Bellman goes Relational

Kristian Kersting[1] KERSTING@INFORMATIK.UNI-FREIBURG.DE
University of Freiburg, Machine Learning Lab, Georges-Koehler-Allee 079, 79110 Freiburg, Germany

Martijn Van Otterlo[1] OTTERLO@CS.UTWENTE.NL
University of Freiburg, Machine Learning Lab, Georges-Koehler-Allee 079, 79110 Freiburg, Germany
Twente University, Department of Computer Science, TKI, P.O. Box 217, 7500 AE Enschede, The Netherlands

Luc De Raedt DERAEDT@INFORMATIK.UNI-FREIBURG.DE
University of Freiburg, Machine Learning Lab, Georges-Koehler-Allee 079, 79110 Freiburg, Germany

Abstract

Motivated by the interest in relational reinforcement learning, we introduce a novel relational Bellman update operator called ReBel. It employs a constraint logic programming language to compactly represent Markov decision processes over relational domains. Using ReBel, a novel value iteration algorithm is developed in which abstraction (over states and actions) plays a major role. This framework provides new insights into relational reinforcement learning. Convergence results as well as experiments are presented.

1. Introduction

There has been a lot of attention and progress in reinforcement learning (RL) and Markov decision processes (MDPs) recently. Several basic algorithms have been proposed and their behavior is relatively well understood today (Sutton & Barto, 1998). This has led to an increased interest into the effects of generalization and to new challenges. One of them concerns the use of RL in relational domains (Džeroski et al., 2001). Even though a number of relational RL algorithms has been developed — essentially through varying the underlying function approximators (Driessens & Ramon, 2003; Gärtner et al., 2003) — the problem of relational RL is still not well understood and a theory of relational RL is lacking.

Appearing in *Proceedings of the 21st International Conference on Machine Learning*, Banff, Canada, 2004. Copyright by the authors.

In traditional RL, the Bellman backup operator is one of the central concepts. A particularly interesting approach is that of Dietterich and Flann (1997), who showed that value backups in model-based RL can be upgraded to region-based backups, where multiple states are updated simultaneously using a backup operator that reverses the action operators.
Inspired by this work, the key contribution of this paper is the introduction of a relational Bellman backup operator, called ReBel. ReBel is developed within a simple probabilistic STRIPS-like relational formalism that incorporates several elements of relational and logical Markov Decision Programming (Kersting & De Raedt, 2003; Van Otterlo, 2004) such as abstract states that are represented using relational queries. Using ReBel, we then develop a model-based relational RL algorithm and demonstrate it on a number of experiments. The approach is also related to that by Boutilier et al. (2001) who employ a situation calculus based language. Although their work is certainly elegant and principled, due to the complexity of the language, they neither report on a complete implementation nor present automated experiments. In contrast, our approach is simpler and therefore fully automated. It deals fully automatically with the same experimental example that Boutilier et al. report on.

Outline: Section 2 briefly reviews relational logic and MDPs. After discussing value iteration (VI) for MDPs in Section 3, we introduce a language to compactly specify MDPs over relational domains in Section 4. In Section 5, we develop a relational VI algorithm based on ReBel. It is empirically validated in Section 6. Before concluding we discuss related work.

[1]Both authors contributed equally to the paper.

2. Preliminaries

Relational Logic, cf. (Nienhuys-Cheng & de Wolf, 1997): An *alphabet* Σ is a set of relation symbols $\mathtt{p}$ with arity $m \geq 0$, and a set of constants $\mathtt{c}$. An *atom* $\mathtt{p(t_1, \ldots t_m)}$ is a relation symbol $\mathtt{p}$ followed by a bracketed m-tuple of terms $\mathtt{t_i}$. A *term* is a variable $\mathtt{X}$ or a constant $\mathtt{c}$. A *conjunction* A is a set of atoms. The set of variables in a conjunction A is denoted as $\text{vars}(A)$. A substitution θ is a set of assignments of terms to variables $\{\mathtt{X_1/t_1, \ldots X_n/t_n}\}$ where $\mathtt{X_i}$ are variables and all $\mathtt{t_i}$ are terms. A term, atom or conjunction is called *ground* if it contains no variables. Conjunctions are implicitly assumed to be *existentially quantified.* A conjunction A is said to be θ-subsumed by a conjunction B, denoted by $A \preceq_\theta B$, if there exists a substitution θ such that $B\theta \subseteq A$. The *most general unifier* (mgu) for atoms $\mathtt{a}$ and $\mathtt{b}$ is denoted by $\text{mgu}(\mathtt{a}, \mathtt{b})$. A (Horn) clause $H \leftarrow B$ consists of a positive atom H and a conjunction B and can be read as *H is true if B is true.* The *greatest lower bound* (glb) of two conjunctions A and B is the most general conjunction that is subsumed by both A and B. Both subsumption and glb are also defined for clauses. The *Herbrand base* of Σ, HB^Σ, is the set of all ground atoms which can be constructed with the predicate symbols and constants of Σ. An *interpretation* is a subset of HB^Σ.

Our running example will be *blocks world.* Here, a block $\mathtt{X}$ can be moved on top of another block $\mathtt{Y}$, denoted as $\mathtt{move(X, Y)}$. Valid relations are $\mathtt{on(X, Y)}$, i.e. block $\mathtt{X}$ is on $\mathtt{Y}$, and $\mathtt{cl(Z)}$, i.e. block $\mathtt{Z}$ is *clear.* To model the floor, we follow a common approach. It is a set of blocks which cannot be on top of other blocks.

Markov Decision Processes, cf. (Sutton & Barto, 1998): A *Markov Decision Process* (MDP) is a tuple $M = \langle S, A, T, R \rangle$, where S is a set of states, A a set of actions, $T : S \times A \times S \to [0, 1]$ a *transition model* and $R : S \times A \times S \to [0, 1]$ a *reward model.* The set of actions *applicable* in a state $s \in S$ is denoted $A(s)$. A transition from state $i \in S$ to $j \in S$ caused by some action $a \in A(i)$ occurs with probability $T(i, a, j)$ and a reward $R(i, a, j)$ is received. T defines a proper probability distribution if for all states $i \in S$ and all actions $a \in A(i)$: $\sum_{j \in S} T(i, a, j) = 1$. A deterministic *policy* $\pi : S \to A$ for M specifies which action $a \in A(s)$ will be executed when the agent is in some state $s \in S$, i.e. $\pi(s) = a$.

3. Value Iteration

Given some MDP $M = \langle S, A, T, R \rangle$, a policy π for M, and a *discount factor* $\gamma \in [0, 1]$, the *state value function* $V^\pi : S \to \mathbb{R}$ represents the value of being in a state following policy π, w.r.t. expected rewards. A similar *state-action value function* $Q^\pi : S \times A \to \mathbb{R}$ can be defined. A policy π^* is optimal if $V^{\pi^*}(s) \geq V^{\pi'}(s)$ $\forall s \in S$ and $\forall \pi'$. Optimal value functions are denoted V^* and Q^*. Bellman's (1957) *optimality equation* states:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')] \quad (1)$$

From this equation, basically all methods for solving MDPs can be derived. For example, the well-known exact solution technique *value iteration* (VI) is obtained from (1) by turning it into an update rule:

$$V_{t+1}(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_t(s')] \quad (2)$$

$$= \max_a Q_{t+1}(s, a). \quad (3)$$

Based on Equation (2), the VI algorithm can be stated as follows: starting with a value function V_0 over all states, we iteratively update the value of each state according to (2) to get the next value functions V_t ($t = 1, 2, 3, \ldots$). VI is guaranteed to converge in the limit towards V^*, i.e. the Bellman optimality equation (1) holds for each state.

Traditional VI as expressed by Equation (2) assumes that all states and values are represented explicitly in a table. This is impractical for all but the smallest state spaces. Furthermore, for relational domains, where the number of states can grow very large (even infinitely large) this is infeasible. Therefore, methods that make abstract from specific states are needed. Such a method is developed in the next sections.

4. Markov Decision Programs

Traditional MDPs are essentially propositional in that each state can be represented using a separate proposition. In *Markov decision programs* these propositional symbols are replaced by abstract states:

Definition 1 *An abstract state is a conjunction Z of logical atoms, i.e., a logical query.*

Abstract states represent sets of states. More formally, a state is an interpretation, i.e. a set of grounds facts. Consider e.g. the state $z = \mathtt{cl(a), cl(b), on(a, c)}$ in the blocks world. An abstract state Z is, e.g., $\mathtt{cl(X)}$. It represents all states that are subsumed by Z, i.e., all interpretations in which there exists something that is clear.

We can now introduce the basic ingredients of Markov decision programs, namely, **abstract actions**, **abstract rewards**, and **integrity constraints**.

An **abstract action** is defined as follows.

Definition 2 *An* action[2] *is a finite set of action rules $H_i \xleftarrow{p_i:A} B$ where A is an atom representing the name and the arguments of the action and B is an abstract state denoting the preconditions of A. H_i is the i-th possible outcome of A. It holds that $\sum_i p_i = 1$.*

We assume that $\text{vars}(A) = (vars(H_i) \cup vars(B))$. The semantics of the action definition are: *If the current state b is subsumed by B, i.e., $b \preceq_\theta B$, then taking action A will result in $[b \setminus B\theta] \cup H_i\theta$ with probability p_i.* So, if the preconditions are fullfilled, all outcomes are possible. As an illustration, consider

$$\begin{array}{lcl} \texttt{on(X,Y),cl(X),cl(Z),} & \xleftarrow{0.9:\texttt{move(X,Y,Z)}} & \texttt{cl(X),cl(Y),on(X,Z),} \\ \texttt{X} \neq \texttt{Y},\texttt{Y} \neq \texttt{Z},\texttt{X} \neq \texttt{Z} & & \texttt{X} \neq \texttt{Y},\texttt{Y} \neq \texttt{Z},\texttt{X} \neq \texttt{Z} \\ \texttt{cl(X),cl(Y),on(X,Z),} & \xleftarrow{0.1:\texttt{move(X,Y,Z)}} & \texttt{cl(X),cl(Y),on(X,Z),} \\ \texttt{X} \neq \texttt{Y},\texttt{Y} \neq \texttt{Z},\texttt{X} \neq \texttt{Z}. & & \texttt{X} \neq \texttt{Y},\texttt{Y} \neq \texttt{Z},\texttt{X} \neq \texttt{Z}. \end{array}$$

which moves block `X` on `Y` with probability 0.9. With probability 0.1 the action fails, i.e., we do not change the state. Applied to the above state z the action tells us that $\texttt{move(a,b,c)}$ will result in $z' \equiv \texttt{on(a,b),cl(a),cl(c)}$ with probability 0.9 and with probability 0.1 we stay in z. This type of action definition implements a kind of probabilistic STRIPS operator.

The model **R** of **abstract rewards** specifies the rewards generated by entering abstract states. In our framework it coincides with our initial *abstract state value* function V_0.

Definition 3 *An abstract state value function V is a finite list of value rules of the form $c \leftarrow B$ were B is an abstract state and $c \in \mathbb{R}$.*

To any abstract state Z, V assigns the maximal value c of all matching value rules $c \leftarrow B$ to Z as value. A rule matches if $Z \preceq_\theta B$. Consider e.g. $\mathbf{R} = V_0$ as

$$10.0 \leftarrow \texttt{on(a,b)}. \quad \text{and} \quad 0.0 \leftarrow \texttt{true}.$$

It assigns 0 to z but 10 to z'. Using `true` in the last value rule assures that all state are assigned a value. To develop ReBel, we will also employ *abstract action-state value* functions, which are similar to abstract state value functions and of which an example can be found in Section 5.2.

Definition 4 *An abstract state action value function Q is a finite set of Q-rules of the form $c : A \leftarrow B$ were B is an abstract state, A is an action and $c \in \mathbb{R}$.*

[2]For the sake of simplicity, we consider cost-free actions. The framework can be adapted to the case of action costs. Note also that the meaning of *abstract action* here differs from that sometimes used in the context of hierarchical RL.

To any abstract state-action "pair" B and A, Q assigns the maximal value c of all abstract state action rules subsumed by $A \leftarrow B$.

Rewards are specified over queries, i.e., existentially quantified goals. Although these are simple, they are expressive enough to specify many interesting problems studied by the (relational) RL community such as *shortest-path problems*. Here, the goal is to reach certain (abstract) states. When a goal state is entered, the process ends. In RL, episodic tasks are encoded using *absorbing states*. We encode it by *artificial* deterministic actions such as $\texttt{on(a,b)} \xleftarrow{1.0:\texttt{absorbing}} \texttt{on(a,b)}$, which denotes that all states that are subsumed by $\texttt{on(a,b)}$ transition only to themselves and generate only zero rewards. For example, z is not absorbing but z' is.

Finally, we need a way to cope with the **integrity constraints** imposed by our domain. For instance, in the `move` definition above we employed symmetry of $\neq$. This can be modeled by a set **C** of integrity constraints. Each integrity constraint is a Horn clause. For instance in the blocks world, no block can be free if there is a block on top of it and no block can be on itself: $\texttt{false} \leftarrow \texttt{on(X,Y),cl(Y)}$ and $\texttt{X} \neq \texttt{Y} \leftarrow \texttt{on(X,Y)}$. The completion of an abstract state Z is the least fixpoint of $\mathbf{C} \cup \{Z\}$, i.e., all facts deducible from $\mathbf{C} \cup \{Z\}$. For example, $\texttt{on(a,b)}$ does not encode that `a` is not `b`. Using the rules above, this state is completed to $\texttt{on(a,b)}, \texttt{a} \neq \texttt{b}$. Furthermore, if the completion includes `false`, the state does not satisfy the constraints, i.e., it is an *illegal* state. To deal with integrity constraints, we also have to adapt our notations of action definitions and generality. Action definitions are now constrained so that they cannot lead to illegal states. For subsumption we employ the integrity constraints as a background theory and use Buntine's *generalized subsumption* framework (Buntine, 1988).

Along the lines of (Kersting & De Raedt, 2003; Van Otterlo, 2004), it can be proven that any Markov decision program induces a (possibly infinite) MDP.

5. Relational Value Iteration: ReBel

We will now develop a value iteration algorithm for Markov decision programs, i.e., **given** an abstract reward model **R**, i.e., initial abstract state value function V_0, **compute** the next abstract state value functions V_t, $t = 1, 2, \ldots$

The main idea is to upgrade Bellman's traditional backup operator in Equation (2). Therefore, we iterate over:

1): Regress all preceding abstract states from V_t.

2): Compute Q_{t+1} over the regressed states.
3): Compute V_{t+1} by maximizing over Q_{t+1}.
We will now discuss each step in turn.

5.1. Regression

Let V_t be the current abstract state value function, say V_0, and consider the abstract action `move`. For a single Bellman backup, all abstract states S which lead to a condition in V_0 when taking action `move` have to be computed. Thus, we have to reason from post- to preconditions. For example, the first outcome of $\mathtt{move(a,b,c)}$ can lead from state $S \equiv \big(\mathtt{cl(a)}, \mathtt{cl(b)}, \mathtt{on(a,c)}, \mathtt{on(b,d)}\big)$ (inequality constraints omitted) to the abstract state $S' \equiv \mathtt{on(a,b)}$. Thus, we have to compute the weakest preconditions for the outcomes of `move` and S.

Definition 5 *All abstract states which lead to S' when following some action rule $H_i \xleftarrow{p_i:A} B$ constitute the so called weakest precondition* $\mathrm{wp}_i(A, S')$ *of the i-th outcome of A.*

For example, S lies in the weakest precondition of S', i.e., $S \in \mathrm{wp}_1(\mathtt{move(X,Y,Z)}, S')$ but it does not lie in $\mathrm{wp}_2(\mathtt{move(X,Y,Z)}, S')$

To compute $\mathrm{wp}_1(\mathtt{move(X,Y,Z)}, S')$ we can assume that we "moved" from S to S'. Thus, 1) the preconditions of the action (rule) are fullfilled in S, and 2) S' is partially caused by the first outcome of the action. As an illustration of 2), consider $\mathtt{on(a,b)}$:

move caused on(a, b): We have been in abstract state $S_1 \equiv \big(\mathtt{cl(a)}, \mathtt{cl(b)}, \mathtt{on(a,Z)}, \mathtt{a} \neq \mathtt{b}, \mathtt{a} \neq \mathtt{Z}, \mathtt{b} \neq \mathtt{Z}\big)$ and moved $X = \mathtt{a}$ on $Y = \mathtt{b}$.

move did not cause on(a, b): We moved `X` on `Y` but not `a` on `b`. Therefore, we have been in abstract states $T \equiv \big(\mathtt{cl(X)}, \mathtt{cl(Y)}, \mathtt{on(X,Z)}, \mathtt{on(a,b)}, \mathtt{X} \neq \mathtt{Y}, \mathtt{X} \neq \mathtt{Z}, \mathtt{Y} \neq \mathtt{Z}\big)$ satisfying that we did not move `a` on `b`, i.e., $\mathtt{on(X,Y)} \neq \mathtt{on(a,b)}$, and that we did not move `a` from `b` away, i.e., $\mathtt{on(X,Z)} \neq \mathtt{on(a,b)}$. The constraints guarantee that applying $\mathtt{move(X,Y,Z)}$ in T preserves $\mathtt{on(a,b)}$. The definition of S simplifies to $S_2 \equiv (T \wedge \mathtt{X} \neq \mathtt{a})$, $S_3 \equiv (T \wedge \mathtt{X} \neq \mathtt{a} \wedge \mathtt{Z} \neq \mathtt{b})$, $S_4 \equiv (T \wedge \mathtt{Y} \neq \mathtt{b} \wedge \mathtt{X} \neq \mathtt{a})$, and $S_5 \equiv (T \wedge \mathtt{Y} \neq \mathtt{b} \wedge \mathtt{Z} \neq \mathtt{b})$. All S_i are completed to the same state namely $S_6 \equiv \mathtt{cl(A)}, \mathtt{cl(B)}, \mathtt{on(a,b)}, \mathtt{on(A,C)}$ where all variables and constants are mutually different.

The abstract states S_1, S_6 together logically define $\mathrm{wp}_1(\mathtt{move(X,Y,Z)}, S') \equiv \big(S_1 \vee S_6\big)$.

So far, we considered a single effect only, namely $\mathtt{on(a,b)}$. In general, however, there can be multiple

```
1: initialize $\mathrm{wp}_i$ to be the empty list
2: for each subset $S''$ of $S'$ and subset $P$ of $H_i$ such
   that $\theta = \mathrm{mgu}(S'', P)$ exists
        or $S'' == \emptyset \wedge P == H_i$, i.e., $\theta = \emptyset$ do
3:    $S := (S'\theta \setminus P\theta) \cup B\theta$
4:    for all pairs $(l, l')$ in
      $\{(l,l') \mid l \in (S'\theta \setminus P\theta) \wedge l' \in H_i\theta \cup B\theta\}$ do
5:      if $\mathrm{mgu}(l, l')$ exists then
6:        add $l \neq l'$ to $S$
7:    add all simplifications of $S$ to $\mathrm{wp}_i$
8: return $\mathrm{wp}_i$
```

Procedure 1: WeakestPre returns the weakest precondition of action rule $H_i \xleftarrow{p_i:A} B$ and abstract state S' given a set of integrity constraints $\mathbf{C}$. We omitted that only legal and completed abstract states are inserted in wp_i.

(combined) effects that are or that are not caused by taking action `move`, cf. WeakestPre in Procedure 1. Consider for example $S \equiv \big(\mathtt{on(a,b)}, \mathtt{on(c,d)}\big)$. Moving a block on some other block can have caused either $\mathtt{on(a,b)}$ or $\mathtt{on(c,d)}$, or neither of them, cf. line 2. Assume that no effect was caused. Then, S'' is empty and $P = H_1$, cf. line 2. Therefore, θ is the empty substitution and $S \equiv \big(\mathtt{on(a,b)}, \mathtt{on(c,d)}, \mathtt{cl(X)}, \mathtt{cl(Y)}, \mathtt{on(X,Z)}\big)$ (inequality constraints omitted) is a possible preimage, cf. line 3. However, we know that `move` did not cause $\mathtt{on(a,b)}, \mathtt{on(c,d)}$. Therefore, it holds $\mathtt{on(X,Z)} \neq \mathtt{on(a,b)} \wedge \mathtt{on(X,Z)} \neq \mathtt{on(c,d)} \wedge \mathtt{on(X,Y)} \neq \mathtt{on(a,b)} \wedge \mathtt{on(X,Y)} \neq \mathtt{on(c,d)}$, cf. lines 4–6. S can be simplified for instance to $S, \mathtt{X} \neq \mathtt{a}, \mathtt{X} \neq \mathtt{c}$ which is a legal abstract state. The case that the action caused some effects is covered by the "$\mathrm{mgu}(S'', P)$ *exists*" conditition in line 2. It is treated analogously.

5.2. Computing Abstract State Action Values

Given the regressed abstract states and the current abstract state value function V_t, we now compute an abstract state-action value function Q_{t+1} according to Procedure 2. To do so, **(A)** we treat each outcome of an action A as though it would be a single action and compute its abstract state action value, cf. line 4. Then, **(B)** we combine the values of all outcomes to an abstract state action value for A, cf. lines 8–12. For the sake of brevity, we will not state constraints in the examples till the end of Section 5.3.

For step **(A)**, consider again the first outcome of `move`. The weakest precondition was $\mathrm{wp}_1(\mathtt{move(X,Y,Z)}, S') \equiv S_1 \vee S_6$. Because S_6, is absorbing, we assign an abstract state action value of 10 for taking action `move`, i.e., $10 : \mathtt{move(X,Y,Z)} \leftarrow S_6$. The value of S_1, however, is dependent on $V_t(S')$, i.e. in our example V_0. Assuming a discount factor of 0.9 this yields $\mathbf{R}(S) + p_1 \cdot 0.9 \cdot V_0(S') = 0{+}0.9{\cdot}0.9{\cdot}10 = 8.1$, i.e., $8.1 : \mathtt{move(a,b,Z)} \leftarrow$

```
1: initialize $Qrules$ to the empty set.
2: for each action rule $H_i \xleftarrow{p_i:A} B$ for $A$ do
3:    for each $v \leftarrow V$ in $V_t$ do
4:       $partialQ := \{\tilde{q} : \tilde{A} \leftarrow S \mid S \in \mathrm{wp}_i(A, V)\}$
         $\tilde{q} := \begin{cases} \mathbf{R}(S) & : S \text{ is absorbing} \\ \mathbf{R}(S) + p_i \cdot \gamma \cdot V_t(V) & : \text{otherwise} \end{cases}$
5:       if $Qrules \neq \emptyset$ then
6:          $Qrules := partialQ$
7:       else
8:          $newQ := \emptyset$
9:          for all pairs $q' : \tilde{A}' \leftarrow S' \in Qrules$ and
                          $q'' : \tilde{A}'' \leftarrow S'' \in partialQ$ do
10:            if $G := \mathrm{glb}(\tilde{A} \leftarrow S', \tilde{A}'' \leftarrow S'')$ exists then
11:               add $q : G$ to $newQ$ with $q = q' + q''$
12:         $Qrules := newQ$
13: return $Qrules$
```

Procedure 2: QRules returns the Q-rules of an action A given the reward model $\mathbf{R}$, the current value function V_t and a discount factor γ. Note that $\tilde{A}$ denotes the action head where we keep the substitution made by wp_i. We also omitted that only legal and completed abstract states g are inserted in *Qrules*.

S_1 . Doing the same for all other rules in V_0 results in:

$$\begin{array}{lrcl} \langle a\rangle & 10 : \mathtt{move(X,Y,Z)} & \leftarrow & \mathtt{cl(X),cl(Y),on(a,b),on(X,Z)} \\ \langle b\rangle & 8.1 : \mathtt{move(a,b,Z)} & \leftarrow & \mathtt{cl(a),cl(b),on(a,Z)} \\ \langle c\rangle & 0.0 : \mathtt{move(X,Y,Z)} & \leftarrow & \mathtt{cl(X),cl(Y),on(X,Z)} \end{array}$$

For the second outcome of `move`, step **(A)** leads to:

$$\begin{array}{lrcl} \langle d\rangle & 1.0 : \mathtt{move(a,X,b)} & \leftarrow & \mathtt{cl(a),cl(X),on(a,b)} \\ \langle e\rangle & 1.0 : \mathtt{move(X,Y,Z)} & \leftarrow & \mathtt{cl(X),cl(Y),on(a,b),on(X,Z)} \\ \langle f\rangle & 0.0 : \mathtt{move(X,Y,Z)} & \leftarrow & \mathtt{cl(X),cl(Y),on(X,Z)} \end{array}$$

For step **(B)**, we note that each of these rules describes situations such as *if we are in a state then we can get some value for achieving the i-th outcome of action A.* This information has to be combined to an abstract state action values for A. To do so, we select a rule from $\langle a\rangle - \langle c\rangle$, say $\langle b\rangle$, and a rule from $\langle d\rangle - \langle f\rangle$, say $\langle f\rangle$, and check whether we can be in both abstract states at the same time and whether we can apply the same action. In other words, we compute the *greatest lower bound* (glb) of the logical clauses underlying both value rules. If the glb (where the actions have to unify) exists and it is a legal state, then it is inserted as a new rule, cf. line 11. The value of the new rule is the sum of values of the combined rules. For $\langle b\rangle$ and $\langle f\rangle$ this yields

$$8.1 : \quad \mathtt{move(a,b,X)} \quad \leftarrow \quad \mathtt{cl(a),cl(b),on(a,X)}.$$

In contrast, $\langle b\rangle$ and $\langle d\rangle$ do not give a new rule.

In our blocks world example, QRules yields the following abstract state action value function when applied on V_0 and `move` and `absorbing`:

```
1: initialize $V_{t+1}$ to the empty set of $V$-rules.
2: sort $Qrules$ in decreasing order of $Q$-values
3: while $Qrules$ not empty do
4:    remove top element $d : A \leftarrow B$ of $Qrules$
5:    if no other rule $d : A' \leftarrow B'$ in $Qrules$ exists such
      that $B'$ subsumes $B$ then
6:       add $d \leftarrow B$ to $V_{t+1}$
7:       remove all rules $d'' \leftarrow B''$ from $Qrules$
         such that $B''$ is subsumed by $B$
8: return $V_{t+1}$
```

Procedure 3: VRules returns the value functions V_{t+1} given the Q-rules computed from V_t for all actions.

$$\begin{array}{lrcl} \langle 1\rangle & 10 : \mathtt{absorbing} & \leftarrow & \mathtt{on(a,b)} \\ \langle 2\rangle & 10 : \mathtt{move(X,Y,Z)} & \leftarrow & \mathtt{cl(X),cl(Y),on(a,b),on(X,Z)} \\ \langle 3\rangle & 8.1 : \mathtt{move(a,b,X)} & \leftarrow & \mathtt{cl(a),cl(b),on(a,X)} \\ \langle 4\rangle & 0.0 : \mathtt{move(X,Y,Z)} & \leftarrow & \mathtt{cl(X),cl(Y),on(X,Z)} \end{array}$$

Note that we have sorted the Q-rules in descending order only for the sake of readability.

5.3. Computing Abstract State Values

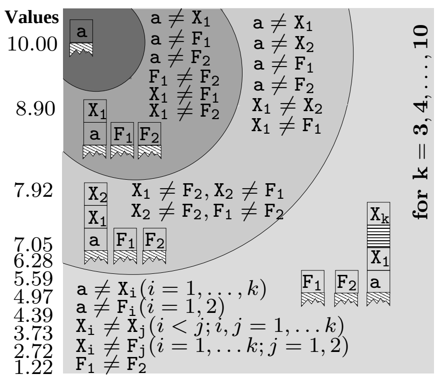
The set of Q-rules enables one to compute the next abstract state value function V_{t+1}. In contrast to the traditional case, Q-rules, i.e., values of abstract state action pairs, can overlap such as Q-rules $\langle 1\rangle$ and $\langle 2\rangle$. To compute abstract state values we make use of the fact that $V_{t+1}(S) = \max_A Q_{t+1}(S, A)$ due to Eq. (3).

In general, any value-preserving transformation can be applied. In this paper, we use a simple separate-and-conquer rule learning approach where the rules to learn and the examples to learn from coincide, see VRules in Procedure 3. We search for a Q-rule m having a maximal Q-value among *Qrules*, lines 3–4, separate the covered Q-rules, line 5, and recursively conquer the remaining Q-rules by selecting more rules until no Q-rules remain, line 6. The main difference is that we select m and add it to V_{t+1} only if there is no other Q-rule left in *Qrules* with the same value whose body subsumes the body of m, cf. line 8. In our running example, we start with rule $\langle 1\rangle$. Because it is not subsumed by any other rule having the same value, we add $10 \leftarrow \mathtt{on(a,b)}$ to V_1 and, because it subsumes $\langle 2\rangle$, we remove $\langle 2\rangle$ from *Qrules*. The remaining highest valued rule is $\langle 3\rangle$, and we iterate. After completing, this yields the new value function V_1 (constraints listed again):

$$\begin{array}{rcl} 10 & \leftarrow & \mathtt{on(a,b)}, \mathtt{a} \neq \mathtt{b}. \\ 8.1 & \leftarrow & \mathtt{cl(a),cl(b),on(a,X)}, \mathtt{a} \neq \mathtt{b}, \mathtt{a} \neq \mathtt{X}, \mathtt{b} \neq \mathtt{X}. \\ 0 & \leftarrow & \mathtt{cl(X),cl(Y),on(X,Z)}, \mathtt{X} \neq \mathtt{Y}, \mathtt{X} \neq \mathtt{Z}, \mathtt{Y} \neq \mathtt{Z}. \end{array}$$

5.4. Relational Bellman Backup Operator

To summarize, the general scheme of ReBel is: **1)** Compute the weakest precondition of each action



Values
10.00
8.90
7.92
7.05
6.28
5.59
4.97
4.39
3.73
2.72
1.22
a
$\mathtt{a} \neq \mathtt{X_1}$
$\mathtt{a} \neq \mathtt{F_1}$
$\mathtt{a} \neq \mathtt{F_2}$
$\mathtt{F_1} \neq \mathtt{F_2}$
$\mathtt{X_1} \neq \mathtt{F_1}$
$\mathtt{X_1} \neq \mathtt{F_2}$
$\mathtt{a} \neq \mathtt{X_1}$
$\mathtt{a} \neq \mathtt{X_2}$
$\mathtt{a} \neq \mathtt{F_1}$
$\mathtt{a} \neq \mathtt{F_2}$
$\mathtt{X_1} \neq \mathtt{X_2}$
$\mathtt{X_1} \neq \mathtt{F_1}$
$\mathtt{X_1}$
a
$\mathtt{F_1}$
$\mathtt{F_2}$
$\mathtt{X_2}$
$\mathtt{X_1}$
a
$\mathtt{F_1}$
$\mathtt{F_2}$
$\mathtt{X_1} \neq \mathtt{F_2}, \mathtt{X_2} \neq \mathtt{F_1}$
$\mathtt{X_2} \neq \mathtt{F_2}, \mathtt{F_1} \neq \mathtt{F_2}$
$\mathtt{X_k}$
$\mathtt{X_1}$
a
$\mathtt{F_1}$
$\mathtt{F_2}$
$\mathtt{a} \neq \mathtt{X_i} (i = 1, \ldots, k)$
$\mathtt{a} \neq \mathtt{F_i} (i = 1, 2)$
$\mathtt{X_i} \neq \mathtt{X_j} (i < j; i, j = 1, \ldots k)$
$\mathtt{X_i} \neq \mathtt{F_j} (i = 1, \ldots k; j = 1, 2)$
$\mathtt{F_1} \neq \mathtt{F_2}$
for $\mathbf{k = 3, 4, \ldots, 10}$

Figure 1. **Blocks World Experiment I:** Abstract state value function for the $\mathtt{cl(a)}$ goal after 10 iterations. It applies for any number of blocks. Values are rounded to the second digit. F_i can be a block or a floor block. States structurally different from the depicted ones get value 0.0 .

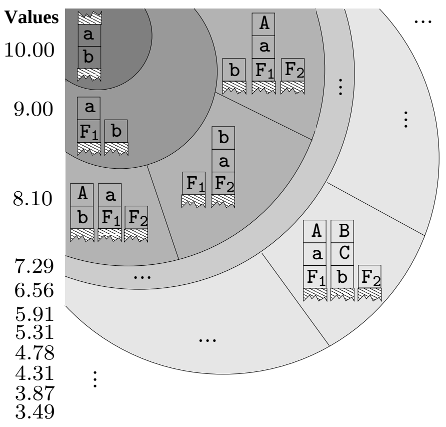


Figure 2. **Blocks World Experiment II:** Parts of the abstract value function for $\mathtt{on(a,b)}$ after 10 iterations (values rounded to the second digit). It applies for any number of blocks. We omitted the inequality constraints: All blocks are mutually different. F_i can be a block or a floor block. State more than 10 steps away from the goal get value 0.0.

outcome for each abstract state in V_t using WEAKESTPRE. As done in QRULES, **2a)** assign to each abstract state – action outcome pair computed in 1) a Q-value and **2b)** combine them based using the glb. **3)** Maximize the Q-rules to compute V_{t+1} using VRULES.

Note that in **2b)**, if there are $n > 1$ many outcomes of an action, then the Q-values of the n-th outcome are combined with already combined Q-values of the $n-1$ previous outcomes. Thus, there are $n-1$ many combinations per action. This might produces many rules. To overcome this, one can adapt VRULES maximizing Q-rules to compress Q-rules: if we are in a state with different currently combined values for compatible actions, then we select only the higher one. This is safe because the higher valued Q-rule subsumes the lower valued one. Therefore, it would have been selected in any case later on.

Formally, this Bellman backup requires an infinite number of iterations to converge to V^*, cf. Section 6. In practice, we stop when the abstract value function changes by only a small amount.

6. Experiments

In this section we empirically validate REBEL. We implemented REBEL with compressing *Qrules* in the Prolog system YAP version 4.4.4. and we used the supplemented *constraint handling rules* library (Frühwirth, 1998). In all experiments we assume a discount factor of 0.9 and a goal reward of 10, i.e., in all other states we receive 0 reward. Only goal states are absorbing. Experiments were run on a 3.1 GHz Linux machine. The running times were estimated using YAP's buildin `statistics(runtime,·)`. We focused on standard examples known from the relational RL literature.

Blocks World Experiment I: We consider $\mathtt{cl(a)}$ as goal in our probabilistic blocks world setting. The experiment shows that even on simple problems REBEL is not guaranteed to converge on the structural level.

Figure 1 shows the abstract state value function after 10 iterations. It took REBEL roughly 1 minute to iterate ten times. Figure 1 highlights that states that are one step further away from the goal get the same value. The value, however, is lower because of the additional block on top of the stack of $\mathtt{a}$. Thus, because the number of blocks is not restricted, value iteration will never stop.

Proposition: Abstraction does not guarantee convergence in infinite domains because an infinite number of abstract states can be required.

This is interesting, because infinite state spaces easily arise when relational representations are used and relational abstraction was hoped to be a solution. Nevertheless, relational value iteration can converge even for infinite domains as our third experiment will show.

Blocks World Experiment II: We consider the goal $\mathtt{on(a,b)}$ in a deterministic blocks world because it is reported to be a hard problem for model-free relational RL (RRL) approaches (Džeroski et al., 2001; Driessens & Ramon, 2003). For instance, Driessens and Ramon (2003) report that on average the learned policies did not reach optimal performance even for 5 blocks.

Using the same experimental set-up as in our first experiment but a deterministic `move` action, REBEL

	V_t									
abstract states	1	2	3	4	5	6	7	8	9	10
`bin(b, p).`	**10.000**	10.000	10.000	10.000	10.000	10.000	10.000	10.000	10.000	10.000
`tin(A, p), on(b, A), not_rain.`	**8.100**	**8.829**	**8.895**	**8.901**	8.901	8.901	8.901	8.901	8.901	8.901
`tin(A, p), on(b, A), rain.`	**6.300**	**8.001**	**8.460**	**8.584**	**8.618**	**8.627**	**8.629**	**8.630**	8.630	8.630
`tin(A, B), on(b, A), not_rain.`		**7.290**	**7.946**	**8.005**	**8.010**	**8.011**	8.011	8.011	8.011	8.011
`tin(A, B), on(b, A), rain.`		**5.670**	**7.201**	**7.614**	**7.726**	**7.756**	**7.764**	**7.766**	**7.767**	7.767
`tin(A, B), bin(b, B), not_rain.`			**5.905**	**6.968**	**7.111**	**7.128**	**7.130**	**7.131**	7.131	7.131
`tin(A, B), bin(b, B), rain.`			**3.572**	**5.501**	**6.282**	**6.563**	**6.658**	**6.689**	**6.699**	**6.702**
`tin(A, B), bin(b, C), not_rain.`				**5.314**	**6.271**	**6.400**	**6.416**	**6.417**	**6.418**	6.418
`tin(A, B), bin(b, C), rain.`				**3.215**	**4.951**	**5.654**	**5.907**	**5.993**	**6.020**	**6.029**
`tin(A, B).`	**0.000**	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Table 1. **Load-Unload Experiment:** The t-th column shows the abstract state value function after the t-th iteration. When no value is given, the abstract state has value 0.0. Bold numbers highlight changed values.

computed V_{10} in less than 12 minutes. The abstract value function is partially shown in Figure 2. Because the `move` action is deterministic, V_{10} is optimal for 10 blocks (more than 58 million ground states). The optimal policy can directly be extracted by computing the maximizing Q-rules for each abstract state. In our example, this results in removing the top elements from the stacks on top of `a` and `b`. However, to compactly represent this strategy, one needs to define the predicate `ontop`. In the experiments Driessens and Ramon (2003) reported on, this was always the case. The policy based on REBEL is optimal no matter how many blocks there are.

Load-Unload Experiment: Our final experiment considers the logistics domain which Boutilier et al. (2001) solved semi-automatically. The domain consists of cities, trucks and boxes. Boxes can be loaded onto and unloaded from trucks, and trucks can be driven between cities. The predicate `on(B, T)` denotes that a box `B` is on the truck `T`, `bin(B, C)` denotes that a box `B` is in some city `C` and `tin(T, C)` denotes that a truck `T` is in city `C`. The actions that can be performed are: `load(B, T)` and `unload(B, T)` specifying how a box `B` can be loaded onto or loaded from a truck `T` and `drive(T, C)` specifying that the truck `T` is driven to city `C`. The actions in this domain have probabilistic effects. The probability of failing a `load` or `unload` action, i.e., staying in the current state, depends on whether it rains or not, denoted by `rain`. The action specification is as follows (we omit the failing specifications for the sake of brevity):

$$
\begin{aligned}
\mathtt{bin(B,C)}, \mathtt{tin(T,C)}, R &\xleftarrow{pr:\mathtt{unload(B,T)}} \mathtt{on(B,T)}, \mathtt{tin(T,C)}, R \\
\mathtt{on(B,T)}, \mathtt{tin(T,C)}, R &\xleftarrow{pr:\mathtt{load(B,T)}} \mathtt{bin(B,C)}, \mathtt{tin(T,C)}, R \\
\mathtt{tin(T,C')}, \mathtt{C} \neq \mathtt{C'} &\xleftarrow{1.0:\mathtt{drive(T,C')}} \mathtt{tin(T,C)}
\end{aligned}
$$

where the probability pr is 0.9 if R is `rain` and 0.7 if R is `not_rain`. To correctly handle the *explicit negation* we used for `rain`, we provided `false` $\leftarrow$ `rain, not_rain` as constraint. The goal in this domain is to get some box `b` in `p` where `p` stands for *Paris*, i.e., in `bin(b, p)` we get a reward of 10.

REBEL ran for less than 6 seconds to compute the results summarized in Table 1. In contrast to the blocks world examples, the solution converges both at the *value* level and at the *structural* level. E.g., take the situation in which a truck is in a city different from Paris and the box is there too. Then, it will take three steps (`load` − `drive` − `unload`) to reach the goal state and the state value in V_{10} is 6.702 in case it rains. The abstract state value function applies no matter how many trucks, boxes and cities are present.

7. Related Work

In the past few years, there has been an increased and significant interest in using rich relational representations for modeling and learning MDPs.

In model-free relational RL, one has studied different relational learners for function approximation (Džeroski et al., 2001; Lecoeuche, 2001; Driessens & Ramon, 2003; Gärtner et al., 2003). Others have applied Q-learning based on pre-specified abstract state spaces: Kersting and De Raedt (2003) investigate pure Q-learning, Van Otterlo (2004) learns the Q-function via learning the underlying transition model. Fern et al. (2003) extended previous work on upgrading learned policies for small relational MDPS (RMDPs) with *approximated policy iteration*. Finally, Guestrin et al. (2003) recently reported on class-based, approximate value functions for RMDPs.

For model-based approaches, there has been a surprising lack of research on *exact* solution methods. From a general point of view, REBEL is closely related to *decision theoretic regression* (DTR) (Boutilier et al., 1999) and, because of that, it is also related to regression planning in the same way as DTR is. Within DTR, most algorithms are designed to work with *propositional* representations. Actually, the only exception the authors know of is that of Boutilier et al. (2001). REBEL relates to this in that it also is a model-based exact solution method for RMDPs. One key difference is that Boutilier et al. employ situation calculus

for representing RMDPs. Situation calculus is very expressive and as a consequence it is harder to simplify the logical descriptions of the abstract value functions states that are obtained. This may also explain why — to the best of the authors' knowledge — that approach has not been fully implemented and experimented with. In contrast, because of the use of a simpler logical language, the simplification in ReBel is computationally feasible. As shown in the experiments, ReBel successfully and fully automatically implements a relational value iteration.
Finally, the work by Dietterich and Flann (1997) is also concerned with generalizing Bellman backups but no relational representation is used.

8. Conclusions

The key contribution of this paper is the introduction of ReBel, a relational upgrade of the Bellman update operator. It has been used to implement a relational value iteration algorithm. It has been shown to be effective in a number of simple though significant examples. This – in turn – has led to a number of novel insights into relational MDPs. First, it has been shown that value-based methods for relational MDPs may not converge because an infinite number of abstract states has to represented. Second, we highlighted that in such cases *background knowledge* may enable the learning of optimal policies. So, depending on the representation of the problem, one can or cannot learn the optimal policy. Therefore, using background knowledge in RMDPs is not only an interesting feature, but in some cases also a necessity for successful learning. In this way, we have given an explanation for and confirmed some of the experimental insights of the early relational RL work (Džeroski et al., 2001).

Further work could address combining ReBel with other types of value-based methods, extending the representation language, efficient data structures, complexity analysis, and employing other learning algorithms to compress value functions.

The authors hope that the theoretical insights, as well as the algorithm developed in this paper, will be helpful in advancing the field of relational RL as well as contribute to an improved understanding of the problems involved.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments. This research was supported by the European Union IST programme, contract no. FP6-508861, *APrIL II*. Martijn Van Otterlo was supported by a Marie Curie fellowship at DAISY, HPMT-CT-2001-00251.

References

Bellman, R. E. (1957). *Dynamic programming*. Princeton, New Jersey: Princeton University Press.

Boutilier, C., Dean, T., & Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *J. Art. Intel. Res.*, *11*, 1–94.

Boutilier, C., Reiter, R., & Price, B. (2001). Symbolic dynamic programming for first-order MDP's. *Proc. of IJCAI'01*.

Buntine, W. (1988). Generalized subumption and its applications to induction and redundancy. *Artificial Intelligence*, *36*, 149–176.

Dietterich, T. G., & Flann, N. S. (1997). Explanation-based learning and reinforcement learning: a unified view. *Machine Learning*, *28*, 169–210.

Driessens, K., & Ramon, J. (2003). Relational instance based regression for relational reinforcement learning. *Proc. of ICML-2003*.

Džeroski, S., De Raedt, L., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, *43*, 7–52.

Fern, A., Yoon, S., & Givan, R. (2003). Approximate policy iteration with a policy language bias. *Proc. of NIPS'03*.

Frühwirth, T. (1998). Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming*, *37*, 95–138.

Gärtner, T., Driessens, K., & Ramon, J. (2003). Graph kernels and Gaussian processes for relational reinforcement learning. *Proc. of ILP'03*.

Guestrin, C., Koller, D., Gearhart, C., & Kanodia, N. (2003). Generalizing plans to new environments in relational MDPs. *Proc. of IJCAI'03*.

Kersting, K., & De Raedt, L. (2003). Logical Markov decision programs. *Proc. of the IJCAI'03 Workshop on Learning Statistical Models of Relational Data*.

Lecoeuche, R. (2001). Learning optimal dialogue management rules by using reinforcement learning and inductive logic programming. *Proc. of the North American Chapter of the Association for Computational Linguistics (NAACL)*. Pittsburgh.

Nienhuys-Cheng, S.-H., & de Wolf, R. (1997). *Foundations of inductive logic programming*, vol. 1228 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Sutton, R., & Barto, A. (1998). *Reinforcement learning: an introduction*. Cambridge: The MIT Press.

Van Otterlo, M. (2004). Reinforcement learning for relational MDPs. *Machine Learning Conference of Belgium and the Netherlands (BeNeLearn'04)*.