Noname manuscript No.
(will be inserted by the editor)

Learning Domain Structure Through Probabilistic Policy Reuse in Reinforcement Learning

Fernando Fernández · Manuela Veloso

Received: date / Accepted: date

Abstract Policy Reuse is a transfer learning approach to improve a reinforcement learner with guidance from previously learned similar action policies. The method uses the past policies as a probabilistic bias where the learner chooses among the exploitation of the ongoing learned policy, the exploration of random unexplored actions, and the exploitation of past policies. In this work we demonstrate that Policy Reuse further contributes to the learning of the structure of a domain. Interestingly and almost as a side effect, Policy Reuse identifies classes of similar policies revealing a basis of *core-policies* of the domain. We demonstrate theoretically that, under a set of conditions to be satisfied, reusing such a set of core-policies allows us to bound the minimal expected gain received while learning a new policy. In general, Policy Reuse contributes to the overall goal of lifelong reinforcement learning, as (i) it incrementally builds a policy library; (ii) it provides a mechanism to reuse past policies; and (iii) it learns an abstract domain structure in terms of core-policies of the domain.

Keywords Probabilistic Policy Reuse · Transfer Learning · Reinforcement Learning · Domain Structure Learning

F. Fernández
Universidad Carlos III de Madrid
Tel.: +34 916248842
Fax: +34 916249129
E-mail: ffernand@inf.uc3m.es

M. Veloso
Carnegie Mellon University
Tel.: +1 4122681474
Fax: +1 4122684801
E-mail: mmv@cs.cmu.edu

1 Introduction

Reinforcement Learning (RL) [1,2] is a powerful technique for learning to solve different kinds of tasks. Solving the task consists of learning a near-optimal policy for such task. In the best case, such policy will be near-optimal for the task, i.e., will maximize the long term sum of the rewards obtained. The learning process is based on a trial and error process guided by reward signals received from the environment. Classical RL algorithms as Q-Learning [3] rely on an intensive exploration of the action and state spaces. Due to the "curse of dimensionality" of such spaces in complex domains, solving a task typically requires an extensive interaction of the learning agent with the environment.

Although the cost (time, resources, etc.) of such a learning process may be very high, sometimes the task can be tackled and successfully solved [4,5]. There have been many different efforts to address the complexity of learning. Reusing the knowledge acquired in the current learning process when solving future problems, so the cost of future learning processes is reduced, is an appealing idea. In RL, several efforts have been done in this line, like the transfer of value functions [6], the reuse of options [7] and the learning of hierarchical decompositions of factored Markov Decision Processes (MDPs) [8].

In this manuscript, we report on Probabilistic Policy Reuse, an approach for transfer learning based on the reuse of similar action policies. It is based on our research in the related areas of Symbolic Plan Reuse [9] and Extended Rapidly-exploring Random Trees (E-RRT) [10]. Planning by analogical reasoning provides a method for symbolic plan reuse. However, when reusing a past plan, if a step becomes invalid to use in the new situation, the traditional reuse questions are either (i) to resolve the locally failed step and direct the search to return back to another past plan step, or (ii) to completely abandon the past plan and re-plan from scratch from the failed step directly towards the goal. E-RRT solves this general reuse question by guiding a new plan probabilistically with a past plan. The past experience is effectively used as a *bias* in the new search, and thus solves the general reuse problem in a probabilistic manner.

Learning structure in complex domains is a key challenge for scaling up applications, in particular because of the difficulty in finding similarity metrics to determine commonalities in a complex domain. In this article, we contribute a method to identify equivalence classes of domain states through our developed Policy Reuse. When solving a new problem, Policy Reuse utilizes the past policies as a probabilistic bias where the learner faces three choices: the exploitation of the ongoing learned policy, the exploration of random unexplored actions, and the exploitation of past policies. As a past policy becomes relevant to solving a new task, such effective reuse *reveals* the similarity between the past and new task. Domain structure is then incrementally learned through Policy Reuse, as we present.

Therefore, a side-effect of Policy Reuse is its capability to identify classes of similar policies revealing a basis of *core-policies* of the domain. That allows to build a library of policies to be reused in the future, by using the PLPR algo-

rithm (Policy Library through Policy Reuse). In this work we contribute new theoretical results, and we show that, under a set of conditions to be satisfied, reusing such a set of core-policies allows us to bound the minimal expected gain received while learning a new policy. We introduce new definitions, as the δ-Basis-Library of a domain, which defines a library of core policies which is large enough to successfully obtain accurate results in a Policy Reuse process.

In this paper, we also include additional evaluations over classical exploration strategies (like ϵ-greedy and Boltzmann) to show the advantages of Policy Reuse in the grid domain. Policy Reuse can also be applied to domains potentially more complex, such as the Keepaway domain [5]. The challenge in Keepaway is to transfer learned knowledge from simpler (although continuous) to larger state and action spaces, e.g., from a Keepaway problem with some number of teammates and opponents to a new one with larger number of agents. The use of Policy Reuse for transfer learning among different state and action spaces (typically called inter-task transfer [11]), and its evaluation in the Keepaway can be found in the literature [12–14]. Variations of Policy Reuse algorithms can also be found for multi-robot reconfiguration [15] and learning from demonstration, also in the Keepaway [16]. In this work we use a grid-based domain that allows us to highlight some properties which are more difficult to represent in other domains. The same domain has been used in other works [17].

In summary, the main contributions of this manuscript are, on the one hand, to show empirical and theoretical results about how the similarity metric among policies work, why it is useful to select the policy to reuse from a set of past policies, and how to use it to bound the gain that can be obtained by reusing a policy library. On the other hand, to demonstrate how Policy Reuse learns the domain structure of a domain in terms of libraries of core-policies, which can be used in future learning tasks.

This manuscript is organized as follows. Section 2 summarizes relevant related work. Section 3 introduces Policy Reuse in the scope of Reinforcement Learning, and formalizes the concepts of task, domain, and gain. Section 4 defines the π-reuse exploration strategy, a similarity metric among policies, and the PRQ-Learning algorithm. Section 5 presents the PLPR algorithm, and provides theoretical and empirical results that demonstrate the capability of the algorithm to build the basis of a domain as a set of core-policies, and bound the sub-optimality of the transfer learning. Section 6 shows the empirical results. Finally, Section 7 summarizes the main conclusions of this work.

2 Related Work

Policy Reuse is a transfer learning method. It uses past policies to balance among exploitation of the ongoing learned policy, exploration of random actions, and exploration toward the past policies. The exploration vs. exploitation problem defines whether to explore new or exploit the knowledge already

acquired. The limits are defined by the random and the greedy strategies, and several can be found in between, as ϵ-greedy and Boltzmann [2]. Directed exploration strategies memorize exploration-specific knowledge that is used for guiding the exploration search [18]. These strategies are based in heuristics that bias the learning so unexplored states tend to have a higher probability of being explored than recently visited ones. These strategies only use knowledge obtained in the current learning process.

Several methods aim at improving learning by introducing additional knowledge into the exploration process. Advice rules [19] define the actions to be preferred in different sets of states. In this case, the source of the advice rules is the user, which is the source of exploration knowledge in many other approaches [20]. Different knowledge sources can be used, as a mentor, from which policies can be learned by imitation [21]. In the previous cases, as in Policy Reuse, the advice is about policies rather than Q values.

Transfer learning refers to the injection of knowledge from previously solved tasks. Memory guided exploration [22] incorporates knowledge from a past policy in a new exploration process by weighting the Q values associated to the new and the past policy. However, that requires that the values of both Q functions are homogeneous and a perfect mapping between the past and the new Q function. The problem can be solved by weighting the probability of selecting each action, instead of the actual Q values [23]. In any case, the choice of a correct weight decay to balance correctly the use of the past and the new policy relies on the designer.

Transfer learning, as knowledge reuse across different learning tasks, can be performed by initializing the Q-values of a new episode with previously learned Q-values [24, 25]. However, if the source and target tasks are very different, transfer learning may require expert knowledge to decide on the feasibility of the transfer, and on the mapping between actions and states from the source and target tasks [26]. Some methods try to solve this problem through a study of actions correlations [27], through state abstraction [28], or by defining the relationships between the state variables of the source and target MDP's [29]. Value function transfer is an alternative but it is restricted to previous learning processes performed also through a value function. Furthermore, they do not focus on the case where several tasks have been previously solved (several value functions have been learned) and are susceptible to be reused.

A different way of introducing previous knowledge is by executing macro-actions or sub-policies. For instance, some algorithms use macro-actions to learn new action policies in Semi-Markov Decision Processes (SMDPs), as it is the case of TTree [30]. These macros can also be defined using a relational language, and learned using Inductive Logic Programming (ILP) techniques [31]. Options can also be used in SMDPs [7]. They require the set of states from which they can be executed, an end condition and the behavior of the option. Such a behavior can be learned on line [32], as well as the other components of the option [33]. Other ways to transfer knowledge is through the use of set of rules that summarizes polices [34] or by composing solutions of elemental sequential tasks [35].

Hierarchical RL uses different abstraction levels to organize subtasks [36], and some approaches are able to learn such a hierarchy [37]. The methods for learning hierarchies or options capture the structure of the domain. Some related algorithms are SKILL [38], which discovers partially defined policies that arise in the context of multiple tasks in the same domain, and L-Cut, which discovers subgoals and corresponding sub-policies [39]. Sub-policies can suboptimally solve a task with computable bounds [40]. Other methods incrementally build a cache of policies for a decomposed MDP [41], but also following a hierarchical approach.

Probabilistic Policy Reuse establishes a huge difference with previous works based on options, macro-actions or hierarchical RL. Those methods are built on a basis where, once a sub-policy is selected, it is followed until an end condition associated to the sub-policy is satisfied or it suffers an external interruption. In our case, past policies provide a bias, and the learning agents interlace the execution of actions suggested by the new and past policies probabilistically. Policy Reuse never executes complete, nor even partial policies, but in each step decides whether to execute an action suggested by one of the past or new policies. This fact avoids the definition of both the conditions when a sub-policy must be executed, nor the conditions when the execution of a sub-policy must be interrupted.

A primary difference of Policy Reuse and using macro-actions, assuming flat macro-actions similar to the exploratory actions used in Policy Reuse, is that Policy Reuse does not learn values for such exploratory actions, but it learns values for the primitive actions. From the values of the primitive actions, the ground policy is derived.

A main contribution of Policy Reuse with respect to other previous approaches is that Policy Reuse does not assume that transferred knowledge is positive. This assumption makes other methods to believe that the transferred knowledge will be useful, as it is highlighted in a previous survey [42]. Policy Reuse owns mechanisms to measure the utility of the transferred policies, and capabilities to decide when to reuse them or not.

3 Policy Reuse in Reinforcement Learning

Reinforcement Learning problems are typically formalized using Markov Decision Processes (MDPs). An MDP is a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} >$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{T}$ is a stochastic state transition function, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \Re$, and $\mathcal{R}$ is a stochastic reward function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \Re$. RL assumes that $\mathcal{T}$ and $\mathcal{R}$ are unknown.

We focus on RL domains where different *tasks* can be solved. The MDP's formalism is not expressive enough to represent all the concepts involved in knowledge transfer [43], so we define domain and task separately to handle different tasks executed in the same domain. We introduce a task as a specific reward function, while the other concepts, $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{T}$ stay constant for all the tasks in the same domain.

Definition 1. *A Domain $\mathcal{D}$ is a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T} >$, where $\mathcal{S}$ is the set of all states; $\mathcal{A}$ is the set of all actions; and $\mathcal{T}$ is a state transition function, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \Re$.*

Definition 2. *A task Ω is a tuple $< \mathcal{D}, \mathcal{R}_\Omega >$, where $\mathcal{D}$ is a domain; and $\mathcal{R}_\Omega$ is the reward function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \Re$.*

We assume that we are solving episodic tasks. A trial or episode starts by locating the learning agent in a random position in the environment. Each episode finishes when the agent reaches a goal state or when it executes a maximum number of steps, H. [1] The agent's goal is to maximize the expected average reinforcement per episode, W, as defined in equation 1:

$$W = \frac{1}{K} \sum_{k=0}^{K} \sum_{h=0}^{H} \gamma^h r_{k,h} \tag{1}$$

where γ ($0 \leq \gamma \leq 1$) reduces the importance of future rewards, and $r_{k,h}$ defines the immediate reward obtained in the step h of the episode k, in a total of K episodes.

An action policy, Π, is a function $\Pi : \mathcal{S} \to \mathcal{A}$ that defines how the agent behaves. If the action policy was created to solve a defined task, Ω, we call that action policy Π_Ω. The gain, or average expected reward, received when executing an action policy Π in the task Ω is called W_Ω^Π. Finally, an optimal action policy for solving the task Ω is called Π_Ω^*. The action policy Π_Ω^* is optimal if $W_\Omega^{\Pi_\Omega^*} \geq W_\Omega^\Pi$, for all policy Π in the space of all possible policies when $K \to \infty$. Action policies can be represented using the action-value function, $Q^\Pi(s, a)$, which defines for each state $s \in \mathcal{S}$, $a \in \mathcal{A}$, the expected reward that will be obtained if the agent starts to act from s, executing a, and after it follows the policy Π. So, the RL problem is mapped to learning the function $Q^\Pi(s, a)$ that maximizes the expected gain. The learning can be performed using different algorithms, such as Q-Learning [3].

The goal of Policy Reuse is to use different policies, which solve different tasks, to bias the exploration process of the learning of the action policy of another similar task in the same domain. We call Policy Library to the set of past policies, as defined next.

Definition 3. *A Policy Library, L, is a set of n policies $\{\Pi_1, \ldots, \Pi_n\}$. Each policy $\Pi_i \in L$ solves a task $\Omega_i =< \mathcal{D}, \mathcal{R}_{\Omega_i} >$, i.e., each policy solves a task in the same domain.*

The previous definition does not restrict the characteristics of the tasks (they may be repeated), nor the characteristics of the policies (they may be sub-optimal),although optimality or near-optimality could affect the reuse process. The scope of Policy Reuse is summarized as: we want to solve the task Ω, i.e., learn Π_Ω^*; we have previously solved the set of tasks $\{\Omega_1, \ldots, \Omega_n\}$ with n policies stored as a Policy Library, $L = \{\Pi_1, \ldots, \Pi_n\}$; how can we use the policy library, L, to learn the new policy, Π_Ω^*?

[1] Constraining Policy Reuse to episodic tasks or to limit the number of steps of an episode, are a relaxation but not a requirement to apply Policy Reuse, which has demonstrated to perform accurately in domains with undefined length of the episodes [44].

Policy Reuse answers this question by adding the past policies into a learning episode as a probabilistic exploration bias. We define an exploration strategy able to bias the exploration process towards the policies of the Policy Library, and a method to estimate the utility of reusing each of them and to decide whether to reuse them or not. Furthermore, Policy Reuse provides an efficient method to construct the Policy Library. We now detail the Policy Reuse approach.

4 Reusing Past Policies

In this section we describe the basic algorithms of Policy Reuse. We first describe how to reuse just one past policy. Then, we show how to reuse a set of past policies. Last, in this section we describe in depth the results obtained in a grid navigation domain.

4.1 The π-reuse Exploration Strategy

The π-reuse strategy is an exploration strategy able to bias a new learning process with a past policy. Let Π_{past} be the past policy to reuse and Π_{new} the new policy to be learned. We assume that we are using a direct RL method to learn the action policy, so we are learning the related Q function. Any RL algorithm can be used to learn the Q function, and Sarsa(λ) and Q(λ) have been applied [13,14].

The goal of π-reuse is to balance random exploration, exploitation of the past policy, and exploitation of the new policy, as represented in Equation 2.

$$a = \begin{cases} \Pi_{past}(s) & \text{w/prob. } \psi \\ \epsilon - greedy(\Pi_{new}(s)) & \text{w/prob. } (1-\psi) \end{cases} \quad (2)$$

The π-reuse strategy follows the past policy with probability ψ, and it exploits the new policy with probability of $1-\psi$. As random exploration is always required, it follows the new policy using an ϵ-greedy strategy.

Table 1 shows a procedure describing the π-reuse strategy integrated with the Q-Learning algorithm. The procedure gets as an input the past policy Π_{past}, the number of episodes K, the maximum number of steps per episode H, and the ψ parameter. An additional υ parameter is added to decay the value of ψ in each step of the learning episode. The procedure outputs the Q function, the policy, and the average gain obtained in the execution, W, which will play an important role in similarity assessment, as the next sections present. The variable ψ_h keeps the value of $\upsilon^h\psi$ in each step of each episode.

4.2 A Similarity Function Between Policies

The exploration strategy π-reuse, as defined in Table 1, returns the learned policy Π_{new}, and the average gain obtained in its learning process, W. Let W_i

π-reuse $(\Pi_{past}, K, H, \psi, \upsilon)$.
Initialize $Q^{\Pi_{new}}(s,a) = 0$, $\forall s \in \mathcal{S}, a \in \mathcal{A}$
For $k = 0$ to $K-1$
Set the initial state, s, randomly.
Set $\psi_1 \leftarrow \psi$
for $h = 1$ to H
With a probability of ψ_h, $a = \Pi_{past}(s)$
With a probability of $1-\psi_h$, $a = \epsilon$-greedy$(\Pi_{new}(s))$
Receive the next state s', and reward, $r_{k,h}$
Update $Q^{\Pi_{new}}(s,a)$, and therefore, Π_{new}:
$Q^{\Pi_{new}}(s,a) \leftarrow (1-\alpha)Q(s,a)^{\Pi_{new}} +$
$\alpha[r + \gamma \max_{a'} Q^{\Pi_{new}}(s',a')]$
Set $\psi_{h+1} \leftarrow \psi_h \upsilon$
Set $s \leftarrow s'$
$W = \frac{1}{K}\sum_{k=0}^{K}\sum_{h=0}^{H}\gamma^h r_{k,h}$
Return W, $Q^{\Pi_{new}}(s,a)$ and Π_{new}

Table 1 π-reuse Exploration Strategy.

be the gain obtained while executing the π-reuse exploration strategy, reusing the past policy Π_i, and using a parameter vector $\boldsymbol{\theta}$ that encapsulates all the parameters of the exploration strategy (K, H, ψ and υ, defined in Table 1). We can use such value to measure the usefulness of reusing the policy Π_i to learn the new policy Π_{new}. The next definitions formalize this idea.

Definition 4. *Given a policy Π_i that solves a task $\Omega_i =< \mathcal{D}, R_i >$, and a new task $\Omega =< \mathcal{D}, R_\Omega >$, the Reuse Gain of the policy Π_i on the task Ω, $W_i^{\boldsymbol{\theta}}$, is the gain obtained when applying the π-reuse exploration strategy with the policy Π_i and a parameter vector $\boldsymbol{\theta}$ to learn the policy Π.*

Vector $\boldsymbol{\theta}$ plays an important role, since the reuse gain obtained when reusing a policy depends on such a vector. However, we can assume that such parameter vector must be fixed "a priory" or after some tuning. Therefore, in the rest of the paper we will assume that such vector is fixed. To simplify the notation, we will also eliminate it from the formulation, and we will use W_i, instead of $W_i^{\boldsymbol{\theta}}$.

Then, given a parameter vector $\boldsymbol{\theta}$, the most useful policy to reuse, Π_k, from a Library Policy, $L = \{\Pi_1, \ldots, \Pi_n\}$, is the one that maximizes the Reuse Gain when learning such a task, as defined in equation 3:

$$\Pi_k = \arg_{\Pi_i} \max(W_i), i = 1, \ldots, n \tag{3}$$

To solve this equation we need to compute the Reuse Gain for all the past policies. Interestingly, such a gain can be estimated on-line at the same time that the new policy is computed. This idea is formalized in the PRQ-Learning algorithm.

4.3 The PRQ-Learning Algorithm

The goal of the PRQ-learning algorithm is to solve a task Ω, i.e., to learn an action policy Π_Ω. We have a Policy Library $L = \{\Pi_1, \ldots, \Pi_n\}$ composed of n past optimal policies that solve n different tasks respectively. Then two main questions need to be answered: (i) given the set of policies $\{\Pi_\Omega, \Pi_1, \ldots, \Pi_n\}$, which consists of the policies in the Policy Library plus the ongoing learned policy, what policy (say Π_k) is exploited? (ii) once a policy is selected, what exploration/exploitation strategy is followed?

The answer to the first question is as follows: let W_i be the Reuse Gain of the policy Π_i on the task Ω. Also, let W_Ω be the average reward that is received when following the policy Π_Ω greedily. The solution we introduce consists of following a softmax strategy using the values W_Ω and W_i, as defined in equation 4, with a temperature parameter τ. This value is also computed for Π_0, which we assume to be Π_Ω. Equation 4 provides a way of deciding, π_k, to select to exploit.

$$\Pi_k = \arg_{\Pi_j, 0 \leq j \leq n} \max P(\Pi_j), \text{ where } P(\Pi_j) = \frac{e^{\tau W_j}}{\sum_{p=0}^{n} e^{\tau W_p}}, \forall \Pi_j, 0 \leq j \leq n \tag{4}$$

The problem of selecting what policy to reuse in the PRQ-Learning is similar to a non-stationary k-armed bandit problem. Most works in non-stationary k-armed bandit problems try to detect when the change in the distributions occurs, and then to re-learn with classical stationary approaches [45].

The answer to the second question (what exploration strategy to follow once a policy is chosen) is an heuristic that depends on the selected policy. If the policy chosen is Π_Ω, the algorithm follows a completely greedy strategy. However, if the policy chosen is Π_i (for $i = 1, \ldots, n$), the π-reuse action selection strategy, defined in previous section, is followed instead. In this way, the Reuse Gain of each of the past policies can be estimated on-line with the learning of the new policy. Thus, the values required in Equation 4 are continuously updated each time a policy is used.

All these ideas are formalized in the PRQ-Learning algorithm (Policy Reuse in Q-Learning) shown in Table 2. The algorithm gets as input: a new task to solve Ω; the policy library L; the temperature parameter of the softmax policy selection equation τ, and a decay parameter $\Delta\tau$; and a set of previously defined parameters: $K, H, \psi, \upsilon, \gamma, \alpha$.

The algorithm initializes the new Q function to 0, as well as the estimated reuse gain of the policies in the library. Then the algorithm executes the K episodes iteratively. In each episode, the algorithm decides which policy to follow. In the first iteration, all the policies have the same probability to be chosen, given that all W_i values are initialized to 0. Once a policy is chosen, the algorithm uses it to solve the task, updating the Reuse Gain for such a policy with the reward obtained in the episode, and therefore, updating the probability to follow each policy. The policy being learned can also be

PRQ-Learning$(\Omega, L, \tau, \Delta\tau, K, H, \psi, \upsilon, \gamma, \alpha)$

- Given:
  1. A new task Ω we want to solve
  2. A Policy Library $L = \{\Pi_1, \ldots, \Pi_n\}$
  3. An initial value of the temperature parameter, τ, and an incremental size, $\Delta\tau$, for the Boltzmann policy selection strategy
  4. A maximum number of episodes to execute, K
  5. A maximum number of steps per episode, H
  6. The parameters ψ and υ for the π-exploration strategy
  7. The parameters γ and α for the Q-learning update equation
- Initialize:
  1. $Q_\Omega(s, a) = 0, \forall s \in \mathcal{S}, a \in \mathcal{A}$
  2. Initialize W_Ω to 0
  3. Initialize W_i to 0
  4. Initialize the number of episodes where policy Π_Ω has been chosen, $U_\Omega = 0$
  5. Initialize the number of episodes where policy Π_i has been chosen, $U_i = 0, \forall i = 1, \ldots, n$
- For $k = 1$ to K do
  - Choose an action policy, Π_k, assigning to each policy the probability of being selected computed by the following equation (equation 4):

$$P(\Pi_j) = \frac{e^{\tau W_j}}{\sum_{p=0}^{n} e^{\tau W_p}}$$

    where W_0 is set to W_Ω, and $0 \leq j \leq n$

$$\Pi_k = \arg_{\Pi_j, 0 \leq j \leq n} \max P(\Pi_j)$$

  - Execute the learning episode k
    - If $\Pi_k = \Pi_\Omega$, execute a Q-Learning episode following a fully greedy strategy
    - Otherwise, use the π-reuse exploration strategy to reuse Π_k, i.e., call π-reuse$(\Pi_k, 1, H, \psi, \upsilon)$
    - In any case, receive the reward obtained in that episode, say R, and the updated Q function, $Q_\Omega(s, a)$
  - Set $W_k = \frac{W_k U_k + R}{U_k + 1}$
  - Set $U_k = U_k + 1$
  - Set $\tau = \tau + \Delta\tau$
- Return the policy derived from $Q_\Omega(s, a)$

Table 2 PRQ-Learning.

chosen, although in the initial steps it behaves as a random policy, given that the Q values are initialized to 0. While new updates are performed over the Q function, it becomes more accurate, and receives higher rewards when executed. After executing several episodes, it is expected that the new policy obtains higher gains than reusing the past policies, so it will be chosen most of the time.

5 Building a Library of Policies

This section describes the *PLPR* algorithm (Policy Library through Policy Reuse), an algorithm to build a library of policies. The algorithm is based

on an incremental learning of policies that solve different tasks. Notice that we are assuming that the tasks that the algorithm will be asked to solve are unknown a priory, and are given in a sequential way. Otherwise, a method to learn them in parallel could be applied [46].

5.1 The PLPR Algorithm

The algorithm works as follows. Initially, the Policy Library is empty, $PL = \emptyset$. Then, the first task, say Ω_1, needs to be solved, so the first policy, say Π_1, is learned. To learn the first policy, any exploration strategy could be used but the policy reuse strategy π-reuse, given that there is not any available policy to reuse. Π_1 is added to the Policy Library, so $PL = \{\Pi_1\}$. When a second task needs to be solved, the PRQ-Learning algorithm is applied, reusing Π_1. Thus, Π_2 is learned. Then, we need to decide whether to add Π_2 to the Policy Library or not. This decision is based on how similar Π_1 is to Π_2, following the equation 5. In the equation, W_2 is the average gain obtained when following Π_2 greedily, and W_1 is the Reuse Gain of Π_1 on task Ω_2. Both values are computed in the execution of the PRQ-Learning algorithm, so no additional computations are required.

$$d_{\rightarrow}(\Pi_1, \Pi_2) = W_2 - W_1 \tag{5}$$

This distance metric estimates how similar Π_1 is to Π_2. We define this distance not by direct comparisons between the policies, but comparing the result of applying them. In our case, if Π_1 is very similar to Π_2, i.e., $d_{\rightarrow}(\Pi_1, \Pi_2)$ is close to 0, to include the second policy in the library is unnecessary. However, if the distance is large, Π_2 is included. Therefore, we can introduce a new concept, δ-similarity, as follows.

Definition 6. *Given a policy, Π_i that solves a task $\Omega_i =< \mathcal{D}, R_i >$, a new task $\Omega =< \mathcal{D}, R_\Omega >$, and its respective optimal policy, Π. Π is δ-similar to Π_i (for $0 \leq \delta \leq 1$) if $W_i > \delta W^*_\Omega$, where W_i is the Reuse Gain of Π_i on task Ω and W^*_Ω is the average gain obtained in Ω when an optimal policy is followed.*

The interesting property of this concept is that for any optimal policy Π, if we know a past policy which is δ-similar to it, we know that such optimal policy can be easily learned just by applying the π-reuse algorithm with the past policy, and that the gain obtained in the learning process (the reuse gain) will be at least δ times the maximum gain in such a task. From this definition, we can formalize the concept of δ-similarity with respect to a Policy Library, L, as follows.

Definition 7. *Given a Policy Library, $L = \{\Pi_1, \ldots, \Pi_n\}$ in a domain $\mathcal{D}$, a new task $\Omega =< \mathcal{D}, R_\Omega >$, and its respective optimal policy, Π. Π is δ-similar with respect to L iff $\exists \Pi_i$ such as Π is δ-similar to Π_i, for $i = 1, \ldots, n$.*

Thus, if we know that a policy Π is δ-similar with respect to a Policy Library L, we know that the policy Π can be easily learned by reusing the policies in L.

The PLPR algorithm is described in Table 3. It is executed each time that a new task needs to be solved. It inputs the Policy Library and the new task to solve, and outputs the learned policy and the updated Policy Library.

PLPR Algorithm

- Given:
  1. A Policy Library, L, composed of n policies, $\{\Pi_1, \ldots, \Pi_n\}$
  2. A new task Ω we want to solve
  3. A δ parameter
- Execute the PRQ-Learning algorithm, using L as the set of past policies. Receive from this execution Π_Ω, W_Ω and W_{max}, where:
  - Π_Ω is the learned policy
  - W_Ω is the average gain obtained when the policy Π_Ω was followed
  - $W_{max} = \max W_i$, for $i = 1, \ldots, n$
- Update PL using the following equation:

$$L = \begin{cases} L \cup \{\Pi_\Omega\} & \text{if } W_{max} < \delta W_\Omega \\ L & otherwise \end{cases} \quad (6)$$

Table 3 PLPR Algorithm.

Equation 6 is the update equation for the Policy Library, derived from equation 5. It requires the computation of the most similar policy, which is the policy Π_j such as $j = \arg_i \max W_i$, for $i = 1, \ldots, n$. The gain that will be obtained by reusing such a policy is called W_{max}. The new policy learned is inserted in the library if W_{max} is lower than δ times the gain obtained by using the new policy (W_Ω), where $\delta \in [0, 1]$ defines the similarity threshold, i.e., whether the new policy is δ-similar with respect to the Policy Library.

The parameter δ has an important role. If it receives a value of 0, the Policy Library stores only the first policy learned, given that the average gain obtained by reusing it will be greater than zero in most cases, due to the positive rewards obtained by chance. If $\delta = 1$, most of the policies learned are inserted, due to the fact that $W_{max} < W_\Omega$, given that W_Ω is maximum if the optimal policy has been learned. Different values in the range $(0, 1)$ provide different sizes of the library, as will be demonstrated in the experiments. Thus, δ defines the size, and therefore the resolution, of the library.

5.2 Suboptimality of Policy Reuse

The PLPR algorithm has an interesting "side-effect," namely the learning of the *structure* of the domain. As the Policy Library is initially empty, and a new policy is included only if it is different enough with respect to the previously stored ones, depending on the threshold δ, when the policies stored are *sufficiently representative* of the domain, no more policies are stored. Thus, the obtained library can be considered as the *Basis-Library* of the domain, and

the stored policies can be considered as the *core policies* of such domain. In the following, we introduce the formalization of these concepts.

Definition 8. *A Policy Library, $L = \{\Pi_1, \ldots, \Pi_n\}$ in a domain $\mathcal{D}$ with a distribution of tasks $\mathcal{T}$, is a δ-Basis-Library of the domain $\mathcal{D}$ iff: (i) $\nexists \Pi_i \in L$, such as Π_i is δ-similar with respect to $L - \Pi_i$; and (ii) the rest of policies Π in the space of all the possible policies in $\mathcal{D}$ are δ-similar with respect to L.*

Here we introduce the idea of a distribution of tasks, $\mathcal{T}$, to limit the distribution of rewards functions. This distribution will be important when we define the conditions to build a δ-Basis-Library of a domain.

Definition 9. *Given a δ-Basis-Library, $L = \{\Pi_1, \ldots, \Pi_n\}$ in a domain $\mathcal{D}$, a new task $\Omega =< \mathcal{D}, R_\Omega >$, each policy $\Pi \in L$ is a δ-Core Policy of the domain $\mathcal{D}$ in L.*

The proper computation of the Reuse Gain for each past policy in the PRQ-Learning algorithm plays an important role, since it allows the algorithm to compute the most similar policy, its reuse distance and therefore, to decide whether to add the new policy to the Policy Library or not. If the reuse gain is not correctly computed, the basis library will not be either. Thus, we introduce a new concept that measures how accurate the estimation of the reuse gain is.

Definition 10. *Given a Policy Library, $L = \{\Pi_1, \ldots, \Pi_n\}$ in a domain $\mathcal{D}$, and a new task $\Omega =< \mathcal{D}, R_\Omega >$, let us assume that the PRQ-Learning algorithm is executed, and it outputs the new policy Π_Ω, the estimation of the optimal gain $\hat{W}_{\Pi_\Omega}$, and the estimation of the Reuse Gain of the most similar policy, say $\hat{W}_{max}$. We say that the PRQ-Learning algorithm has been Properly Executed with a confidence factor η ($0 \leq \eta \leq 1$), if Π_Ω is optimal to solve the task Ω, and the error in the estimation of both parameters is lower than a factor of η, i.e.:*

$$\begin{gathered} \hat{W}_{max} \geq \eta W_{max} \text{ and } \eta \hat{W}_{max} \leq W_{max} \text{ and} \\ \hat{W}_{\Pi_\Omega} \geq \eta W_{\Pi_\Omega} \text{ and } \eta \hat{W}_{\Pi_\Omega} \leq W_{\Pi_\Omega} \end{gathered} \tag{7}$$

where W_{max} is the actual value of the Reuse Gain of the most similar policy and W_{Π_Ω} is the actual gain of the obtained policy.

Thus, if we say that the PRQ-Learning algorithm has been Properly Executed with a confidence of 0.95, we can say, for instance, that the estimated Reuse Gain, $\hat{W}_{max}$ of the most similar policy, has a maximum deviation over the actual Reuse Gain of 5%. The proper execution of the algorithm depends on how accurate the parameters selection is. Such a parameter selection depends on the domain and the task, so no general guidelines can be provided. The definition requires that that Π_Ω is optimal to solve the task Ω, which theoretically may require an infinite number of episodes. However, in practice, optimal policies may be obtained in a finite number of episodes or, at least, the suboptimality could be bounded.

The previous definition allows us to enumerate the conditions that make the PLPR algorithm build a δ-Basis-Library, as described in the following theorem.

Theorem 1. *The PLPR algorithm builds a δ-Basis-Library of a domain $\mathcal{D}$ for a task distribution $\mathcal{T}$ if (i) the PRQ-Learning algorithm is Properly*

Executed with a confidence of 1; (ii) the Reuse Distance is symmetric; and (iii) the PLPR algorithm is executed infinite times over random tasks in the distribution $\mathcal{T}$.

Proof. The proper execution of the PRQ-Learning algorithm ensures that the similarity metric, and all the derived concepts, are correctly computed. The first condition of the definition of δ-Basis-Library can be demonstrated by induction. The base case is when the library is composed of only one policy, given that no policy is δ-similar with respect to an empty library. The inductive hypothesis states that a Policy Library $L_n = \{\Pi_1, \ldots, \Pi_n\}$ is a δ-Basis-Library. Lastly, the inductive step is that the library $L_{n+1} = \{\Pi_1, \ldots, \Pi_n, \Pi_{n+1}\}$ is also a δ-Basis-Library. If the PLPR algorithm has been followed to insert Π_{n+1} in L, we ensure that Π_{n+1} is not δ-similar with respect to L, given this is the condition to insert a new policy in the library, as described in the PLPR algorithm. Furthermore, Π_i (for $i = 1, \ldots, n$) is not δ-similar with respect to $L_{n+1} - \Pi_i$, given that (i) Π_i is not δ-similar with respect to $L_n - \Pi_i$ (for inductive hypothesis); and (ii) Π_i is not δ-similar to Π_{n+1} because the reuse distance is symmetric (by second condition of the theorem), and that ensures that if Π_i is not δ-similar to Π_{n+1}, then Π_{n+1} is not δ-similar to Π_i. Finally, the second condition of the definition of δ-Basis Library becomes true if the algorithm is executed infinite times, which is satisfied by the third condition of the theorem, which also constrain the distribution of tasks for which policies are computed.

The achievement of the conditions of the theorem depends on several factors. The symmetry of the Reuse Distance depends on the task and the domain. The proper execution of the PRQ-Learning algorithm also depends on the selection of the correct parameters for each domain. However, although the previous theorem requires the PRQ-Learning algorithm to be properly executed with a confidence of 1, a generalized result can be easily derived when the confidence is under 1, say η, as the following theorem claims.

Theorem 2. *The PLPR algorithm builds a $(2\eta\delta)$-Basis-Library if (i) the PRQ-Learning algorithm is Properly Executed with a confidence of η; (ii) the Reuse Distance is symmetric; and (iii) the PLPR algorithm is executed infinite times over random tasks.*

Proof. The proof of this theorem only requires a small consideration over the inductive step of the proof of the previous theorem, where a policy Π_{n+1} is inserted in the δ-Core Policy $L_n = \{\Pi_1, \ldots, \Pi_n\}$ following the PLPR algorithm. The policy is added only if it is not δ-similar with respect to L_n. In that case, if the PRQ-Learning algorithm has been properly executed with a confidence of η, we can only ensure that the policy Π_{n+1} is not $(2\eta\delta)$-similar with respect to L_n, because of the error in the estimation of the gains (reuse gain and optimal gain) in the execution of the PRQ-Learning algorithm.

Finally, we define a lower bound of the learning gain that is obtained when reusing a δ-Basis-Library to solve a new task.

Theorem 3. *Given a δ-Basis-Library, $L = \{\Pi_1, \ldots, \Pi_n\}$ of a domain $\mathcal{D}$, and a new task $\Omega =< \mathcal{D}, R_\Omega >$. The average gain obtained, say W_Ω, when learning a new policy Π_Ω to solve the task Ω by properly executing the PRQ-*

Learning algorithm over Ω reusing L with a confidence factor of η is at least $\eta\delta$ times the optimal gain for such a task, W^*_Ω, i.e.,

$$W_\Omega > \eta\delta W^*_\Omega \tag{8}$$

Proof. When executing the PRQ-Learning properly, we reuse all the past policies, obtaining an estimation of their reuse gain. In the definition of Proper Execution of the PRQ-Learning algorithm, the gain generated by the most similar task, say Π_i, was called $\hat{W}_{max}$, which is an estimation of the real one. In the worst case, the gain obtained in the execution of the PRQ-Learning algorithm is generated only by the most similar policy, Π_i, and the gain obtained by reusing any other different policy is 0, i.e., $W_j = 0, \forall \Pi_j \neq \Pi_i$. By the definition of δ-Basis-Library we know that every policy Π in the domain $\mathcal{D}$ is not δ-similar with respect to L. Thus, the most similar policy in L, Π_i is such that its Reuse Gain, W_{max} satisfies $W_{max} > \delta W^*_\Omega$ (by definition of δ-similarity). However, given that we have executed the PRQ-Learning algorithm with a confidence factor of η, the obtained gain W_Ω only satisfies that $W_\Omega \geq \eta W_{max}$ by definition of proper execution of the PRQ-Learning algorithm. Thus, $W_\Omega \geq \eta W_{max}$, and $W_{max} > \delta W^*_\Omega$, so $W_\Omega > \eta\delta W^*_\Omega$.

6 Empirical Results

We use a grid-based robot navigational domain (see Figure 1) with multiple rooms. The environment is represented by walls, free positions and goal areas, all of them of size 1×1. The whole domain is $N \times M$ (24×21 in our case). The actions that the robot can execute are "North," "East," "South," and "West", all of size one. The final position after executing an action is modified by adding a random value that follows a uniform distribution in the range $(-0.20, 0.20)$.

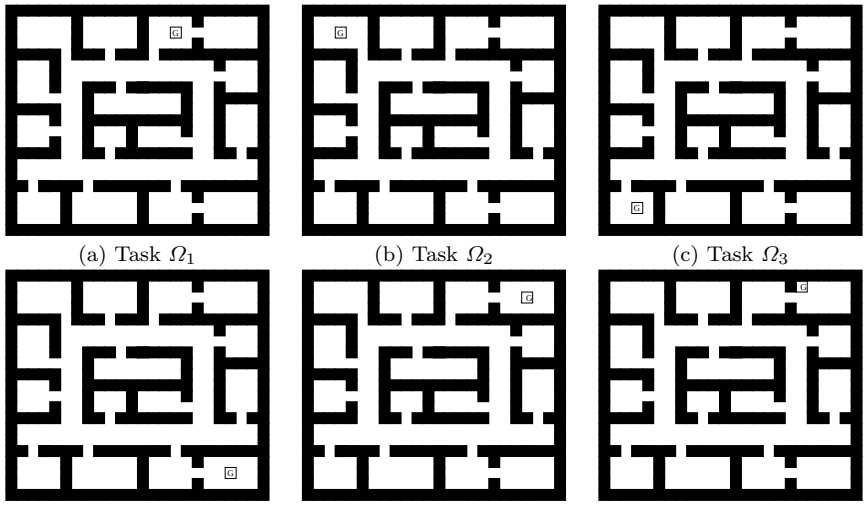
Walls block the robot's motion, i.e., when the robot tries to execute an action that would crash it into a wall, the action keeps the robot in its original position.

The robot knows its location in the space through continuous coordinates (x, y). We assume that we have the optimal uniform discretization of the state space (which consists of 24×21 regions) [2]. The goal in this domain is to reach the area marked with 'G', in a maximum of H actions. When the robot reaches it, it is considered a successful episode, and it receives a reward of 1. Otherwise, it receives a reward of 0.

Figure 1 shows 6 different tasks, Ω_1, Ω_2, Ω_3, Ω_4, Ω_5 and Ω, given that the goal states, and therefore, the reward functions, are different. All these tasks are used in the experiments described in the next sections.

We choose the robot navigation domain for experimentation because it has been widely used in transfer learning papers (e.g., [38,25,43]) and provides

[2] Different methods for function approximation have been successfully applied on this domain [47]. We have simplified the state space representation to a uniform discretization to focus on the study of Policy Reuse.



(a) Task Ω_1 (b) Task Ω_2 (c) Task Ω_3

(d) Task Ω_4 (e) Task Ω_5 (f) Task Ω

Fig. 1 Grid-based Office Domain

us an empirical demonstration of the theoretical results. Policy Reuse has also been succesfully applied in more complex domains, as the Keepaway task in robot soccer, which requires: i) a mapping between tasks that use different state and action spaces; and ii) function approximation methods since the state space is continuous [44,14].

The learning process has been first executed following different exploration strategies that do not use any past policy. Specifically, we have used four different strategies: (i) random; (ii) completely greedy; (iii) an ϵ-greedy (i.e., with probability ϵ follows the greedy strategy, and with probability $(1-\epsilon)$ acts randomly), with an initial value of $\epsilon = 0$, which is incremented by 0.0005 in each episode; (iv) the Boltzmann strategy ($P(a_j) = \frac{e^{\tau Q(s,a_j)}}{\sum_{p=1}^{n} e^{\tau Q(s,a_p)}}$), initializing $\tau = 0$, and increasing it by 5 in each learning episode.

Figure 2 shows the results. Each learning process have been executed 10 times, the average value is shown and error bars show standard deviations. When acting randomly, the average gain in learning is almost 0, given that acting randomly is a very poor strategy. However when a greedy behavior is introduced, (strategy 1-greedy), the curve shows a slow increment, achieving values of almost 0.1. The curve obtained by the Boltzmann strategy does not offer significant improvements. The ϵ-greedy strategy seems to compute an accurate policy in the initial episodes, and it corresponds to the highest average gain at the end of the learning.

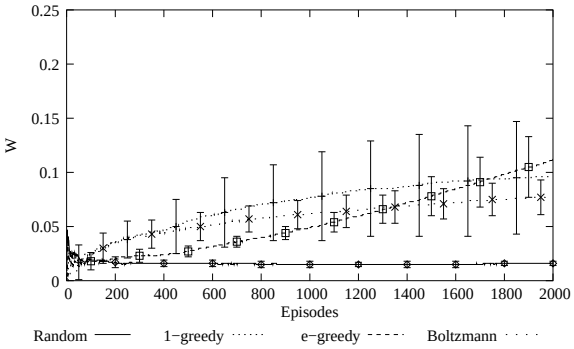


Fig. 2 Results of the learning process for different exploration strategies that learn from scratch.

6.1 Parameter Setting

In the π-reuse exploration strategy, there are three probabilities involved: the probability of exploiting the past policy, i.e., ψ_h, the probability of using the current policy, i.e., $\epsilon(1-\psi_h)$, and the probability of acting randomly, i.e., $(1-\epsilon)(1-\psi_h)$. These probabilities are shown in Figure 3, for input values of $H = 100$, $\psi = 1$ and $\upsilon = 0.95$.

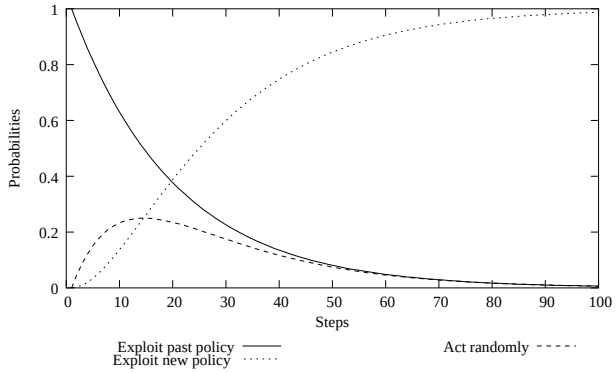


Fig. 3 Evolution of the probabilities of exploring and exploiting in an episode for the π-reuse exploration strategy.

With this parameter setting, the exploration is biased with the past policy mainly in the initial steps of the episode. Assigning to ϵ the value of $(1-\psi_h)$ makes the strategy very greedy in the final steps of each episode, given that we assume that the last steps are the ones that are learned faster (since rewards are also propagated fast from the goal). The figure shows that in the initial steps of each episode, the past policy is exploited. As the number of steps

increases, the probabilities of exploiting the new policy and acting randomly increases. In the final steps of the episode, only the new policy is exploited. The transition from exploiting the past policy and exploiting the new one depends on the υ parameter. If this parameter is low, the transition occurs in the initial steps, while if it is high, the transition is delayed. This parameter setting should be tuned for each domain, in a similar way with the parameters of any other exploration strategy.

In the PRQ-Learning algorithms, the ϵ parameter is set to $1 - \psi_h$ in each step. The rest of parameters $\tau = 0$, and $\Delta\tau = 0.05$, that depends on the number of episodes that we can execute, were obtained empirically after an informal evaluation. Extended analysis on how to set the transfer rate has been performed by different authors [48].

6.2 Computing the Reuse Gain with PRQ-Learning

We use the PRQ-Learning algorithm for learning the task Ω, defined in Figure 1(f). We assume that we have 3 different libraries of policies, so we distinguish three different cases. In the first one, the policy library is $L_1 = \{\Pi_2, \Pi_3, \Pi_4\}$, assuming that the tasks Ω_2, Ω_3 and Ω_4, defined in Figure 1(b), (c) and (d) respectively, were previously solved. All these tasks are very different from the one we want to solve, so their policies are not supposed to be very useful in learning the new one. In the second case, Π_1 is added, so $L_2 = \{\Pi_1, \Pi_2, \Pi_3, \Pi_4\}$. The third case uses the Policy Library $L_3 = \{\Pi_2, \Pi_3, \Pi_4, \Pi_5\}$. The PRQ-Learning algorithm is executed for the three cases. The learning curves are shown in Figure 4.

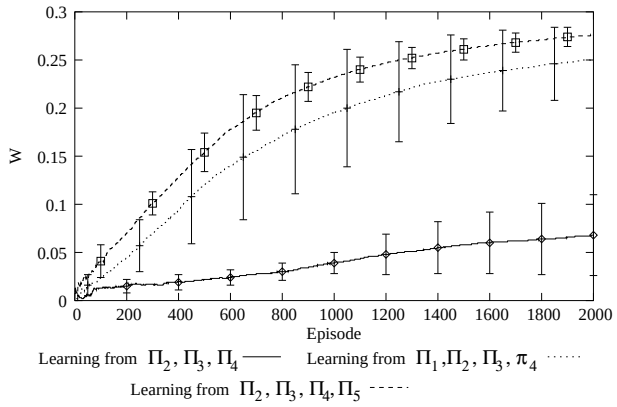


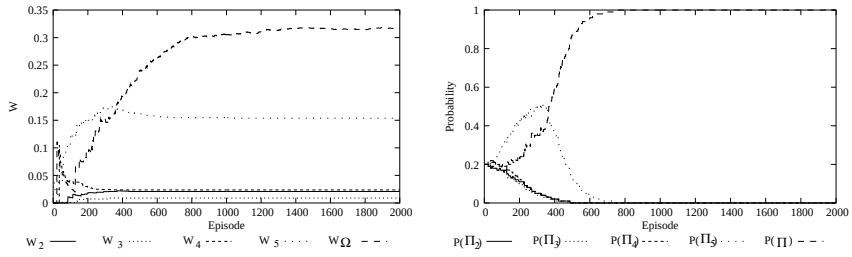
Fig. 4 Learning curve when learning the task of Figure 1(f) reusing different sets of policies.

Figure 4 shows two main conclusions. First, when a very similar policy is included in the set of policies to be reused, the improvement on learning is

very high. For instance, when reusing Π_1 and Π_5, the average gain is greater than 0.1 in only 500 iterations, and more than 0.25 at the end of the episode. Secondly, when no similar policy is available, the learning curve is similar to the results obtained when learning from scratch with the 1-greedy strategy, as shown in Figure 2. Interestingly, that is the strategy followed by PRQ-Learning for the new policy, as defined by the PRQ-Learning algorithm. This demonstrates that the PRQ-learning algorithm has discovered that reusing the past policies is not useful, so it follows the best strategy available, which is to the 1-greedy strategy with the new policy.

The process of learning the most similar policy is illustrated in Figure 5, which reports about the learning process when reusing the Policy Library $L_3 = \{\Pi_5, \Pi_2, \Pi_3, \Pi_4\}$. Figure 5(a) shows the evolution of the Reuse Gain computed for each policy involved, W_5, W_2, W_3, W_4, and the gain W_Ω. On the x axis, the number of episodes is shown, while the y axis shows the gains. Initially, the Reuse Gain of all the policies is set to 0. After a few episodes, W_2, W_3 and W_4 stabilize below 0.05. However, W_5 increases up to 0.15. These values demonstrate that the most similar policy (Π_5) is correctly computed. The gain of the new policy, W_Ω, starts to increase around iteration 100, achieving a value higher than 0.3 by iteration 500, demonstrating that the new policy is very accurate.

The values of the Reuse Gain computed for each policy are used to compute the probability of selecting them in each iteration of the learning process, using the formula introduced in equation 4, and the parameters introduced above (initial $\tau = 0$, and $\Delta\tau = 0.05$). Figure 5(b) shows the evolution of these probabilities. In the initial steps, all the past policies have the same probability of being chosen (0.2) given that the gain of all them is initialized to 0. While the gain values are updated, the probability of Π_5 grows. For the other past policies, the probability decreases down to 0. The probability of the new policy also increases, and after 400 iterations, its bigger than the rest. After a few iterations more, it achieves the value of 1, given that its gain is the highest, as shown in Figure 5(a).



(a)Evolution of W_i (b) Evolution of $P(\Pi_i)$

Fig. 5 Evolution of W_i and $P(\Pi_i)$.

Figure 5(b) demonstrates how the balance between exploiting the past policies or the new one is achieved. It shows how in the initial episodes, the

algorithm chooses to reuse the past policies to find the most similar. Then, it reuses the most similar policy until the new policy is leaned and improves the result of reusing any past policy.

In summary, we can say that the PRQ-learning algorithm has demonstrated to successfully reuse a predefined set of policies, and how it can compute the reuse gain for each of the past policies. The remaining issue consists of demonstrating how the reuse gain is successfully used to build a library of policies and to learn the domain structure.

6.3 Learning the Structure of the Domain

In this experiment, we want to evaluate the PLPR algorithm. With this purpose, we try to learn the action policies for different tasks in the navigation domain. Performing a task consists of trying to solve it $K = 2000$ times. Each of these times is called an episode. Each episode consists of a sequence of actions until the goal is achieved or until the maximum number of actions, $H = 100$, is executed. Notice that there is no separation between learning and test, so the correct balance between exploration and exploitation must be achieved to maximize the average gain in each performance.

In this domain, the task distribution is represented by 50 different tasks, each of them with a different reward function. The different reward functions are derived from goal states located in different positions of the different rooms of the domain, as shown in Figure 6. Notice that the figure does not represent a unique task with 50 different goals, but the 50 different goal areas of the 50 different tasks.

The results provided are the average of 10 different executions, in which the 50 different tasks are sequentially performed following a random order.

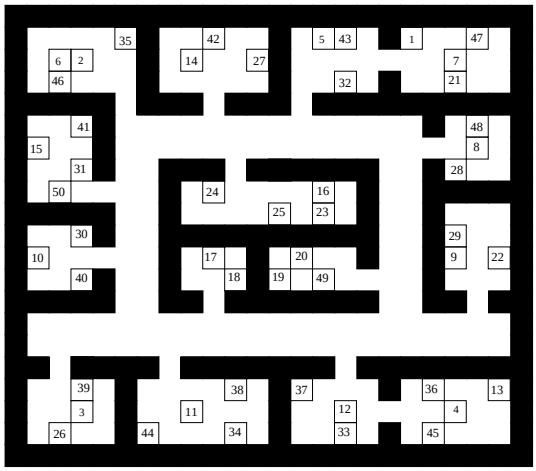


Fig. 6 Navigation Domain.

In these experiments, we use the same parameter setting than in previous experiments; for the Q-Learning algorithm, $\gamma = 0.95$ and $\alpha = 0.05$; for the π-epsilon exploration strategy, $\psi = 1$, $\upsilon = 0.05$, and ϵ is set to $1 - \psi_h$ in each step. In the PRQ-Learning algorithm, τ is initially set to 0, and is increased by 0.05 after each trial.

The first element to study is the size of the Policy Library built while performing the tasks with the PLPR algorithm, i.e., the number of core-policies stored in the Policy Library, shown in Figure 7. The figure shows in the y axis the size of the Policy Library, and in the x axis, the number of tasks performed up to that moment. As introduced above, when $\delta = 0$, only 1 policy is stored. When $\delta = 0.25$, the number of core-policies is around 14. Interestingly, this is very close to the number of rooms in the domain (15). While increasing δ, the number of core-policies increases and when $\delta = 1$, almost all the learned policies are stored.

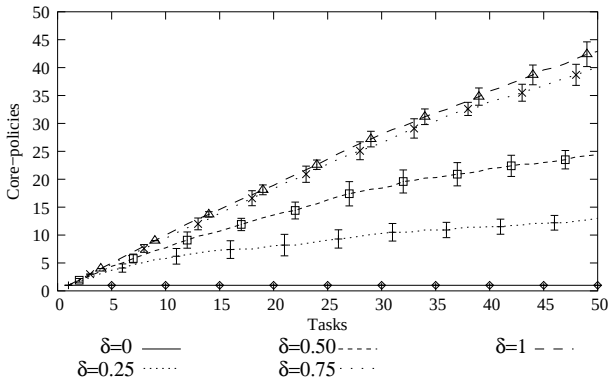


Fig. 7 Number of core-policies obtained.

Figure 8 shows an example of the core-policies obtained in one execution, with $\delta = 0.25$. The figure represents the Policy Library obtained after performing the 50 tasks which, as defined above, is composed of 14 core-policies. In the figure, we assume that a policy is represented by the goal area of the task that it solves. An core-policy is represented also by the goal area, but in this case, the area is shaded. The figure demonstrates that for most of the rooms, one and only one core-policy has been learned. The algorithm has discovered that if two different tasks are given two goal areas in the same room, their respective policies are very similar, so only one of them needs to be stored in the Policy Library. That allows us to say that the structure of the domain has been learned by the PLPR algorithm, and is represented by the core-policies.

Figure 9(a) shows the average gain obtained when performing the 50 different tasks with the PLPR algorithm, for the different values of δ. In most of the cases, $\delta = 0.25, 0.50, 0.75$ and 1, the average gain increases up to more than 0.2, and no significant differences exist between them. Only in the case

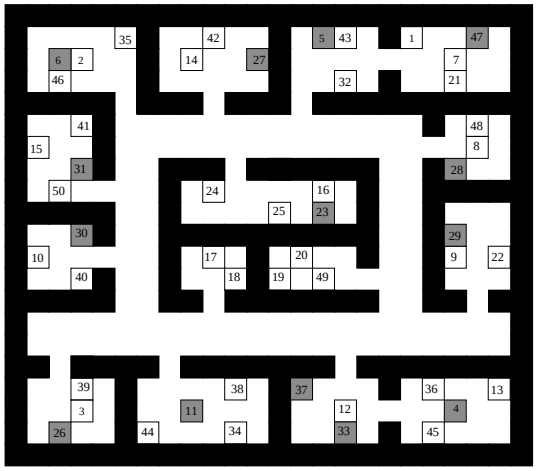
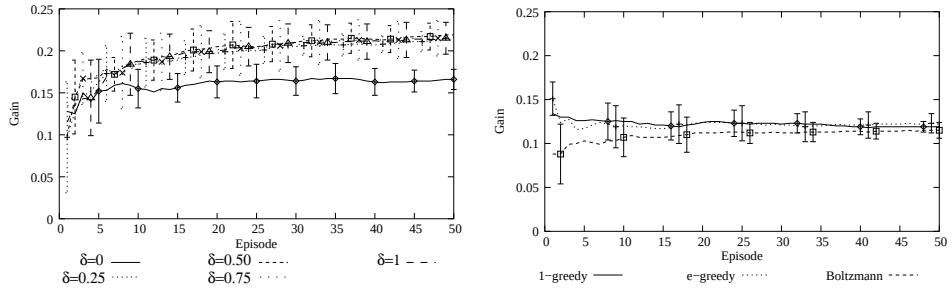


Fig. 8 Core-Policies ($\delta = 0.25$).

of $\delta = 0$, the average gain stays low, around 0.16, given that, as introduced above, $\delta = 0$ generates a Policy Library with only one policy (the first one learned). For comparisons, the same learning process has been executed with different exploration strategies that learn from scratch, and summarized in Figure 9(b).



(a) PLPR for different values of δ (b) Learning from scratch

Fig. 9 Results of PLPR.

The average gain obtained while new policies are learned stabilizes around 0.12 for all the strategies, without very significant differences. This demonstrates that Policy Reuse can obtain an increment of almost a 100% gain in the performance of the 50 tasks over the results obtained when the 50 tasks are learned from scratch. Interestingly, when $\delta = 0$, and only one policy is stored, it also obtains improved results over learning from scratch, due to a good behavior of the π-reuse exploration strategy. That confirms that provid-

ing the learning process with a bias improves the performance, even when that bias may not be the best for all the learning processes.

7 Conclusions

Policy Reuse is a transfer learning method that contributes to Reinforcement Learning with three main capabilities. First, it provides Reinforcement Learning algorithms with a mechanism to probabilistically bias an exploration process by reusing a Policy Library. Our proposed Policy Reuse algorithm, called PRQ-learning, improves the learning performance over exploration strategies that learn from scratch. Second, Policy Reuse provides an incremental method to build the Policy Library. The library is built at the same time that new policies are learned and past policies are reused. And last, our method to build the Policy Library allows the learning of the structure of the domain in terms of a set of δ-core-policies or δ-Basis-Library. Reusing this set of policies ensures that a minimum gain will be obtained when learning a new task, as demonstrated theoretically.

Policy Reuse defines a completely new way to reuse previous knowledge. It should be easy to identify policies with classical macro (or SMDP) actions. However, the way we reuse policies is completely different to the way macro-actions or options are used. Let us take the case of an option. An option is defined as a mapping between states and actions, an applicability condition, and an end condition. The first and second components have a direct mapping to a policy, since a policy is an state-action mapping applicable in any situation of the domain. However, options are defined to be executed until an end condition is satisfied, or until the option is interrupted. Opposite to this scheme, Policy Reuse never executes complete policies, nor even partial ones. Instead, Policy Reuse executes individual actions suggested by past policies probabilistically. Thus, past policies are only a bias.

The worst scenario for transferring knowledge through Policy Reuse is trying to reuse a policy library where none of the stored policies is useful to solve the current task, i.e., when none of the stored policies are similar to the one is being leaned. The evaluation with the PRQ-Learning algorithm demonstrated that when the policy library reused included a similar policy, that produced a higher performance when compared with other exploration strategies, like ϵ-greedy. Interestingly, when the library does not include any similar policy, the algorithm does not perform worse than when learning from scratch.

Another difference of Policy Reuse with macros/options and hierarchical based approaches is that Policy Reuse learns policies in the same level as past policies, while hierarchical methods learn in different abstraction levels. Last, we would like to point out that hierarchical methods typically require the structure of the domain, i.e., the hierarchy of the domain, is known a priory. We have shown that Policy Reuse learns the structure of the domain in terms of a library of core-policies. We believe that such core-policies could be used in the future to support the learning of hierarchies or abstractions of the domain.

In addition, Policy Reuse is very novel since it is able to transfer knowledge, not only from a source task to a target task, but from many tasks to many tasks. We have demonstrated that a Policy Library can be incrementally built. This property is due to the capability of Policy Reuse to decide (i) given a set of policies, which one to reuse, and (ii) given a new policy, whether it is useful to include it in the Policy Library or not, so it can be reused in future tasks. These mechanisms permit to discover when policies are useful for solving a new task, minimizing the effects of negative transfers.

Acknowledgements This research was conducted while the first author was visiting Carnegie Mellon from the Universidad Carlos III de Madrid, supported by a generous grant from the Spanish Ministry of Education and Fulbright. This research was partly sponsored by the Spanish *Ministerio de Ciencia en Innovacin* project number TIN2008-06701-C03-03 and by Comunidad de Madrid-UC3M project number CCG08-UC3M/TIC-4141. This research was partly sponsored also by Rockwell Scientific Co., LLC under subcontract no. B4U528968 and by BBNT Solutions under subcontract no. 950008572. The views and conclusions contained in this document are those of the authors only, and should not be interpreted as representing any other entity.

References

1. L.P. Kaelbling, M.L. Littman, A.W. Moore, International Journal of Artificial Intelligence Research **4**, 237 (1996)
2. R.S. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, 1998)
3. C. Watkins, Learning from delayed rewards. Ph.D. thesis, Cambridge University, Cambridge, England (1989)
4. G. Tesauro, Machine Learning **8**, 257 (1992)
5. P. Stone, R.S. Sutton, G. Kuhlmann, Adaptive Behavior **13**(3) (2005)
6. M.E. Taylor, P. Stone, Y. Liu, in *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)* (2005)
7. R.S. Sutton, D. Precup, S. Singh, Artificial Intelligence **112**, 181 (1999)
8. A. Jonsson, A. Barto, J. Mach. Learn. Res. **7**, 2259 (2006)
9. M.M. Veloso, *Planning and Learning by Analogical Reasoning* (Springer Verlag, 1994). Revised PhD Thesis Manuscript, Carnegie Mellon University, technical report CMU-CS-92-174
10. J. Bruce, M. Veloso, in *Proceedings of IROS-2002* (Switzerland, 2002). An earlier version of this paper appears in the Proceedings of the RoboCup-2002 Symposium
11. M. Taylor, P. Stone, AI Magazine **32**(1) (2012)
12. F. Fernández, M. Veloso, in *ICML'06 Workshop on Structural Knowledge Transfer for Machine Learning* (2006)
13. F.J. García, M. Veloso, F. Fernández, in *Proceedings of the 12th Conference of the Spanish Association for Artificial Intelligence (CAEPIA'07+TTIA)* (2007)
14. F. Fernández, J. García, M. Veloso, Robotics and Autonomous Systems **58(7)**, 866 (2010). DOI 10.1016/j.robot.2010.03.007
15. P. Dasgupta, K. Cheng, B. Banerjee, in *Advanced Agent Technology*, *Lecture Notes in Computer Science*, vol. 7068, ed. by F. Dechesne, H. Hattori, A. Mors, J. Such, D. Weyns, F. Dignum (Springer Berlin Heidelberg, 2012), pp. 330–345
16. M.E. Taylor, H.B. Suay, S. Chernova, in *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2* (International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2011), AAMAS '11, pp. 617–624. URL http://dl.acm.org/citation.cfm?id=2031678.2031705
17. B.N. da Silva, A. Mackworth, in *Proceedings of the Autonomous Agents and Multi agent Systems (AAMAS 2010)* (2010), pp. 317–324

18. S. Thrun, Efficient exploration in reinforcement learning. Tech. Rep. C,I-CS-92-102, Carnegie Mellon University (1992)
19. R. Maclin, J. Shavlik, L. Torrey, T. Walker, E. Wild, in *Proceedings of the Twentieth National Conference on Artificial Intelligence* (2005)
20. W.D. Smart, L.P. Kaelbling, in *Proceedings of the International Conference of Machine Learning* (2000), pp. 903–907
21. B. Price, C. Boutilier, Journal of Artificial Intelligence Research **19**, 569 (2003)
22. J. Carroll, T. Peterson, N. Owens, in *Proceedings of the Internatioanal Joint Conference on Neural Networks* (2001)
23. K. Dixon, R. Malak, P. Khos, Incorporating prior knowledge and previously learned information into reinforcement learning agents. Tech. rep., Carnegie Mellon University, Institute for Complex Engineered Systems (2000). January
24. J. Carroll, T. Peterson, in *Proceedings of the International Conference on Machine Learning and Applications* (2002)
25. M.G. Madden, T. Howley, Artificial Intelligence Review **21**, 375 (2004)
26. M. Taylor, P. Stone, Y. Liu, Journal of Machine Learning Research **8**(1), 2125 (2007)
27. M.E. Taylor, P. Stone, in *Proceedings of the Twenty-first National Conference on Artificial Intelligence (AAAI'06)* (2006)
28. T.J. Walsh, L. Li, M. Littman, in *Proceedings of the ICML'06 Workshop on Structural Knowledge Tranfer for Machine Learning* (2006)
29. V. Soni, S. Singh, in *Proceedings of the National Conference on Artificial Intelligence(AAAI'06)* (2006)
30. W.T.B. Uther, Tree based hierarchical reinforcement learning. Ph.D. thesis, Carnegie Mellon University (2002)
31. L. Torrey, J. Shavlik, T. Walker, R. Maclin, in *Proceedings of 17th Conference on Inductive Logic Programming* (2007)
32. R.S. Sutton, D. Precup, S. Singh, in *Proceedings of the Internacional Conference on Machine Learning (ICML'98)* (1998)
33. M. Stolle, D. Precup, in *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, *Lecture Notes In Computer Science*, vol. 2371 (Springer, 2002)
34. M. Taylor, P. Stone, in *Proceedings of the 24th International Conference on Machine Learning (ICML'07)* (2007)
35. S.P. Singh, Machine Learning **8** (1992)
36. T.G. Dietterich, Journal of Artificial Intelligence Research **13**, 227 (2000)
37. B. Hengst, in *Proceedings of the Nineteenth International Conference on Machine Learning* (2002)
38. S. Thrun, A. Schwartz, in *Advances in Neural Information Processing Systems 7* (MIT Press., 1995)
39. O. Simsek, A.P. Wolfe, A.G. Barto, in *Proceedings of the Twenty-Second International Conference on Machine Learning* (2005)
40. M. Bowling, M. Veloso, in *Proceedings of IJCAI-99* (1999)
41. R. Parr, in *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence (UAI-98)* (Morgan Kaufmann, San Francisco, CA, 1998)
42. M. Taylor, P. Stone, Journal of Machine Learning Research **10**, 1633 (2009)
43. A.A. Sherstov, P. Stone, in *Proceedings of the Twentieth National Conference on Artificial Intelligence* (2005)
44. F. Fernández, M. Veloso, in *Proceedings of the International Conference on Automated Planning and Schedulling (ICAPS'06)* (2006)
45. J.Y. Yu, S. Mannor, in *ICML'09: Proceedings of the 26th Annual International Conference on Machine Learning* (2009)
46. R.B. Ollington, P.W. Vamplew, International Journal of Intelligent Systems **20**, 1037 (2005)
47. F. Fernández, D. Borrajo, International Journal of Intelligent Systems **23**(2), 213 (2008)
48. Y. Chevaleyre, A.M. Pamponet, J.D. Zucker, in *Knowledge Acquisition: Approaches, Algorithms and Applications (PKAW'2008)*, *Lecture Notes in Artificial Intelligence*, vol. 5465 (2009)