Relational reinforcement learning with non-binary symbols

Arbeit zur Erlangung des Grades
eines Bachelor of Science
am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

von

Andreas Henne

Berlin
5. Juli 2012

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur mit Hilfe der angegebenen Quellen verfasst habe.

Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Andreas Henne
Berlin, den 5. Juli 2012

Freie Universität Berlin
Fachbereich Mathematik und Informatik

Institut für Informatik
Machine Learning and Robotics Lab
Prof. Dr. Marc Toussaint

Relational reinforcement learning with non-binary symbols

Bachelorarbeit

Autor Andreas Henne (Matr.-Nr: 4356288)

Eingereicht im Juli 2012

Erstgutachter Dr. Tobias Lang
Zweitgutachter Prof. Dr. Marc Toussaint

Summary

This thesis addresses the problem of goal-directed behavior in stochastic relational worlds. These worlds are abstract models of natural environments and represent objects, their properties and their relationships to each other explicitly. Typically, relational state representations use binary symbols which can either be true or false in a given situation. Many object properties such as the size or the height of an object can not be represented accurately with binary predicates. Instead, they require symbols with non-binary ranges, such as integers or values from a fixed set. However, existing approaches to goal-directed behavior in relational reinforcement learning focus on binary symbols. This thesis investigates how symbolic relational transition models which use changing non-binary symbols can be represented, learned and used for planning towards high reward states. An example scenario is presented in which a physically simulated robot has to learn how to interact with a weighing machine using cubes of different sizes.

Contents

Chapter 1

Introduction

A basic requirement for autonomous agents is the ability to select actions in order to reach goals. For many domains like playing chess it is possible to build agents that accomplish this task very effectively, often outmatching human intelligence. However these agents are usually limited to one specific domain and one specific goal. An autonomous agent must be able to reach various changing goals in many different scenarios. Building agents that perform goal-directed behavior in unknown environments is still a largely unsolved problem. These agents must *explore* the environment, *learn* from observations and *plan* towards goals, applying the learned knowledge.

The agent is given the ability to perceive its environment, building an internal model of the world. A lot of recent research has focused on approaches that represent the world on a relational level. These relational worlds consist of objects, their properties and their relationships to each other. The agent can interact with the environment by performing actions. While the execution of a single action such as moving an arm is predetermined the effects of these actions are unkown to the agent and must be learned. The example environment used for all tests in this thesis is a simulated robot manipulation domain in which the agent interacts with cubes, balls and other objects. The simulation has the advantage that the task of perception can be solved easily. The following is a possible relational representation of the scene shown in Figure 1.1:

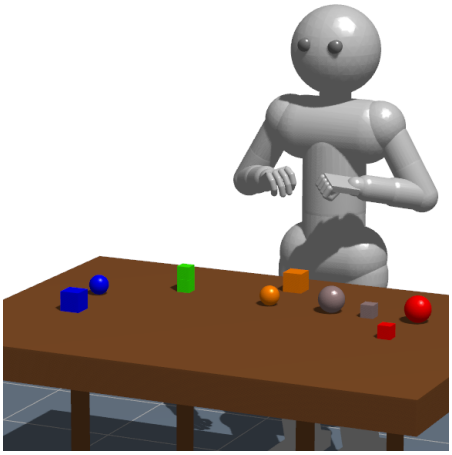


Figure 1.1: A simple scene in the 3D simulated robot manipulation domain

$table(a), cube(b), cube(c)$, ... $sphere(g), sphere(h), sphere(i), inhand(b)$,
$on(b, a)$, $on(c, a)$, $on(d, a)$, $on(f, a)$, ... $size(a)$=4, $size(b)$=2, $size(c)$=1, ...

It states which object exist and provides some information about them: For example, the symbol on(b, a) is used to indicate that the block with the identifier b stands on the table a. The non-binary symbol *size* assigns a number (in this case from $1 - 5$) to objects, providing information about their size. A possible task for the agent in this simple scenario is to build a high tower of objects. Initially, the agent knows nothing about the consequences of actions. For example the agent must learn by exploring that grabbing an object o causes $inhand(o)$ to be true. However grabbing an object that is below another object causes the upper object to fall down. The process of exploring an unknown relational environment, learning from interactions without a human supervisor and planning towards goals is a form of *relational reinforcement learning* (Tadepalli et al., 2004). Pasula et al. (2007) presented an effective algorithm that learns a set of relational probabilistic rules from gathered experiences. Lang (2011) developed a planning algorithm PRADA that can exploit this learned knowledge to plan towards high reward states. The task of building high towers is one of the exemplary scenarios that were demonstrated using these learning and planning algorithms. One of the goals of this thesis is to investigate how good this existing relational reinforcement learning approach performs in other scenarios. This is worth investigating because autonomous agents should be able to perform goal-directed behavior in all kinds of scenarios. Testing different scenarios can help to identify limits and problems of the approach or demonstrate its powerfulness.

Especially in more complex scenarios it is important that the agent can deal with non-binary object properties that may change as a result to its actions. While existing research in the field of relational reinforcement learning considered these scenarios in theory, empirical tests focussed on binary symbols. Changing non-binary symbols do not seem useful in the case of *size*, because a solid object seldom changes its size. But there are many thinkable scenarios in which a mechanism to express *non-binary change* in the world is important. One example are scenarios that include resources that are consumed or refilled by certain actions. An example for such a resource is money that can be spent or received. Obviously there are many scenarios where the agent needs to reason about money in some way, for example a household roboter that goes shopping. Other possible applications are spatial properties similar to *size* such as distances between objects or how they are oriented to each other. Analogous to *size* a non-binary symbol could be used to encode at least a rough approximation (for example on a scale from 1 to 10) of

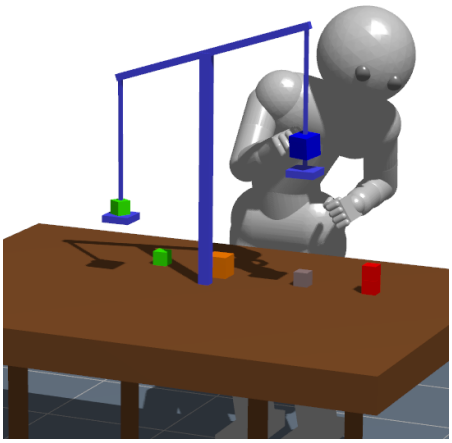


Figure 1.2: The agent interacts with a weighing machine

how far away an object is from another. Figure 1.2 shows a related scenario inside the simulated robot manipulation domain that was implemented and tested as part of the thesis. Here, a non-binary symbol is used to represent the balance of a simple weighing machine which is determined by the positions of the two scale pans.

The main goal of this thesis is to enable learning and planning in scenarios that highly depend on non-binary symbols. While the planning algorithm PRADA is slightly extended emphasis lies on the learning algorithm by Pasula et al. (2007). Most notably we add the ability to learn how the value of a non-binary symbol changes relative to the precedessor state. Using the developed extensions goal-directed behavior is achieved in scenarios that were not covered by the existing techniques.

Chapter 2

Background

2.1 Models of the environment

An agent that shall perform goal-directed behavior in a complex environment needs an internal model of it. Consider the robot manipulation domain, where every object has a lot of properties that are defined using floating point numbers: A position, a shape, a rotation and so on. This leads to extremely many possible world states and makes it very hard to model interactions with the environment. The internal model is a simplified abstraction from the environment, potentially discarding lots of information. It determines how world states are represented and models interactions with the environment, predicting how the world state representation changes as a result of actions.

2.1.1 State representation

A state is a (possibly simplified) description of the environment at a specific point in time. The state representation must provide all information the agent needs to plan towards its goal. Choosing the right level of abstraction is crucial for the agent's performance. One possibility to describe states of the environment are relational symbolic representations (Pasula et al., 2007), (Lang, 2011). Relational logic symbols allow to describe objects, their properties and their relations to each other. Several kinds of symbols are used. Binary symbols such as *cube* or *on* are always either true or false. They characterize binary properties of objects and indicate whether objects are in a certain relation like *on* to each other or not. For example $cube(X)$ states that the object X is a cube and $on(X, Y)$ indicates that the object X stands on the object Y. Non-binary symbols are typically integer functions such as $size$ and can be used to represent non-binary object properties. These non-binary symbols allow a more accurate approximation of the environment in many cases. Non-binary symbols may also map to a finite and fixed non-binary set which is an important distinction with respect to learning and planning algorithms.

Binary or non-binary symbols may be *primitive* or *derived*. While primitive symbols can not be expressed using other existing symbols, derived symbols (also called concept

definitions) are implied by primitive or other derived symbols and can be defined via formulas. One example is the relation $above(X, Y)$, which is true when an object X stands on an object Z that is either Y or for that $above(Z, Y)$ is true. This can be expressed with the following formula:

$$above(X, Y) = on(X, Y) \vee \exists Z : (on(X, Z) \wedge (above(Z, Y))$$

Symbols that are applied to world objects such as $cube(X)$ or $size(X)$ are called atoms. Formally, a state is described by a conjunction of atoms. The agent is given the ability to translate the current state of the environment into the symbolic relational representation given a vocabulary of symbols. It is assumed that the perception is perfect and the agent always sees the complete environment. Of course this is a highly idealistic assumption and certainly not suitable for huge and complex worlds, but it allows to focus on the problems of planning and learning. Note that while the model is complete (the agent perceives all objects and specified relationships and properties) and correct it is still an abstraction from the environment. For example the symbol *size* is obviously a great simplification of the real object shapes. In reality, we do not perceive the world directly in terms of objects and relationships directly but instead see colors, hear noises and feel resistance. However we certainly often think in terms of objects and their relationships to each other when performing goal-directed behavior, such as opening a door or planning the next move in a chess game. This thesis is about how learning and planning can be achieved given such a symbolic state representation, assuming that the task of constructing such a relational state representation is solved somehow.

2.1.2 Markov decision process

The model needs to specify how actions performed by the agent affect the state of the environment. In some cases a deterministic model is sufficient, but especially when using a high level of abstraction in a complex environment, actions often have uncertain effects. Planning under uncertainty can be modeled with Markov decision processes (MDPs) (Boutilier and Thomas Dean, 1999). MDPs can be used with all kinds of state representations, including the presented above. Let A be the finite set of possible actions and S the finite set of modeled states. At each time step the agent has to choose an action $a \in A$ with respect to the current world state $s \in S$. $P(s'|a, s)$ determines the probability that choosing the action a in the state s will lead to the successor state s'. These state transition distributions are specified by the transition model T. They possess the Markov property, which states that given s and a the transition distribution $P(s'|a, s)$ is independent from all past and future states and actions. The Markov property can be exploited for efficient inference methods.

A reward function $R : S \to \mathbb{R}$ assigns a reward to each state s. The reward function can be used to formulate any desired goal within the chosen level of abstraction. For example, if the goal is to reach one specific state, a reward function is suitable that maps all states to zero except the desired state, which is mapped to an arbitrary positive real number. Another example are goals where a certain state property shall be maximized

or minimized. In this case a reward function is chosen that increases proportionally with the desired property. Consider the task of building high towers in the robot manipulation domain. A suitable reward function may retrieve the height of the highest tower or the average height of all towers.

2.2 Planning

In the context of a Markov decision process, *planning* is the task of finding a sequence of actions that lead to states with high rewards, avoiding states with low rewards. More formally, it is the problem of choosing a policy function $\pi : S \to A$ which tells the agent which action it should choose in each state. A value function $V^\pi : S \to \mathbb{R}$ retrieves the expected sum of rewards when following the policy π from state s:

$$V^\pi(s) = E[r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + ...|s_0 = s; \pi]$$

$0 < \gamma < 1$ is the discount factor (typically close to 1) which ensures that near-term rewards are weighted higher than long-term rewards. The value function can also be written down as a recursive function:

$$V^\pi(s) = R(s) + \gamma \sum_{s'} P(s'|\pi(s), s) V^\pi(s')$$

The agent needs to choose the optimal policy function π^* which maximizes the value function. π^* is defined by the Bellman equation:

$$V^{\pi^*}(s) = R(s) + \gamma \max_{a \in A} [\sum_{s'} P(s'|s, a) V^{\pi^*}(s')]$$

Many algorithms exist that compute the optimal policy for a MDP, for example using linear programming or dynamic programming. Unfortunately, the time complexity of these algorithms depend on the number of possible states. When using a relational state representation, the number of possible states increases exponentially with the the number of world objects: Even with only one unary predicate and n objects the number of possible states is 2^n. This makes practical planning in interesting domains with many objects, predicates and functions difficult, which is why efficient approximation algorithms are actively researched by AI scientists.

In this thesis, we use an efficient planning algorithm for relational representatoins called PRADA (Lang and Toussaint, 2010). PRADA was never tested with changing primitive non-binary symbols before. We extend it slightly to deal with changing non-binary symbols.

2.3 Reinforcement learning

Reinforcement learning is the task of learning how to act in an unknown environment by interactions with it. Russel and Norvig (1996) adduces the example of an agent that

shall learn how to play chess. One possibility is that a skilled chess player provides a lot of game situations and what move he would choose in the specific situation. This technique is called *supervised learning*. In a reinforcement learning scenario, the agent has to play on its own. Without any knowledge how to play chess, it performs random moves and observes the consequences: Did it capture an opponent's piece? How does the opponent react? And, in the end, does it win or loose the game? Received rewards as results of actions like these are called *reinforcements*. After receiving a reinforcement, the agent does not know which individual actions were responsible for it. Without a human teacher providing information which actions are good or bad, it must distribute the received reinforcements over the various actions it has taken. Using this learned knowledge, the agent should be able to plan towards high reward states - in this case winning a chess game.

The problem of reinforcement learning can be formulated as a Markov decision process where the transition model T is unknown and must be learned from interactions with the environment. More formally, these interactions can be regarded as a set of experiences $\{(s, a, s')\}$. Each experience is a triplet (s, a, s') where s is the the state observed before performing the action a and s' the observed successor state. In the stated example, the goal of winning the game is always the same: The reward function for the MDP is fixed. *Model-free* reinforcement learning may be applied, in which the transition model is not learned explicitly. Instead, model-free reinforcement learning algorithms only try to learn ation-state values (Strehl et al., 2006). In contrast, *model-based* reinforcement learning techniques learn an approximation of the transition model T from interactions with the environment. The learned knowledge about the effects of actions in certain contexts can be used to plan towards various goals. In the chess example, the agent could be given the task to capture all opponent's towers instead of winning the game. This thesis focuses on model-based relational reinforcement learning, which is model-based reinforcement learning with relational state and action representations. The next section is about how transition models for this task can be represented and learned.

2.4 Transition models

A transition model for a Markov decision process specifies the transition distribution $P(s'|s, a)$ for every possible state and action. In relational worlds there are usually too many possible states to store the transition distribution for each state explicitly. Probabilistic relational rules can be used to encode the transition model by abstracting from concrete world objects.

2.4.1 NID rules

Probabilistic relational rule describe how actions performed by the agent affect the state of the world under the assumption of a certain context. They are probabilistic because they may specify several possible outcomes with a probability assigned to each of them.

Probabilistic relational rule abstract from specific objects and instead refer to variables. Noisy indeterministic deictic (NID) rules (Pasula et al., 2007) are probabilistic rules that allow to model outcomes as noise and the usage of *deictic references*. The following is an example for a NID rule:

$$puton(X) : table(Z), ball(X), inhand(Y), size(Y) = 1 \rightarrow \begin{cases} 0.9 : on(YZ), \neg inhand(Y) \\ 0.05 : on(Y, X), \neg inhand(Y) \\ 0.05 : noise \end{cases}$$

It predicts that if the agent puts the object Y (which must be in its hand) with the size 1 on the object X which is a ball, the following will happen:

- With a probability of 90%, the object Y will fall down on the table Z.
- With a probability of 5% the object Y will stand on the ball X.
- With a probability of 5% anything can happen. For example, the object Y may fall down causing another stack of objects to collapse.

The variables Y and Z are not arguments of the action, but they are useful to predict the outcome. They are called *deictic references*. A rule only covers a state if if there is one unique substitution for all deictic references. In this case, the rule is only applicable for cases in which there is only one table and the agent only has one object in its hand. Formally, a NID rule is given as (Pasula et al., 2007):

$$a_r(X) : \Phi_r(X) \rightarrow \begin{cases} p_{r,1} : \Omega_{r,1}(X) \\ p_{r,m_r} : \Omega_{r,m_r}(X) \\ p_{r,0} : \Omega_{r,0}(X) \end{cases}$$

a_r is the performed action and X is a set of logical variables that are used in the rule. The context $\Phi(X)_r$ and the outcomes $\Omega_{r,i}(X)$ are conjunctions of literals, specifying predicates and function values for the logical variables in X. The context may also contain inequality literals such as $f(X) > 5$. $p_{r,i}$ is the probability for the outcome $\Omega_{r,i}$, with $\sum_{i=0} p_{r,i} = 0$. $\Omega_{r,0} > 0$ is the noise outcome, which is used to model rare and complex behavior that is not covered by the other outcomes and can not be modeled with the variables in X. The noise outcome probability is distributed over all possible states.

A *grounded* rule is a rule in which all arguments are constants. An abstract rule that contains variables can be grounded by applying a substitution that maps all variables to concrete world objects. A rule with the context $\Phi_r(X)$ and the action $a_r(X)$ *covers* a state s and an action a if there is a substitution σ for all variables in X so that the grounded rule context is entailed by s and the grounded rule action matches a. For all state/action tuples (s, a) that are covered, the rule encodes the distribution $P(s'|s, a)$ over all possible successor states.

$$P(s'|s, a) = \sum_{i=1}^{m_r} p_{r,i} P(s'|\Omega_{r,i}, s) + p_{r,0} P(s'|\Omega_{r,0}, s)$$

$P(s'|\Omega_{r,i}, s)$ is 1 if the application of the outcome $\Omega_{r,i}$ to the state s leads to the successor state s', otherwise it is 0. $P(s'|\Omega_{r,0}, s)$ is the distribution of the noise outcome over all states. It can be approximated with a fixed value p_{min} (Pasula et al., 2007).

The transition model T in a Markov decision process must specify the transition distribution $P(s'|a, s)$ for every possible state s. To encode a transition model with a set of probabilistic relational rules, it is required that every possible state is covered by one of the rules. Especially when learning rules this is typically not the case. A default rule with an empty context can be used that covers all states but is only applied when no other rule covers by convention. The default rule specifies two outcomes with probabilities assigned to each of them: A noise outcome and a no-change outcome.

One limitation of these NID rules is that the outcomes can not refer to the values of the context literals and always specify absolute values. For example it is not possible to encode that a non-binary symbol increases or decreases by a value. In chapter 3 we will discuss how relative change can be represented with NID rules.

2.4.2 Learning a transition model

A rule learning algorithm solves the problem of learning a transition model from experiences. It takes a set of observed training examples $\{(s, a, s')\}$ and retrieves a set of probabilistic relational rules that best explain the observations. A good rule set should contain rules that predict the observed outcomes of the training examples with a high probability, but it also should not be overly complex and generalize over the observed experiences. Pasula et al. (2007) proposed a scoring metric that assigns a score to a rule set Γ for a set of observed experiences E that correponds to these criteria:

$$S(\Gamma) = \sum_{(s,a,s') \in E} \log P(s'|s, r_{(s,a)}) - \alpha \sum_{r \in \Gamma} PEN(r)$$

$r_{(s,a)}$ is the rule with the action a that covers the state s. $P(s'|s, r_{(s,a)})$ is the outcome probability stated by the rule, that with the covering argument substitution the action will lead to the observed successor state s'. Note that the scoring metric assumes that each observed state/action tuple (s, a) is covered by one rule only.

$PEN(r)$ retrieves the number of literals in the context of r. The term $\alpha \sum_{r \in \Gamma} PEN(r)$ is the complexity penalty. It is higher for more specialized rules that cover a very specific context, specified by many literals. The penalty ensures that there is *generalization* over the given training data, which is crucial for learning. If α is set to a low value (for example 0.1), the optimal rule set will contain rules that accurately cover the observed experiences, typically with deterministic rules (meaning there is only one outcome). If α is set to a higher value (for example 10), the rule set will likely contain fewer and less complex rules, predicting the training experiences less accurately. Instead of constructing accurate rules for every situation seldom experiences outcomes will be modeled as noise. With $\alpha = 0$, the algorithm will always favor accuracy over simplity and every training experience will be covered by a rule. In some cases this may lead to one rule for each training experience, completely lacking generalization.

In general, the problem of computing a set of NID rules with a maximum score for a given set of training examples is NP-hard (Walsh, 2010). Pasula et al. (2007) proposed a greedy algorithm, that offers good results for many cases. The algorithm starts with a rule set that only contains the default rule:

$$default() : \text{ (empty context) } \rightarrow \begin{cases} 1 : noise \\ 0 : no\text{-}change \end{cases}$$

It predicts all possible training examples with a probability > 0, but it obviously contains no useful knowledge. The greedy algorithm tries to improve the rule set step by step by applying various search operators. Each operator aims to improve the existing rule set by adding new rules or altering existing ones. They all use the procedure *induceOutcomes*, which takes an incomplete rule without outcomes and computes the outcomes and their optimal probabilities according to the training experiences covered by the rule. The operators typically produce multiple candidate rule sets with modified added or dropped rules. The algorithm compares the best produced rule set with the current optimal rule set and replaces the optimal rule set if the new one is better. The algorithm always ensures that each observed state/action tuple (s, a') is only covered by one rule. When an existing rule is generalized so that it covers an experience that is already covered by a second rule, the second rule is removed from the new candidate rule set. The following is a rough summary of selected search operators, for further details see Pasula et al. (2007):

- *ExplainExperience* creates a new candidate rule set for each experience that is not yet covered by a rule. It adds a new rule that covers the experience which contains deictic references to model the outcomes as accurately as possible.

- *DropRules* drops existing rules.

- *DropContextLiterals* reduces the complexity of rules by removing context literals. For each rule, it creates multiple candidate rule sets with one context literal removed in every new rule set. The operator often generalizes existing rules (often produced by *ExplainExperiences*) so that they cover more training experiences.

- *SplitOnLiterals* splits rules into two more specific rules by adding a positive binary symbol to the one rule and the negated symbol to the other rule. The two created rules are added to a new candidate rule set. This is done for each binary atom that is absent in the selected rule.

- *SplitOnEqualities* operates like *SplitOnLiterals* but it deals with non-binary symbols. It adds multiple rules for each absent function atom such as $size(X)$, so that in each new rule the symbol is compared to another value. In the case of $size(X)$, one rule would contain the additional equality literal $size(X) = 1$, another $size(X) = 2$ etc.

- *SplitOnInequalities* also splits existing rules into more specific rules. For each function atom $f(X)$ that is absent in the selected rule it it creates a new candidate rule set for each possible value n of the symbol. In each candidate rule set the selected rule is replaced with two rules, in one rule the inequality literal $f(X) < n$ is added, in the other one $f(X) \geq n$ is added.

- *AddReferences* extends existing rules by adding a deictic reference.

- *ChangeRange* changes existing rules. For each equality literal $f(X) = n$ in the context of the selected rule it creates a new rule, so that in every new rule the function is compared to another value. Each new rule is inserted into a new candidate rule set, in which the original rule is not present.

- *MakeInterval* creates a new rule for each inequality literal in the selected rule. For example for the literal $f(X) < n$ it adds a new rule to the rule set in which the literal is replaced with $f(X) \geq n$.

- *GeneralizeEquality* generalizes rules by replacing equality literals with inequality literals (lesser and greater comparison).

A weight is assigned to each operator, which determines the probability that the operator is chosen by the algorithm in each step. When an operator does not find a better rule set, it is marked and not used again until another operator finds a new best rule set. The algorithm terminates when no operator can find a better rule set.

InduceOutcomes

As stated above, all search operators use the procedure *induceOutcomes* that computes the outcomes and their probabilities for an incomplete rule according to the training experiences. Consider the following incomplete rule:

$$grab(X) : block(Y), on(X, Y) \rightarrow \{$$

The first step is to compute the outcome for a single experience covered by the rule. The following experience triplet (s, a, s') is an example:

$((table(a), block(b), block(c), block(d), on(b, a), on(c, b), on(d, c)),$
$grab(c),$
$(table(a), block(b), block(c), block(d), inhand(c), on(b, a), on(d, a)))$

A pile of blocks stands on a table. Now the agent grabs the middle block (c) and as a result the upper block falls down on the table. This experience is covered by the rule because there is a valid substitution σ for X and Y so that the rule action is equal to the given action and the rule's context is entailed by the predecessor state: $X = c, Y = b$

The following steps are performed to compute the outcome for this experience:

1. The algorithm computes all literals that changed their truth values from predecessor to successor state, in this case: $inhand(c), \neg on(c, b), \neg on(d, c), on(d, a)$. This can be computed once for every experience and stored for future uses. This step can be easily adopted so that the algorithms also considers the change of non-binary symbols that change their values from predecessor to successor state which will be discussed in chapter 3.

2. Now, the inverse substitution of σ is applied to this outcome, which results in the outcome $inhand(X), \neg on(X, Y), \neg on(d, X), on(d, a)$.

Rules that contain constants typically lead to wrong generalizations. For example the outcomes $\neg on(d, X)$ and $on(d, a)$ are very specific to this experience because they only occur if the cube d stands on X. This is why outcomes that contain constants as arguments are dropped. That means that this experience can not be modeled by the rule and the outcome for this experience must be modeled as noise.

These individual outcomes are computed for every experience that is covered by the rule. Outcomes that contain constants are dropped and identical outcomes are merged, while it is stored which outcome covers how many experiences. The final and most challenging task to decide which observed experience outcomes should be modeled as noise and to compute the optimal probabilities for each outcome. Sometimes it can also improve the score to merge two outcomes into one if the combined outcome still covers enough experienced outcomes. Computing the optimal probabilities for n outcomes can be seen as the task of maximizing a n-dimensional vector function that takes the probabilities as input and returns the rule set score. An approximated solution can be found with various methods, for example gradient ascent. In order to decide which outcomes are modeled as noise and which should be merged the algorithm uses a greedy approach with two search operators *add* and *remove*. *add* tries to improve the score by merging non-contradicting outcomes, while *remove* drops outcomes.

Chapter 3

Learning and planning with non-binary symbols

The existing rule learning algorithm already considers non-binary symbols when building the context of a rule. This is achieved by the operators *ExplainExperiences*, *SplitOnEqualities*, *SplitOnInequalities*, *GeneralizeEquality*, *ChangeRange* and *MakeInterval*. In many cases this is important to distinguish different world situations, resulting in a more accurate model of the world. For instance, consider the task of building a tower by stacking boxes. If the agent puts a huge block with size 3 on a small block with size 1, the probability is high that the huge block will fall down. Putting a small block on a huge block is more likely to be successful, which is important knowledge for example in order to successfully reach the goal of building a high tower.

While non-binary symbols are successfully used to define contexts, the rule learning algorithm by Pasula et al. (2007) cannot learn how actions performed by the agent may change the value of a non-binary symbol. In the following we present how function value transitions can be represented and learned. Concrete changes to the rule learning algorithms are presented and open problems are discussed. Finally we investigate how the the planning algorithm PRADA can be used to achieve planning with the learned rules.

3.1 Learning with non-binary symbols

As a first step, the algorithm should be able to learn outcomes that can contain integer functions. The following rule is an example:

$$puton(X) : inhand(Y), f(X) = 3 \rightarrow \begin{cases} 1 : \neg inhand(Y), f(X) = 5 \\ 0 : noise \end{cases}$$

This is a NID rule according to Pasula et al. (2007), who explicitly allows integer functions with a constant value bound to them in the outcome of a NID rule. However, the

proposed algorithm only considers binary symbols when constructing the rule outcome. In order to learn outcomes that potentially contain non-binary symbols, *induceOutcomes* must be slightly extended. Fortunately, the same approach that is used for binary symbols can be used for non-binary symbols. The only step that must be extended is the computation of the individual outcomes for each experience. For each experience (s, a, s') it is computed once and stored which functions values in s' have changed compared to s. The individual outcome for an experience is computed by applying the inverse covering substitution to all changed symbols (binary or non-binary). Given these individual outcomes that potentially contain non-binary symbols, the final outcomes are computed as described in chapter 2.4.1.

3.1.1 Deictic references

Deictic references are an important feature of NID rules. In many cases actions affect objects that do not occur in the argument list of the action. Deictic references allow to include these objects in the rule which allows to model outcomes that could not be modeled otherwise. This is why the rule learning algorithm tries to use deictic references to improve the rule set score. However this may lead to problems in scenarios with several objects that have a changing non-binary property. Let's consider a scenario with multiple vending machines. Each machine has a button that causes the machine to give some water which decreases the amount of water in the water container. The button and the container are different objects that are in a relation with each other. The predicate $buttonTriggers(X, Y)$ is used to indicate the relation between the button X and the water container Y. The non-binary symbol $filled(Y)$ encodes the amount of water in percent in the container Y. Now the agents interacts with one machine, gathers some experiences and performs rule learning. The following is a possible rule that illustrates the problem:

$$push(X) : button(X), filled(Y) = 20 \rightarrow \begin{cases} 1 : filled(Y) = 15 \\ 0 : noise \end{cases}$$

The problem with this rule is that the button X and the container Y are not in a relation with each other. A rule context only covers a state if there is one unique substitution for all deictic references. In this case only one container was filled with 20% water in all training experiences, so Y is a covering deictic reference in the rule. The relation $buttonTriggers$ is not required and the literal was dropped to increase the rule set score. But in general, this is not a useful rule because it is not applicable if there is another container with a fuel of 20. It is even misleading and wrong if there is only one container with a fuel of 20 that belongs do a different machine. Avoiding these wrong generalizations completely would require 100 additional containers that each have a different $filled$ value in the training scenario. This example shows the general problem that non-binary symbols often lead to unwanted deictic references in learned rules. The reason is that because a non-binary symbol can take many values the probability is high that a value is unique among all training experiences. The first step to overcome this

problem is to establish the convention that no free non-binary symbols are allowed. This means that if a literal $f(Z) = 3$ occurs in a rule Z must occur in a positive binary symbol before. However the problem can still occur in some cases, consider the vending machine example:

$$push(X) : button(X), container(Y), filled(Y) = 20 \rightarrow \begin{cases} 1 : filled(Y) = 15 \\ 0 : noise \end{cases}$$

Now Y occurs in a positive binary symbol first but this changes nothing. The rule is still misleading if there is a container with a fuel of 20 that belongs do a different machine. The correct rule must include the relation between the button and the container. The problem can be solved by requiring that a unique covering substitution must also be unique when all non-binary symbols are removed from the rule context. This makes deictic references that are only unique because of a non-binary symbol impossible. In the given example, this forces the rule learning algorithm to use the relation between the button and the container:

$$push(X) : button(X), buttonTriggers(X, Y), filled(Y) = 20 \rightarrow \begin{cases} 1 : filled(Y) = 15 \\ 0 : noise \end{cases}$$

3.1.2 Learning change relative to the precedessor state

Learning that an action performed in a certain context leads to a successor state where a non-binary symbol is set to a constant value is a useful addition to the rule learning algorithm. However, there are a lot of examples where this approach performs poorly. Consider a function that represents a resource that is consumed or refilled by certain actions. In this example, the function $filled(X)$ retrieves the amount of liquid in percent that is in a bottle, thus the function ranges from 0 to 100. The agent makes the following observations:

$\{((bottle(a), filled(a) = 100), drink(a), (bottle(a), filled(a) = 98)),$
$((bottle(a), filled(a) = 98), drink(a), (bottle(a), filled(a) = 96))$
$((bottle(a), filled(a) = 96), drink(a), (bottle(a), filled(a) = 94))$
$((bottle(a), filled(a) = 94), drink(a), (bottle(a), filled(a) = 92))\}$

What rules should the agent learn from these experiences? The agent should have a model that predicts how $filled$ changes when it performs an action. The only way to achieve this is with fixed values in the outcomes:

$$drink(X) : bottle(X), filled(X) = 100 \rightarrow \begin{cases} 1 : filled(X) = 98 \\ 0 : noise \end{cases}$$

$$drink(X) : bottle(X), filled(X) = 98 \rightarrow \begin{cases} 1 : filled(X) = 96 \\ 0 : noise \end{cases}$$

$$drink(X) : bottle(X), filled(X) = 96 \rightarrow \begin{cases} 1 : filled(X) = 94 \\ 0 : noise \end{cases}$$

$$drink(X) : bottle(X), filled(X) = 94 \rightarrow \begin{cases} 1 : filled(X) = 92 \\ 0 : noise \end{cases}$$

This rule set contains 4 rules to cover 4 experiences, and it is not very useful when planning towards high reward states in most cases. How does $filled(X)$ change when the agents drinks in the context $filled(X) = 50$? The problem is that there is no generalization over the observed data. Dependant on the complexity penalty α, the rule learning algorithm would model all outcomes as noise to ensure generalization. Ideally, the agent should learn that the action *drink* decreases the resource function $filled$ by 2. Unfortunately, the only way to express this with the existing formalism for NID rules is to have one rule for each possible function value. This is not efficient and typically leads to poor generalization over the observed data, which is why an extension to the rule formalism is required.

Solving this problem requires rules that can express change relative to the predecessor state. This can be achieved by allowing arithmetic expressions in the outcome that may contain constants and function values from the precedessor state. To access the predecessor value of a non-binary symbol, we allow non-binary symbols in the rule context to be compared to variables, whereat only equality comparisons are permitted. These variables are treated the same way as variables that donate to world objects: A rule still covers a state if there is a suitable substitution for all variables. Consider the rule context $f(X) = \nu$ that only contains one equality literal. The rule covers all states in which there is a valid substitution σ for X so that $f(\sigma(X))$ is defined, because in this case $f(\sigma(X))$ is a valid substitution for ν. Now ν may be used inside the outcomes the same way X may be used, for example with $f(X) = \nu + 1$. This mechanism introduces a new kind of abstraction to the rule learning algorithm. Formerly the agent only abstracted from concrete world objects, now it may apply the same principle to function values.

With variables in equality literals and arithmetic expressions in rule outcomes where involving these variables, one single rule can be formulated that covers all the given experiences above:

$$drink(X) : bottle(X), filled(X) = \nu \rightarrow \begin{cases} 1 : filled(X) = \nu - 2 \\ 0 : noise \end{cases}$$

This rule encodes the same transition model T for a Markov decision process as 100 rules that each cover a different function value of $filled(X)$ (at least almost, see below). In contrast to a rule set with one rule for each value of $filled(X)$, this is actually a good rule set according to the scoring metric since $\sum_{(s,a,s') \in E} \log P(s' | s, r_{(s,a)})$ is maximal and the complexity penalty $\alpha \sum_{r \in \Gamma} PEN(r)$ is very low.

There exists one problem with the presented rule concerning the range of the function (in this case $0 - 100$). For $\nu = 1$ the rule it predicts that the function value will change to -1 which is out of range. This can be fixed by restricting the rule to cases where $filled(X)$ is greater than 1:

$$drink(X) : bottle(X), filled(X) = \nu, filled(X) > 1 \rightarrow \begin{cases} 1 : filled(X) = \nu - 2 \\ 0 : noise \end{cases}$$

Another possibility is to assume that the arithmetic expressions are implicitly clamped to fit in the symbol's range.

3.1.3 Search operators for rules with non-binary symbols

The algorithm of Pasula et al. (2007) cannot deal with non-binary symbols as presented above. In the following we propose extensions to learn rules that encode how a non-binary symbol changes relative to the precedessor state. We introduce two new search Operators *AbstractEquality* and *AddAbstractEquality*.

- *AbstractEquality* operates similar to *GeneralizeEquality* that generalizes equality literals by replacing them with inequality literals. *AbstractEquality* also generalizes existing rules, but by substituting concrete integer values in equality literals with function value variables. For each equality literal $f(X) = n, n \in \mathbb{N}$ in the context of the selected rule, it creates a new candidate rule set with a new rule in which the constant integer n is replaced with a new funcrion value variable.
- *AddAbstractEquality* is the counterpart to *SplitOnEqualities*. For each non-binary atom that is absent in the selected rule *SplitOnEqualities* creates a new candidate rule set. The new rule set contains one rule for each possible value of the symbol, splitting the rule into multiple more specific rules. *AddAbstractEquality* also creates a new candidate rule set for each non-binary atom $f(X)$ that is absent in the context of the selected rule. But instead of splitting the selected rule it only adds the comparison $f(X) = \nu$ to its context so that the symbol is compared to a new function value variable.

The two added search operators construct rules that contain abstract function values. In addition to that *induceOutcomes* must be extended to make use of these function value variables. The easiest way to achieve that is again to adopt the computation of a single outcome according to a particular experience. Consider the experience (s, a, s') and the incomplete rule r with the covering substitution σ.

$(s, a, s') = ((bottle(a), filled(a) = 100), drink(a), (bottle(a), filled(a) = 98))$

$$r = drink(X) : bottle(X), filled(X) = \nu \rightarrow \{$$

$$\sigma(X) = a, \;\; \sigma(\nu) = 100$$

The only changed atom in the experience is $filled(a)$, so the grounded outcome for the rule is $filled(a) = 98$. Now the inverse substitution is applied to the outcome, which

results in $filled(X) = 98$. However the 98 remains as a constant, so the algorithm tries to find an arithmetic expression that given the input $\sigma(\nu) = 100$ returns the value 98. Obviously infinitly fitting arithmetic expressions exist so there is no unique solution. Since according to Ockham's Razor the easiest explanation is usually the best we focus on simple arithmetic expression of the form $\sigma(\nu) + constant$. $constant$ is simply the difference between the function value in the successor state (in this case 98) and $\sigma(\nu)$ (which is the function value in the predecessor state). Following this approach leads to the outcome $filled(X) = \nu - 2$.

Given the individual outcomes for each experience, the computation of the final rule outcomes and their probabilities remains unchanged. In the case of the four example experiences, $filled(X)$ always decreases by 2 so the individual outcomes are equal and the algorithm merges them into one. Note that in this example the problem concerning the function range is present in the final rule. This is due to the fact that no example experience is provided with $filled(X) = 1$ or $filled(X) = 0$, in which the function does not decrease by 2. Hence, if functions ranges are treated as prior knowledge, we will follow the convention that the arithmetic expressions are implicitly clamped to fit in the function range.

3.1.4 Open problems and future work

With our approach transitions with a constant offset can be modeled. This is sufficient in many scenarios, but sometimes a different arithmetic expression could explain the observed experiences better. For instance, the agent could throw the bottle on the ground (by accident or on purpose) so that $filled(X)$ gets halved (roughly).
When building the arithmetic expression for the outcome, the presented approach only considers the change from a value to another in one experience at a time. A more powerful technique considers several predecessor and successor function values at once and tries to find an arithmetic expression that covers as many observed outcomes as possible. This way, the algorithm can find a pattern beyond constant offsets. In other words, this is the task of finding a function that best fits a given set of input/output value pairs. There are various techniques to achieve this, in this case it is reasonable to focus on first order polynomials of the form $ax + b$ that can be learned using linear regression with linear features. However, there remains a problem because the scoring metric only rewards outcomes that exactly match the observed experiences. Consider the following (admittedly somehow constructed) experiences:

$\{((bottle(a), filled(a) = 100), overthrow(a), (bottle(a), filled(a) = 49)),$
$((bottle(a), filled(a) = 49), overthrow(a), (bottle(a), filled(a) = 25))\}$

Intuitively, the following rule seems useful to explain the experiences because it is a good approximation:

$$overthrow(X) : bottle(X), filled(X) = \nu \rightarrow \begin{cases} 1 : filled(X) = \lfloor 0.5 \cdot \nu \rfloor \\ 0 : noise \end{cases}$$

However, the score for this rule with respect to the experiences is 0, because it does not

cover any of the observed outcomes. The scoring metric does not take into account how close the outcome function values are to the observed values, but only whether they are equal or not. A scoring metric that rewards predictions that only differ slightly from the actual observation higher than predictions that differ strongly would be a better measure for useful rule sets in many cases. This includes almost all thinkable scenarios that make use of non-binary symbols with large ranges (for example 0-100), especially when the agent's perception is exposed to noise. This requires to adopt the existing scoring metric because the term $P(s'|s, r_{(s,a)})$ only retrieves something different than the noise probability if the state s' is predicted exactly by the grounded rule in at least one of the outcomes.

One possibility is to introduce a distance metric $d(s_1, s_2)$ between two states s_1 and s_2. The distance could be defined as the sum of the differences between the individual primitive symbols. For binary symbols the difference would be always either 0 if the values are equal in both states or 1 if they are not equal. For a non-binary symbol f with the value v_1 in s_1 and v_2 in s_2 the difference would be $\frac{|v_1 - v_2|}{range(f)}$.

For each experience (s, a, s') the new scoring metric would choose the maximum of the probabilities $P(s''|s, r_{s,a})$, where s'' is a state equal or similar to the actual observed s' according to the distance metric, so that $d(s', s'')$ is lesser than a maximum distance d_{max}. In addition to that the probabilities should be multiplied with some factor to reward predictions with a minor distance to the actual observed state higher than states where the distance is closer to d_{max}. In the example given above a d_{max} of 0.05 could be used to allow a deviation of 5 percent for $filled(X)$ in the outcome, leading to a good generalization over the observed data.

3.2 Planning with changing non-binary symbols

Using the rule learning algorithm the agent is able to learn a MDP transition model from gathered experiences. As mentioned the algorithm algorithm PRADA (Lang and Toussaint (2010), see chapter 2) is used to achieve efficient planning towards high reward states based on the learned knowledge. Given a set of NID rules that encode a transition model and a reward function, PRADA computes an approximation of the optimal policy function π^* for the current state.

PRADA operates in *ground relational domains* that contain only concrete world objects. Hence, as a first step the abstract NID rules must be grounded according to the existing world objects before they can be used. In every ground rule the variables that occur in the original abstract rule are substituted with concrete world objects. The extended NID rules presented in this chapter not only contain object variables but also function value variables, so those must be grounded as well. One rule is created for each possible value the variable in the rule context can take. This requires the corresponding non-binary symbol to map to a fixed range, because otherwise there are inifinite possible ground rules. As part of the grounding process the arithmetic expressions in the rule outcomes are evaluated with respect to the particular substitution. The mentioned

issue with function ranges is addressed at this point by clamping the resulting grounded function values in the outcomes so that they fit in the function range if necessary.

PRADA constructs a dynamic bayesian network (DBN) that represents the set of grounded NID rules. It samples action sequences and computes for each sequence of actions the expected sum of rewards using approximated inference. It retrieves the sequence of actions that according to the ground rules leads to the highest expected sum of rewards. Binary and non-binary symbols are treated as random variables. It makes no difference for the algorithm whether these random variables are binary or not as long as they map to a finite set. This is why allthough it has not been investigated empirically yet PRADA is suitable to achieve planning with respect to changing non-binary symbols. For derived non-binary symbols that do not map to a fixed range (such as the derived symbol *height*) the algorithm does not compute the probabilities for each possible value but the expected value.

Efficiency Problems

The number of ground rules for an abstract rule depends on the number of possible substitutions for the variables that occur in the rule. Therefore the number of possible ground rules increases exponentially with the number of variables. This leads to efficiency problems when planning in scenarios with many objects and rules that contain multiple variables. With additional function value variables this is an even bigger problem, especially for functions with large ranges. Probably the most desirable solution to this problem is to perform planning on the level of abstract objects, enabling abstract reasoning. This approach does not require rule grounding and therefore avoids the exponential worst case time complexity. However while there has been promising results in this field of research many challenges remain and there exists no generic solution yet (Boutilier et al., 2001), (Sanner, 2008). They also did not investigate the usage of non-binary symbols empirically.

Another approach to overcome the time complexity problem is to limit the number of ground rules. Lang and Toussaint (2009) developed a technique named *Relevance Grounding* to focus on objects that are relevant to the task at hand. The basic idea is to derive a measure for the relevance of objects and only consider relevant objects as possible variable substitutions. In many cases, this approach reduces the number of ground rules significantly. A similar technique could be applied to function value variables as well. Consider the following rule that is grounded according to a world state that contains the literal $filled(a) = 40$:

$$drink(X) : bottle(X), filled(X) = \nu \rightarrow \begin{cases} 0.5 : filled(X) = \nu - 1 \\ 0.5 : filled(X) = \nu - 2 \\ 0 : noise \end{cases}$$

Obviously there are only very few relevant values for ν when planning with a horizon of four (the values from 32 to 40). This example illustrates that it may be worthwhile to investigate a relevance measure for function value variables.

Chapter 4

Evaluation

4.1 RMSim

In this thesis, the software RMSim written by Marc Toussaint and Tobias Lang was used for all tests. RMSim is a simulator written in C++ in which a virtual agent interacts with objects like cubes and balls. The behavior of the objects is simulated by the physics engine ODE (Open Dynamics Engine), while OpenGL is used for rendering.

One common criticism is that simulators like RMSim only provide idealized and unrealistic environments. RMSim addresses this problem by simulating noise using pseudo random numbers. For example when the agent puts an object on another object the exact drop position is exposed to noise so the object may sometimes fall down and sometimes not. Of course this environment is still very simple compared to the real world but this introduces a new challenge for the learning and planning algorithms in contrast to deterministic simulations. RMSim provides an interface that allows to convert the current world state into a symbolic relational representation. The usual testing scenario proceeds the following way:

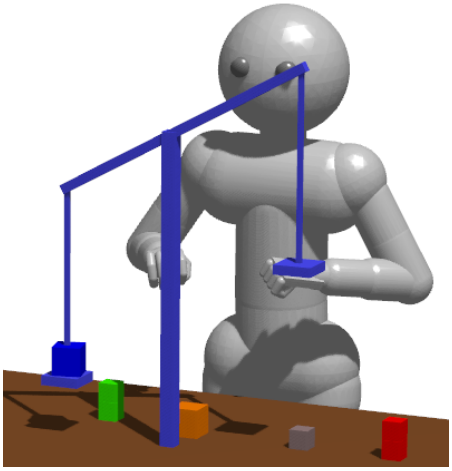
1. *Exploration*: The agent performs a sequence of actions inside the environment simulated by RMSim. RMSim saves a text file that consists of experience triplets (s, a, s').

2. *Learning*: Given the file exported by RMSim the rule learning algorithms learns an optimal set of NID rules (approximatively) and exports them into a file.

3. *Planning*: Given the MDP transition model encoded by the rule set file and a reward function the agent performs goal-directed behavior inside an environment simulated by RMSim. It is important to acknowledge that the environment may be vastly different from the environment in step 1. The agent has to apply his generalized learned knowledge in this unknown environment.

As part of this thesis, RMSim was extended to support a new object and the corresponding symbols.

4.2 Weighing machine



The weighing machine is an interesting addition to the simulator because it is a more complex object than boxes and balls. The agent has to learn that putting an object on a scale pan or grabbing an object from one changes the balance of the weighing machine. It also has to learn that putting an object on an object that is above a scale pan (or grabbing one) affects the balance. In addition to that, the agent should learn that bigger objects affect the weighing machine balance stronger than tiny objects (given that bigger objects are also heavier). The balance of the weighing machine can not be represented with a binary symbol in a meaningful way. At the simplest level three values are necessary to encode that the machine is either unbalanced to one side or balanced. A more accurate representation requires even more possible states so clearly a non-binary symbol is required. The scale scenario is an example how non-binary symbols can be used to encode geometric information.

The weighing machine is constructed using ODE bodies and joints. Using joints it is possible to specify a relationship between two objects, only allowing certain positions and orientations relative to each other (Smith (2002)). ODE supports various kinds of joints, including hinge joints. A hinge joint only allows rotation around a specified axis and is suitable to attach the upper beam to the main beam. This way, the behavior of the weighing machine is completely simulated by ODE like the other objects. This means that the agent also has to deal with noise when interacting with the machine. For example objects may fall down from the scale pans.

4.2.1 Choosing appropriate symbols

The first step is to choose an appropriate relational representation for the weighing machine. As a first approach the scale is represented with only two objects, one for each scale pan. This allows to use the familiar predicate *on* to indicate that an object stands on one of the scale pans. The predicate $scalePan(X)$ indicates that X is a scale pan and the relation $scalePanPair(X, Y)$ links the scale pans together. This relation is always commutative so that every state that contains $scalePanPair(X, Y)$ also contains $scalePanPair(Y, X)$. Note that in theory this representation allows absurd scales with multiple scale pans. However this is in fact the case for most symbols, for example the *on*-relation may be used to express that X stands on Y and Y on X. It is important to remember that the agent has no idea of the meaning we as humans associate with certain symbols or objects. What counts is that the symbolic vocabulary can be used to describe states that provide all information the agent needs to know in order to

interact with the scale. A possible way to encode the scale balance is to use a predicate $isDown(X)$ to indicate whether the scale pan X is down or not. A first problem with this representation is that it allows to encode a state that is never entered in which both scale pans are down. While impossible states may also be constructed with the predicate $scalePanPair$ it is more important here because $isDown$ is a predicate that will change often as a consequence to actions. In the rule learning algorithm the agent will always need both scale pans as deictic references to construct an outcome that covers any experiences. However this disadvantage seems rather insignificant since both scale pans are linked with the relation $scalePairPan$ and thus the other pan is available as a deictic reference. A second problem with this representation is that it is very inaccurate by making no difference whether a scale pan is completely down or only slightly heavier than the other pan. The be solved by using a non-binary symbol $scalePanHeight(X)$ instead of $isDown$ with a range from 1 to 5 which should be sufficient for most tasks. Unfortunately there exists a serious problem with this representation which concerns the introduced learning of relative transitions. The search operators $AbstractEquality$ and $AddAbstractEquality$ only add one function value variable at once to the processed rule but $scalePanHeight(X)$ and $scalePanHeight(Y)$ will always change silmultaneously with the invariant $scalePanHeight(X) = 5 - scalePanHeight(Y)$. For example, if $AbstractEquality$ adds $scalePanHeight(X) = \nu$ to the rule context there will still be many different outcomes in which $scalePanHeight(Y)$ has a different value. This is why the two search operators will seldom find a better rule set. Even if they were adopted to add two function value variables at once there would be efficiency problems when planning with PRADA because the rule grounding process would produce a lot of unnecessary ground rules.

To overcome this problem the scale balance is represented with only one non-binary symbol. Three objects are visible to the agent: The two scale pans and a third object that represents the weighing machine as a whole. The scale pans are in a relationship with the main scale object. The predicate $scalePart1(X, Y)$ states that X is a scale pan of the scale Y and the predicate $scalePart2(Z, Y)$ states Z is the other scale pan of Y. The function $scaleBalance(Y)$ encodes the current balance of the scale Y and ranges from 1 to 5. For example $scaleBalance(Y) = 1$ states that the first scale pan is completely down and the other up, $scaleBalance(Y) = 3$ states that the scale is balanced and $scaleBalance(Y) = 5$ states that the first scale pan is completely up and the other down. This representation also has a disadvantage: Each rule always applies to only one scale pan. The agent may learn the behavior of the left scale pan while having no knowledge about the right scale pan.

Altogether this is a good example how the chosen symbolic representation may influence the quality of the learned rules. It is noteworthy that some time was required to figure out appropriate symbols, which questions the autonomy of the agent: Where do the symbols come from? It seems that for many new scenarios a human developer is required who specifies corresponding symbols.

Derived symbols We also define some derived symbols to make relevant information more accessible for the agent.

- $aboveScalePan(X)$ is true if X is above a scale pan. It is simply the conjunction of $above(X, Y)$ and $scalePan(Y)$. This makes it easy for the agent to differentiate whether a grabbed object is above a scale pan or not. Consider a rule with the action $grab(X)$. It is very likely that the search operator $SplitOnLiterals$ will produce a better rule set at one point by splitting on $aboveScalePan(X)$.

- $aboveScalePan1(X, Y)$ (short $asp1$) identifies the object X that lies on top of the stack of objects on the left scale pan that is part of the scale Y. This may also be the left scale pan itself. This symbol allows an effective generalization over the observed experiences by merging the cases in which the agent puts a cube directly on a scale pan and the case in which the agent puts a cube on a cube that is above a scale pan. $aboveScalePan2(X, Y)$ (short $asp2$) is the counterpart to $aboveScalePan1$ and retrieves the same for the other pan.

- The derived non-binary symbol $scaleTowerDiff(X)$ (short $tdiff$) retrieves the difference of the heights of the two towers of objects that stand on both scale pans of the scale X. Consider the case in which many objects stand on one scale pan and few on the other. Putting an object on the other scale pan may have no visible consequence because the scale remains in the same balance. When putting an object on a scale pan the agent sometimes needs to factor in the number of objects that stand on the other scale pan.

4.2.2 Rule learning

A first test ist performed without the introduced derived symbols. The agent performs a provided sequence of 14 actions that should lead to meaningful observations. The actions are chosen so that the agent puts objects of different sizes on scale pans or grabs them. The observed experience triplets $\{(s, a, s')\}$ are stored in are stored in a file and serve es input for the rule learning algorithm. With a complexity penalty of 0.1, the resulting rule set contains 10 deterministic rules. This first test reveals some insights that must be considered for future tests. We examine two exemplary rules:

$$puton(X) : inhand(Y), scalePart1(X, Z), scaleBalance(Z) = \nu$$

$$\rightarrow \begin{cases} 1.0 : on(Y, X), scaleBalance(Z) = \nu - 2, \neg inhand(Y) \\ 0.0 : noise \end{cases}$$

This is a compact and useful rule although the probability of 1.0 is probably a bit unrealistic which is again due to the limited number of experiences. However it demonstrates the mentioned problem that rules only apply to one scale pan. The rule set does not contain the corresponding rule for the other scale pan because no suitable experiences are provided. The task of exploration and rule learning would be much easier if the

symbolic representation of the scale would allow rules that can be applied to both scale pans.

The other rule is another typical example that occurs when too few experiences where provided:

$$grab(X) : on(X,Y), scale(Z), size(X) = 1, scaleBalance(Z) = 3$$

$$\rightarrow \begin{cases} 1.0 : inhand(X), scaleBalance(Z) = 2, \neg on(X,Y) \\ 0.0 : noise \end{cases}$$

It states that given the balance of the scale Z is 3 grabbing an object X of the size 1 from an object Y will change the balance to 2. Obviously this is not a useful rule because Y may be *any* object such as a cube or the table. However it is a perfectly fine rule with respect to the provided experiences. This example shows that it is not enough to concentrate the exploration on the scale, the agent also has to gather other experiences such as grabbing a cube from another cube to avoid wrong generalizations. In addition to that it is a good idea to place another scale in the scene so that Z is not a valid deictic reference but must be linked to the scale pan.

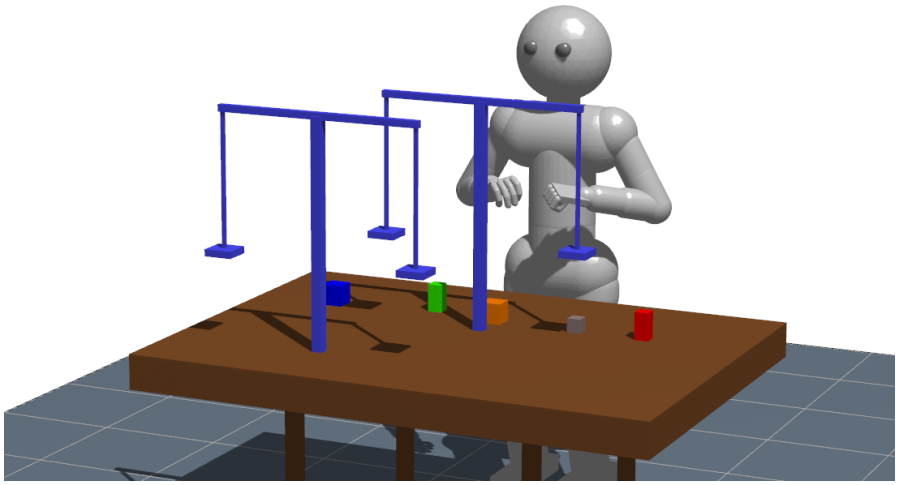
An important parameter for the rule learning algorithm are the priorities of the search operators that are crucial for the quality of the learned rules. This is due to the greedy nature of the algorithm: A search operator may produce a better rule set than the current but prevent the appliance of another search operator (or a sequence of search operators) that would produce an even better rule set in the same situation. This is why the priorities must be carefully adjusted. The priorities shown in figure 4.1 have proven to produce good results in many cases. Very important is the priority of *ExplainExperience*, a priority of 4 already leads to a different rule set typically. A high priority is assigned to the new search operators *AbstractEqualities* and *AddAbstractEqualities*.

Operator	priority
ExplainExperience	5.0
DropContextLiterals	100.0
DropReferences	2.0
DropRules	3.0
SplitOnLiterals	1.0
AddLiterals	1.0
AddReferences	2.0
SplitOnEqualities	1.0
SplitOnInequalities	0.5
ChangeRange	1.0
MakeInterval	1.0
GeneralizeEqualities	1.0
AbstractEqualities	2.0
AddAbstractEqualities	3.0

Figure 4.1: Search operator priorities

We now set up an exploration scenario with two scales and ten cubes of different sizes. Derived symbols are enabled. The agent only interacts with one of the scales, the other one only exists to prevent wrong generalizations. The sequence of actions is randomly generated, however the scale pans (more specifically the highest object standing on them) are priorized for *puton*-actions. This way the agent concentrates on the scale but also gathers some other experiences which allows it to learn what behavior is really specific

to the scale. Also, while the agent may grab any cube it only puts cubes on objects that are on top of a stack. A sequence of 160 experiences is generated and executed by the agent.



The complexity penalty α is set to 0.4. On a Core 2 Duo P8700, the rule learning algorithm terminates after 1290.0 seconds. The resulting rule set contains 11 rules:

1. $default() :\rightarrow \begin{cases} 0.05 : nochange \\ 0.95 : noise \end{cases}$
2. $puton(X) : inhand(Y), asp2(X,Z), size(Y) \leq 1, balance(Z) = \nu$
   $\rightarrow \begin{cases} 1 : on(Y,X), balance(Z) = \nu + 1, \neg inhand(Y) \\ 0 : noise \end{cases}$
3. $puton(X) : inhand(Y), asp2(X,Z), size(Y) > 1, balance(Z) = \nu, tdiff(Z) > -2$
   $\rightarrow \begin{cases} 1 : on(Y,X), balance(Z) = \nu + 1, \neg inhand(Y) \\ 0 : noise \end{cases}$
4. $puton(X) : inhand(Y), asp1(X,Z), tdiff(Z) > 1$
   $\rightarrow \begin{cases} 0.646 : on(Y,X), \neg inhand(Y) \\ 0.354 : on(Y,X), balance(Z) = 4, \neg inhand(Y) \\ 0 : noise \end{cases}$
5. $puton(X) : inhand(Y), asp1(X,Z), size(Y) \leq 1, balance(Z) = \nu, tdiff(Z) \leq 1$
   $\rightarrow \begin{cases} 1 : on(Y,X), balance(Z) = \nu - 1, \neg inhand(Y) \\ 0 : noise \end{cases}$
6. $puton(X) : table(Z), inhand(Y), \neg aboveScalePan(X)$
   $\rightarrow \begin{cases} 0.768 : on(Y,X), \neg inhand(Y) \\ 0.232 : on(Y,Z), \neg inhand(Y) \\ 0 : noise \end{cases}$

7. $puton(X) : inhand(Y), asp1(X,Z), size(Y) > 1, balance(Z) = \nu, tdiff(Z) \leq 1$
$$\rightarrow \begin{cases} 0.68 : on(Y,X), balance(Z) = \nu - 2, \neg inhand(Y) \\ 0.32 : on(Y,X), balance(Z) = \nu - 1, \neg inhand(Y) \\ 0 : noise \end{cases}$$

8. $puton(X) : inhand(Y), asp2(X,Z), size(Y) > 1, tdiff(Z) \leq -2$
$$\rightarrow \begin{cases} 1 : on(Y,X), balance(Z) = 2, \neg inhand(Y) \\ 0 : noise \end{cases}$$

9. $grab(X) : on(X,Y), clear(X), \neg aboveScalePan(X)$
$$\rightarrow \begin{cases} 1 : inhand(X) \neg on(X,Y) \\ 0 : noise \end{cases}$$

10. $grab(X) : on(X,Y), asp1(X,Z), balance(Z) = \nu, \neg block(Y)$
$$\rightarrow \begin{cases} 1 : inhand(X), balance(Z) = \nu + 2, \neg on(X,Y) \\ 0 : noise \end{cases}$$

11. $grab(X) : on(X,Y), block(Y), asp1(X,Z), balance(Z) = \nu$
$$\rightarrow \begin{cases} 1 : inhand(X), balance(Z) = \nu + 1, \neg on(X,Y) \\ 0 : noise \end{cases}$$

As intended the algorithm uses the derived symbol $tdiff$ to differentiate how putting an object on a scale pan affects the balance of the scale. The rules model the behavior of the scale for *puton* actions very accurately by using a function value variable for the scale balance. *grab* actions are modeled less accurately and a lot of experiences are covered by the default rule. The reason is that the agent often grabbed objects that are below other objects which leads to complex outcomes. Rule 9 is important because it states the agent can safely grab objects that are not above a scale pan and not below other objects.

Without the search operators *AddAbstractEquality* and *AbstractEquality* the resulting rule set contains 22 rules. They model the given experiences accurately with specific rules for each value of *balance*. However for some values of *balance* the rules do not differentiate between the sizes of objects that are put on a scale pan. For example the agent learned the following rules:

$$puton(X) : inhand(Y), asp2(X,Z), size(Y) = 1, balance(Z) = 2$$
$$\rightarrow \begin{cases} 1 : on(Y,X), balance(Z) = 3, \neg inhand(Y) \\ 0 : noise \end{cases}$$
$$puton(X) : inhand(Y), asp2(X,Z), balance(Z) = 3$$
$$\rightarrow \begin{cases} 1 : on(Y,X), balance(Z) = 4, \neg inhand(Y) \\ 0 : noise \end{cases}$$

In the first case the agent learned that the balance only changes by one if Y is a small cube. In the second case no experience were provided in which the agent puts a big cube on the scale pan so the algorithm only created one rule independant from the size of Y. This example shows that without abstract function values many experiences are needed to learn good rules for a changing non-binary symbol. This is even more problematic for symbols with larger ranges. In contrast, the new search operators allow one rule that covers all values of *balance*. This way the agent transfers the observation that only small cubes lead to a change by one to all possible values of *balance*.

Future work Three specialised derived symbols were used to achieve good learning results. However an autonomous agent should not depend on specialised given derived symbols and instead learn these symbols or achieve the same results without them. In the case of $scaleTowerDiff$ the algorithm needs to consider the relation between two non-binary symbols to construct a suitable rule such as:

$$\begin{aligned} &puton(X) : inhand(Y), asp1(X,U), scalePart1(Z,U), scalePart2(W,U), \\ &numAbove(Z) \geq numAbove(W) - 2, size(Y) = 1, balance(U) = \nu \\ &\rightarrow \begin{cases} 1 : on(Y,X), balance(U) = \nu - 1, \neg inhand(Y) \\ 0 : noise \end{cases} \end{aligned}$$

4.2.3 Planning

All planning scenarios are executed in the same world that consists a table a, a scale b, the two scale pans and seven cubes of different sizes. We use PRADA with 1200 samples and a planning horizon of 4. PRADA's type system is used to limit the number of ground rules. We specify that only objects of type *physical_free* can be grabbed and assign this type to all cubes. The agent can put objects on all objects of type *physical* which includes the table and the scale pans and on objects of type *physical_free*. We also specify that the function *scaleBalance* is only defined on objects of the type *scale*. This way the type system ensures that no unnecessary rules are created which improves the performance. By excluding actions such as grabbing a scale pan this also ensures that only sensible actions are sampled which leads to better planning results. Finally we also ensure that the agent only grabs objects if it has no other object in the hand by adding the derived predicate $inhandNil()$ to the contexts of all *grab* rules.

A first simple task is to reach the state $scaleBalance(b) = 4$. The agent fulfils this task by putting a block f with $size(f) = 1$ on the right scale pan.

As a second task the agent shall build a high tower on the left scale pan while the balance of the scale shall be 3. More formally, the following reward function is provided:

$$r(s) = \begin{cases} numAbove(c) & \text{if } scaleBalance(b) = 3 \\ 0 & \text{otherwise} \end{cases}$$

The agent puts two cubes of the same size on both scale pans, resulting in a scale balance of 3. Now the agent reached a state that has a reward of 1. After that the agent chose to stay in this state rather than building a higher tower. The reason is that to reach a state with a higher reward the agent has to go through two states without any reward: If it puts a block on the right scale pan the scale will become unbalanced. With a planning horizon of 4 the sum of expected rewards is higher if the agent just stays in the current state. To overcome this problem we change the reward function so that the current observed tower height is subtracted from the reward. This way the agent is always discontent with the observed state and wants to build a higher tower. We also include some time steps after the planning horizon into the calculation of the expected sum of rewards by assuming that the agent does nothing after the last planned action. This compensates that the agent has to go through states without reward to reach a higher reward. Now the agent grabs cubes of the same size and puts them on both scale pans alternately. More specifically, the agent puts objects on the highest object above a scale pan. Although the agent is only explicitly instructed to build a tower on left scale pan it also builds a tower on the right pan to fulfil the constraint $scaleBalance(c) = 3$. In theory the agent could do better by putting the two big cubes on the right pan and four small cubes on the left. However PRADA can not find this strategy with a planning horizon of four because the agent has to go through too many states without reward. From the initial state 6 actions are required to reach a higher reward state: The agent has to put two small cubes on the left pan and a big cube on the right. With a planning horizon of 6 and 2000 samples PRADA retrieves this sequence of actions as the best plan.

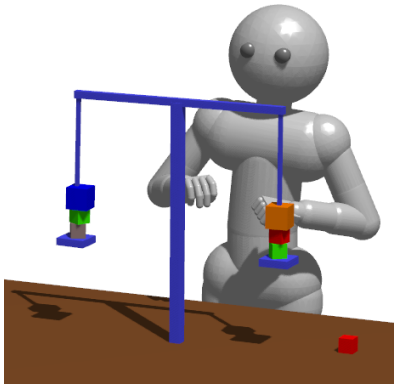


Figure 4.2: The agent builds towers on both scale pans to ensure $scaleBalance(b) = 3$

The task of building a high tower on the left scale pan is a bit more challending if the scale shall have a balance of 4 instead of 3. Putting an object on the left pan means that the balance decreases so the goal of building a high tower there is contrary to the constraint that the balance shall be 4. The agent solves this problem by putting a big block on the right scale pan first and then a small cube on the left pan. Now the scale has a balance of 4 and the agent can apply the strategy of putting cubes of the same size on both scale pans alternately again. Since the relevant *puton* rules abstract from the concrete scale balance value the same rules that were applied for the previous scenario are applied now.

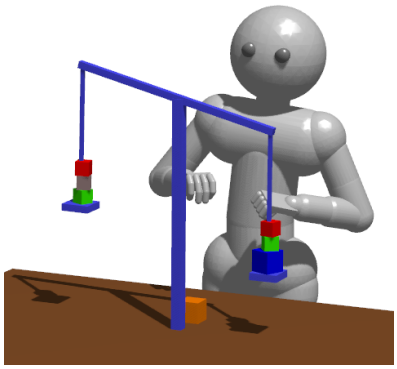


Figure 4.3: The same task with the constraint $scaleBalance(b) = 4$

Chapter 5

Conclusions

Non-binary symbols are often essential to derive accurate symbolic world models. In this thesis we investigated how symbolic relational transition models that include changing non-binary symbols can be represented, learned and used for planning towards high reward states. An important problem is that encoding the change of a non-binary symbol requires one NID rule for each value the symbol can take. To overcome this problem we have extended the rule learning algorithm by Pasula et al. (2007) so that it can learn NID rules that encode the change of a non-binary symbol relative to the precedessor state. This introduces a new possibility to generalize over the training experiences, leading to significantly better learning results in many scenarios. Non-binary symbols need to be handled differently in some respects compared to binary symbols to achieve good learning results. Deictic references, an important feature for effective learning, can often lead to wrong generalizations in combination with changing non-binary symbols. This issue was solved by adding the constraint that a deictic reference must not be identified with a non-binary symbol. We have shown empirically in a physically simulated robot manipulation domain that with the extended learning algorithm the agent is able to learn how to interact with a scale that requires a non-binary symbol to represent the balance. As a second contribution of this thesis the planning algorithm PRADA (more specifically the rule grounding process) was extended to use NID rules with changing non-binary symbols for planning towards high reward states. In our experiments the agent was able to fulfill tasks like building towers on a scale pan while maintaining a specified balance.

Future work The quality of the learned rules with changing non-binary symbols highly depend on the provided experiences and the symbolic representation. In the future, the agent should not depend on given derived symbols but either be able to learn these symbols or achieve the same results without them. In the investigated scale scenario this would require the learning algorithm to consider the relation between two non-binary symbols. Another more general problem is the rule learning algorithm's scoring metric which only rewards predicted outcomes that exactly match the observed outcome. This could be problematic in scenarios that depend on non-binary symbols

with large ranges or without fixed ranges. A possible improvement with respect to non-binary symbols would be a scoring metric that also rewards predictions that only differ slightly from the actual observed values.

Appendix A

Supplementary notes

A.1 Rule learning algorithm implementation changes

Lang and Toussaint (2010) implemented the algorithm presented by Pasula et al. (2007) using the C++ programming language as part of *libPRADA*. The program can read training data produced by the simulator *RMSim* and the resulting rule set can be used for planning with PRADA. As part of this thesis, the additions to the algorithm presented in this chapter were incorporated into this implementation. In addition to that, some changes were made to the code in order to deal better with non-binary symbols.

One noteworthy change to the existing implementation libPRADA (Lang and Toussaint, 2010) concerns the search operators *SplitOnEqualities*, *ChangeRange* and *MakeInterval*. These operators try out every possible function value for function atoms in some way. As an optimization, the implementation only considered function values that actually occured in the experiences. This is a reasonable optimization (especially for large ranges) and it also allows that these operators are applied to symbols without a fixed range. In this case it is impossible to try out every possible value but it is possible to try out every observed value. However the implementation computed these values only from one experience, assuming that the used function values are equal in all given experiences. Since non-binary symbols may change, this optimization must be abandoned or adopted in order to find all eligible function values for a non-binary symbol. This can be achieved by building a map that stores for each symbol a list of occuring function values by inspecting every experience. This is done once at initializing and the map is used by the mentioned search operators after that. This way never occuring function values are still skipped, but all eligible function values are considered.

Another change to the implementation is the addition of the seach operator *SplitOnInequalities*. For each function atom that is absent in the selected rule this operator splits the rule into two more specific rules by adding the inequality literal $f(X) < c$ to one rule and $f(X) \geq c$ to the other rule. This search operator was described by Pasula et al. (2007) as part of *SplitOnLiterals* but was not implemented in *libPRADA*.

Another extension made to the algorithm does not directly concern non-binary symbols, but it can improve the results in complex domains with many predicates and functions. The rule learning algorithm performs a greedy search through the space of possible rules. This is achieved by the various search operators that augment or add rules, trying to improve the rule set score step by step. This leads to good results in most cases, however sometimes the greedy approach has its drawbacks. Generally speaking, a rule augmentation that does not improve the rule set set directly may be a requirement for another step that drastically improves the score. This problem is visible in the search operator *AddReference*, which adds a new deictic reference to existing rules. Sometimes a predicate that contains a new deictic reference like $on(X, Y)$ (where Y is the new deictic reference) may not be useful by itself, but it may serve as a link to another deictic reference which drastically improves the rule set score. There is no proper solution to this problem because it arises due to the very nature of the greedy technique, but there is a simple workaround to overcome this limitation in most practical cases. The search operator *AddIndirectReference* was added that adds two references at once two the rule. However this search operator is disabled by default because it weakens the performance. In the evaluation this problem was adressed by defining a derived symbol ($aboveScalePart1$ is a good example) which is more efficient and makes the task of learning good rules easier. However it is not a perfect solution because the agent should learn things like this on his own. *AddIndirectReference* may be useful for that in the future.

Literature

C. Boutilier and S. H. Thomas Dean. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

C. Boutilier, R. Reiter, and B. Price. Symbolic programming for first-order mdps. 2001.

T. Lang. *Planning and Exploration in Stochastic Relational Worlds.* PhD thesis, FU Berlin, 2011.

T. Lang and M. Toussaint. Relevance grounding for planning in relational domains. In *Proc. of the European Conf. on Machine Learning (ECML)*, 2009.

T. Lang and M. Toussaint. Planning with noisy probabilistic relational rules. *Journal of Artificial Intelligence Research*, 39, 2010.

H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29, 2007.

S. Russel and P. Norvig. *Artificial intelligence: A modern approach*, page 598. Prentice Hall International Editions, 1996.

S. P. Sanner. *First-order decision-theoretic planning in structured relational environments.* PhD thesis, University of Toronto, 2008.

R. Smith. *How to make new joints in ODE.* http://www.ode.org/joints.pdf, 2002.

A. L. Strehl, L. Li, E. Wiewiora, J. Langford, and M. L. Littman. Pac model-free reinforcement learning. 2006.

P. Tadepalli, R. Givan, and K. Driessens. Relational reinforcement learning: An overview. *Proceedings of the ICML'04 workshop on Relational Reinforcement Learning*, 2004.

T. J. Walsh. *Efficient learning of relational models for sequential decision making.* PhD thesis, The State University of New Jersey, 2010.