Evolving and Transferring Probabilistic Policies for Relational Reinforcement Learning

Martijn van Otterlo [a] Tim De Vuyst [a]

[a] *Katholieke Universiteit Leuven, Celestijnenlaan 200A, Heverlee, Belgium*

Abstract

The last decade reinforcement learning has been lifted to relational (or, first-order) representations of states and actions. In worlds consisting of objects and relations between them, mostly *value-based* techniques have been studied, even though value functions can be ill-defined and moreover, are difficult to learn. We present a novel evolutionary *policy search* algorithm called GAPI. It evolves *probabilistic policies* for (probabilistic) planning problems. We show good results on typical BLOCKS WORLD benchmark problems as well as on a small logistics domain. We show how GAPI can exploit the probabilistic relational policy representation to i) learn *parameterized* policies over abstract goal specifications, ii) abstract simple actions into macro-actions, and iii) to tackle complex problems by learning individual subtasks first.

1 Introduction

The world consists of *objects* and *relations* between them. There are things like apples, books and tables, but also increasingly more conceptual things like clouds, governments, and happiness. It would be hard to imagine an adaptive, generally intelligent system that would not see the world in those terms. The last decade, a large amount of techniques have been developed to solve *sequential decision making problems under uncertainty* in such worlds [18]. Many of these techniques focus the computation on *utility value functions* that can predict how much *reward* could be obtained if a particular strategy is followed. An alternative studied here, is the use of search techniques to find good strategies (or, *policies*) directly from interaction with the environment. We introduce the GAPI system that uses *evolutionary algorithms* (see [11]) to do the actual search.

Outline: In Section 2 we introduce our problem setting, and in Section 3 we describe our novel GAPI system for learning policies. The experiments in Section 4 then show the viability of GAPI on a number of benchmark tasks. After that, we show results of transferring policies to more complex tasks, and we present related experiments in a different domain. Section 5 concludes and outlines directions for further research.

2 Learning to Take Decisions in Relational Worlds

We target the task of *learning* how to take decisions in large, *probabilistic* worlds consisting of *objects* and *relations*. A generic formalization for such problems is the *Markov decision process* (MDP). These, and relevant solution techniques will be briefly described after which we discuss MDPs in a relational setting.

2.1. Solving Markov Decision Processes An MDP is a tuple $M = \langle S, A, T, R \rangle$, where S is a finite set of *states*, A a finite set of *actions*, T a probabilistic *transition function* over $S \times A \times S$, and R a *reward function* that maps states to real values. A *policy* is a function $\pi : S \rightarrow A$ and the *state-action value function* Q^π for π expresses how much cumulative (discounted) reward can be expected when executing action a in a state $s \in S$ and following the actions suggested by π thereafter. The existence of *optimal* policies and value functions is guaranteed for MDPs. For any given MDP the goal is to find an optimal policy. An example of an MDP is a grid maze, where all states (S) correspond to grid positions in the maze, actions (A) are movements (T) between grid positions, and the reward function (R) for each step is -1 except for reaching

the exit of the maze in which case the reward is $+100$. An optimal policy would take one to the exit in the least number of steps possible.

Solution techniques for MDPs: The many algorithms that exist to compute optimal policies can be divided into two groups (e.g. see [15, 18]). The first group assumes that the model of the MDP (i.e. T and R) is known, and that value functions can be computed using (variants of) iterative *dynamic programming* (DP) algorithms such as *value iteration* and *policy iteration*. The resulting value function can be used by a *greedy* policy to select actions accordingly, i.e. $\forall s \in S : \pi(s) = \arg\max_{a \in A} Q(s, a)$. The second group of techniques is called *model-free*, and these compute solutions based on *samples* of interaction with the environment (i.e. the MDP). Such sample-based algorithms are generally denoted *reinforcement learning* (RL), of which Q-learning is a popular example. Within the model-free techniques an important distinction is made between *value-based* and *policy-based* algorithms. Whereas the first learn value functions and derive policies from them, the latter search directly for good policies without explicitly representing value functions. An advantage of policy-based methods is that the policy search space is often much smaller. Either way, in complex environments (where S and A are large) it becomes necessary to apply *generalization* and *approximation* techniques for *representing* and *learning* value functions and policies. Many approaches exist including function approximators such as neural networks, hierarchical task decompositions [14] and efficient data structures (see further [18]).

Evolutionary algorithms for reinforcement learning (EARL): Several policy-based approaches employ *evolutionary computing* to search in the space of policies (see [12] for an overview). Evolutionary techniques are *population-based* search algorithms, mimicking biological evolution, that keep a *population* of solutions to a particular search problem, i.e. in this case consisting of policies for a particular MDP. A well-known evolutionary technique are *genetic algorithms* (see [11]). Each *generation* in the computation takes such a population of which the *individuals* (i.e. its members) are all *evaluated* on the MDP. The evaluation is expressed as a *fitness value* which is naturally computed in terms of the cumulative reward obtained while executing the policy in the MDP. Individuals with a relatively high fitness value are then *selected* to generate *offspring* by *recombination* of individuals. That is, parts of individuals (e.g. action selections for individual states) are interchanged between individuals to generate two new individuals built up from the parts of individuals with high fitness. After recombination, some individuals are then randomly *mutated* slightly (e.g. changing the policy action for some states randomly). Recombination combines useful *building blocks* found in the search space, while mutation introduces variation (e.g. random perturbations in the search process). Popular representational devices for EARL are neural networks and lists of rules.

2.2. MDPs in the Relational Setting Most existing techniques for solving MDPs are based on *atomic* or *propositional* (i.e. *attribute-value*) representations. The last decade, however, techniques have been developed that allow for the solution of MDPs posed in terms of *first-order logical* (FOL) representations. The FOL setting is more general than a propositional setting and is covered in *machine learning* [3].

Relational MDPs: In a *relational* MDP, (RMDP) the state space S is built from a logical alphabet consisting of a set C of *constants* (denoting *objects* in the domain) and a set of predicates $\{P_i/\alpha\}$. Each predicate has an *arity* α denoting the number of arguments of the *relation* it represents. For arity 1 predicates denote *properties* of a particular object, whereas for larger arities a relation between multiple objects can be expressed. In logical terms, the state space consists of all *Herbrand interpretations* of a given alphabet. The action space of an RMDP is built from a separate set of action predicates P_A..

An example RMDP that we will later use in our experiments is the BLOCKS WORLD, a typical benchmark problem for planning. This world contains a finite number of blocks, and a table. A typical task here is to compute or learn action sequences (e.g. a *plan*) that transform an *initial* into a *goal* state. The set of constants in a four-block world is $\{\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3, \mathtt{b}_4, \mathtt{table}\}$, a typical set of predicates is $\{\mathtt{clear}/1, \mathtt{on}/2\}$, and the only action is $\mathtt{move}/2$. The relation $\mathtt{on}(\mathtt{b}_1, \mathtt{b}_2)$ expresses that block $\mathtt{b}_1$ stands on block $\mathtt{b}_2$, and $\mathtt{clear}(\mathtt{b}_3)$ expresses that there is nothing on top of block $\mathtt{b}_3$. A *plan* that transforms an initial state $s_1 \equiv \{\mathtt{on}(\mathtt{b}_1, \mathtt{b}_2), \mathtt{on}(\mathtt{b}_2, \mathtt{b}_3), \mathtt{on}(\mathtt{b}_3, \mathtt{b}_4), \mathtt{on}(\mathtt{b}_4, \mathtt{table}), \mathtt{clear}(\mathtt{b}_1)\}$ to state $s_2 \equiv \{\mathtt{on}(\mathtt{b}_2, \mathtt{b}_1), \mathtt{on}(\mathtt{b}_3, \mathtt{b}_4), \mathtt{clear}(\mathtt{b}_2), \mathtt{clear}(\mathtt{b}_3), \mathtt{on}(\mathtt{b}_4, \mathtt{table}), \mathtt{on}(\mathtt{b}_2, \mathtt{table})\}$ is $\langle \mathtt{move}(\mathtt{b}_1, \mathtt{table}), \mathtt{move}(\mathtt{b}_2, \mathtt{table}), \mathtt{move}(\mathtt{b}_2, \mathtt{b}_1) \rangle$.

Relational generalization and solution techniques for RMDPs: RMDP state spaces quickly become huge for modest amounts of objects. To be able to specify RMDPs, to learn value functions and policies, and to generalize experience while learning, typically FOL *languages* are used. These exploit powerful capabilities such as *quantification* and *variables* to abstract away from specific objects. For example, typically T is specified using (probabilistic) STRIPS operators, and value functions as FOL *decision trees*.

In the *model-based* setting, optimal value functions can – in principle – be *deduced* from a logical specification of the RMDP (e.g. see [9] and [17]). The resulting value functions exploit variables to compactly represent (in terms of FOL formulas) state sets. For example, $\langle \exists \mathtt{X}\ \mathtt{on}(\mathtt{b}_1, \mathtt{table}) \wedge \mathtt{on}(\mathtt{X}, \mathtt{b}_1), 10 \rangle$ expresses that all states in which block $\mathtt{b}_1$ is on the table with something on it, have a value 10. Note that this *abstract state* can be used in worlds with different numbers of blocks. In the *model-free* setting we can again identify two types of algorithms. Most *relational* RL techniques are *value-based*, including the first approach [6] which learned logical decision trees based on batches of examples generated in a standard Q-learning approach. More recent techniques have explored other logical abstractions, instance-based representations and kernels, for example. Several *policy-based* techniques have been explored in the relational setting too. Some gather samples of state-action pairs and use these for classification learning of the policy ([7]), some define a *gradient* on a policy structure and optimize (probabilistic) action selection based on a gradient search [8], and a third group upgrades evolutionary search techniques to the relational case and includes GAPI which we describe here. For a more thorough description of all solution techniques for RMDPs we refer to [18].

3 GAPI: Evolving Probabilistic Relational Policies

We introduce a new algorithm for policy-based, model-free solving of RMDPs, called GAPI. The main idea is to lift genetic algorithms to search for relational policy representations for a particular problem. We have outlined that evolutionary policy search relies on a number of components such as *fitness evaluation*, *individuals*, *population*, *selection*, *recombination*, *mutation* and so on. In this section we show how this can be done in the GAPI system [4].

Individuals and population: The representation that is used in a GA is very important in determining which results can be obtained. Each individual in GAPI is a *probabilistic relational policy* for a given RMDP. That is, a policy π is a set of *policy rules* $\mathtt{cond}(\tilde{\mathtt{X}}) \rightarrow \mathtt{act}(\tilde{\mathtt{X}})$, expressing that action $\mathtt{act}$ can be taken in states where the $\mathtt{cond}$ holds. When the rule is applied, all variables $\tilde{\mathtt{X}}$ are bound to objects that are matched in the current state. There are several possibilities for the condition of a rule. It can be empty (or: $\mathtt{true}$), which means that the condition is always fulfilled. It can be one literal, that is one predicate or the negation of a predicate, and that predicate comes from the set of predicates used for representing the state, P_S, or from the set of predicates from a separate set of background knowledge definitions, P_{BK}. The arguments of the predicates are constants from the problem domain that are also used in the goal, or variables. Only constants appearing in the goal specification may appear in a policy rule, to improve the generality of the rules and to avoid creating rules that are over-fitted to the test environments. The last possibility for a condition, is a sequence of literals. The *order of the literals* is important, since it can sometimes have an influence on the way PROLOG interprets them. The representation of rules is similar to Horn clauses in logic-based systems systems [5, 3] and many other techniques for RMDPs use similar representational devices [18]. The following example on the left is a policy for putting block $\mathtt{b}_1$ on block $\mathtt{b}_2$.

$$\begin{cases} \mathtt{above}(\mathtt{X}, \mathtt{b}_1), \mathtt{clear}(\mathtt{X}) & \rightarrow \mathtt{put\_on\_table}(\mathtt{X}) \\ \mathtt{above}(\mathtt{X}, \mathtt{b}_2), \mathtt{clear}(\mathtt{X}) & \rightarrow \mathtt{put\_on\_table}(\mathtt{X}) \\ \mathtt{clear}(\mathtt{b}_1), \mathtt{clear}(\mathtt{b}_2) & \rightarrow \mathtt{put\_on}(\mathtt{b}_1, \mathtt{b}_2) \end{cases} \qquad \begin{cases} \mathtt{on}(\mathtt{X}, \mathtt{Y}), \mathtt{clear}(\mathtt{X}) & \rightarrow \mathtt{put\_on\_table}(\mathtt{X}) \\ \mathtt{clear}(\mathtt{b}_2), \mathtt{clear}(\mathtt{b}_5) & \rightarrow \mathtt{put\_on}(\mathtt{b}_2, \mathtt{b}_5) \\ \mathtt{true} & \rightarrow \mathtt{put\_on\_table}(\mathtt{b}_3) \end{cases} \quad (1)$$

The semantics of a policy π is such that the applicable rules may suggest several actions, and the one action to be executed is picked *probabilistically* from this. Hence, the *order of the rules* is not important for the *execution* of the policy. A consequence is that some actions have a greater probability than others to be selected, if they appear more than once in the resulting set of actions. An alternative would be to see a policy as an (ordered) *decision list*, where only the first that applies suggests an action. Such policies are often used, but we will see that a probabilistic interpretation is quite useful in our setting. An example of policy execution can be given for the policy on the right in Equation 1. Let the state be $s \equiv \{\mathtt{on}(\mathtt{b}_1, \mathtt{table}), \mathtt{on}(\mathtt{b}_2, \mathtt{b}_1), \mathtt{on}(\mathtt{b}_3, \mathtt{b}_2), \mathtt{on}(\mathtt{b}_4, \mathtt{table}), \mathtt{on}(\mathtt{b}_5, \mathtt{b}_4), \mathtt{clear}(\mathtt{b}_3), \mathtt{clear}(\mathtt{b}_5)\}$ The first rule generates $\{\mathtt{put\_on\_table}(\mathtt{b}_3), \mathtt{put\_on\_table}(\mathtt{b}_5)\}$, since $\mathtt{b}_3$ and $\mathtt{b}_5$ are the only blocks that are on top of something else and clear. The second rule generates $\emptyset$, because $\mathtt{b}_2$ and $\mathtt{b}_5$ are not clear. The third rule generates $\{\mathtt{put\_on\_table}(\mathtt{b}_3)\}$, since that rule always fires. The total set of actions is thus $\{\mathtt{put\_on\_table}(\mathtt{b}_3), \mathtt{put\_on\_table}(\mathtt{b}_3), \mathtt{put\_on\_table}(\mathtt{b}_5)\}$. This means that the action $\mathtt{put\_on\_table}(\mathtt{b}_3)$ has a probability of $\frac{2}{3}$ of being selected, and $\mathtt{put\_on\_table}(\mathtt{b}_5)$ a probability of $\frac{1}{3}$.

Fitness evaluation: The fitness of a policy is defined in terms of the rewards obtained while executing it, the number of steps taken to reach the goal, plus a small penalty for large policy structures. Each policy is evaluated several times in the given environment, starting in a state in which the goal has not yet been fulfilled.

The reward obtained by the policy, is the reward for accomplishing the goal, if it did so, plus other rewards that it obtained during execution. These rewards can also be negative, for example as a punishment for executing an illegal action. The total reward obtained by a policy is $\text{total reward}_\pi = \sum_{i=0}^{N} max\,(0, \text{Reward}_i)$ where N is the number of test environments on which the policy was executed, and Reward_i the total reward obtained in the ith test. We only take into account rewards that are positive, to encourage policies that actually work in some situations, and to avoid negative total reward values. Negative fitness value cannot properly be used in our selection mechanism. One component goal perf, is a combination of the reward that is obtained, and the number of steps: $\text{goal perf}\,(\pi) = \left(1.4 - \frac{\text{total steps}_\pi}{\text{max total steps}}\right) \times \text{total reward}_\pi$, where max total steps is the sum of the maximum number of steps that can be done in each test environment that was used. The constant 1.4 avoids that it can happen that if the simulation completed without reaching the goal, the reward gained during the simulation gets lost.

The *complexity* of a policy π is defined in terms of the number of rules in a policy, and the number of literals in its rules, as: $\text{complexity}\,(\pi) = |\pi| + 0.2 \cdot \sum_{i=0}^{\#\text{ rules}} |\pi(i)|$ where $|\pi|$ is the number of rules in π, $\pi(i)$ is rule *i* in π, and $|r|$ is the number of condition literals of rule *r*. The number of *0.2* is chosen so that sub-conditions have a smaller weight than rules, but a new rule is preferred over a very complex condition, e.g. the policy in Equation 1(left) has a complexity $3 + 0.2 \times (2 + 2 + 2) = 4.2$. Now, the fitness can be computed as follows:

$$\text{fitness}\,(\pi) = \text{goal perf}\,(\pi) + 0.4 \cdot \text{goal perf}\,(\pi) \cdot \left(1 - \frac{\text{complexity}\,(\pi)}{\text{max policy complexity}}\right) \tag{2}$$

When all policies of one generation have been evaluated, we check if they do not all have a fitness value of zero. If that is the case, we give them all a small equal fitness value. Both constants in the fitness evaluation have been chosen based on experience on the tests we describe later, but more extensive experiments could be conducted to find possibly better parameter settings for different domains.

Selection: The selection process picks out the best individuals in the population, based on their fitness values, and has a profound influence on the variation of the population and on the convergence speed of the system. We use a *fitness proportionate* mechanism, meaning that the probability that an individual gets selected increases with its fitness value. There are many ways to do this, and here we use *stochastic universal sampling* [1], which is an improved variant of universal sampling, or the well-known *roulette wheel selection*. It works as follows; instead of doing a wheel spin (as in roulette wheel selection) for every individual that is selected, it uses a single spin. This results in a number *r* in the interval $[0..1]$. Suppose each selected individual has a number *i*. Then the value used to select that individual from the population is $\text{index value}_i = \frac{r+i}{N}$ where N is the total number of individuals that has to be selected. This results again in a number in the interval $[0..1]$. The advantage comes from the fact that all generated index values are equally spaced. This results in a more uniform distribution of these index values, and therefore enlarges the variation in the population.

Recombination: One way to search is to *combine* elements of policies that have high fitness. In this way, useful *building blocks* can be interchanged between fit individuals, thereby spreading good partial solutions throughout the population. The *recombination* operator used by GAPI is a *one-point crossover* operator on the policy structures of two parent individuals. In both parents, a crossover point is chosen randomly between rules. The first offspring consists of the first part of the first parent, and the second part of the second parent. The second offspring consists of the first part of the second parent, and the second part of the first parent. For each individual that is selected, we decide whether we will use it as a parent or not, based on the probability for crossover, p_c. From the set of parents, we randomly choose couples. As an example, the recombination of two parent policies $\pi_1 = \{p_1, p_2, \cdot, p_3\}$ and $\pi_2 = \{q_1, \cdot, q_2, q_3\}$, where crossover points are highlighted with a dot, results in $\pi_3 = \{p_1, p_2, q_2, q_3\}$ and $\pi_4 = \{q_1, p_3\}$.

Mutation Given the symbolic structure of policies, so-called *mutation operators* that modify the rule and policy structure, are very important in the search process. GAPI supports two kind of mutators. The first kind operates on the policy as a whole. Here we employ two mutators for *adding* and *removing* rules. The latter is simple: it removes a randomly selected rule. The former is more involved. A new rule is generated by first taking an action randomly from the available actions. The arguments of the action are filled with either terms that are mentioned in the goal (which can be constants or variables), or with new variables. If no free variables were chosen as arguments, the condition is left empty. Otherwise, a simple condition is added to the rule, to increase the chance that the rule is useful.

The second kind of mutators operates on the level of a single rule. Therefore, they are applied on each rule in the policy with a probability $p_i = \frac{1}{N}$, with *N* the number of rules in the policy. We currently have

defined seven mutators that operate on rules. These mutators are very similar to the *refinement operators* used in logical machine learning systems [3]. The first two mutators *add or remove a condition* (one literal) in a rule. Then, there are two mutators that *replace a term* (e.g. a variable) in the rule, either in one place, or all places in the rule where it occurs. This can, for example, correspond to the instantiation of a variable with a constant. An additional mutator can be used to *unify two terms* in the rule. To introduce negated conditions, we employ another mutator that *negates one of the conditions* in the rule. A last mutator switches the positions of two conditions in a rule. As we have noted before, this can matter in the context of PROLOG's way of dealing with negation. To be able to handle all these mutators, a meta-mutator is used as an interface for the generic GA. It takes a policy as input, applies all mutators on it with a certain probability, and returns the result. It is possible that multiple mutators are applied on the same policy at once, which can be beneficial, if a single change does not have an advantage, but the combination of two changes has.

Domain description and goals: GAPI needs information about the problem. First of all, a language bias is expected in the form of the set of predicates used to represent the state of the environment, P_S, a set of extra predicates that exist as background knowledge, P_{BK} (plus their definitions), a set of free variables that can be used in policy rules, a set of action predicates, P_A, and a goal specification. Furthermore, a number that indicates the maximum complexity of candidate solutions should be provided, as well as a set of test environments for the given problem (i.e. the RMDP) on which candidate policies should be tested. Each test environment consists of the goal specification and (implicitly) a way to generate an initial state.

A *goal* is a condition on the state of the environment, that has to be fulfilled at the end of the execution of a policy. The predicates in that condition can be predicates from the set for describing the state of the environment, P_S, or predicates from the background knowledge, P_{BK}. The arguments of the predicates can be constants in the problem domain or variables. If used in a goal, we distinguish two kinds of variables. The first kind are free variables, that can get any value, as long as they fulfill the condition stated in the goal. The second kind are in fact parameters of the goal, which enables GAPI to learn policies for abstract tasks, and in that way, make the generated policies more powerful. An abstract goal such as $\texttt{clear}(\texttt{X}), \texttt{on}(\texttt{X}, \texttt{table})$ can be used to learn the task of obtaining one un-stacked block, without knowing beforehand which block. The parameters of the goal are filled in when executing the policy for a specific task.

Related work: As described in the previous section, there are several methods for solving RMDPs. The only two other algorithms that use evolutionary techniques are FOXCS [10] and GREY [13]. FOXCS is based on learning classifier systems differs from GAPI in that it learns to predict action values, and uses rule-based evolution (as opposed to evolving complete policies). GREY is more related to GAPI, in that it too uses lifted genetic algorithms, though GREY was implemented in pure PROLOG and did not support the kind of options, nor the transfer capabilities, we will describe in the experiments. Related to the transfer of policies, some efforts have been made on this aspect in other recent systems [16]. More in general, there are several other techniques for evolutionary search for relational representations, but these techniques focus on classification tasks, not RMDPs (e.g. see [5]). For a more thorough description of other related work, and a concise description of the differences between relational value functions and policies can be found in [18].

4 Experimental Evaluation

We present a number of experiments with GAPI on two different domains, the BLOCKS WORLD and the GOLDFINDER domain. For both sets we first briefly describe the setting and the specific goal of the experiment. All experiments were performed on a laptop with an Intel Core 2 Duo at 2.8 GHZ. with 4GB RAM. GAPI is implemented as a JAVA program that calls a PROLOG engine through a standard file interface. In this way, the logical reasoning part is handled by a dedicated program, execution can be partly parallelized, and one is free to choose a particular engine. The output reported back by PROLOG engine consists of performance information about the tested policy. In this work we use the YAP [1] implementation.

Experiment I – benchmark BLOCKS WORLD problems: Here we first show that GAPI can handle simple RMDP benchmark problems. Then we show how abstract goals can be learned, and how learned policies can be *transferred*. All tasks in BLOCKS WORLD contain 10 blocks. We employ a standard BLOCKS WORLD with two possible actions: $\texttt{put\_on}(\texttt{X}, \texttt{Y})$ for stacking blocks and $\texttt{put\_on\_table}(\texttt{X})$ for putting blocks on the table. Three standard tasks here are `stack`, in which the goal is to put all blocks into one tower, `unstack`, in which the goal is to have all blocks un-stacked on the table, and $\texttt{on}(\texttt{a}, \texttt{b})$ which asks for stacking two specific blocks. Here we only report results for the `stack` task, but the results for the other two tasks are

[1] YAP homepage: http://www.dcc.fc.up.pt/~vsc/Yap/ Universidade do Porto

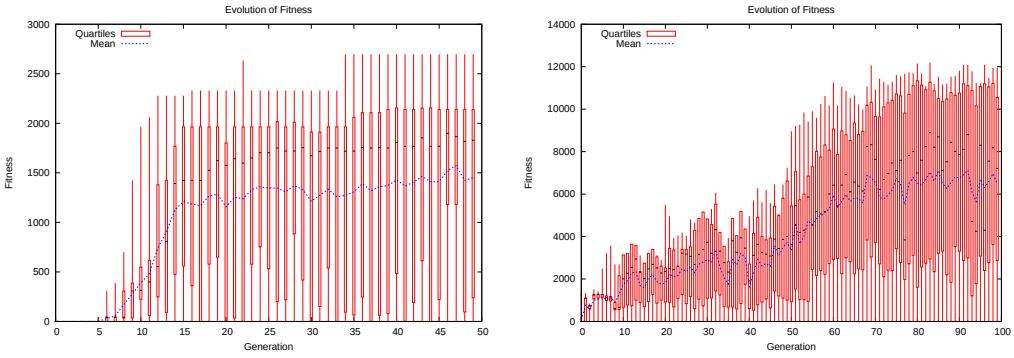

Figure 1: **(left)** Evolution of the fitness values in the population for the *stack* task. The maximum possible fitness is 3204. **(right)** The onXY task. The maximum possible fitness is 14364.

similar. We employ simple background predicates such as $\texttt{above}(\texttt{X},\texttt{Y})$ and $\texttt{highest\_stack}(\texttt{X})$ with their intuitive meanings. The reward function in all tasks is: one gets $+100$ upon reaching the goal, -1 for illegal actions, and 0 otherwise. A learned policy is:

$$\pi = \texttt{clear}(\texttt{X}), \texttt{highest\_stack}(\texttt{Y}), \texttt{block}(\texttt{X}), \texttt{X} \backslash = \texttt{Y} \rightarrow \texttt{put\_on}(\texttt{X},\texttt{Y}).$$

For execution, this policy is optimal, but the literal $\texttt{block}(\texttt{X})$ is unnecessary. The algorithm converged to this after only 34 generations, taking less than 5 minutes. Figure 1, depicts the fitness value evolution.

Experiment II – abstract goals and options: Here we move to more complex tasks in the BLOCKS WORLD. The first task is onXY, a generalized version of the benchmark $\texttt{on}(\texttt{a},\texttt{b})$ task, which shows how GAPI can learn parameterized policies. We use 100 test environments and allow for 100 iterations, and get:

$$\pi(\texttt{X},\texttt{Y}) = \begin{cases} \texttt{clear}(\texttt{X}), \texttt{clear}(\texttt{Y}) & \rightarrow \texttt{put\_on}(\texttt{X},\texttt{Y}) \\ \texttt{clear}(\texttt{Z}), \texttt{above}(\texttt{Z},\texttt{X}) & \rightarrow \texttt{put\_on\_table}(\texttt{Z}) \\ \texttt{clear}(\texttt{Z}), \texttt{above}(\texttt{Z},\texttt{Y}) & \rightarrow \texttt{put\_on\_table}(\texttt{Z}) \end{cases}$$

X and Y are the parameters of the goal, and of course, of the policy. The policy is optimal and was found in 15 minutes. Learning results are shown in Figure 2. The maximum fitness values do not increase quasi monotonically as in the previous. This is caused by the fact that the parameters are in each generation filled in with different values, and thus for some values the goals are easier to accomplish than for other values. Since this policy can reach a useful (sub)goal, namely ensuring that two blocks are stacked on top of each other, a useful thing to do is to transform it into a new (complex) action. Based on the declarative nature of policies in GAPI this can be done in a principled way (see [4]), and it resembles the use of *options*, *skills* or *macro-actions* typically used in *hierarchical* RL [14].

The second task is to learn $\{\texttt{on}(\texttt{X},\texttt{Y}), \texttt{on}(\texttt{Y},\texttt{Z})\}$. This task is even more complex than the previous task. First we let GAPI try to find a policy from scratch and we give it 500 iterations. After approximately 2 hours the result was the following policy:

$$\pi(\texttt{X},\texttt{Y},\texttt{Z}) = \begin{cases} \texttt{clear}(\texttt{X}), \texttt{block}(\texttt{A}), \texttt{on}(\texttt{Y},\texttt{Z}) & \rightarrow \texttt{put\_on}(\texttt{X},\texttt{Y}) \\ \texttt{clear}(\texttt{A}) & \rightarrow \texttt{put\_on\_table}(\texttt{A}) \\ \texttt{clear}(\texttt{X}), \texttt{clear}(\texttt{Y}), \texttt{block}(\texttt{A}), \texttt{block}(\texttt{X}), \texttt{clear}(\texttt{Z}) & \rightarrow \texttt{put\_on}(\texttt{Y},\texttt{Z}) \end{cases}$$

This policy works, but far from optimally, taking more steps than necessary and containing unnecessary literals. It may also produce illegal actions. In Figure 2(left) we see that the learning process has reached a plateau. The second attempt uses the result of the $\texttt{on}(\texttt{X},\texttt{Y})$ task (we rename this to $\texttt{make\_sure\_on}(\texttt{Y},\texttt{Z})$). We let GAPI learn for only 100 iterations in which it learns a policy with two rules $\pi(\texttt{X},\texttt{Y},\texttt{Z}) = \{\texttt{true} \rightarrow \texttt{make\_sure\_on}(\texttt{Y},\texttt{Z}), \texttt{true} \rightarrow \texttt{make\_sure\_on}(\texttt{X},\texttt{Y})\}$. This policy is learned in less than 15 minutes. The best solution was already found after 60 generations. It is not completely optimal, it could be improved by adding $\texttt{not}(\texttt{on}(\texttt{Y},\texttt{Z}))$ to the condition of the first rule, and $\texttt{on}(\texttt{Y},\texttt{Z})$ to the condition of the second rule. A comparison of the evolution of the results is shown in Figure 2(right). This experiment shows clearly that the learning task has become easier with the transferred knowledge.

Experiment III – GOLDFINDER: A second series of experiments was conducted in the GOLDFINDER domain, which consists of a simple grid, in which the agent can walk around. Blocks of gold are distributed over the locations of the grid. One or more vaults are also present somewhere on the grid.

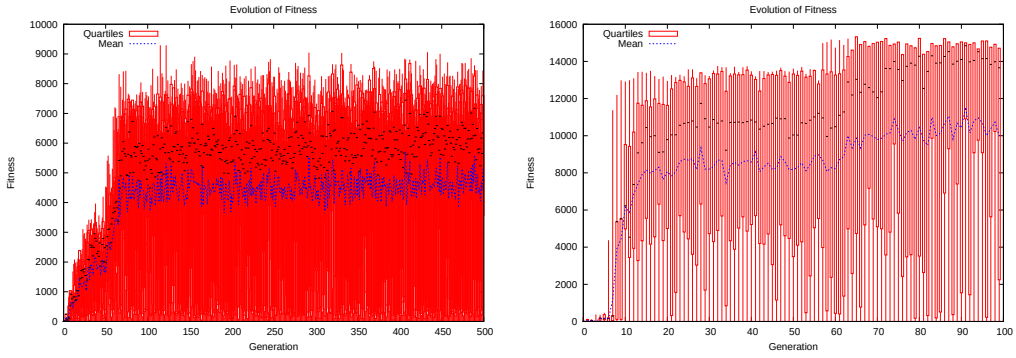

Figure 2: **(left)** The onXYZ task, when learning from scratch. The maximum possible fitness is 13214. **(right)** Same task, now using the results from onXY, with maximum fitness 16740.

The agent has a number of possible actions. It can take a step, in the direction it chooses, left, right, up or down. Furthermore, it can pick up gold and put it in its bag, and it can put the gold it has collected in a vault, both only at specific locations. The agent can sense whether there is gold on its current position, or a vault. Also, it can sense the directions to the nearest gold piece or vault. The goal is to collect all the gold in the environment, and put it in a vault. The agent gets rewarded for picking up gold ($+20$), for putting gold in the vault ($+20$) and for completing the task ($+100$). For illegal actions it is punished (-1).

We start with the simplest goal where the agent has to collect all the gold in the environment, and put it in a vault. The amount of gold that the agent can carry is unlimited. We let GAPI run for 200 iterations, which took about 24 minutes, and get (left):

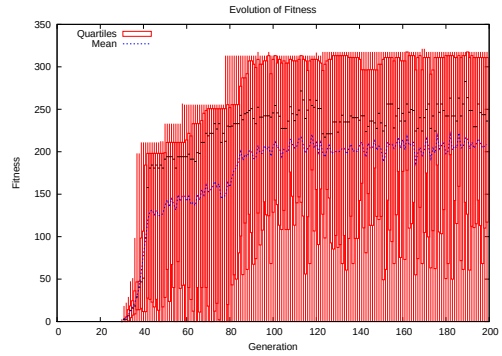

Figure 3: Evolution of the fitness values in the GOLDFINDER domain, with unlimited bag size.

$$\begin{cases} \texttt{not(on\_gold)}, \texttt{dng(X)} & \rightarrow \texttt{do\_move(X)} \\ \texttt{on\_gold} & \rightarrow \texttt{pick\_gold} \\ \texttt{dnv(X)}, \texttt{no\_gold\_left} & \rightarrow \texttt{do\_move(X)} \\ \texttt{on\_vault} & \rightarrow \texttt{put\_gold\_in\_vault} \\ \texttt{on\_gold} & \rightarrow \texttt{do\_move(X)} \end{cases} \qquad \begin{cases} \texttt{dng(X)} & \rightarrow \texttt{do\_move(X)} \\ \texttt{dng(A)} & \rightarrow \texttt{do\_move(X)} \\ \texttt{on\_vault} & \rightarrow \texttt{put\_gold\_in\_vault} \\ \texttt{on\_vault} & \rightarrow \texttt{do\_move(X)} \\ \texttt{true} & \rightarrow \texttt{pick\_gold} \end{cases} \tag{3}$$

Here dng(X) and dnv(X) stand for directions to nearest gold and vault. The policy is optimal, except for the (unnecessary) last rule. In Figure 4(left) we see a learning curve; the first 30 iterations no policy is found that obtains a positive reward, but once such a policy is found, the learning progresses until it more or less converges in generation 125.

One extension is to limit the gold the agent can carry to a certain amount. We set this to three, and ran the same experiment as above. The result is in Equation 3(right). Obviously, this is not a good policy, since it does not take into account the amount of gold that is already in the bag, and therefore it takes unnecessary steps. Instead of letting GAPI run longer, we now try to transfer one of the earlier policies:

$$\begin{cases} \texttt{on\_gold}, \texttt{place\_left\_in\_bag} & \rightarrow \texttt{pick\_gold} \\ \texttt{not(on\_gold)}, \texttt{dng(X)} & \rightarrow \texttt{do\_move(X)} \end{cases} \qquad \begin{cases} \texttt{dnv(X)}, \texttt{place\_left\_in\_bag} & \rightarrow \texttt{collect\_gold} \\ \texttt{dnv(X)} & \rightarrow \texttt{do\_move(X)} \\ \texttt{on\_vault} & \rightarrow \texttt{put\_gold\_in\_vault} \\ \texttt{dnv(X)}, \texttt{place\_left\_in\_bag} & \rightarrow \texttt{collect\_gold} \end{cases} \tag{4}$$

We rewrite this result to the action collect_gold, and get the policy structure on the right. Only the literal dnv(X) in the first and last rule is unnecessary. The learning process was much easier. The fact that it contains two rules that are the same alter only the probabilities of the action selection; it will prefer collecting more gold if there is still room in the bag. Figure 4 shows learning results for this experiment.

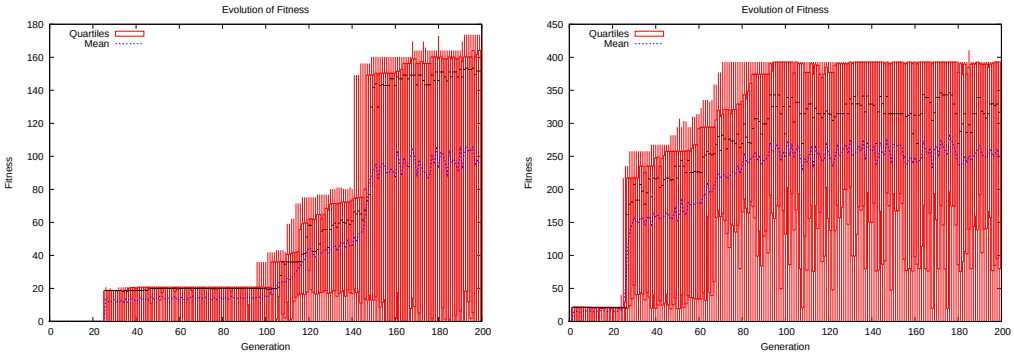

Figure 4: **(left)** GOLDFINDER, from scratch, with limited bag size. **(right)** GOLDFINDER with learned skills.

5 Conclusions

We have introduced GAPI for evolving relational, probabilistic policies for RMDPs. The experimental results show that it can handle several standard problems, and in addition, that it can evolve abstract, parameterized, policies. Furthermore, we have shown that based on an automated recoding scheme, policies can be transferred to tasks of higher complexity.

Future work includes many enhancements in terms of the implementation, of the evolutionary techniques used, and a variety of other problem settings. More specifically, we have started investigating GAPI for *simultaneous* goals. For example, by adding monsters to the GOLDFINDER domain, we get a simple version of PACMAN in which the agent constantly has to choose between avoiding monsters and gathering gold (e.g. as a kind of multi-objective (R)MDP, see also [2]). These policies can be evolved separately, and combined through the recombination operator. Or, one can start learning the complete task based on a population consisting of subtask policies. The declarative nature of probabilistic, relational policies enables an elegant framework for transferring useful building blocks.

References

[1] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *GA*, 1987.

[2] L. Barrett and S. Nrayanan. Learning all optimal policies with multiple criteria. In *ICML*, 2008.

[3] L. De Raedt. *Logical and Relational Learning*. Springer, 2008.

[4] T. De Vuyst. Breeding logic. Master's thesis, Dept. of Computer Science, K.U. Leuven, Heverlee, Belgium, 2009.

[5] F. Divina. *Hybrid genetic relational search for inductive learning*. PhD thesis, Department of Computer Science, Vrije Universiteit, Amsterdam, the Netherlands, 2004.

[6] S. Džeroski, L. De Raedt, and H. Blockeel. Relational reinforcement learning. In *ICML*, 1998.

[7] A. Fern, S. W. Yoon, and R. Givan. Approximate policy iteration with a policy language bias: Solving relational Markov decision processes. *JAIR*, 25:75–118, 2006.

[8] C. Gretton. Gradient-based relational reinforcement-learning of temporally extended policies. In *ICAPS*, 2007.

[9] K. Kersting, M. van Otterlo, and L. De Raedt. Bellman goes relational. In *ICML*, 2004.

[10] D. Mellor. *A Learning Classifier System Approach to Relational Reinforcement Learning*. PhD thesis, School of Electrical Engineering and Computer Science, University of Newcastle, Australia, 2008.

[11] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, Massachusetts, 1996.

[12] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for RL. *JAIR*, 11:241–276, 1999.

[13] T. J. Muller and M. van Otterlo. Evolutionary reinforcement learning in relational domains. In *Proceedings of the 7th European Workshop on Reinforcement Learning*, 2005.

[14] M. R. K. Ryan. Hierarchical decision making. In J. Si, A. G. Barto, W. B. Powell, and D. Wunsch, editors, *Handbook of Learning and Approximate Dynamic Programming*. Wiley-IEEE Press, 2004.

[15] R. S. Sutton and A. G. Barto. *Reinforcement Learning: an Introduction*. The MIT Press, 1998.

[16] L. Torrey, J. Shavlik, T. Walker, and R. Maclin. Relational macros for transfer in reinforcement learning. In *ILP*, 2007.

[17] M. van Otterlo. Intensional dynamic programming: A Rosetta stone for structured dynamic programming. *Journal of Algorithms in Cognition, Informatics and Logic*, 2009. in press.

[18] M. van Otterlo. *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for Adaptive Sequential Decision Making under Uncertainty in First-Order and Relational Domains*. IOS Press, Amsterdam, The Netherlands, 2009.