Cost-Sensitive Reinforcement Learning

Robby Goetschalckx and **Kurt Driessens**
Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
B-3001 Heverlee – Belgium
robby.goetschalckx@cs.kuleuven.be
kurt.driessens@cs.kuleuven.be

Abstract

We introduce cost-sensitive regression as a way to introduce information obtained by planning as background knowledge into a relational reinforcement learning algorithm. By offering a trade-off between using knowledge rich, but computationally expensive knowledge resulting from planning like approaches such as minimax search and computationally cheap, but possibly incorrect generalizations, the reinforcement learning agent can automatically learn when to apply planning and when to build a generalizing strategy. This approach would be useful for problem domains where a model is given but which are too large to solve by search. We discuss some difficulties that arise when trying to define costs that are semantically well founded for reinforcement learning problems and present a preliminary algorithm that illustrates the feasibility of the approach.

Introduction

A lot of decision making problem domains can be modeled as a *Markov decision process* (MDP), where an agent interacts with his environment by making observations and taking actions, while trying to attain certain goals. If the model behind such an environment is (completely) known, various planning algorithms can be used to find the optimal solution, at least in theory. In practice these algorithms are often too time-consuming to be practical in many interesting but large domains.

Another way of solving these problems is by performing reinforcement learning (Sutton & Barto 1998). When the environment is structured (for example, states are defined by the existence of objects in the world and relations between those objects), *relational reinforcement learning* (RRL) (Džeroski, De Raedt, & Driessens 2001) can be used. When doing reinforcement learning, the agent does not compute the exact optimal solution through reasoning with a given model, but it will improve its policy or solution strategy through exploration of the environment. Reinforcement learning agents build a utility function which represents the value of every state-action pair. This utility function can be used after the training period, usually without much extra computational cost, to find the best action for every situation. (This in contrast to many planning approaches that generally need to re-compute the path to the goal for every new situation they encounter.) A problem with the RRL approach (a variation of Q-learning (Watkins 1989)) and with many reinforcement learning approaches in general, is that the agent treats his environment as a black-box, ignoring any information he has about the problem domain. This results in the need for a large amount of exploration experience before the agent is able to find the correct, or even an appropriately accurate approximation of the utility function.

For large environments for which an accurate model is given, but that are too large to make planning or search practical, the combination learning and planning could yield substantial benefits. In this paper we will discuss one approach that combines the two. The intuition behind our approach is that the agent *learns* when it needs to use planning, and when it can build a generalized policy such as those resulting from reinforcement learning. To prevent the agent solving the task solely through planning (which is guaranteed to find an optimal solution), we *penalize* the use of planning through a cost-function that reflects the added computational effort the agent needs to make when opting to plan instead of generalize. This combination of using models and experience could yield substantial benefits in applications where a model is given but which are too large to solve through search, or when a strong time-constraint is imposed by the environment during the decision making phase ((McMillen & Veloso 2007)).

The rest of this paper is structured as follows: in the next section we will elaborate on the motivation behind our suggested approach. This is followed by a section in which related work is discussed. The section after explains the approach we suggest, the difficulties which need to be considered and a more pragmatic variation which will be evaluated and discussed in more detail in the section following it.

Motivation

A standard *decision making problem* can be defined as follows:

Given an MDP $< S, A, R, \delta >$, where S is the set of possible states, A the set of available actions, $R : S \rightarrow \mathbb{R}$ a reward function, and $\delta(s, a, s') : S \times A \times S \rightarrow [0, 1]$ the transition function that expresses the probability that performing action a in state s leads to state s', and a discount

Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

factor γ which indicates the relative value of future versus immediate rewards,

Find the policy $\pi : S \times A \rightarrow [0,1]$ indicating the probability with which a action will be selected in any given state, that results in the highest expected (discounted and cumulative) reward:

$$\mathbb{E}\left[\sum_{s_j \in S; i} \gamma^i d_i^\pi(s_j) R(s_j)\right]$$

where $d_i^\pi(.)$ indicates the probability distribution over states encountered at time step i when following policy π.

When a complete model of the MDP is given to the agent, the agent can use a planning algorithm (for example a dynamic programming approach) to construct an optimal strategy for the domain, i.e., a mapping from every possible state to the action which will give the maximal expected reward. However, if the state-space is large, these approaches become too time-consuming to be of practical use. Consider for example the game of chess, where in theory it is possible to find a perfect strategy using the minimax algorithm, but where this approach is not feasible in practice.

Another approach is *learning* and building a generalized strategy through experience: if a state-action-pair is explored, give it a utility value which indicates whether a reward is received and what the value of the best (according to current knowledge of the agent) action in the next state is. This reinforcement learning algorithm, *Q-learning* (Watkins 1989), is guaranteed to converge to the actual utility function, given enough training time. In structured domains one can use a variation of this, the RRL algorithm (Džeroski, De Raedt, & Driessens 2001), to learn a first-order generalization of the utility function. In RRL-TG (Driessens, Ramon, & Blockeel 2001), the agent performs tests on the states of the environment and the chosen actions and uses a relational regression tree to partition the state-space according to the results of these tests. Through exploration, the agent learns which tests are relevant for an accurate approximation of the utility function (for example, for the game of chess, a test '*Does this move put my opponent in checkmate?*' would be a highly significant test). An advantage of this approach is that it learns a generalization of the utility function, which can be readily used after a training or exploration phase. A problem with this standard RRL approach is however, that, while it allows the introduction of some background knowledge about the domain through the introduction of a task specific language bias for the tree building algorithm, it neglects given information about the model of the task, and that it takes a lot of valuable training time to compensate for this.

For many domains, one would like to combine the advantages of both these approaches: use the model to compute the optimal solution unless this is too time-consuming, and learn a generalized utility function for the situations where planning is impractical. This is comparable to, for example, the way a human chess-expert plays the game. At the start of the game, an expert player will rely on his generalized knowledge about openings and their responses (built from experience), while further into the game, when the number of pieces and possible moves goes down, the player will often think a number of steps ahead (i.e. using search and adversarial-planning) to choose his or her actions.

The combined use of models and experience, particularly useful for domains where an accurate model of the underlying MDP is given, but which is too complex to solve through search, is underdeveloped. To this extend, we propose to automatically trade off computation (i.e. search) and generalization through the use of a cost function, that defines the cost of choosing an action in a given state. In this respect, we characterize the cost function c as: $c(\pi, s) : (S \times A) \times S \rightarrow \mathbb{R}$.

The task to solve is therefor transformed into:

Given an MDP $< S, A, R, \delta >$, a discount factor γ and a *cost function* $c : (S \times A) \times S \rightarrow \mathbb{R}$ which maps policies to the cost they incur in a state,

Find the policy π which gives the highest expected (again discounted and cumulative) *net return*:

$$\mathbb{E}\left[\sum_{s_j \in S; i} \gamma^i d_i^\pi(s_j)(R(s_j) - c(\pi, s_j))\right]$$

In the suggested approach, which is based on RRL-TG, we supply extra available tests on the state-action space to the learning agent (in the form of computable background knowledge) that correspond to the knowledge used by a planning approach. An example of such a test for a two-player game would be '*If the agent performs this move, for any possible response of the opponent, can we win the game in the next move ?*', which allows the agent to make a distinction between the utility value of the different state-action pairs by means of a minimax-type approach. Due to the computational cost, the agent should only use these tests if another test, that is just as informative but computationally cheaper, is not available. To this extend, we introduce a *cost function* on tests, which gives an indication of the computational complexity of the test, and use a *cost-sensitive* regression-tree algorithm. We will discuss this approach in more detail later.

Related Work

There are other approaches which combine reinforcement learning with planning, and most of them follow the approach of the TD-leaf algorithm (Baxter, Tridgell, & Weaver 2000) for two-player games. This is a minimax-based algorithm: it learns a utility function (by using the TD algorithm (Tesauro 1995)), but instead of it using only for the possible actions of the current state to decide on the best move, it generates a tree of all possible sequences of moves up to a certain depth and uses the utility function of the resulting leaf-nodes to find the action which gives the highest expected reward. This algorithm combines planning and reinforcement learning in a more direct way, but the advantage of our approach is that the agent can learn only to use planning when necessary. In a way, our approach is related to *quiescence search* where the search depth (and thus

the amount of planning knowledge) is increased in those branches of the tree where the agent thinks it is necessary.

Also related is the work on symbolic dynamic programming that solves relational MDPs through the use of a model at the abstract, i.e. non-instantiated level. This approach was first proposed by Boutilier et al. (Boutilier, Reiter, & Price 2001) using a situation calculus language. Later, it was implemented as a working system by Kersting et al. (Kersting, van Otterlo, & De Raedt 2004) using a probabilistic STRIPS-like formalism. The ReBel algorithm introduced in this work is able to reason backwards from the goal: first it looks for all abstract states from which the goal can be reached, then the abstract states from which the first set can be reached, and so on. This works well for simple and small domains, and returns an optimal generalized strategy, but does not scale up to larger domains. In later work by Sanner and Boutilier (Sanner & Boutilier 2005), approximate linear programming techniques and first order basis function were used to approximate the value-function instead of having to explicitly represent all necessary abstract states. Although this approach presented impressive results, we believe there could be substantial gain in letting the agent automatically trade-off planning and learning.

There is quite some literature on cost-sensitivity and learning, but most of this concentrates on supervised classification tasks, for example (Domingos 1999). Some related material exists in the field of sensor planning (Koenig & Liu 2000), but to the best of our knowledge, it is limited in the field of reinforcement learning. The lack of cost-sensitive reinforcement learning algorithms might be due to some complications which will be elaborated on in the next section.

Combining Planning and Learning using Costs

In this section we will first discuss how a cost-sensitive RRL-TG algorithm should behave according to our intuition. As explained in the previous section, the agent has a set of different *tests* available that can be used on its environment. Using these tests the agent should be able to decide which action to take in the given situation. Every test has a cost related to it which indicates its computational requirements. In the cost-sensitive classification literature, the cost of a test is expressed in the same terms as the misclassification costs, which give the cost a clear semantical interpretation: a test is useful if its cost does not outweigh the decrease in expected misclassification cost. We would like to give similar semantics to the cost-function in our cost-sensitive RRL-TG algorithm, replacing misclassification costs by differences in the obtained (cumulative and discounted) rewards. This would result in the following approach: The agent finds itself in a state $s \in s_0$ for which, without other tests, he would perform action a_0 with expected utility u_0. Consider a test, with cost c which would enable the agent to distinguish between two separate parts of the state space, s_1 and s_2 ($s_1 \cup s_2 = s_0$), with corresponding actions a_1 and a_2 and utilities u_1 and u_2. As the agent uses more information, both u_1 and u_2 will be better than u_0, on average. If the increase of total expected utility outweighs the cost of the test, i.e. when

$$c \leq p(s_1)u_1 + p(s_2)u_2 - u_0$$

where $p(s_i)$ represents the probability that a state $s \in s_0$ belongs to s_i, then the test should be used[1].

There are some problems with this approach, however. To learn whether the increase of utility is worth the cost of a test, the test must be used often enough to enable accurate estimates of its value. If a lot of costly tests are to be considered, all of these should be experimented with during the exploration phase. This implies a large cost for learning a good strategy that will have to be taken into account on any exploration-exploitation strategies.

The advantage is that in the end a generalized utility function is learned which, when used, only uses a cost which is accounted for by an increase of average reward. The suggested approach would be specifically useful in systems with long exploitation phases and a good generalization of a low-cost, high-return policy is more important than time used while training (e.g. be a system which is trained in the factory, where the learned policy is copied and sold to a large group of customers).

Another (and in our case more acute) problem is that the RRL-TG algorithm does not directly map a state to the appropriate action, but for the current state, computes the utility value for every possible action in that state. This means that, when using a cost-sensitive approach, the total cost of a decision depends largely on the number of available actions. This defeats our intuitive interpretation of a clearly interpretable cost function when using the current implementation of RRL.

This latter consideration led us to a more pragmatic preliminary implementation. This implementation will be discussed in the next section.

A Preliminary Approach

Our algorithm is a variation of the RRL-TG algorithm (Driessens, Ramon, & Blockeel 2001). The RRL-TG algorithm uses an incremental, first-order logic, regression tree learner named TG. The TG algorithm starts with a tree consisting of a single leaf. During training, all encountered state-action pairs are gathered in this leaf, together with their estimated utility value. If a test is found which significantly decreases the variance of the average utility value of the encountered examples, the leaf is replaced by a node containing this test, with two leaf nodes as branches (each containing those examples for which the test was either positive or negative). When the training or exploration phase is over and a regression tree is learned (a simple example of such a tree for the connect-4 game can be seen in Fig. 1 – the meaning of the tests in the nodes will be explained later), the *exploitation* of the learned utilities starts. When the agent needs to select an action, all actions available in its current state are considered. The test in the root node is performed on every possible state-action pair; if it tests positive, the

[1] In fact, given the problem definition above, the definition of $p(.)$ should incorporate the shift in the probability distribution over the states in s_0 caused by the change in the resulting policy π

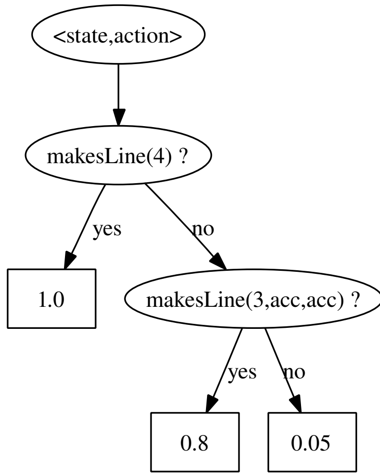

Figure 1: An example of a regression tree such as used by RRL-TG

algorithm goes on to the positive branch, otherwise to the negative branch. When a leaf node is reached the average utility of the examples stored there will be returned as the estimated utility. The agent then performs that action which has the highest estimated utility.

Our variation of this algorithm is that instead of judging the worth of a test solely by the decrease of the variance of the utility of the examples stored in that node, we judge it by the decrease of the standard deviation of the utility, minus the cost of the test. The reasoning behind this is that a useful test reduces the average prediction error by some amount larger than the cost of the test.

The experiments were conducted in the abstract board-game 'connect-four'. The game consists of a grid of 7 columns, 6 places high, in which the players can drop markers (see Fig 2). The first player to get four of his markers in

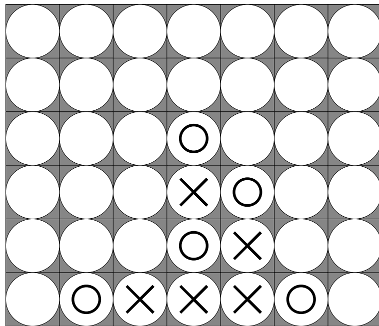

Figure 2: A Connect-4 game in progress

a row (horizontal, vertical or diagonal), wins the game. The rewards used in the game are $+1$ for a won game, -1 for a lost game, and 0 for a drawn game. Our RRL-agent plays against a rather weak opponent, who uses minimax with a depth of 2 (he is able to spot a way in which to win in two moves, and will not play a move which allows the agent to win in two moves (if possible), and plays a random move otherwise).

The available tests in the agent's language bias are some cost-free tests of the form '*Does the move make* 3 *in a row where the two points next to the row are accessible?*'. This is a very limited set of tests.

The set of tests was extended by a number of planning-tests, which did have a cost corresponding to the computational cost. These tests were inspired by the minimax search approach and took the form '*After performing this move in this situation, for any possible move of the opponent, do we have a guaranteed win in n moves?*'. We used a cost of $c \times 49^{(n-1)}$, with c a constant indicating how important the cost function is. The cost function used follows from the fact that we compute n levels of moves of both the opponent and the agent, and both have (in most situations) a choice of 7 moves. The complete set of available tests can be found in Fig 3. Here *End1* and *End2* refer to the points in the grid at both ends of the line, and are either *empty* (there is no marker at the endpoint of the line, *accessible* (empty and the spot below is not empty, such that in the next move a marker can be played here), *o* to indicate there is a marker of the opponent in this spot or *null*, meaning that the line ends at the border of the playing grid. *ForcedWin(n)* means that a win is guaranteed in at most n moves, *forcedLose(n)* indicates that after this move, the opponent has a forced win in n moves.

test	level	cost
makesLine(4)	0	0
makesLine(3,End1,End2)	0	0
makesLine(2,End1,End2)	0	0
forcedWin(1)	1	c
forcedWin(2)	2	49 $\times c$
forcedWin(3)	3	2401 $\times c$
forcedLose(1)	1	c
forcedLose(2)	2	49 $\times c$
forcedLose(3)	3	2401 $\times c$

Figure 3: Available tests in the connect-4 experiment

In the experiments we used five different setting for the cost constant c, $c \in \{0.0; 0.0001; 0.001; 0.01; 1.0\}$. As was explained in the previous section, it is difficult to implement a semantically well founded cost function for these tests in our current setup. This is why we tested a wide range of cost-values and report on the different behaviors that arise.

We let the algorithm train for 1000 games and then let it play 100 test-games using the learned regression tree. We counted the number of won games and the number of tests of each level which were used in the entire tree (level 0 indicating the cost-free tests). Reported values are averages over 5 runs. The results can be seen in Table 1. From this table we see that as the costs decrease, the algorithm will indeed use higher level tests in the tree, as was expected. The algorithm did not learn a prefect tree for the $c = 0$ case, and only 22 % of the games were won - while the agent could, in theory, perform minimax to a depth of 3, which is better than the opponent. Given more time and more practice games, the agent might improve his strategy. This points to an important future improvement: supply the algorithm

c	% won	level 0	level 1	level 2	level 3
1.0	12.0	10.8	0.0	0.0	0.0
0.01	11.6	11.0	0.4	0.2	0.0
0.001	11.2	10.8	0.8	1.6	0.0
0.0001	24.5	10.0	0.4	0.8	2.0
0.0	21.8	11.4	0.4	1.2	2.0

Table 1: Results of the connect-4 experiment

with an initial, high-cost strategy and let it *revise* this strategy to a lower cost, without losing too much accuracy (Ramon, Driessens, & Croonenborghs 2007). From the Table 1 we see that there is a great increase of performance, and of the use of higher-cost tests, between values $c = 0.001$ and $c = 0.0001$. This indicates that a value between these two should be appropriate for this domain. Indeed, for higher values the performance is not significantly worse than for $c = 0.001$, neither is the performance for the cost-free version better than the performance with $c = 0.0001$.

Conclusions and Future Work

In this paper we present a cost-sensitive approach to value-function approximation. Introducing computational costs for adding planning knowledge into a relational reinforcement learning algorithm could present an intuitive approach to automatically trade-off planning and generalization in large but structured decision making problem-domains where a model of the underlying MDP is given.

We illustrated the feasibility of this approach through an empirical evaluation of a prototype algorithm using the connect-four board game.

Cost-sensitive reinforcement learning can be useful for other things than trading off planning and generalization. Imagine for example a robot in an environment which, using the robot's standard sensors, is only partially observable. The robot has the ability of using a number of extra sensors which give additional information about the exact state of the environment (and itself) but these sensors use more, and often precious, power. The robot could learn to only use these costly sensors when having more information is crucial for its task.

In future work, we intend to expand on the presented semantics for the costs connected to the added background knowledge and implement a system that uses these well-defined costs. This will however also require changes to the current representational format of policies as they are learned by the RRL system. To this extend, we will require a direct translation of the current state into the action that needs to be performed, without the necessity of listing (and testing) all actions that are available in that state.

One important driving point for the line of research we are pursuing is the option to provide our reinforcement learning agent with an initial, high-cost policy that uses computationally complex knowledge and allowing the agent to revise this strategy into a computationally cheaper but nearly equally (or possibly even better) performing policy.

References

Baxter, J.; Tridgell, A.; and Weaver, L. 2000. Learning to play chess using temporal differences. *Mach. Learn.* 40(3):243–263.

Boutilier, C.; Reiter, R.; and Price, B. 2001. Symbolic dynamic programming for first-order MDPs. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 690–700.

Domingos, P. 1999. Metacost: A general method for making classifiers cost-sensitive. In *Proceedings of the 5th International Conference on Knowledge Discovery and Data Mining*, 155–164.

Driessens, K.; Ramon, J.; and Blockeel, H. 2001. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In De Raedt, L., and Flach, P., eds., *Proceedings of the 12th European Conference on Machine Learning*, volume 2167 of *Lecture Notes in Artificial Intelligence*, 97–108. Springer-Verlag.

Džeroski, S.; De Raedt, L.; and Driessens, K. 2001. Relational reinforcement learning. *Machine Learning* 43:7–52.

Kersting, K.; van Otterlo, M.; and De Raedt, L. 2004. Bellman goes relational. In *Proceedings of the Twenty-First International Conference on Machine Learning (ICML-2004)*, 465–472.

Koenig, S., and Liu, Y. 2000. Sensor Planning with Non-linear Utility Functions. In *ECP '99: Proceedings of the 5th European Conference on Planning*, 265–277.

McMillen, C., and Veloso, M. 2007. Thresholded Rewards: Acting Optimally in Timed, Zero-Sum Games. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07) (to appear)*.

Ramon, J.; Driessens, K.; and Croonenborghs, T. 2007. Transfer learning in reinforcement learning problems through partial policy recycling. In *Proceedings of European Conference on Machine Learning, Warsaw, Poland (to appear)*.

Sanner, S., and Boutilier, C. 2005. Approximate linear programming for first-order MDPs. In *Proceedings of the 21st conference on Uncertainty in AI (UAI)*.

Sutton, R., and Barto, A. 1998. *Reinforcement Learning: An Introduction*. Cambridge, MA: The MIT Press.

Tesauro, G. 1995. Temporal difference learning and TD-Gammon. *Communications of the ACM* 38(3):58–67.

Watkins, C. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation, King's College, Cambridge.