the Q+ trees for *clear*(*a*) were correct for all states with four blocks (RMS equal to zero), whereas the Q-trees were not. This experiment indicates that in order for relational reinforcement learning to work one must first get the Q-trees correct for states with a fixed number of blocks, and then the P-trees will abstract away to a variable number of blocks. This experiment also confirms that one needs the right representations for learning. In the context of relational learning and relational

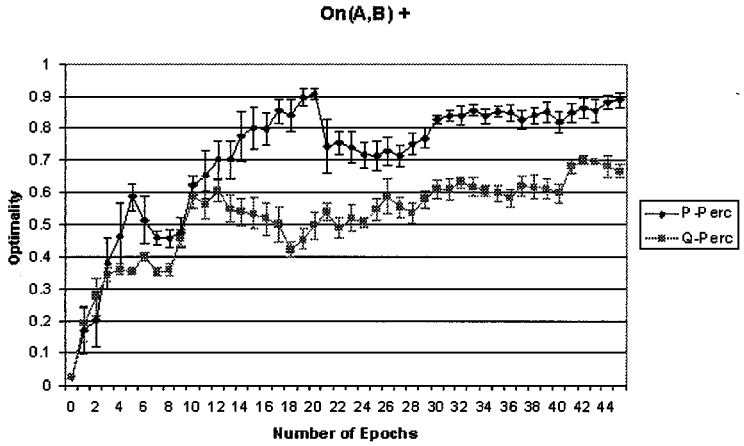

Figure 15. Learning curves for the goal on(a,b) in the blocks world, where the number of blocks is varied during learning. The curves denoted with + are obtained using an additional background knowledge predicate.

reinforcement learning this translates to the requirement that the ensemble of background theory and bias must allow to encode the Q- and P-trees.

Finally, let us take a look at the learning curves for the goal $on(a, b)$ in the blocks world, shown in figure 15 where the number of blocks is varied during learning and the additional background knowledge predicate is used. With the new predicate in the background knowledge, steady improvement of performance can be observed for the P-function after the 20-th episode, which was not the case previously.

6.7. Efficiency

Concerning the efficiency of the relational reinforcement learning algorithm, one has to distinguish between the different goals. The number of training examples for TILDE/TILDE-RT are different for the different goals. Figure 16 shows how the total number of learning examples increases per episode for the different goals. For the P-trees, more learning examples are generated than for the Q-trees. This is because the examples for the P-tree are generated looking at every possible action at every visited state, instead of just the actions executed at that state in the case of the Q-tree.

There is also a large difference between the number of learning examples for the different goals. The reason for this is the large difference in the number of possible actions when one approaches the goal-state. As stated in Section 6.2.1, one step away from the goal in a state space with n blocks there are $(n-1)(n-2)+1$ possible actions when unstacking, while there are only three actions when stacking. This difference is the reason for the large difference in the number of learning examples for the different goals.

This difference also influences execution time. Where 30 episodes with three blocks only take 6.25 minutes (total time required for learning) if the goal is stacking, the same

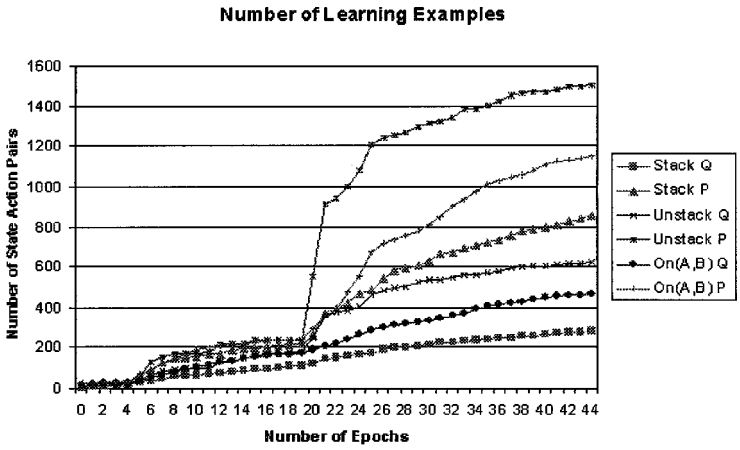

Figure 16. Number of learning examples after each episode for each of the three goals.

experiment with the unstacking goal takes 8.75 minutes. The same experiment again but with $on(a, b)$ as a goal requires 20 minutes. The larger time for the $on(a, b)$ experiment is due to the need for larger trees for both the Q-function and the policy.

When increasing the state-space from 3 to 4 blocks, the learning time grows to 62.4 minutes for stacking. The same test with 5 blocks already takes 306 minutes. When compared to the learning time for the experiment with a variable number of blocks (231 minutes, cf. Table 10), the gain from bootstrapping on easier problems is obvious.

The other timing results can be found in Table 10. Actually, testing the learned policies required more cputime than learning in the experiments we carried out. This justifies some of the choices we made in the experimental setup (e.g. the use of 'only' 156 states). Testing the generated trees takes a lot of time due to the same problem as discussed before. Testing the P-trees is faster because the first optimal action (out of randomly generated possible actions) is chosen, so not all actions have to be examined. The time needed for inducing the trees depends largely on the number of State/Action pairs used for TILDE, so the last tree induced uses the most cputime. RRL could be sped-up by making TILDE/TILDE-RT incremental. Various incremental decision tree algorithm exist and could be adopted within TILDE/TILDE-RT (e.g. (Utgoff et al., 1995)).

Table 10. Execution time of the RRL Algorithm on Sun Ultra 5/270 machines. The second column states the total accumulated time required for learning during the 45 episodes, the third and fourth column state the time required to test one Q- or P-tree, the last two columns list the time needed to induce the final P and Q-trees.

	45 Episodes	Testing		Induction of Final	
	3→5 Blocks	Q-tree	P-tree	Q-tree	P-tree
Stacking	3.85 hrs	16.1 min	10.2 min	4.85 min	3.85 min
Unstacking	10.1 hrs	45.1 min	16.7 min	18.8 min	6.61 min
On(a,b)	12.4 hrs	24.5 min	13.3 min	21.5 min	15.6 min

It should also be pointed out that special techniques have been developed within the data mining community to handle large data sets. These techniques have recently been incorporated in TILDE and TILDE-RT, cf. (Blockeel et al., 1999) and could improve the efficiency of RRL.

6.8. Summary of experimental results

To illustrate the advantages and limitations of RRL, we try to give brief answers to the questions posed in Section 6.1.

1. *Is RRL effective for different goals?* RRL was successfully used for stacking and unstacking, and after some representational engineering also for $on(a, b)$. Policies learned for $on(a, b)$ can be used for solving $on(A, B)$ for any A and B.
2. *Can P-RRL and Q-RRL learn optimal policies for state spaces with a fixed number of blocks?* Yes, though this becomes more difficult when the number of blocks increases.
3. *Can P-RRL and Q-RRL learn optimal policies for state spaces with a varying number of blocks?* Q-functions optimal for state spaces with a fixed number of blocks are not optimal for state spaces with a varying number of blocks. But we can learn optimal P-functions from the Q-functions. These P-functions often are optimal for state spaces with a varying number of blocks as well.
4. *Can P-RRL and Q-RRL learn from experience in which the number of blocks is varied?* Learning with a fixed number of blocks is increasingly difficult when we increase the number of blocks. Starting with a small number of blocks and gradually increasing this number allows for a bootstrapping process, where optimal policies are learned faster.
5. *Is P-RRL to be preferred over Q-RRL?* If Q-RRL doesn't work, then P-RRL won't work either. But once Q-RRL learns a Q-function that does the job right (even for states with a fixed number of blocks), one is better off using the P-function learned from the Q-function. The latter usually generalizes nicely to larger numbers of blocks than seen during training.
6. *Under which conditions does relational reinforcement learning work?* As general reinforcement learning, RRL works less well for goals that require more complex policies. However more appropriate background knowledge and more training might help in such cases.

7. Discussion

We have presented an approach to planning with incomplete knowledge that combines reinforcement learning and relational learning into a technique called relational reinforcement learning. The advantages of this approach include the ability to use structured representations, which enables us to also describe infinite worlds, and the ability to use variables, which allows us to abstract away from specific details of the situations (such as, e.g., the goal, the number of blocks). The ability to carry over the policies learned in simple situations (with few blocks) to more complex situations was demonstrated. It is hard to see how this could be realized without the use of relational representations.

We continue the discussion by discussing scalability, related work and further work.

7.1. Scalability

Even for standard reinforcement learning, scaling-up as the dimensionality of the problem increases can be a problem. Using a richer description language may seem to make things even worse. However, there are reasons to expect that using a richer representation actually enables relational Q-learning to scale-up better than standard Q-learning. Let us illustrate these on the blocks world.

First, in the representation employed, the relational theories learned abstract away the block names, causing the number of states that are essentially different to decrease. For instance, with $goal(on(a, b))$ the states $\{on(a, c), on(c, b), on(b, floor), on(d, floor)\}$ and $\{on(a, d), on(d, b), on(b, floor), on(c, floor)\}$ are essentially the same as c and d are interchangeable. In standard Q-learning, they would be considered different. In our 4-blocks example, the number of states that essentially differ from one another is 73 for a standard Q-learner, but only 38 for a relational one. This ratio increases combinatorially (since all blocks that do not occur in the goal have no special status and are thus interchangeable, the ratio increases roughly with $(n - 2)!$, where n is the total number of blocks).

Second, the use of background knowledge makes it possible to abstract even further from specific situations that do not essentially differ. For instance, when a has to be cleared in order to be able to move it, it is not essential whether there are 1, 5 or 17 blocks above a: the top of the stack on a should be moved. Using background definitions such as $above(X, Y)$, it is possible to state a rule such as "if there are blocks on a, move the topmost of those blocks to the floor" which captures a very large set of specific cases.

However, the exact scale-up behavior of relational reinforcement learning has still to be determined experimentally. The experimental evaluation of our approach done so far is mainly intended to highlight the principal advantages of using a relational representation for reinforcement learning. We hope that this paper will inspire further research into the combination of relational and reinforcement learning, as much work remains to be done. This includes considering more complex and demanding planning problems.

7.2. Related work

The main contribution of our work is to address the generalization problem in reinforcement learning within a relational setting. The task of finding optimal plans within the blocks world was already considered by Langley in his book (Langley, 1996), to illustrate reinforcement learning. However, instead of using a relational learner for generalization he employs a neural net using a fixed set of propositional features. Indeed, typical generalizers in reinforcement learning are based on neural nets, cf. e.g. (Tesauro, 1991), whereas we employ decision trees.

The use of decision trees in a reinforcement context is not new. It was first proposed by Chapman's and Kaelbling's (Chapman & Kaelbling, 1991), who developed an incremental decision tree learning algorithm with special heuristics to cope with concept-drift (Widmer & Kubat, 1998) in the reinforcement learning context. Our approach is distinguished from the one by Chapman and Kaelbling by the use of a relational representation. However,

it would be extremely useful to integrate algorithm by Chapman and Kaelbling (i.e. the heuristics and the incremental aspect) with the representations provided by TILDE and TILDE-RT.

Another piece of work that is very much related to our presentation of RRL is that of Baum (Baum, 1996). He uses a kind of genetic approach to learning rules in the blocks world for goals such as stacking and unstacking. To this aim he employs a special rule language (at a level between propositional and first order logic) for expressing policies and uses genetic ingredients to learn and modify the set of rules. Whereas this approach is elegant, it does not employ the basic principles of *temporal difference learning* as we do.

The combination of learning and planning has received a lot of attention in the artificial intelligence literature (see (Langley, 1996) for an excellent overview). Also, most approaches to learning in a planning context do employ relational representations. It can be no surprise that various types of learning tasks have been considered in this context.

- A first line of research attempts to improve the domain knowledge of the planner. This corresponds to learning more accurate definitions of the operators, i.e. the effects, pre- and post-conditions. This approach has been integrated in Prodigy, cf. (Carbonell & Gill, 1990). The planner then exploits the learned knowledge in order to construct better plans. The difference with relational reinforcement learning is that our approach does not rely on a planner. This is important as one might consider relational reinforcement learning outside a planning context.
- A second line of research concerns the learning of control knowledge. E.g. the work on LEX (Mitchell et al., l983) and SAGE (Langley, 1985) learned when to apply certain operators. The goal of this work is thus similar to that of relational reinforcement learning. However, the approach is quite different. E.g. to solve symbolic integration problems, LEX would construct a search tree (a trace) in which all legal operators were applied to a given integration problem until a solution was found at a certain depth. Once the solution was found, LEX would label all applications of operators on the path leading to the (optimal) solution as positive examples, and all applications diverging from this path as negative examples. The examples were then fed into a kind of relational learning algorithm and used to refine the control knowledge. So, the mechanism for learning is quite different. The LEX and SAGE method does not apply in the context of autonomous agents, because it assumes that one can backtrack to earlier states (which may not be possible—and which is certainly problematic without adequate domain knowledge). On the other hand, our method to construct examples for learning the P-tree is certainly similar in spirit to LEX.
- A third line of research analytically learns control knowledge often using a form of explanation based learning. The difference with relational reinforcement learning is that explanation based learning relies on complete knowledge about the domain.

The work by Stone and Veloso (Stone & Veloso, 1999) is closely related to ours in two ways. First, they use decision-trees to learn a Q-function, thereby generalizing. Second, they use a mapping on states to transform large state-spaces into learnable ones. This mapping

is hand-coded. Our approach uses a mapping which generalizes across state-action pairs; this mapping is implicitly defined by the relational representation and the background knowledge.

Somewhat related to our approach is work on hierarchical reinforcement learning, such as options (Sutton et al., 2000). Options are macro actions defined by a region of the state space where execution can begin, a policy and a termination condition. Options can be viewed as background knowledge, albeit different in nature from the one currently used by RRL.

7.3. Further work

The reinforcement learning part of the work presented in this paper is admittedly simple. We have taken a standard textbook description of reinforcement learning (Mitchell, l997) and incorporated an implementation of it within our approach. We have considered a deterministic setting and a goal-oriented formulation of the learning problem. However, both restrictions can be easily lifted to extend to non-zero rewards on non-terminal states (the RRL algorithm actually makes no assumption on the reinforcement received) and nondeterministic actions. To handle nondeterministic actions an appropriate update rule (see page 382 of (Mitchell, l997)) has to be used to generate examples for the TILDE-RT algorithm. Other points where the reinforcement learning part can be improved include the initialization of Q-values and the exploration strategy.

The current implementation of TILDE-RT is—according to reinforcement learning standards—not optimal. One of the reasons is that it is not incremental. However, incrementality is not sufficient, as the (estimated) values of Q are changing with time. These problems are taken care of by Chapman and Kaelbling's decision tree algorithm that was specifically designed for reinforcement learning (Chapman & Kaelbling, 1991). A natural direction for further work is thus to develop a first order regression tree algorithm combining the representations of TILDE-RT with the algorithm and performance measures of the approach by Chapman and Kaelbling. Such an integrated approach would not suffer from the abovementioned problems.

An interesting direction for further work would be the integration of relational reinforcement learning with some approaches to hierarchical reinforcement learning, such as options (Sutton et al., 2000). As mentioned above, options are macro actions defined by a region of the state space where execution can begin, a policy and a termination condition. Parametrized options, such as *clearblock*(A) would make sense in the RRL setting: the termination condition for this option would be *clear*(A). Such an option could also be learned. The use of such options could alleviate some of the problems encountered during our experiments with *on*(a, b).

Appendix A: TILDE and TILDE-RT algorithms

The pseudo-code for the TILDE and TILDE-RT algorithms is given in Table I.

Table I. Pseudo-code for the TILDE and TILDE-RT algorithms.

```
proc inducetree(E: examples)
   create a root node n for the tree t
   split(n,E,t )
   return t
   endproc

proc split(n: node; E: examples, t: tree)
   best := false
   for all possible tests q in node n do
      compute quality(q)
      if quality(q) is better than quality(best)
      then best := q
      endif
   if best yields improvements
   then
      test(n) := best
      create two subnodes n1, n2 of n in t
      E1 := {e ∈ E | e satisfies best in t}
      E2 := {e ∈ E | e does not satisfy best in t }
      call split(n1, E1, t)
      call split(n2, E2, t)
   else turn n into a leaf
   endif
   endproc
```

The TILDE and TILDE-RT algorithms are similar to classical decision trees except that only binary trees are induced and also that the computation of the possible tests in a node may depend on the variables in nodes higher in the tree. Also, when determining whether an example satisfies a test one must also take into account the tests higher in the tree. Finally, the heuristics employed by TILDE are the same as in C4.5, and those in TILDE-RT will minimize the variance of the target variable within each subnode and will maximize the variance among the two subnodes.

B. Background knowledge for TILDE

Besides the predicates *clear*(A) and *on*(A, B) used to represent states, the following predicates can be used in the trees induced by TILDE and TILDE-RT: *above*(A, B), *eq*(A, B), *height*(A, H), *number of blocks*(N), *number of stacks*(M) and *diff*(X, Y, Z). The same background knowledge is used for both TILDE and TILDE-RT. It is listed below.

```
eq(X,X).
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
action_move(X,Y) :- action(move(X,Y)).

goal_on(A,B) :- goal(on(A,B)).
```

```
goal_stack :- goal(\+ (on(A,floor),on(B,floor), A\=B)).
goal_unstack :- goal(\+ (on(A,B), B\=floor)).

diff(X,Y,Z) :- Z is X - Y.

height(floor,0).
height(A,H) :- block(A), height1(A,H).

height1(A,1) :- on(A,floor).
height1(A,H) :- on(A,B), B\=floor, height1(B,HB), H is HB+1.

numberofblocks(N):- myblocks(X), mylength(X,N).

numberofstacks(N):-mystacks(X), mylength(X,N).

myblocks(List) :- findall(X, block(X), List).

mystacks(List) :- findall(X, on(X,floor), List).

mylength(X,L) :- mylen(X,0,L).
mylen([],L,L) :- ! .
mylen([X | R],N,L) :- N1 is N + 1, mylen(R,N1,L).

block(X) :- on(X,Y).
```

C. Settings for TILDE and TILDE-RT

C.1. TILDE-RT settings

These are used for learning the Q-functions. Since the number of steps to the goal essentially defines the Q-function, heights of stacks and differences between these and the number of blocks, comparisons of these to constant values are allowed.

```
heuristic(eucl).
euclid(qvalue(X), X).

tilde_mode(regression).
confidence_level(1).
minimal_cases(1).
output_options([c45e,prolog]).

talking(0).

typed_language(yes).
type(clear(block)).
```

```
type(on(block,block)).
type(eq(block,block)).
type(above(block,block)).
type(action_move(block,block)).
type(height(block,number)).
type(numberofblocks(number)).
type(numberofstacks(number)).
type(number < number).
type(number = number).
type(diff(number,number,number)).
type(goal_on(block,block)).
type(goal_stack).
type(goal_unstack).
type(member(number,list)).


rmode(10: clear(+-X)).
rmode(10: on(+-X,+-Y)).
rmode(10: on(+-X, floor)).
rmode(10: eq(+X,+Y)).
rmode(10: eq(+X,floor)).
rmode(10: above(+-X,+-Y)).
rmode(10: action_move(+-X,+-Y)).
rmode(10: action_move(+-X,floor)).
rmode(10: (height(+-X,-H), height(+-X2,-H2), H < H2)).
rmode(10: (height(+-X,-H), height(+-X2,-H2), H2 < H)).
rmode(10: (height(+-X,-H), diff(+N,H,-D), height(+-X2,-H2),
                  diff(N,H2,-D2), D < D2)).
rmode(10: (height(+-X,-H), diff(+N,H,-D), height(+-X2,-H2),
                  diff(N,H2,-D2), D2 < D)).
rmode(10: #(C: member(C,[0,1,2,3,4,5,6,7,8,9,10]),
                  (height(+-X,-H), H = C))).
rmode(10: #(C: member(C,[0,1,2,3,4,5,6,7,8,9,10]),
                  (height(+-X,-H), H < C))).
rmode(10: #(C: member(C,[0,1,2,3,4,5,6,7,8,9,10]),
                  (height(+-X,-H), diff(+N,H,-D), D = C))).
rmode(10: #(C: member(C,[0,1,2,3,4,5,6,7,8,9,10]),
                  (height(+-X,-H), diff(+N,H,-D), D < C))).
rmode(10: #(C: member(C,[0,1,2,3,4,5,6,7,8,9,10]),
                  (numberofstacks(-S), S = C))).
rmode(10: #(C: member(C,[0,1,2,3,4,5,6,7,8,9,10]),
                  (numberofstacks(-S), S < C))).
rmode(10: #(C: member(C,[0,1,2,3,4,5,6,7,8,9,10]), (
                  numberofstacks(-S), diff(+N,S,-D), D = C))).
```

```
rmode(10: #(C: member(C,[0,1,2,3,4,5,6,7,8,9,10]),
                  (numberofstacks(-S), diff(+N,S,-D), D < C))).

root((goal_on(A,B), numberofblocks(N), action_move(X,Y))).
```

C.2. TILDE settings

These are used for learning the P-functions. Since the optimality of actions does not depend on the number of steps to the goal, comparisons of heights and the number of stacks to constants are not allowed.

```
heuristic(gain).
euclid(qvalue(X), X).

tilde_mode(classify).
classes([optimal,nonoptimal]).
confidence_level(1).
minimal_cases(1).
output_options([c45e,prolog,elaborate]).

talking(0).

typed_language(yes).
type(clear(block)).
type(on(block,block)).
type(eq(block,block)).
type(above(block,block)).
type(action_move(block,block)).
type(height(block,number)).
type(numberofblocks(number)).

type(numberofstacks(number)).
type(number < number).
type(number = number).
type(diff(number,number,number)).
type(goal_on(block,block)).
type(goal_stack).
type(goal_unstack).

rmode(10: clear(+-X)).
rmode(10: on(+-X,+-Y)).
rmode(10: on(+-X, floor)).
rmode(10: eq(+X,+Y)).
rmode(10: eq(+X,floor)).
```

```
rmode(10: above(+-X,+-Y)).
rmode(10: action_move(+-X,+-Y)).
rmode(10: action_move(+-X,floor)).
rmode(10: (height(+-X,-H), height(+-X2,-H2), H < H2)).
rmode(10: (height(+-X,-H), height(+-X2,-H2), H2 < H)).
rmode(10: (height(+-X,-H), diff(+N,H,-D), height(+-X2,-H2),
                              diff(N,H2,-D2), D < D2)).
rmode(10: (height(+-X,-H), diff(+N,H,-D), height(+-X2,-H2),
                              diff(N,H2,-D2), D2 < D)).
root((goal_on(A,B), numberofblocks(N), action_move(X,Y))).
```

D. Q-policies and P-policies induced in the 4-blocks world by the P-RRL algorithm

D.1. P-policy for unstack

```
ptree(nonoptimal) :- goal_unstack , numberofblocks(A) ,
   action_move(B,C) , clear(C), !.
ptree(optimal).
```

D.2. Q-policy for unstack

```
qtree(0.729) :- goal_unstack , numberofblocks(A) , action_move(B,C) ,
   height(D,E) , E = 2 , on(C,D), !.
qtree(0.9) :- goal_unstack , numberofblocks(A) , action_move(B,C) ,
   height(D,E) , E = 2 , action_move(D,floor) , numberofstacks(F) ,
   F = 2, !.
qtree(1) :- goal_unstack , numberofblocks(A) , action_move(B,C) ,
   height(D,E) , E = 2 , action_move(D,floor), !.
qtree(0.81) :- goal_unstack , numberofblocks(A) , action_move(B,C) ,
   height(D,E) , E = 2 , height(C,F) , height(B,G) , F < G ,
   height(B,H) , H = 4, !.
qtree(0.8286) :- goal_unstack , numberofblocks(A) , action_move(B,C) ,
   height(D,E) , E = 2 , height(C,F) , height(B,G) , F < G ,
   height(C,H) , diff(G,H,I) , I = 2, !.
qtree(0.9) :- goal_unstack , numberofblocks(A) , action_move(B,C) ,
   height(D,E) , E = 2 , height(C,F) , height(B,G) , F < G ,
   clear(D), !.
qtree(0.9) :- goal_unstack , numberofblocks(A) , action_move(B,C) ,
   height(D,E) , E = 2 , height(C,F) , height(B,G) , F < G, !.
qtree(0.816429) :- goal_unstack , numberofblocks(A) ,
   action_move(B,C) , height(D,E) , E = 2 , on(C,floor), !.
qtree(0.81) :- goal_unstack , numberofblocks(A) , action_move(B,C) ,
   height(D,E) , E = 2, !.
qtree(0).
```

D.3. P-policy for stack

```
ptree(nonoptimal) :- goal_stack , numberofblocks(A) ,
   action_move(B,C) , height(C,D) , height(E,F) , D < F, !.
ptree(optimal).
```

D.4. Q-policy for stack

```
qtree(0) :- goal_stack , numberofblocks(A) , action_move(B,C) ,
   height(B,D) , D = 4, !.
qtree(1) :- goal_stack , numberofblocks(A) , action_move(B,C) ,
   height(C,D) , D = 3, !.
qtree(0.9) :- goal_stack , numberofblocks(A) , action_move(B,C) ,
   height(C,D) , D = 2, !.
qtree(0.81) :- goal_stack , numberofblocks(A) , action_move(B,C) ,
   height(B,D) , D = 3 , clear(C), !.
qtree(0.81) :- goal_stack , numberofblocks(A) , action_move(B,C) ,
   height(B,D) , D = 3, !.
qtree(0.754716) :- goal_stack , numberofblocks(A) , action_move(B,C) ,
   clear(C) , on(B,floor) , height(C,D) , height(E,F) , D < F, !.
qtree(0.771525) :- goal_stack , numberofblocks(A) , action_move(B,C) ,
   clear(C) , on(B,floor), !.
qtree(0.7965) :- goal_stack , numberofblocks(A) , action_move(B,C) ,
   clear(C), !.
qtree(0.763574) :- goal_stack , numberofblocks(A) , action_move(B,C) ,
   numberofstacks(D) , D = 2, !.
qtree(0.694373).
```

D.5. P-policy for on(A,B)

```
ptree(nonoptimal) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(D,A) , eq(E,B), !.
ptree(nonoptimal) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(D,A) , on(E,B), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   above(D,A) , on(A,floor) , on(D,A) , clear(B), !.
ptree(nonoptimal) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(D,A) , on(A,floor) , on(D,A), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   above(D,A) , on(A,floor) , on(B,E), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   above(D,A) , on(A,floor) , clear(B), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   above(D,A) , on(A,floor) , on(B,A) , clear(E), !.
```

```
ptree(nonoptimal) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(D,A) , on(A,floor) , on(B,A) , on(D,B), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   above(D,A) , on(A,floor) , on(B,A), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   above(D,A) , on(A,floor), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   above(D,A), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   action_move(A,B), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   above(D,B) , on(A,E), !.
ptree(nonoptimal) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(D,B) , on(A,B), !.
ptree(nonoptimal) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(D,B) , eq(E,A), !.
ptree(nonoptimal) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(D,B) , clear(A) , action_move(A,floor) ,
   on(B,E), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   above(D,B) , clear(A) , action_move(A,floor), !.
ptree(optimal) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   above(D,B) , clear(A), !.
ptree(nonoptimal) :- goal_on(A,B) , numberofblocks(C) ,
   action_move(D,E) , above(D,B), !.
ptree(nonoptimal).
```

D.6. Q-policy for on(A,B)

```
qtree(0) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(A,B), !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(A,F) , F = 4 , on(B,E), !.
qtree(0.430467) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E),
   height(A,F) , F = 4, !.
qtree(1) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   action_move(A,B), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(D,F) , height(B,G) , F < G , eq(E,A), !.
qtree(0.531441) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E),
   height(D,F) , height(B,G) , F < G , clear(A) , on(E,floor), !.
qtree(0.430467) :- goal_on(A,B) , numberofblocks(C), action_move(D,E) ,
   height(D,F) , height(B,G) , F < G , clear(A) , clear(B), !.
qtree(0.38742) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
```

```
   height(D,F) , height(B,G) , F < G , clear(A), !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(D,F) , height(B,G) , F < G , on(E,A), !.
qtree(0.6561) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(D,F) , height(B,G) , F < G, !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(E,A) , numberofstacks(F) , F = 2 , on(B,A), !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(E,A) , numberofstacks(F) , F = 2 , on(D,floor), !.
qtree(0.6561) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(E,A) , numberofstacks(F) , F = 2, !.
qtree(0.348678) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E),
   on(E,A) , eq(D,B), !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(E,A), !.
qtree(0.6561) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(E,F) , F = 3, !.
qtree(0.710775) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E),
   on(E,B) , clear(A), !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(E,B), !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   eq(E,B) , on(E,floor) , on(A,floor) , height(D,F) , height(G,H) ,
   F < H, !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   eq(E,B) , on(E,floor) , on(A,floor) , on(D,floor), !.
qtree(0.7695) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   eq(E,B) , on(E,floor) , on(A,floor) , clear(A), !.
qtree(0.78975) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   eq(E,B) , on(E,floor) , on(A,floor), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   eq(E,B) , on(E,floor), !.
qtree(0.430467) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E),
   eq(E,B), !.
qtree(0.77805) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   action_move(A,floor) , on(B,E) , height(D,F) , F = 2 , clear(B), !.
qtree(0.7695) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   action_move(A,floor) , on(B,E) , height(D,F) , F = 2, !.
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   action_move(A,floor) , on(B,E) , clear(B), !.
qtree(0.7695) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   action_move(A,floor) , on(B,E), !.
qtree(0.478297) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E),
   action_move(A,floor), !.
```

```
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(D,A) , clear(B) , on(B,E), !.
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(D,A) , clear(B) , on(B,floor), !.
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(D,A) , clear(B) , clear(E), !.
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(D,A) , clear(B), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   on(D,A), !.
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(A,F) , height(E,G) , F < G, !.
qtree(0.478297) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E),
   height(F,G) , G = 3 , on(F,B) , clear(A), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(F,G) , G = 3 , on(F,B) , clear(F), !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(F,G) , G = 3 , on(F,B), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(F,G) , G = 3 , action_move(F,floor) , on(B,E), !.
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(F,G) , G = 3 , action_move(F,floor) , clear(A), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(F,G) , G = 3 , action_move(F,floor), !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(F,G) , G = 3 , on(A,E), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(F,G) , G = 3 , on(A,floor) , action_move(B,A), !.
qtree(0.7695) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(F,G) , G = 3 , on(A,floor) , clear(B), !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(F,G) , G = 3 , on(A,floor), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   height(F,G) , G = 3, !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   eq(E,A) , clear(B) , on(B,floor) , on(D,floor) , height(E,F) ,
   height(G,H) , F < H, !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   eq(E,A) , clear(B) , on(B,floor) , on(D,floor), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   eq(E,A) , clear(B) , on(B,floor), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   eq(E,A) , clear(B), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
```

```
   eq(E,A) , on(D,B), !.
qtree(0.729) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   eq(E,A), !.
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   clear(A) , height(D,F) , height(G,H) , F < H , clear(B), !.
qtree(0.81) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   clear(A) , height(D,F) , height(G,H) , F < H, !.
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   clear(A) , on(D,floor), !.
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   clear(A) , clear(E), !.
qtree(0.9) :- goal_on(A,B) , numberofblocks(C) , action_move(D,E) ,
   clear(A), !.
qtree(0.81).
```

Acknowledgments

This work was supported in part by the ESPRIT IV Project 20237 ILP2. During the initial phases of this work Luc De Raedt was supported by the Fund for Scientific Research of Flanders. The authors would like to thank Hendrik Blockeel for integrating TILDE and TILDE-RT in RRL, and to Leslie Pack Kaelbling and Pat Langley for interesting discussions and suggestions concerning this work. Finally, we would like to thank the referees for their patience, interest and suggestions, which helped improve the paper significantly.

Note

1. To some extent this is similar to what happens in L.-J. Jin's experience replay technique (Lin, 1992). The idea of experience replay is to memorize all experiences gathered so far and to repeatedly present them to the learning engines. The memorization of past experiences is similar to our work. However, the reasons for memorizing are different. We memorize because TILDE is non-incremental and thus has to start from scratch again each time. Experience replay is aimed at neural networks which will converge more rapidly when processing the evidence more than once.

References

Baum, E. B. (1996). Toward a model of mind as a laissez-faire economy of idiots. In *Proc. 13th Intl. Conf. on Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Blockeel, H. & De Raedt, L. (1997). Lookahead and discretization in ILP. In *Proc. 7th Intl. Workshop on Inductive Logic Programming* (pp. 77–84). Berlin: Springer.

Blockeel, H., De Raedt, L., & Ramon, J. (1998). Top-down induction of clustering trees. In *Proc. 15th Intl. Conf. on Machine Learning* (pp. 55–63). San Francisco, CA: Morgan Kaufmann.

Blockeel, H. & De Raedt, L. (1998). Top-down induction of first order logical decision trees. *Artificial Intelligence*, *101*(1/2), 285–297.

Blockeel, H., De Raedt, L., Jacobs, N., & Demoen, B. (1999). Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, *3*, 59–93.

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees*. Belmont: Wadsworth.

Carbonell, J. & Gill, Y. (1990). Learning by experimentation: The operator refinement method. In Y. Kodratoff & R. Michalski (Eds.), *Machine learning: An artificial intelligence approach* (pp. 191–213). San Mateo, CA: Morgan Kaufmann.

Chapman, D. & Kaelbling, L. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proc. 12th Intl. Joint Conf. on Artificial Intelligence* (pp. 726–731). San Mateo, CA: Morgan Kaufmann.

De Raedt, L. & Blockeel, H. (1997). Using logical decision trees for clustering. In *Proc. 7th Intl. Workshop on Inductive Logic Programming* (pp. 133–141). Berlin: Springer.

Fikes, R. E. & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving. *Artificial Intelligence*, *2*(3/4), 189–208.

Kaelbling, L., Littman, M., & Moore, A. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Karalic, A. & Bratko, I. (1997). First order regression. *Machine Learning*, *26*, 147–176.

Koenig, S. & Simmons, R. G. (1996). The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms. *Machine Learning*, *22*, 227–250.

Kramer, S. (1996). Structural regression trees. In *Proc. 13th Natl. Conf. on Artificial Intelligence* (pp. 812–819). Menlo Park, CA: AAAI Press.

Langley, P. (1985). Strategy acquisition governed by experimentation. In L. Steels & J. A. Campbell (Eds.), *Progress in artificial intelligence* (P. 52). Chichester: Ellis Horwood.

Langley, P. (1996). *Elements of machine learning.* San Matco, CA: Morgan Kaufmann.

Lavrač, N. & Džeroski, S. (1994). *Inductive logic programming: Techniques and applications.* Chichester: Ellis Horwood.

Lin, L.-J. (1992). Self-Improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, *8*, 293–321.

Mitchell, T. (1997). *Machine learning*. New York: McGraw-Hill.

Mitchell, T., Keller, R., & Kedar-Cabelli, S. (1986). Explanation based generalization: A unifying view. *Machine Learning*, *1*(1), 47–80.

Mitchell, T., Utgoff, P. E., & Banerji, R. (1984). Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. Springer-Verlag. Palo Alto, CA: Tioga.

Mooney, R. J. & Califf, M. E. (1995). Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research*, (3), 1–24.

Muggleton, S. & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming 19/20*, 629–679.

Nedellec, C., Rouveirol, C., Adé, H., Bergadano, F., & Tausend, B. (1996). Declarative bias in ILP. In L. De Raedt (Ed.), *Advances in inductive logic programming* (pp. 82–103). Amsterdam: IOS Press.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*, 81–106.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Quinlan, J. R. (1993). *C 4.5: Programs for machine learning*. Morgan Kaufmann.

Stone, P. & Veloso, M. (1999). Team partitioned, opaque transition reinforcement learning. In *Proc. Third Annual Conference on Autonomous Agents* (pp. 206–212). San Matco: Morgan Kaufmann. ACM Press.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211.

Tesauro, G. (1995). Temporal difference learning and TD-GAMMON. *Communications of the ACM*, *38*(3), 58–68.

Utgoff, P. E., Berkman, N. C., & Clause, J. A. (1997). Decision tree induction based on efficient tree restructuring. *Machine Learning*, *29*, 5–44.

Watkins, C. & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

Widmer, G. & Kubat, M. (Eds.) (1998). Special issue on context sensitivity and concept drift. *Machine Learning*, *32*(2), 83–201.

Received April 6, 1999
Revised December 13, 1999 & April 12, 2000
Accepted May 24, 2000
Final manuscript June 20, 2000