Induction of Macro-Actions Based on Local State Features in Reinforcement Learning

Pascal Garcia
INSA/IRISA
20 Avenue des Buttes de Coësmes, F-35043 Rennes Cedex, France
pgarcia@irisa.fr

Abstract
Given several problems to solve in some domains, a standard reinforcement learning agent learns an optimal policy from scratch for each problem. It would be far more useful to have systems that can solve several problems over time, using the knowledge obtained from previous problem instances to guide in learning on new problems. In this paper, we present an algorithm to induce automatically macro-actions based on the solutions of previous related tasks and local state features. Those macro-actions will be added to the primitive actions of a new task. They will help the reinforcement learning agent to explore more efficiently the new environment, while ensuring that the optimal solution can still be found.

1 Introduction
Reinforcement learning is a general framework in which an autonomous agent learns which actions to choose in particular situations (states) in order to optimize some reinforcements (rewards or punishments) in the long run [Sutton and Barto, 1998]. A fundamental problem of its standard algorithms is that although many tasks can be formulated in this framework, in practice, for moderately large state space, they are not solvable in reasonable time. There are two principal approaches for addressing these problems. The first approach is to apply generalization techniques, which involve approximations of the value function [Bertsekas and Tsitsiklis, 1995]. The second approach is to guide the agent with a priori knowledge about the environment. We can incorporate this knowledge either in modifying the reward function as in the reward shaping techniques [Ng et al., 1999] or we can create macro-actions [Parr, 1998; Precup, 2000; Dietterich, 2000]. A macro-action is a way of grouping actions to create a new one. For example, if the primitive actions of a problem are “make a step in a given direction”, a macro-action could be “make ten steps to the north followed by two steps to the west”. Macro-actions represent the problem at different levels of abstraction. It has been shown that given the right set of macro-actions, a reinforcement learning agent can increase its learning rate drastically [Precup, 2000].

A number of researchers have investigated methods for learning macro-actions across several tasks by identifying subgoals. Finding subgoals involves using an agent’s experience to discover states which, if the agent is able to navigate to them, speeds-up the learning process. Subgoals are generally considered to be “bottleneck” states in the environment. A common example of such a state is a passageway between rooms in a grid-world [McGovern, 2002]. Digne [1998] proposed a hierarchical method in which states that are visited frequently or have a high reward gradient become subgoals. Stolle and Precup [2002] defined “bottleneck” states as states that are frequently visited on system trajectories. MacGovern [2002] added the success condition to the frequency measure (states serve as potential subgoals if they are frequently visited on successful path but are not visited on unsuccessful ones). Menache et al. [2002] defined subgoals as states that separate well the initial and target states. Mannor et al. [2004] deviated from this approach by considering clusters of states rather than unique states.


In this paper, we define a method using local state features to create macro-actions. We first allow a reinforcement learning agent to learn the optimal policy for one or more related tasks. Then, we use local state features to induce a decision tree which represents the macro-action. The leaves of this tree probabilistically propose to execute an action (primitive or macro), or to stop the current macro-action. A local state feature in a grid-world environment, where each object in the grid is described by its coordinates, could be the direction of a given object relatively to the agent. In the grid-world, there can be a lot of positions where a cross, for example, is in the north-east direction with respect to the agent. Therefore, local state features have the potential to represent state generalization, and to be applicable in more situations than the methods reviewed so far for macro-actions induction. Our method is related to the work of Thrun and Schulte [1998], Singer and Veloso [1999] and Madden and Howley [2004], even if they do not create macro-actions. In [Thrun and Schulte, 1998], a decision tree is constructed based on the policies learnt on previous tasks to bias the initial value.
of an action in the new environment. In [Madden and Howley, 2004], different symbolic learners are tested to learn this exploration bias. In [Singer and Veloso, 1999], neural networks are trained with the learnt policies, and the outputs of the neural networks are combined with the actions values in the new environment. One drawback of the previous methods is that it is difficult to set the appropriate value for the exploration bias. Indeed, when a bias is used to initialize the values of the actions as in [Thrun and Schultz, 1998; Madden and Howley, 2004], it is not easy to know how long this bias will be effective, whereas it is very easy to fix the mean duration of a macro-action as we will see in section 3. Moreover, because macro-actions are treated as the primitive actions, they can always be selected (according to their learnt values). When a bias is combined with the value of an action, as in [Singer and Veloso, 1999], we can lose the solution of a given problem, depending on this bias value. When using macro-actions, because they will be added to the primitive actions, we are sure to find the optimal solution [Precup, 2000].

At last, macro-actions can be used to induce other macro-actions. The paper is organized as follows: in section 2, we briefly review reinforcement learning. We present our method in section 3: the decision tree learning algorithm, the criteria used to split a node and the way a policy is assigned to the leaves. Subsequent sections present empirical results on a complex grid-world task and conclusions.

2 Reinforcement learning

In the reinforcement learning framework, a learning agent interacts with an environment over a series \( t = 0, 1, 2, 3, \ldots \) of time steps. This environment is represented by a Markov decision process \((S, A, T, R)\) where:

- \( S \) is a set of states,
- \( A(s) \) is the set of available primitive actions in \( s \),
- \( T(s, a, s') \) is a transition function from \( S \times A \times S \) to \([0, 1] \),
- \( R(s, a, s') \) represents the probability of transition between states \( s \) and \( s' \) when action \( a \) is executed in \( s \),
- \( R(s, a, s') \) represents the mean reward received from transition between states \( s \) and \( s' \) when action \( a \) is executed in \( s \).

At each time step \( t \), the agent observes the environment state, \( s_t \in S \), and chooses an action, \( a_t \in A(s_t) \), which causes the environment to make a transition to state \( s_{t+1} \) with probability \( T(s_t, a_t, s_{t+1}) \) and to emit a reward \( r_{t+1} \). The objective of the agent is to learn to maximize the expected value of rewards received over time. It does this by learning a (possibly stochastic) mapping \( \pi \) from states to actions called a policy. More precisely, the objective is to choose each action \( a_t \) so as to maximize the expected return:

\[
E\left\{\sum_{i=0}^{\infty} \gamma^i r_{i+t+1}\right\}, \quad \text{where} \quad \gamma \in [0, 1] \text{ is a discount-rate parameter.}
\]

Many reinforcement learning algorithms estimate value functions which can be viewed as utilities of states and actions. An action-value function \( Q^\pi(s, a) \) mapping state-action pair to real value is defined as follows:

\[
Q^\pi(s, a) = E_{\pi}\left[r_{t+1} + \gamma r_{t+2} + \ldots | s_t = s, a_t = a\right].
\]

Intuitively, \( Q^\pi(s, a) \) is the amount of reward the agent expects to receive by executing action \( a \) starting in state \( s \) and then following the policy \( \pi \) forever. There exists at least an optimal policy \( \pi^* \) for which \( Q^{\pi^*}(s, a) = \max_{a'} Q^{\pi^*}(s, a) \) for all \( s \) and \( a \). Note that because each optimal policy has the same value function, \( Q^{\pi^*}(s, a) \) is noted \( Q^*(s, a) \).

Some algorithms have been designed to learn this optimal action-value function while interacting within an environment [Sutton and Barto, 1998]. The most used is the Q-learning [Watkins, 1989]. When the agent executes an action \( a_s \) in a state \( s_t \) and receives the reward \( r_{t+1} \) and the next state \( s_{t+1} \), this algorithm updates a Q-table according to the following rule:

\[
Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t (r_{t+1} + \gamma \max_{a' \in A(s_{t+1})} Q(s_{t+1}, a') - Q(s_t, a_t))
\]

where \( \alpha_t \) is a learning rate \((0, 1]) \) and \( \gamma \) the discount factor.

The difference between a macro-action and a primitive action is that the macro-action can last more than one time step. We now recall the generalization of the Q-learning algorithm [Watkins, 1989] to Macro-Q-learning [McGovern et al., 1997] (or learning with macro-actions). When a macro-action or a primitive action \( m \) is executed in a state \( s_t \), the agent updates a Q-table according to the following rule:

\[
Q(s_t, m) \leftarrow Q(s_t, m) + \alpha_t \left(\gamma^k \max_{m' \in A'} Q(s_{t+k}, m') - Q(s_t, m) + r_{t+1} + r_{t+2} + \ldots + \gamma^{k-1} r_{t+k}\right)
\]

where \( k \) is the actual duration of \( m \) (\( k = 1 \) if \( m \) is a primitive action), \( \alpha_t \) is the learning rate, \( \gamma \) the discount factor, and \( A'(s_{t+k}) \) is the set of macro-actions and primitive actions in \( s_{t+k} \). If only primitive actions are used, the Macro-Q-learning algorithm is equivalent to the Q-learning one.

3 Induction of macro-actions

To illustrate the concepts described in this section, we first present a variant of the Sokoban game, on which is tested our method in section 5. We put an agent (which has four available actions: up, right, down and left) in a grid-world where a ball, a goal and some walls are present. The aim of the agent is to push the ball into the goal (see Figure 3-b, where the agent, the ball, the goal and the walls are represented by a triangle, a filled circle, a cross and filled cells respectively). As the agent can only push the ball but not pull it, there are many situations in which the ball can become stuck or can have a limited set of cells in which it can be moved. Note that in the real Sokoban game, there can be more than one ball and goal location.

3.1 Local state features

In the Sokoban task, a state will be described by the coordinates of all entities in the grid-world. With this state representation, a reinforcement learning agent is sure to find, in theory, the optimal solution to any grid configuration. But, unfortunately, when confronted to a new one, the agent will have to learn from scratch and will not be able to use previously learnt knowledge. To have such a detailed representation of the environment is important to solve the task, but if we want to extract knowledge re-usable in other grid configurations, we need to consider local state features which are...
more independent of a specific grid configuration. For example, the direction (north, north-east, south, ... ) of the cross relatively to the agent is a local state feature. Note that one of these features not uniquely determine the exact state, but a set of states (e.g. the set of states where the cross is in the south direction with respect to the agent).

A test function is a function from \( S \) (the set of states) to a set of local state features. We give the set of test functions used for the Sokoban game in Table 1. If \((x, y)\) are the agent’s coordinates in the grid, the test function \(\text{ContentIn}(dx, dy)\) gives the content of the cell in position \((x + dx, y + dy)\). For example, \(\text{ContentIn}(0, +1)\), gives the content of the cell immediately to the north of the agent. There is such a test function for each of the 24 closest cells (for the Manhattan distance) around the agent.

### 3.2 Macro-actions representation

In our method, a macro-action is composed of two elements: a probability \(\eta \in [0, 1]\) and a tree \(T\). A macro-action is noted \(\text{macroAction}(\eta, T)\). The number \(\eta\) gives the probability to stop the macro-action at each time step. Indeed, even if the macro-action is good on the training set, it can encounter situations which are not present in this set. Therefore, a macro-action should have a non-null probability to stop. The tree is the decision part of the macro. In this tree:

- each node is a test function,
- each branch is labeled by a local state feature,
- a leaf is a stochastic policy.

An example of such a tree is represented in Figure 1. The root of this tree is labeled by the test function: \(\text{ContentIn}(0, +1)\). Each branch starting from this node is one of the local features among \{wall, empty, cross, ball\}. The leftmost leaf is a stochastic policy which proposes the primitive action right with probability 0.8, and to stop the macro-action with probability 0.2.

When the agent executes a macro-action in a given state \(s\) of the environment, with probability \(\eta\) it stops the macro-action. With probability \((1 - \eta)\) it evaluates the tree \(T\): the test function at the root (e.g. \(\text{ContentIn}(0, +1)\)) is applied to \(s\), and the branch corresponding to the local feature (e.g. wall) is followed. The same process is conducted, down the tree, up to reach a leaf. Then, the policy corresponding to the leaf is applied. If the action returned by the policy is primitive, the agent executes it in the environment. If it is a stop the macro-action is terminated, and if the action is a macro-action, the agent executes it just as described. A macro-action cannot be initiated, in a given state, if it leads with probability one to a stop. Indeed, this macro-action would be useless in this state. Moreover, the \(Q\)-value associated with this state and macro-action would keep its initial value and will potentially mislead the agent. By doing so, we are sure to find the optimal solution of a given task when we add one or more macro-actions to the primitive actions [Precup, 2000].

Note that the number \(\eta\) controls the mean duration of a macro-action. We will discuss about the influence of this number in section 5.

### 3.3 Induction

The principle of the macro-action induction process is first to let the agent solves one or more related tasks using the Macro-\(Q\)-learning algorithm. After the completion of those tasks, we know, for each state, which actions are optimal. Then, we construct a tree using the general top-down induction of decision tree principle [Quinlan, 1986]. Ideally, the tree should devide the state space into regions such that, in all states of a given region, the same action(s) is (are) optimal.

The macro-action creation algorithm is described in Algorithm 1. The parameters of this algorithm are the set of test functions, a set indicating if an action \(a\) is optimal in a given state \(s\), and the \(\eta\) parameter to probabilistically stop the macro-action. This algorithm creates an empty tree and completes it with the help of the \text{buildTree} procedure. Then, it returns the macro-action \(\text{macroAction}(\eta, T)\) where \(T\) is the completed tree.

At line 1 of the \text{buildTree} procedure, the development of the tree is stopped if the current set of test functions is empty or if the number of states in the current position of the tree is less than a given threshold (e.g. less than 5% of the number of states present at the root). Then, the \text{createPolicy} function is called to construct a policy for this leaf. If there is no state (\(E = 0\)), this function returns a policy that just prescribes to stop the macro with probability 1 . Else,

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CrossAgentDirection</td>
<td>Direction of the cross w.r.t. the agent</td>
<td>{N, S, E, W, NE, NW, SE, SW, here}</td>
</tr>
<tr>
<td>2 BallAgentDirection</td>
<td>Direction of the ball w.r.t. the agent</td>
<td>{N, S, E, W, NE, NW, SE, SW}</td>
</tr>
<tr>
<td>3 CrossBallDirection</td>
<td>Direction of the cross w.r.t. the ball</td>
<td>{N, S, E, W, NE, NW, SE, SW, here}</td>
</tr>
<tr>
<td>5 ContentIn(+1, 0)</td>
<td>Content of the (+1, 0) cell w.r.t. the agent</td>
<td>{wall, empty, cross, ball}</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>27 ContentIn(-1, +2)</td>
<td>Content of the (-1, +2) cell w.r.t. the agent</td>
<td>{wall, empty, cross, ball}</td>
</tr>
</tbody>
</table>

Table 1: Test functions and local state features used to describe environments in the experiments.

![Figure 1: An example of the decision part of a macro-action.](image-url)
the action $a^*$ which is optimal in the maximum number of states is found. The proportion of states where $a^*$ is optimal is noted $p^*$. At line (*)& of the createPolicy function, we test if the proportion $p^*$ is less than a given threshold $\theta$ of $[0, 1]$. The policy prescribed to stop the macro-action everywhere if $p^*$ is less than $\theta$. This parameter controls the degree of optimality of the actions proposed in a leaf of the tree. We will study the effect of this parameter in section 5. At line (**), we fix the probability to choose the action $a^*$ to $p^*$, and the probability to stop the macro-action to $(1 - p^*)$. To be more precise, at line (**), we incorporate in the policy all the actions that are optimal in $p^%$ of the states. For example, if we have $p_{\text{north}} = p_{\text{east}} = p^*$, we set $\pi(\text{north}) \leftarrow \frac{p^*}{2}$, $\pi(\text{east}) \leftarrow \frac{p^*}{2}$ and, as before, $\pi(\text{stop}) \leftarrow 1 - p^*$. When $p^* < 1$, because we know that the given action is not good in all states, we let the agent have the possibility to choose another action ($\pi(\text{stop}) > 0$) instead of the current macro.

From line 2 to the end of the buildTree procedure, the aim is to find (greedily) the best test function to develop the tree in its current position. At line 3, $T_{\text{test}} \leftarrow f_1$ means that the root of the local tree $T_i$ is the test function $f_1$. At line 4, we construct one branch of $T_i$ (the one corresponding to $\text{test } = 1$). Because the set of test functions passed to buildTree (line 4) is empty, each position $T_{\text{test}}$ will be a leaf. To evaluate the test function $f_1$, it just remains to evaluate $T_i$ (line 5). The evaluation of a tree proceeds as follows:

- if $l_1, \ldots, l_k$ are the leaves of the tree,
- $p_{l_1} \ldots, p_{l_k}$ the $p^*$ values (as computed by the createPolicy function) for each leaf,
- $n_1, \ldots, n_k$ the number of states dispatched to each leaf,
- then, the value of the tree is: $\sum_{i=1}^{k} p_{l_i} \times n_i$.

At last, at line 6, we continue the construction process with the best test function we have found at the current position.

Algorithm 1: Create macro-action

**Input:** $L$: set of test functions $E$: $\{(s, a) \in S \times A': a$ is optimal in $s\}$ $\eta$: a value in $[0, 1]$ 
**Output:** the induced macro-action $T$: empty tree $\text{buildTree}(T, L, E)$ 
\textbf{return} macroAction($\eta, T$)

4 Empirical results

We conduct experiments to:

- compare the performance of using a previously learnt macro-action with learning from scratch,
- test the influence of the $\eta$ and $\theta$ parameters.

In all the experimentations, the rewards are set to $-0.1$ for all actions except the actions which push the ball into the goal location. In this case, the reward is set to 100. The $\gamma$ parameter of the Macro-Q-learning algorithm is set to 0.9, and the

![Figure 2: The different grid configurations used to get experience on the Sokoban task.](sokoban_grid.png)
Table 2: Sample of rules extracted from the induced macro-action, based on the experience of the grid of Figure 2-a.

\[
\begin{align*}
\text{if BallAgentDirection is N and CrossBallDirection is N then } & [(1.0, up)] \\
\text{if BallAgentDirection is N and CrossBallDirection is S then } & [(0.5, left); (0.5, right)]
\end{align*}
\]

Table 3: Sample of rules extracted from the induced macro-action, based on the experience of the grids of Figure 2.

\[
\begin{align*}
\text{if BallAgentDirection is E and ContentIn(+1, 0) is wall then } & [(0.72, up); (0.28, stop)] \\
\text{if BallAgentDirection is SW and ContentIn(-1, 0) is empty and CrossBallDirection is NE and ContentIn(0, -1) is wall then } & [(0.97, left); (0.03, stop)]
\end{align*}
\]

Figure 3: Grids configurations used to test the induced macro-actions.

Figure 4: We compare the learning rate for Macro-\(Q\)-learning using, or not, the induced macro-action on the grid of Figure 3-a. The macro-action was induced based on the experience of the grid of Figure 2-a (with \(\theta = 1\)). We test different values of the \(\eta\) parameter.

In the second experimentation, we induce a macro-action based on the experience gained on the grids of Figure 2. We test different values of the \(\theta\) parameter. Parts of the rules are given in Table 3, for \(\theta = 0.7\). We test the induced macro-action on the configuration represented in Figure 3-b. We plot 6 curves: one represents Macro-\(Q\)-learning with only primitive actions, and the others, Macro-\(Q\)-learning with the induced macro-action and for different values of the \(\theta\) parameter. The parameter \(\eta\) is set to 0.01. We have not tried to optimize it, we have just taken the best value found in the last experimentation. When \(\theta\) is set to 1, using the induced macro-action is not better than not using it (the two curves are nearly the same). This is because the induced macro-action has only one leaf that proposes another choice than to stop the macro. The induced macro-action is better when we decrease the value of \(\eta\) up to the value 0.7. Then, the macro-action is getting worse for lower values. Note that macro-actions for values of the \(\eta\) parameter smaller than 0.7 are not so bad because, at this level, it does not remain a lot of leaves which propose only to stop the macro-action.
To be able to lower the $\theta$ parameter and to have at the same time better macro-actions is an interesting property. Indeed, during the induction process, it is more difficult to find an action which is optimal in all states of a given leaf than in 70% of them. At last, we have tested the macro-action induced in the first experimentation, when there was no wall, in the current grid configuration and for a value of $\eta$ set to 0.01. The results with this macro-action are worse than with only primitive actions.

![Figure 5](image-url)

**Figure 5:** We compare the learning rate for Macro-$Q$-learning using, or not, the induced macro-action on the grid of Figure 3-b. The macro-action was induced based on the experience of the grids of Figure 2. We test different values of the $\theta$ parameter to induce the macro-actions. The $\eta$ parameter is fixed to 0.01.

## 5 Conclusion

This paper has described a method to induce macro-actions based on local state features. Those local state features enable to abstract some details of the environment, and so have the potential to represent state generalization. Moreover, because the macro-action can stop itself, during the induction process we can choose actions that are not optimal in all states of a given leaf of the decision tree. We have tested our method on the Sokoban game and demonstrated its effectiveness. It seems difficult to apply the macro-actions induction methods reviewed in the introduction to this complex grid-world task. To improve the expressive power of our macro-actions, we could use logical decision trees [Blockeel and Raedt, 1998]. The nodes in logical decision trees are first order logic tests, rather than simple propositional logic tests as used in the decision part of our macro-actions. Finally, it is intended to test scenarios in which macro-actions are induced based on other macro-actions, and not only based on primitive actions. Like this, the previously learnt macro-actions can become building blocks for the newly induced ones.

## References


