Florian Halbritter, Peter Geibel: Learning Models of Relational MDPs Using Graph Kernels.
Proceedings of the Mexican Conference on Artificial Intelligence (MICAI 2007),
LNAI 4827, pp. 409-419. Springer.

Learning Models of Relational MDPs using Graph Kernels

Florian Halbritter and Peter Geibel

Institute of Cognitive Science, University of Osnabrück
Albrechtstrasse 28, 49076 Osnabrück, Germany
{fhalbrit,pgeibel}@uos.de

Abstract. Relational reinforcement learning is the application of reinforcement learning to structured state descriptions. Model-based methods learn a policy based on a known model that comprises a description of the actions and their effects as well as the reward function. If the model is initially unknown, one might learn the model first and then apply the model-based method (indirect reinforcement learning). In this paper, we propose a method for model-learning that is based on a combination of several SVMs using graph kernels. Indeterministic processes can be dealt with by combining the kernel approach with a clustering technique. We demonstrate the validity of the approach by a range of experiments on various *Blocksworld* scenarios.

1 Introduction

In the past decade, reinforcement learning (RL) has received a lot of attention in the field of artificial intelligence and machine learning [1, 2]. More recently, various researchers have suggested an extension of this technique called *relational reinforcement learning* (RRL) [3–6]. In contrast to the traditional approach, where states are usually represented as some sort of feature vector, RRL makes use of structured representations. A relational view of states and actions allows to abstract from the very situation itself to its structural features and therefore to generalize from specific tasks to the important characteristics of the problem.

In this article, we present an indirect model-based framework, in which we employ a series of support vector machines (SVMs; cp. [7]) to learn a model, which can thereafter be used as a basis for model-based RL methods to learn policies including dynamic programming methods like relational variants of value iteration and policy iteration [2, 8–10]. The support vector machines use a powerful product graph kernel [11, 4] to deal with graph-based representations of the state-action space.

The main contribution of our work is the learning of the model using a combination of SVMs. This way, we can extend the expressiveness of classical STRIPS-like production rule, and, although we focus on qualitative models in this article, our approach can be extended for including real-valued information easily; indeterministic models can be obtained by using a clustering method in a previous step, for grouping possible outcomes of an action application.

In order to show the feasibility of our approach, we evaluate our framework through numerous experiments on planning tasks in the *Blocksworld* [12, 13] and show its clear superiority with respect to the need for training data over a model-free variant of our algorithm. Note that our focus is on learning the model. We therefore employ a relatively simple, straightforward method for learning the policy given the already learned model. The method requires complete retraining the value function from time to time. Although this method works well, it is possible to combine the model-learning approach with other model-based learning techniques (e.g. [8–10]).

The remainder of this paper is structured as follows: In section 2, we begin with a brief review of the current state of model-learning in RRL.Afterwards, in the sections 3 and 4, we explain the proposed framework in detail, by showing how we use SVMs to learn a model of the environment and how we utilize these to train an RRL agent. Experiments in the *Blocksworld* will be the focus of section 4. We present numerous experiments performed in different settings of the *Blocksworld* and the results obtained. Finally, we conclude with an extensive discussion of the implications of our results and this study as a whole and propose some issues for future work (section 6).

2 Model-Learning for Relational MDPs

Model-free RRL techniques [3, 11] do not presuppose a known model of the process to be controlled. Instead, learning an evaluation function for state-action pairs allows to derive an optimal policy without having learned the model comprising the state transition probabilities and the reward function. On the other hand, it is well-known that model-based methods like Value Iteration and Policy Iteration usually converge faster because they can update a value function for states based on *all* its successor states [2, 8–10]. Moreover, it is possible to combine model-free algorithms with model-based planning methods, see [14]. The work of Croonenborghs et al. is directly related to our approach: it uses probabilistic first order decision trees that specify for single literals their probability of being true in the successor state, assuming conditional independence. In contrast, we use an expressive, kernel-based method which also allows to predict quantitative information. Although the kernel part as such is deterministic, a clustering of successor states for the same action yield an indeterministic model that assigns probabilities to successor states as a whole instead of to a single literal, thus better accounting for correlations.

Classical RL approaches treat the states as being either atomic or fixed-length vectors. For atomic states, model learning (or estimation of the Markov Decision Process (MDP)) consists of learning the reward function (or distribution) together with determining which successor states can occur with what probability when an action is carried out in a state. The problem of learning a relational MDP additionally requires learning suitable action descriptions. For deterministic planning problems, one usually considers STRIPS-rules consisting of preconditions and effects, mostly represented by three lists of literals.

In the scientific literature, the induction of first order rules has been considered relatively rarely [15–18], although it had first been addressed in the 1970s [19]. Finding the relevant preconditions is either based on specializing a most general (empty) precondition, or by computing generalizations of appropriate training examples (graphs or logical descriptions). The effects of an action can be obtained by comparing a state and its successor state.

While Benson [16] and Wang [17] also consider examples of unsuccessful rule applications (negative examples), Pasula et al. [15] and Gil [18] assume the presence of positive examples only. In the latter case, either additional criteria for finding appropriate rules or the generation of negative examples is required. Here, we assume that examples of unsuccessful actions applications are given as well or might be generated from examples of other rules.

In this work, instead of explicitly generating STRIPS-like rules, we specify the model by trained support vector machines operating on annotated graphs. This allows us to incorporate numerical information in a straightforward manner and we are not restricted by properties of the STRIPS language or its variants.

3 Model Learning Using Graph Kernels

A model for RL has the sole purpose to predict the effects of actions that an agent carries out, i.e. the resulting state, as well as the immediate reward the agent obtains. In this work, we use a combination of SVMs to produce these predictions. An approximate, yet efficient way for describing graphs representing the states, is the use of so-called label sequences [20]. In this paper, we employ the product graph kernel [11, 4], which provides a powerful means for comparing graphs with respect to common label sequences within them. The computation of the graph kernel for g and g' is based on the adjacency matrix, A, of the so-called product graph whose node set consists of pairs of nodes (n, n') from the original graphs. The kernel is computed as

$$k(g, g') = \sum_{r=1}^{R} \sum_{n,m \in g, n', m' \in g'} A^r((n, n'), (m, m')) \ ,$$

where A^r denotes the r-th power of the adjacency matrix of the product graph. $A^r((n, n'), (m, m'))$ corresponds to the number of common label sequences of length r from n to m in g and from n' to m' in g'. R is the maximum path length, which has to be selected by the user. Note that by allowing continuous values in the product graph, we can generalize the kernel for using quantitative information. Using it e.g. with support vector regression thus allows to represent complex real-valued functions based on structural and numerical information.

As an input, all SVMs receive a graph-representation of the current state plus some additional information depending on the type of the SVM, see below. Such a graph has one node for each entity occurring in the state. The label of a node is either empty or corresponds to the type of the entity. In contrast to the type, node properties like color or size will be encoded by a labelled loop

for the respective node, i.e. an edge. Edges exist also between nodes for which certain relations hold (the labels corresponding to the names of the relations). Additionally, it is important to indicate the action that is to be carried out from this state in order to let the SVMs know how the state will be affected. To do so, we introduce an *action marker* to the state graph, that is, an additional node with a label corresponding to the name of the action and edges pointing from the marker node to each of the entities involved in the action. Furthermore, we augment the labels of the affected nodes to show that they are arguments of the action.

An example of a graph corresponding to a state in the Blocksworld, as we will encounter later, can be seen in Fig. 1 (left) . Each block is represented by a node, with the node type given in the respective circle (we left out the node identifiers from N). Nodes are characterized by the two properties `onTable` and `clear` (i.e. there is no other block on top). `on` is the only (real) binary relation. The action of moving one block on top of another has been named `move` and is represented by a dashed circle. The arguments nodes of the action were labelled as `move/1` and `move/2`, respectively. The `move`-action node is connected to its arguments by edges also labeled `move/1` and `move/2`.

By adding the action to the graph, additional paths are introduced to the graph. These paths will be identical in all graphs with the same marker. Thus, the graph kernel will recognize a higher degree of similarity for graphs containing the same marker, because it is based on the number of common label sequences in the two graphs it operates on. The chosen action representation allows the model to consider actions affecting more than two nodes.

In order to determine the successor state when applying an action, we employ one SVM S_r for each type of feature/relation r in the state descriptions (examples for such features are `on`, `clear` and `onTable`). A single SVM S_r represents the effects of an action for a specific feature and operates on pairs of nodes (excluding action and query nodes, see below). In this work, we chose to predict if the respective label changes or not, although it also possible to predict edge/no edge directly, or a real value in the case of qualitative information like `distance`, etc. For each relation r the respective SVM will, hence, be called several times for each possible combination of two nodes in the graph[1] representing the state. For the respective pair, the SVM S_r predicts whether the corresponding relation will exist in the next state or not. In order to tell the SVM which pair of nodes we are considering at the moment, an additional marker, the *query marker* $q(n_1, n_2)$, is added in the same way as the action marker.

Combining the predictions of all SVMs S_r for all node pairs allows us to create the next state graph. In order to reduce the complexity (number of pairs), we assume that nodes affected by an action must be explicitly "mentioned" in the arguments of an action $a(n_0, \ldots, n_k)$, which is a common assumption in AI problem solving, so not all pairs of nodes have to be considered.

[1] n-ary relations have not been investigated in this work. However, this would not pose a problem, in principle, since it is a well-known fact that arbitrary n-ary predicates can be decomposed into a set of binary predicates.

The model uses an additional SVM S_{PRE} solely for predicting whether an action is *applicable* in a state or not. This SVM, which operates on the state graph augmented with the action marker, checks the pre-conditions of the action. Since non-applicable actions leave the state unchanged, we can therefore neglect a majority of possibly costly transition predictions for non-applicable actions in favour of speeding up the whole process.

The model now consists of an SVM S_r for every relation r, which maps a state graph, an action marker, and a query marker to {change, nochange}. The successor state is obtained by combining the predictions for every relation and every node pair, given the current state. The SVM S_{PRE} operating on states with an action marker predicts the applicability of an action. A regression SVM S_{REW} determines the reward, when an action is applied to a state.

Indeterministic processes can be handled by first clustering the state transitions belonging to the same action m. This clustering of graphs can be achieved with standard techniques operating on a suitable measure of graph similarity or distance. We can, for instance, use the kernel as the similarity measure or we might resort to alternatives that have already been applied to such tasks successfully. The method described above is then applied to each single cluster. The proportion of the examples in each cluster can be used for estimating the probability for the corresponding state transition.

4 Model-Based Approximate, Asynchronuous Value Iteration

Grounding on the SVM model just explained, any well-established model-based RL method can be used, in principle, to derive an optimal policy. For the purpose of this work, we decided to stick to use a variant of the Value Iteration [1, 2] procedure which is a simple, yet powerful method for learning optimal policies in the case of discounted cumulative returns. Value Iteration is based on the update

$$V^{t+1}(x) = \max_u \left[\sum_{x'} p_{x,u}(x')(r_{x,u}(x') + \gamma V^t(x')) \right]$$

where $p_{x,u}(x')$ is the probability of reaching x' when action u is used in x, and $r_{x,u}(x')$ is the respective reward. V^t denotes the t-th approximation of the optimal value function. The learnt model is necessary for evaluating the RHS of the equation.

Unfortunately, a pure table-based representation of the value function will be practically infeasible for all but the smallest state spaces. Therefore we again employ another support vector machine S_{VAL} acting on state graphs to store the value function. In the following, we will concentrate on typical planning problems where the task is to reach a goal state as quickly as possible. Since the reward for every single step is -1, we did not learn the reward function (i.e., S_{REW}), but assumed it to be known.

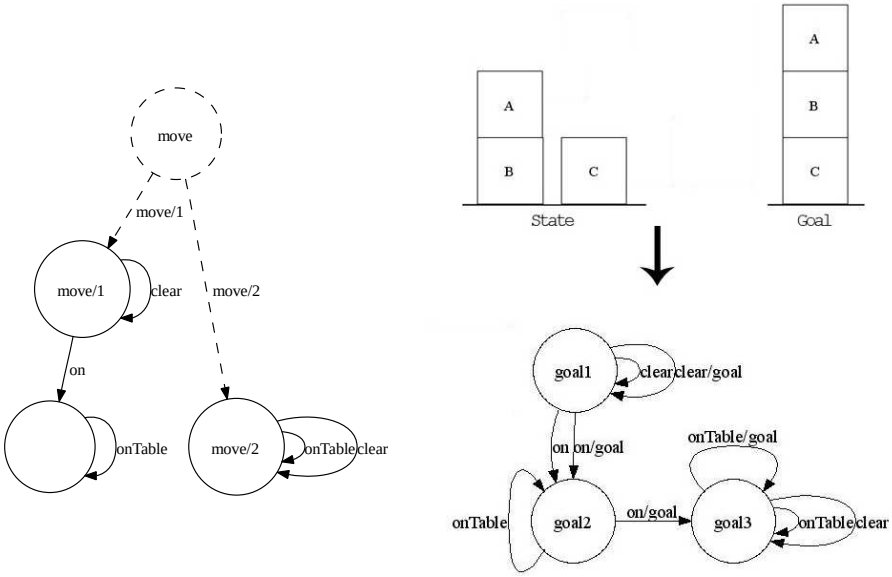

Fig. 1. *Blocksworld* states and graphs. For explanation see text.

Because the utility of a state does not ground purely on the state itself, but actually on the relation of the state to the goal state, it increases the flexibility to mark the goal conditions in the graph in order to predict the expected cumulative return with S_{VAL}. This can be easily accomplished by adding all relations from the description of the goal state to the original graph. In order to distinguish them from the normal relations in the state, their labels are augmented by adding a distinct string "goal" to them. Additionally, the labels of all nodes corresponding to the entities occurring in the goal conditions are changed by adding "goal" to them, too. Both an example state and the corresponding graph, are shown in Fig. 1 (right).

Value updates are performed asynchronously based on the current estimates. Since support vector machines are incapable of online learning, the function S_{VAL} needs to be retrained from scratch periodically, that is, whenever the number of erroneous predictions exceeds a certain threshold (the default value is 1, meaning updates are performed whenever a value estimate changes).

5 Experiments and Results

In this section we evaluate the proposed framework for model-based reinforcement learning by investigating its performance on several planning tasks in the *Blocksworld*. For all experiments, we used an extended version of the LIBSVM [21]. We will assess the performance of the model on predicting state transitions

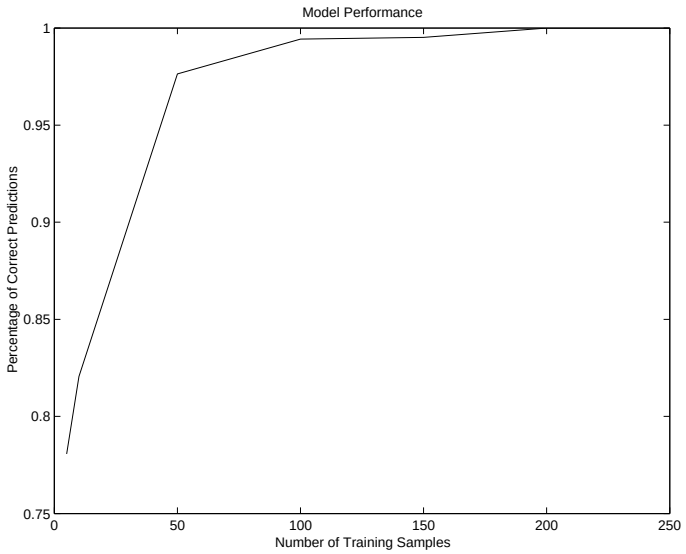

Fig. 2. Performance of the model: The graph shows the number of correct state predictions depending on the number of samples the model has been trained with.

and the agent's capability of learning optimal policies for stacking blocks in a particular order.

5.1 Problem Settings

A configuration in the *Blocksworld* consists of a pre-defined number of blocks and a table. Our aim is to stack the blocks in a specific order using a set of available actions. Only two actions are available: Moving a block X on top of another block Y (`move(X,Y)`) and moving a block X onto the table (`moveTable(X)`). The states themselves are characterized via various properties (features, predicates), here: A block X is on top of another block Y (`on(X,Y)`), a block X is on the table (`onTable(X)`) or a block X is clear (`clear(X)`), i.e. there is no other block on top of it. Clearly, the actions only succeed if specific conditions hold. For example, we can only move block X onto block Y (`move(X,Y)`), if both blocks are clear (`clear(X)`,`clear(Y)`). Evidently, a block Y cannot be clear, if there is another block Z on top of it (`on(Z,Y)`). Thus, the purpose of our model is to learn which conditions apply to which actions and how the actions affect the states.

Representing the *Blocksworld* states as graphs can be done in a straightforward manner by introducing one node for each block mentioned in the state description and edges for the relations. Figure 1 shows an example of such a *Blocksworld* state with additionally an action having been marked in it in the way described in the previous chapter.

Table 1. Percentage of correct predictions of transitions when confronting a perfect predictor trained in a simple environment with a more complex *Blocksworld*. The values in the brackets refer to the percentage of correct predictions when looking only at the positive samples, i.e. those where the action changed the state.

Trained with...	**Tested with...**	**Accuracy**
3 Blocks	5 Blocks	100.0 (100.0)
3 Blocks	8 Blocks	88.6 (72.6)
5 Blocks	8 Blocks	83.6 (32.4)
3 to 5 Blocks	8 Blocks	100.0 (100.0)

5.2 Results

In a first run of experiments we evaluated the capability of the proposed model to predict state transitions in *Blocksworlds* with three to eight blocks. We trained the model with increasingly many random transition samples (10, 50, 100, 150, 200, 250) once for each number of blocks. Afterwards we tested the model with 1000 further random transition examples and recorded the percentage of correct predictions. In order to reduce the effect of randomness, we repeated this procedure five times and took an average over the results. Figure 2 shows the results obtained.

Evidently, the model achieved a perfect prediction in all cases when provided with about 200 training samples. Note that for the easier cases drastically less samples were sufficient. In particular, it was possible to train a perfect model for three blocks with only 10 carefully hand-picked, most representative transition examples. It is furthermore interesting to notice that the mispredictions mostly resulted from 'positive' transitions, i.e. transitions where the state actually changed. The model can easily distinguish between successful and unsuccessful applications of actions, but needs some more training data to correctly predict state changes.

It is now important to assess the model's generalization ability. To do so, we first trained the model with 250 transition samples with only three blocks. As we have seen previously, this is likely to give us a good predictor. Then we tested the performance on 1000 random transitions in a *Blocksworld* with 5 or 8 blocks and recorded the performance. Again, we averaged over five runs. Refer to Table 1 for the results.

The results prove that the model was able to generalize well to a slightly more complex environment, but was significantly flawed the more different the environment was from the one it was trained in. While it was still mostly able to distinguish successful from unsuccessful actions, predicting novel states posed a harder problem to the model. However, when we trained the model with 250 transitions samples with 3 to 5 blocks, a very good prediction could be achieved even in the 8-blocks scenario. In this case, the model apparently succeeded to extract the relevant characteristics of the problem and was no longer restricted to the one particular scenario it was trained with.

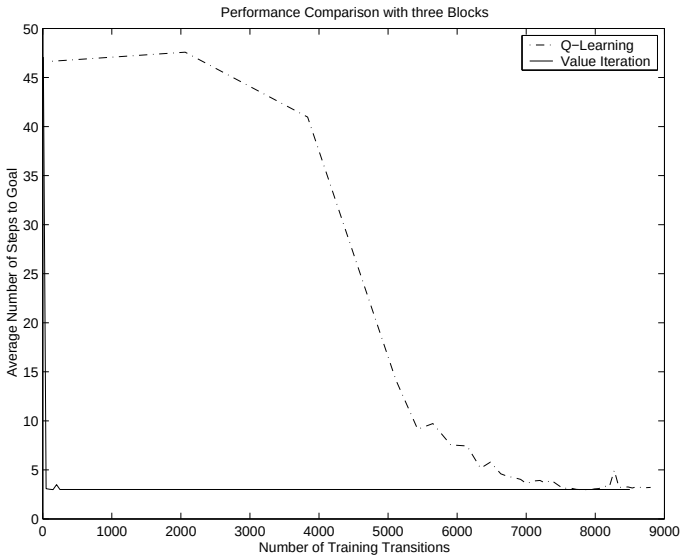

Fig. 3. Performance of the RRL agent in comparison to Q(λ)-Learning : The graph shows the average number of steps required to reach the goal step after the agent has been trained with a total of the stated transition samples.

In a second batch of experiments, the RRL agent's ability to find optimal policies on the basis of this model was to be investigated. In a similar fashion as before, we tested the agent in a 4-blocks *Blocksworld*, by first training its model with a varying number of transition samples and then allowing it to establish a policy using Value Iteration. After the training had converged, the performance was tested using 100 random starting states and we recorded the number of steps it took to reach the goal (stacking all three blocks in a fixed order) in each run. If the agent failed to reach the goal within at most 50 steps, the run was considered to have failed, and a count of 50 steps was recorded. Again, we averaged over five runs. As a means for comparison, we furthermore implemented a Q(λ)-learning agent using replacing eligibility traces[2] and trained it to find an optimal policy. For this agent, we paused the learning procedure every 10 episodes and performed 100 test runs to assess its current performance. We also recorded the total number of actual transitions the agent had seen by that time in order to gain a comparable scale for the RRL agent. Figure 3 depicts the results obtained.

$Q(\lambda)$-learning needed about 7000 transition examples (about 300 episodes) to find an optimal policy reaching the goal in some 3-5 steps (depending on the

[2] In a number of previous experiments we investigated the influence of the different parameters on the agent and eventually fixed them to the best possible choice, which was $\alpha = 0.8$ (learning rate), $\epsilon = 0.1$ (exploration rate) and $\lambda = 0.6$ (influence of future experience on earlier state updates).

starting position). In contrast, the RRL agent could derive an equivalent policy from the model which needed only some 50 transition examples. The advantage could not be any more striking. Once the number of transition samples sufficed to create a perfect model, the agent could also learn an optimal policy. When provided with a flawed model, the policy will be largely random and therefore, least surprisingly, suboptimal.

Additionally, we investigated the possibility of speeding up training by previously training the agent on a simpler task with only three blocks. A remarkable decrease in the number of steps required for Value Iteration to reach convergence could be observed: While usually about 1200 cycles were required before Value Iteration converged to an optimal policy, starting with an optimal policy found in the smaller *Blocksworld* (which usually needed some 100 cycles), convergence could be reached in only 375 cycles. Large parts of the previously learnt knowledge could apparently be reused for the more complicated scenario.

6 Conclusion

We have presented a framework for the automated creation of a powerful model for reinforcement learning and a simple learning procedure based on Value Iteration. An agent equipped with these tools has been shown to clearly outperform an agent based on $Q(\lambda)$-Learning with respect to the number of training data required to find an optimal policy. Furthermore, we were able to show that the agent was capable of benefiting from experience gained in a simpler, but fundamentally similar task when trying to learn a more complex scenario.

When comparing the results of our approach to those reported for the RRL-KBR procedure suggested by Gärtner and Driessens [4], we see that in general at least 100 episodes were required to find an optimal policy for their tasks, which amounts to thousands of transition examples. Other agents based on episodic, trial-and-error learning perform similarly. Moreover, they reported stacking blocks in a fixed order, in particular, to be a hard task for the agent to solve, which, however, did not pose any problems to our agent using the graph formalism proposed in this work. Future work, however, has to incorporate the comparison with model-based or indirect RRL learning methods like, for instance, FOALP, FLUCAP, ReBel, etc.

There is a lot of potential for future work in this field. Further investigations of the applicability of this framework to real-world tasks would be desirable. In a next step, we will also try to integrate the possibility of using real-valued properties (and possibly actions) into the graph kernel and the framework on the whole. First experiments with this have provided promising results, however, need further fine-tuning. Moreover, it will certainly be necessary to further improve the reinforcement learning component of the framework. While the Value Iteration procedure with the SVM-based function approximator was sufficient for the investigations at hand, it is unlikely that it will be computationally feasible for more complex tasks involving large graphs.

References

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (1998)
2. Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-Dynamic Programming. Athena Scientific (1996)
3. Dzeroski, S., Raedt, L.D., Driessens, K.: Relational reinforcement learning. Machine Learning **43**(1-2) (2001) 7–52
4. Driessens, K., Ramon, J., Gärtner, T.: Graph kernels and gaussian processes for relational reinforcement learning. Machine Learning **64**(1-3) (2006) 91–119
5. Tadepalli, P., Givan, R., Driessen, K.: Relational reinforcement learning: An overview. In: Proceedings of the ICML 2004 Workshop on Relational Reinforcement Learning. (2004)
6. van Otterlo, M.: A survey of reinforcement learning in relational domains. Technical report, CTIT Technical Report, TR-CTIT-05-31, July 2005, 70 pp, CTIT Technical Report Series, ISSN 1381-3625 (2005)
7. Vapnik, V.N.: The nature of statistical learning theory. Springer-Verlag New York, Inc., New York, NY, USA (1995)
8. Kersting, K., Otterlo, M.V., Raedt, L.D.: Bellman goes relational. In Brodley, C.E., ed.: ICML, ACM (2004)
9. Scanner, S., Boutilier, C.: Approximate linear programming for first-order mdps. In: Proceedings UAI 2005. (2005)
10. Hoelldobler, S., Karabaev, E., Skvortsova, O.: FluCaP: a heuristic search planner for first-order mdps. JAIR **27** (2006) 419–439
11. Gärtner, T.: A survey of kernels for structured data. SIGKDD Explor. Newsl. **5**(1) (2003) 49–58
12. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1995)
13. Gupta, N., Nau, D.S.: Complexity results for blocks-world planning. In: Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91). Volume 2., Anaheim, California, USA, AAAI Press/MIT Press (1991) 629–633
14. Croonenborghs, T., Ramon, J., Blockeel, H., Bruynooghe, M.: Online learning and exploiting relational models in reinforcement learning. In Veloso, M.M., ed.: IJCAI. (2007) 726–731
15. Pasula, H., Zettlemoyer, L.S., Kaelbling, L.P.: Learning probabilistic relational planning rules. In: ICAPS. (2004) 73–82
16. Benson, S.: Inductive learning of reactive action models. In: International Conference on Machine Learning. (1995) 47–54
17. Wang, X.: Learning planning operators by observation and practice. In: Artificial Intelligence Planning Systems. (1994) 335–340
18. Gil, Y.: Learning by experimentation: Incremental refinement of incomplete planning domains. In: ICML. (1994) 87–95
19. Vere, S.A.: Inductive learning of relational productions. In Waterman, D., Hayes-Roth, F., eds.: Pattern-Directed Inference Systems. Academic Press (1978)
20. Geibel, P., Wysotzki, F.: Learning relational concepts with decision trees. In Saitta, L., ed.: Machine Learning: Proceedings of the Thirteenth International Conference, Morgan Kaufmann Publishers, San Fransisco, CA (1996) 166–174
21. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. (2001) Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.