A New Way to Introduce Knowledge into Reinforcement Learning

Pascal Garcia

INSA de Rennes/IRISA, F-35043 Rennes Cedex, France
pascal.garcia@irisa.fr

Abstract. We present in this paper a method to introduce *a priori* knowledge into reinforcement learning using temporally extended actions. The aim of our work is to reduce the learning time of the Q-learning algorithm. This introduction of initial knowledge is done by constraining the set of available actions in some states. But at the same time, we can formulate that if the agent is in some particular states (called exception states), we have to relax those constraints. We define a mechanism called the propagation mechanism to get out of blocked situations induced by the initial knowledge constraints. We give some formal properties of our method and test it on a complex grid-world task. On this task, we compare our method with Q-learning and show that the learning time is drastically reduced for a very simple initial knowledge which would not be sufficient, by itself, to solve the task without the definition of exception situations and the propagation mechanism.

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 punitions) in the long run [1]. A fundamental problem of its standard algorithms is that although many tasks can be formulated in this framework, in practice for 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 (e.g., [2,3]). The second approach is to use temporally extended actions (e.g., [4,5,6,7,8,9]). A temporally extended 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 temporally extended action could be "make ten steps to the north followed by two steps to the west". Temporally extended actions represent the problem at different levels of abstraction.

The aim of our work is to give a method to incorporate easily some *a priori* knowledge, about a task we try to solve by reinforcement learning, to speed-up the learning time. To introduce knowledge into reinforcement learning, we use some temporally extended actions for which the set of available actions can change during learning. We try to reduce the blind exploration of the agent by constraining the set of available actions. But because the *a priori* knowledge

N. Lavrač et al. (Eds.): ECML 2003, LNAI 2837, pp. 157–168, 2003.

could be very simple, those constraints can make the agent unable to solve a task. So we define a way to relax those constraints (with what we call the exception conditions and the propagation mechanism). The structure of this paper is as follows. First we described our method in section 2. We give its two main properties in section 3. In section 4 and 5 we describe a complex grid-world task to compare our method with Q-learning [10]. We show that the learning time is drastically reduced for a very simple initial knowledge which is not sufficient by itself, to solve the task and so, must be updated.

2 Formalism

In this section we develop our method which we call *EBRL* for Exception-Based Reinforcement Learning. To make it easier for the reader we explain it with the help of the artificial problem presented in Figure 1. In this grid-world, the agent has to reach the cross; he can move in eight directions (north, north-east, ...) and some walls can be put in the grid (the agent is blocked by them).

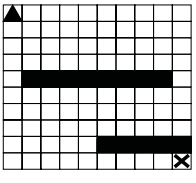


Fig. 1. The agent (triangle) has to reach the cross to solve the task.

2.1 Procedure, Rule and Exception

We define in this sub-section the syntax in which our temporally extended actions will be written. The semantic associated with this syntax is also explained. We represent a temporally extended action by a procedure:

`Procedure_name(state,list_of_parameters)` $\rightarrow$
`termination` : termination condition
`rule` : set of actions S_1
{`exception` : (exception condition, set of actions S_2)}$^{0/1}$
`next` : continuation

The semantic associated with this syntax is:

- `state`: the state of the underlying Markov Decision Process (*MDP*);
- `list_of_parameters`: optional parameters, each parameter has a finite number of different possible values;
- `termination`: this is the condition of termination of the procedure. This condition only depends on the state and the parameters;

- `rule`: this rule produces a finite set of primitive or temporally extended actions (procedures). This rule is applied only if the exception condition is not fulfilled;
- `exception`:
  - exception condition: if this condition is fulfilled, we do not use the set of actions of the rule part, but instead, the set of actions produced by the exception part;
  - set of actions: a finite set of primitive or temporally extended actions (procedures). This rule is applied only if the exception condition is fulfilled. We have $S_1 \subseteq S_2$;
- `next`: continuation after the execution of an action of the rule or exception part. This part is a call to another procedure. If this procedure has parameters, they only depend on the state and the parameters of the current procedure.

When entering a procedure, we first test the termination condition; if it is not fulfilled, we test the exception condition; if this condition is true, we choose one of the exception actions and after its execution, we continue in the `next` part. If the exception condition is not fulfilled, we choose one of the action of the `rule` part and after its carrying out, we continue in the `next` part. In the remaining of the article, we call *program* a finite set of procedures and *main* the first procedure of the program to be executed.

2.2 Example

We illustrate, in this section, the syntax described above, to solve the artificial problem.

```
main(grid configuration) ⟶
termination : the agent is on the cross
rule        : { the set of actions which make the agent get closer to the cross }
exception   : (all the actions of the rule part lead to a wall, { all the primitive actions } )
next        : main()
```

The *a priori* knowledge is just to choose in each state of the underlying *MDP* the set of actions (amongst the eight possible ones) which gets the agent closer to the cross without taking the walls into account (there is between one and three such actions). The exception to this rule is when all those actions lead the agent to a wall; when this is the case, we relax the constraints and allow all actions.

2.3 Full State Representation

We use a program in interaction with an *MDP*. This program will help the learning agent to solve the problem represented by this *MDP*. We have seen that each procedure has the state of the underlying *MDP* as a parameter. We define the full state representation of a procedure as a 4-tuple `(procedure_name,state, list_of_parameters, next_list)` where `procedure_name` is the name of the

procedure, `state` is the state of the underlying *MDP* when we enter this procedure, `list_of_parameters` is the possible parameters of the procedure and `next_list` is the list of procedures to be executed after the execution of `procedure_name`.

2.4 Induced Semi-Markov Decision Process

In this section we describe an algorithm called `construct-SMDP` which constructs a Semi-Markov Decision Process (*SMDP*) (see [11] and [5]), from a program $\mathcal{P}$ and an underlying *MDP* $\mathcal{M}$ (similar construction can be found in [4] and [5]). Note that this algorithm serves to demonstrate that the execution of a program on an *MDP* is an *SMDP*. We will never have to construct explicitly this *SMDP* when executing a program on an *MDP*. In the remaining of this article we note:

- $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$ the underlying *MDP* where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{T}$ is a Markovian transition model mapping $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ into probabilities in $[0, 1]$, $\mathcal{R}$ is a reward function mapping $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ into real-valued rewards;
- $\mathcal{P}$ the program;
- $Main\_parameters$ the set of all possible lists of values for the parameter list of the main procedure;
- $\mathcal{A}'$ the set of all the actions of the rule and exception part of all the procedures of the program and, for the temporally extended actions, all possible instantiations of their parameters;
- $\mathcal{A}'(s')$, where $s' = (p, s, l, n)$ is a full state representation, is the set of all actions of the rule and exception part of the procedure p. If the terminal condition of p is fulfilled, $\mathcal{A}'(s') =$ `termination`;
- ${\mathcal{A}'}_r(s')$, where $s' = (p, s, l, n)$ is the set of all actions of the rule part of the procedure p;
- ${\mathcal{A}'}_e(s')$ where $s' = (p, s, l, n)$ is the set of all actions of the exception part of the procedure p, if there is no exception part, ${\mathcal{A}'}_e(s') = {\mathcal{A}'}_r(s')$. We recall that ${\mathcal{A}'}_r(s') \subseteq {\mathcal{A}'}_e(s')$;
- `next`(p, s, l), where p is a procedure, s a state of $\mathcal{M}$ and l a list of parameters, is the procedure of the next part of p, with its instantiated parameters.
- `add`(e, l) returns the list with first element e and tail l. head(l) returns the first element of l and tail(l) returns the list l without its first element.

A state of the constructed *SMDP* is a full state representation of a procedure. The *SMDP* $(\mathcal{S}', \mathcal{A}', \mathcal{T}', \mathcal{R}', \beta)$ (where $\mathcal{S}'$, $\mathcal{A}'$, $\mathcal{T}'$, $\mathcal{R}'$ have the same meaning as $\mathcal{S}, \mathcal{A}, \mathcal{T}$ and $\mathcal{R}$ respectively and β is a mapping from $\mathcal{S}' \times \mathcal{A}' \times \mathcal{S}'$ into real value -see [5]-) is constructed as follows:

```
algorithm construct−SMDP(MDP : M, program : P,
discount factor : γ ∈]0, 1[)
begin
S' ← {main} × S × Main_parameters × {[]}
repeat
        forall t = (p, s, l, n) ∈ S', forall a ∈ A'(t) do
                if a = p'(l') then
                        t' ← (p', s, l', add(next(p, s, l), n))
                        S' ← S' ∪ {t'}
                        T'(t, a, t') ← 1
                        β(t, a, t') ← 1
                elseif a = termination and p ≠ main then
                        t' ← next−state(s,n)
                        S' ← S' ∪ {t'}
                        T'(t, a, t') ← 1
                        β(t, a, t') ← 1
                elseif a ∈ A then
                        forall s' ∈ S do
                                n' ← add(next(p, s, l), n)
                                t' ← next−state(s',n')
                                S' ← S' ∪ {t'}
                                T'(t, a, t') ← T(s, a, s')
                                R'(t, a, t') ← R(s, a, s')
                                β(t, a, t') ← γ
                        endforall
                endif
        endforall
until S' is stable (means that no new state
                    has been added to S')
All unspecified value for T', R' and β are set to 0
return (S', A', T', R', β)
end
```

```
algorithm next−state(s : s ∈ S, n : next_list)
begin
if n ≠ [] then
        let p(l) = head(n)
        t ← (p, s, l, tail(n))
        return t
else
        in this case, the program is not correct,
        the program is not terminated and there is
        no next procedure to call
endif
end
```

We only consider program $\mathcal{P}$ and *MDP* $\mathcal{M}$ for which the `construct-SMDP` algorithm terminates. The `construct-SMDP` satisfies the definition of an *SMDP*. The Markov property is preserved because of the full state representation. This *SMDP* is what the agent faces when executing $\mathcal{P}$ in $\mathcal{M}$. Note that we do not discount by 1 when calling a temporally extended action (procedure) and the immediate reward is zero. Moreover, we discount by γ and receive the immediate reward of the underlying *MDP* when executing a primitive action so, the solution of the constructed *SMDP* defines an optimal policy that maximizes the expected discounted sum of rewards (with discount factor γ) received by the agent executing $\mathcal{P}$ in $\mathcal{M}$. Note that the optimal policy in the *SMDP* can be sub-optimal in the underlying *MDP*.

Proposition 1 *The* `construct-SMDP` *algorithm terminates if the* MDP $\mathcal{M}$ *is finite and the* `next_list` *of all possible full state representation is finite.*

Proof: By definition of a procedure, $\mathcal{A}'$ is finite and if $\mathcal{M}$ is finite and the `next_list` is finite for all full state representation, there is only a finite number of possible full state representation so, after a finite number of the repeat loop, no more new state will be added to $\mathcal{S}'$. □

Definition 1 *A procedure p is said to create next_list cycle if and only if p can be reached by a temporally extended action of its rule or exception part.*

Proposition 2 *A sufficient condition to insure finite* `next_list` *is that there does not exist next_list cycle procedures.*

Proof: The `next_list` grows only when we choose in a procedure p a temporally extended action. If we do not have next_list cycle procedures, in the next part of p we cannot call a procedure already in `next_list` (else a temporally extended action of p has reached p) and because the program is composed of a finite number of procedures, the `next_list` cannot grow indefinitly. □

Fig. 2. A wall is located between the agent and the cross.

Fig. 3. Without propagation mechanism, the agent cannot escape of the dead-end.

2.5 Exception State and Propagation

We call direct exception state a state for which the exception condition is fulfilled. For example for the artificial problem and the program described in section 2.2, the state `(main,(3,3),[],[])` (we represent the grid configuration only by the agent's position, see Figure 2 (a)) is said to be a direct exception state because the rule part prescribes to go in a wall and so the exception condition is fulfilled. We associate with a program and an underlying *MDP* a table $\mathcal{E}$ from full state representation to boolean. Let $s' = (p, s, l, n) \in \mathcal{S}'$. If $\mathcal{E}(s') = false$ then, in s', we only use the action of the rule part of procedure p. If $\mathcal{E}(s') = true$ then, in s', we only use the action of the exception part of procedure p. We call a state s for which $\mathcal{E}(s) = true$ an exception state. Initially all entries of this table are set to false. In the above exemple, when the agent, executing the program in the underlying *MDP*, encounters the state `(main,(3,3),[],[])`, we set $\mathcal{E}$(`(main,(3,3),[],[])`) to true and now, in this state, the agent will rely only on the actions of the exception part.

After few iterations of the program we present in Figure 2 (b), in shaded cells, the exception states (a shaded cell with coordinates (x, y) as to be interpreted as a full state representation `(main,`(x, y)`,[],[])`. In this case, it is sufficient to solve the task.

But this mechanism is very limited, for example, Figure 3 (a), the agent cannot escape of the dead-end because only states `(main,`$(6, 4)$`,[],[])`, `(main,`$(6, 5)$`,[],[])` and `(main,`$(6, 6)$`,[],[])` of the induced *SMDP* will become exception states. We need a way to propagate this information back to the predecessor states. This way of propagating exception states is called the *propagation mechanism*.

Definition 2 *We say that* $s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n$ *is a rule part action path from* s_1 *to* s_n *in* $\mathcal{M}'$ *iff for all* s_i *where* $1 \leq i < n$*, there exists an action* $a \in \mathcal{A'}_r(s_i)$ *for which* $\mathcal{T}'(s_i, a, s_{i+1}) > 0$*.*

Definition 3 *We denote by* $s \xrightarrow{a} s'$ *that the action* a *has led to state* s' *starting from state* s*.*

We now define the property that must fulfill a propagation mechanism.

Definition 4 *Let* s *be a state of the induced* SMDP *where* $\mathcal{E}(s) = false$*. If for each action* a *of* $\mathcal{A'}_r(s)$*, there exists a rule part action path* $s \xrightarrow{a} \cdots \rightarrow s_n$ *where* $\mathcal{E}(s_n) = true$ *then, the propagation mechanism insures that after a finite number of time steps,* $\mathcal{E}(s)$ *will become true.*

For example, with any propagation mechanism, the program of the section 2.2, will now solve the dead-end example (see the properties section). An example of exception states with a propagation mechanism is given Figure 3 (b) for the dead-end example and after few iterations of the program.

Those propagation mechanisms can be designed by the user of our method but, we give here, a very simple and general propagation mechanism we will use in the result section. We call it the basic propagation mechanism:

Definition 5 *Let* s_1 *and* s_2 *be two states of* $\mathcal{S}'$ *where* $\mathcal{E}(s_1) = false$ *and* $\mathcal{E}(s_2) = true$*. If the agent makes a transition between* s_1 *and* s_2 *then,* $\mathcal{E}(s_1)$ *is set to true.*

The basic propagation mechanism fulfills the propagation mechanism property (note that the basic propagation propagates more than needed by the definition 4). Note that the basic propagation mechanism is cheap to compute, when executing an action a in s and ending up in state s', for example, we just have to look at the value of $\mathcal{E}(s')$ to know if we have to propagate. The size of the table $\mathcal{E}$ is always less than the size of the Q-table and each entry is just a boolean.

2.6 Learning

For a learning agent interacting with a Semi-Markov Descision Process, there exists a learning algorithm, called SMDP Q-learning which updates a state-action value function Q - which maps state-action pairs into real values (where actions can be primitive or temporally extended) - at every time period with the formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \beta_t \max_{a \in \mathcal{A}'(s_{t+1})} Q(s_{t+1}, a) - Q(s_t, a_t))$$

where a_t is the action (primitive or temporally extended) taken by the agent in s_t (a state of the *SMDP*) , s_{t+1} is the new state after executing a_t in s_t, r_t is the reward and β_t the discount factor received by the agent and α is the learning rate. This learnt Q-function converges to the optimal Q-function under technical conditions similar to those for conventional Q-learning (see [5]).

3 Properties

We give in this section two properties of the *EBRL* method. We illustrate them with our artificial problem. We suppose in this section that every action gets executed in every state infinitely often.

Definition 6 *We say that $s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n$ is an exception part action path from s_1 to s_n in $\mathcal{M}'$ iff for all s_i, s_{i+1} where $1 \leq i < n$, there exists an action $a \in \mathcal{A'}_e(s_i)$ for which $\mathcal{T}'(s_i, a, s_{i+1}) > 0$.*

Definition 7 *For a procedure p, we note $\mathcal{S'}_p \subseteq \mathcal{S}'$ the set of states of the form (p, s, l, n).*

Definition 8 *For a procedure p, we note $\mathcal{B}_p \subseteq \mathcal{S'}_p$ the set of states of the form (p, s, l, n) for which there exists an exception part action path in $\mathcal{M}'$ leading to a state s' for which the termination condition of the procedure p is fulfilled.*

Theorem 1 *For a procedure p, if for each $s \in \mathcal{S'}_p$, for each action $a \in \mathcal{A'}_r(s)$, there exists a rule part action path $s \xrightarrow{a} \cdots \rightarrow s'$ leading to a state s' for which, either the termination condition of the procedure p is fulfilled or, s' is a direct exception state then, for each state of $\mathcal{B}_p$, the agent executing the procedure p can reach a state in $\mathcal{S'}_p$ for which the terminal condition of the procedure p is fulfilled.*

Proof:

a) For all $s \in \mathcal{S'}_p$, by hypothesis,
- Either there exists, for an action $a \in \mathcal{A'}_r(s)$, a rule part action path $s \xrightarrow{a} \cdots \rightarrow s'$ for which s' fulfills the termination condition of p. As $\mathcal{A'}_r(s) \subseteq \mathcal{A'}_e(s)$ using either set of actions ($\mathcal{A'}_r(s)$ or $\mathcal{A'}_e(s)$, depending of the value of $\mathcal{E}(s)$) we can reach the terminal condition of p.
- Or, for all $a \in \mathcal{A'}_r(s)$ there exists a rule part action path $s \xrightarrow{a} \cdots \rightarrow s'$ for which $\mathcal{E}(s') = true$ and so, by definition of the propagation mechanism, $\mathcal{E}(s)$ will become true after a finite number of iterations.

b) Let $s_1 \in \mathcal{B}_p$ then, by definition, there exists an exception part action path $s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n$ where s_n is a state for which the termination condition of the procedure p is fulfilled. For all s_i in this path where $1 \leq i < n$, If $\mathcal{E}(s_i) = false$ then by a), the agent can reach a state s' from s_i for which the terminal condition of the procedure p is fulfilled or $\mathcal{E}(s_i)$ will become true after a finite number of time steps. But, if $\mathcal{E}(s_i) = true$ then, there exists an exception part action which leads to s_{i+1}. □

For example we can prove with this theorem that the program defined in section 2.2, with a propagation mechanism, will get the agent to the cross, if it is possible to go to it in the underlying *MDP* using all the primitive actions. This is true even if the actions are stochastics (p% of the time, the action is executed correctly and $(1-p)$% of the time a randomly chosen action is executed instead).

Definition 9 *We note* $\mathcal{E}(\mathcal{M}')$ *the* SMDP *obtained from* $\mathcal{M}'$ *in which for each state* $s \in \mathcal{S}'$ *if* $\mathcal{E}(s) = true$*, we only use the actions of* $\mathcal{A}'_e(s)$ *else, we only use the actions of* $\mathcal{A}'_r(s)$*.*

Proposition 3 *After a finite number of time steps,* $\mathcal{E}$ *does not change anymore. We then note the* $\mathcal{E}$ *table by* $\mathcal{E}_f$*.*

Proof: The table $\mathcal{E}$ has a finite number of entries and for a state s for which $\mathcal{E}(s) = false$ by the definition of the direct exception states and the propagation mechanism, either $\mathcal{E}(s)$ remains $false$ or after a finite number of time steps $\mathcal{E}(s)$ becomes *true*. □

Theorem 2 *SMDP Q-learning, applied to an agent executing* $\mathcal{P}$ *in* $\mathcal{M}$ *with exception table* $\mathcal{E}$*, will converge to the optimal policy in* $\mathcal{E}_f(\texttt{construct-SMDP}(\mathcal{M}, \mathcal{P}, \gamma))$ *w.p.1. if* $\sum \alpha = \infty$ *and* $\sum \alpha^2 < \infty$*.*

Proof: $\mathcal{E}_f(\texttt{construct-SMDP}(\mathcal{M}, \mathcal{P}, \gamma))$ is a finite *SMDP* fulfilling the preconditions of the theorem 2 of Parr, R. [5]. □

This theorem tells us that we will obtain the best policy in the *SMDP* obtained by the agent executing the program in the underlying *MDP*. This policy could be non optimal in the underlying *MDP*. Note that we can increase the search space and possibly the quality of the solution by relaxing the constraints in the states for which $\mathcal{E}(s) = false$. In doing so, for the problem of section 2.2 and for the problem of the following section, we are guaranted to find the optimal solution in the underlying *MDP*.

4 Example

We will use a task very similar to the Sokoban game to illustrate our method because of its complexity.
We put an agent (who can move in 8 directions: north, north-east, east, ...) in a grid. A ball, a goal and walls are placed in the grid. The aim of the agent is to push the ball into the goal (see Figure 5 (a), where the agent, the ball, the goal and the walls are represented by a triangle, a filled circle, a cross and filled cells respectively). We assume the agent knows the ball and goal location. 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. The actions are stochastics: 90% of the time the action is executed correctly and 10% of the time another randomly chosen action is executed.

4.1 Task Program

We write in our formalism a program to help the agent solve a given grid configuration, this program is described in Figure 4.

The program is broken-down into two sub-tasks: firstly, go to the ball and secondly push the ball to the goal location. The {go to ball actions} set of the

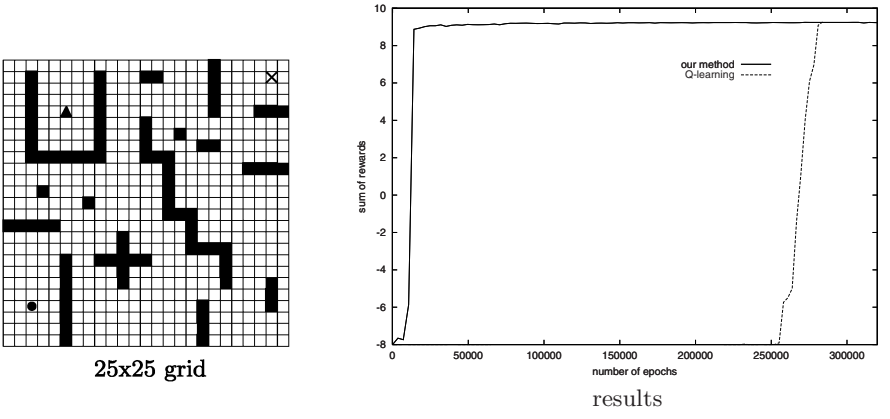
```
main(state) ⟶
termination : the ball is in the goal location
rule        : Go_to_ball()
next        : Go_to_goal()

go_to_ball(state) ⟶
termination : the agent is next to the ball
rule        : {go to ball actions}
exception   : (all the actions in {go to ball actions} prescribe
              to go into a wall,{all the primitive actions})
next        : go_to_ball()

go_to_goal(state) ⟶
termination : the ball is in the goal location
rule        : if the agent is next to the ball then
              {go to goal actions} else go_to_ball()
exception   : (all the actions in {go to goal actions} prescribe
              to push the ball in a wall or the cell where the
              agent has to go to push the ball in this
              direction is a wall,
              {push_north([ball_location]),
              push_north_east([ball_location]), ...})
next        : go_to_goal()

push_north(state,[last_ball_location]) ⟶
termination : ball has been pushed
rule        : let (x, y) the cell to go, to push to
              the north.
              if can go to (x, y) then
              go_to([last_ball_location;x;y]) else
              learn_to_go_to([x;y])
next        : push([last_ball_location])

go_to(state,[last_ball_location;x;y]) ⟶
termination : ball has been pushed or agent on (x, y)
rule        : {go to actions}
next        : go_to([last_ball_location;x;y])

learn_to_go_to(state,[last_ball_location;x;y]) ⟶
termination : ball has been pushed or agent on (x, y)
rule        : {learn to go to actions}
exception   : (all the actions in {learn to go to actions}
              prescribe to go into a wall,
              {all the primitive actions})
next        : learn_to_go_to([last_ball_location;x;y])
```

Fig. 4. Program to help the agent to solve the task, see the text for details about the rule parts.

`go_to_ball` procedure is the set of actions which make the agent get closer to the ball without taking the walls into account (there is between one and three such actions for a given state). The {go to goal actions} set of the `go_to_goal` procedure is the set of procedures amongst `push_north`, `push_north_east`, ... which would make the ball get closer to the goal if there are no walls (there is between one and three such procedures). Note that in the rule part of the `go_to_goal` procedure, we test if the agent is next to the ball, this is because the actions could be stochastics. We only write the code for `push_north` because `push_north_east`, ..., are similar. In `push_north`, we test if we can go to the (x, y) cell. We just look at the 8 cells around the ball and test if there is a path using only those 8 cells; the {go to actions} set contains the action to take to go to (x, y) using those 8 cells when there is a path using only those 8 cells. The {learn to go to actions} set of the `learn_to_go_to` procedure is the set of actions which make the agent get closer to the ball without taking the walls into account. The `push` procedure terminates if the agent is not next to the ball and else pushes the ball. We can notice that the program gives explicitly only a very high level knowledge. It does not know how to avoid obstacles.

5 Results

In this section, we test our method on the grid configuration, presented in Figure 5 (a), which is quite difficult for the *a priori* knowledge given by the program. If we use the program without propagation mechanism, this task could not be solved. An epoch consists of 800 primitive actions, -0.01 reinforcement is given on each transition in the grid except when the ball is pushed into the goal location where a reinforcement of 10 is given. We use an ϵ-greedy policy of parameter 0.9, the discount factor γ is 0.999 and the learning rate α is 0.1. We plot two curves (Figure 5 (b)), one for the Q-learning algorithm and one for our method. Each 10 epochs we set the policy to the greedy one and plot the results which are averaged over ten runs and smoothed. The mean first time the greedy policy



results

Fig. 5. Comparaison between Q-learning and our method.

solves the task is after 458 epochs with our method and after 248621 epochs for the Q-learning one. The number of states memorized at the end of this experimentation is 138214 for our method and 248948 for the Q-learning one. Note that the basic propagation is quite expansive in the number of memorized states but, explores a larger state space and so can find a better solution. Using various grid configurations, we noted that the learning time with our method depends more on the difficulty of the grid compare to the initial knowledge than on the state space size. Moreover, we also came to the conclusion that the larger the state space the better our method compared to Q-learning.

6 Conclusion

We have presented in this paper a method to introduce knowledge into reinforcement learning to speed-up learning using temporally extended actions. Our work can be related with the Parti-game algorithm of Moore [12] where a greedy controler helps the agent to reach a goal state. To deal with getting trapped, the Parti-game algorithm divides more and more thinly the state space to circumvent becoming trapped. Our method do not divide the state space, the resolution of the state space is given but the number of choices to be made in each state is variable. With variable resolution we can potentially store less states but note that if the *a priori* knowledge allows only one action in a given state, this state does not have to be stored (no choice has to be made in this state). We can formulate in our method that more than one action seems good *a priori* and so test different potentially good paths to the goal before increasing the search space. Moreover, we do not assume that the dynamic of the environment is deterministic, we can learn arbitrary reward functions and we do not need to learn a greedy controller. One of the main drawback of constraining the number of available actions is that it can be difficult to guarantee that with this *a priori* knowledge the agent can still solve the task. In this paper, we formulate and prove a theorem which can be used to guarantee that a given task can still be

solved with our method. We tested our method on a complex grid-world task and showed that the learning time is drastically reduced compared to Q-learning. We currently test our method in a continuous state space.

References

1. Sutton, R.S. & Barto, A.G. (1998). *Introduction to Reinforcement Learning*. MIT Press/Bradford Books.
2. Singh, S.P., Jaakkola, T. & Jordan, M.I. (1995). *Reinforcement Learning with Soft State Aggregation* (pp. 361–368). NIPS 7. The MIT Press.
3. Tsitsiklis, J.N & Van Roy, B. (1997). *An analysis of temporal-difference learning with function approximation* (42(5):674–690). IEEE Transactions on Automatic Control.
4. Sutton, R.S., Precup, D. & Singh, S. (1999). *Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning* (112:181-211). Artificial Intelligence.
5. Parr, R. (1998). *Hierarchical control and learning for Markov decision processes*. PhD thesis, University of California, Berkeley, California.
6. Dietterich, T.G. (2000). *An Overview of MAXQ Hierarchical Reinforcement Learning* (pp. 26–44). SARA.
7. Randlov, J. (1999). *Learning Macro-Actions in Reinforcement Learning*. NIPS 11, MIT Press.
8. Stone, P. & Sutton, R.S. (2001). *Scaling Reinforcement Learning Toward RoboCup Soccer*. Proceedings of the 18th International Conference on Machine Learning.
9. Menache, I., Mannor, S. & Shimki, N. (2002). *Q-Cut - Dynamic Discovery of Sub-goals in Reinforcement Learning* (pp. 295–306). European Conference on Machine Learning. LNAI 2430.
10. Watkins, C.J.C.H. (1989). *Learning from Delayed Rewards*. PhD Thesis. University of Cambridge, England.
11. Puterman, M.L. (1994). *Markov Decision Processes*. Wiley, New York.
12. Moore, A.W. & Atkeson, C.G. (1995). *The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces*. Advances in Neural Information Processing Systems.