Relational Reinforcement Learning

Sašo Džeroski
Department of Intelligent Systems
Jožef Stefan Institute
Jamova 39, SI-1000 Ljubljana, Slovenia
Saso.Dzeroski@ijs.si

Luc De Raedt, Hendrik Blockeel
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{Luc.DeRaedt, Hendrik.Blockeel}@cs.kuleuven.ac.be

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, make abstraction of specific goals pursued and exploit the results of previous learning phases when addressing new (more complex) situations.

1 INTRODUCTION

Within the field of machine learning, both reinforcement learning [8] and inductive logic programming (or relational learning) [12, 10] 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. Furthermore, due to the use of variables in relational representations, it is possible to make abstractions of some specific details of the learning tasks, such as the 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. Using relational reinforcement learning retraining is unnecessary. Relational reinforcement learning also allows us to exploit 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 a preliminary 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 [14, 8, 11] where the Q-function is learned using a relational regression tree algorithm (see [6, 9]). 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.

This paper is organized as follows. In section 2, we view planning (under uncertainty) as a reinforcement learning task, and in section 3, we briefly review reinforcement and in particular Q-learning. Section 4 introduces relational reinforcement learning that combines Q-learning and logical regression trees. In section 5, we present some experiments, and finally, in section 6, we conclude and touch upon related work.

2 LEARNING TO PLAN AS REINFORCEMENT LEARNING

Consider a planning agent with the following task:

Given

- a set of possible states S,
- a set of possible actions A,
- an UNKNOWN function δ: $S \times A \rightarrow A$,
- a function pre:$S \times A \rightarrow \{t, f\}$,
- a goal $goal$:$S \rightarrow \{t, f\}$, and
- a starting state $s \in S$,

find a sequence of actions $a_1, ..., a_n$ ($a_i \in A$) such that

- $goal(\delta(...\delta(s, a_1))...), a_n)) = t$, and
- $pre(\delta(...\delta(s, a_1))...), ...a_i)) = t$.

The agent can be in one of the states of S. It can execute action $a \in A$ in a given state s if the preconditions for a are true in s ($pre(s, a) = t$), e.g., as in STRIPS [7]. Executing an action a in a state s will put the agent in a new state $\delta(s, a)$. When placed in a state s the task of the agent is to find a (shortest) sequence of actions $a_1, ..., a_n$ that will lead it to a goal state. The prototypical AI task belonging to this category is planning.

It is assumed here that the agent does not know the effect of its actions, hence the function δ is unknown to the agent. The above task specification thus contrasts with classical planning in that the δ function is unknown to the agent. Therefore, this task requires a learning component.

Example: The best known (toy)-domain to study planning is 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 below 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.

```
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 goal is to stack a onto b, i.e., `goal(S) :- member(on(a,b),S).` □

3 REINFORCEMENT LEARNING

Planning with incomplete knowledge as outlined above can be recast as a reinforcement learning problem.

3.1 THE BASICS OF REINFORCEMENT LEARNING

The basic notions of reinforcement learning can be outlined as follows (we follow the notation used by Mitchell [11]).

- The task of the agent is to learn a policy $\pi : S \rightarrow A$ for selecting its next action a_t based on the current state s_t; that is $\pi(s_t) = a_t$.
- The reward at time t is $r_t = r(s_t, a_t)$. We will assume here that $r_t = 1$ if $goal(\delta(s_t, a_t)) = t$ 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 $goal(s_t) = f$; otherwise $s_{t+1} = s_t$. This captures the idea that goal states are absorbing states, i.e., once a goal state is reached the only available action is to stay in the state.
- The learned policy should be optimal, i.e., it should maximize

  $$V^{\pi}(s_t) = \sum_{i=o}^{\infty} \gamma^i r_{t+1}$$

  where $0 \leq \gamma < 1$. We will denote the optimal policy by π^*.

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 our planning performance.

3.2 Q-LEARNING

It is well-known that under the conditions sketched in the previous subsection, 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.,

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

The problem with this formulation of π^* is that it requires knowledge of δ and r, which the learner does not have at its disposal.

The Q-function is defined as follows :

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

Knowing Q allows us to rewrite the definition of π^* as follows :

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

According to Mitchell, this rewrite is important as it shows that if the agent can learn the Q function instead of the V^{π^*} function, it will be able to act optimally. The Q-function for a fixed *goal* can then be approximated by $\hat{Q}$, for which a look-up table is learned by the following algorithm (cf. [11]).

for each s, a **do**
  initialize the table entry $\hat{Q}(s, a) = 0$
do forever
  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
  for j=i-1 to 0 **do**
    update $\hat{Q}(s_j, a_j) := r_i + \gamma max_{a'} \hat{Q}(s_{j+1}, a')$

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

$$P(a_i|s) = k^{\hat{Q}(s,a_i)} / \sum_j k^{\hat{Q}(s,a_j)} \quad (1)$$

Higher values of k give stronger preference to actions with high values of $\hat{Q}$ causing the agent to exploit what it has learned, while lower values of k reduce this preference allowing the agent to explore actions that currently do not have high values of $\hat{Q}$.

4 RELATIONAL REINFORCEMENT LEARNING

4.1 THE NEED FOR RELATIONAL REPRESENTATIONS

Given the above classical framework for Q-learning 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 has however a number of disadvantages:

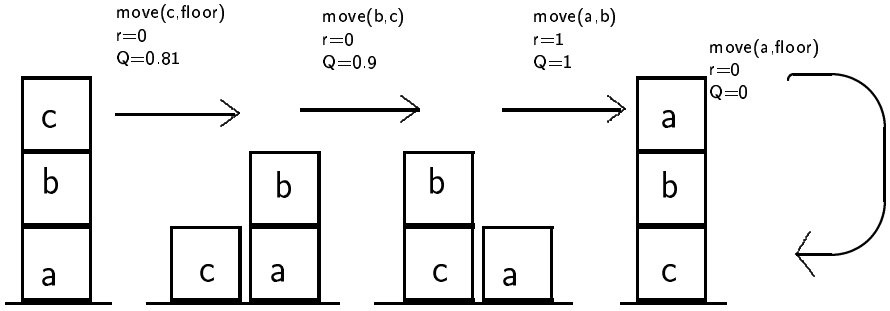

Figure 1: A blocks-world example for relational Q-learning.

- 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 unkown 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) to approximate $\hat{Q}$. The three other problems can only be solved by using a *relational* learning algorithm that can make abstraction of the specific blocks and goals using variables. We now present such a relational learning algorithm.

4.2 THE RRL ALGORITHM

The relational reinforcement learning (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, an implicit representation of the Q-function is learned in the form of a logical regression tree, called a Q-tree.

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. This for-loop now becomes :

for j=i-1 to 0 **do**
  generate example $(s_j, a_j, \hat{q}_j)$,
    where $\hat{q}_j := r_i + \gamma max_{a'} \hat{Q}_e(s_{j+1}, a')$
update $\hat{Q}_e$ using TILDE-RT
  to produce $\hat{Q}_{e+1}$ using the examples $(s_j, a_j, \hat{q}_j)$

TILDE-RT [6] is an algorithm for learning logical regression trees and will be described briefly below.

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.

Example: A possible initial episode ($e = 0$) 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 1. 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 1. It is exactly this input that would be used by TILDE-RT to generate the Q-tree Q_1. □

TILDE-RT is not incremental, so we currently simulate the update of $\hat{Q}$ by keeping all (s, a) pairs encountered and the most recent $\hat{q}$ value for each pair, and inducing a relational regression tree $\hat{Q}_e$ from these examples after each episode e. This tree is then used to select actions in episode $e + 1$.

Table 1: Examples for TILDE-RT generated from the blocks-world Q-learning episode in Figure 1.

qvalue(0.81).	qvalue(0.9).	qvalue(1.0).	qvalue(0.0).
action(move(c,floor)).	action(move(b,c)).	action(move(a,b)).	action(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).	

4.3 TOP-DOWN INDUCTION OF LOGICAL REGRESSION TREES

Logical regression trees are similar to propositional regression trees [3]: leaves predict a value for a continuous class, while internal nodes contain conditions that partition the example space. The difference is that examples here are not feature or attribute-value vectors, but sets of relational facts, representing, e.g., a state of the blocks world, a goal, and an action to be taken, all at the same time. Similarly, internal nodes are not restricted to attribute-value tests but can be first order literals containing predicates, variables and complex terms.

The TILDE-RT system [6] induces such first order logical regression trees (or relational regression trees) from examples (cf. [9] for a related approach). The input for TILDE-RT is a set of state-action pairs together with the corresponding Q-values, represented as sets of facts. From this TILDE-RT induces (using the classical TDIDT-algorithm) a tree in which the classes correspond to real numbers (Q-values).

To illustrate the above notions, consider the episode shown in Figure 1. The examples for TILDE-RT generated by the RRL algorithm are given in Table 1. The relational regression tree induced by TILDE-RT from these examples is shown in Figure 2.

Nodes in the tree correspond to Prolog-queries. If the query succeeds in an example the `yes` subtree is taken, otherwise the `no` subtree. Different nodes in the tree may share variables, e.g., the bottom node in the tree (containing `action(move(D,B))`) refers to the variable `D` that first appear in the root of the tree (`goal(on(C,D))`). The Prolog program corresponding to the tree is shown in the lower part of Figure 2.

The semantics of logical decision trees is extensively discussed in [1], as well as the correspondence between a tree and a Prolog program. The method to induce the trees is described in [6] and is - for the case of regression trees - very similar to Kramer's SRT system [9]. We refer to these papers for more details on the representation and learning of such 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(C,D)` and `on(C,D)` holds in the state (goal states are absorbing). On the other hand, if the goal `on(C,D)` does not yet hold and the action is `move(C,D))`, then a Q-value of one is returned (reward of one for achieving a goal state).

```
action(move(A,B)) , goal(on(C,D))
on(C,D) ?
+--yes: [0]
+--no:  action(move(C,D)) ?
        +--yes: [1]
        +--no:  action(move(D,B)) ?
                +--yes: [0.9]
                +--no:  [0.81]
```

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

Figure 2: A relational regression tree generated by TILDE-RT from the examples in Table 1 and its equivalent Prolog program.

```
action(move(A,B)) , goal(on(C,D))
on(C,D) ?
+--yes: [0]
+--no:  action(move(C,D)) ?
        +--yes: [1]
        +--no:  on(B,C) ?
                +--yes: [0.729]
                +--no:  on(B,D) ?
                        +--yes: [0.729]
                        +--no:  action(move(A,C)) ?
                                +--yes: [0.81]
                                +--no:  action(move(A,D)) ?
                                        +--yes: [0.81]
                                        +--no:  clear(D) ?
                                                +--yes: on(C,B) ?
                                                |       +--yes: on(A,C) ?
                                                |       |       +--yes: [0.9]
                                                |       |       +--no:  clear(C) ?
                                                |       |               +--yes: [0.9]
                                                |       |               +--no:  [0.81]
                                                |       +--no:  [0.9]
                                                +--no:  clear(C) ?
                                                        +--yes: on(C,B) ?
                                                        |       +--yes: [0.9]
                                                        |       +--no:  [0.81]
                                                        +--no:  [0.81]
```

Figure 3: The Q-tree generated by RRL in the 3 blocks world after 10 episodes.

5 EXPERIMENTS

We applied the RRL algorithm described above to learn how to stack one block onto another in worlds with three and four blocks, respectively. In particular, the goal to achieve was $on(a, b)$, the two other blocks being c and d. An example episode in the three blocks world is depicted in Figure 1.

The discount factor γ had the value 0.9. When selecting states stochastically according to equation 1, the constant k was set to $e^{0.2}$. Examples for learning Q-trees were generated after each episode, as described in the section above.

TILDE-RT was used to induce an updated Q-tree after each episode. The minimal number of cases in a leaf was set to one and TILDE-RT generated unpruned trees, which exactly reproduce the Q-values for the state-action pairs seen during the learning phase.

Using the above settings, the RRL algorithm was first run for 10 episodes in the 3 blocks world. The tree shown in Figure 3 was generated by TILDE-RT after the final episode. This tree represents the optimal policy for the given reinforcement learning problem. The top two levels of the tree match those of the tree in Table 1, which was generated from a single episode.

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 relation representation for Q-learning.

The Q-tree obtained after 10 episodes in the 4 blocks worlds was much larger (44 nodes as opposed to the 12 nodes of the 3-blocks Q-tree). It also represents an optimal policy: it chooses a shortest path to a goal state from all initial states, if the action with the highest Q-value is always selected.

The 3 top levels of the tree match with the tree from the 3 blocks world. This indicates that the result of learning in the 3 blocks world could be used to bootstrap learning in the 4 blocks world. Indeed, if we take the Q-tree learned in the 3 blocks world shown in Figure 3 and use it to select actions in the 4 blocks world, it selects an optimal path to a goal state from all but 9 of the 73 possible initial states. In 4 of the 9 cases a looping behavior is produced, in the remaining 5 cases one extra action is needed as compared to an optimal plan.

Using the Q-tree from Figure 3 to bootstrap RRL in the 4 blocks world helps improve performance, especially in the initial episodes. Without bootstrapping, after two episodes a tree is learned which produces nonoptimal behavior in 12 of the 73 initial states. With bootstrapping, the behavior of the learned tree is nonoptimal for 8 of the 73 possible initial states. After ten episodes, the learned Q-tree produces optimal behavior and is much smaller (27 nodes) as compared to the Q-tree learned without bootstrapping (44 nodes).

6 DISCUSSION

We have presented an approach to planning with incomplete knowledge that combines reinforcement learning and relational regression 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 ability to use results of simpler tasks to bootstrap learning in more complex tasks is also an advantage worth mentioning. Finally, it is easy to incorporate nondeterministic actions within the proposed approach.

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)* (the recursive closure of *on(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 preliminary and 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 work in the line of proper performance assessment, both in terms of standard performance tests in reinforcement learning fashion (root mean square errors of learned Q-values wrt. the Q-values of the optimal policy) and in considering more complex and demanding planning problems.

More complex problems can be obtained by increasing the number of blocks in the world, considering more complex goals, such as building a stack of all available blocks, and considering problems outside the blocks world.

This work is related to work on generalization in reinforcement learning, which has however mainly addressed the use of neural networks for this purpose [13]. The closest related work is probably Chapman's and Kaelbling's decision tree algorithm that was specifically designed for reinforcement learning [5]. Note however that our approach is distinguished from the mainstream work in reinforcement learning by the use of a relational representation.

Relational representations are commonly used in planning approaches. There have also been some attempts to combine planning with relational learning within those approaches, e.g., within the PRODIGY approach [2]. Our approach is related to them through the use of a relational representation. However, it seems that the combination of planning, reinforcement learning and relational learning has not been addressed so far.

The reinforcement learning part of the work presented in this paper is admittedly simple. We have taken a standard textbook description of reinforcement learning [11] 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 non-deterministic actions. To handle nondeterministic actions an appropriate update rule (see page 382 of [11]) 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 standards - not optimal. One of the reasons is that it is not incremental. However, incrementality is not enough, as the (estimated) values of Q are changing with time. These problems are taken care of within the Chapman and Kaelbling's decision tree algorithm that was specifically designed for reinforcement learning [5]. 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, which is currently under development, would not suffer from the abovementioned problems.

Acknowledgements

This work was supported in part by the ESPRIT IV Project 20237 ILP2. Luc De Raedt is supported by the Fund for Scientific Research of Flanders. Hendrik Blockeel is supported by the Flemish Institute for the Promotion of Scientific and Technological Research in Industry (IWT).

References

[1] Blockeel, H., and De Raedt, L. (1997) Experiments with Top-down Induction of Logical Decision Trees. *Artificial Intelligence*. Forthcoming.

[2] Borrajo, D., and Veloso, M. (1997) Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review*, 11(1-5): 371–405.

[3] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984) *Classification and Regression Trees*. Wadsworth, Belmont.

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

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

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

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

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

[9] Kramer, S. (1996) Structural regression trees. In *Proc. 13th Natl. Conf. on Artificial Intelligence*. AAAI Press, Menlo Park, CA.

[10] Lavrač, N. and Džeroski, S. (1994) *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, Chichester.

[11] Mitchell, T. (1997) *Machine Learning*. McGraw-Hill, New York.

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

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

[14] Watkins, C., and Dayan, P. (1992) Q-learning. *Machine Learning*, 8: 279–292.