Reinforcement Learning as Heuristic for Action-Rule Preferences

Joost Broekens, Koen Hindriks, Pascal Wiggers

Man-Machine Interaction department (MMI)
Delft University of Technology

Abstract. A common action selection mechanism used in agent-oriented programming is to base action selection on a set of rules. Since rules need not be mutually exclusive, agents are often underspecified. This means that the decision-making of such agents leaves room for multiple choices of actions. Underspecification implies there is potential for improvement or optimalization of the agent's behavior. Such optimalization, however, is not always naturally coded using BDI-like agent concepts. In this paper, we propose an approach to exploit this potential for improvement using reinforcement learning. This approach is based on learning rule priorities to solve the rule-selection problem, and we show that using this approach the behavior of an agent is significantly improved. Key here is the use of a state representation that combines the set of rules of the agent with a domain-independent heuristic based on the number of active goals. Our experiments show that this provides a useful generic base for learning while avoiding the state-explosion problem or overfitting.

Categories and subject descriptors: I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents*

General terms: Agent programming languages; Robotics; AI; Methodologies and Languages

Keywords: Agent-oriented programming, rule preferences, reinforcement learning

1 Introduction

Agent platforms, whether agent programming languages or architectures, that are rule-based and use rules to generate the actions that an agent performs introduce the problem of how to select rules that generate the most effective choice of action. Such agent programming languages and architectures are based on concepts such as rules, beliefs, and goals to generate agent behavior. Here, rules specify the agent's behavior. A planning or reasoning engine tries to resolve all rules by matching the conditions and actions with the current mental state of the agent. Multiple instantiations of each rule can therefore be possible. An agent can select any one of these instantiations, resulting in a particular action. So,

the rule-selection problem is analogous to but different from the action-selection problem [13]. Rule selection is about which uninstantiated rule to chose; action selection, in the context of rule-based agent frameworks, is about which instantiated rule to chose. In this paper, when we refer to rule we mean uninstantiated rule, i.e, rules that still contain free variables.

Rule-based agent languages or architectures typically underspecify the behavior of an agent, leaving room for multiple choices of actions. The reason is that multiple rules are applicable in a particular situation and, as a result, multiple actions may be selected by the agent to perform next. In practice, it is often hard to specify rule conditions that are mutually exclusive. Moreover, doing so is undesirable as the BDI concepts used to develop agents often are not the most suitable for optimizing agent behavior. An alternative approach is to optimize agent behavior based on learning techniques.

In this paper we address the following question: how to automatically prioritize rules in such a way that the prioritization reflects the utility of a rule given a certain goal. Our aim is a generic approach to learning such preferences, that can be integrated in rule-based agent languages or architectures. The overall goal is to optimize the agent's behavior given a predefined set of rules by an agent programmer, but our approach can also be used by agent programmers to gain insight into the rule preferences and use these to further specify the agent program. As such, we focus on a useful heuristic for rule preferences. We have chosen reinforcement learning (RL) as heuristic as it can cope with delayed rewards and state dependency. These are important aspects in agent behavior as getting to a goal state typically involves a chain of multiple actions, and rules can have different utility depending on the state of the agent/environment.

We present experimental evidence that reinforcement learning can be used to learn rule priorities that can subsequently be used for rule selection. This heuristic for rule priorities works very well, and results in sometimes optimal agent behavior. We demonstrate this with a set of experiments using the Goal agent programming language [5]. Key in our approach is that the RL mechanism uses a state representation based on a combination of the set of rules of the agent and the number of active goals. Our state representation is as abstract as possible while still being a useful base for learning. We take this approach for two main reasons: (1) we aim for a generic learning mechanism; RL should be a useful addition to all programs, and the programmer should not be bothered by the state representation or state-space explosions; (2) an abstract state helps generalization of the learning result as a concrete state representation runs the risk of over fitting on a particular problem instance.

It is important to immediately explain one aspect of our approach that is different from the usual setup for reinforcement learning. In reinforcement learning it is common to learn the *action* that has to be selected from a set of possible actions. In our approach, however, we will apply reinforcement learning to select an *uninstantiated rule* from a set of rules in an agent program. An uninstantiated rule (called *action rule* in Goal, see also Listing 1) is a generic rule defined by the agent programmer.

```
if goal(tower([X|T])), not(bel(T=[])) then move(X,table)
```

is an example of such a rule. We refer to an instantiated rule as a completely resolved (grounded) version of an action rule generated by the reasoning engine responsible for matching rules to the agent's current state. This also means that the action in an instantiated rule may be selected for execution, as the conditions of the rule have been verified by the engine.

```
if goal(tower([a,b])), not(bel([b]=[])) then move(a,table)
```

is an example of an instantiated rule and the action `move(a,table)` is the corresponding action that may be selected. One instantiated rule thus is the equivalent of one action (as it is completely filled in). Many different instantiated rules may be derived from one and the same program rule, depending on the state. An uninstantiated rule is more generic as it defines many possible actions. We focus on learning preferences for uninstantiated rules.

The paper is organized as follows. Section 2 discusses some related work and discusses how our approach differs from earlier work. In Section 3 we briefly introduce the agent language GOAL and use it to illustrate the rule selection problem. Section 4 presents our approach to this problem based on reinforcement learning and presents an extension of GOAL with a reinforcement learning mechanism. In Section 5 experimental results are presented that show the effectiveness of this mechanism. Finally, Section 6 concludes the paper and discusses future work.

2 Related Work

There is almost no work on incorporating learning mechanisms into agent programming. More generally, BDI agents typically lack learning capabilities to modify their behavior [1], although several related approaches do exist.

With regards to related work, several studies attempt to learn rule sets that produce a policy for solving a problem in a particular domain. Key in these approaches is that the rules themselves are learned, or more specific, rule instantiations are generated and evaluated with respect to a utility function. The best performing rule instantiations are kept and result in a policy for the agent. The evaluation mechanism can be different, for example genetic programming [9] or supervised machine learning [7]. In any case, the main difference is that our approach tries to learn rule preferences, i.e., a priority for pre-existing rules given that multiple rules can be active, while the previously mentioned approaches try to learn rule instantiations that solve a problem.

Other studies attempt to learn rule preferences like we do. However, these approaches are based on learning preferences for instantiated rules [10][4], not preferences for the uninstantiated, generic, rules. Further, the state used for learning is often represented in a much more detailed way [10][4]. Finally, as the state representation strongly depends on the environment, the use of learning mechanisms often involves effort and understanding of the programmer [10].

Reinforcement learning has recently been added to cognitive architectures such as Soar [8] and Act-R [2]. In various respects these cognitive architectures are related to agent programming and architectures. They use similar concepts to generate behavior, using mental constructs such as knowledge, beliefs and goals, are also based on an sense-plan-act cycle, and generate behavior using these mental constructs as input for a reasoning- or planning-based interpreter. Most importantly, cognitive architectures typically are rule-based, and therefore also need to solve the rule (and action) selection problem. For example, Soar-RL has been explicitly used to study action selection in the context of RL [6]. Soar-RL [10] is the approach that comes closest to ours in the sense that it uses a similar reinforcement learning mechanism (Sarsa) to learn rule preferences. As explained above, the key difference is that we attempt to learn uninstantiated rule preferences, while Soar-RL learns preferences for instantiated rules [10]. Another key difference is that we use an abstract rule-activity based state representation complemented with a 'goals left to fulfill' counter, as explained in section 4.2.

Finally, [1] present a learning technique based on decision trees to learn the context conditions of plan rules. The focus of their work is to make agents adaptive in order to avoid failures. Learning a context condition refers to learning when to select a particular plan/action, while learning a rule preference refers to attaching a value to a particular plan/action. Our work is thus complementary in the sense that we do not learn context conditions, but instead propose a learning mechanism that is able to guide the rule selection mechanism itself.

3 The Agent Language GOAL

In this Section we briefly present the agent programming language GOAL and use it to illustrate the rule selection problem in agent languages and architectures. For a more extensive discussion of GOAL we refer the reader to [5]. The approach to the rule selection problem introduced in this paper is not specific to GOAL and may be applied to other similar BDI-based platforms. As our approach involves a domain-independent heuristic based on counting the number of goals that need to be achieved, the language GOAL is however particularly suitable to illustrate the approach as declarative goals are a key concept in the language.

GOAL, for Goal-Oriented Agent Language, is a programming language for programming *rational agents*. GOAL agents derive their choice of action from their beliefs and goals. A GOAL agent program consists of five sections: (1) a knowledge section, called the *knowledge base*, (2) a set of beliefs, collectively called the *belief base*, (3) a set of *declarative* goals, called the *goal base*, (4) a *program section* which consists of a set of *action rules*, and (5) an *action specification section* that consists of a specification of the pre- and postconditions of actions of the agent. Listing 1 presents an example GOAL agent that manipulates blocks on a table.

The knowledge, beliefs and goals of a *GOAL* agent are represented using a knowledge representation language. Together, these make up the mental state of

```
main stackBuilder {
 knowledge{
   block(a), block(b), block(c).
   clear(table).
   clear(X) :- block(X), not(on(Y,X)).
   tower([X]) :- on(X,table).
   tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
 }
 beliefs{
   on(a,table), on(b,table), on(c,a), on(d,c).
 }
 goals{
   on(a,d), on(b,c), on(c,table), on(d,b).
 }
 program{
   if goal(tower([X|T])),
     bel((T=[Y|T1], tower(T)); (T=[], Y=table))
     then move(X,Y).
   if goal(tower([X|T])), not(bel(T=[]))
     then move(X,table).
 }
 actionspec{
   move(X,Y) {
     pre{ clear(X), clear(Y), on(X,Z) }
     post{ not(on(X,Z)), on(X,Y) }
   }
 }
}
```

Table 1. Agent for Solving a Blocks World Problem

an agent. Here, we use *Prolog* to represent mental states. An agent's knowledge represents general conceptual and domain knowledge, and does not change. An example is the definition of the concept `tower` in Listing 1. In contrast, the beliefs of an agent represent the *current state of affairs* in the environment of an agent. By performing actions and possibly by events in the environment, the environment changes, and it is up to the agent to make sure its beliefs stay up to date. Finally, the goals of an agent represent *what* the agent wants the environment to be like. For example, the agent of Listing 1 wants to realise a state where block `a` is on top of block `b`. Goals are to be interpreted as *achievement goals*, that is as a goal the agent wants to achieve at some future moment in time and does not believe to be the case yet. This requirement is implemented by imposing a rationality constraint such that any goal in the goal base must not believed to be the case. Upon achieving the *complete* goal, an agent will drop the goal. The agent in Listing 1 will drop the goal `on(a,b), on(b,c), on(c,table)` if this configuration of blocks has been achieved, and only if the *complete* configuration has been achieved.

As GOAL agents derive their choice of action from their knowledge, beliefs and goals, they need a way to inspect their mental state. GOAL agents do so by means of *mental state conditions*. Mental state conditions are Boolean combinations of so-called basic *mental atoms* of the form **bel**(ϕ) or **goal**(ϕ). For example, **bel**`(tower([c,a]))` is a mental state condition which is true in the initial mental state specified in the agent program of Listing 1.

A GOAL agent uses so-called *action rules* to generate possible actions it may select for execution. This provides for a rule-based action selection mechanism, where rules are of the form **if** ψ **then** $a(\boldsymbol{t})$ with ψ a mental state condition and $a(\boldsymbol{t})$ an action. A mental state condition part of an action rule thus determines the states in which the action $a(\boldsymbol{t})$ may be executed. Action rules are located in the program section of a GOAL agent. The first action rule in this section of our example agent generates so-called *constructive moves*, whereas the second rule generates actions to move a *misplaced* block to the table. Informally, the first rule reads as follows: if the agent wants to construct a tower with `X` on top of a tower that has `Y` on top and the agent believes that the tower with `Y` on top already exists, or believes `Y` should be equal to the table, then it may consider moving `X` on top of `Y`; in this case the move would put the block `Y` *in position*, and it will never have to be moved again. The second rule reads as follows: if the agent finds that a block is misplaced, i.e. believes it to be in a position that does not match the (achievement) goal condition, then it may consider moving the block to the table. These rules code a strategy for solving blocks world problems that can be proven to always achieve a goal configuration. As such, they already specify a *correct* strategy for solving blocks world problems. However, they do not necessarily determine a unique choice of action. For example, the agent in Listing 1 may either move block `d` on top of block `b` using the first action rule, or move the same block to the table using the second action rule. In such a case, a GOAL agent will nondeterministically select either of these actions. It is important for our purposes to note here that the choice of rule is at stake here, and not a particular *instantiation* of a rule. Moreover, as in the blocks world it is a good strategy to prefer making constructive moves rather than other types of moves, the behavior of the agent can be improved by preferring the application of the first rule over the second whenever both are applicable. It is exactly this type of preference that we aim to learn automatically.

Finally, to complete our discussion of GOAL agents, actions are specified in the action specification section of such an agent using a STRIPS-like specification. When the preconditions of the action are true, the action is executed and the agent updates its beliefs (and subsequently its goals) based on the postcondition. Details can be found in [5].

As illustrated by our simple example agent for the blocks world, rule-based agent programs or architectures may leave room for applying multiple rules, and, as a consequence, for selecting multiple actions for execution. Rule-based agents thus typically are *underspecified*. Such underspecification is perfectly fine, as long as the agent achieves its goals, but may also indicate there is room for improvement of the agent's behavior (though not necessarily so). The problem of optimizing the behavior of a rule-based agent thus can be summarized as follows, and consists of two components: First, solving a particular task efficiently depends on using the appropriate rule to produce actions (the rule selection problem) and, second, to select one of these actions for execution (the action selection problem). The latter problem is actually identical to selecting an *instantiated* rule where all variables have been grounded, as instantiated rules

that are applicable yield unique actions that may be executed. Uninstantiated rules only yield *action templates* that need to be instantiated before they can be executed.

In this paper we explore a generic and fully automated approach to this optimization problem based on learning, and we propose to use reinforcement learning. Although reinforcement learning is typically applied to solve the action selection problem, here instead we propose to use this learning technique to (partially) solve the rule selection problem. The reason is that we want to incorporate a *generic* learning technique into a rule-based agent that does not require domain-specific knowledge to be inserted by a programmer. As we will show below, applying learning to the rule selection problem in combination with a domain-independent heuristic based on the number of goals still to be achieved provides just such a mechanism.

4 Learning to Solve the Rule Selection Problem

In this Section, we first briefly review some of the basic concepts of reinforcement learning, and then introduce our approach to the rule selection problem based on learning and discuss how we apply reinforcement learning to this problem. We use the agent language GOAL to illustrate and motivate our choices.

4.1 Reinforcement Learning

Reinforcement Learning is a mechanism that enables machines to learn solutions to problems based on experience. The main idea is that by specifying *what* to learn, RL will figure out *how* to do it. An approach based on reinforcement learning assumes there is an environment with a set of possible states, S, a reward function $R(S)$ that defines the reward the agent receives for each state in the environment, and a set of actions A that enable to effect changes to the environment (or an agent in that environment) and move the environment from one state to another according to the state transition function $T(S, A) \rightarrow S$. An RL mechanism then learns a value function, $V(S, A)$, that maps actions in states to values of those actions in that state. It does so by propagating back the reinforcement (reward) received in later states to earlier states and actions, called *value propagation*. RL should do this in such a way that the result of always picking the action with the highest value will lead to the best solution to the problem (the best sequence of actions to solve the problem is the sequence with the highest cumulative reward). Therefore, RL is especially suited for problems in which the solution follows only after a sequence of actions and in which the information available for learning takes the form of a reward (e.g. pass/fail or some utility value).

In order for RL to learn a good value function, it must explore the state space sufficiently, by more or less randomly selecting actions. Exploration is needed to gather a representative sample of interactions so that the transition function T (in case the model of the world is not known) and the reward function R can be

learned. Based on T and R, the value function V is calculated. After sufficient exploration, the learning agent switches to an exploitation scheme. Now the value function is used to select the action with highest predicted cumulative reward (the action with the highest $V(s, a)$). For more information on RL see [12].

4.2 GOAL-RL

The idea is to use RL to learn values for the rules in an agent program or architecture, so that a priority of rules can be given at any point during the execution of the agent. Here, we use GOAL to illustrate and implement these ideas, and we call this RL-enabled version GOAL-RL. The basic idea of our contribution is that the GOAL interpreter determines which rules are applicable in a state, while RL learns what the values for applying these same rules are in that state. GOAL will then again be responsible for using these values in the context of rule selection. Various selection mechanisms may be used, e.g., selecting the best rule greedy, or selecting a rule based on a Boltzmann distribution, etc. This setup combines the strenghts of a qualitative, logic-based agent platform such as GOAL with the strengths of a learning mechanism such as reinforcement learning.

RL needs a state representation for learning. Unfortunately, using the agent's mental state or the world state, as is typically done in RL, quickly leads to intractably large state spaces and makes the solutions (if they can be learned at all) domain and even problem-instance specific. Still, our goal is to create a domain-independent mechanism that takes the burden of finding a good rule selection mechanism away from the programmer.

We propose the following approach. Instead of starting to train with a state representation that is as close as possible to the actual agent state, and make that representation more abstract in case of state explosion problems, as is common in RL, we start with a representation that is very abstract, while still being meaningful and specific enough to be useful for learning rule preferences. The benefits of this choice are twofold. First, a trained RL model based on such an abstract state and action representations is potentially more suitable for reuse in different domains and problem instances (learning transfer). Second, by using an abstract state our approach is less vulnerable to large state-spaces and the state-space explosion problem, and, consequently, will learn faster.

The state representation we propose is composed of the following two elements. First, our state representation contains the set of rule-activation pairs itself (i.e. the list of rules and whether a rule is applicable or not). However, for many environments this representation does not contain enough information for the RL algorithm to learn a good value function. Essentially, what is missing is information that guides the RL algorithm towards the end goal. A state representation that only keeps track of the set of rules that are and are not applicable does not contain any information about the approriateness of rules in a particular situation. We add such information by including a second element in the state representation: a version of a well-known progress heuristic used also in planning. The heuristic, which is easily implemented in an agent language or architecture that keeps track of the goals of an agent explicitly, is to count the

number of subgoals that still need to be achieved. This is a particularly easy way to compute the so-called *sum cost heuristic* introduced in [3]. Due to its simplicity this heuristic causes almost no overhead in the learning algorithm. This heuristic information is added to the state used by the reinforcement learning mechanism in order to guide the learning. Adding a heuristic like this will keep the mechanism domain independent, but gives useful information to the RL mechanism to differentiate between states.

Even with this heuristic many states differentiated by the agent itself are conflated in the limited number of states used by the reinforcement learner. Such a state space reduction will sometimes prevent the algorithm from finding optimal solutions (as many RL mechanisms, including the one we use, assume a Markovian state transition). It should be noted, though, that we are not aiming for a perfect learning approach that is always able to find optimal solutions. Instead, we aim for an approach that provides two benefits: it is generic and therefore poses no burden on the programmer, and the approach is able to provide a significant improvement of the agent's behavior, even though this may still not be optimal (optimal being the smallest number of steps possible to solve a problem). The approach to learn rule preferences thus should result in significantly better behavior than that generated by agents that do not learn rule preferences. In the remainder of this paper, we will study and demonstrate how well the domain-independent approach is able to improve the behavior of agents acting in different domains.

In more detail, the approach introduced here consists of the following elements. A state s is a combination of the number of subgoals the agent still has to achieve and the set of rule states. A rule state is either 0, 1 or 2, where 0 means the rule is not active, 1 means there is an instantiation of the rule in which the rule's preconditions are true and 2 means there is an instantiation in which also the preconditions for the action the rule proposes are true meaning the rule fires. For example, if a program has a list of 3 rules, of which the last two in the program fire while the agent still has 4 subgoals to achieve, the state equals to $s = 022 : 4$. An action is represented by a hash based on the rule (in our case simply the index of the rule in the program list; so the action uses the same hash as the rule in the rule-activation pairs used for the state). For example, if the agent would execute an action coming from the first rule in the list, the action equals to $a = 0$, indicating that the agent has picked the first rule for action generation. In our setup, the reward function R is simple. It defines a reward $r = 1$ when all goals are met (the goal list is empty) and $r = 0$ otherwise. The current and next state-action pairs (s, a) and (s', a') are used together with the received reward r' as input for the value function V. A transition function T is learned based on the observed state-action pairs (s, a) and (s', a'). The transition function is used to update the value function according to standard RL assumptions, with one exception: the value for a state-action pair (s, a) is updated according to the probabilistically correct estimate of the occurance of (s', a'), not the maximum. In order to construct the probabilities, the agent counts state occurrences, $N(s)$, and uses this count in a standard weighting mechanism. Values of states are

updated as follows:

$$RL(s,a) \leftarrow RL(s,a) + \alpha \cdot (r - RL(s,a)) \tag{1}$$

$$V(s,a) \leftarrow RL(s,a) + \gamma \cdot \sum_i V(s_{a_i}, a_i) \frac{N(s_{a_i}, a_i)}{\sum_j N(s_{a_j}, a_j)} \tag{2}$$

So, a state-action pair (s,a) has a learned reward $RL(s,a)$ and a value $V(s,a)$ that incorporates predicted future reward. $RL(s,a)$ converges to the reward function $R(s,a)$ with a speed proportional to the learning rate α (set to 1 in our experiments). $V(s,a)$ is updated based on $RL(s,a)$ and the weighted average over the values of the next state-action pairs reachable by action $a_{1...i}$ (with a discount factor of γ, set to 0.9 in our experiments). So, we use a standard model-based RL approach [12], with an update function comparable to Sarsa [11].

5 Experiments

In order to assess if rule preferences can be learned using RL with a state representation as described, we have conducted a series of experiments. The goal of these experiments was to find out the sensitivity of our mechanism with respect to (a) the problem domain (we tested two different domains), (b) different problem instantiations within a domain (e.g. random problems), (c) rules used in the agent program (different rule sets fire differently and thus result in both a different state representation and different agent behavior), (d) different goals (a different goal implies a different reward function because $R(s) = 1$ only when all goals are met).

In total we tested 8 different setups. Five setups are in an environment called the *blocks world*, in which the agent has to construct a goal state consisting of a predefined set of stacks of numbered blocks from a start state following standard physics rules (block cannot be removed from underneath other blocks). The agent can grab a block from and drop a block at a particular stack. In principle, it can build infinitely many stacks (the table has no bounds). The agent program lists two rules.

Three setups were in the *logistics domain* in which the agent has to deliver two orders each consisting of two different packages to two clients at different locations. In total there are three locations, with all packages at the starting location and each client at a different location. A location can be reached directly in one action. The agent can load and unload a package as well as goto a different location. The agent program lists five rules.

5.1 Setup

Each experiment consisted of a classic learning experiment in which a training phase of 250 trials (random rule selection) was followed by a exploitation phase of 30 trials (greedy rule selection based on learned values). For each experiment we

present a bar graph showing the average number of actions needed to solve the problem during the training phase (reflecting the goal agent as it would perform on average without learning ability) and during the exploitation phase (reflecting the solution including the trained rule preferences). As a measure of optimality we also show the minimum number of actions needed in one trial to get to a goal state as observed during the first 250 trials (which equals the minimum number of steps to reach a solution, except in Figure 3 as explained later). This number is shown in the bar graphs as reference number for the optimality of the learned solution.

5.2 Blocks world experiments

Five experiments were done using the blocks world. As described in section 3, there are two rules for the *GOAL* agent, one rule designed to correctly stack blocks on goal stacks (constructive rule) and the other designed to put ill-placed blocks on the table (deconstructive move). Given these two rules, it is easy to see (and prove) that given a choice between the constructive and deconstructive move, the constructive move is always as good as the deconstructive one. It involves putting blocks at their correct position. These blocks do not need to be touched anymore. A deconstructive move involves freeing underlying blocks. This might be necessary to solve the problem, but the removed blocks might also need to be moved again from the table to their correct place at a goal stack.

The first experiment is a test, constructed to find out if the *GOAL*-RL agent can learn the correct rule preferences for a fundamental three-blocks problem. In this problem, three blocks need to be put on one stack starting with C, BA ending with the goal stack ABC. The agent should learn a preference for the constructive move, as this allows a solution of the problem in two moves ($B > C$ and $A > BC$), while the deconstructive move needs three ($A > Table$ then $B > C$ and $A > BC$). Indeed, the agent learns this preference, as shown in Figure 1.

The reward in the last experiment comes rather quickly, and the state transitions are provably Markovian, so the positive learning result presented here is not surprising. In the second experiment, we tested if a reward given at a later stage together with a more complex state-space would also give similar results. We constructed a problem of which it is clear that constructive moves are better than deconstructive moves: the inverse-tower problem. Here, the agent is to inverse a tower $IHGFEDCBA$ to $ABCDEFGHI$. Obviously, constructive moves are to be preferred as they build a correct tower, while deconstructive moves only delay building the tower. The rules used by the agent are the same as in the previous experiment. As can be seen in Figure 1, the *GOAL*-RL agent is able to learn the correct rule preferences and thereby produce the optimal solution.

As one of the reasons for choosing an abstract state representation is to find out if this helps learning a solution to multiple problem instances with a problem domain, not just the one trained for, we set up a third experiment based on tower building problem in which the starting configuration is random. This means

that at each trial the agent is confronted with a different starting configuration but always has $ABCDEFGHI$ as goal stack. Being able to learn the correct preferences for the rules in this case involves coping with a large amount of environment states that are mapped to a much smaller amount of rule-based states. We have kept the goal static to be able to interpret the result. If the goal is to build a high tower, constructive moves should be clearly preferred over deconstructive ones. Therefore we know that in this experiment the constructive move is clearly favorite and the learning mechanism should be able to learn this. As shown in Figure 2 the agent can indeed learn to generalize over the training samples and learn rule preferences. Note that if we would have taken a state representation more directly based on the actual world (e.g., the current blocks configuration), this generalization is difficult as each new configuration is a new state, and in RL unseen states cannot be used to predict values (unless a RL mechanism is used that uses some form of state feature extraction). Therefore, this result that shows that our approach is able to optimize rule selection in a generic way.

Up until now, the two rules of the agent are relatively smart. Each rule helps solving the problem, i.e., each rule moves forward towards the goal, as even the deconstructive rule never removes a block from a goal stack. In the next experiment we changed the deconstructive move to one that always enables the agent to remove a block form any stack. This results in a *dumb* tower building agent as it can deconstruct correct towers. For this agent to learn correct preferences, it needs to cope with much longer action sequences before the goal is reached as well as many cycles in the state transitions (e.g., when the agent undoes a constructive move). As shown in Figure 2, left and middle, the agent can learn the correct rule preferences and converge to the optimal solution. This is an important result as it shows that the mechanism can cope with different rule sets solving the same problem, as well as optimize agent behavior given a rule set that is clearly sub-optimal (the *dumb* deconstructive move).

In our last experiment with the blocks world, we evaluated whether the learning mechanism is sensitive to the goal itself. It is based on the inverse tower problem, with one variation: instead of having one high tower as goal, we now have three short towers ABC, DEF, GHI as goal stacks as well as a random starting configuration. This variation thus de-emphasizes the merit of constructive moves for the following reason. In order to solve the problem from any random starting configuration, the agent also has to cope with those situations in which one or two long towers are present at the start. These towers need to be deconstructed. As such, even though constructive moves are never worse than deconstructive moves, deconstructive moves become relatively more valuable. As shown in Figure 2, right, the agent still improves the agent behavior significantly, but is not able to always learn the optimal solution. As such our learning approach provides a useful heuristic for rule preferences. The decrease in learning effectiveness is due to the abstractness of the state representation. In the previous experiments, the agent's RL mechanisms could know where it was building the tower, as the number of active subgoals (incorrectly placed blocks)

decreases with each well-placed block. In this experiment, however, the number of active subgoals does not map to the environment state in the same fashion (all three towers contribute to this number, but it is impossible to deduce the environment state based on the number of subgoals: e.g., the number 6 does not reflect that tower one and two are build and we are busy with tower three). This means that there is more *state-overloading* in the last experiment, more risk at non Markovian state transitions, hence the RL mechanism will perform worse.

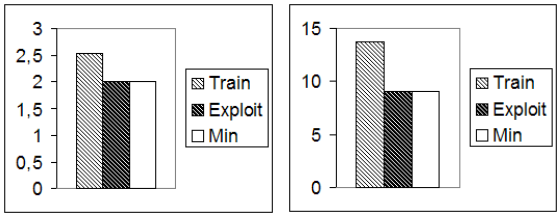

Fig. 1. Left: three-block test. Right: inverse tower.

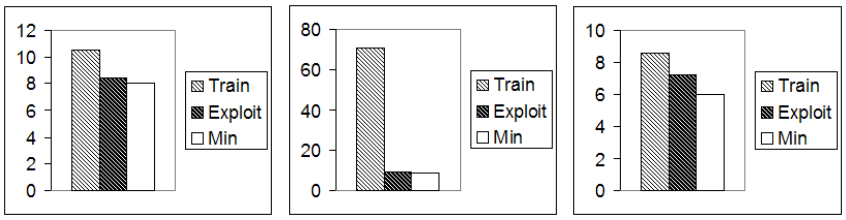

Fig. 2. Left: random start tower. Middle: random start tower dumb. Right random start 3 towers.

5.3 Logistics domain experiments

In this set of three experiments we test the behavior of our mechanism in a different domain. The domain is called the *logistics domain*. As explained above, this domain involves a truck that needs to distribute from a central location two different orders containing two different items to two clients, making it a total of four items to be delivered. A truck can move between three locations (client 1, client 2 and the distribution center). The agent has two goals: deliver order 1, and deliver order 2. It can pick up and drop an item. When two items are delivered, a subgoal is reached. The agent has five rules, two of which handle pickup, one handles dropping, 1 handles moving to a client, and one handles moving to the distribution center.

In the first experiment, we tested if the agent can learn useful preferences in this domain. As Figure 3, left and middle, shows, it can. This suggests that our results are not specific to a single domain.

In the second experiment, we modified the rule that controls moving to clients such that it also allows the truck to move to clients when empty (the *dumb* delivery truck). This mirrors the dumb tower builder in the blocksworld as it significantly increases the average path to the goal state and it introduces much more variation in the observed states (more random moves). As shown in Figure 3, left and middle, the agent can also learn rule preferences that enable it to converge to the optimal solution. We would like to note that the average learning result is better than the minimum result observed during exploration. This shows that the learned rule preferences perform a strategy that is better than any solution tried in the 250 exploration trials. In other words, learning based on rule-based representations can generalize to a better solution than observed during training.

In the last experiment we manipulated a last important factor: the reward function $R(s)$. In the previous two experiments, the agent was positively reinforced when the last item had been delivered. In this experiment, the agent is reinforced when it returns to the distribution center after having delivered the last item. As shown in Figure 3, right, this results in a suboptimal strategy, although still far better a strategy than the standard *GOAL* agent. This shows that the mechanism is influenced by the moment the reward is given, even if from a logical point of view this should not matter. The reason for this is simple (and resembles the one proposed for the slightly worse performance in the last blocksworld experiment). Due to our abstract state representation, the RL mechanism of the agent cannot differentiate between a state in which it just returned to the distribution center after delivering the *last* item of the last order versus the *first* item of the last order. This means that both environment states are mapped to the same RL state. This RL state receives a reward, and therefore returning to the distribution center gets rewarded. As such, the agent emphasizes returning to the distribution center and learns the suboptimal solution in which it picks up an item and brings it to the client as soon as possible in order to get to the center ASAP because that is where the reward is. The best strategy is of course to pick both items for a client and then move to the client. However, as the RL mechanism cannot differentiate between two important states, it cannot learn this solution. This clearly shows a drawback of a too abstract state representation. However, the drawback is relative, as the agent still performs much better than the standard *GOAL* agent, showing that even in this case our mechanism is useful as a rule preference heuristic.

6 Conclusion

In this paper we have focused on the question of how to automatically prioritize rules in an agent program. We have proposed an approach to exploit the potential for improvement in rule-selection using reinforcement learning. This approach

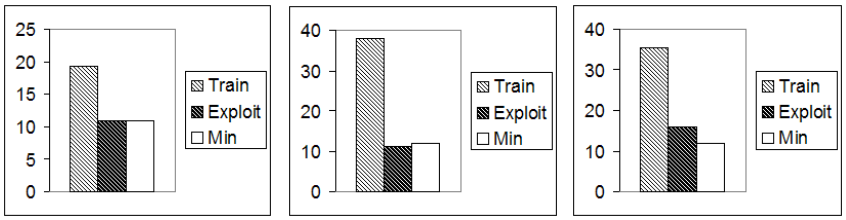

Fig. 3. Left: delivery world. Middle: delivery world dumb. Right: delivery world manipulated $R(s)$

is based on learning state-dependent rule priorities to solve the rule-selection problem, and we have shown that using this approach the behavior of an agent is significantly improved. We demonstrate this with a set of experiments using the *GOAL* agent programming language, extended with a reinforcement learning mechanism. Key in our approach, called *GOAL*-RL, is that the RL mechanism uses a state representation based on a combination of the set of rules of the agent and the number of active goals. This state representation, though very abstract, still provides a useful base for learning. Moreover, this approach has two important benefits: (1) it provides for a generic learning mechanism; RL should be a useful addition to all programs, and the programmer should not be bothered by the state representation or state-space explosions; (2) an abstract state helps generalizing the learning result as a concrete state representation runs the risk of over fitting on a particular problem instance. One of the advantages is that it does not involve the agent programmer or the need to think about state representations, models, rewards and learning mechanisms. In the cases explored in our experiments the approach often finds rule preferences that result in optimal problem solving behavior. In some case the resulting behavior is not optimal, but is still significantly better than the non-learning agent.

Given that we have implemented a very generic, heuristic approach there is still room for further improvement. Two topics are particularly interesting for future research. First, we want to investigate whether adding other domain-independent features and making the state space in this sense more specific may improve the learning even more. Second, we want to investigate whether the use of different learning mechanisms that are better able to cope with non Markovian worlds and state overloading such as methods based on a partially observable Markov assumption (POMDP) will improve the performance.

7 Acknowledgments

This research is supported by the Dutch Technology Foundation STW, applied science division of NWO and the Technology Program of the Ministry of Economic Affairs. It is part of the Pocket Negotiator project with grant number VIVI-project 08075.

References

1. S. Airiau, L. Padham, S. Sardina, and S. Sen. Enhancing adaptation in bdi agents using learning techniques. *International Journal of Agent Technologies and Systems*, 1(2):1–18, 2009.
2. J. R. Anderson and C. Lebiere. *The atomic components of thought.* Lawrence Erlbaum, Mahwah, NY, 1998.
3. B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97*, pages 714–719, 1997.
4. S. Deroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43(1):7–52, 2001.
5. K. V. Hindriks. Programming Rational Agents in GOAL. In *Multi-Agent Programming: Languages, Tools and Applications*, chapter 4, pages 119–157. Springer, 2009.
6. E. Hogewoning, J. Broekens, J. Eggermont, and E. Bovenkamp. Strategies for Affect-Controlled Action-Selection in Soar-RL. In *Nature Inspired Problem-Solving Methods in Knowledge Engineering*, pages 501–510. 2007.
7. R. Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113:125–148, 1999.
8. J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: an architecture for general intelligence. *Artif. Intell.*, 33(1):1–64, 1987.
9. J. Levine and D. Humphreys. Learning action strategies for planning domains using genetic programming. In *Applications of Evolutionary Computing*, pages 50–55. 2003.
10. S. Nason and J. E. Laird. Soar-rl: integrating reinforcement learning with soar. *Cognitive Systems Research*, 6(1):51–59, 2005.
11. G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, Cambridge University Engineering Department., 1994.
12. R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, 1998.
13. T. Tyrrell. *Computational Mechanisms for Action Selection.* Phd, University of Edinburgh, 1993.