Accelerating autonomous learning by using heuristic selection of actions

R. A. C. Bianchi (rbianchi@fei.edu.br)

Centro Universitário da FEI

C. H. C. Ribeiro (carlos@comp.ita.br)

Instituto Tecnológico de Aeronáutica

A. H. R. Costa (anna.reali@poli.usp.br)

Escola Politécnica da Universidade de São Paulo

Abstract. This paper investigates how to make improved action selection for online policy learning in robotic scenarios using reinforcement learning (RL) algorithms. Since finding control policies using any RL algorithm can be very time consuming, we propose to combine RL algorithms with heuristic functions for selecting promising actions during the learning process. With this aim, we investigate the use of heuristics for increasing the rate of convergence of RL algorithms and contribute with a new learning algorithm, Heuristically Accelerated Q-learning (HAQL), which incorporates heuristics for action selection to the Q-Learning algorithm. Experimental results on robot navigation show that the use of even very simple heuristic functions results in significant performance enhancement of the learning rate.

Keywords: Reinforcement learning, heuristic function, robot navigation, action selection.

1. Introduction

Reinforcement learning (RL) algorithms are very attractive for solving a wide variety of control and planning problems when neither analytical model nor a sampling model is available a priori, since many of them are known to guarantee convergence to equilibrium in the limit (Szepesvari and Littman, 1996) and to provide model-free learning of adequate sub-optimal control strategies.

In RL, learning is carried out online, through trial-and-error interactions of the agent with the environment. Unfortunately, convergence of any RL algorithm may only be achieved after extensive exploration of the state-action space, which can be very time consuming.

However, the rate of convergence of an RL algorithm can be increased by using heuristic functions for selecting actions in order to guide the exploration of the state-action space in a useful way. This paper investigates how to make improved action selections based on heuristics in on-line policy learning for robotic scenarios. We present a new algorithm, HAQ-Learning, which incorporates heuristics for action selection to the well-known Q-Learning algorithm.

A series of empirical evaluation of the algorithm in a commercial simulator for the robot navigation domain were carried out. We show

that even using very simple heuristic functions, the performance of the learning algorithm can be improved.

The paper is organized as follows: Section 2 briefly reviews the reinforcement learning approach, describes the Q-Learning algorithm, and presents some approaches to speeding up RL. Section 3 presents a general formulation and relevant issues for heuristically-accelerated RL algorithms. Section 4 describes some approaches for the design of the Heuristics functions. Section 5 shows how the learning rate can be improved by using heuristics to select actions to be performed during the learning process in a modified formulation of the Q-Learning algorithm. Section 6 describes the robotic navigation domain used in the experiments, presents the experiments performed, and shows the results obtained. Finally, Section 7 provides the conclusions and indicates avenues through which the research proposed in this paper can be extended.

2. Reinforcement Learning

In this section we first review some basic principles of Markov Decision Process (MDP) and then present the basic formulation of the Q-learning algorithm, a well-known RL technique for solving MDPs.

2.1. Markov Decision Processes

Let us consider a single agent interacting with its environment via perception and action. On each interaction step t, the agent senses the current state $\mathbf{s}_t$ of the environment, and chooses an action a_t to perform. The action a_t alters the state $\mathbf{s}_t$ into a new state $\mathbf{s}_{t+1}$, and a scalar reinforcement signal r_t (a reward or penalty) is provided to the agent to indicate the desirability of the resulting state.

Formally, a Markov Decision Process problem is defined as follows. Given:

- A finite set of possible actions $a \in \mathcal{A}$ and process states $\mathbf{s} \in \mathcal{S}$,
- A stationary discrete-time stochastic process, modeled by transition probabilities $P(\mathbf{s}_{t+1}|\mathbf{s}_t, a_t)$, and
- A finite set of bounded reinforcements (payoffs) R, with $r(\mathbf{s}, a) \in \Re$,

the agent must try to find out a stationary policy of actions $a_t^* = \pi^*(\mathbf{s}_t)$ which maximizes the *expected discounted value function*:

$$V^{\pi}(\mathbf{s}_0) = \lim_{M \to \infty} \mathrm{E}[\sum_{t=0}^{M} \gamma^t r(\mathbf{s}_t, \pi(\mathbf{s}_t))] \tag{1}$$

for every state $\mathbf{s}_0$. The superscript π indicates the dependency on the followed action policy, via the transition probabilities $P(\mathbf{s}_{t+1}|\mathbf{s}_t, a_t =$

$\pi(\mathbf{s}_t)$). The *discount factor* $0 \leq \gamma < 1$ forces recent reinforcements to be more important than remote ones. It can be shown that the *optimal cost function*

$$V^*(\mathbf{s}_0) = \lim_{M \to \infty} \mathrm{E}[\sum_{t=0}^{M} \gamma^t r(\mathbf{s}_t, \pi^*(\mathbf{s}_t))] \tag{2}$$

is unique, although there can be more than a single optimal policy π^* (Puterman, 1994). It must be stressed that formulations other than based on the discounted cost function are also possible (Bertsekas, 1995).

2.2. The Q-Learning Algorithm

The goal of the agent in the most common formulation of the RL problem is to learn an optimal policy of actions π^* that maximizes the expected discounted value function for any starting state, when the reinforcements R and the transition probabilities P are not known. As the agent does not have a model for R and P, the action policy must be learned through trial-and-error interactions of the agent with the environment, *i.e.*, the RL learner must explicitly explore its environment.

In an MDP there is at least one optimal policy π^* that is stationary and deterministic (Bertsekas, 1995). One strategy to learn an optimal policy π^* when P and R are not known in advance is to allow the agent

to learn the action value measurement $Q(\mathbf{s}_t, a_t)$, defined as

$$Q(\mathbf{s}_t, a_t) = r(\mathbf{s}_t, a_t) + \gamma \sum_{\mathbf{s}_{t+1} \in \mathcal{S}} P(\mathbf{s}_{t+1}|\mathbf{s}_t, a_t) V^*(\mathbf{s}_{t+1}) \tag{3}$$

which represents the expected discounted cost for taking action a_t when visiting state $\mathbf{s}_t$ and following an optimal policy thereafter.

The Q-learning algorithm (Watkins, 1989) iteratively approximates the function Q, provided the system can be modeled as an MDP, the reinforcement function is bounded, and actions are chosen so that every state-action pair is visited an infinite number of times. The Q learning rule is:

$$\begin{aligned} &Q(\mathbf{s}, a) \leftarrow Q(\mathbf{s}, a) + \\ &\alpha[r(\mathbf{s}, a) + \gamma \max_{a'} Q(\mathbf{s}', a') - Q(\mathbf{s}, a)] \end{aligned} \tag{4}$$

where $\mathbf{s}$ is the current state, a is the action performed in s, $r(\mathbf{s}, a)$ is the reinforcement received after performing a in $\mathbf{s}$, $\mathbf{s}'$ is the new state, γ is a discount factor ($0 \leq \gamma < 1$), and α is the learning rate ($\alpha > 0$).

One of the problems with the Q-learning algorithm is that, as the agent iteratively estimates Q, at early stages learning is basically random exploration. Also, update of the Q value is made one state-action pair at a time, for each iteration with the environment. The larger the

environment, the longer trial-and-error exploration takes to approximate the function Q. To alleviate these problems, several techniques, described in the next section, were proposed.

2.3. Approaches to speed up Reinforcement Learning

The SARSA Algorithm (Sutton, 1996) is a modification of Q-learning that admits the next action to be chosen randomly according to a predefined probability, separating the choice of the actions to be taken from the update of the Q values.

SARSA learning rule does not includes the maximization that exists in Q-learning learning rule (see equation 4), and is defined as:

$$Q(\mathbf{s}, a) \leftarrow Q(\mathbf{s}, a) + \alpha[r(\mathbf{s}, a) + \gamma Q(\mathbf{s}', a') - Q(\mathbf{s}, a)]. \quad (5)$$

If a' is chosen according to a greedy policy, SARSA becomes equivalent to Q-learning and $a' = \max_{a'} \hat{Q}(s', a')$. But if a' is selected randomly according to a predefined probability distribution, the SARSA algorithm can achieve better performance than Q-learning, particularly in cases where the set of actions is large (Sutton, 1996).

It is straightforward to design an algorithm to learn the policy of actions in execution time: at each iteration the algorithm estimates $\hat{Q}^{\pi}$ from π, and, at the same time, changes the probability distribution that is used to choose the next action towards greediness. In this way, SARSA estimates the policy of actions at the same time used to interact with the environment. Like Q-learning, this method has been proved to converge to optimal action values, provided that actions asymptotically approach a greedy policy (Szepesvari, 1997).

Two different methods have been proposed that combine eligibility traces with Q-learning and SARSA: the $Q(\lambda)$ (Watkins, 1989) and the $SARSA(\lambda)$ (Rummery and Niranjan, 1994). Eligibility traces, proposed initially in the $TD(\lambda)$ algorithm (Sutton, 1988), are used to speed up the learning process by tracking visited states and adding a portion of the reward received to each state that has been visited in an episode.

The eligibility of a state defines how often it was visited in the recent past. It can be used to increase the influence of states closer to the reward, and can be defined as:

$$e(u) = \sum_{k=1}^{t} (\lambda\gamma)^{(t-k)} \delta_{s,s_k}, \tag{6}$$

where: u is the state that is being updated, t is the update time, λ is a discount factor for the temporal differences ($0 \leq \lambda \leq 1$), γ is a discount

factor for future rewards ($0 \leq \gamma < 1$) and δ_{s,s_k} equals 1 if $s = s_k$ and 0 otherwise (Kronecker Delta).

$Q(\lambda)$ and $SARSA(\lambda)$ extends the concept of eligibility to include the state-action pair, by using:

$$e(u,v) = \begin{cases} \gamma\lambda e(u,v) + 1 & \text{if } u = s_t \text{ e } v = a_t, \\ \gamma\lambda e(u,v) & \text{otherwise.} \end{cases} \tag{7}$$

Instead of updating a state-action pair at each iteration, all pairs with eligibilities different from zero are updated, allowing the rewards to be carried over several state-action pairs.

The Dyna architecture was proposed by Sutton (1990) as a way to find an optimal policy by learning the environment model. It is characterized by the iterative learning of a direct model of the transition probabilities P and of rewards R, simultaneously with the application of some method to compute the action values and action policy.

The Dyna architecture allows for the execution of real and hypothetical experiences: real experiences receive the reinforcement from the environment and are used to update the world model; hypothetical experiences are executed using the learned model, and can be used to explore actions that the real agent has never tried before or to probe state-action pairs. In this way, the Dyna Architecture allows

the exploration to be done both in real and hypothetical experiences. For the latter, this exploration can be done in a way to encourage other exploration policies.

Sutton (1990) proposed two different algorithms for the Dyna architecture: Dyna-PI, that uses the well known Policy Iteration method (Bertsekas, 1987) to compute the action values and action policy; and Dyna-Q, that determines the policy in a way similar to the one implemented in the Q-learning algorithm. In both algorithms, at each iteration one real experiment and k hypothetical experiments can be executed. In this way, Dyna-Q requires k more times to execute than Q-learning, but in practice requires an order of magnitude less steps to find the optimal policy (Kaelbling et al., 1996).

A natural improvement that can be made to the methodology implemented in the Dyna architecture was proposed independently in two similar techniques, called Queue-Dyna (Peng and Williams, 1993) and Prioritized Sweeping (Moore and Atkeson, 1993). In these techniques, instead of randomly choosing the states for updates from the hypothetical experiments, a dynamic priority mechanism is used. It indicates the importance of updating a state: the priorities of the states that do not need update are diminished, and higher priorities are set to states in need of updating. The main difference between Prioritized Sweeping

and the Queue-Dyna is that the former updates only the value function V, while the latter considers the action value function Q.

Butz (2002) proposed the combination of an online model learner with a state value learner in a MDP. The model learner learns a predictive model that approximates the state transition function of the MDP in a compact, generalized form. State values are evaluated by means of the evolving predictive model representation. In combination, actual action choice depends on state values predicted by the predictive model yielding anticipatory behavior. It is shown that this combination can be applied to further speed up learning of an optimal policy.

In most RL algorithms, even if two problems are very similar at some abstract level, solving one will not help to solve the other and an extensive re-learning effort may be required. Drummond (2002) proposed a system that accelerates RL by transferring parts of previously learned solutions to a new problem, exploiting the results of prior learning to speed up the process. The system identifies subtasks on the basis of stable features that arise in the multi-dimensional value function due to the interaction of the learning agent with the world during the learning process. A partitioning of the state space and of the value function is made, defining individual subtasks. The relationship among subtasks is represented in a composite graph, which represents the whole task.

As the agent learns, it can access a case base that contains clipped parts of previous learned value functions that solves individual problems. The graphs are used to index solution functions stored in that case base. Knowledge transfer is achieved by transforming and composing functions from the base to form an approximation to the solution of the new task. This approximation is then used to reinitialize the value function of the new learning process. Drummond argues that "it is not necessary for the transfer to produce an exact solution for the new task, and that it is sufficient that the solution is close enough to the final solution to produce an average speed up" (Drummond, 2002, page 60).

The idea of identifying features in the value function and thus producing learning acceleration, as proposed by Drummond (2002), as well as the idea of learning the environment model to speed up the action policy computation are ideas that inspired the present work. We propose to integrate all information available in a heuristic function that guides the state-space exploration.

2.4. Combining Heuristics and Reinforcement Learning

An interesting property of the RL algorithms described in previous sections is that, although the exploration-exploitation tradeoff must be addressed, the Q values will converge to the optimal value Q^*,

independently on the exploration strategy employed, provided all state-action pairs are visited often enough (Watkins, 1989). Our idea is to use this facility to propose a general formulation for a variation of the RL algorithms that considers the use of heuristic-guided exploratory action policies. Conveniently chosen heuristics can be used for selecting appropriate actions to perform in order to guide exploration of the state-action space during the learning process. This way, the learning process can be conducted in the direction of useful regions of the state-action space, improving the learner behavior. In many cases, the total learning time for finding an optimal policy may not be reduced, but learning is quickly conducted towards regions of the state space which allow for reasonably good action policies, already at initial stages of the learning process.

The $Q(\lambda)$ and the $SARSA(\lambda)$ algorithms extend the original Q-learning and SARSA algorithms by, instead of updating a state-action pair at each iteration, updating all pairs in the eligibility trace, allowing the reward to be spread over several pairs. By doing this, both algorithms modify the way in which the update of the value function estimate is done, without explicitly modifying the exploration policy.

The Dyna architecture main characteristic is to allow the execution of hypothetical experiences, integrating execution-time-planning and learning by alternately interacting with the real environment and on

its learned model. In this way, this architecture aim at encouraging and increasing exploration.

The use of heuristics to guide exploration is the fact that most distinguishes the algorithm proposed in next section from algorithms based on the Dyna architecture – including algorithms such as Dyna-Queue and Prioritized Sweeping – as well as from every other RL algorithm. The use of an explicit heuristic function to guide exploration of the state-action space reduces learning time, and, as the action value update rule is not modified, learning will further refine the defined heuristics and quickly remove any error.

Finally, using heuristics in a learning algorithm is an idea pursued in other contexts, such as in Ant Colony Optimization (Bonabeau, Dorigo and Theraulaz, 2000). However, the full possibilities and specially a convenient formalization for their use have not been fully explored in the Machine Learning literature. This motivates the definition of a general class of algorithms that combines heuristics and RL techniques.

3. Heuristically Accelerated Learning - HAL

Formally, a Heuristically Accelerated Learning (HAL) algorithm is a way to solve a MDP problem with explicit use of a heuristic function

$\mathcal{H} : \mathcal{S} \times \mathcal{A} \rightarrow \Re$ for influencing the choice of actions by the learning agent. $H_t(\mathbf{s}_t, a_t)$ defines the heuristic that indicates the importance of performing (at time t) action a_t when visiting state $\mathbf{s}_t$.

An important characteristic of a HAL algorithm is that the heuristic function can be modified or adapted online, as learning progresses and new information for enhancement of the heuristic becomes available. In particular, either prior domain information or initial learning stage information can be used to define heuristics to accelerate learning.

Additionally, we must point out that there are many methods, which can be used to extract a convenient heuristic. Considering the great number of domains suitable for RL algorithms and the vast number of knowledge extraction methods, it is not difficult to informally validate this assumption.

A generic procedure for HAL can be defined as a four-step process, sequentially repeated until a stopping criteria is met, according to Table I.

3.1. The Heuristic Function $\mathcal{H}$

The heuristic function is an action policy modifier which does not interfere with the standard bootstrap-like update mechanism of RL algorithms. A proposed strategy for action choice is an $\epsilon-Greedy$ mech-

Table I. The "Heuristic Accelerated Learning" algorithm

Produce an arbitrary estimation for the value function.
Define an initial heuristic function $H_t(\mathbf{s}, a)$ using an appropriate method.
Observe the current state $\mathbf{s}$.
Repeat:
    Select an action a by adequately combining the heuristic function and the value function.
    Execute action a.
    Receive reinforcement $r(\mathbf{s}, a)$ and observe next state s'.
    Update $H_t(\mathbf{s}, a)$ using an appropriate method.
    Update value function.
    Update state $\mathbf{s} \leftarrow \mathbf{s}'$.
until a stopping criteria is met,
where $\mathbf{s} = \mathbf{s}_t$, $\mathbf{s}' = \mathbf{s}_{t+1}$ and $a = a_t$.

anism where a heuristic mechanism formalized as a function $H_t(\mathbf{s}_t, a_t)$ is considered, thus:

$$\pi(\mathbf{s}_t) = \begin{cases} \arg\max_{a_t} \left[\mathsf{F}_t(\mathbf{s}_t, a_t) \bowtie \xi H_t(\mathbf{s}_t, a_t)^\beta \right] & \text{if } q \leq p, \\ a_{random} & \text{otherwise} \end{cases} \quad (8)$$

where:

– $\mathcal{F} : \mathcal{S} \times \mathcal{A} \rightarrow \Re$ is an estimate of a value function that defines the expected cumulative reward. If $\mathsf{F}_t(\mathbf{s}_t, a_t) \equiv \hat{Q}_t(\mathbf{s}_t, a_t)$ we have an algorithm similar to standard *Q–Learning*.

– $\mathcal{H} : \mathcal{S} \times \mathcal{A} \rightarrow \Re$ is the heuristic function that plays a role in the action choice. $H_t(\mathbf{s}_t, a_t)$ defines the importance of executing

action a_t in state $\mathbf{s}_t$. Note that the t index means that the heuristic function can change through time.

– $\bowtie$ is a function that operates on real numbers and produces a value from an ordered set which supports a maximization operation.

– ξ and β are design parameters that control the influence of the heuristic function.

– q is a random value uniformly distributed over $[0, 1]$ and p ($0 \leq p \leq 1$) is a parameter that defines the exploration/exploitation tradeoff: the larger p, the smaller is the probability of executing a random exploratory action.

– a_{random} is an action randomly chosen among those available in state $\mathbf{s}_t$.

In general, the value of $H_t(\mathbf{s}_t, a_t)$ must be larger than the variation among the values of $\mathsf{F}(\mathbf{s}_t, a)$ for a given $\mathbf{s}_t \in S$, so that it can influence the action choice. On the other hand, it must be as small as possible in order to minimize the error. If $\bowtie$ is a sum and $\xi = \beta = 1$, a heuristic can be defined as:

$$H_t(\mathbf{s}_t, a_t) = \begin{cases} \max_a \left[\mathsf{F}_t(\mathbf{s}_t, a)\right] - \mathsf{F}_t(\mathbf{s}_t, a_t) + \eta & \text{if } a_t = \pi^H(\mathbf{s}_t), \\ 0 & \text{otherwise.} \end{cases} \quad (9)$$

where η is a small value and $\pi^H(\mathbf{s}_t)$ is a heuristic obtained using an appropriate method.

For instance, let [1.0 1.1 1.2 0.9] be the values of $\mathsf{F}(\mathbf{s}_t, a)$ for four possible actions $[a_1\ a_2\ a_3\ a_4]$ for a given state $\mathbf{s}_t$. If the desired action is the first one (a_1), we can use $\eta = 0.01$, resulting in $H(\mathbf{s}_t, a_1) = 0.21$ and zero for the other actions (see Figure 1). The heuristic can be defined similarly for other definitions of $\bowtie$ and values of ξ and β.

The function $\bowtie$ is the last remaining item to be discussed, according to the formulation presented in Equation 8. Any function with a real-valued domain producing values from an ordered set can be used. The use of simple addition is particularly convenient, as it allows for an analysis of the influence of $\mathcal{H}$ in a manner similar to the analysis of heuristics in informed search algorithms, such as A^* (Hart et al., 1968; Russell and Norvig, 2002).

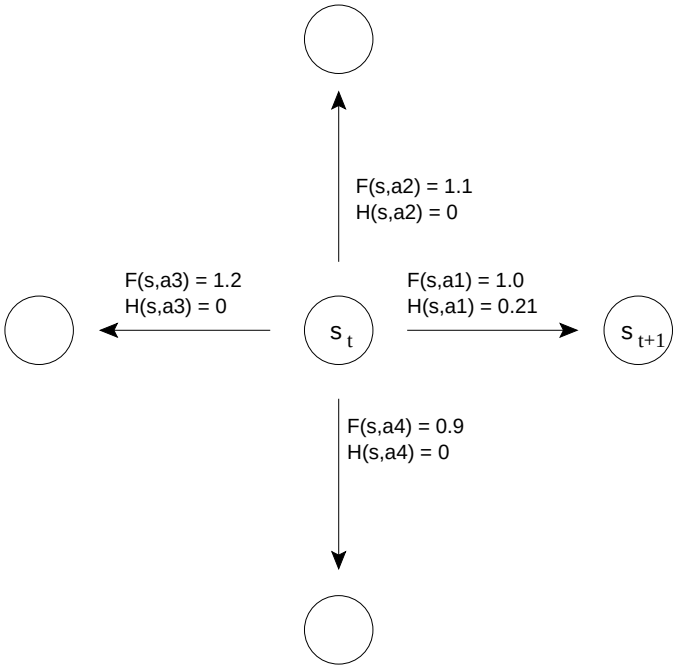


Figure 1. Suppose a state s_t and a desired state s_{t+1}. The value of $H(s_t, a_1)$ for the action that leads to s_{t+1} is 0.21, and zero for the other actions.

4. Definining the Heuristic Function $\mathcal{H}$

A fundamental point to be addressed is how to discover, at an initial learning stage, which action policy must be used in order to accelerate learning. From the definition of a HAL algorithm and from the analysis of heuristic functions of the last section, this issue translates into how to define the heuristic function in an initial situation.

Naturally, there are many methods that can be used to define a heuristic function. We divide these in two main classes:

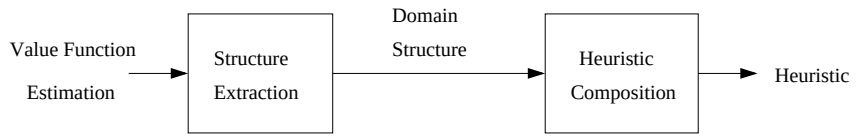


Figure 2. General scheme for methods to infer the Heuristic Function.

– The class of methods that use prior domain knowledge to infer a heuristic. These methods can either establish an ad-hoc heuristic for the problem to be addressed, or reuse action policies previously learned in similar tasks to define heuristics in a case-based approach.

– The class of methods that use information from the learning process itself to infer a heuristic in execution time. Information used to infer the heuristic can originate from, for instance, the current action policy, the value function, the state space trajectory, etc.

Excluding heuristics that are defined in an *ad hoc* manner, methods to infer a heuristic function work in two stages. The first stage extracts domain structure information from the value function F, and the second stage finds the heuristic for the action policy – either in execution time or from a database of cases – using the information extracted from F. We call these stages respectively *Structure Extraction* and *Heuristic Composition* (see Figure 2).

As an example of a case-based method for heuristic extraction is a policy reuse approach, where information extracted from the value

function is matched against a case base for defining a policy that can be used as an heuristic for a new problem (Drummond, 2002). In general, case-based methods can face two problems: first, how to extract relevant features of a case for indexing the case base, and second, how to adapt a previous case for a current matched situation.

4.1. Structure Extraction

A simple technique for extracting information about the problem structure during the learning algorithm execution time is to use the information derived from exploration. This technique, called "Structure from Exploration", is exemplified in Subsection 6.1 for a mobile robot domain.

This technique derives a crude estimate of the transition probabilities $\hat{P}_t(\mathbf{s}_t, a_t, s_{t+1})$, by annotating the results of every action performed by the agent. In the case of a mobile robot, for every move a record of success or failure (in the case of an obstacle blocking the move) is done. Along time, this will generate a representation of the environment. Other approaches for extracting structure are describe elsewhere (Bianchi, 2004).

4.2. An Approach for Heuristics Composition: Heuristic Backpropagation

In the same way that a variety of methods can be used to extract structure, different approaches can be used in the heuristic composition stage. We present here a method for on-line automatic composition of a heuristic from the extracted domain structure: the *Heuristic Backpropagation* method. The basic idea is to backpropagate, from a set of final states, the correct policies that lead to those states. For instance, in a single goal problem, once a terminal state is reached, the heuristic is defined as the set of actions that, from the immediately preceding states, leads to the terminal state. This heuristic is propagated recursively to the predecessors of the states that the heuristic have already defined, until the heuristics for all states are defined.

In a problem with many goal states the same algorithm can be used, but instead of starting propagation from a single state, it considers all the states from the set of final states. If the complete set of final states is not known a priori, the algorithm generates the heuristic using all known final states and, in the event of the discover of new final state, the heuristic is recomputed.

Heuristic Backpropagation is an application of the basic Dynamic Programming algorithm (Bertsekas, 1987, p. 12), which proceeds back-

wards from the final states to the initial states, defining the policy and the cost of the transitions. In domains where all the transition probabilities and states are known, both work in the same way. If only part of the environment structure is known, Heuristic Backpropagation is performed only for states that have been visited. In the case of autonomous robotic mapping, an environment model (i.e., a map) is gradually constructed. In this case, Heuristic Backpropagation can then be performed in parts of the environment that were already mapped.

Combining any structure extraction method with Heuristic Backpropagation generates an algorithm to define the heuristic function. In Section 6, we consider this to define a new algorithm, named "Heuristics from Exploration". It extracts information about the structure of the environment by means of the "Structure from Exploration" method, and then defines the heuristic using Heuristic Backpropagation.

5. Heuristically Accelerated *Q–Learning*: HAQL

HAQL is a heuristic-based extension of the *Q–Learning* algorithm (Section 2.2). In order to implement HAQL, it is necessary to define an action policy rule based on Equation 8 and a method for heuristic updating. We first consider a modification of the standard $\epsilon - Greedy$

Q–Learning action policy, incorporating the heuristic function as an addition (with $\beta = \xi = 1$) to the action value function. Thus:

$$\pi(\mathbf{s}_t) = \begin{cases} \arg\max_{a_t} \left[\hat{Q}(\mathbf{s}_t, a_t) + H_t(\mathbf{s}_t, a_t)\right] & \text{if } q \leq p, \\ a_{random} & \text{otherwise.} \end{cases} \tag{10}$$

The value of the heuristic is defined by instantiating Equation 9:

$$H(\mathbf{s}_t, a_t) = \begin{cases} \max_a \hat{Q}(\mathbf{s}_t, a) - \hat{Q}(\mathbf{s}_t, a_t) + \eta & \text{if } a_t = \pi^H(\mathbf{s}_t), \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

Convergence of this algorithm is presented in (Bianchi, 2004), together with the definition of an upper bound for the error.

The complete HAQL algorithm is presented in Table II. Note that the only modifications made to the original Q-learning algorithm are a heuristic function for the action choice and an update step for the function $H_t(\mathbf{s}_t, a_t)$.

The function $H_t(\mathbf{s}_t, a_t)$ can be extracted by using any method. Naturally, a careful choice can accelerate computation and increase the generality of the algorithm. The experiments described below show how HAQL can be used together with "Heuristic from Exploration" method described in Section 4 in Mobile Robot domains.

Table II. The HAQL Algorithm.

Initialize $Q(s, a)$.
Define an initial heuristic function $H_t(\mathbf{s}, a)$ using an appropriate method.
Observe the current state $\mathbf{s}$.
Repeat:
    Select action a using equation (10).
    Execute action a.
    Receive reinforcement $r(\mathbf{s}, a)$ and observe next state s'.
    Update the values of $H_t(\mathbf{s}, a)$ using an appropriate method.
    Update the values of $Q(\mathbf{s}, a)$ according to the update rule:
    $Q(\mathbf{s}, a) \leftarrow Q(\mathbf{s}, a) + \alpha[r(\mathbf{s}, a) + \gamma \max_{a'} Q(\mathbf{s}', a') - Q(\mathbf{s}, a)]$.
    Update state $\mathbf{s} \leftarrow \mathbf{s}'$.
Until a stopping criteria is met,
where: $\mathbf{s} = \mathbf{s}_t$, $\mathbf{s}' = \mathbf{s}_{t+1}$, $a = a_t$ and $a' = a_{t+1}$.

6. Experimental Results in Mobile Robot Domains

A basic task of an autonomous mobile robot is to navigate in an environment, aiming to reach goals in specified positions, while deviating from obstacles. This section present results for three different experiments: a study on the "Heuristic from Exploration" method (Subsection 6.1), the use of the HAQL for simulated navigation in the grid world domain (Subsection 6.2) and the use of HAQL for simulated navigation of a real mobile robot (Subsection 6.3).

As reinforcement learning requires a large number of episodes for convergence, we initially analyzed HAQL in simulated domains. In all the experiments (with the exception of the one related in Section 6.3) the domain is a $N \times M$ grid world where a mobile robot can — at discrete time steps — choose among four possible actions N, S, E e W

(respectively North, South, East and West), which identify the movement direction in global coordinates. Each action moves the robot to the cell corresponding to the movement direction established by the action. The environment is composed by walls, identified as positions to which the robot cannot move (see Figure 3). This domain has been used very often for experiments on mobile robot navigation learning (Drummond, 2002; Foster and Dayan, 2002; Kaelbling et al., 1996).

Training consists of a sequence of executed episodes: each episode starts by positioning the robot in a randomly chosen cell and finished once the goal cell (goal state) is reached (Mitchell, 1997).

The experiments reported here were programmed in C++ (GNU g++ compiler under Linux) and executed in a 256 MB RAM Pentium 3-500MHz computer (for the experiments in Subsections 6.1 and 6.2) and in a 512 MB RAM Pentium 4m-2.2GHz computer (Subsection 6.3).

Finding an optimal policy for small grid worlds is not difficult and can be done via a Dynamic Programming algorithm. Figure 4 shows the optimal policy found by the Policy Iteration algorithm after 38 iterations that took a total of 811 seconds for completion.

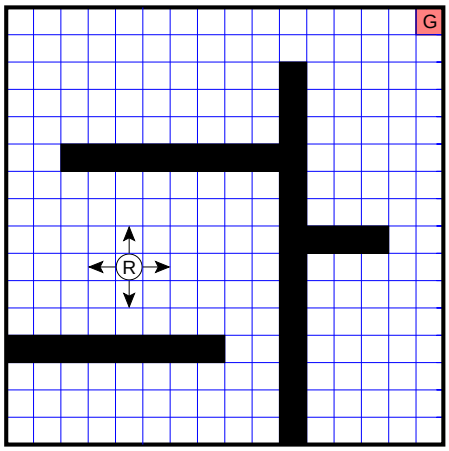


Figure 3. Room with walls (represented by dark lines) discretized over a position grid (represented by lighter lines). G represents the goal position and R the robot.

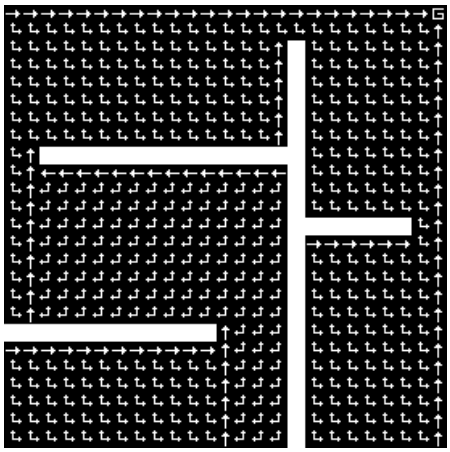


Figure 4. Optimal policy for a mobile robot in a 25 x 25 environment with some walls. Double arrows mean that for a given position it does not make any difference which of the two actions to choose.

6.1. Use of "Heuristics from Exploration" Methods in Execution Time

This section presents experiments that corroborate the hypothesis that information derived from the initial stages of training allows the defini-

tion of heuristics that can be used to accelerate learning. We used the *Q– Learning* algorithm in the domain presented in Section 3, with the following parameters: $\gamma = 0.99$, $\alpha = 0.1$ and exploration rate of 10%. Reinforcements were defined as 10 for reaching a goal state (located in the upper right corner of the environment) and -1 for every executed action.

6.1.1. *Structure Extraction*

For the experiments in the robotic domain reported in this paper, good results were obtained with the method "Structure from Exploration".

As presented in Subsection 4.1, this technique derives a crude estimate of the transition probabilities $\hat{P}_t(\mathbf{s}_t, a_t, s_{t+1})$, by annotating the results of every action performed by the agent. In a grid world with thick walls (i.e., walls that correspond to entire cells), this method generates a sketch of the map environment by annotating the visited cells, similarly to occupancy-grid methods (Elfes, 1989). The generated structure is shown in Figure 5, where in black are cells that were visited at least once and in white are never visited cells. This result was obtained after 100 training steps, using exploration rate of 10% to allows the visitation of all possible cells.

For static deterministic environments this method can construct a perfect map of the environment, provided all the state-action pair

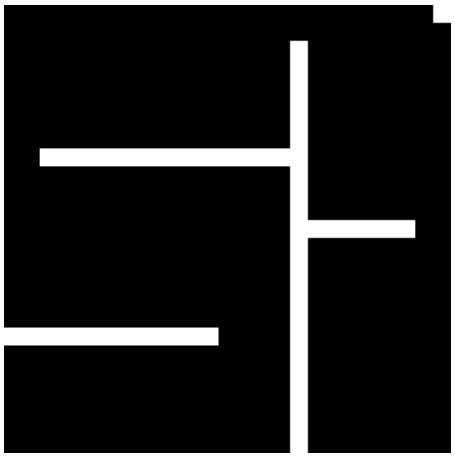


Figure 5. Structure generated by the method "Structure from Exploration", where in black are visited cells and in white never visited cells.

visitations are recorded. In the case of non-deterministic environments subject to localization errors — a very common situation in mobile robot navigation — it is necessary to find out which states have been visited less often, using a thresholding process with respect to the visitation frequency (Subsection 6.3).

The results above can also be observed for grid worlds with thin walls (i.e., walls that correspond to cell frontiers). Notice that this is the case where state transition is forbidden not necessarily due to a physical obstacle. Tests with other methods for structure extraction are described elsewhere (Bianchi, 2004).

6.1.2. *Construction of the Heuristic*

For constructing a heuristic from the structure we used the "Heuristic Backpropagation" method proposed in Subsection 4.2.

Figure 6 shows, step by step, the construction of the heuristic for an agent that must reach a single goal state located at the upper right corner of the image, based on the structure presented in Figure 5.

From left to right and top to bottom, we can see the propagation of the heuristic for the first 24 iterations of the algorithm. At each iteration, the heuristic is defined for all the states predecessors of the states that the heuristic have already been defined. For example, in the first image, there are only two states that lead to the final state. After 71 iterations, the same optimal policy presented in Figure 4 is produced (Figure 8-a). The only difference between them is that double arrows in the former figure mean that for a given position there are two equally good actions to choose, while in the latter only one action of all equally good actions for a state is defined.

Figure 7 presents the result of the same algorithm for problems with many goal states. Instead of only one final state, there are four final states, located at the four corners of the environment. The algorithm builds up the heuristic by starting propagation from all the states in the set of final states. From left to right and top to bottom, it is shown the propagation of the heuristic for the first 24 iterations (from a total

of 27) of the algorithm. The resulting heuristic is presented in Figure 8-b. Finally, Figure 9 presents the heuristic computed for four different multiple-goals problems, with two, four, five and twenty-five final states.

6.2. Experiments on HAQL

Once a heuristic has been found, we must assess its performance as a learning acceleration mechanism for the HAQL algorithm. For a first test, we produced three experiments using the "Heuristic from Exploration" method in a mobile robot domain: a) robotic navigation in an unknown environment, b) robotic navigation in a modified environment and c) robotic navigation with goal repositioning. These experiments have been used elsewhere for testing learning algorithms in mobile robotic domains (Drummond, 2002).

The heuristic function is defined from Equation 11 as:

$$H(\mathbf{s}_t, a_t) = \begin{cases} \max_a \hat{Q}(\mathbf{s}_t, a) - \hat{Q}(\mathbf{s}_t, a_t) + 1 & \text{if } a_t = \pi^H(\mathbf{s}_t), \\ 0 & \text{otherwise.} \end{cases} \quad (12)$$

This is calculated only once, in the first learning episode. In the following episodes, the value of the heuristic is maintained, allowing the learning algorithm to overcome bad action indications (e.g., if $H(\mathbf{s}_t, a_t)$

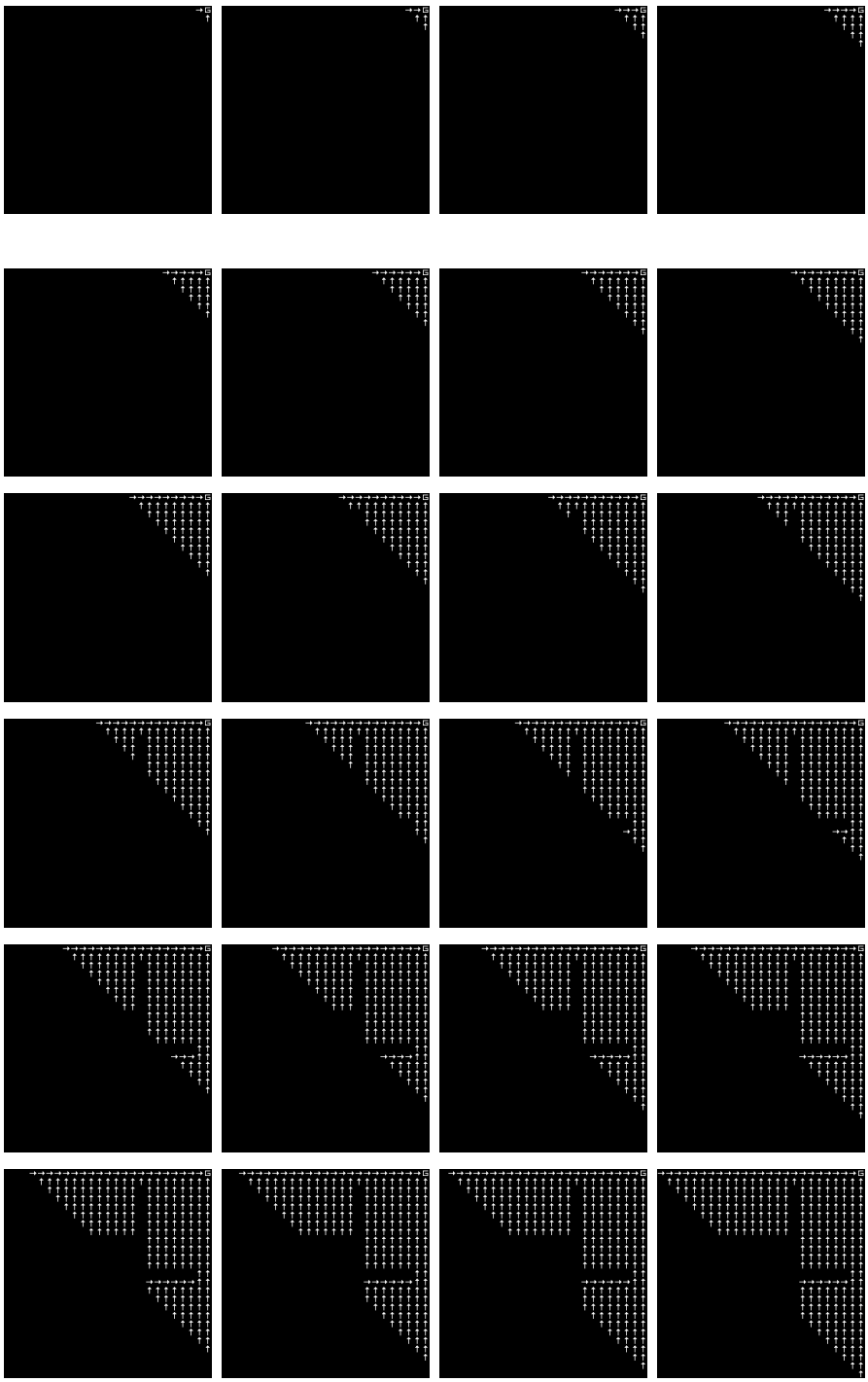


Figure 6. Heuristic defined based on the structure presented in Figure 5.

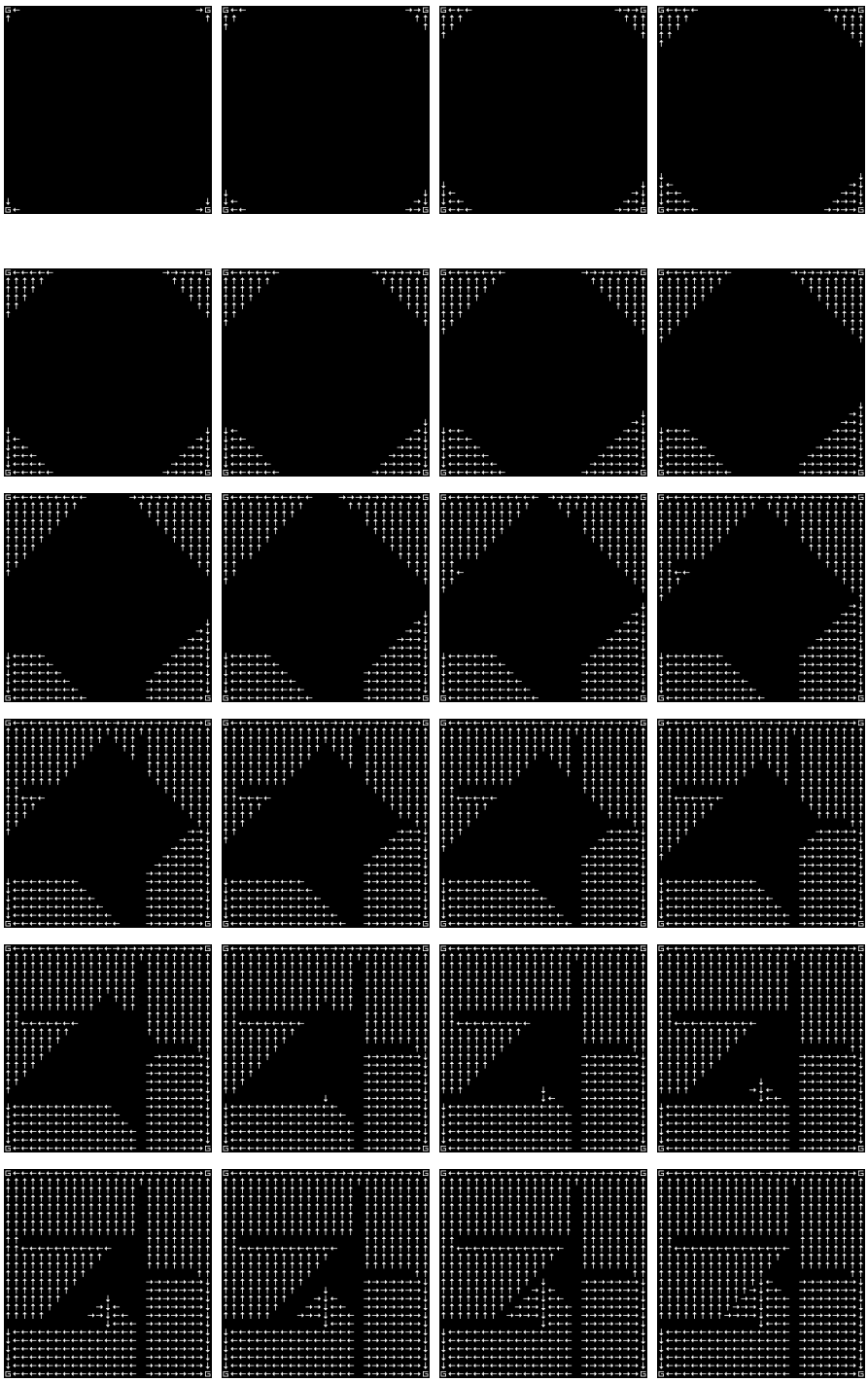
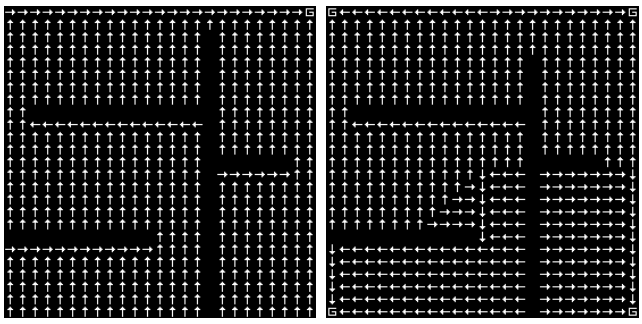
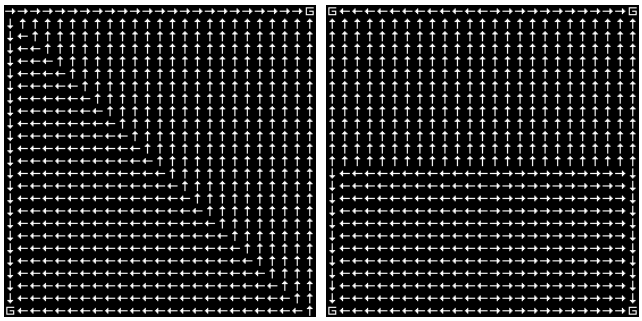


Figure 7. Heuristic defined for a problem with four final states, based on the structure presented in Figure 5.

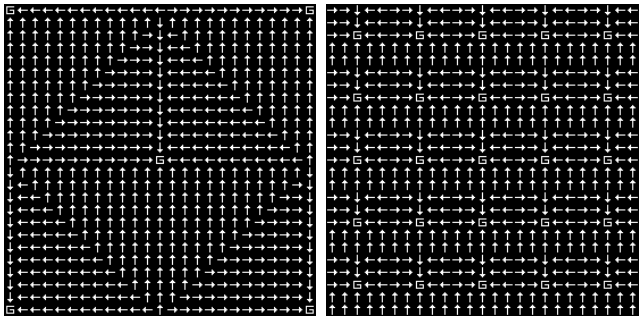


(a) (b)

Figure 8. Heuristic defined for a problem with (a) one and (b) four final states, based on the structure presented in Figure 5.



(a) (b)



(c) (d)

Figure 9. Heuristic defined for a problem with (a) two, (b) four, (c) five and (d) twenty-five final states, in an enviromnent without any walls.

was to be calculated for every episode, it would be hard to overcome a bad heuristic).

For comparison purposes, the same experiments were run with the *Q–Learning* algorithm. The parameters for the next three experiments (using *Q–Learning* and HAQL) were: learning rate $\alpha = 0.1$, $\gamma = 0.99$, exploration rate 0.1. Reinforcements were 10 units reward for reaching the goal state and -1 unit punishment for every action execution.

6.2.1. *Robotic Navigation in an Unknown Environment*

In this experiment, a randomly positioned robot must learn a policy that leads it to the goal in an unknown environment. For this task we carried out a comparison between *Q–Learning* and HAQL using the method "Heuristic from Exploration" for accelerating learning from the 10^{th} learning episode (defined empirically).

From the first to the ninth learning episode, HAQL extracts the domain structure, with no use of heuristics. Thus, it behaves similarly to *Q–Learning*, but with a mechanism for structure extraction operating in parallel. At the end of the ninth episode, the heuristic is built up using Heuristic Backpropagation, and the values of $H(\mathbf{s}_t, a_t)$ are defined from Equation 12.

The 10^{th} episode was chosen for initiating the acceleration process, thus allowing some prior exploration. The likelihood of finding a good

heuristic is increased because the environment is small and the robot is repositioned at a random position at the start of each episode.

Results were averaged over 30 training sessions held in nine different configurations for a 55×55 positions environment – a room with walls and hallways – as shown in Figure 10. In this experiment, the goal is located in the upper left corner.

Results in Figure 11 present the average of all 270 training sessions (30 sessions for each of the 9 configurations). It can be seen that whilst *Q–Learning* keeps on looking for a best action policy, HAQL starts using an optimal policy immediately after acceleration, executing the minimum number of steps to reach the goal after the 10^{th} episode.

The Student T-test (Spiegel, 1975) was used to validate the hypothesis that HAQL actually accelerates learning. The absolute value of T was calculated for each episode using the same data presented in Figure 11. Figure 12 shows that from the tenth iteration onwards Q-learning and HAQL have remarkably distinct behavior (with a superior performance by the latter), with a significance level of 0.01% (that is, if the absolute value of T is greater than the 0.01% bound, the performance is statistically different).

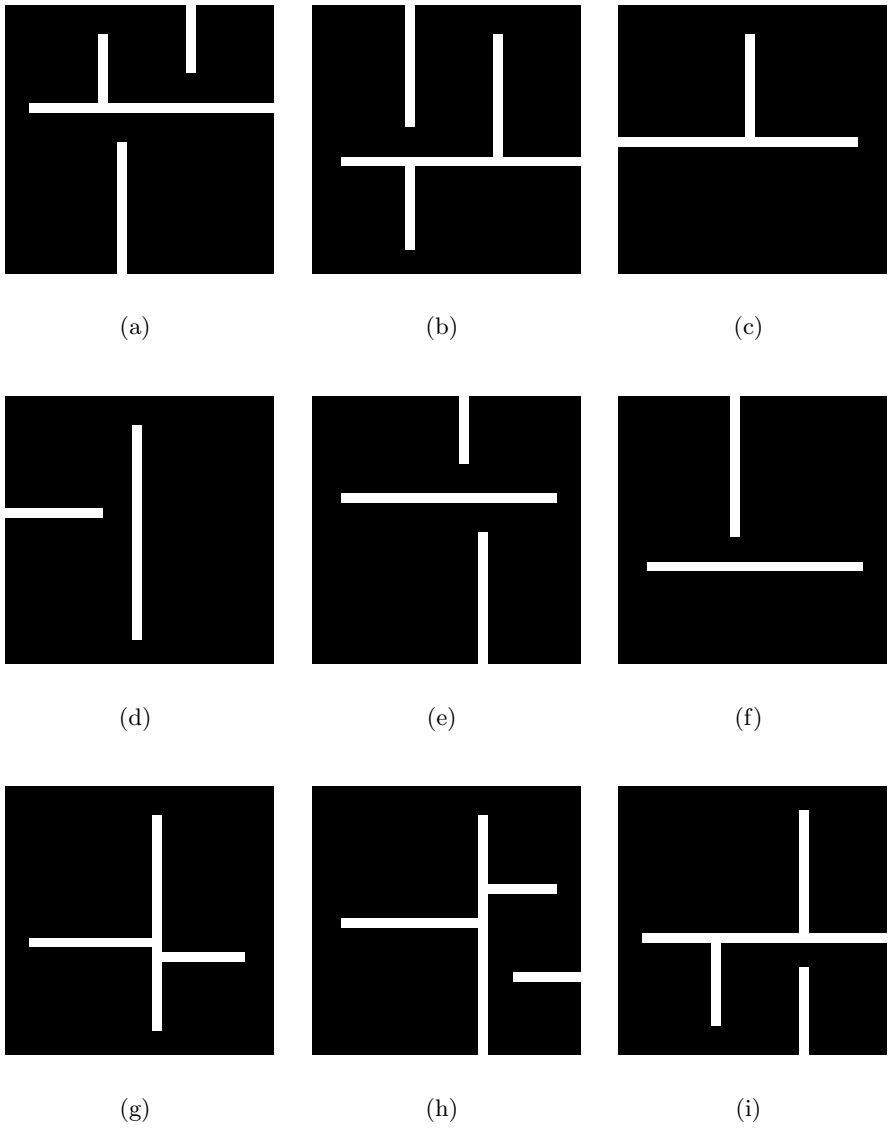


Figure 10. Nine different configurations of a room with walls (the white lines). The goal is located in one of the corners.

6.2.2. *Robotic Navigation in a Modified Environment*

The heuristic created by the "Heuristic from Exploration" method does not always correspond to an optimal policy. The environment map

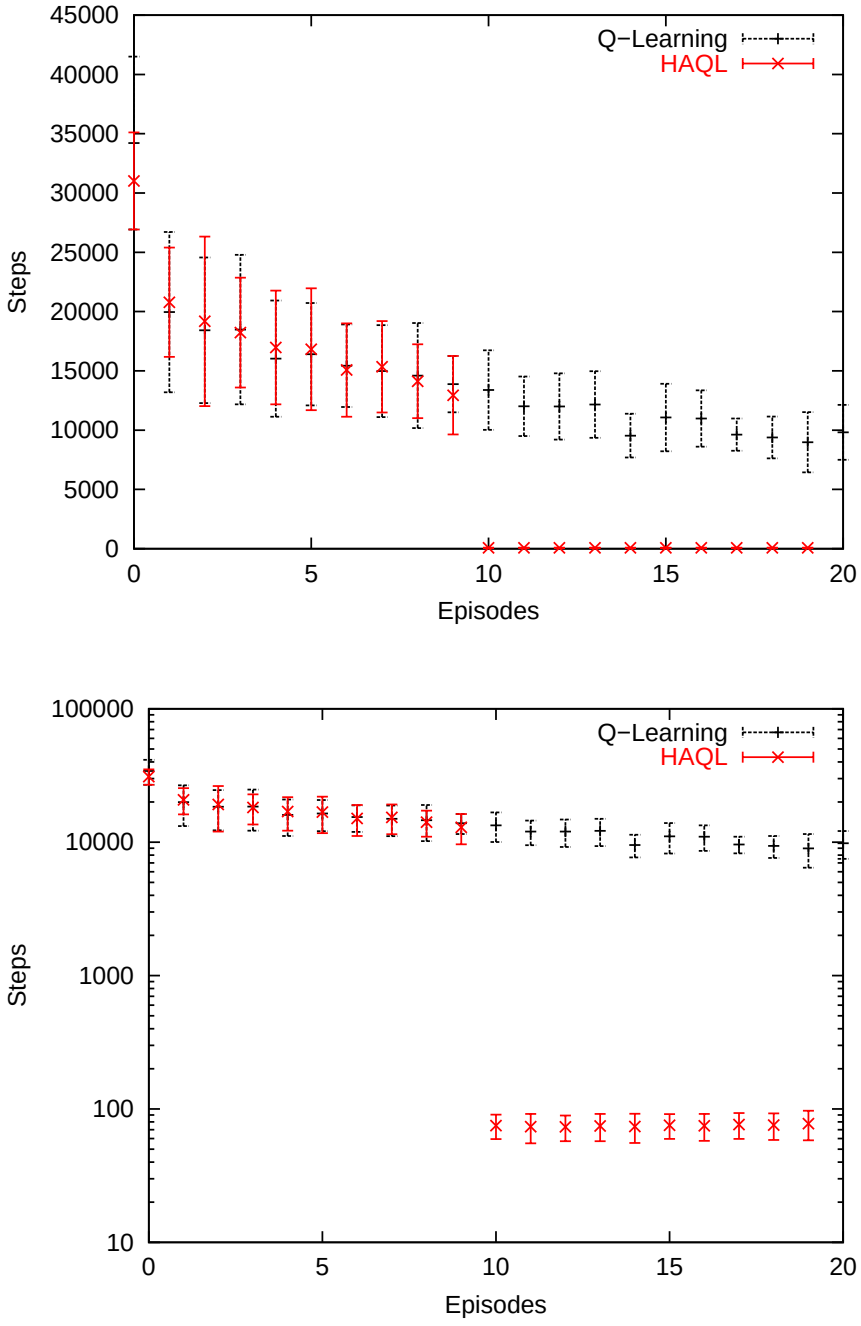


Figure 11. Learning results for acceleration in an unknown environment, at the end of the 10th episode, using "Heuristic from Exploration". (Number of steps to reach the goal x episode number, with error bars. Lower in monolog.)

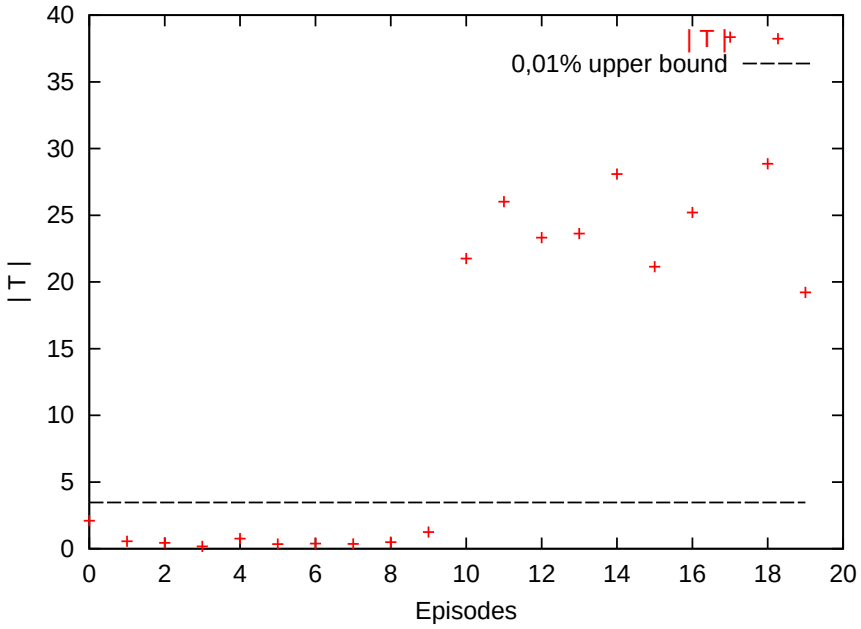


Figure 12. Results of the Student T-test for an unknown environment.

sketch can be far from perfect, and some actions pointed as adequate by the heuristic may not be. For instance, if the structure extraction process fails to completely determine a wall, the heuristic can force the robot to keep trying to move to an invalid position.

Similarly to the experiment described in the last section, here a robot must learn how to reach a goal once it is placed in a random position of an unknown environment. However, at the end of the ninth iteration a small modification of the environment is artificially produced. Our aim

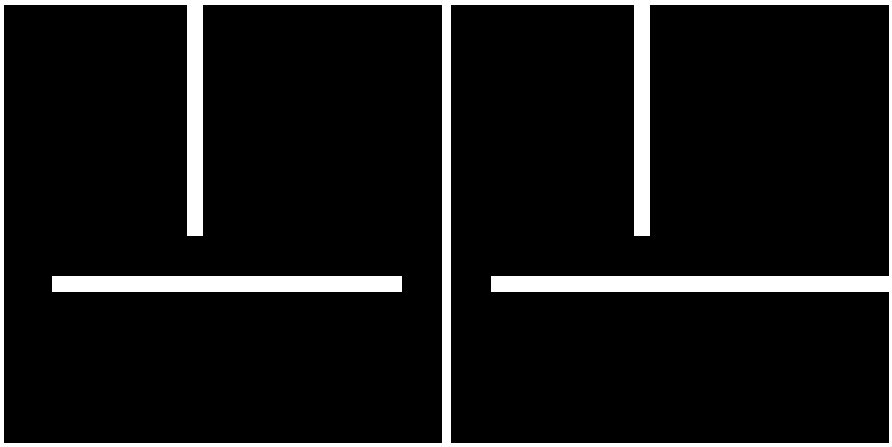
is to verify how HAQL performs when such modifications occur, thus simulating a heuristic that is not completely adequate.

Between the first and the ninth episode, HAQL simply extracts the environment structure with no use of a heuristic. At the end of the ninth episode (and therefore coincidentally with the environment modification), the heuristic is designed using the "Heuristic backpropagation", method, and the values of $H(\mathbf{s}_t, a_t)$ are defined from Equation 12. From the 10th episode onwards this heuristic is then used to accelerate learning.

For this experiment a single environment configuration with 55×55 positions was used, as shown in Figure 13. The goal is located in the upper left corner. Figure 13-a shows the environment up to the end of the ninth episode. Notice that there are hallways to the left and right of the horizontal wall. At the start of the tenth episode, the right hallway is closed, as can be seen in Figure 13-b.

The heuristic generated by the method "Heuristic from Exploration" for this problem is shown in Figure 14-a. Notice that is very similar to the optimal policy for the environment with closed hallway in the right horizontal wall (Figure 14-b).

Figure 15 presents results for HAQL with the heuristic created for the environment with an incomplete wall (averaged over 30 training sessions). At the moment of the start of the acceleration process, which



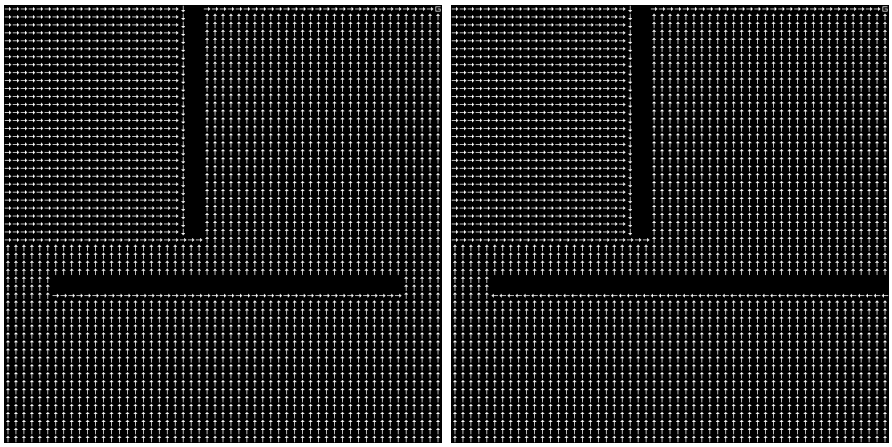
(a) Environment up to the end of the ninth episode

(b) Environment from the tenth episode onwards

Figure 13. (a) The environment used for defining the heuristic and (b) the environment where the heuristic is used.

is the same moment that the environment is modified, a worsening of performance occurs, but acceleration begins as soon as the agent learns to ignore the heuristics in the states they are not effective.

The result of the Student T-test for the data presented in Figure 15 can be seen in Figure 16. Notice that from the 60th iteration onward the majority of the results is significantly distinct due to the acceleration process in HAQL, with a confidence level of at least 95%.



(a) The heuristic extracted (b) The optimal policy

Figure 14. (a) The heuristic extracted from the environment of Figure 13-a and (b) the optimal policy for the environment of Figure 13-b. (55 x 55 cells.)

6.2.3. *Robotic Navigation with Goal Repositioning*

The goal of this experiment is to compare how the *Q–Learning* algorithm and the HAQL algorithm behave when the goal is repositioned in an advanced stage of the learning process.

Once more HAQL initially behaves as *Q–Learning*, however with simultaneous execution of the structure extraction algorithm. At the end of the 4999^{th} episode (empirically defined as an advanced stage of the learning process) the goal is repositioned from the upper right corner to the lower left corner.

As a result, both algorithms have to find the new goal position, but as both are following the previously learned policy, there is performance

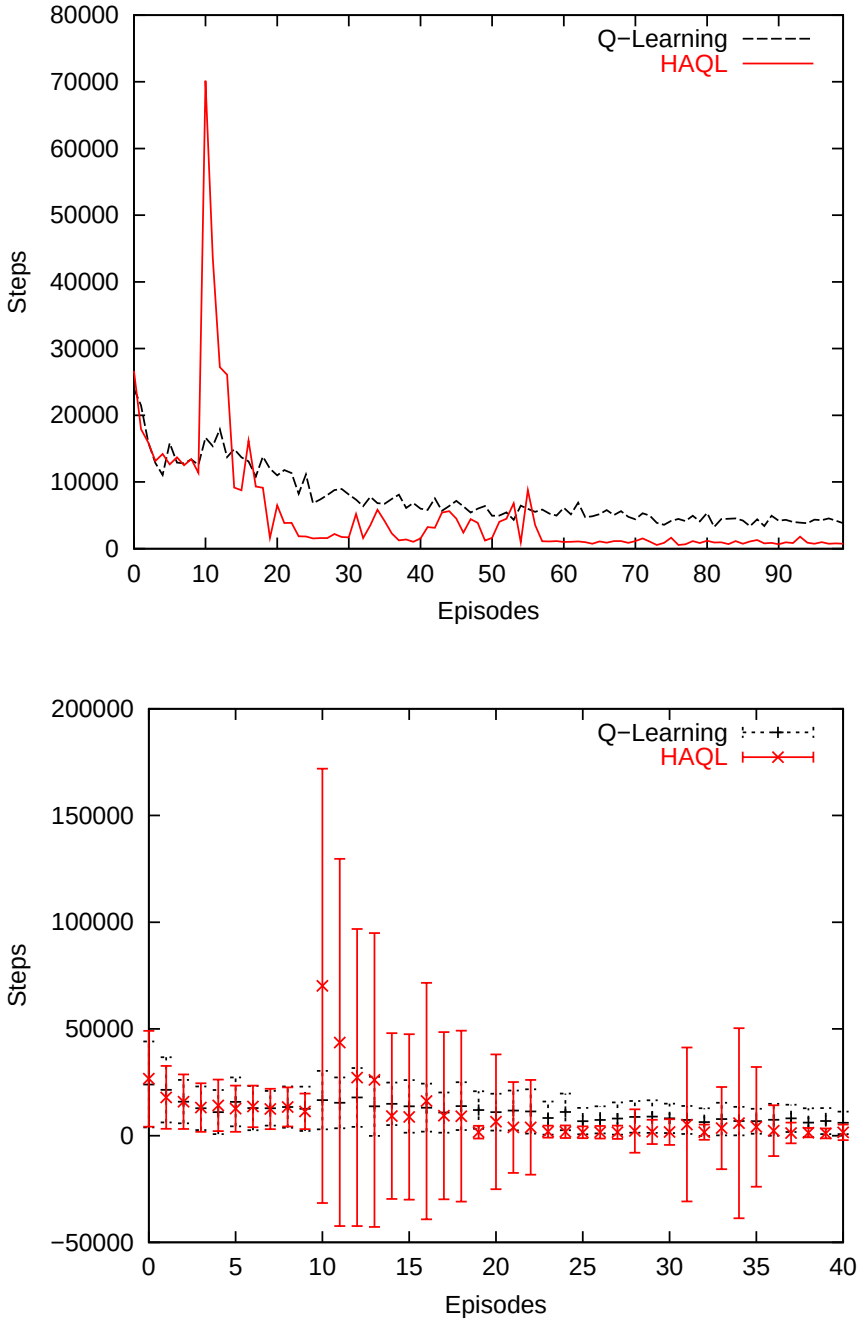


Figure 15. Learning results for acceleration in a modified environment at the end of the tenth episode, using "Heuristic from Exploration". (Number of steps to reach the goal x episode number. Lower with error bars.)

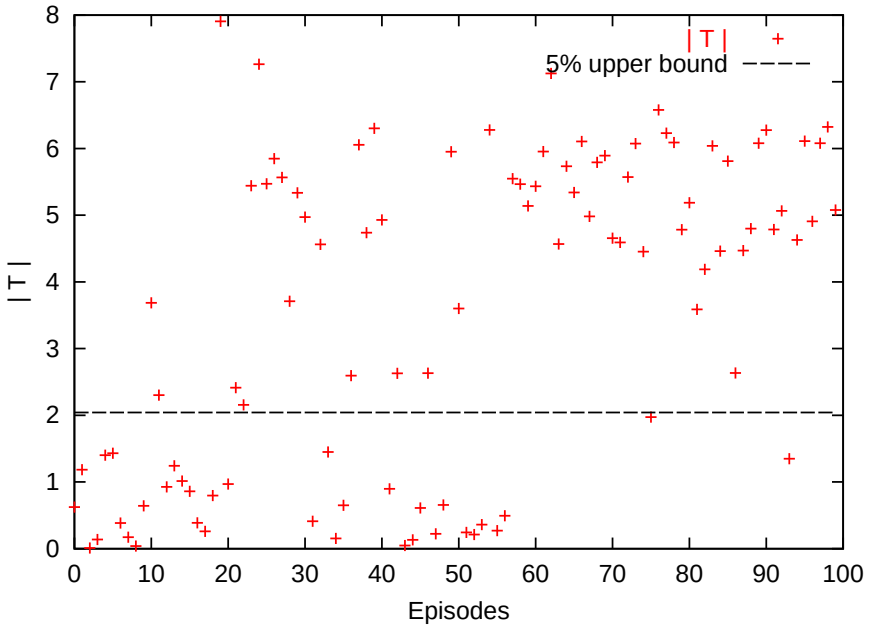


Figure 16. Results of the Student T-test for acceleration at the tenth iteration in a modified environment.

degradation and new requirements for execution of a large number of steps to reach the new goal position.

As soon as the HAQL-controlled robot reaches the goal (at the end of the 5000^{th} episode), the heuristic to be used is designed using "Heuristic Backpropagation" from the (not modified) environment structure and the new goal position, and the $H(\mathbf{s}_t, a_t)$ values are defined. This heuristic is then used from the 5001^{st} episode, resulting in better performance with respect to *Q–Learning*, as shown in Figure 17.

As expected, HAQL has performance similar to *Q–Learning* up to the 5000^{th} episode. At this stage, both *Q–Learning* and HAQL need around 1 million steps to reach the new goal position.

After goal repositioning, while *Q–Learning* needs to relearn the whole policy, HAQL always performs the minimum number of steps required for reaching the goal. Figure 17 also shows that the deviation in each episode is small due to the fact that the agent is positioned in different initial states for each training session.

The Student T-test was used to statistically validate the results. The absolute value of T was calculated for each episode using the same data presented in Figure 17. The results are shown in Figure 18. From the 5001st iteration results are significantly different, with a level of confidence of more than 99%.

6.2.4. *Discussion*

This section presented a comparison between *Q–Learning* and HAQL in an autonomous mobile robot domain. It was verified that the use of heuristics by HAQL accelerates learning through the use of information acquired during the learning process.

Very often, slight structural modifications in the domain causes havoc in the problem solving process by initializing a learned policy and requiring complete relearning. Subsections 6.2.2 and 6.2.3 show that

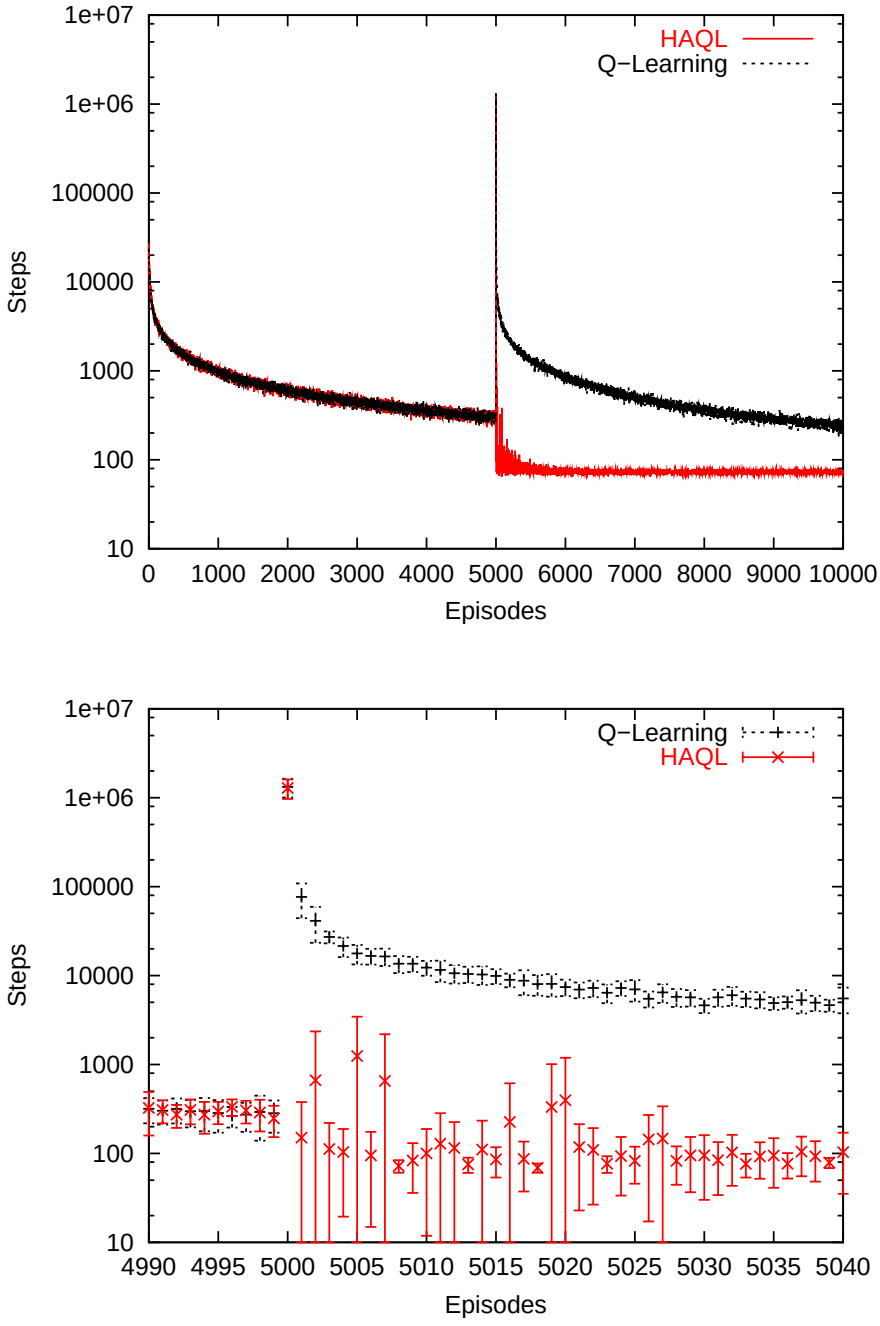


Figure 17. Learning results for goal repositioning in the 5000*th* episode, using "Heuristic from Exploration" in HAQL. (Number of steps to reach the goal x episode number, in monolog scale in the upper figure, with error bars in the lower figure.)

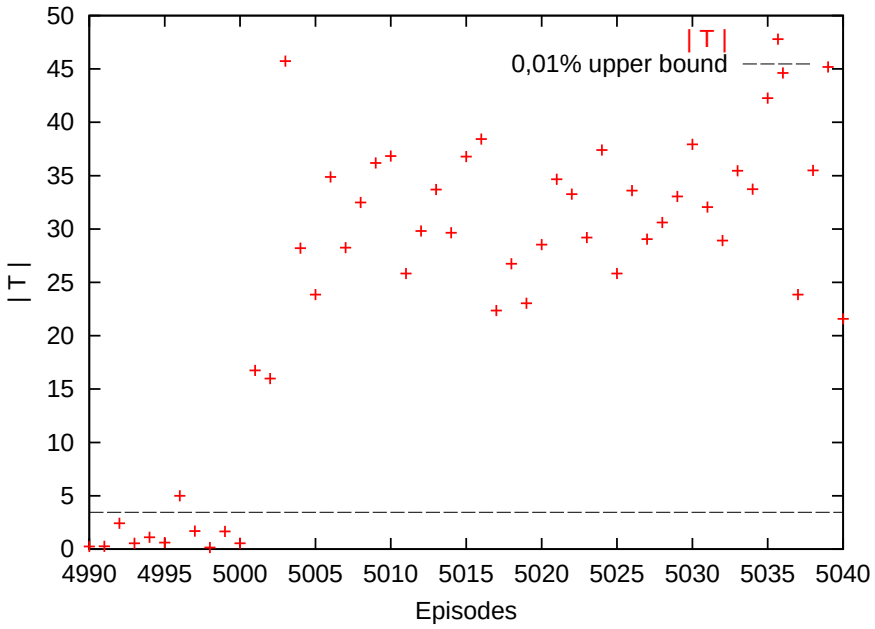


Figure 18. Results of the Student T-test for goal repositioning in monolog.

HAQL permits reuse of knowledge acquired during the learning process. Reuse is carried out in a very simple manner, in the policy itself and not in the value function estimate (Drummond, 2002). Another interesting feature of HAQL is that even if the heuristic is not completely adequate, performance enhancement can take place (as shown in subsection 6.2.2) due the partial correctness of the heuristic.

6.3. Simulation of a real robot

The objective of this section is to verify the performance of *Heuristically Accelerated Q–Learning* in a navigation problem for a simulated mobile

robot, with a more complex dynamics than a grid world, operating in a non-deterministic environment and subject to positioning errors.

The chosen platform is the Saphira 8.0 simulator (Konolige and Myers, 1996) controlling a Pioneer 2DX mobile robot. Its simulation module incorporates error models that are very close in behavior to those of the real sensors – sonars and encoders – in such a way that if a software routine operates in the simulator it is very likely that it will be also adequate for a real implementation.

Saphira uses a client-server architecture. The server module uses a global map based on a reference coordinate system, accessible only to the simulator, which represents the robot environment. The client module controls the robot and uses a local map on coordinates centered on the robot.

The reference coordinate system is a 2-D cartesian system. If any path is initiated from a known position with a known velocity and movement direction are accurately measured, the robot state (or pose) can be defined by integrating measurements of wheel turns in a process known as dead-reckoning. Unfortunately, dead-reckoning is prone to cumulative errors due to wheel slippage, surface irregularities and measurement errors. Thus, in realistic situations, techniques that combine information from an action model and from sensors are required for reducing the localization error.

For the experiment reported in this section (and in contrast with the experiments on grid worlds) the robot state was determined from local sensor and action models for state estimation based on belief updating through the Monte Carlo localization technique (Fox et al, 1999; Sebastian Thrun et al., 2001). Determination of a global state from local sources of information is actually a basic competence for a mobile robot.

Figure 19 shows the Saphira 8.0 simulation environment. The upper image corresponds to the client, which controls the mobile robot. The small squares around the robot result from the sonar sensor readings, and the points correspond to the particles (state candidates) used for Monte Carlo localization. The arrow indicates the likeliest robot orientation. The lower image shows the simulator screen, presenting the real pose of the robot. Figure 20 is a detailed view of the particles used for Monte Carlo localization.

The environment is a simulated 10×10 meters square, discretized as a grid of 20×20 cells (the length of each cell — 0.5 meters — corresponds approximately to the diameter of the robot). The robot orientation was partitioned in 16 discrete values, each corresponding to a 0.35 radians sector. Thus, the set of discretized state variables (x, y, θ) used for learning is a coarse representation of the robot real state.

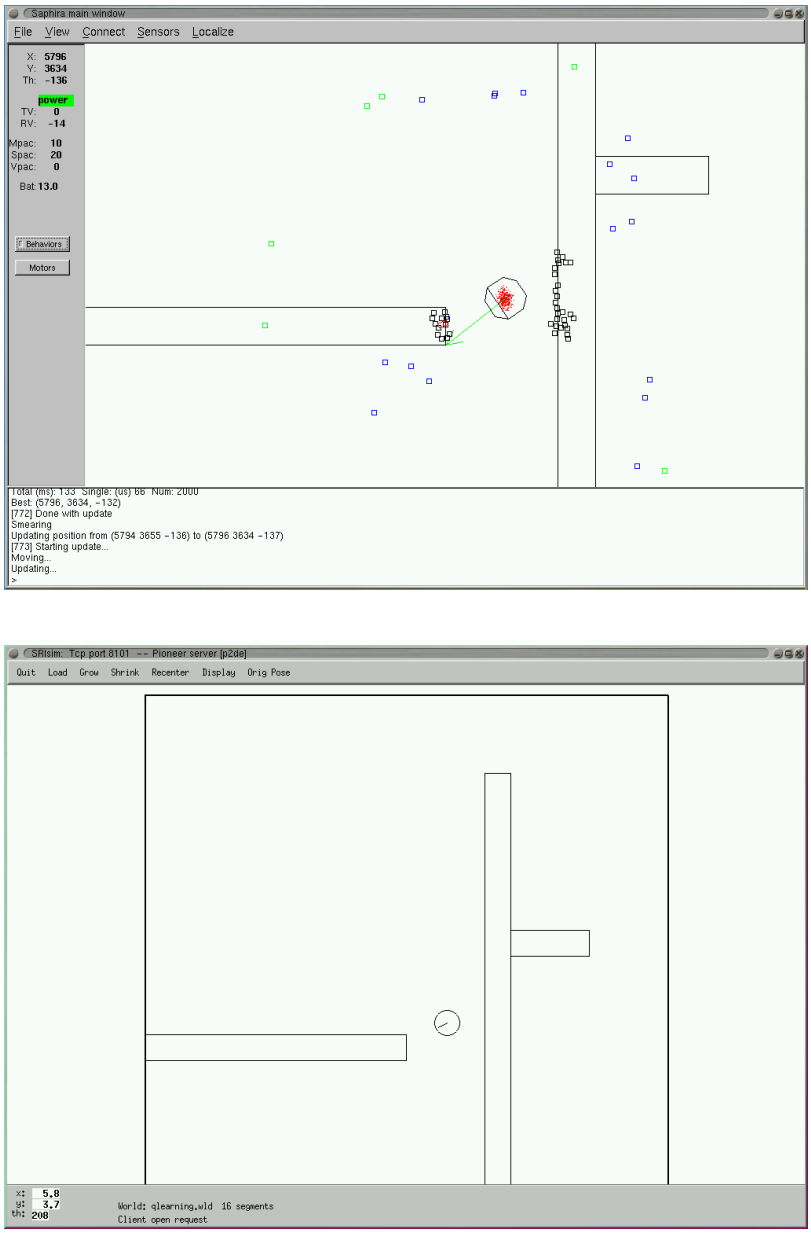


Figure 19. The Saphira 8.0 simulation platform. The upper figure shows the client screen (with the robot position in the reference plane). The lower figure is the simulator screen (showing the real position of the robot).

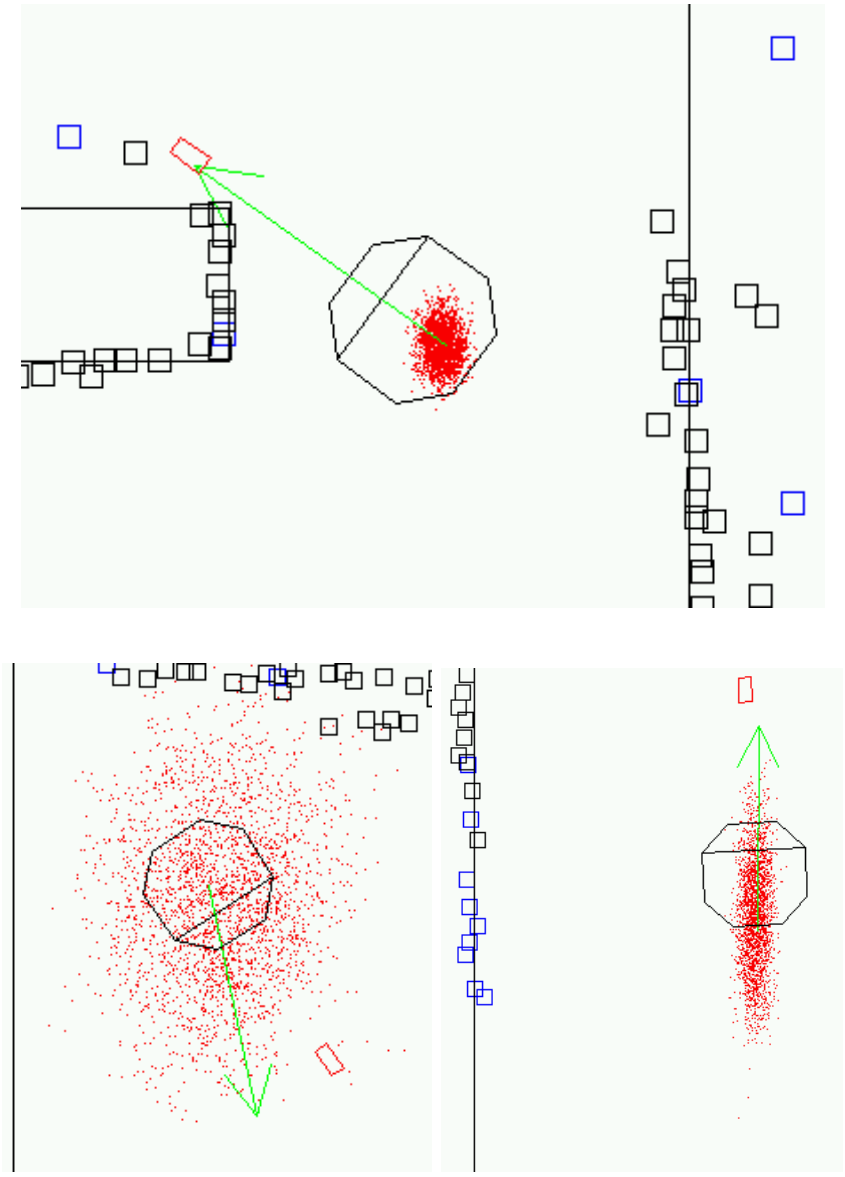


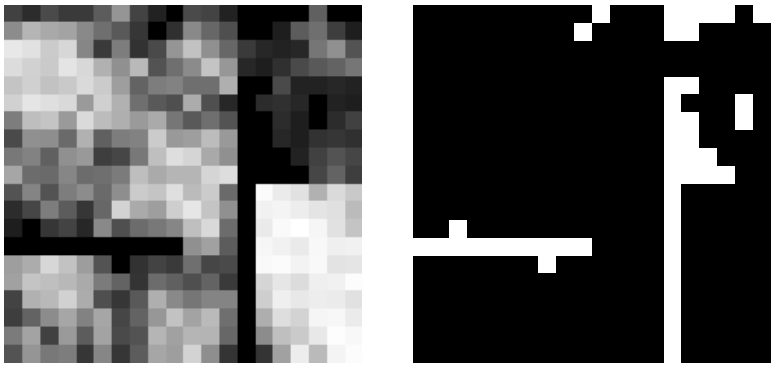
Figure 20. Monte Carlo localization: points indicate the position of particles, each one defining a possible location of the robot. State uncertainty is lower in the upper image than in the lower images.

Four actions were defined as executable by the robot: move ahead or backwards (a distance corresponding to the robot diameter) and turn 0.35 radians either clockwise or anticlockwise.

In this domain, we made a comparison between *Q–Learning* and HAQL using the "Heuristic from Exploration" method to accelerate learning from the 5^{th} episode onwards. The environment can be seen in the lower part of Figure 19.

To extract information about the environment structure we used the "Structure from Exploration" method: every time the robot visits a cell, a frequency counter for that cell is incremented. At the end of the fifth iteration, this computation is thresholded, resulting in a environment map sketch that is used to crete the heuristic via Heuristic Backpropagation. Figure 21 shows one of the created map sketches. As can be seen, the map is not homomorphic with respect to the environment, resulting in a heuristic which does not indicate the optimal policy. Furthermore, the created heuristic (presented in Figure 22) drives the robot to regions very close to the walls, allowing it to get immobilized.

Parameters for both *Q–Learning* and HAQL were: learning rate $\alpha = 0.1$, $\gamma = 0.99$ and exploration rate of 10%. The Monte Carlo localization algorithm operated over a set of $10,000$ particles. Reinforcements were: 1000 for reaching the goal state and -10 for any executed action. The robot starts each training episode from a random pose, and the goal



(a) Number of visits (b) Map

Figure 21. Number of visits (lighter tones indicate more visits) and map sketch created using the method "Structure from Exploration" for the Saphira environment.

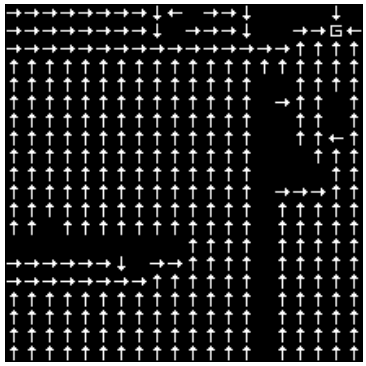


Figure 22. The heuristic created from the map sketch presented in Figure 21.

corresponds to a region in the upper right corner (defined as the region $9250mm \leq x, y \leq 9750mm$).

Results averaged over 30 experiments are shown in Figure 23. Note that, even with the use of Monte Carlo localization, there is large variation on the results for the Q–Learning algorithm, caused by cumu-

Table III. Results of the Student T-test: heuristic learning acceleration in the fifth episode — HAQL in the Saphira simulation environment.

Episode	Q–Learning (Number of steps to reach the goal)	HAQL (Number of steps to reach the goal)	$\|T\|$	Confidence Level
5	7902 ± 8685	65 ± 51	4.942	0.01%
6	1010 8 ± 10398	55 ± 43	5.295	0.01%
7	8184 ± 12190	72 ± 61	3.644	0.02%
8	8941 ± 8367	75 ± 71	5.803	0.01%
9	8747 ± 9484	63 ± 55	5.015	0.01%

lative localization errors derived from the intensive exploration of the environment. This variation can also be seen in the data presented on the 2^{nd} column of Table III. As the number of training steps is greatly reduced by using the heuristic, the cumulative localization error and the variation is much lower for HAQL (Table III, 3^{rd} column). The Student T-test on this experiment (Table III, 4^{th} column) shows that from the fifth iteration results are significantly distinct, with a significance level close to 0.01% (Table III, 5^{th} column).

Each 10-episode training experiment took approximately 24 hours for completion using *Q–Learning* and an average of 1 minute for HAQL.

Finally, Figure 24 shows the learned paths followed by the robot using *Q–Learning* (upper) and HAQL (lower). The paths are a result of the fifth training episode for both algorithms (therefore, with a heuristic actuation on HAQL). The robot was released from the bottom left cell and was required to reach the goal located in the top right corner.

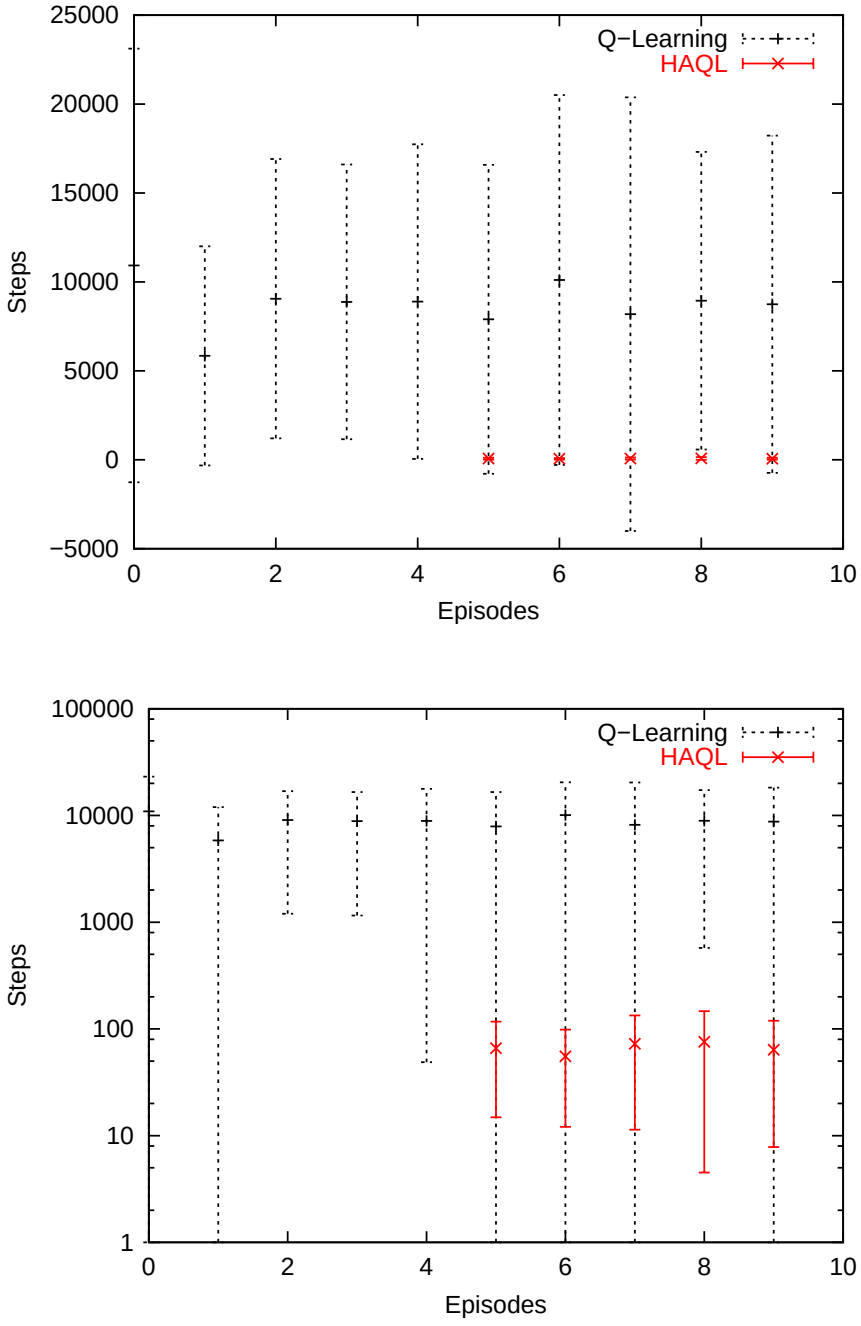


Figure 23. Results of heuristic learning acceleration in the fifth episode — HAQL in the Saphira simulation environment. (Number of steps to reach the goal x episode number.)

Note that the exploratory nature of *Q–Learning* produced random-like walks, generating 12081 learning steps for reaching the goal. HAQL produced only 86 steps for reaching the goal. Clearly, the role of the heuristic is to limit exploration.

7. Conclusions and Future Work

We have proposed a general formulation for reinforcement learning algorithms – HAL – that combines on-line policy learning with heuristic functions for selecting promising actions during the learning process. This way, heuristics can guide exploration of the state-action space so that the rate of convergence of an RL algorithm can be increased.

An important characteristic of HAL algorithms is that the heuristic function can be modified or adapted online, as learning progresses and new information for enhancement of the heuristic becomes available. In particular, either prior domain knowledge or previous learning stage information can be used to accelerate learning. Another important feature is that the heuristic function is an action policy modifier that does not interfere with the standard bootstraping update mechanism of RL algorithms.

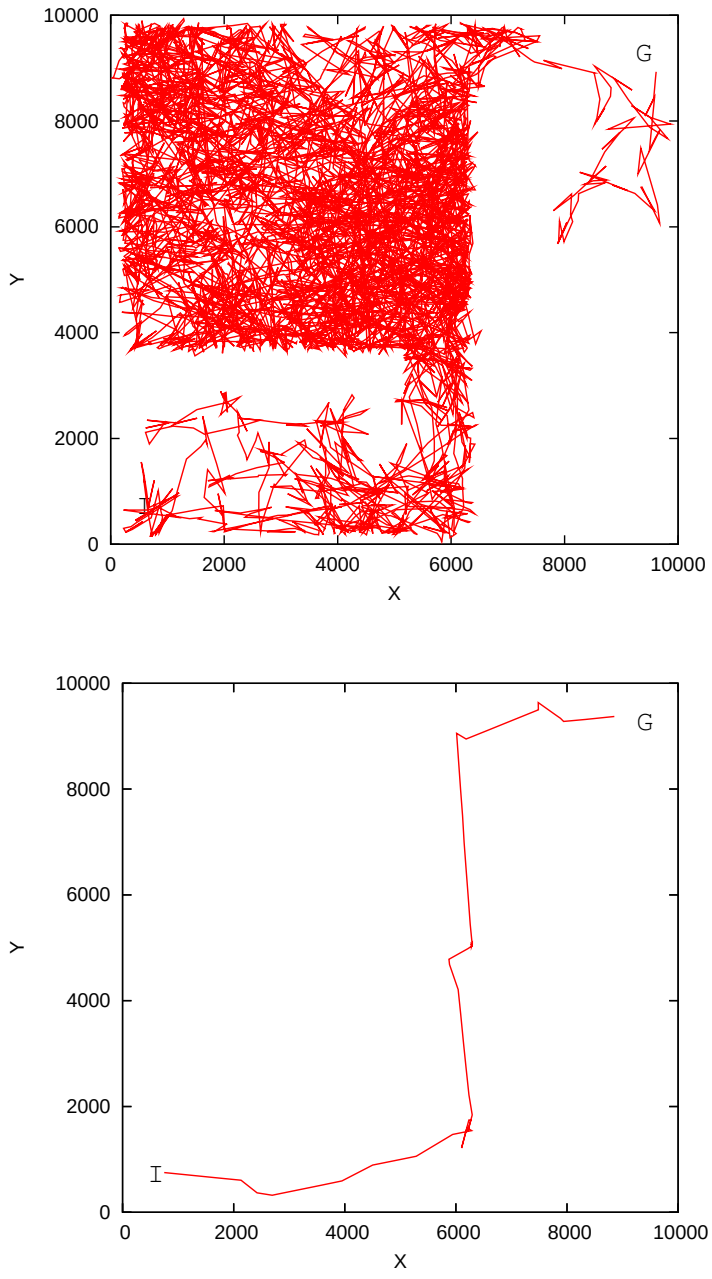


Figure 24. Paths followed by the robot using *Q–Learning* (upper) and HAQL (lower) in the Saphira 8.0 simulation environment, after the 5^{th} training episode.

We also have contributed a new learning algorithm, Heuristically Accelerated Q-learning (HAQL), which incorporates heuristics for action selection to the Q-Learning algorithm. A series of empirical evaluation of HAQL in a commercial simulator for the robot navigation domain were carried out. Experimental results showed that the performance of the learning algorithm can be improved even using very simple heuristic functions.

The experiments reported in this work were carried out in static environments. Despite the small number of experiments conducted in a non-deterministic environment (Section 6.3), we believe that the framework proposed applies to stochastic problems as well as to deterministic ones.

Another important topic to be investigated in future works is the use of generalizations of the value function space used to generate the heuristic function.

Future works also include investigations on how to accelerate reinforcement learning by using heuristics transfer from related tasks. Without such transfer, even if two tasks are very similar at some abstract level, an extensive re-learning effort is required. However, it seems more promising to transfer parts of previously learned solutions rather than a single complete solution. These solution pieces represent knowledge (built in heuristics) about how to solve certain subtasks.

The same idea can be extended to multiagent learning, so that knowledge can be exchanged among learning agents in order to improve the multiagent performance.

Acknowledgements

This research was conducted under the CAPES/GRICES Project Multi-Bot(Grant no. 099/03). Carlos H. C. Ribeiro is grateful to CNPq (Grant no. 301228/97-3-NV).

References

Banerjee, B.; Sen, S. and Peng, J. Fast concurrent reinforcement learners. In: *Procs. of the International Joint Conference on Artificial Intelligence (IJCAI'2001)*, pp. 825–830, 2001.

Bertsekas, D. P. *Dynamic Programming: Deterministic and Stochastic Models* . Prentice-Hall, Upper Saddle River, NJ, 1987.

Bertsekas, D. P. *Dynamic Programming and Optimal Control, Vol. 1.* Athena Scientific, Belmont, MA, 1995.

Bianchi, R. A. C. *Using Heuristics to accelerate Reinforcement Learning algorithms (in portuguese).* PhD Thesis, University of São Paulo, 2004.

Bonabeau, E.; Dorigo, M. and Theraulaz, G. Inspiration for optimization from social insect behaviour. *Nature 406*, 2000.

Butz, M. V. State Value Learning with an Anticipatory Learning Classifier System in a Markov Decision Process. *Technical Report 2002018 at the Illinois Genetic Algorithms Laboratory*, 2002.

Drummond, C. Accelerating Reinforcement Learning by Composing Solutions of Automatically Identified Subtasks. *Journal of Artificial Intelligence Research 16*, 2002, pp. 59–104.

Elfes, A. Using Occupancy Grids for Mobile Robot Perception and Navigation. *Computer 22*, 1989, pp. 46–57.

Foster, D. and Dayan, P. Structure in the Space of Value Functions. *Machine Learning 49*, 2002, pp. 325–346.

Fox, D; Burgard, W. and Thrun, S. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research 11*, 1999, pp. 391–427.

Hart, P. E., Nilsson, N. J. and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 1968, pp. 100–107.

Kaelbling, L. P.; Littman, M. L. and Moore, A. W. Reinforcement Learning: A survey. *Journal of Artificial Intelligence Research 4*, 1996, pp. 237–285.

Karlin, S., and H. M. Taylor. *A First Course in Stochastic Processes*. Academic Press, 1975.

Konolige, K. Improved occupancy grids for map building. *Autonomous Robots 4*, 1997, pp. 351–367.

Konolige, K. and Myers, K. The Saphira Architecture for Autonomous Mobile Robots. In: *AI-based Mobile Robots: Case studies of successful robot systems*, MIT Press, Cambridge, MA, 1996.

Littman, M. L. Markov games as a framework for multi-agent reinforcement learning. In: *Proceedings of the Eleventh International Conference on Machine Learning (ICML'94)* (1994), Morgan Kaufmann, pp. 157–163.

Littman, M. L., and C. Szepesvári A generalized reinforcement learning model: Convergence and applications. In: *Procs. of the Thirteenth International Conf. on Machine Learning (ICML'96)* (1996), pp. 310–318.

Mitchell, T. *Machine Learning*. McGraw Hill, New York, 1997.

Moore, A. W. Variable Resolution Dynamic Programming: Efficiently Learning Action Maps in Multivariate Real-valued State-spaces. In: *Proceedings of the Eight International Conference on Machine Learning (ICML'91)* (1991), Morgan Kaufmann, pp. 57–69.

Moore, A. W. and Atkeson, C. G. Prioritized Sweeping: Reinforcement Learning With Less Data and Less Time. *Machine Learning 13*, 1993, pp. 103–130.

Peng, J. and Williams, R. J. Efficient Learning and Planning within the Dyna framework. *Adaptive Behavior 1*, 1993, pp. 437–454.

Puterman, M. L. *Markovian Decision Problems*. John Wiley, 1994.

Rummery, G. and Niranjan, M. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

Russell, S. and Norvig, P. *Artificial Intelligence: a modern approach*. Prentice Hall, 2nd edition, 2002.

Thrun, S.; Fox, D. Burgard, W. and Dellaert, F. Robust Monte Carlo localization for mobile robots. *Artificial Intelligence 128*, 2001, pp. 99–141.

Singh, S., Jaakkola, T. and Jordan, M. Reinforcement Learning with Soft State Aggregation. *Advances in Neural Information Processing Systems 7*, pp. 361–368, 1995.

Spiegel, M. R. *Probability and Statistics*. McGraw-Hill, New York, 1975.

Sutton, R. S. Learning to Predict by the Methods of Temporal Differences. *Machine Learning 3*, 1988, pp. 9–44.

Sutton, R. S. *Integrated architectures for learning, planning and reacting based on approximating dynamic programming*. Proceedings of the 7th International Conference on Machine Learning, Austin, TX, 1990.

Sutton, R. S. Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. *Advances in Neural Information Processing Systems 8*, 1996, pp. 1038–1044.

Sutton, R. S. and Barto, A. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

Szepesvári, C. Static and Dynamic Aspects of Optimal Sequential Decision Making. PhD thesis, Jozsef Attila University, Szeged, Hungary.

Szepesvári, C., and Littman, M. Generalized markov decision processes: Dynamic-programming and reinforcement-learning algorithms. CS-96-11, Brown University, Department of Computer Science, Brown University, Providence, Rhode Island 02912, 1996.

Tsitsiklis, J. and Roy, B. V. Feature-based Methods for Large Scale Dynamic Programming. *Machine Learning 22*, 1996, pp. 59–64.

Tesauro, G. Temporal Difference Learning and TD-Gammon. *Communications of the ACM 38*, 1995, pp. 58–67.

Watkins, C. J. C. H. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, 1989.