Speeding Up Relational Reinforcement Learning through the Use of an Incremental First Order Decision Tree Learner

Kurt Driessens, Jan Ramon, and Hendrik Blockeel

Department of Computer Science K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{kurt.driessens,jan.ramon,hendrik.blockeel}@cs.kuleuven.ac.be

Abstract. Relational reinforcement learning (RRL) is a learning technique that combines standard reinforcement learning with inductive logic programming to enable the learning system to exploit structural knowledge about the application domain.
This paper discusses an improvement of the original RRL. We introduce a fully incremental first order decision tree learning algorithm TG and integrate this algorithm in the RRL system to form RRL-TG.
We demonstrate the performance gain on similar experiments to those that were used to demonstrate the behaviour of the original RRL system.

1 Introduction

Relational reinforcement learning is a learning technique that combines reinforcement learning with relational learning or inductive logic programming. Due to the use of a more expressive representation language to represent states, actions and Q-functions, relational reinforcement learning can be potentially applied to a wider range of learning tasks than conventional reinforcement learning. In particular, relational reinforcement learning allows the use of structural representations, the abstraction from specific goals and the exploitation of results from previous learning phases when addressing new (more complex) situations.

The RRL concept was introduced by Džeroski, De Raedt and Driessens in [5]. The next sections focus on a few details and the shortcomings of the original implementation and the improvements we suggest. They mainly consist of a fully incremental first order decision tree algorithm that is based on the G-tree algorithm of Chapman and Kaelbling [4].

We start with a discussion of the implementation of the original RRL system. We then introduce the TG-algorithm and discuss its integration into RRL to form RRL-TG. Then an overview of some preliminary experiments is given to compare the performance of the original RRL system to RRL-TG and we discuss some of the characteristics of RRL-TG.

L. De Raedt and P. Flach (Eds.): ECML 2001, LNAI 2167, pp. 97–108, 2001.

Initialise $\hat{Q}_0$ to assign 0 to all (s, a) pairs
Initialise Examples to the empty set.
e := 0
while true
  generate an episode that consists of states s_0 to s_i and actions a_0 to a_{i-1}
    (where a_j is the action taken in state s_j) through the use of a standard
    Q-learning algorithm, using the current hypothesis for $\hat{Q}_e$
  for j=i-1 to 0 **do**
    generate example $x = (s_j, a_j, \hat{q}_j)$,
      where $\hat{q}_j := r_j + \gamma max_{a'} \hat{Q}_e(s_{j+1}, a')$
    if an example $(s_j, a_j, \hat{q}_{old})$ exists in Examples, replace it with x,
      else add x to Examples
  update $\hat{Q}_e$ using TILDE to produce $\hat{Q}_{e+1}$ using Examples
  for j=i-1 to 0 **do**
    for all actions a_k possible in state s_j **do**
      if state action pair (s_j, a_k) is optimal according to $\hat{Q}_{e+1}$
      then generate example (s_j, a_k, c) where $c = 1$
      else generate example (s_j, a_k, c) where $c = 0$
  update $\hat{P}_e$ using TILDE to produce $\hat{P_{e+1}}$ using these examples (s_j, a_k, c)
  e := e + 1

Fig. 1. The RRL algorithm.

2 Relational Reinforcement Learning

The RRL-algorithm consists of two learning tasks. In a first step, the classical Q-learning algorithm is extended by using a relational regression algorithm to represent the Q-function with a logical regression tree, called the Q-tree. In a second step, this Q-tree is used to generate a P-function. This P-function describes the optimality of a given action in a given state, i.e., given a state-action pair, the P-function describes whether this pair is a part of an optimal policy. The P-function is represented by a logical decision tree, called the P-tree. More information on logical decision trees (classification and regression) can be found in [1,2].

2.1 The Original RRL Implementation

Figure 1 presents the original RRL algorithm. The logical regression tree that represents the Q-function in RRL is built starting from a knowledge base which holds correct examples of state, action and Q-function value triplets. To generate the examples for tree induction, RRL starts with running a normal episode just like a standard reinforcement learning algorithm [10,9,6] and stores the states, actions and q-values encountered in a temporary table. At the end of each episode, this table is added to a knowledge base which is then used for the

tree induction phase. In the next episode, the q-values predicted by the generated tree are used to calculate the q-values for the new examples, but the Q-tree is generated again from scratch.

The examples used to generate the P-tree are derived from the obtained Q-tree. For each state the RRL system encounters during an episode, it investigates all possible actions and classifies those actions as optimal or not according to the q-values predicted by the learned Q-tree. Each of these classifications together with the according states and actions are used as input for the P-tree building algorithm.

Note that old examples are kept in the knowledge base at all times and never deleted. To avoid having too much old (and noisy) examples in the knowledge base, if a state–action pair is encountered more than once, the old example in the knowledge base is replaced with a new one which holds the updated q-value.

2.2 Problems with the Original RRL

We identify four problems with the original RRL implementation that diminish its performance. First, it needs to keep track of an ever increasing amount of examples: for each different state-action pair ever encountered a Q-value is kept. Second, when a state-action pair is encountered for the second time, the new Q-value needs to replace the old value, which means that in the knowledge base the old example needs to be looked up and replaced. Third, trees are built from scratch after each episode. This step, as well as the example replacement procedure, takes increasingly more time as the set of examples grows. A final point is that leaves of a tree are supposed to identify clusters of equivalent state-action pairs, "equivalent" in the sense that they all have the same Q-value. When updating the Q-value for one state-action pair in a leaf, the Q-value of all pairs in the leaf should automatically be updated; but this is not what is done in the original implementation; an existing (state,action,Q) example gets an updated Q-value at the moment when exactly the same state-action pair is encountered, instead of a state-action pair in the same leaf.

2.3 Possible Improvements

To solve these problems, a fully incremental induction algorithm is needed. Such an algorithm would relieve the need for the regeneration of the tree when new data becomes available. However, not just any incremental algorithm can be used. Q-values generated with reinforcement learning are usually wrong at the beginning and the algorithm needs to handle this appropriately.

Also an algorithm that doesn't require old examples to be kept for later reference, eliminates the use of old information to generalise from and thus eliminates the need for replacement of examples in the knowledge base. In a fully incremental system, the problem of storing different q-values for similar but distinct state-action pairs should also be solved.

In the following section we introduce such an incremental algorithm.

3 The TG Algorithm

3.1 The G-Algorithm

We use a learning algorithm that is an extension of the G-algorithm [4]. This is a decision tree learning algorithm that updates its theory incrementally as examples are added. An important feature is that examples can be discarded after they are processed. This avoids using a huge amount of memory to store examples.

On a high level (cf. Figure 2), the G-algorithm (as well as the new TG-algorithm) stores the current decision tree, and for each leaf node statistics for all tests that could be used to split that leaf further. Each time an example is inserted, it is sorted down the decision tree according to the tests in the internal nodes, and in the leaf the statistics of the tests are updated.

```
create an empty leaf
while data available do
    split data down to leafs
    update statistics in leaf
    if split needed then
        grow two empty leafs
endwhile
```

Fig. 2. The TG-algorithm.

The examples are generated by a simulator of the environment, according to some reinforcement learning strategy. They are tuples $(State, Action, QValue)$.

3.2 Extending to First Order Logic

We extend the G algorithm in that we use a relational representation language for describing the examples and for the tests that can be used in the decision tree. This has several advantages. First, it allows us to model examples that can't be stored in a propositional way. Second, it allows us to model the feature space. Even when it would be possible in theory to enumerate all features, as in the case of the blocks world with a limited number of blocks, a problem is only tractable when a smaller number of features is used. The relational language can be seen as a way to construct useful features. E.g. when there are no blocks on some block A, it is not useful to provide a feature to see which blocks are on A. Also, the use of a relational language allows us to structure the feature space as e.g. on(State,block_a,block_b) and on(State,block_c,block_d) are treated in exactly the same way.

The construction of new tests happens by a refinement operator. A more detailed description of this part of the system can be found in [1].

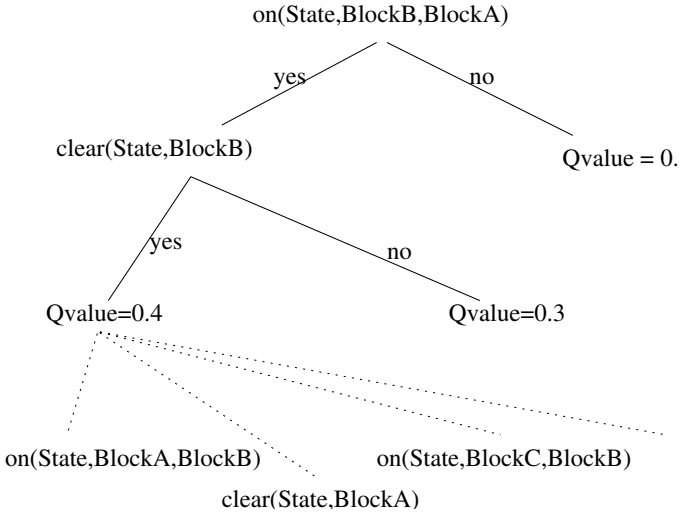

Fig. 3. A logical decision tree

Figure 3 gives an example of a logical decision tree. The test in some node should be read as the existentially quantified conjunction of all literals in the nodes in the path from the root of the tree to that node.

The statistics for each leaf consist of the number of examples on which each possible test succeeds, the sum of their Q values and the sum of squared Q values. Moreover, the Q value to predict in that leaf is stored. This value is obtained from the statistics of the test used to split its parent when the leaf was created. Later, this value is updated as new examples are sorted in the leaf. These statistics are sufficient to compute whether some test is significant, i.e. the variance of the Q-values of the examples would be reduced by splitting the node using that particular test. A node is split after some minimal number of examples are collected and some test becomes significant with a high confidence.

In contrast to the propositional system, keeping track of the candidate-tests (the refinements of a query) is a non-trivial task. In the propositional case the set of candidate queries consists of the set of all features minus the features that are already tested higher in the tree. In the first order case, the set of candidate queries consists of all possible ways to extend a query. The longer a query is and the more variables it contains, the larger is the number of possible ways to bind the variables and the larger is the set of candidate tests.

Since a large number of such candidate tests exist, we need to store them as efficiently as possible. To this aim we use the query packs mechanism introduced in [3]. A query pack is a set of similar queries structured into a tree; common parts of the queries are represented only once in such a structure. For instance, a set of conjunctions $\{(p(X), q(X)), (p(X), r(X))\}$ is represented as a term $p(X), (q(X); r(X))$. This can yield a significant gain in practice. (E.g., as-

suming a constant branching factor of b in the tree, storing a pack of n queries of length l ($n = b^l$) takes $O(nb/(b-1))$ memory instead of $O(nl)$.)

Also, executing a set of queries structured in a pack on the examples requires considerably less time than executing them all separately. An empirical comparison of the speeds with and without packs will be given in section 1.

Even storing queries in packs requires much memory. However, the packs in the leaf nodes are very similar. Therefore, a further optimisation is to reuse them. When a node is split, the pack for the new right leaf node is the same as the original pack of the node. For the new left sub-node, we currently only reuse them if we add a test which does not introduce new variables because in that case the query pack in the left leaf node will be equal to the pack in the original node (except for the chosen test which of course can't be taken again). In further work, we will also reuse query packs in the more difficult case when a test is added which introduces new variables.

3.3 RRL-TG

To integrate the TG-algorithm into RRL we removed the calls to the TILDE algorithm and use TG to adapt the Q-tree and P-tree when new experience (in the form of $(State, Action, QValue)$ triplets) is available. Thereby, we solved the problems mentioned in Section 2.2.

- The trees are no longer generated from scratch after each episode.
- Because TG only stores statistics about the examples in the tree and only references these examples once (when they are inserted into the tree) the need for remembering and therefore searching and replacing examples is relieved.
- Since TG begins each new leaf with completely empty statistics, examples have a limited life span and old (possibly noisy) q-value examples will be deleted even if the exact same state-action pair is not encountered twice.

Since the bias used by this incremental algorithm is the same as with TILDE, the same theories can be learned by TG. Both algorithms search the same hypothesis space and although TG can be misled in the beginning due to its incremental nature, in the limit the quality of approximations of the q-values will be the same.

4 Experiments

In this section we compare the performance of the new RRL-TG with the original RRL system. Further comparison with other systems is difficult because there are no other implementations of RRL so far. Also, the number of first order regression techniques is limited and a full comparison of first order regression is outside the scope of this paper.

In a first step, we reran the experiments described in [5]. The original RRL was tested on the blocks world [8] with three different goals: unstacking all blocks,

stacking all blocks and stacking two specific blocks. Blocks can be on the floor or can be stacked on each other. Each state can be described by a set (list) of facts, e.g., $s_1 = \{clear(a), on(a,b), on(b,c), on(c,floor)\}$. The available actions are then $move(x,y)$ where $x \neq y$ and x is a block and y is a block or the floor. The unstacking goal is reached when all blocks are on the floor, the stacking goal when only one block is on the floor. The goal of stacking two specific blocks (e.g. a and b) is reached when the fact $on(a,b)$ is part of the state description, note however that RRL learns to stack any two specific blocks by generalising from $on(a,b)$ or $on(c,d)$ to $on(X,Y)$ where X and Y can be substituted for any two blocks. In the first type of experiments, a Q-function was learned for state-spaces with 3, 4 and 5 blocks. In the second type, a more general strategy for each goal was learned by using policy learning on state-spaces with a varying number of blocks.

Afterwards, we discuss some experiments to study the convergence behaviour of the TG-algorithm and the sensitivity of the algorithm to some of its parameters.

4.1 Fixed State Spaces

For the experiments with a fixed number of blocks, the results are shown in Figure 4. The generated policies were tested on 10 000 randomly generated examples. A reward of 1 was given if the goal-state was reached in the minimal number of steps. The graphs show the average reward per episode. This gives an

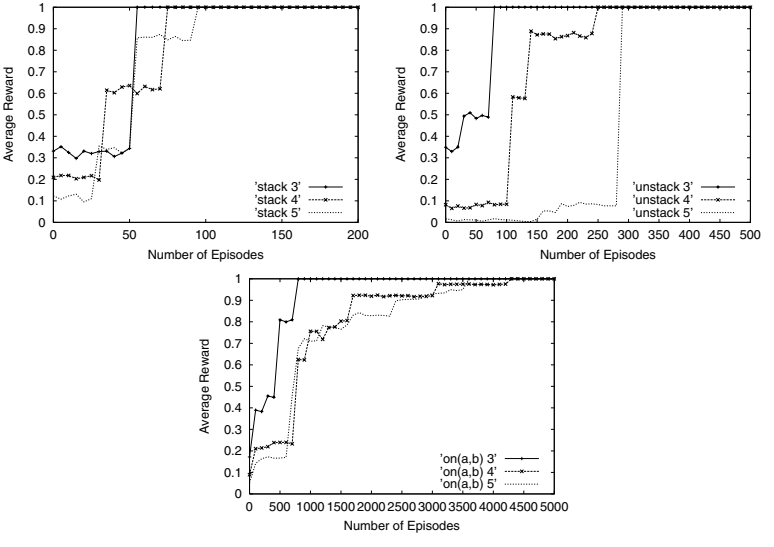

Fig. 4. The learning curves for fixed numbers of blocks

indication of how well the Q-function generates the optimal policy. Compared to the original system, the number of episodes needed for the algorithm to converge to the correct policy is much larger. This can be explained by two characteristics of the new system. Where the original system would automatically correct its mistakes when generating the next tree by starting from scratch, the new system has to compensate for the mistakes it makes near the root of the tree — due to faulty Q-values generated at the beginning — by adding extra branches to the tree. To compensate for this fact a parameter has been added to the TG-algorithm which specifies the number of examples that need to be filed in a leaf, before the leaf can be split. These two factors — the overhead for correcting mistakes and the delay put on splitting leaves — cause the new RRL system to converge slower when looking at the number of episodes.

However, when comparing timing results the new system clearly outperforms the old one, even with the added number of necessary episodes. Table 1 compares the execution times of RRL-TG with the timings of the RRL system given in [5]. For the original RRL system, running experiments in state spaces with more than 5 blocks quickly became impossible. Learning in a state-space with 8 blocks, RRL-TG took 6.6 minutes to learn for 1000 episodes, enough to allow it to converge for the stacking goal.

Table 1. Execution time of the RRL algorithm on Sun Ultra 5/270 machines.

		3 blocks	4 blocks	5 blocks
Original RRL	Stacking (30 episodes)	6.15 min	62.4 min	306 min
	Unstacking (30 episodes)	8.75 min	not stated	not stated
	On(a,b) (30 episodes)	20 min	not stated	not stated
RRL-TG without packs	Stacking (200 episodes)	19.2 sec	26.5 sec	39.3 sec
	Unstacking (500 episodes)	1.10 min	1.92 min	2.75 min
	On(a,b) (5000 episodes)	25.0 min	57 min	102 min
RRL-TG with packs	Stacking (200 episodes)	8.12 sec	11.4 sec	16.1 sec
	Unstacking (500 episodes)	20.2 sec	35.2 sec	53.7 sec
	On(a,b) (5000 episodes)	5.77 min	7.37 min	11.5 min

4.2 Varying State Spaces

In [5] the strategy used to cope with multiple state-spaces was to start learning on the smallest and then expand the state-space after a number of learning episodes. This strategy worked for the original RRL system because examples were never erased from the knowledge base, so when changing to a new state-space the examples from the previous one would still be used to generate the Q- and P-trees.

However, if we apply the same strategy to RRL-TG, the TG-algorithm first generates a tree for the small state-space and after changing to a larger state-

space, it expands the tree to fit the new state space, completely forgetting what it has learned by adding new branches to the tree, thereby making the tests in the original tree insignificant. It would never generalise over the different state-spaces. Instead, we offer a different state-space to RRL-TG every episode. This way, the examples the TG-algorithm uses to split leaves will come from different state-spaces and allow RRL-TG to generalise over them. In the experiments on the blocks world, we varied the number of blocks between 3 and 5 while learning. To make sure that the examples are spread throughout several state-spaces we set the minimal number of examples needed to split a leaf to 2400, quite a bit higher than for fixed state-spaces. As a result, RRL-TG requires a higher number of episodes to converge.

At episode 15000, P-learning was started. Note that since P-learning depends on the presence of good Q-values, in the incremental tree learning setting it is unwise to start building P-trees from the beginning, because the Q-values at that time are misleading, causing suboptimal splits to be inserted into the P-tree in the beginning. Due to the incremental nature of the learning process, these suboptimal splits are not removed afterwards, which in the end leads to a more complex tree that is learned more slowly. By letting P-learning start only after a reasonable number of episodes, this effect is reduced, although not necessarily entirely removed (as Q-learning continues in parallel with P-learning).

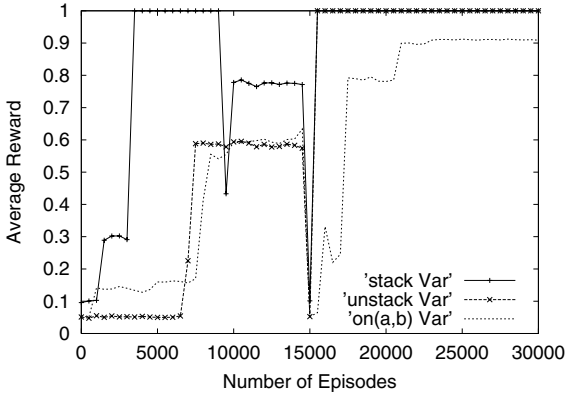

Fig. 5. The learning curves for varying numbers of blocks. P-learning is started at episode 15 000.

Figure 5 shows the learning curves for the three different goals. The generated policies were tested on 10 000 randomly generated examples with the number of blocks varying between 3 and 10. The first part of the curves indicate how well the policy generated by the Q-tree performs. From episode 15000 onwards, when P-learning is started, the test results are based on the P-tree instead of the Q-tree; on the graphs this is visible as a sudden drop of performance back to almost

zero, followed by a new learning curve (since P-learning starts from scratch). It can be seen that P-learning converges fast to a better performance level than Q-learning attained. After P-learning starts, the Q-tree still gets updated, but no longer tested. We can see that although the Q-tree is not able to generalise over unseen state-spaces, the P-tree — which is derived from the Q-tree — usually can.

The jump to an accuracy of 1 in the Q-learning phase for stacking the blocks is purely accidental, and disappears when the Q-tree starts to model the encountered state-spaces more accurately. The generated Q-tree at that point in time is shown in Figure 6. Although this tree does not represent the correct Q-function in any way, it does — by accident — lead to the correct policy for stacking any number of blocks. Later on in the experiment, the Q-tree will be much larger and represent the Q-function much closer, but it does not represent an overall optimal policy anymore.

```
state(S),action_move(X,Y),numberofblocks(S,N)
  height(S,Y,H),diff(N,H,D),D<2 ?
    yes:
      qvalue(1.0)
    no:
      height(S,B,C),height(S,Y,D), D < C ?
        yes:
          qvalue(0.149220955007456)
        no:
          qvalue(0.507386078412001)
```

Fig. 6. The Q-tree for Stacking after 4000 episodes

The algorithm did not converge for the $On(A, B)$ goal but we did get better performance than the original RRL. This is probably due to the fact that we were able to learn more episodes.

4.3 The Minimal Sample Size

We decided to further explore the behaviour of the RRL-TG algorithm with respect to the delay parameter which specifies the minimal sample size for TG to split a leaf. To test the effect of the minimal sample size, we started RRL-TG for a total of 10 000 episodes and started P-learning after 5000 episodes. We then varied the minimal sample size from 50 to 2000 examples. We ran these tests with the unstacking goal.

Figure 7 shows the learning curves for the different settings, Table 2 shows the sizes of the resulting trees after 10 000 episodes (with P-learning starting at episode 5000. The algorithm does not converge for a sample size of 50 examples. Even after P-learning optimality is not reached. The algorithm makes too many

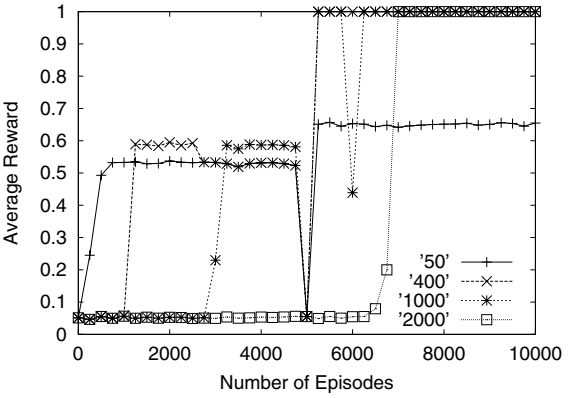

Fig. 7. The learning curves for varying minimal sample sizes

mistakes by relying on too small a set of examples when choosing a test. With an increasing sample size comes a slower convergence to the optimal policy. For a sample size of 1000 examples, the dip in the learning curve during P-learning is caused by a change in the Q-function (which is still being learned during the second phase). When looking at the sizes of the generated trees, it is obvious that the generated policies benefit from a larger minimal sample size, at the cost of slower convergence. Although experiments with a smaller minimal sample size have to correct for their early mistakes by building larger trees, they often still succeed in generating an optimal policy.

Table 2. The generated tree sizes for varying minimal sample sizes

	50	400	1000	2000
Q-tree	31	26	16	9
P-tree	9	10	8	8

5 Concluding Remarks

This paper described how we upgraded the G-tree algorithm of Chapman and Kaelbling [4] to the new TG-algorithm and by doing so greatly improved the speed of the RRL system presented in [5].

We studied the convergence behaviour of RRL-TG with respect to the minimal sample size TG needs to split a leaf. Larger sample sizes mean slower convergence but smaller function representations.

We are planning to apply the RRL algorithm to the Tetris game to study the behaviour of RRL in more complex domains than the blocks world.

Future work will certainly include investigating other representation possibilities for the Q-function. Work towards first order neural networks and Bayesian networks [7] seems to provide promising alternatives.

Further work on improving the TG algorithm will include the use of multiple trees to represent the Q- and P-functions and further attempts to decrease the amount of used memory. Also, some work should be done to automatically find the moment to start P-learning and the optimal minimal sample size. Both will influence the size of the generated policy and the convergence speed of TG-RRL.

Acknowledgements. The authors would like to thank Sašo Džeroski, Luc De Raedt and Maurice Bruynooghe for their suggestions concerning this work. Jan Ramon is supported by the Flemish Institute for the Promotion of Science and Technological Research in Industry (IWT). Hendrik Blockeel is a post-doctoral fellow of the Fund for Scientific Research of Flanders (FWO-Vlaanderen).

References

1. H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.
2. H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of clustering trees. In *Proceedings of the 15th International Conference on Machine Learning*, pages 55–63, 1998. `http://www.cs.kuleuven.ac.be/~ml/PS/ML98-56.ps`.
3. H. Blockeel, B. Demoen, L. Dehaspe, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ILP. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference in Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 60–77, London, UK, July 2000. Springer.
4. David Chapman and Leslie P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisions. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1991.
5. S. Džeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43:7–52, 2001.
6. L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
7. K. Kersting and L. De Raedt. Bayesian logic programs. In *Proceedings of the tenth international conference on inductive logic programming, work in progress track*, 2000.
8. P. Langley. *Elements of Machine Learning*. Morgan Kaufmann, 1996.
9. T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
10. R. Sutton and A. Barto. *Reinforcement Learning: an introduction*. The MIT Press, Cambridge, MA, 1998.