# Relational Reinforcement Learning for Agents in Worlds with Objects

Sašo Džeroski

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

**Abstract.** In reinforcement learning, an agent tries to learn a policy, i.e., how to select an action in a given state of the environment, so that it maximizes the total amount of reward it receives when interacting with the environment. We argue that a relational representation of states is natural and useful when the environment is complex and involves many inter-related objects. Relational reinforcement learning works on such relational representations and can be used to approach problems that are currently out of reach for classical reinforcement learning approaches. This chapter introduces relational reinforcement learning and gives an overview of techniques, applications and recent developments in this area.

## 1 Introduction

In reinforcement learning (for an excellent introduction see the book by Sutton and Barto [13]), an agent tries to learn a policy, i.e., how to select an action in a given state of the environment, so that it maximizes the total amount of reward it receives when interacting with the environment. In cases where the environment is complex and involves many inter-related objects, a relational representation of states is natural. This typically yields a very high number of possible states and state/action pairs, which makes most of the existing tabular reinforcement learning algorithms inapplicable. Even the existing reinforcement learning approaches that are based on generalization, such as that of Bertsekas and Tsitsiklis [1], typically use a propositional representation and cannot deal directly with relationally represented states.

We introduce relational reinforcement learning, which uses relational learning algorithms as generalization engines within reinforcement learning. We start with an overview of reinforcement learning ideas relevant to relational reinforcement learning. We then introduce several complex worlds with objects, for which a relational representation of states is natural. An overview of different relational reinforcement learning algorithms developed over the last five years is presented next and illustrated on an example from the blocks world. Finally, some experimental results are presented before concluding with a brief discussion.

## 2 Reinforcement Learning

This section gives an overview of reinforcement learning ideas relevant to relational reinforcement learning. For an extensive treatise on reinforcement learning, we refer the reader to Sutton and Barto [13]. We first state the task of reinforcement learning, then briefly describe the Q-learning approach to reinforcement learning. In its basic variant, Q-learning is tabular: this is unsuitable for problems with large state spaces, where generalization is needed. We next discuss generalization in reinforcement learning and in particular generalization based on decision trees. Finally, we discuss the possibility of integrating learning by exploration (as is typically the case in reinforcement learning) and learning with guidance (by a human operator or some other reasonable policy, i.e., a policy that yields sufficiently dense rewards).

### 2.1 Task definition

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 $\pi$. 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 $r_t = r(s_t, a_t)$. The function $V^{\pi}(s)$ denotes the value (expected return; discounted cumulative reward) of state $s$ under policy $\pi$.

The agent does not necessarily know what effect its actions will have, i.e., what state it will end up in after executing an action. This means that the function $\delta$ is unknown to the agent. In fact, it may even be stochastic: executing the same action in the same state on different occasions may yield different successor states. We also assume 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. We will assume episodic learning, where a sequence of actions ends in a terminal state.

## 2.2 Tabular Q-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. The optimal policy $\pi^*$ 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 Q-function for policy $\pi$ 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 $\pi^*$ as follows

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

An approximation to the $Q^*$-function, $Q$, in the form of a look-up table, is learned by the following algorithm.

**Table 1.** The $Q$-learning algorithm.

```
Initialize $Q(s,a)$ arbitrarily
**repeat** (for each episode)
    Initialize $s_0$; $t \leftarrow 0$
    **repeat** (for each step of episode)
        Choose $a_t$ for $s_t$ using the policy derived from $Q$
        Take action $a_t$, observe $r_t$, $s_{t+1}$
        $Q(s_t,a_t) \leftarrow r_t + \gamma max_a Q(s_{t+1},a)$
        $t \leftarrow t+1$
    **until** $s$ is terminal
**until** no more episodes
```

The agent learns through continuous interaction with the environment, during which it exploits what it has learned so far, but also explores. In practice, this means that the current approximation $Q$ is used to select an action most of the time. However, in a small fraction of cases an action is selected randomly from the available choices, so that unseen state/action pairs can be explored.

For smoother learning, an update of the form

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_t + \gamma max_a Q(s_{t+1},a) - Q(s_t,a_t)]$$

would be used. This is a special case of temporal-difference learning, where algorithms such as SARSA [12] also belong. In SARSA, instead of considering all possible actions $a$ in state $s_{t+1}$ and taking the maximum $Q(s_{t+1},a)$, only the action $a_{t+1}$ actually taken in state $s'$ during the current episode is considered. The update rule is thus $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$. For the algorithm in Table 1, the learned action-value function $Q$ directly approximates $Q^*$, regardless of the policy being followed.

## 2.3 Generalization / G-trees

Using a tabular representation for the learned approximation to the Q-function or V-functions is only feasible for tasks with small numbers of states and actions. This is due to both issues of space (large table) and time (needed to fill the table accurately). The way out is to generalize over states and actions, so that approximations can be produced also for states (and possibly actions) that the agent has never seen before.

Most approaches to generalization in reinforcement learning use neural networks for function approximation [1]. States are represented by feature vectors. Updates to state-values or state-action values are treated as training examples for supervised learning. Nearest-neighbor methods have also been used, especially in the context of continuous states and actions [11].

**Table 2.** The G-algorithm.

```
Create an empty leaf
while data available do
    Sort data down to leaves and update statistics in leaves
    if a split is needed in a leaf then grow two empty leaves
```

The G-algorithm [3] is a decision tree learning algorithm designed for generalization in reinforcement learning. An extension of this algorithm has been used in relational reinforcement learning: we thus briefly summarize its main features here. The G-algorithm updates its theory incrementally as examples are added. An important feature is that examples can be discarded after they are processed. This avoids using a huge amount of memory to store examples. At a high level, the G-algorithm (Table 2) stores the current decision tree: for each leaf node statistics are kept for all tests that could be used to split that leaf further. Each time an example is inserted, it is sorted down the decision tree according to the tests in the internal nodes; in the leaves, the statistics of the tests are updated.

## 2.4 Exploration and guidance

Besides the problems with tabular Q-learning, large state/action spaces entail another type of problem for reinforcement learning. Namely, in a large state/action space, rewards may be so sparse that with random exploration (as is typical at the start of a reinforcement learning run) they will only be discovered extremely slowly. This problem has only recently been addressed for the case of continuous state/action spaces.

Smart and Kaelbling [11] integrate exploration in the style of reinforcement learning with human-provided guidance. Traces of human-operator performance are provided to a robot learning to navigate as a supplement to its reinforcement learning capabilities. Using nearest-neighbor methods together with precautions to avoid overgeneralization, they show that using the extra guidance helps improve the performance of reinforcement learning.

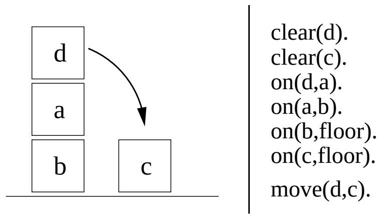

**Fig. 1.** Example state and action in the blocks-world.

## 3 Some Worlds with Objects

In this section, we introduce three domains where using a relational representation of states is natural. Each of the domains involves objects and relations between them. The number of possible states in all three domains is very large. The three domains are: the blocks world, the Digger computer game, and the Tetris computer game.

### 3.1 The blocks world

In the blocks world, 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., $s = \{clear(c), clear(d), on(d,a), on(a,b), on(b, floor), on(c, floor)\}$ represents the state in Figure 1. The available actions are then $move(X,Y)$ where $X \neq Y$, $X$ is a block and $Y$ is a block or the floor. The number of states in the blocks world grows rapidly with the number of blocks. With 10 blocks, there are close to 59 million possible states.

We study three different goals in the blocks world: stacking all blocks, unstacking all blocks (i.e., putting all blocks on the floor) and putting a specific block on top of another specific block. In a blocks world with 10 blocks, there are 3.5 million states which satisfy the stacking goal, 1.5 million states that satisfy a specific $on(A,B)$ goal (where $A$ and $B$ are bound to specific blocks) and one state only that satisfies the unstacking goal. A reward of 1 is given in case a goal-state is reached in the optimal number of steps; the episode ends with a reward of 0 if it is not.

### 3.2 The Digger game

Digger[1] is a computer game created in 1983, by Windmill Software. It is one of the few old computer games which still hold a fair amount of popularity. In this game, the player controls a digging machine or "Digger" in an environment that contains emeralds, bags of gold, two kinds of monsters (nobbins and hobbins) and tunnels. The object of the game is to collect as many emeralds and as much gold as possible while avoiding or shooting monsters.

[1] http://www.digger.org

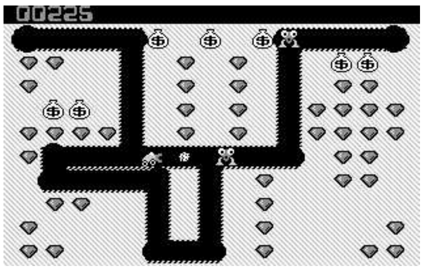

**Fig. 2.** A snapshot of the DIGGER game.

In our tests we removed the hobbins and the bags of gold from the game. Hobbins are more dangerous than nobbins for human players, because they can dig their own tunnels and reach Digger faster, as well as increase the mobility of the nobbins. However, they are less interesting for learning purposes, because they reduce the implicit penalty for digging new tunnels (and thereby increasing the mobility of the monsters) when trying to reach certain rewards. The bags of gold we removed to reduce the complexity of the game.

A state representation consists of the following components:

– the coordinates of digger, e.g., *digPos(6,9)*
– information on digger of the form `digInf(digger_dead, time_to_reload, level_done, pts_scored,steps_taken)`, e.g., *digInf(false,63,false,0,17)*,
– information on tunnels as seen by digger (range of view in each direction, e.g., *tunnel(4,0,2,0)*; information on the tunnel is relative to the digger; there is only one digger, so there is no need for a digger index argument)
– list of emeralds (e.g., [*em(14,9), em(14,8), em(14,5),* ...]),
– list of monsters (e.g., [*mon(10,1,down), mon(10,9,down)* ...]), and
– information on the fireball fired by digger (coordinates, travelling direction, e.g., *fb(7,9,right)*).

The actions are of the form *moveOne*$(X)$ and *shoot*$(Y)$, where $X$ and $Y$ are in [*up,down,left,right*].

### 3.3 The Tetris game

Tetris[2] is a widespread puzzle-video game played on a two-dimensional grid. Differently shaped blocks fall from the top of the game field and fill up the grid. The object of the game is to score points while keeping the blocks from piling up to the top of the game field. To do this, one can move the dropping blocks right and left or rotate them as they fall. When one horizontal row is completely filled, that line disappears and the player scores points. When the blocks pile up to the top of the game field, the game ends.

[2] Tetris was invented by Alexey Pazhitnov and is owned by The Tetris Company and Blue Planet Software.

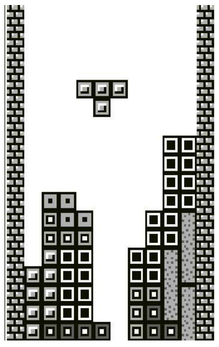

**Fig. 3.** A snapshot of the TETRIS game.

In the tests presented, we only looked at the strategic part of the game, i.e., given the shape of the dropping and the next block, one has to decide on the optimal orientation and location of the block in the game-field. (Using low level actions — turn, move left or move right — to reach such a subgoal is rather trivial and can easily be learned by (relational) reinforcement learning.) We represent the full state of the Tetris Game, the type of the next dropping block included.

## 4 Relational Reinforcement Learning

Relational reinforcement learning (RRL) addresses much the same task as reinforcement learning in general. What is typical of RRL is the use of a relational (first-order) representation to represent states, actions and (learned) policies. Relational learning methods, originating from the field of inductive logic programming [10], are used as generalization engines.

### 4.1 Task definition

While the task definition for reinforcement learning (as specified earlier in this chapter) applies to RRL, a few details are worth noting. States and actions are represented relationally. Background knowledge and declarative bias need to be specified for the relational generalization engines.

All possible states would not be listed explicitly as input to the RRL algorithm (as they might be for ordinary reinforcement learning). A relational language for specifying states would rather be defined (in the blocks world, this language would comprise the predicates $on(A, B)$ and $clear(C)$). Actions would also be specified in a relational language ($move(A, B)$ in the blocks world) and not all actions would be applicable in all states; in fact, the number of possible actions may vary considerably across different states.

Background knowledge generally valid about the domain (states in $S$) can be specified in RRL. 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 $B$.

Declarative bias for learning relational representations of policies can also be given. In the blocks world, e.g., we do not allow policies to refer to the exact identity of blocks ($A = a$, $B = b$, etc.). The background knowledge and declarative bias taken together specify the language in which policies are represented.

## 4.2 The RRL algorithm

The RRL algorithm (Table 2) is obtained by combining the classical Q-learning algorithm (Table 1) and a relational regression tree algorithm (TILDE [2]). Instead of 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. After a Q-tree is learned, a classification tree is learned that classifies actions as optimal or non-optimal. This tree, called a P-tree, is usually much more succinct than the Q-tree, since it does not need to distinguish among different levels of non-optimality.

**Table 3.** The RRL algorithm for relational reinforcement learning.

```
Initialize $\hat{Q}_0$ to assign 0 to all $(s, a)$ pairs
Initialize $\hat{P}_0$ to assign 1 to all $(s, a)$ pairs
Initialize Examples to the empty set.
e := 0
**while** true **do**
   generate an episode that consists of states $s_0$ to $s_i$
     and actions $a_0$ to $a_{i-1}$ through the use of
     a standard Q-learning algorithm,
     using the current hypothesis for $\hat{Q}_e$
   **for** j=i-1 to 0 **do** [generate examples for learning Q-tree]
      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 $\hat{Q}_e$ by applying TILDE to Examples,
     i.e., $\hat{Q}_{e+1}$ = TILDE(Examples)
   **for** j=i-1 to 0 **do** [generate examples for learning P-tree]
      **for** all actions $a_k$ possible in state $s_j$ **do**
         **if** state action pair $(s_j, a_k)$ is optimal
            according to $\hat{Q}_{e+1}$
         **then** generate example $(s_j, a_k, c)$ where $c = 1$
         **else** generate example $(s_j, a_k, c)$ where $c = 0$
   update $\hat{P}_e$: apply TILDE to the examples $(s_j, a_k, c)$
     to produce $\hat{P_{e+1}}$
   e := e + 1
```

The RRL algorithm is given in Table 3. In its initial implementation [7], RRL keeps a table of state/action pairs with their current Q-values. This table is used to create a generalization in the form of a relational regression tree (Q-tree) by applying TILDE. The Q-tree is then the policy used to select actions to take by the agent. The reason the table is kept is the nonincrementality of TILDE.

In complex worlds, where states can have a variable number of objects, an exact Q-tree representation of the optimal policy can be very large and also depend on the number of objects in the state. For example, in the blocks world, a state can have a varying number of blocks: the number of possible values for the Q-function (and the complexity of the Q-tree) would depend on this number. Choosing the optimal action, however, can sometimes be very simple: in the unstacking task, we simply have to pick up a block that is on top of another block and put it on the floor. This was our motivation for learning a P-tree by generating examples from the Q-tree.

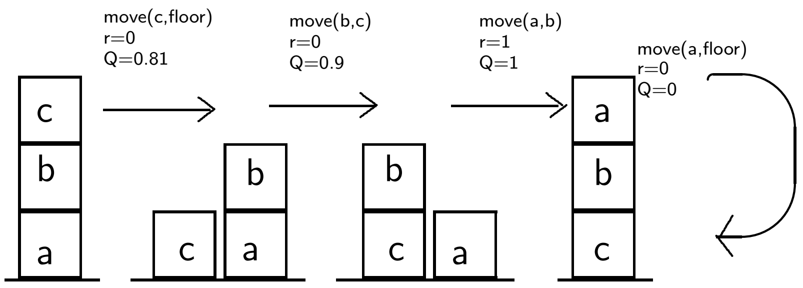

**Fig. 4.** A blocks-world episode for relational Q-learning.

**Table 4.** Examples for TILDE generated from the blocks-world Q-learning episode in Figure 4.

| 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). |  |

## 4.3 An example

To illustrate how the RRL algorithm works, we use an example from the blocks world. The task here is to stack block $a$ on block $b$, i.e., to achieve $on(a, b)$. An example episode is shown in Figure 4. As for the tabular version of Q-learning, updates of the Q-function are generated for all state/action pairs encountered during the episode. These are also listed in the figure.

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

**Fig. 5.** A relational regression tree (Q-tree) generated by TILDE from the examples in Table 4.

The examples generated for TILDE from this episode are given in Table 4. A reward is only obtained when moving $a$ onto $b$ ($r = 1, Q = 1$): the Q-value is propagated backwards to also reward the actions preceding and leading to $move(a, b)$. Note that a Q-value of zero is assigned to any state/action pair where the state is terminal (the last state in the episode), as no further reward can be expected.

From the examples in Table 4, the Q-tree in Figure 5 is learned. The root of the tree (*goal_on(A,B), action_move(D,E)*) introduces the state-action pair evaluated, while the rest of the tree performs the evaluation, i.e., calculates the Q-value. The tree correctly predicts zero Q-value if the goal is already achieved and a Q-value of one for any action, given that block $A$ is clear. This is obviously overly optimistic, but does capture the fact that $A$ needs to be clear in order to stack it onto $B$. Note that the goal $on(A, B)$ explicitly appears in the Q-tree.

If we use the Q-trees to generate examples for learning the optimality of actions, we obtain the P-tree in Figure 6. Note that the P-tree represents a policy much closer to the optimal one. If we want to achieve $on(A, B)$, it is optimal to move a block that is above $A$. Also, the action $move(A, B)$ is optimal whenever it is possible to take it.

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

**Fig. 6.** A P-tree for the three blocks world generated from the episode in Figure 4.

### 4.4 Incremental RRL/TG trees

The RRL algorithm as described in the previous section has a number of problems. It needs to keep track of an ever increasing number of examples, needs to replace old Q-values with new ones if a state-action pair is encountered again, and builds trees from scratch after each episode. The G-tree algorithm (also mentioned earlier) does not have these problems, but only works for propositional representations. Driessens et al. [6] upgrade G-tree to work for relational representations yielding the TG-tree algorithm. At the top level, the TG-tree algorithm is the same as the G-tree algorithm. It differs in the fact that TG can use relational tests to split on; these are the same type of tests that TILDE can use. Using TG instead of TILDE within RRL yields the RRL-TG algorithm.

**Table 5.** The G-RRL algorithm: This is the RRL-TG algorithm with integrated guidance (k example traces).

```
Initialise $\hat{Q}_0$ to assign 0 to all $(state, action)$ pairs
for $i = 0$ to $k$ do
    transform $trace_i$ into $(state, action, qvalue)$ triplets
    process generated triplets with TG algorithm
        transforming $\hat{Q}_i$ into $\hat{Q}_{i+1}$
run normal RRL-TG starting with $\hat{Q}_k$ as the
    initial Q-function hypothesis
```

### 4.5 Integrating experimentation and guidance in RRL

Since RRL typically deals with huge state spaces, sparse rewards are indeed a serious problem. To alleviate this problem, Driessens and Džeroski [5] follow the example of Smart and Kaelbling [11] and integrate experimentation and guidance in RRL. In G-RRL (guided RRL), traces of human behavior or traces generated by following some reasonable policy (that generates sufficiently dense rewards) are provided at the beginning and are followed by ordinary RRL. Note that a reasonable policy could also be a previously learned policy that we want to improve upon. The G-RRL algorithm is given in Table 5.

## 5 Experiments

Here we summarize the results of experiments with RRL. RRL was extensively evaluated experimentally on the blocks world by Džeroski et al. [8]. We first summarize these results. We then proceed with an overview of the most recent experiments with RRL, which involve the use of guidance in addition to pure reinforcement learning [5], i.e., the use of the G-RRL algorithm. These experiments involve the three domains described earlier in this chapter: the blocks world, the Digger game and the Tetris game.

### 5.1 Blocks world experiments with RRL

We have conducted experiments [8] in the blocks world with 3, 4, and 5 blocks, considering the tasks of stacking, unstacking and $on(a, b)$ mentioned earlier. They consider both settings with a fixed number of blocks (either 3, 4 or 5) and a varying number of blocks (first learn with 3 blocks, use this to bootstrap learning with 4 blocks, and similarly learn with 5 blocks afterwards). In addition to the state and action information, the RRL algorithm was supplied with the number of blocks, the number of stacks and the following background predicates: *equal/2, above/2, height/2* and *difference/3* (an ordinary subtraction of two numerical values).

The experiments show that RRL is effective for different goals: it 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 $B$. Both can learn optimal policies for state spaces with a fixed number of blocks (both with Q-trees and P-trees), but this becomes more difficult when the number of blocks increases. An explanation for this is that the sparse rewards problem becomes more and more severe as the number of possible states skyrockets with increasing the number of blocks.

Even when learning from experience with a fixed number of blocks, RRL can learn policies that are optimal 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 are often optimal for state spaces with a varying number of blocks as well. RRL can also learn from experience in which the number of blocks is varied. Starting with a small number of blocks and gradually increasing it allows for a bootstrapping process, where optimal policies are learned faster.

If the Q-tree learned does not work, then the P-tree will not work either. But once a Q-tree is learned that does the job right (even for states with a fixed number of blocks), one is better off using the P-tree learned from it. The latter usually generalizes nicely to larger numbers of blocks than seen during training.

### 5.2 Experiments with G-RRL

The experiments with G-RRL involve the three domains described earlier: the 10-blocks world, the Digger game and the Tetris game. Only Q-trees were built.

**The blocks world** In the blocks world, the three tasks mentioned earlier (stacking, unstacking and $on(a, b)$) were addressed. Traces of the respective optimal policies were provided at the beginning of learning, followed by an application of the RRL-TG algorithm.

In summary, a moderate number of optimal traces helps the learning process converge faster and/or to a higher level of performance (average reward). The learning curves for the stacking and $on(a, b)$ problems are given in Figures 7 and 8. G-RRL is supplied with 5, 20, and 100 optimal traces. Providing

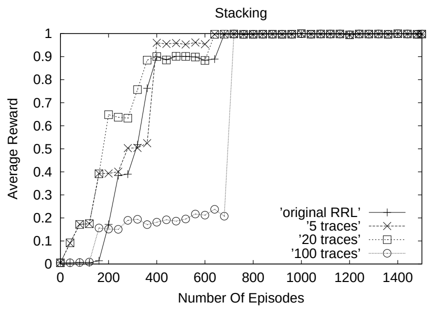

**Fig. 7.** The learning curves of RRL and G-RRL for the stacking task.

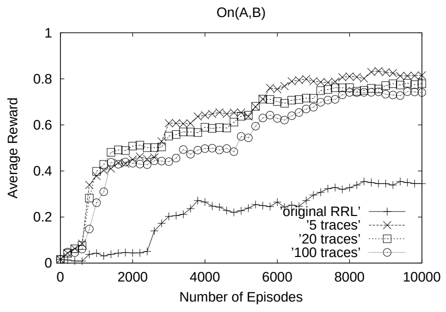

**Fig. 8.** The learning curves of RRL and G-RRL for the on(a,b) task.

guidance clearly helps in the $on(a, b)$ case, but less improvement is achieved when more traces are provided.

For stacking, better performance is achieved when providing 5 or 20 traces. Providing 100 traces, however, actually causes worse performance as compared to the original RRL algorithm. The experiment takes longer to converge and during the presentation of the 100 traces to G-RRL no learning takes place. The problem is that we supply the system with optimal actions only and it overgeneralizes, failing to distinguish between optimal and nonoptimal actions.

**The Digger game** In the Digger Game, in addition to the state and action representation mentioned earlier, predicates such as *emerald/2, nearestEmerald/2, monster/2, visibleMonster/2, distanceTo/2, getDirection/2, lineOfFire/1*, etc., were provided as background knowledge for the construction of the Q-tree. Since it is hard to write an optimal policy, we used a policy generated in earlier work [4] by RRL (which already performed quite well).

Figure 9 shows the average reward obtained by the learned strategies over 640 digger test-games divided over the 8 different Digger levels. It shows that G-RRL is indeed able to improve on the policy learned by RRL. Although the speed of convergence is not improved, G-RRL reaches a significantly higher level of overall performance.

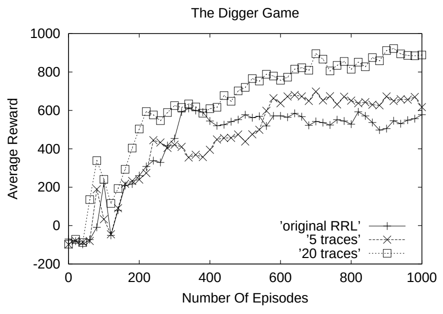

**Fig. 9.** Learning curves for RRL and G-RRL for the Digger game.

**The Tetris game** For the Tetris game, RRL could use the following predicates (among others): *blockwidth/2, blockheight/2, rowSmaller/2, topBlock/2, holeDepth/2, holeCovered/1, fits/2, increasesHeight/2, fillsRow/2* and *fillsDouble/2*. Like with the Digger Game, it is very hard (if not impossible) to generate an optimal or even "reasonable" strategy for the Tetris game. This time, we opted to supply G-RRL with traces of non-optimal playing behavior from a human player.

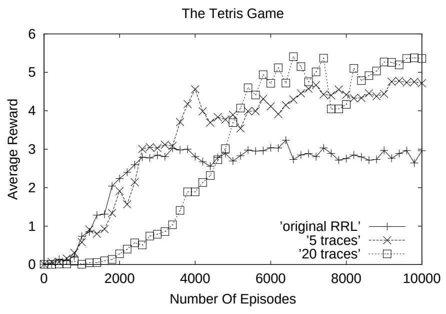

**Fig. 10.** Learning curves for RRL and G-RRL for the Tetris game.

The results for learning Tetris with RRL and G-RRL are below our expectations. We believe that this is due to the fact that the future reward in Tetris is very hard to predict, especially by a regression technique that needs to discretize these rewards like the TG algorithm. However, even with these disappointing results, the added guidance in the beginning of the learning experiment still has its effects on the overall performance. Figure 10 shows the learning curves for RRL and G-RRL supplied with 5 or 20 manually generated traces. The data points are the average number of deleted lines per game, calculated over 500 played test games.

## 6 Discussion

Relational reinforcement learning (RRL) is a powerful learning approach that allows us to address problems that have been out of reach of other reinforcement learning approaches. The relational representation of states, actions, and policies allows for the representation of objects and relations among them. Background knowledge that is generally valid in the domain at hand can also be provided to the generalization engine(s) used within RRL and adds further power to the approach.

We expect RRL to be helpful to agents that are situated in complex environments which include many objects (and possibly other agents) and where the relations among objects, between the agent and objects, and among agents are of interest. The power of the representation formalism used would allow for different levels of awareness of other agents, i.e., social awareness [9]. Knowledge about the existence and behavior of other agents can be either provided as background knowledge or learned.

There are many open issues and much work remains to be done on RRL. One of the sorest points at the moment is the generalization engine: it turns out that G-trees and TG-trees try to represent all policies followed by the agent during its lifetime and can thus be both large and ineffective. Developing better incremental and relational generalization engines is thus a priority. Finding better ways to integrate exploration and guidance also holds much promise for RRL. Finally, we are seeking to apply RRL to difficult, interesting and practically relevant problems.

### Bibliographic notes

This chapter summarizes research on relational reinforcement learning that has previously been published elsewhere. Relational reinforcement learning (RRL) was introduced by Džeroski, De Raedt and Blockeel [7] and further extended and experimentally evaluated on the blocks world by Džeroski, De Raedt, and Driessens [8]. Driessens, Ramon, and Blockeel [6] replaced the non-incremental generalization engine in RRL the with TG-tree algorithm, a relational version of the G-algorithm, yielding the RRL-TG algorithm.

Driessens and Blockeel [4] applied RRL to the Digger game problem. Driessens and Džeroski [5] extended RRL-TG to take into account guidance from existing reasonable policies, either human generated or learned. They applied G-RRL to the Digger and Tetris games.

### Acknowledgements

The author would like to thank Hendrik Blockeel, Kurt Driessens and Luc De Raedt for the exciting and productive cooperation on the topic of relational reinforcement learning. Special thanks to Kurt Driessens for some of the figures and results included in this chapter.

## References

1. Bertsekas, D.P., & Tsitsiklis, J.N. (1996). *Neuro-Dynamic Programming.* Belmont, MA: Athena Scientific.
2. Blockeel, H., De Raedt, L., & Ramon, J. (1998). Top-down induction of clustering trees. In *Proc. 15th International Conference on Machine Learning*, pages 55–63. San Francisco: Morgan Kaufmann.
3. Chapman, D., & Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proc. 12th International Joint Conference on Artificial Intelligence*, pages 726–731. San Mateo, CA: Morgan Kaufmann.
4. Driessens, K., & Blockeel, H. (2001). Learning Digger using hierarchical reinforcement learning for concurrent goals. In *Proc. 5th European Workshop on Reinforcement Learning*, pages 11-12. Utrecht, The Netherlands: CKI Utrecht University.
5. Driessens, K., & Džeroski, S. (2002) Integrating experimentation and guidance in relational reinforcement learning. *In Proc. 19th International Conference on Machine Learning*, pages 115–122. San Francisco, CA: Morgan Kaufmann.
6. Driessens, K., Ramon, J., & Blockeel, H. (2001). Speeding up relational reinforcement learning through the use of an incremental first order decision tree algorithm. In *Proc. 12th European Conference on Machine Learning*, pages 97–108. Berlin: Springer.
7. Džeroski, S., De Raedt, L., & Blockeel, H. (1998). Relational reinforcement learning. In *Proc. 15th International Conference on Machine Learning*, pages 136–143. San Francisco, CA: Morgan Kaufmann.
8. Džeroski, S., De Raedt, L., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, *43*, 7–52.
9. Kazakov, D., & Kudenko, D. (2001). Machine learning and inductive logic programming for multi-agent systems. In Luck, M., Marik, V., Stepankova, O., and Trappl, R., editors, *Multi-Agent Systems and Applications*, pages 246–270. Berlin: Springer.
10. Lavrač, N. and Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications.*, New York: Ellis Horwood. Freely available at `http://www-ai.ijs.si/SasoDzeroski/ILPBook/`
11. Smart, W. D., & Kaelbling, L. P. (2000). Practical reinforcement learning in continuous spaces. *In Proc. 17th International Conference on Machine Learning*, pages 903–910. San Francisco, CA: Morgan Kaufmann.
12. Sutton, R. S. (1996) Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Proc. 8th Conference on Advances in Neural Information Processing Systems*, pages 1038–1044. Cambridge, MA: MIT Press.
13. Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press.