Building a Library of Policies through Policy Reuse

Fernando Fernández **Manuela Veloso**

July 2005
CMU-CS-05-174

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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 Fullbright. The second author was partially sponsored by Rockwell Scientific Co., LLC under subcontract no. B4U528968 and prime contract no. W911W6-04-C-0058 with the US Army, and by BBNT Solutions, LLC under contract no. FA8760-04-C-0002 with the US Air Force. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring institutions, the U.S. Government or any other entity.

Keywords: Reinforcement Learning, Policy Reuse, Policy Library, Eigen-policy.

Abstract

Policy Reuse (PR) provides Reinforcement Learning algorithms with a mechanism to bias an exploration process by reusing a set of past policies. Policy Reuse offers the challenge of balancing the exploitation of the ongoing learned policy, the exploration of new random actions, and the exploitation of past policies. Efficient application of Policy Reuse requires a mechanism to build, for each domain, a library of policies which is useful and accurate enough to efficiently solve any task in such domain. In this work, we propose a mechanism to create a library of policies based on a similarity metric among policies. If the new policy is similar to any of the past ones, it is not added to the library. Otherwise, it is stored together with the other policies, so it can be reused in the future. Thus, the Policy Library stores the *basis* or *eigen-policies* of each domain, i.e., the core past policies that are effectively reusable. Empirical results demonstrate that the Policy Library can be efficiently created and that the stored eigen-policies can be understood as a representation of the structure of the domain.

1 Introduction

Policy Reuse (PR) is a learning process in which learned policies are saved and reused for similar tasks in the same domain. The domain defines how the agent behaves in the environment, i.e. the state transition function; each different task in the same domain is characterized through its reward function.

Policy Reuse is built upon two previous contributions: symbolic plan reuse [10] and extended rapidly-exploring random trees (E-RRT) [1]. 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 solving the general reuse problem in a probabilistic manner.

Building upon these two approaches we have recently developed a probabilistic policy reuse algorithm for tasks within the same domain in Reinforcement Learning, that we called PRQ-Learning [4]. It is based on two cornerstones. Firstly, an exploration strategy able to bias the exploration of the domain with a predefined past policy; and second, a similarity metric that allows the estimation of the similarity of past policies with respect to a new one [3]. The PRQ-Learning algorithm uses the similarity metric to estimate the usefulness of reusing each of the past policies, so the most useful one is selected and exploited to learn the new one.

Policy Reuse requires a set of policies to reuse. Thus, a mechanism to create this set is required. In this work, we contribute an incremental method to build a library of policies. When solving a new problem by reuse, the algorithm determines whether the learned policy is or is not "sufficiently" different from the past policies, as a function of the effectiveness of the reuse. The idea is to identify the core policies that need to be saved to solve any new task in the domain within a threshold of similarity. Given a threshold δ defining the success of the reuse, our algorithm identifies a set of "δ-eigen-policies," as the basis or learned structure of the domain. Thus, our method to build the Policy Library has a novel "side-effect" in terms of learning the structure of the domain, i.e., the basis or the "eigen-policies" of the domain.

Policy Reuse and the learning of the structure of a domain are still challenge areas, although several related works can be found in the bibliography. For instance, the integration of previously learned sub-policies or options is applied to improve the learning of new tasks [9, 6]. Hierarchical RL [2] tries to find the relationship among different abstraction levels of action policies. Life-long learning improves new learning processes by using the experience of past ones [7], and some methods to find the structure of the domain can be found [8]. However, this is the first work in which Policy Reuse is applied to learn the structure of a domain.

This report is organized as follows. Section 2 introduces Policy Reuse, the similarity metric among policies, and the PRQ-Learning algorithm, which efficiently reuses a defined set of policies. Section 3 defines the concept of Policy Library, and describes PLPR, an algorithm to build it. Section 4 describes the experiments performed. Lastly, Section 5 summarizes the main conclusions of this work.

2 Policy Reuse

The goal of this section is to summarize Policy Reuse. Firstly, we describe the concepts of task, domain, and gain. Then, we define how the reuse of a past policy is used as a bias in a new exploratory process. We also introduce a similarity concept between policies, which motivation is deeply described in [3]. Lastly, we describe the PRQ-learning algorithm [4].

2.1 Domain, Tasks and MDPs

A Markov Decision Process [5] is represented with a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} >$, where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ is the set of all possible actions, $\mathcal{T}$ is an unknown stochastic state transition function, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \Re$, and $\mathcal{R}$ is an unknown stochastic reward function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \Re$. We focus on RL domains where different *tasks* can be solved. We introduce a task as a specific reward function, but the other concepts, $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{T}$ stay constant for all the tasks. Thus, we extend the concept of an MDP by introducing two new concepts: domain and task. We characterize a domain, $\mathcal{D}$, as a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T} >$. We define a task, Ω, as a tuple $< \mathcal{D}, \mathcal{R}_\Omega >$, where $\mathcal{D}$ is a domain as defined before, and $\mathcal{R}_\Omega$ is the stochastic and unknown reward function.

In this work we assume that we are solving a task with absorbing goal states. Thus, if s_i is a goal state, $\mathcal{T}(s_i, a, s_i) = 1$, $\mathcal{T}(s_i, a, s_j) = 0$ for $s_i \neq s_j$, and $\mathcal{R}(s_i, a) = 0$, for all $a \in \mathcal{A}$. A trial starts by locating the learning agent in a random position in the environment. Each trial finishes when a goal state is reached or when a maximum number of steps, say H, is achieved. Thus, the goal is to maximize the expected average reinforcement per trial, say W, defined as $W = \frac{1}{K} \sum_{k=0}^{K} \sum_{h=0}^{H} \gamma^h r_{k,h}$, 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 trial k, in a total of K trials. An action policy, $\Pi : \mathcal{S} \rightarrow \mathcal{A}$, defines for each state, the action to execute. The action policy Π^* is optimal if it maximizes the gain W in such a task, say W^*_Ω.

The goal of Policy Reuse is to describe how learning can be sped up if different policies, which solve different tasks, are used to bias the exploration process of the learning of the action policy of another similar task. Then, the scope of this work is summarized as: (i) we need to solve the task Ω, i.e. learn Π^*_Ω; (ii) we have previously solved the set of tasks $\{\Omega_1, \ldots, \Omega_n\}$, so we have the set of policies, $\{\Pi^*_1, \ldots, \Pi^*_n\}$, to solve them respectively; (iii) how can we use the previous policies, Π^*_i, to learn the new one, Π^*_Ω?

An efficient solution to this problem is the PRQ-Learning algorithm. This algorithm automatically answers two questions: (i) what policy, from the set $\{\Pi^*_1, \ldots, \Pi^*_n\}$, is used to bias the new learning process? (ii) once a policy Π_i is selected, how is it integrated in the learning process? The algorithm is based on an exploration strategy, π-reuse, which is able to bias the learning of a new policy with only one past policy. From this strategy, a similarity metric between policies is obtained, providing a method to select the most accurate policy to reuse. Both the π-reuse strategy and the similarity metric, defined in [3], are summarized in the next subsection.

2.2 A Similarity Metric Between Policies

The goal of the π-reuse strategy is to balance random exploration, exploitation of the past policy, and exploitation of the new policy, which is being learned currently. The π-reuse strategy follows

the past policy, say Π_{past}, with a probability of ψ. However, with a probability of $1 - \psi$, it exploits the new policy. Obviously, random exploration is always required, so when exploiting the new policy, it follows an ϵ-greedy strategy, as defined in Table 1. Lastly, the υ parameter allows the decay of the value of ψ in each trial.

π-reuse $(\Pi_{old}, K, H, \psi, \upsilon)$.
for $k = 1$ to K
Set the initial state, s, randomly.
Set $\psi_1 \leftarrow \psi$
for $h = 1$ to H
With a probability of ψ_h, $a = \Pi_{old}(s)$
With a probability of $1 - \psi_h$, $a = \epsilon$-greedy$(\Pi_{new}(s))$
Receiv current state s', and reward, $r_{k,h}$
Update $Q^{\Pi_{new}}(s, a)$, and therefore, Π_{new}
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 and Π_{new}

Table 1: π-reuse Exploration Strategy.

Interestingly, the π-reuse strategy also contributes a similarity metric between policies, based on the gain obtained when reusing each policy. Let's call W_i the gain obtained while executing the π-reuse exploration strategy, reusing the past policy Π_i. We call Π^*_Ω the optimal action policy for solving the task Ω. W^*_Ω is the gain obtained when using the optimal policy, Π^*_Ω, to solve Ω. Therefore, W^*_Ω is the maximum gain that can be obtained in Ω. Then, we can use the difference between W^*_Ω and W_i to measure the similarity among both policies using the distance metric shown in equation 1.

$$d_{\rightarrow}(\Pi_i, \Pi) = W^*_\Omega - W_i \tag{1}$$

In this case the distance metric is not symmetric, so $d_{\rightarrow}(\Pi_i, \Pi_j)$ could be different from $d_{\rightarrow}(\Pi_j, \Pi_i)$. This distance metric is also useful to estimate how useful to reuse the policy Π_i is to learn to solve the new task. Then, the most useful policy to reuse, from a set $\{\Pi_1, \dots, \Pi_n\}$, is $\arg_{\Pi_i} \max(W_i), i = 1, \dots, n$. Notice that W^*_Ω has disappeared of the formula, given that is independent of i. Thus, W_i, or the average reward obtained when reusing the policy Π_i with the π-reuse exploration strategy, is used as an estimation of how similar the policy Π_i is to the one we are currently learning. The set of W_i values, for $i = 1, \dots, n$, is unknown a priori, but it can be estimated on-line while the new policy is computed. This idea is formalized in the PRQ-Learning algorithm.

2.3 PRQ-Learning Algorithm

The PRQ-Learning algorithm (Policy Reuse in Q-Learning) [4] is shown in Table 2. The learning algorithm used is Q-Learning [11]. It has been chosen because it is an off-policy algorithm, and therefore, it allows to learn a policy while following a different one. The goal is to solve a task Ω, i.e. to learn an action policy Π_Ω. We have n past policies to solve n different tasks respectively. For simplicity of the notation, we will call these policies $\Pi_1, \ldots, \Pi_n$. Let's call W_i the expected average reward that is received when reusing the policy Π_i with the π-reuse exploration strategy. Also, let's call W_Ω the average reward that is received when following the policy Π_Ω greedily. The algorithm uses the W values in a softmax way to choose between reusing a past policy with the π-reuse exploration strategy, or following the ongoing learned policy greedily.

This algorithm has demonstrated to successfully reuse a predefined set of policies [4]. The problem is that it requires the existence of such a set of policies. This work contributes a method to incrementally construct the Policy Library, so each time a new policy is learned, the method decides whether to add it to the library or not, depending on a threshold of similarity, δ. The algorithm is described in the next section.

3 An Algorithm to Learn a Library of Policies

This section describes the $PLPR$ algorithm (Policy Library through Policy Reuse). 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 priori. Otherwise, a method to learn them in parallel could be applied.

The algorithm works as follows. Let's call PL the Policy Library, and let's define it as a set of policies. 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 similarity metric defined in equation 1, instantiated in equation 2. In the equation, W_2 is the average gain obtained when following Π_2 greedily, and W_1 is the average gain obtained when reusing Π_1. Both values are computed in the execution of the PRQ-Learning algorithm, so no additional computations are required.

$$d_{\rightarrow}(\Pi_1, \Pi_2) = W_2 - W_1 \tag{2}$$

As defined in the previous section, this distance metric estimates how similar Π_1 is to Π_2. 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.

The PLPR algorithm is defined 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.

Equation 3 is the update equation for the policy library, derived from equation 2. It requires the computation of the most similar policy, which is the policy Π_j such as $j = \arg_i \max W_i$, for

- Given:
  1. A set of n policies, $\{\Pi_1^*, \dots, \Pi_n^*\}$ to solve different tasks
  2. A new task Ω we want to solve
  3. A maximum number of trials to execute, K
  4. A maximum number of steps per trial, H
- 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 trials where policy Π_Ω has been chosen, $U_\Omega = 0$
  5. Initialize the number of trials where policy Π_i has been chosen, $U_i = 0, \forall i = 1, \dots, n$
- For $k = 1$ to K do
  - Choose an action policy, Π_j, randomly, assigning to each policy the probability of being selected computed by the following equation:
  $$P(\Pi_j) = \frac{e^{\tau W_j}}{\sum_{p=0}^{n} e^{\tau W_p}}$$
  - Initialize the state s to a random state
  - Set $R = 0$
  - for $h = 1$ to H do
    * Use Π_j to compute the next action to execute, a, following an exploitation strategy.
    * Execute a
    * Receive current state, s'
    * Receive current reward, r
    * Update $Q_\Omega(s,a)$ using Q-Learning update function:
    $$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a')]$$
    * Set $R = R + \gamma^h r$
    * Set $s \leftarrow s'$
  - Set $W_j = \frac{W_j U_j + R}{U_j + 1}$
  - Set $U_j = U_j + 1$
  - Set $\tau = \tau + \Delta\tau$

Table 2: PRQ-Learning

$i = 1, \dots, n$. The gain 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 a similarity threshold.

The PLPR algorithm has an interesting "side-effect" in terms of learning the structure of the domain. Notice that the Policy Library is initialized to 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 number of policies stored is fully representative of the domain, no more policies are stored. Thus, the stored ones can be considered as the basis or *eigen-policies* of the domain, so any

PLPR Algorithm

- Given:
  1. A Policy Library, LP, 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 LP 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:

$$PL = \begin{cases} PL \cup \{\Pi_\Omega\} & \text{if } W_{max} < \delta W_\Omega \\ PL & otherwise \end{cases} \quad (3)$$

Table 3: PLPR Algorithm

task can be efficiently learned by reusing such a library of tasks. 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.

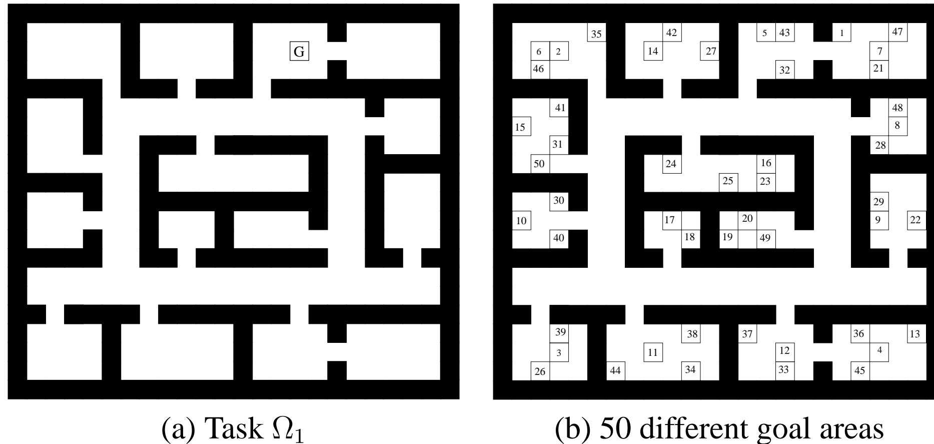
4 Experiments

This section describes the experiments performed in a navigation domain, which is described next.

4.1 Navigation Domain

This domain consists of a robot moving inside of an office area, as shown in Figure 1(a), similar to the one used in other RL works [8]. 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 this case). The possible actions that the robot can execute are "North", "East", "South" and "West", all of size one. The final position after each action is noised by a random variable following a uniform distribution in the range $(-0.20, 0.20)$. The robot knows its location in the space through continuous coordinates (x, y) provided by some localization system. In this work, we assume that we have the optimal

uniform discretization of the state space (which consists of 24×21 regions). Furthermore, the robot has an obstacle avoidance system that blocks the execution of actions that would crash it into a wall. The goal in this domain is to reach the area marked with 'G'. When the robot reaches it, it is considered a successful trial, and it receives a reward of 1. Otherwise, it receives a reward of 0.



(a) Task Ω_1 (b) 50 different goal areas

Figure 1: Office Domain.

Performing a task consists of trying to solve it $K = 2000$ times. Each of these times is called a trial. Each trial 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 the following experiments, 50 different tasks are sequentially performed, each of them with a different reward function, located in different positions of the different rooms of the domain, as shown in Figure 1(b). 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.

4.2 Results

In the experiments, the following parameter setting is used. 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. All the previous parameters have empirically demonstrated that provide good results in this domain [3, 4].

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 eigen-policies stored in the Policy Library, shown in Figure 2. 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 in Section 3, when $\delta = 0$, only 1 policy is stored. When $\delta = 0.25$, the number of eigen-policies is around 14. Interestingly, this is

the number of rooms in the domain. While increasing δ, the number of eigen-policies increases and when $\delta = 1$, almost all the learned policies are stored.

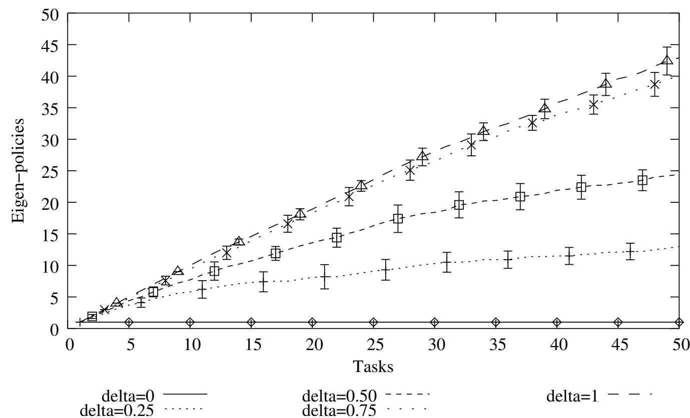


Figure 2: Number of eigen-policies obtained.

Figure 3 shows an example of the eigen-policies obtained in one execution, with $\delta = 0.25$. It represents the Policy Library obtained after performing the 50 tasks which, in this case, is composed of 14 eigen-policies. In the figure, we assume that a policy is represented by the goal area of the task that it solves. An eigen-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 eigen-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 eigen-policies.

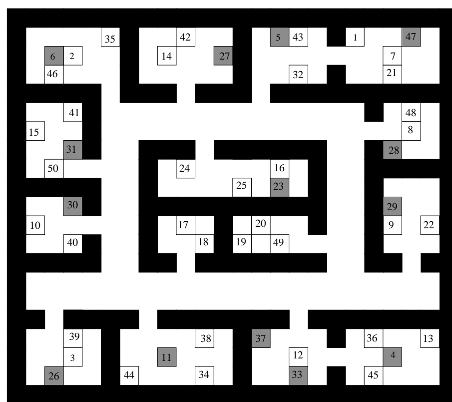


Figure 3: Eigen Policies.

These results demonstrate empirically the influence of the δ parameter in the size of the library, and enforce the idea of defining the δ-eigen-policies as the policies stored in the Policy Library, when learning with the PLPR algorithm with a defined value of δ. Lastly, Figure 4 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 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 the Boltzmann exploration strategy, with different settings of the temperature parameter. The maximum average gain obtained by them is around 0.12, demonstrating that Policy Reuse obtains 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.

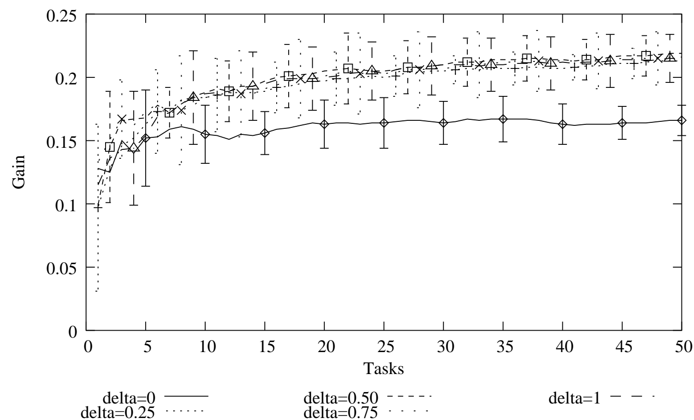


Figure 4: Average gain obtained in the life long term.

5 Conclusions

The goal of this work is to extend Reinforcement Learning to domains where policies to solve different tasks, must be learned. In this report we describe a method, the PLPR algorithm, to build a library of policies based on the concepts of Policy Reuse and similarity between polices. The work contributes three main results. Firstly, the PLPR algorithm allows the construction of the Policy Library. Second, reusing the policies stored in the Policy Library for learning a new policy provides a better performance than when learning the new policy from scratch. And last, the Policy Library is composed of a set of eigen-policies, which has demonstrated to represent the structure of the domain. Future work is oriented to the use of the knowledge learned about the structure of the domain, and how it can be transferred to new learning processes.

References

[1] James Bruce and Manuela Veloso. Real-time randomized path planning for robot navigation. In *Proceedings of IROS-2002*, Switzerland, October 2002. An earlier version of this paper appears in the Proceedings of the RoboCup-2002 Symposium.

[2] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[3] Fernando Fernández and Manuela Veloso. Exploration and policy reuse. Technical Report CMU-CS-05-172, School of Computer Science, Carnegie Mellon University, 2005.

[4] Fernando Fernández and Manuela Veloso. Learning by probabilistic reuse of past policies. Technical Report CMU-CS-05-173, School of Computer Science, Carnegie Mellon University, 2005.

[5] M. L. Puterman. *Markov Decision Processes - Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY., 1994.

[6] Richard S. Sutton, Doina Precup, and Satinder Singh. Intra-option learning about temporally abstract actions. In *Proceedings of the Internacional Conference on Machine Learning (ICML'98)*, 1998.

[7] Sebastian Thrun and Tom Mitchell. Lifelong robot learning. *Robotics and Autonomous Systems*, 15:25–46, 1995.

[8] Sebastian Thrun and A. Schwartz. Finding structure in reinforcement learning. In *Advances in Neural Information Processing Systems 7*. MIT Press., 1995.

[9] William T. B. Uther. *Tree Based Hierarchical Reinforcement Learning*. PhD thesis, Carnegie Mellon University, August 2002.

[10] Manuela M. Veloso. *Planning and Learning by Analogical Reasoning*. Springer Verlag, December 1994. Revised PhD Thesis Manuscript, Carnegie Mellon University, technical report CMU-CS-92-174.

[11] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.