Planning for Relational Rules

Yannick Schroecker
Department of Computer Science
Technische Universitaet Darmstadt
Darmstadt, Germany
yannickschroecker@googlemail.com

Abstract

Planning refers to the notion of finding actions that are near-optimal in order to accomplish a specified goal. Relational rules are a powerful language to represent domains and have been used for a long time in a subfield of machine learning called Inductive Logic Programming. The combination of these two fields results in planning algorithms for domains that are modelled using relational rules and has attracted the interest of researchers for the past decade. This paper aims to give an introduction to this topic.

1 Introduction

Planning, i.e. finding optimal sequences of actions or *plans* in order to accomplish a specified goal, has been an important topic in the field of artificial intelligence for a while now. Recent developments in the field of robotics have shown that robots will need to be ready to act autonomously in bigger and more complicated domains, efficient planning algorithms that are able to find near-optimal strategies in a short space of time are a necessity to accomplish this goal. The representation of the world model is of special importance; The language in which the model is to be defined has to be complex enough to express the characteristics of interest but at the same time simple enough to be able to create algorithms that are computationally tractable. The use of relational logic for this purpose has been analyzed in the last decade and has shown to be an efficient way to model large and realistic domains. Promising learning, planning and exploration algorithms have been proposed and will be described in this paper with a strong focus on planning for probabilistic domains.

Deterministic planning algorithms usually view planning as a search through state space while probabilistic relational planning algorithms often try to solve a relational adaptation of Markov Decision Processes ([vanOtterlo-2009]). They can be further divided in model-free planning algorithms (e.g. [Driessens-2006]) which find their plans directly on a set of example state transitions and model-based algorithms (e.g. [Lang-2010], [Kerstings-2004], [Hoffmann-2001]) that work on a set of transition rules that describe the effect of particular actions on the current state. Due to the representation as relational logic, these rules are intuitively understandable and can be crafted by hand. Automatic learning of rules, however, is often desired (especially for probabilistic domains) and promising methods have been introduced as well (e.g. [Pasula-2007]). Another distinction for model-based algorithms can be made depending on whether an algorithm reasons in the grounded

(e.g. [Lang-2010]) or in the lifted (e.g. [Kerstings-2004]) domain, i.e. whether an algorithm replaces all variables by constants denoting particular objects or not.

The representation of the problem domain with relational rules both in general and in specific (using Noisy Deictic Rules, [Pasula-2007]) will be described in chapter 2, followed by a detailed description of the PRADA algorithm ([Lang-2010]) as one particular example in chapter 3. Chapter 4 will review some key design decisions by introducing alternative planning algorithms.

2 Relational World Models

2.1 Domain representations for planning

There are various possible ways to represent domains for a planning task, this section aims to describe a few of them and motivate a relational representation for state- and action-spaces. The easiest way to represent states and actions in a planning domain is to enumerate them. This approach is used in Markov Decision Processes (MDPs). Slightly different definitions of MDPs exist and this paper will use the same definition as [vanOtterlo-2009] where a Markov Decision Process is defined as a 4-tuple (S, A, T, R) where S is a finite set of states, A is a finite set of actions, T is a ternary function $S \times A \times S \rightarrow [0, 1]$ which assigns each $(s, a, s') \in S \times A \times S$ triple a probability value that corresponds to the probability that applying action a in state s results in the new state s' and R is a reward-function $S \times A \times S \rightarrow \mathbb{R}$ which assigns a reward value to each transition and might not necessarily depend on all three parameters.

Regular MDPs can be solved efficiently but are not very suitable for tasks with a very large state space. This becomes especially apparent when learning state transitions as there has to be an example for each possible state since there are no means to abstract over them. A possible solution is to add structure to the state-space by representing the states as a vector where each vector component is element of a finite set of values. Efficient planning algorithms for this kind of structured state-spaces can be utilized by representing the domain as a Factored Markov Decision Process ([Guestrin-2003]). Propositional logic can be represented this way as well by using boolean values as components of the state-vector.

Another possible representation of states would be to use a vector with real-numbered components for both state and actions (parameter-vector). This representation is used e.g. when learning optimal control. The problem can then be modelled as a transition-function f which determines the next state based on the current state and the parameter-vector $\mathbf{u}$ and a reward-function R and the optimal action can then be derived by optimizing R w.r.t. $\mathbf{u}$. This problem can be solved analytically for linear f and quadratic R (using the Linear Quadratic Regulator algorithm) but will often need to be solved numerically.

Consider now the blocks world domain: The blocks world is a simple and commonly used example domain which consists of a robot, a table and several blocks. The task is for the robot to stack the blocks on the table in a particular way by taking the correct high-level actions such as "move block a from table onto block b". While it is possible to represent this problem e.g. with enumerated states or using propositional logic, these methods lack abstraction; It would have certainly a very similar effect to grab one block which lies directly on the table as it has to grab another block which lies directly on the table but the aforementioned representations will need different actions and different rules for each block as they are generally unable to generalize from one block to others. Relational logic is an intuitive and more powerful but yet still simple enough representation for high-level problems such as the blocks world and provides the necessary abstraction capabilities.

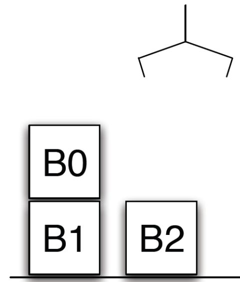

Figure 1:
Simple instance of the blocks world with three blocks. Figure taken from [Pasula-2007]

2.2 Relational Logic

The basic building blocks of relational representations are constants, variables and predicates. Constants are used to represent concrete objects of the domain, a possible constant example that is also used in the blocks world would be $table$. Variables are abstract symbols that can be used in rules to abstract over different objects such as X and predicates are relations of n objects, where n is called the arity of the predicate; an example predicate would be $on/2$ which forms a relation between two objects that may be interpreted as on object being located on top of another. Some other useful definitions include that of terms which are constants or variables, atoms which are predicates of arity n together with n terms (e.g. $on(X, table)$ which could denote that an unknown object represented by the variable X is located on the table), literals which are atoms or their negations and facts which are atoms that are known to be true. Also of interest is the notion of ground-atoms which are atoms that contain only constants and lifted atoms which are atoms that contain only variables. Grounding refers to the act of replacing all variables with constants and lifting refers to the act of replacing all constants with variables.

With these definitions we can already represent the state of relational domains like blocks world. Consider for example a blocks world where we have 3 different blocks: a,b,c; b is stacked on a while a and c lie on the table. We could model this by choosing the constants a, b, c and table, a predicate on/2 (reads: on with an arity of 2) which determines which block is stacked on what and another predicate block/1 which determines if something is a block; the state could then be represented as a conjunction of facts:

$$on(b, a), on(a, table), on(c, table), block(a), block(b), block(c)$$

On top of that, relational logic languages sometimes allow for derived predicates, also called concept predicates, and functions with values that may be compared using an equality operator. Functions and derived predicates have to be defined by some concept language. The concept language used by [Pasula-2007] allows for functions to have integer-values, object-values or values from a discrete set. They also introduce $\leq$ and $\geq$ operators for comparison with integer-valued functions and a counting operator $\#$ that determines how many objects satisfy a given formula. The concept language allows for derived predicates to be determined by conjunction ($\wedge$), quantification ($\exists$, $\forall$) and transitive hull (*). One example for a derived predicate that is true if no object is located on the object denoted by the predicates parameter would be clear/1:

$$clear(X) = \nexists Y.on(Y, X)$$

One example for a function would be height, which is a function that determines how many blocks are below the given block, i.e. the height of the stack with the denoted block as its top

$$height(X) = \#Y.on(X, Y)*$$

2.3 Noisy Deictic Rules

Now that we have already established how to express the state of the world using relational logic we will need to represent actions and transition rules. Actions can be represented as atoms as well but to model rules, i.e. the changes that are achieved by taking those actions, we need an additional representation. STRIPS operators are a basic method to accomplish that and the baseline for most rule definition languages. STRIPS operators consist of preconditions that must be true in order to take the associated action and postconditions that must be true afterwards. They work under the assumption that changes only happen if they are induced by a postcondition (frame assumption) and that all these changes happen at once (outcome assumption). Typically extensions of STRIPS operators are used and this paper will describe Noisy Deictic Rules (NDR), also referred to as Noisy Indeterministic Deictic Rules (NID Rules) presented in [Pasula-2007] since this is the representation that is used in PRADA which will be discussed in detail in chapter 3. The basic form could be defined as follows:

$$r_a(X) : \Phi(X) \rightarrow \Omega(X)$$

where X is a set of variables, r is a rule for an action a with a precondition Φ(called context) which is a list of literals that has to be true before executing a and a postcondition Ω which is a list of literals that shall be true afterwards and represents the change induced by executing a. This is already sufficient to model a number of deterministic domains; A $move/3$ action which makes the robot move a block X which is currently on Y onto block Z could for example be modelled by a rule that is is defined as follows:

$$r_{move}(X, Y, Z) : on(X, Y), clear(X), clear(Z) \rightarrow on(X, Z), \neg on(X, Y) \tag{1}$$

However, actions in the real world rarely have deterministic results. The stack might collapse or the robot might simply fail in its action which would lead to no change to the state, we therefore need a way to deal with indeterminism. Probabilistic domains are usually modelled by a derivative of probabilistic STRIPS operators which define multiple possible outcomes and a probability distribution over these outcomes. On top of that [Pasula-2007] add a noise outcome to each rule which models the fact that things happen which we cant foresee; a collapsing stack will often lead to all blocks laying on the table but it might also do things like knocking over other stacks and some blocks might even fall on top of another. Rules with indeterminism and noise look as follows:

$$r_a(X) : \Phi(X) \rightarrow \begin{cases} p_1 : \Omega_1(X) \\ \vdots \\ p_n : \Omega_n(X) \\ p_{noise} : noise \end{cases} \tag{2}$$

$p_{r,o}$ will be used to denote the probability of the outcome with the index o in the rule r. [Pasula-2007] also introduce a default-rule which has 2 outcomes: no-change and noise to model situations in which no context applies and they add deictic references to rules which are variables that are not

part of the parameters of the actions:

$$r_a(X) : \{V_1 : \Psi_1(X), ..., V_n : \Psi_n(X, V_1, ..., V_{n-1})\} \ \Phi(X) \rightarrow \begin{cases} p_1 : \Omega_1(X) \\ \vdots \\ p_n : \Omega_n(X) \\ p_{noise} : noise \end{cases} \tag{3}$$

Where $V_1, ..., V_n$ are deictic references and $\Psi_1, .., \Psi_n$ are conditions that these references must fulfill. We could now model the $move/2$ action which attempts to move block X onto block Y but fails in 30 percent of the cases by doing nothing or something unexpected as follows:

$$r_{move}(X, Y) : \{V : on(X, V)\} \ clear(X), clear(Y) \rightarrow \begin{cases} 0.7 : on(X, Y), \neg on(X, V) \\ 0.2 : \\ 0.1 : noise \end{cases}$$

2.4 Modelling the Problem: Relational MDP

As already established in chapter 2.4, MDPs are an efficient problem-representation for planning tasks that work with enumerated or factored state-spaces. Many relational planning algorithms adapt the notion of MDPs to Relational Markov Decision Processes (RMDPs). [vanOtterlo-2009] define the Relational Markov Decision Process to be a 4-tuple (S, A, T, R) where S is a subset of the set of ground atoms that can be constructed using a set of constants C and a set of predicates, A is the set of ground atoms that can be constructed using C and a different set of predicates that represent actions A' and T and R are defined as for MDPs. [Kerstings-2004] use a similar definition for ReBel but do not require the atoms to be grounded as the therein presented algorithm ReBel reasons in the abstract domain. While it is possible to find exact solutions to RMDPs ([Boutilier-2001]) it is intractable, therefore most planning methods concentrate on finding an approximate solution.

3 One Algorithm: PRADA

In [Lang-2010] PRADA is presented, a planning algorithm for Noisy Deictic Rules. The algorithm works by building a Dynamic Bayesian Network out of the rules, approximately inferring a reward function and then sampling action sequences, but to begin with, we need to have a set of NDRs. In most real world scenarios we will not be able to compose a useful set of rules by hand and therefore need an algorithm to learn them, [Lang-2010] rely on the algorithm presented in [Pasula-2007] that will be discussed in the following.

3.1 Learning Noisy Deictic Rules

To learn NDRs we need examples of the form (S, A, S') where A is an action, S is the state before and S' is the state after that action is taken. The primitive predicates, objects and possible actions thus need to be already given, for the Blocks World example, the objects could be detected by the robot while $on/2$ could already be given. The examples may be obtained by taking actions after a suboptimal or arbitrary policy.

Concept predicates and functions may be given as well but they can also be learned. To do so, the algorithm takes existing unary predicates and binary predicates and uses the inductive definition of the concept language to create new concept predicates, for example by introducing an existence quantifier for one of the variables. For each possible set of predicates the algorithm learns the ruleset.

This procedure is repeated as long as it improves the scoring metric defined as follows:

$$S'(R) = \sum_{(s,a,s')\in E} log(\hat{P}(s'|s,a,r_{(s,a)})) - \alpha \sum_{r\in R} PEN(r) - \alpha' c(R) \tag{4}$$

where R is the current ruleset, E is the set of examples, α and α' are regularization parameters and $c(R)$ is the amount of learned concept predicates. $\alpha' c(R)$ thus regularizes the amount of learned concepts and penalizes huge amount of very specific concepts. $PEN(r)$ is a penalty for complex rules and could simply be the length of a rule, $\hat{P}$ is the probability of the next state given a rule r, an action a and the previous state s and will be useful later. Given objects, actions, predicates and functions we now want to learn rules for each action. We therefore initialize the ruleset somehow, for example with the default rule where the noise probability is the fraction of examples in which the state is changed and then refine it. [Pasula-2007] define 11 operators that they apply to the rule set and which return one or more new rule sets, then they choose the rule set that scores best and repeat. The algorithm terminates when there are no improvements after applying the 11 operators. 9 of the operators modify the rule set by generalizing or specializing the context, deictic reference set or arguments of a rule and thus try to improve the ruleset, the DropRules operator drops single rules to delete those that are too specialized or useless and the ExplainExamples operator is the only one that defines new rules based on the examples. It operates by choosing examples that are covered only by the default rule and introduces deictic references for everything that changes in S' but is not already an action argument. The context as well as the conditions of the deictic references will be initialized with the conjunction of all literals that can be formed by abstracting over the original state S in the training examples which contain action arguments or deictic references, then it removes every literal that is always true in the training examples as well as concept predicates and functions that can already be deduced by the other predicates. To induce the outcomes for this new rule the algorithm takes the outcomes of the examples and then tries to combine them by conjunction or remove outcomes entirely depending on which action maximizes the resulting score. Then the probability-distribution of the set of outcomes is chosen which maximizes the likelihood term of the scoring metric (4)

$$\sum_{(s,a,s')\in E} log(\hat{P}(s'|s,a,r_{(s,a)}))$$

this usually has to be done numerically as there is generally no analytical solution. PRADA reasons in the grounded domain, so for the next steps all rules are grounded.

3.2 Representation as Dynamic Bayesian Network

Now that we have given a set of NBRs the next step is to determine the expected reward given a sequence of actions. To do so, we set the reward u to 1 if the current state satisfies the goal and 0 otherwise. The expected reward will then be the probability of u being 1 given an action sequence of length t: $a^{0:t-1}$ and an initial state s_0

$$P(u^t = 1|a^{0:t-1}, s^0)$$

To calculate this probability [Lang-2010] build a Dynamic Bayesian Network for $P(u, \mathbf{s}, o, r, \Phi|a, \mathbf{s})$ and use it for approximate inference. A Dynamic Bayesian Network (DBN) is a Bayesian Network that represents the dependencies between the current state and its successor by having both as random variables.

For NDRs the reward u is solely determined by the state s just as the reward of of the next state u' is solely determined by the next state $\mathbf{s}'$. The state $\mathbf{s}$ is a vector of s_i where each s_i represents a ground atom. The probability of u' for a state $\mathbf{s}'$ is 1 iff all ground predicates s'_j equal the value of the goal

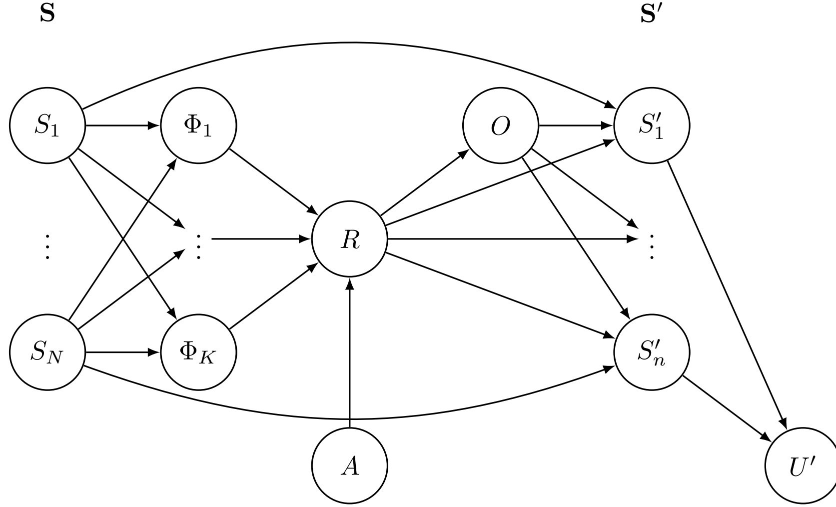

Figure 2:
The DBN $p(u', \mathbf{s}, o, r, \Phi | a, \mathbf{s})$ as described by [Lang-2010]

state τ_j where $\pi(u)$ returns the set of indices of the ground predicates that are relevant to reach the goal.

$$P(u' = 1 | \mathbf{s}') = I(\bigwedge_{j \in \pi(u')} s'_j = \tau_j) \tag{5}$$

The probability distribution for the next state s depends on the output of the chosen rule for the parts that change and on the current state s for the parts that are not changed by the rule. The probability distribution over the outputs is represented by an integer variable o that is the index of the output for a rule r and therefore identifies the output effects together with the rule r.

$$P(\mathbf{s}' | o, r, \mathbf{s}) = \prod_i P(s'_i | o, r, s_i) \tag{6}$$

where $P(s'_i | o, r, s_i)$ is a deterministic distribution: 1 if that ground atom is true in the outcome o and the rule r or if it is unchanged by it and 1 in s_i.
The probability of the outcome index o is part of the rule, namely $p_{r,o}$

$$P(O = o | r) = p_{r,o} \tag{7}$$

R is a random variable determining the rule that is being used for this transition and is 0 if no rule is being used. R can be deterministically determined by the contexts $\Phi_1, ..., \Phi_k$ together with the action a. A rule r is used iff it is defined for the action a and if it is the unique covering rule and therefore Φ_r is true while all other Φ'_r are not. R is 0 iff this holds for no rule r, therefore:

$$P(R = r | a, \Phi) = I(r \in \Gamma(a) \wedge \Phi_r = 1 \wedge \bigwedge_{r' \in \Gamma(a) \setminus \{r\}} \Phi_{r'} = 0) \tag{8}$$

$$P(R = 0 | a, \Phi) = \bigwedge_{r \in \Gamma(a)} \neg I(\Phi_r = 1 \wedge \bigwedge_{r' \in \Gamma(a) \setminus \{r\}} \Phi_{r'} = 0) \tag{9}$$

where r_i is the ith rule, and $\pi(\Phi_i)$ yields the indices of the ground predicates that appear in the context. The probability for a context to be true however, is the joint probability of all its atoms to

be true in the current state:

$$P(\mathbf{\Phi}|\mathbf{s}) = \prod_{i=1}^{K} P(\Phi_i|\mathbf{s}_{\pi(\Phi_i)}) = \prod_{i=1}^{K} I(\bigwedge_{j\in\pi(\Phi_i)} S_j = s_{r_i,j}) \tag{10}$$

where $\pi(\Phi_i)$ returns the indices of the atoms that are part of the context Φ_i Using these equations it is possible to build a DBN to represent the joint probability distribution based on the current state and the action to be taken as follows:

$$p(u', \mathbf{s}, o, r, \Phi|a, \mathbf{s}) = P(u'|\mathbf{s}')P(\mathbf{s}'|o, r, \mathbf{s})P(o|r)P(r|a\Phi)P(\Phi|\mathbf{s}) \tag{11}$$

3.3 Inference

As stated earlier, our goal is to infer $P(u^t = 1|a^{0:t-1}, \mathbf{s}^0)$, [Lang-2010] propose an approximate inference method in order to reach this goal. This probability distribution can be easily determined if we have the probability distribution of the current state $\mathbf{s}^t$

$$P(u^t = 1|a^{0:t-1}, \mathbf{s}^0) = \sum_{\mathbf{s}^t} P(u^t = 1|\mathbf{s}^t, \mathbf{s}^0)P(\mathbf{s}^t|a^{0:t-1}, \mathbf{s}^0) \tag{12}$$

To approximate the latter formula, [Lang-2010] assume conditional independence of states

$$\alpha(\mathbf{s}^t) := P(\mathbf{s}^t|a^{0:t-1}, \mathbf{s}^0) \approx \prod_{i=1}^{N} P(s_i^t|a^{0:t-1}, s_i^0) =: \alpha(s_i^t) \tag{13}$$

where N is the number of ground predicates. To calculate $\alpha(s_i^{t+1})$ we analyze the last action a^t and determine the rule that would be taken at that step. For each rule we can take the probability of that rule and multiply it with the distribution of states afterwards and since the rule r^t implies a^t this distribution is no longer dependant on it:

$$\alpha(s_i^{t+1}) = P(s_i^{t+1}|a^{0:t}, s^0) = \sum_{r^t} P(s_i^{t+1}|r^t, a^{0:t-1}, s^0)P(r^t|a^{0:t}, s^0) \tag{14}$$

To calculate the first formula in this equation we want to model the last state transition explicitly and then recursively use our α for the previous transitions:

$$P(s_i^{t+1}|r^t, a^{0:t-1}, \mathbf{s}^0) = \sum_{s_i^t} P(s_i^{t+1}|r^t, s_i^t)P(s_i^t|r^t, a^{0:t-1}, \mathbf{s}^0) \tag{15}$$

The second equation can be approximated by α if we assume conditional independence of s_i^t and r^t. This is not true as the rule r^t already tells us which contexts hold and this changes the probabilities of the ground predicates but it works as an approximation, therefore:

$$P(s_i^t|r^t, a^{0:t-1}, \mathbf{s}^0) \approx P(s_i^t|a^{0:t-1}, \mathbf{s}^0) = \alpha(s_i^t) \tag{16}$$

The interesting part is the first formula which represents the probability of one single state transition given a rule, this depends on the output o so we can introduce o:

$$P(s_i^{t+1}|r^t, s_i^t) = \sum_{i} P(s_i^{t+1}|o, r^t, s_i^t)P(o|r^t) \tag{17}$$

These probabilities are already given by equation 6 and equation 7. We still need to infer the second equation of equation 14 which represents the probability of the rule r^t given the actionsequence

$a^{0:t}$. The rule r^t is determined by the action a^t and the contexts Φ_r^t, we can introduce the contexts as before:

$$P(R^t = r|a^{0:t}, \mathbf{s^0}) = \sum_{\Phi^t} P(R^t = r, \Phi^t, a^{0:t}) P(\Phi^t|a^{0:t}, \mathbf{s}^0) \tag{18}$$

where $P(R^t = r|\Phi^t, a^{0:t}, \mathbf{s^0})$ is 1 iff $r \in \Gamma(a^t)$, therefore:

$$\sum_{\Phi^t} P(R^t = r, \Phi^t, a^{0:t}) P(\Phi^t|a^{0:t}, \mathbf{s}^0) = I(r \in \Gamma(a^t)) P(\Phi^t|a^{0:t}, \mathbf{s}^0) \tag{19}$$

where $P(\Phi^t|a^{0:t}, \mathbf{s}^0)$ being the probability that Φ_r holds and is the only context that does which means that r is the unique covering rule, this is done because if r isn't the unique covering rule we can't know what happens if we take that action and we can therefore make the assumption that we do not want to take that action. We can write this as:

$$P(\Phi^t|a^{0:t}, \mathbf{s}^0) = P(\Phi_r^t = 1, \bigwedge_{r' \in \Gamma(a^t) \setminus \{r\}} \Phi_{r'}^t = 0 | a^{0:t-1}) \tag{20}$$

$$= P(\Phi_r^t = 1|a^{0:t-1}) P(\bigwedge_{r' \in \Gamma(a^t) \setminus \{r\}} \Phi_{r'}^t = 0 | \Phi_r^t = 1, a^{0:t-1}, \mathbf{s}^0) \tag{21}$$

$a^{0:t-1}$ gets us to the state $\mathbf{s}^t$, we can model this explicitly and then approximate using 13

$$P(\Phi_r^t = 1|a^{0:t-1}, \mathbf{s}^0) = \sum_{s^t} P(\Phi_r^t = 1|s^t)\alpha(\mathbf{s}^t) \approx \sum_{\mathbf{s}^t} P(\Phi_r^t = 1|\mathbf{s}^t) \prod_j \alpha(s_j^t) \tag{22}$$

where the former probability can be calculated using equation 10. To calculate the probability that every other context is false [Lang-2010] first make the assumption of statistical independence for all contexts except Φ_r

$$P(\bigwedge_{r' \in \Gamma(a^t) \setminus \{r\}} \Phi_{r'}^t = 0 | \Phi_r^t = 1, a^{0:t-1}, \mathbf{s}^0) \approx \prod_{r' \in \Gamma(a^t) \setminus \{r\}} P(\Phi_{r'}^t = 0 | \Phi_r^t = 1, a^{0:t-1}, \mathbf{s}^0) \tag{23}$$

and then model $\mathbf{s}^t$ explicitly again and approximate the probability, it is 1 if the contexts contradicts Φ_r as it can't be true then and otherwise can be approximated by using the probability that the other contexts are fulfilled as well while ignoring the ground predicates that already have to be set to fulfill Φ_r

$$P(\Phi_{r'}^t = 0 | \Phi_r^t = 1, a^{0:t-1}, \mathbf{s}^0) \approx \begin{cases} 1.0 & \Phi_r \Phi_{r'} \to impossible \\ 1.0 - \prod_{i \in \pi(\Phi_{r'}^t), \neg i \in \pi(\Phi_r^t)} \alpha(s_i^t = s_{r',i}) & otherwise \end{cases} \tag{24}$$

This allows us to compute the probability distribution for the reward given the previous action sequence and enables us to do the next step.

3.4 Planning

To calculate the value of an action sequence we are not only interested if the action sequence leads to a state that fulfills the goal but also if any prefix of the action sequence already reaches the goal. Since PRADA only uses action sequences of the same length T we can calculate that by a summation over all prefixes. PRADA prefers immediate rewards by adding a discount factor $0 < \gamma < 1$

$$Q(a^{0:T-1}, \mathbf{s}^0) := \sum_{t=1}^{T} \gamma^t P(u^t = 1|a^{0:t-1}, \mathbf{s}^0) \tag{25}$$

To do the actual planning PRADA samples a fixed amount of action sequences and calculates their values. Sampling continues as long as none of the action sequences exceeds a threshold ζ, otherwise the algorithm carries out either the whole action sequence which maximizes Q or the first n actions of that action sequence and then starts over if the goal is not yet reached. The actual sampling trys to choose only actions that have a unique covering rule as PRADA cannot do useful planning otherwise. Therefore PRADA samples an action a^t according to the probability that such a unique covering rule exists given the previous actions $a^{0:t-1}$:

$$P^t_{sample}(a) \propto \sum_{r \in \Gamma(a)} P(\Phi^t_r = 1, \bigwedge_{r' \in \Gamma(a) \setminus \{r\}} \Phi^t_{r'} = 0 | a^{0:t-1}) \tag{26}$$

[Lang-2010] also propose an extension, Adaptive-PRADA, which checks if deleting particular actions from the found sequence improves the Q-Value in order to delete useless actions.

4 Alternative Approaches

4.1 Reasoning in the abstract domain: ReBel

PRADA does its reasoning in the grounded domain, alternatively one could also plan in the lifted domain. Planning in the grounded domain is easier, the way PRADA handles noise, for example, wouldn't be possible if we wouldn't know what states exist. Planning in the lifted domain however, enables generalization not only for the learned rules but also for the planning itself. [Kerstings-2004] propose ReBel, a planner for probabilistic STRIPS operators which reasons in the lifted domain by defining the Bellman Update Operator for relational models to solve RMDPs with value iteration. The goal is therefore to define a value function V_{t+1} that assigns each state a value based on an abstract reward function R which assigns each state its immediate reward and the previous value function V_t (a useful choice for V_0 is R). To define such a value function ReBel performs the following steps:

1. For each state S' find each possible preceding state S by looking at each rule and possible outcome of that rule. ReBel then checks for each subset of S' if it could be the result of the rule outcome, if so ReBel applies the reverse of the rule to that subset and adds it and all its simplifications to S if it is a valid and consistent rule. This results in state, action-outcome pairs

2. Compute the Q-function which assigns a value to each state-action pair, this is done by assigning values to each state-action outcome (A, S) pair first: if S is absorbing then the q-value equals the reward function $R(S)$, otherwise it is the reward of the state plus the discounted value of the next state weighted with its probability where the value is calculated with the previous value-function: $R(S) + p_i * \gamma * V_t(S')$. This yields a set of q-values for each outcome, ReBel then tries to combine them if the outcomes are compatible. That is if it is possible to be in both states at the same time using the same action, then the algorithm adds a new q value for the greatest lower bound of the 2 states that is the addition of the two values.

3. Define the new value function V_{t+1} as the function that maximizes the resulting Q-function w.r.t. a

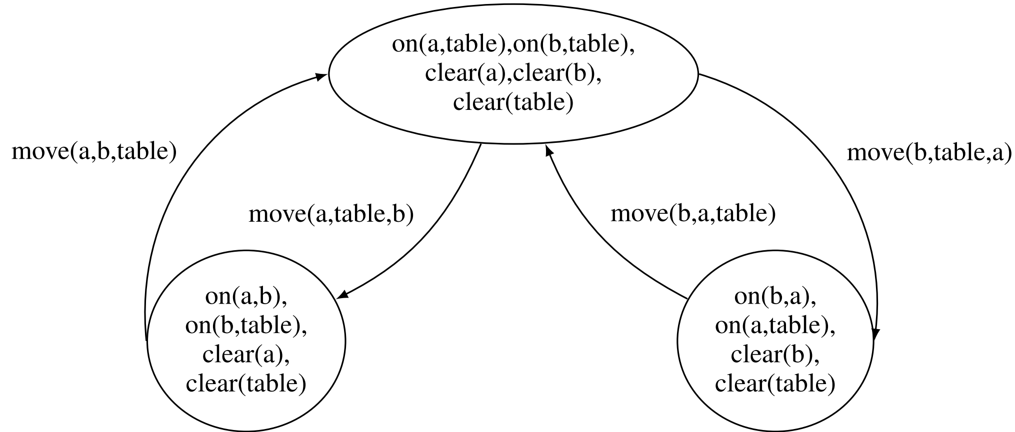

Figure 3: state-graph for a blocks world scenario with 2 blocks and one deterministic move-action as in equation 1

4.2 Reusing deterministic planners: FF-Replan

So far we have concentrated on planners that try to solve MDPs and therefore focus on finding functions that assign values to each state; deterministic planners typically take a different approach and can be adapted for nondeterministic domains as well. In the deterministic world, planning is often considered to be a search where the search-space is made up of the different states (see figure 3) and by taking actions we can navigate through this space. Planning then corresponds to finding the optimal path from one state (the current state) to another.

A forward search algorithm is a planning-algorithm that begins its search at the current state and then searches for a state that fulfills the goal. The state space can and will usually be very large therefore a good search heuristic is a necessity.

The Fast-Forward (FF) algorithm presented in [Hoffmann-2001] is a forward search algorithm that introduces a variation of Hill Climbing search and a heuristic that considers only the atom that an action adds to the current space but not the atoms that it removes. The heuristic works by building a planning graph that consists of n fact and action layers. The first fact layer consists of all facts that are part of the state S that is being evaluated by the heuristic and the first action layer, like all action layers, consists of the actions that are applicable to the corresponding state layer. The rest is then constructed iteratively, each fact layer contains all facts from the previous layer and adds all facts that are added by applying each action of the previous layer until a fact layer fulfills all goals. The algorithm then constructs actions-sets O_i and fulfilled goal sets G_i (G_n being our real goal) for each layer by iterating backwards over the constructed planning graph and looking at the goals in G_i: If they are already fulfilled in the $i-1$th layer, then add them to G_{i-1}, otherwise, for each missing fact, take an action that would introduce the missing fact and add it to O_{i-1} and its precondition to G_{i-1} to consider it a fulfilled goal by now. The heuristical value for S is then the amount of actions in each action set O_i.

Having a heuristic, [Hoffmann-2001] then introduce the Enforced Hill Climbing algorithm which is a search algorithm that starts at the current state and then uses breadth first search to find a state that has a better heuristical value then the current state. This is then repeated until a state is found that fulfills the goal.

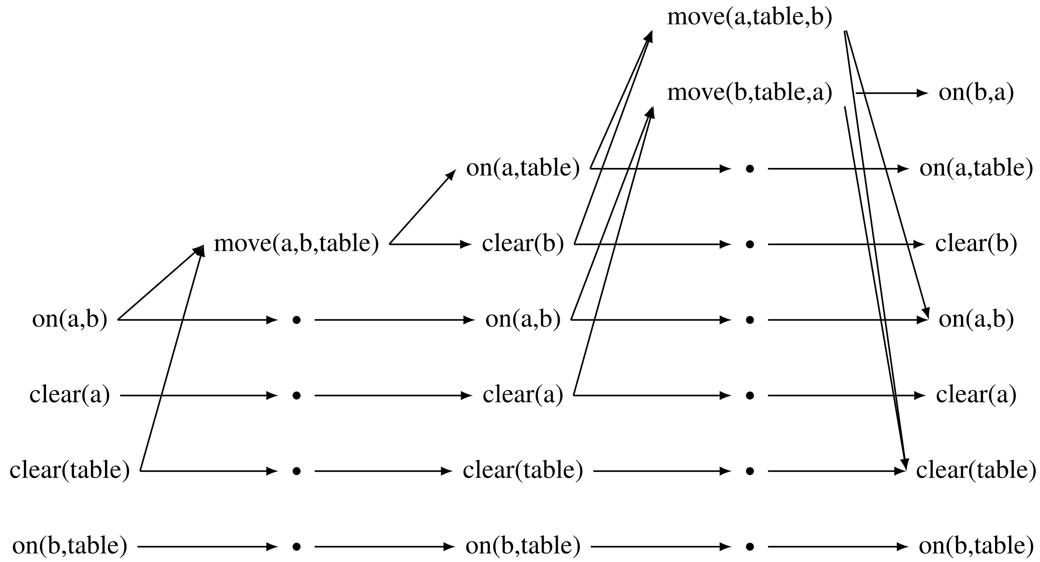

Figure 4: planning graph for the same blocks world domain as in figure 3, starting with a stacked on b with the goal $\tau : on(b, a)$ to stack b on a

The FF-Planner requires a deterministic domain, [Yoon-2007] propose FF-Replan which is a probabilistic planner that create a deterministic domain out of a probabilistic one and then calls FF to retrieve a plan. They propose 2 different kinds of determinization:

1. Single-outcome determinization which creates a deterministic action for each probabilistic action and uses the outcome that has the highest probability while ignoring all other outcomes and
2. All-outcomes determinization which creates multiple deterministic actions for each probabilistic action, one for each possible outcome. This takes all outcomes into account but ignores their probabilities.

Both methods have drastic drawbacks but can work well in many domains.

4.3 Learning the policy directly from state-action pairs: RRL

All previously mentioned planning method were model-based, meaning that they operated on a given, i.e. separately learned or handcrafted, and complete rule set. [Dzeroski-2001] introduce the Relational Reinforcement Learning (RRL) algorithm which is a model-free reinforcement learning approach for relational domains and is based on Q-learning. The algorithm needs a reward function R and a set of actions and their preconditions. In order to learn a Q function, it initializes the Q-function so that it assigns 0 to each state-action pair and then iterates: First it obtains new examples using the current Q function and some exploration strategy, e.g. an action-sampling proportional to the current Q values. These states are then assigned $\hat{q}$ estimates which are calculated by taking the reward given by R and adding the maximum value of the current Q function for all possible further actions.

$$\hat{q}_j(s_j) = r_j + \gamma max_a \hat{Q}_e(s_{j+1}, a) \tag{27}$$

The new Q function is then created using these state-action-$\hat{q}$ tuples and a relational regression algorithm of which several have been proposed:

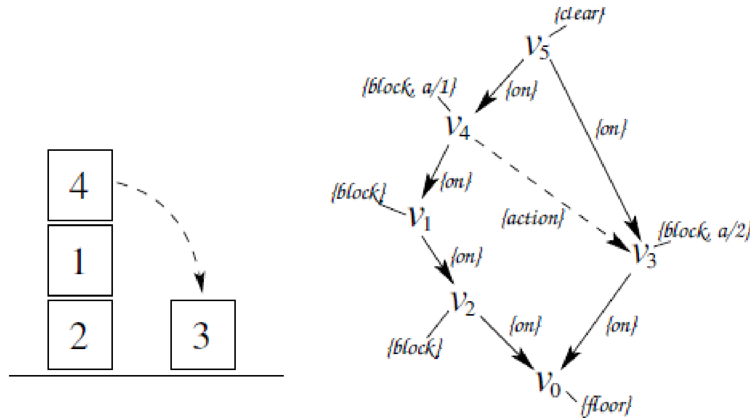

Figure 5:
Example graph (right) representing a state-action pair (left) in the blocks world. Taken from [**?**]

- [Dzeroski-2001] propose the use of TILDE-RT ([Blockeel-98]) which is a relational extension of the C4.5 decision tree algorithm.
- [Driessens-2003] propose the use of instance based learning for relational rules. They use a k-nearest-neighbour approach and therefore require a given distance metric to predict the q values by averaging over its neighbours weighted by their distance

$$\hat{q}_i = \frac{\sum_j \frac{q_j}{dist_{i,j}}}{\sum_j \frac{1}{dist_{i,j}}} \tag{28}$$

  They further propose optimizations of the instance set based on the ideas of the IB2 and IB3 algorithms ([Aha-1991]), these optimization consist of rules to discard new examples if they do not add enough new information and to throw away examples if they have a large contribution to errors that are being made, consult [Driessens-2003] for details.

- [Driessens-2006] propose to represent the state-action pairs as graphs and use gaussian process regression ([MacKay-1998]) with graph kernels to approximate the Q function. Gaussian process regression is an effective bayesian regression method that does not require to evaluate the input data points (state-action pairs) but only the result of a positive definite kernel function $k(x, x')$ where x and x' are state-action pairs. The definition of a kernel for state-action pairs suffices to use GPR as regression method. [Driessens-2006] propose to use graph kernels, which requires the conversion of the state-action pairs into a labeled directed graph. The method to accomplish this can be tailored to the domain and could for example model objects as vertices, unary relations as labels for these vertices, binary relations as labeled edges between nodes and the action akin to the relations.

  Having a graph [Driessens-2006] then propose to use product graph kernels using direct product graphs. A direct product graph of 2 graphs G and G' contains a node for each pair of nodes (v, v') where v is a node in G and v' is a node in G' and where both nodes have the same labels. Edges are inserted between two vertices (v, v') and (u, u') if there was an edge between v and u and one between v' and u' with the same labels.

The product graph kernel then counts the number of walks weighted by a factor based on their length and is defined as

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} [\sum_{n=0}^{\infty} \lambda_n E_\times^n]_{i,j} \tag{29}$$

Where G_1 and G_2 are the state-action graphs, $|V_\times|$ is the number of vertices in their product graph, λ_n is the weight factor for the length n and E_x is the adjacency-matrix of the product graph where each entry at line i and column j corresponds to the number of edges between the vertex with index i and the vertex with index j.

5 Conclusion & Evaluation

This paper introduced a number of relational planning algorithms including FF-Replan which has shown good results in the international probabilistic planning competition (IPPC) 2004 and 2006. However, since this is achieved without using all outcomes or without taking the actual probabilities into account [Yoon-2007] conclude that artificial domains such as those used in the IPPC-04 and 06 may not make full use of the difficulty that can be introduced by probabilistic actions. This brings up the importance of using more realistic domains for evaluation which has been done by [Pasula-2007] by implementing the blocks world in a realistic physics simulation with blocks of different sizes and spheres. They have shown results of a simple planning algorithm using rules learned with their rule learning algorithm that come near those of a human controlling the robot in the physics simulation at stacking objects to a high tower. [Lang-2010] show that PRADA surpasses the aforementioned planner as well as FF-Replan in the same task and showed promising results with other tasks that were more complicated than the simple planner was able to handle.

These results show that planners that work on relational rules can be useful in realistic domains which raises the question on how to utilize them for autonomous agents. [Lang-2012] therefore introduce *Rex*, a reinforcement learning algorithm for relational world models that makes use of NID-Rule-learning and PRADA planning and proposes an exploration strategy that improves upon E^3 ([Kearns-1998]) and R-Max ([Brafman-2001]) by exploiting the structure of relational models. Both E^3 and R-Max are based on a count-function $\kappa : S \rightarrow \mathbb{N}$ which returns the amount of times that a state has been visited and is not suitable for relational world models due to the large size of the state space. [Lang-2012] propose to group states by queries such as the contexts Φ that were learned by the NID-Rulelearner thus κ would return the amount of states that have been already visited which share at least one context with the given state. They also propose to measure the similarity to the contexts using graph kernels as introduced by [Driessens-2006] instead of hard counts. For evaluation they use the same simulated domain as for PRADA and show success for tasks such as stacking movable objects of the same color.

Planning algorithms for relational rules have shown good results and complete reinforcement learning algorithms have already been proposed. While there is lots of room for improvement it appears to be a promising approach for acting in the real world.

References

[Aha-1991] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

[Blockeel-98] H. Blockeel and Luc de Raedt. Top-down induction of first order logical decision trees. In *Artificial Intelligence*, volume 101, pages 285–297, 1998.

[Boutilier-2001] Craig Boutilier, Ray Reiter, and Bob Price. Symbolic dynamic programminig for first-order mdps. In *Proceedings of the Seventeenth Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 1211–1217, 2001.

[Brafman-2001] Ronen I. Brafman, Moshe Tennenholtz, and Pack Kaelbling. R-max - a general polynomial time algorithm for near-optimal reinforcement learning, 2001.

[Driessens-2003] Kurt Driessens and Jan Ramon. Relational instance based regression for relational reinforcement learning. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 123–130, 2003.

[Driessens-2006] Kurt Drieesens, Jan Ramon, and Thomas Gärtner. Graph kernels and gaussian processes for relational reinforcement learning. *Machine Learning*, pages 146–163, 2006.

[Dzeroski-2001] Sage Dzeroski, Luc de Raedt, and Kurt Driessens. Relational reinforcement learning. In *Machine Learning*, volume 43, pages 7–52, 2001.

[Guestrin-2003] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored mdps. *Journal for Atificial Intelligence Research*, 19:399–468, 2003.

[Hoffmann-2001] Joerg Hoffmann. Ff: The fast-forward planning system. *AI Magazine*, 22:57–62, 1 2001.

[Kearns-1998] Michael Kearns. Near-optimal reinforcement learning in polynomial time. In *Machine Learning*, pages 260–268. Morgan Kaufmann, 1998.

[Kerstings-2004] Kristian Kersting, Martijn Van Otterlo, and Luc De Raedt. Bellman goes relational. In Carla E. Brodley, editor, *Proceedings of the Twenty-First International Conference on Machine Learning*, pages 465–472, 2004.

[Lang-2010] Tobias Lang and Marc Toussaint. Planning with noisy probabilistic relational rules. *Journal of Artificial Intelligence Research (2010), Pages 1-49*, 2010.

[Lang-2012] Tobias Lang, Marc Toussaint, and Kristian Kersting. Exploration in relational domains for model–based reinforcement learning. *Journal of Machine Learning Research (JMLR)*, 13, 2012.

[MacKay-1998] D. J. C. MacKay. Introduction to Gaussian processes. In C. M. Bishop, editor, *Neural Networks and Machine Learning*, NATO ASI Series, pages 133–166. Kluwer Academic Press, 1998.

[Pasula-2007] Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29, 2007.

[Yoon-2007] Sungwork Yoon, Alan Fern, and Robert Givan. Ff-replan: A baseline for probabilistic planning. In *ICAPS07*, pages 352–359, 2007.

[vanOtterlo-2009] Martijn van Otterlo. *The Logic of Adaptive Behavior*. IOSPress, 2009.