Policy Reuse in a General Learning Framework

No Author Given

No Institute Given

Abstract. Policy reuse is a kind of transfer learning to improve a reinforcement learning system by reusing part of the information of the state-value function from previous problems to new problems. In this paper we overhaul this aproach in the context of a general learning framework for structured prediction using user-defined operators and a functional programming language for the representation of rules and operators. The redefinition of what policy reuse is in this context is motivated by the representation of states and actions as feature vectors, and the use of a Q matrix which is actually a table, from which a supervised model is learnt. This makes it possible to have a more flexible mapping between old and new problems, since we work with an abstraction of rules and actions. We do some experiments with the system gErl on some structured prediction problems (list patterns).
Keywords: policy reuse, reinforcement learning, inductive programming, transfer learning, complex data, heuristics, Erlang.

1 Introduction

The reuse of knowledge which has been acquired in previous learning processes in order to improve or accelerate the learning of future tasks is an appealing idea. Several solutions have been proposed to reuse knowledge, in areas such as meta-learning, incremental learning, employment of background knowledge or transfer learning. As the task and the learning system become more elaborated, this knowledge reuse becomes more important. In this paper we deal with a general rule-based learning setting using a functional programming representation, where operators can be defined and customised for each kind of problem. While one particular problem may require generalisation operators, another problem may require operators which add recursive transformations to explore the structure of the data. An appropriate choice of operators can embed transformations on the data but can also determine the way in which rules are generated and modified, so leading to (apparently) different learning systems. Making the user or the problem adapt its own operators is significantly different to the use of feature transformations or specific background knowledge. In fact, it is also significantly more difficult, since operators can be very complex things and usually embed the essence of a machine learning system. As a result, having a system which can work with different kinds of operators at the same time is a challenging proposal beyond the frontiers of the state of the art in machine learning.

Given the versatility of this system, the heuristics for exploring the search space must be more flexible. A reinforcement learning (RL) system determines which rules and operators are used and how they are combined. In this work, we will focus on how RL policies are reused between totally different tasks. In order

to do that, we use an appropriate feature space for describing the kinds of rules and operators that are giving good solutions (and high rewards), so this history is reused for other problems, even when the task and operators are different.

The paper is organised as follows. Section 2 makes a short account of the related and previous work. Section 3 introduces briefly the gErl system. Section 4 describes the RL-based heuristics used to guide the learning process and section 5 discusses how to transfer knowledge between tasks. Section 6 includes an empirical study which illustrates the way in which operators are defined and solutions are reached. Section 7 closes the paper.

2 Previous work

The goal of this paper is to describe the policy reuse that we will use in our system to take advantage of previous learning episodes (problems). Consequently, we will focus on previous work on policy reuse in the areas of reinforcement learning (RL) and transfer learning (TL), see [14] for a survey.

There are three main families of TL methods (in the RL area). Firstly, the source and target tasks use the same state variables and set of actions. Here, transfer learning is performed by initializing the Q-values of a new episode with previously learned Q-values [7]. This family includes those TL methods which use multiple source tasks by leveraging all experienced source tasks when learning a novel target task [9, 10] or by choosing a subset of previously experienced tasks (*Probabilistic policy reuse* [5] and Hierarchical RL [2]).

The second family of methods are those which are able to transfer between tasks with different state variables and actions, so that no explicit mapping between the tasks is needed. Instead the agent reasons over abstractions of the Markov Decision Process that are invariant when the actions or state variables change. Some methods use macro-actions or *options* [12] to learn new action policies in Semi-Markov Decision Processes. *Relational Reinforcement Learning* [4] is another particularly attractive formulation in this context of transfer learning.

The TL methods of the last family are more flexible than those previously discussed as they allow the state variables and available actions to differ between source and target tasks using inter-task mappings. Namely, explicit mappings are needed in order to transfer between tasks with different actions and state representations. This mapping may be provided to the learner (*advise* rules [11]) or may be autonomously learned (*qualitative dynamic Bayes networks* [8]).

Although TL in RL has made significant progress in recent years, there is a poor understanding about the reuse of knowledge for solving future totally different problems. This makes sense in our system since we can have a pool of user-defined operators to solve a problem, while only a subset is needed to solve it. The ultimate goal is to use old policy information to speed up the learning process of another different problem (which may need a different subset of operators).

3 The gErl system

As we have mentioned in Section 1, in this paper we use the gErl system [1], a general learning system which can be configured with different (possibly user-defined) operators. The system can be described as a flexible architecture (shown

in Figure 1) which works with populations of rules (expressed as unconditional / conditional equations) and programs in the functional language Erlang, which *evolve* as in an evolutionary programming setting or a learning classifier system [6]. Operators are applied to rules and generate new rules, which are combined with existing or new programs. With appropriate operators, using some optimality criteria (based on coverage and simplicity) and using a reinforcement learning-based heuristic (where the application of an operator over a rule is seen as a decision problem fed by the optimality criteria) many complex problems can be solved. In the rest of this section we will introduce some notation and concepts of the system that will be required to understand how our policy reuse techniques are devised and implemented. We refer the reader to [1] for details.

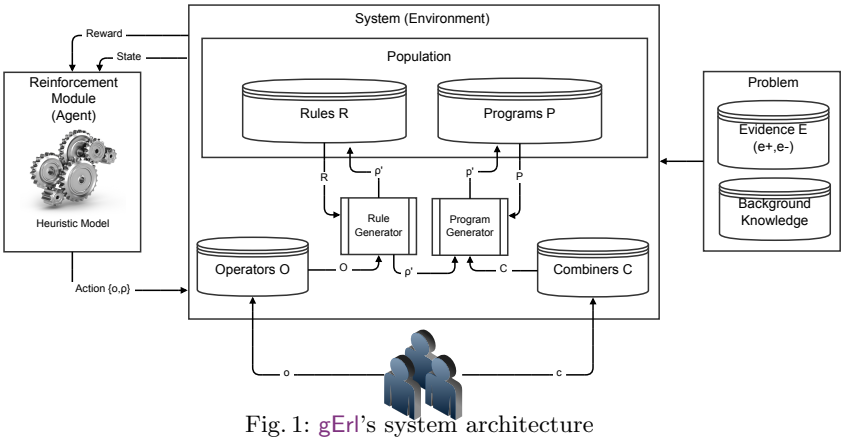

Fig. 1: gErl's system architecture

3.1 Data and model representation and learning problem statement

gErl can be considered and inductive (functional) programming system, as data and rules (and hence solutions) are represented in a functional programming language, Erlang. Let us see how these elements are represented and how the learning problem is stated.

Let Σ be a set of *function symbols* together with their arity and $\mathcal{X}$ a countably set of *variables*, then $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of *terms* built from Σ and $\mathcal{X}$. The set of variables occurring in a term t is denoted *Var(t)*. A term t is a *ground term* if $Var(t) = \varnothing$. An equation is an expression of the form $l = r$ where l (the left hand side, *lhs*) and r (the right hand side, *rhs*) are terms. $\mathcal{R}$ denotes the space of all (conditional) functional rules ρ of the way l [when G] $\rightarrow T, r$ where l and r are the lhs and the rhs of ρ (respectively), $G = \{g_1, g_2, \ldots g_m \mid m \geq 0\})$ is a set of conditions or Boolean expressions called guards, and $T = b_1, \ldots, b_n$, the tail of ρ, is a sequence of equations. If $G = \varnothing$, then ρ is said to be an unconditional rule. Let $\mathcal{P} = 2^{\mathcal{R}}$ be the space of all possible functional programs formed by sets of rules $\rho \in \mathcal{R}$. Given a program $p \in \mathcal{P}$, we say that term t reduces to term s with respect to p, $t \rightarrow_p s$, if there exists a rule l [when G] $\rightarrow T, r \in p$ such that a subterm of t at occurrence u matches l with substitution θ, all conditions $g_i\theta$ hold, for each equation $b_{i_l} = b_{i_r} \in T$, $b_{i_l}\theta$ and $b_{i_r}\theta$ have the same normal form

(that is, $b_{i_l}\theta \rightarrow_p^* b$, and $b_{i_r}\theta \rightarrow_p^* b$ and b can not be further reduced) and s is obtained by replacing in t the subterm at occurrence u by $r\theta$.

An example e is a ground rule $l \rightarrow r$ (that is, without condition nor tail), being r in normal form. We say that e is covered by a program p (denoted by $p \models \{l \rightarrow r\}$) if l and r have the same normal form with respect to p. A program $p \in \mathcal{P}$ is a solution of a learning problem defined by a set of positive examples E^+, a (possibly empty) set of negative examples E^- and a background theory B if it covers all positive examples, $B \cup p \models E^+$ (posterior sufficiency or completeness), and does not cover any negative example, $B \cup p \not\models E^-$ (posterior satisfiability or consistency). Our system has the aim of obtaining complete solutions, but their consistency is not a mandatory property, so approximate solutions are allowed. The positive coverage of a program $p \in 2^{\mathcal{R}}$ is defined as $Cov^+(p) = Card(\{e \in E^+ : B \cup p \models e\})$, where $Card(S)$ denotes the cardinality of the set S. The negative coverage Cov^- is defined analogously.

As we can see in Figure 1, our system works with two sets: a set of rules $R \subseteq \mathcal{R}$ and a set of programs $P \subseteq \mathcal{P}$, where each program $p \in P$ is composed by rules belonging to R. Initially, R is populated with E^+ and the set of programs P is populated with as many unitary programs as there are rules in R.

3.2 Operators and learning process

The definition of customised operators is one of the key concepts of our proposal. The idea is to transform the set of rules R using a set of *operators* $O \subset \mathcal{O}$ (provided by the user or existing in the system).

An operator can be seen as a piece of code (as complex as the user may wants) which performs modifications over the *lhs* or *rhs* of a rule and which is written in the same functional language as the system (Erlang) to take advantage of its high-order and reflection capabilities. The main idea is that, when the user wants to deal with a new problem, he/she can define his/her own set of *operators*, especially suited for the data structures of the problem. This feature allows our system to adapt to the problem at hand. Our system also has a special kind of operators $c \in C$, called *combiners*, that only apply to programs. The *Program Generator* module (Figure 1) applies a combiner to the last rule ρ' generated by the *Rule Generator* module and the population of programs P.

As the process progresses, new rules and programs will be generated. First, the *Rule Generator* process (Figure 1) gets the operator o and the rule ρ returned as an action $a = \langle o, \rho \rangle$ by the *Reinforcement Learning Module* (policy). This process applies the operator over the rule obtaining a new rule ρ' (if the operator is not suitable for the rule selected, the process returns the same rule) which is added to R. Then, the *Program Generator* process takes the new rule generated ρ' (if appropriate) as input, the set of programs P and the set of combiners C and generates a new program p' (which is added to P) applying the combiners over the previous inputs.

4 RL-based heuristics

In this section we describe the reinforcement learning approach followed by gErl in order to guide the learning process. Given that the users can define their own

operators, the search heuristics have been conceived as decisions about the operator to be used at each particular state of the process. For this, a model-based reinforcement learning approach has been developed, where the application of an operator over a rule is seen as a decision problem, for which learning also takes place, guided by the optimality criteria which feed a rewarding module.

4.1 RL problem statement

To guide the learning process we need a picture of the system in each step of the process (before and after applying an action) in terms of the quality of the set of rules and programs generated until now. Based on it, we model the decision process as a typical reinforcement learning task.

Formally, we define a state at each iteration t of the system as a tuple $\sigma_t = \langle R, P \rangle$ which represent the population of rules R and programs P in t. An action is a tuple $\langle o, \rho \rangle$ with $o \in \mathcal{O}$ and $\rho \in \mathcal{R}$ that represents the operator o to be applied to the rule ρ[1]. Our decision problem is a four-tuple $\langle \mathcal{S}, \mathcal{A}, \tau, \omega \rangle$ where: $\mathcal{S}$ is the state space; $\mathcal{A}$ is a finite actions space ($\mathcal{A} = \mathcal{O} \times \mathcal{R}$); $\tau : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a transition function between states and $\omega : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. These components are defined below:

- **States.** As we want to find a good solution to the learning problem, we describe each state σ_t by a tuple of features $s_t = \langle \phi_1, \phi_2, \phi_3 \rangle$ from which to extract relevant information in t:
  1. *Global optimality* (ϕ_1): This feature shows the average optimality of all programs in P_t. In turn, the optimality of a program p is computed by weighting three simpler heuristics according to its importance:

$$Opt(p) = \beta_1 \cdot CRD(p) - \beta_2 \cdot OU(p) + \beta_3 \cdot RP(p) \tag{1}$$

     where CRD (coverage rate difference) is a normalised measure of coverage difference between positive and negative examples covered, OU (operator usage) is a normalised measure of how many operators have been used to derive the rule and RP is a measure of the expressiveness of the programs in terms of number of variables, functions and constants. As for the current implementation of gErl, the weights are $\beta_1 = 1$, $\beta_2 = 0.2$ and $\beta_3 = 0.1$. Finally, the *Global optimality* factor is then calculated as the average of the optimalities of all programs in the system:

$$OptGlobal(P_t) = \frac{1}{Card(P_t)} \sum_{p \in P_t} Opt(p) \tag{2}$$

  2. *Average Size of Rules* (ϕ_2): measures the average complexity of all rules in R, using the RP measure mentioned above.
  3. *Average Size of programs* (ϕ_3): measures the average cardinality of all the programs in P_t in terms of the number of rules.
- **Actions.** Each rule is described by a tuple of features $\rho = \langle \varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5, \varphi_6, \varphi_7, \varphi_8 \rangle$ from which we extract relevant information:

[1] The probable infinite number of states and rules makes the abstraction of states and rules necessary.

1. *Size* (φ_1): expressiveness of the rule using RP.
2. *Positive Coverage Rate* (φ_2).
3. *Negative Coverage Rate* (φ_3).
4. *NumVars* (φ_4): number of variables of ρ.
5. *NumCons* (φ_5): number of constants (functors with arity 0) of ρ.
6. *NumFuncs* (φ_6): number of functors with arity greater than 0 of ρ.
7. *NumStructs* (φ_7): number of structures (lists, graphs, ...) of ρ.
8. *isRec* (φ_8): indicates if the rule ρ is recursive or not.

We use a natural index as the only feature for operators. As an action consists of a choice of operator and rule, an action is finally a tuple of *nine* features.

- **Transitions.** Transitions are deterministic. A transition τ evolves the current sets of rules and programs by applying the operators selected (together with the rule) and the combiners.
- **Rewards.** The optimality criteria seen above is used to feed the rewards. In particular, we use the result returned by equation (1) as reward.

At each point in time, the reinforcement learning policy π can be in one of the states $s_t \in \mathcal{S}$ and may select an action $a_t = \pi(s_t) \in \mathcal{A}$ to execute. Executing such action a_t in s_t will change the state into $s_{t+1} = \tau(s_t, a_t)$, and the policy receives a reward $w_t = \omega(s_t, a_t)$. The policy does not know the effects of the actions, i.e. τ and ω are not known by the policy and need to be learned. This is the typical formulation of reinforcement learning [13] but using features to represent the states and the actions. With all these elements, the aim of our decision process is to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximises:

$$V^{\pi}(s_t) = \sum_{i=0}^{\infty} \gamma^i w_{t+i} \tag{3}$$

for all s_t, where $\gamma \in [0, 1]$ is the *discount parameter* which determines the importance of the future rewards ($\gamma = 0$ only considers current rewards, while $\gamma = 1$ strives for a high long-term reward).

4.2 Modelling the state-value function: using a regression model

In our setting, for the RL module, we use a hybrid between model-free value-function methods (which update a state-value matrix) and model-based methods (which learn models for τ and ω) [13]. In particular, our approach uses the *state-value* function ($Q(s, a)$, which returns quality (q) *values*, $q \in \mathbb{R}$, as in Q-learning [15]) generalising it with a regression model. A model $M : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ calculates the optimality or q value for each state and action and tries to generalise Q. By using $a_t = \arg\max_{a \in \mathcal{A}} \{M(s_t, a_i)\}$ we get the best action for state s_t.

In order to train the model we use a 'matrix' Q (which is actually a table), whose rows are in $\mathcal{S} \times \mathcal{A} \times \mathbb{R}$ where $\mathcal{S}$ is a tuple of state features, $\mathcal{A}$ is the tuple of rule features and operator, and $\mathbb{R}$ is a real value for q. Table 1 shows an example of Q. Abusing notation, to work with Q as a function (like the original Q-matrix in many RL methods), we will denote by $Q[s, a]$ the value of q in the row of Q for that state s and action a. Each row in Q records the state and action for each time step in the system. So, Q grows in terms of the number of rows.

A supervised model M (using q as the output) is retrained periodically from table Q (gErl uses linear regression by default). Once the system has started, at each step, Q is updated using the following formula, as in Q-learning:

state (s)			action (a)								q
ϕ_1	ϕ_2	ϕ_3	o	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	φ_8	
1.223	1.473	6.431	2	17	3	0	4	1	2	0	0	**0.78**
1.301	1.511	6.253	2	16	3	1	3	2	1	0	0	**0.65**
...												

Table 1: Example of a Q matrix (represented as a table).

$$Q[s_t, a_t] \leftarrow \alpha \left[w_{t+1} + \gamma \max_{a_{t+1}} M(s_{t+1}, a_{t+1}) \right] + (1 - \alpha) Q[s_t, a_t] \quad (4)$$

where the maximum future value is obtained by the model instead of a Q-matrix. The previous formula has two parameters: the discount parameter $\gamma \in [0, 1]$, and the *learning rate* α ($\alpha \in [0, 1]$), which determines to what extent the newly acquired information will override the old information ($\alpha = 0$ makes the agent not to propagate anything, while $\alpha = 1$ makes the agent consider only the most recent information). By default, $\alpha = 0.5$ and $\gamma = 0.5$.

5 Reusing past policies

In this section, we describe how to reuse and apply policies.The abstract representation of states and rules in actions (the ϕ and φ features) facilitates the transfer of learning information between related but also different tasks (rules and actions may be completely different but their *features* can still be similar).

As we have seen in Section 2, in other TL methods the knowledge is transferred in several ways (via modifying the learning algorithm, biasing the initial action-value function, etc.) and, if the source and target task are very different, a mapping between actions and/or states is needed. Instead of that, in gErl the reuse of previous acquired knowledge is done in a totally transparent way.

The main reason why we can use the past policies (the learned "matrix" Q) in order to accelerate the learning of different new tasks is due to the abstract representation of states and actions which allows the system does not start from the scratch and reuse the optimal information, namely, actions successfully applied from certain states (from the previous task) when it reaches a similar (with similar features) new state. Due this abstract representation of states and actions, how different are the source and target task does not matter, so the reuse of knowledge in gErl as a transfer learning approach would overlap with any of the TL families showed in Section 2 and goes beyond, allowing to deal with totally different tasks.

The scope of this knowledge transfer as a model transferring is summarised as follows: when gErl reaches the solution of a given problem (or it executes a maximum number of steps), Q_{source} and the model M_{source} can be viewed as knowledge acquired that can be transferred to the new situation. Concretely, when gErl learns the new task, Q_{source} is used to train a new model which is used from the beginning (M_{target}) and it is updated with the new information acquired in each time step of the system (in order to retrain the model M_{target}). Figure 2 briefly shows the process.

6 Empirical study

In this section, we describe an experiment which illustrates the policy reuse strategy used in gErl.

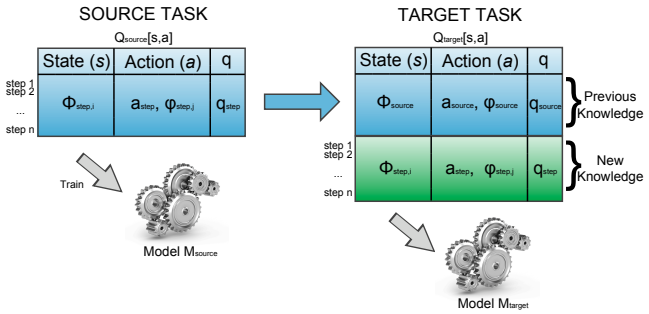

Fig. 2: The knowledge transfer process in the gErl system.

We use list processing problems as a structured prediction [3] domain where not only the input is structured but also the output. For the first part of our study we have selected five different problems that use the Latin alphabet $\Sigma = \{a, b, \ldots, z\}$ as a finite set of symbols and perform the following transformations:

1. $d \rightarrow c$: replaces *"d"* with *"c"*. Instances would look like this: $trans([t, r, a, d, e]) \rightarrow [t, r, a, c, e]$
2. $e \rightarrow ing$: replaces *"e"* with *"ing"* located at the last position of a list. Instances would look like this: $trans([t, r, a, d, e]) \rightarrow [t, r, a, d, i, n, g]$
3. $d \rightarrow pez$: replaces *"d"* with *"pez"* located at any position of a list. Instances would look like this: $trans([t, r, a, d, e]) \rightarrow [t, r, a, p, e, z, e]$
4. $Prefix_{over}$: adds the prefix *"over"*. Instances would look like this: $trans([t, r, a, d, e]) \rightarrow [o, v, e, r, t, r, a, d, e]$
5. $Suffix_{mark}$: adds the suffix *"mark"*. Instances would look like this: $trans([t, r, a, d, e]) \rightarrow [t, r, a, d, e, m, a, r, k]$

According to the data structure, we need a way to navigate the structure and apply local or global changes. In order to do this we need to define appropriate operators. The first operator, *oneSust*, is a two-step mechanism that takes the input and output lists in order to go through them comparing the lists until it finds any difference. As a result, it returns the input rule where its *rhs* has been substituted by a high-order function *map* (which applies a parametrised function to the whole list). The definition of this operator is written in Erlang, but it can be informally defined using an example: $oneSust(trans([a, b, c]) \rightarrow [d, b, c]) \Rightarrow trans([a, b, c]) \rightarrow map(a \rightarrow d, [a, b, c])$.

The second operator, *nSust*, which fits with the second and the third problem, also is a two-step mechanism similar to the previous one, but, instead of returning only one rule with a simple substitution, it returns as many rules as possible "complex" changes can be done from the difference between the input and output lists. This operator can be informally defined using an example: $nSust(trans([a, b, c]) \rightarrow [a, d, e, c]) \Rightarrow \{ \begin{smallmatrix} trans([a, b, c]) \rightarrow map(b \rightarrow de, [a, b, c]) \\ trans([a, b, c]) \rightarrow map(b \rightarrow dec, [a, b, c]) \end{smallmatrix}$.

The third and fourth operators, $addPrefix$ and $addSuffix$, just check whether the input list is a suffix or prefix, respectively, of the output list and take the difference. If so, it is returned the input rule where the *rhs* has been substituted by the concatenation of the difference and the input list (prefix) or by the concatenation of the input list and the difference. For instance: $addPrefix(trans([a, b, c]) \rightarrow$

$[z, a, b, c]) \Rightarrow trans([a, b, c]) \rightarrow [z] + +[a, b, c])$, $addSuffix(trans([a, b, c]) \rightarrow [a, b, c, z]) \Rightarrow trans([a, b, c]) \rightarrow [a, b, c] + +[z])$.

Finally, we need a way of generalising the rules. That is performed by the fifth and the sixth operators, $genLHS$ and $genRHS$, which generalise the input and output strings, respectively: $genLHS(trans(X) \rightarrow Y) \Rightarrow trans(V_S) \rightarrow Y$ and $genRHS(trans(X) \rightarrow Y) \Rightarrow trans(X) \rightarrow V_S$ where V_S is a string variable.

With these six operators gErl is able to solve any of the previous problems simply applying a sequence of the correct operators for each problem. For instance, given the instance $trans([a, b, c]) \rightarrow [a, d, c]$, we have the sequence of operator applications:

$$\begin{aligned} genPat(trans([a, b, c]) \rightarrow [a, d, c]) &\Rightarrow trans([a, b, c]) \rightarrow map(b \rightarrow d, [a, b, c]) \\ genLHS(trans([a, b, c]) \rightarrow map(b \rightarrow d, [a, b, c])) &\Rightarrow trans(V_S) \rightarrow map(b \rightarrow d, [a, b, c]) \\ genRHS(trans(V_S) \rightarrow map(b \rightarrow b, [a, b, c])) &\Rightarrow trans(V_S) \rightarrow map(b \rightarrow d, V_S) \end{aligned}$$

where the latter rule $trans(V_S) \rightarrow map(t \rightarrow u, V_S)$ is the solution.

Since we want to analyse the ability of the system to improve the learning process when reusing past policies, we will solve each of the previous problems and we will reuse the policy (each model) obtained on a previous problem to solve the rest of the problems (including itself). We will also add a few number of non-relevant operators to increase the difficulty to solve each problem. The set of operators O used has ten operators. To make the experiments independent of the operator index, we will set up 5 random orders for them. Each problem has 20 positive instances e^+ and no negative ones. From each problem we will extract 5 random samples of ten positive instances in order to learn a policy from them with each of the five order of operators (5 problems $\times$ 5 samples $\times$ 5 operator orders =125 different experiments).

First, we show the results without the use of previous policies. The mean number of steps needed to solve each problem is shown in Table 2. Next, we see the results with policy reuse in Table 3, showing the aggregated means (in number of steps) of each sample and operator order.

To analyse whether the differences are significant (between the results that do not reuse and those that do), we used the Wilcoxon signed-ranks test with a confidence level of $\alpha = 0.05$ and $N = 25$ (5 samples $\times$ 5 operators orders). The results which significantly improve the performance obtained without using previous policies are shown in bold. Interestingly, the results obtained reusing works for most combinations (in those combinations where the difference is not significant, it is still better in magnitude), including those cases where the problems have nothing to do and do not reuse any operator, suggests that the abstract description of states and rules is beneficial even when the problems are not related and gives support to the idea of a general system that can perform better as it sees more and more problems, one of the reasons why the reinforcement model and the abstract representations were conceived in gErl.

7 Conclusions and future work

One of the problems of reusing knowledge from previous learning problems to new ones is the representation and abstraction of this knowledge. In this paper we have investigated how policy reuse can be useful even in cases where the problems have no operators in common, simply because some abstract characteristics of two learning problems are similar at a more general level. In order to make more conclusive claims, more experiments should be done on other domains.

	$l \rightarrow c$	$e \rightarrow ing$	$d \rightarrow pez$	$Prefix_{over}$	$Suffix_{mark}$
Steps	108.68	76.76	74.24	61.28	62.28

Table 2: Results not reusing previous policies.

	Problem				
PCY from	$l \rightarrow c$	$e \rightarrow ing$	$d \rightarrow pez$	$Prefix_{over}$	$Suffix_{mark}$
$l \rightarrow c$	**65.68**	**58**	70, 64	**48.84**	**49.12**
$e \rightarrow ing$	**66.48**	**50.04**	**56.4**	**45.2**	**45.36**
$d \rightarrow pez$	**56.36**	**49.6**	**57.32**	52.24	**45.84**
$Prefix_{over}$	**58.8**	**48.96**	**60.6**	**43.8**	**46.88**
$Suffix_{mark}$	102, 72	**64.4**	67.32	56.16	**57.48**
Average	70.01	54.2	62.46	49.25	48.94

Table 3: Results reusing policies

There are many other things to explore in the context of gErl. We would like to include features for the operators as well. In more general terms, we think that a way (or measure) of similarity between problems would help us to better understand when the system is able to detect these similarities, from the point of view of better assessing the achievements of the system. Finally, while we have focussed on the system gErl, we think that many of the ideas in this paper could also be applied to other kinds of systems, most especially learning classifier systems, reinforcement learning and other evolutionary techniques.

References

1. gErl system. `http://goo.gl/XlTmN`, 2012.
2. T. Dietterich. Hierarchical Reinforcement Learning with the MAXQ value function decomposition. *J. Artif. Int. Res.*, 13(1):227–303, 2000.
3. T. Dietterich, P. Domingos, L. Getoor, S. Muggleton, and P. Tadepalli. Structured Machine Learning: the next ten years. *Machine Learning*, 73:3–23, 2008.
4. K Driessens. *Relational Reinforcement Learning. Phd Thesis*. Department of Computer Science, K.U.Leuven, Leuven, Belgium, 2004.
5. F. Fernandez and M. Veloso. Probabilistic policy reuse in a Reinforcement Learning agent. In *AAMAS 06*, pages 720–727. ACM Press, 2006.
6. J. Holland and Booker. What is a learning classifier system? In *Learning Classifier Systems*, volume 1813 of *LNCS*, pages 3–32. 2000.
7. J.Carroll. Fixed vs Dynamic Sub-transfer in Reinforcement Learning. In *ICMLA'02*. CSREA Press, 2002.
8. Y. Liu and P. Stone. Value-function-based transfer for reinforcement learning using structure mapping. AAAI, pages 415–20, July 2006.
9. N. Mehta. Transfer in variable-reward hierarchical reinforcement learning. In *In Proc. of the Inductive Transfer workshop at NIPS*, 2005.
10. T. J. Perkins and D. Precup. Using options for knowledge transfer in Reinforcement Learning TITLE2:. Technical report, Amherst, MA, USA, 1999.
11. B. Price and C. Boutilier. Accelerating Reinforcement Learning through implicit imitation. *Journal of Artificial Intelligence Research*, 19:2003, 2003.
12. R. Sutton. Between MDPs and semi-MDPs: A framework for temporal abstraction in Reinforcement Learning. *Artificial Intelligence*, 112:181–211, 1999.
13. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
14. M. Taylor and P. Stone. Transfer learning for Reinforcement Learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.
15. C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.