Chapter 9
Relational Representations and Traces for Efficient Reinforcement Learning

Eduardo F. Morales
National Institute of Astrophysics, Optics and Electronics, México

Julio H. Zaragoza
National Institute of Astrophysics, Optics and Electronics, México

ABSTRACT

This chapter introduces an approach for reinforcement learning based on a relational representation that: (i) can be applied over large search spaces, (ii) can incorporate domain knowledge, and (iii) can use previously learned policies on different, but similar, problems. The underlying idea is to represent states as sets of first order relations, actions in terms of those relations, and to learn policies over such generalized representation. It is shown how this representation can produce powerful abstractions and that policies learned over this generalized representation can be directly applied, without any further learning, to other problems that can be characterized by the same set of relations. To accelerate the learning process, we present an extension where traces of the tasks to be learned are provided by the user. These traces are used to select only a small subset of possible actions increasing the convergence of the learning algorithms. The effectiveness of the approach is tested on a flight simulator and on a mobile robot.

INTRODUCTION

Sequential decision making under uncertainty has been studied in fields such as decision-theoretic planning (Puterman, 1994), reinforcement learning (Sutton & Barto, 1989) and economics. The idea is to decide on each state, which is the best action to perform, given uncertainty on the outcomes of the actions and trying to obtain the maximum benefit in the long run. Optimal sequence decision making can be formalized as a Markov decision process or MDP, where several dynamic programming techniques have been developed (Puterman, 1994). These techniques, obtain optimal policies, i.e., the best action to perform

DOI: 10.4018/978-1-60960-165-2.ch009

on each state, however, they require knowing a transition model. Reinforcement learning (RL), on the other hand, can learn optimal or near optimal policies while interacting with an external environment, without a transition model (Sutton & Barto, 1989). In either case, the representation used in traditional models for MDPs require all the possible states to be explicitly represented. This restricts its applicability to only very simple domains as the number of possible states grows exponentially with the number of relevant variables. In order to cope with this curse of dimensionality several approaches have been suggested in recent years. Some of these methods include, function approximation (e.g., (Chapman & Kaelbling, 1991)), hierarchical (e.g., (Dietterich, 2000)) and temporal abstraction (e.g., (Sutton, Precup, & Singh, 1999a)), and factored representations (e.g., (Kaelbling, Littman, & Cassandra, 1998)). Despite recent advances, there is still on-going research into trying to deal more effectively with large search spaces, to incorporate domain knowledge, and to transfer previously learned policies to other related problems. Most work, however, uses a propositional framework and still do not scale well as many domains are more clearly defined in terms of objects and relations.

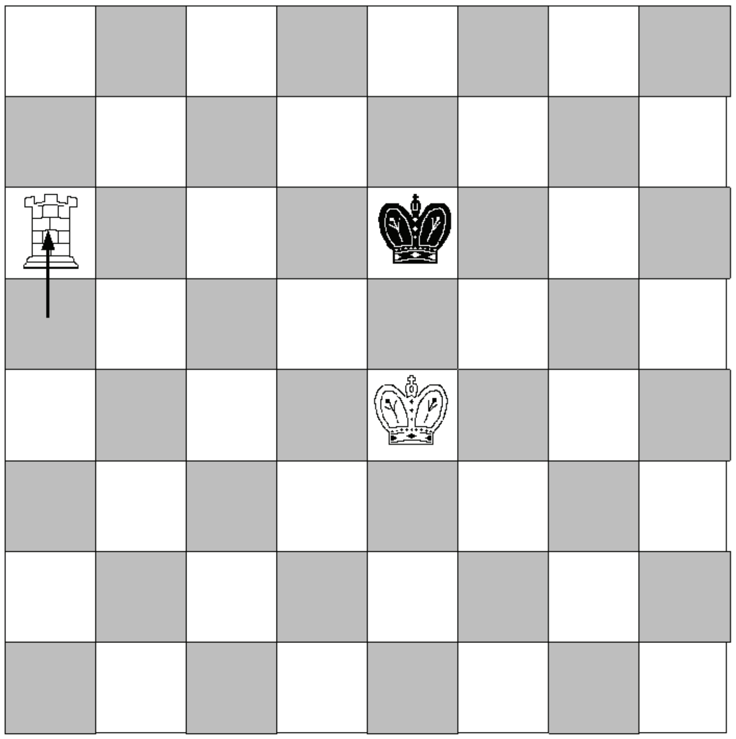
Suppose we want to learn a policy to play a simple chess endgame from a particular side. Even for learning how to win in a simple and deterministic endgame like king-rook vs. king (KRK), there are more than 175,000 not-in-check legal positions. The number of possible actions for the king-rook side is in general 22 (8 for the king and 14 for the rook), which sum up to nearly 4 million possible state-action pairs. Even with modern computers, learning directly in this representation is just too slow. In this domain, however, there are many states that are essentially the same, in the sense that they all share the same set of relations. For a chess player, the exact location of each piece is not as important as the relations that hold between the pieces to decide which movement to perform (see also (Charness, 1977, Groot, 1965)). For instance, in the KRK domain, whenever the two kings are *in_opposition* (in the same rank or file with one square in between), the rook is in the file/rank that divides both kings, and the rook is at least two squares apart from the opposite king (i.e., it is safe to check), a good move is to check the opposite king to force it to move towards a border (see Figure 1). This action is applicable to all the chess board positions where the previously mentioned relations hold between the pieces, and already captures more than 1,000 chess positions with a white rook. In fact, it is applicable to chess boards of different sizes.

In this chapter, a relational representation for reinforcement learning is used to represent a small set of abstract states with a set of properties (e.g., *in_opposition, rook_divides*, etc.), represent actions in terms of such properties (e.g., If *in_opposition* And *rook_divides* Then *check*) and learn which action to apply on each state.

There has been a growing interest in using first-order logic for modeling MDPs (e.g., (Driessens & Ramon, 2003, Dzeroski, Raedt, & Driessens, 2001, Fern, Yoon, & Givan, 2003, Morales, 2003, Otterlo, 2009)). The use of a relational representation has many advantages. In particular, logical expressions may contain variables and consequently make abstractions of many specific grounded states or transitions. This considerably reduces the number of states and actions and simplifies the learning process that can be carried out at the abstract level. Furthermore, it is possible to transfer policies to other instances and to other, although similar, domains.

Since the seminal work by Dzeroski et al. (Dzeroski et al., 2001, Driessens, Ramon, & Blockeel, 2001, Driessens & Dzeroski, 2002) an increasing number of systems have been proposed in the literature. In particular, at the same time and independently, three related approaches defined abstract states and actions in terms of relations (Kersting & Raedt, 2003, Morales, 2003, Otterlo, 2003). In this chapter, an extended and more formal description of the approach described in

Figure 1. Forcing the opponent king to move towards a border



(Morales, 2003) is given including other tests and results. A more complete description of recent relational reinforcement learning algorithms is given in (Otterlo, 2009).

This chapter also shows that we can significantly reduce the convergence time of the learning process by using traces, provided by the user, of the task we want the agent to learn. Finally, it is shown how to transform a discrete actions policy into a continuous actions policy using Locally Weighted Regression.

The objectives of this chapter are to formalize the use of relational representations in reinforcement learning, show their benefits, describe an approach to produce faster convergence times and show the capabilities on two challenging domains; a flight simulator and a mobile robot.

This chapter is organized as follows. It first introduces the basic setting and standard notation. An overview of the most relevant related work is given next. It then describes in detail the proposed relational representation and how to apply a reinforcement learning algorithm over this representation. It then shows how to learn a subset of relational actions from traces provided by the user to speed-up the learning process. This chapter provides experimental evidence of the proposed approach in a flight simulator and a robotics domain. Conclusions and suggests future research directions are given at the end of the chapter.

Preliminaries

Logic

A first-order alphabet Σ is a set of predicate symbols p with arity $m \geq 0$ and a set of function symbols f with arity $n \geq 0$. If $m = 0$ then p is called a proposition and if $n = 0$, f is called a constant. An atom $p(t_1,\ldots,t_m)$ is a predicate symbol p followed by a bracketed m-tuple of terms t_i. A term is a variable or a constant. A conjunction is a set of (possibly negated) atoms. A substitution τ is a set of assignments of terms t_i to variables V_i, $\Theta = \{V_1 / t_1, \ldots, V_l / t_l\}$. A term, atom or conjunction is called ground if it contains no variables. For more information, on Logic and Logic Programming see, for instance, (Lloyd, 1987).

Markov Decision Processes

A Markov Decision Process (MDP) is a tuple $M = <S,A,T,R>$, where S is a set of states, A is a set of actions, $A(s) \in A$ is a set of actions for each state $s \in S$, T is the transition function $T : S \times A \times S \to [0,1]$ and R is the reward function $R : S \times A \times S \to R$. A transition from state $s \in S$ to state $s' \in S$ caused by some action $a \in A(s)$ occurs with probability $P(s'|a,s)$ and receives a reward $R(s,a,s')$. A policy $\pi : S \to A$ for M specifies which action $a \in A(s)$ to execute when an agent is in some state $s \in S$, i.e., $\pi(s) = a$.

A solution for a given MDP $M = <S,A,T,R>$ consists of finding a policy that maximizes the long-time reward sequence. A deterministic policy $\pi : S \to A$ specifies which action $a \in A(s)$ to perform on each state $s \in S$. The policy is associated with a value function $V^{\pi} : S \to R$. For each state $s \in S$, $V^{\pi}(s)$ denotes the expected accumulated reward that will be obtained from state s and following the actions suggested by π. This can be expressed in a discounted infinite horizon by:

$$V^{\pi}(s) = E_{\pi}\left(\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_t = s\right)$$

Similarly, the action-value function for policy π, denoted by $Q^{\pi}(a,s)$ is defined as:

$$Q^{\pi}(s,a) = E_{\pi}\left(\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_t = s, a_t = a\right)$$

The expression for V can be recursively defined in terms of the Bellman Equation:

$$V^{\pi}(s) = \sum_{s' \in S} P(s \mid \pi(s), s')\Big(R(s) + \gamma V^{\pi}(s')\Big).$$

For more information on Markov Decision Process and Reinforcement Learning see (Puterman, 1994, Sutton & Barto, 1989).

Several approaches have been suggested in the literature to learn optimal policies and value functions in RL. In this chapter we used a modified Q-learning approach over an abstracted space based on a relational representation.

Related Work

Other researchers have also turned their attention towards abstracting the search space in different ways in order to tackle larger problems. State aggregation clusters "similar" states together and assigns them the same value, effectively reducing the state space. Work on tile-coding (e.g., (Lim & Kim, 1991)), coarse coding (e.g., (Hinton, 1984)), radial basis functions (e.g., (Poggio & Girosi, 1990)), Kanerva coding (e.g., (Kanerva, 1993)), and soft-state aggregation (e.g., (Singh, Jaakkola, & Jordan, 1996)) are some of the representatives of this approach. In this paper, we also do state aggregation, but we use a relational representation, grouping together states that share the same set of relations. This has several advantages over previous approaches: (i) it is easy to define useful and powerful abstractions, (ii) it is easy to incorporate domain knowledge, and (iii) the learned policies can be directly used to other similar domains without any further learning, which is not possible with previous approaches.

Relational Reinforcement Learning (RRL) (Džeroski et al., 2001, Driessens et al., 2001, Driessens & Džeroski, 2002) uses a relational representation for states and actions; however their main focus has been on approximating value functions with a relational representation. This extends previous work on incremental regression trees (e.g., (Chapman & Kaelbling, 1991)) to a relational representation. One disadvantage with an incremental tree approach is that once a split in the tree is decided it cannot be undone, which makes the approach highly dependent on the initial sequence of experiments. Recently, Croonenborghs *et al.* (Croonenborghs, Driessens, & Bruynooghe, 2007) extended this work and introduced a method to learn relational options for transfer learning in RL. They first learn a relational policy with the RRL approach presented in (Sutton, Precup, & Singh, 1999b) where the goal is to decompose and learn tasks as a set of options. An option can be viewed as a subroutine, consisting of an option policy that specifies which action to execute for a subset of the environment states. Besides learning these sets of options for the tasks they also generate examples or traces guided from the learned sets of options. They do this in order to provide some help or guidance to the person or method that generates the examples. These examples and the sets of options are given to *TILDE* (Blockeel & Raedt, 1998) to learn relational decision trees that allow the knowledge from the learned policies to be transferred between similar domains. Unfortunately the method was only applied to the blocks world and the actions from the sets of options are discrete.

The closest research work to the relational reinforcement learning approach described in this chapter, which is a clearer formalization of what was introduced in (Morales, 2003), was independently proposed by Kersting and De Raedt (Kersting & Raedt, 2003, 2004) and by van Otterlo (Otterlo, 2003, 2004). The three approaches introduced similar abstractions based on predicates. In fact, all three transform the underlying relational MDP into a much smaller abstract MDP that is solved using modifications of traditional reinforcement learning algorithms.

Kersting and De Raedt (Kersting & Raedt, 2003, 2004) introduced a first-order probabilistic STRIPS-like language to specify an MDP on a relational domain. An abstract policy is an ordered set of rules that can be seen as an abstraction of a Q-value table. Van Otterlo (Otterlo, 2003, 2004) abstraction is also a conjunction of first-order literals (with negation). Our approach and van Otterlo's use a slightly more powerful state abstraction, than Kersting and De Raedt, as they can include negation and arbitrary state predicates.

Kersting and Driessens (Kersting & Driessens, 2008) proposed a method to learn Parametric Policy Gradients for learning tasks. This model-free policy gradient approach deals with relational and propositional domains. They represent policies as weighted sums of regression models grown in a stage-wise optimization. Each regression model can be viewed as defining several new feature combinations. The idea is to start with an initial policy for a given task. Then calculate the gradient (through Friedmanns gradient boosting (Friedman, 2001)) values of this policy and move or adapt the parameters of the initial policy into the direction of the gradient. This method is recursively applied until no improvement on the policy is achieved. The method can developed control policies with continuous actions and deals with continuous states. It was applied to the blocks world and for teaching a robot how to traverse through a one-dimensional corridor. However the method is complex as it needs to compute the gradient of initially as many regression models as possible feature combinations (which for real robots with several sensors each of them providing readings at very high sample rates might become infeasible) and the method can lead to local maxima.

In (Raedt, Kimmig, & Toivonen, 2007) the authors developed ProbLog which is a probabilistic extension of Prolog. All clauses (facts) are labeled with the probability that they are true and these probabilities are mutually independent. Once a clause is tested as true, its probability value is calculated (given its parents from the derivation tree or previous clause's probability values) and then propagated to the subsequent clauses. Besides allowing relational representations the method is useful when we need to know the probability of a given predicate (which can be an action or state predicate) in order to make choices when having incomplete or uncertain information, however, there is no straightforward way to make ProbLog learn tasks that require the execution of continuous actions.

Another approach for tasks planning using a relational representation is the Planning Domain Definition Language (PDDL) (McDermott, 1998) which is an attempt to standardize planning domain and problem description languages. It was first developed by Drew McDermott mainly to make the 1998/2000 International Planning Competitions possible[1]. The learning task is defined by objects, predicates, the initial state, the goal specification, and actions/operators to change the state of the world. The language also allows the use of temporal logic where predicates are evaluated until some condition is achieved. The inference mechanisms of PDDL allow users to generate the sets of actions that make the goal specifications true. These goal specifications are evaluated true when their previous predicates are also evaluated true. By using this relational domain (through first order predicate logic) the learned tasks can be transferred between similar domains. PDDL was recently extended to specify MDPs (Younes & Littman, 2004), however, the user is responsible for specifying the transition and reward functions, as well as the actions. In this chapter the transition and reward function are not given in advance and the agent is able to learn the action from human traces.

Other languages have been defined to combine probability with first-order logic, such as Markov Logic Networks (Richardson & Domingos, 2006), Probabilistic Relational Models (Getoor, Koller, Taskar, & Friedman, 2000), FOCIL (Natarajan et al., 2005) and Bayesian Logic Programs (Kersting, Raedt, & Kramer, 2000) among others (see also (Getoor & Taskar, 2007)). This chapter provides an extended and more formal description of the approach given in (Morales, 2003), with other tests and results. A key issue in all these relational MDPs is the representation of states and actions which is normally provided by the user. A clear distinction of this chapter with previous approaches is the incorporation of Behavioral Cloning, to learn relational actions and produce faster convergence times. It also introduces two extensions, one is an exploration strategy to complete the information provided by human traces (illustrated with a flight simulator) and the other one is an on-line process to transform a discrete actions policy into a continuous actions policy (illustrated with a mobile robot).

Relational Representation for Reinforcement Learning

The key idea underlying relational reinforcement learning is to use relations instead of flat symbols. In this work, states are represented as sets of predicates that can be used to characterize a particular state and which may be common to other states. This representation allows us to:

- Create powerful abstractions, as states are characterized by a set of predicates. This makes it useful for large search spaces and applicable to relational domains.
- Learn policies which are, in general, independent of the exact position of the agent and the goal. This allows us to transfer policies to different problems where the same relations apply without any further learning, which is not possible with a propositional representation.

Exhibit 1.

```
r_action(2,S,rook,sq(D,E),sq(I,J),State1,State2):-
            rook_divs(S,king,sq(B,C),S,rook,sq(D,E),OS,king,sq(G,H),State1),
            opposition(S,king,sq(B,C),OS,king,sq(G,H),State1),
            make_move(S,rook,sq(D,E),sq(I,J),State1,State2),
            not threatkR(OS,king,sq(G,H),S,rook,sq(I,J),State2),
            l_patt(OS,king,sq(G,H),S,king,sq(B,C),S,rook,sq(I,J),State2).
```

Definition 1*: A relational state or r-state, s_R, is a conjunction of logical atoms.*

The extension of an *r-state* is the set of states that are covered by its description. That is, an *r-state* represents a set of states, where each state is represented by a conjunction of ground facts. More formally an *r-state*, s_R, represents all the states $s \in S$ for which there is a substitution Θ such that $s_R\Theta \subseteq s$. In our framework, each state $s \in S$ is an instance of one and only one *r-state*, which creates a partition over the state space *S*. In the chess domain, individual predicates could be *rook_threatened*, *kings_in_opposition*, *rook_divides_kings*, etc., which involve, in this case, the relative position of one, two or three pieces. A conjunction of these predicates represent an *r-state*, one of which could be *kings_in_opposition and rook_divides_kings*, that covers all the chess positions where these two relations hold (more than 3,000 positions). Once a set of relations has been defined, the search space in the relational space is completely defined.

The set of actions also use a first-order relational representation, similar to STRIPS operators or PDDL actions (McDermott, 1998).

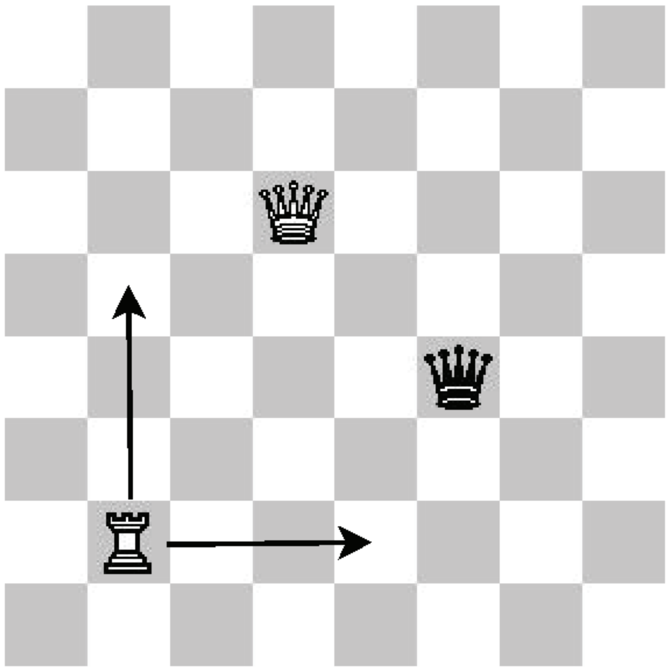
Definition 2*: A relational action or r-action, $a_R(s_R)$, is a set of pre-conditions, a generalized action, and possibly a set of post-conditions. The pre-conditions are conjunctions of relations that need to hold for the r-action to be applicable, and the post-conditions are conjunctions of relations that need to hold after a particular primitive action is performed. The generalized action represents all the instantiations of primitive actions that satisfy the conditions.*

For example, the following *r-action* (Exhibit 1), using Prolog notation, with id number 2 (first argument), says to move the rook of side S, from square (D,E) to square (I,J) if the rook divides the two kings and both kings are in opposition (pre–conditions), provided the rook is not threatened after the move and the three pieces form an L-shaped pattern (post–conditions) (see Figure 1).

For an *r-action* to be properly defined, the following condition must be satisfied: *If an r-action is applicable to a particular instance of an r-state, then it should be applicable to all the instances of that r-state*. This is not a problem for *r-actions* without post-conditions, as the pre-conditions are subsets of relations used to represent *r-states*, and consequently applicable to all the instances. This assures that *r-actions* associated with a particular *r-state* are always applicable in that *r-state*.

Similarly to the representation of states, the actions can be provided by the user. In this chapter, however, it is shown that once a set of relations has been defined, it is possible to induce the *r-actions* using a Behavioral Cloning approach, as described below. Each *r-action* can have different instantiations, resulting in different instantiations of the resulting *r-state*. When several possible instantiations of actions are possible, one is chosen randomly (uniform distribution). As can be seen, even assuming a deterministic policy in a relational

Figure 2. A non deterministic r-action, where two possible actions are possible satisfying all the conditions of the r-action



representation, the policy is still non deterministic at the ground level since different action instances can be selected. For example, an *r-action* could be to move the rook to divide both kings either horizontally or vertically. Figure 2 shows this case where there are two equally possible rook moves to divide both kings, both possible states are instances of the same *r-state*.

Once *r-state*s and *r-action*s are defined, a relational transition function can be defined giving the probability of being at state $s_{R'}$ given that an action $a_R(s_R)$ is applied in *r-state* s_R.

Definition 3: *A relational transition or r-transition* $P(s_R|a_R,s_R)$ *is defined by an r-state, an r-action and the resulting r-state, where each resulting instance in* s_R *has a uniform probability of occurring.*

A reward function is defined as in traditional MDP, giving a real number for each state $R_R : S \rightarrow R$. The same reward is given to all the instances of an *r-state* s_R.

Definition 4: *A relational MDP, or r-MDP, is a tuple* $M_R = <S_R, A_R, T_R, R_R>$ *where* S_R *is a set of r-states,* $A_R(S_R)$ *is a set of r-actions (one per r-state),* T_R *is an r-transition and* R_R *is the reward function.*

To summarize, abstract states are created with a set of properties expressed in a relational representation and a set of actions, using such properties, are defined for each abstract state. Reinforcement learning is then used to decide which action to perform on each state. The main advantages of the approach are that it is relatively easy to express and incorporate background knowledge, powerful abstractions can be achieved, and it is possible to re-use a previously learned policy on other instances of the problem and even on similar, although different, problems. The following

Algorithm 1. *The rQ-learning algorithm*

```
        Initialize $Q(s_R, a_R)$ arbitrarily (where $s_R$ is an r-state and $a_R$ is an
r-action)
        for each episode do
                Initialize s {a ground state}
                $s_R \leftarrow rel(s)$ {set of relations on state s}
                Repeat
                        for each step of episode do
                        Choose from using a persistently exciting policy
(e.g., greedy)
                        Randomly choose action a applicanle in
                        Take action a, observe r, $s'$
                        $Q(s_R, a_R) \leftarrow Q(s_R, a_R) + \alpha(r_R + \gamma max_{a'_R} Q(s'_R, a'_R) - Q(s_R, a_R))$
       $s_R \leftarrow s'_R$
                end for
        until s is terminal
end for
```

section describes how to learn (near) optimal policies for r-MDPs.

Reinforcement Learning on R-Space

For any Markov decision process, the objective is to find the optimal policy, i.e., one which achieves the highest cumulative reward among all policies. The main purpose to learn in an *r-space* is to reduce the size of the search space, and take all the advantages of a richer representation language. However, in the *r-space* there is no guarantee that the defined *r-actions* are adequate to find an optimal sequence of primitive actions and sub-optimal policies can be produced. We can however, defined optimality in terms of an *r-space*.

A policy consistent with our representation, which we will refer to as an *r-space* policy (π_R), is a scheme for deciding which *r-action* to select when entering an *r-state*. An *r-space* optimal policy (π_R^*) is a policy that achieves the highest cumulative reward among all *r-space* policies.

Definition 5*: A deterministic policy for an r-MDP, called r-space policy, $\pi_R : S_R \rightarrow A_R$ specifies which r-action, $a_R(s_R)$, to perform on each r-state, s_R.*

The expected reward, in this case, is the expected *average* reward over all the instances of the *r-state*. When several *r-actions* are applicable in a particular *r-state*, the best policy will prefer the *r-actions* which lead to an *r-state* with the best expected average reward.

The rQ-Learning Algorithm

This paper focuses on applying Q-learning (Watkins, 1989) in *r-space*, although a similar argument can be applied to other reinforcement learning algorithms, such as TD(λ) or SARSA (Sutton & Barto, 1989). Algorithm 1 gives the pseudo-code for the rQ-learning algorithm. This is very similar to the Q-learning algorithm, but the states and actions are characterized by relations. The algorithm still takes primitive actions (a's) and moves over primitive states (*s*'s), but learns over *r-state*-*r-action* pairs.

It will be shown, empirically, that obtaining an *r-space* policy is good enough to solve complex problems and that in many cases, it also corresponds to the optimal policy at the ground level, although this depends on the description of the *r-space*. Also, since we are learning over generalized actions and states, the same policy is applicable to different instances of the problem and sometimes to other problems where the same set of relations hold. Even with an abstract representation, reinforcement learning can take a considerable time to converge to a suitable policy. In the following section, it is shown how to accelerate the learning process using traces provided by users.

Accelerating the Learning Process

So far, we have assumed that the r-actions are defined by the user. One way to learn *r-actions* is from log traces of humans, or other computer systems, solving a task. Behavioural Cloning (BC) induces control rules from traces of skilled operators, e.g., (Sammut, Hurst, Kedzier, & Michie, 1992, Michie, Bain, & Hayes-Michie, 1990, Bratko, Urbančič, & Sammut, 1998). The general idea is that if the human is capable of performing a task, rather than asking him/her to explain how it is performed, he/she is asked to perform it. Machine Learning is then used to produce a symbolic description of the skill.

In this chapter we use logs of human traces to learn and identify only a subset of potentially relevant actions and then use reinforcement learning over this abstracted and reduced search space to learn a policy. Our approach can use traces from several experts, which may choose different actions. The traces provided by the user(s) are transformed into a relational representation. From these relational traces a set of *r-actions* is learned for each *r-state*. The BC approach can learn good and bad *r-actions* but then relational reinforcement learning is used to decide which are the best *r-actions* to use on each *r-state*. In this sense, it is an incremental approach, as new traces can be given at any time.

Contrary to other related approaches, like Programming by Demonstration, we are not limited by the quality of the traces provided by the user, as we accept traces from several users and of different quality and our reinforcement learning algorithm finds which is the best action to perform on each state. Also we are focusing the search space to a limited number of actions which significantly reduces the convergence times of reinforcement learning.

In order to learn a new *r-action* each frame in the human trace is transformed, if necessary, to a set of predefined predicates, and the action performed by the user is observed. The action may also need to be transformed into a predicate. A new *r-action* is constructed with the conjunction of the above predicates, unless the control action is an instance of an already defined *r-action* (see also (Morales, 1997) for a similar approach used in chess end-games). The predicates that are true in the current frame are used as conditions that need to be satisfied in order to consider such action. In general, an *r-action* has the following format (Exhibit 2):where *predicate i,s*1(*Args$_i$*,*State*1) is a predicate that needs to be true for the action to be executed (a precondition) and may have some particular arguments (*Args*) and *predicate_action*(*Action*,*State*1), is the predicate that performs the action. In some domains, like chess, it is possible to predict the next state after the action is executed and extract predicates that can then be used as post-conditions. In that case, an *r-action* has the following format (Exhibit 3):where *State*2 contains information of the next state after the action has been executed and *predicate j,s*2 is a post-condition.

In general, a trace will contain low level information of the domain. For example, a trace in chess will consist of board positions (e.g., white in rook in position g3, black king in b6, etc.) with

Exhibit 2.

```
r_action(Action,State1):-
             predicate1,s1 (Args1,State1),
             predicate2,s1 (Args2,State1),
             ...
             predicate_action(Action,State1).
```

Exhibit 3.

```
r_action(Action,State1,State2):-
             predicate 1,s1 (Args1,State1),
             predicate 2,s1 (Args2,State1),
             ...
             predicate_action(Action,State1,State2),
             predicate 1,s2 (Args3,State2),
             predicate 2,s2 (Args4,State2),
             ...
```

plies (e.g., white rook moves from g3 to g6). A trace in robotics will consist of sensors' information (e.g., laser reading 1 is 0.32, laser reading 2 is 0.31, ..., sonar1 reading is 2.78, sonar2 reading is 3.18, etc.) and movement information (e.g., speed is 5.1 cm/sec). This information is transformed into predicates such as, *rook_threatened*, *in_opposition* and *make_move* for chess or *door_detected*, *obstacle_detected* and *go_forward* or *turn_right* for robotics, from which the *r-actions* are build.

The approach only learns the *r-actions* used in the traces. This substantially reduces the learning process, as only a subset of actions are considered per state, but can also generate sub-optimal policies as not all the applicable actions are considered per state. It is also possible that some *r-states* are never visited. These issues are later discussed in this chapter. Algorithm 2 summarizes the behavioral cloning approach used for reinforcement learning.

Experiments

We illustrate the approach in two challenging domains and illustrate two possible improvements to the proposed approach. The first domain is a flight simulator, where the goal is to learn how to fly an aircraft. In this experiment we illustrate how to incorporate an exploration strategy to produce more robust policies. The second domain is a mobile robot, where the goal is to learn navigation tasks. In this experiment we show how to transform a discrete action policy into a continuous action policy.

Learning to Fly

Trying to learn how to control an aircraft is a challenging task for reinforcement learning as it may typically involve 20 to 30 variables, most of them continuous, describing an aircraft moving in a potentially "infinite" three dimensional space.

A flight simulator based on a high fidelity model of a high performance aircraft (a Pilatus PC-9 acrobatic air plane) was used in our experiments.[2] The PC-9 is an extremely fast and maneuverable aircraft used for pilot training. The model, provided by the Australian Defense Science and Technology Organization (DSTO), is

Algorithm 2. Behavioral cloning approach for reinforcement learning

```
Given a set a trace-logs (Each composed of a set of frames) of the task we
want to learn for each frame doTransform the information of the frame into a
relational representation (r-state) using a pre-defined set of
    Predicates
    Transform the information from the action performed by the user into a re-
lational representation (predicate-
    action)
   Construct, if new, an r-action with the conjunction of the r-state and the
predicate-action
end for
```

based on wind tunnel and in-flight performance data. Since the flight simulator is of an acrobatic aircraft, small changes in control can result in large deviations in the aircraft position and orientation. This chapter only deals with controlling the ailerons and elevators, which are the two most difficult tasks to learn. In all the experiments, it was assumed that the aircraft was already in the air with a constant throttle, flat flaps and retracted gear. Turbulence was added during the learning process (both behavioral cloning and reinforcement learning) as a random offset to the velocity components of the aircraft, with a maximum displacement of 10*ft*/*s* in the vertical direction and 5*ft*/*s* in the horizontal direction.[3]

The goal is to learn how to fly through a sequence of ways points, each way point with a tolerance of 100*ft* vertically and horizontally. Thus the aircraft must fly through a "window" centered at the way point.

We decided to divide the task into two independent reinforcement learning problems: (i) forward-backward movements to control the elevation of the aircraft and (ii) left-right movements to control the roll and heading of the aircraft. We assume that in normal flight, the aircraft is approximately level so that the elevators have their greatest effect on elevation and the ailerons on roll.

To characterize the states for elevation control the following predicates and discretized values were defined:

- *distance_goal*: relative distance between the plane and the current goal. Possible values: *close* (less than 100 ft), *near* (between 100 and 1,000 ft), and *far* (more than 1,000 ft).
- *elevation_goal*: difference between current elevation of the aircraft and the goal elevation, considering the plane's current inclination. Possible values: *far_up* (more than 30°), *up* (between 5° and 30°), *in_front* (between 5° and -5°), *down* (between -5° and -10°), and *far_down* (less than -30°).

For aileron control, in addition to *distance_goal*, the following predicates and discretized values were also defined:

- *orientation_goal*: relative difference between current yaw of the aircraft and goal yaw, considering the current orientation of the plane. Possible values: *far_left* (less than -30°), *left* (between -5° and -30°), *in_front* (between -5° and 5°), *right* (between 5° and 30°), and *far_right* (more than 30°).
- *plane_rol*: current inclination of the plane. Possible values: *far_left* (less than -30°),

Exhibit 4.

```
r_action(Num,el,State,move_stick(StickMove)) ←
          distance_goal(State,DistGoal) | ∧
          elevation_goal(State,ElevGoal) ∧
          move_stick(StickMove).
```

Exhibit 5.

```
r_action(Num,al,State,move_stick(StickMove)) ←
          distance_goal(State,DistGoal) ∧
          orientation_goal(State,Orient_Goal) ∧
          plane_rol(State,PlaneRol) ∧
          plane_rol_trend(State,RolTrend) ∧
          move_stick(StickMove).
```

left (between -5° and -30°), *horizontal* (between -5° and 5°), *right* (between 5° and 30°), and *far_right* (more than 30°).

- *plane_rol_trend*: current trend in the inclination of the plane. Possible values: *inc* (more than +1), *std* (between +1 and −1), *dec* (less than −1).

The ranges of the discretized values were chosen arbitrarily at the beginning of the experiments and defined consistently across different variables with no further tuning. The exact values appear not to be too relevant, but further tests are needed.

The actions were discretized as follows. The X component of the stick can have the following values: *farleft* (if stick X component value is less than -0.1), *left* (if it is between -0.1 and -0.03), *nil* (between -0.03 and 0.03), *right* (between 0.03 and 0.1), and *farright* (greater than 0.1). For the Y component of the stick movements the following discretization was used: *fardown* (above 0.4), *down* (between 0.3 and 0.4), *nil* (between 0.2 and 0.3), *up* (between 0.1 and 0.2), and *farup* (below 0.1). These discretizations were based on the actions performed by human pilots. These predicates were used to construct, from human traces, a set of aileron and elevation *r-actions*. Elevation *r-actions* have the following format (Exhibit 4): where *Num* is an identification number, *StickMove* is one of the possible values for stick on the Y coordinate, *DistGoal* is one of the possible values for *distance_goal*, and *ElevGoal* is one of the possible values for *elevation_goal*. For elevation there can be 75 possible *r-actions* (3 possible values for *DistGoal*, 5 for *ElevGoal*, and 5 for *StickMovement*).

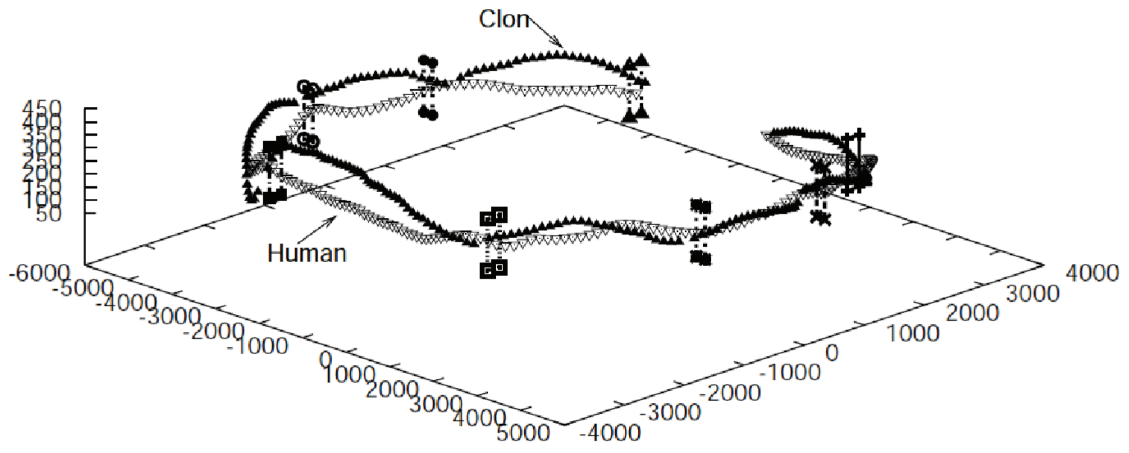
Similarly, the format for the aileron *r-actions* is as follows (Exhibit 5):where there can be 1,125 possible aileron *r-actions*.

A total of 222 *r-actions* (180 for aileron and 42 for elevation) were learned after 5 consecutive mission logs over the flight plan shown in Figure 3.

Exploration Mode

As we are learning only from seen cases, there may be some states descriptions not covered by the *r-actions* but which may occur in other flight maneuvers. To compensate for this, the learned *r-actions* were used to fly the aircraft to try to reach previously unseen situations. In cases where there were several applicable *r-actions*, one was chosen randomly. Whenever the aircraft reached a new state description (where there was no

Figure 3. Human trace and the trace using the learned policy with reinforcement learning and behavioral cloning on the 8-goal mission with the maximum level of turbulence



applicable *r-action*), the system prompted the user for a suitable action, from which a new *r-action* was induced. Also the user was able to perform a different action in any state if he/she wished, even if there were some applicable *r-actions*. Exploration mode continued until almost no new *r-actions* were learned, which was after 20 consecutive exploratory flights. In total, 407 *r-actions* were learned, 359 for aileron (out of 1,125 which is ≈32%) and 48 for elevation (out of 75, which is ≈64%). So although, we are still learning a substantial number of *r-actions* behavioral cloning helps us to learn only a subset of the possible *r-actions* (only one third) focusing the search space and simplifying the subsequent reinforcement learning task (an average of 1.6 *r-actions* per aileron state and 3.2 per elevation state).

When performing an *r-action*, the actual value of the stick position was assigned as the mid point of the intervals, except for the extreme ranges, as follows. For the X coordinate: *farleft* = −0.15, *left* = −0.05, *nil* = −0.01, *right* = 0.05, and *farright* = 0.15. For the Y coordinate: *fardown* = 0.45, *down* = 0.35, *nil* = 0.25, *up* = 0.15, and *farup* = 0.05. This discrete actions policy was adequate to learn a reasonable flying strategy as it will be shown in the experimental results. We will describe, in the robotics domain, how to transform this type of policies into policies with continuous actions.

Once the state has been abstracted and the *r-actions* induced, rQ-learning is used to learn a suitable policy to fly the aircraft.

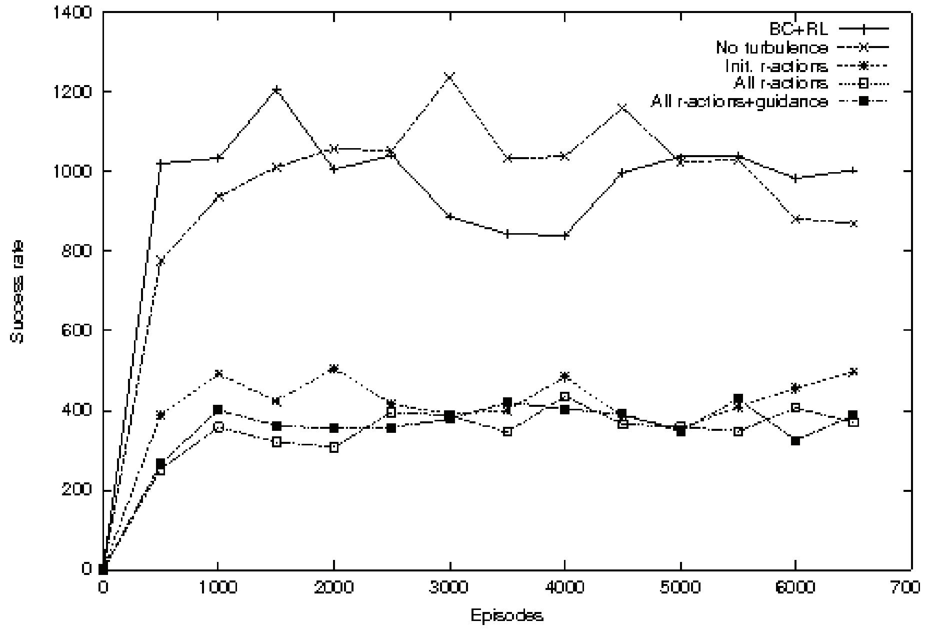
Results

In all the experiments, the Q values were initialized to -1, $\varepsilon = 0.1$,$\gamma = 0.9$,$\alpha = 0.1$, and $\lambda = 0.9$. The experiments were performed on the 8 goals mission shown in Figure 3. If the aircraft increases its distance to the current goal, after 20 time steps have elapsed from the previous goal, it is assumed that it has passed the goal and it changes to the next goal.

The following experiments were performed:

1. Positive reinforcement (+20) was given only when crossing a goal within 100 ft. with negative rewards otherwise (-1). In case the aircraft crashed or got into a state with no applicable *r-action*, a negative reward was given (-10). The experiments were performed with the maximum level of turbulence.
2. Same as (1) without turbulence.

Figure 4. Learning curve for aileron for the different experimental set-ups while training



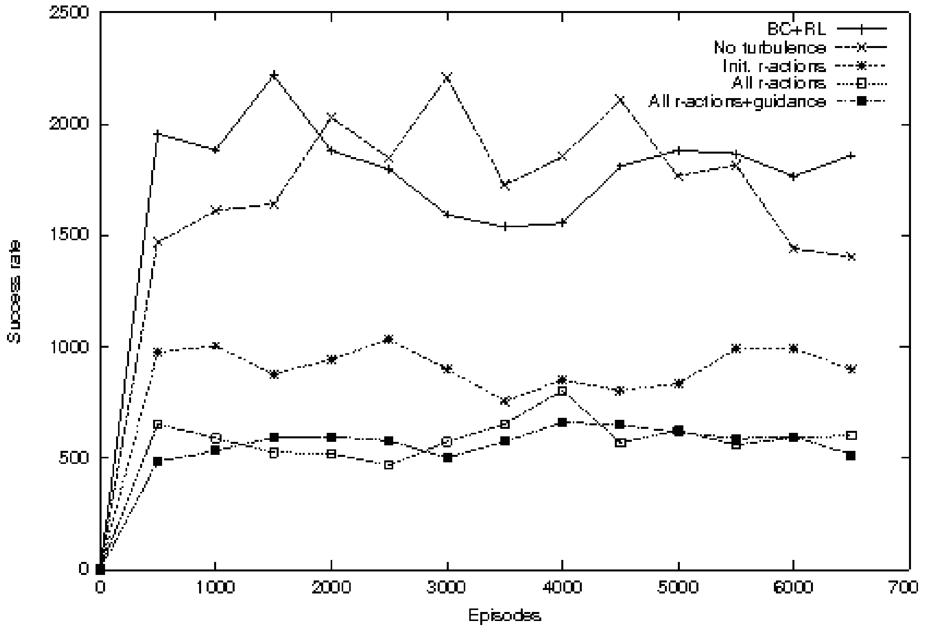
3. Same as (1) but only with the *r-actions* learned from the original traces, i.e., without the exploration stage (222 *r-actions* in total).
4. Same as (1) but we automatically generate all the possible *r-actions* per state (5 with our discretization scheme) with 1200 *r-actions* in total.
5. Same as (4) but we use the original traces to "seed" Q-values, providing initial guidance.

Figures 4 and 5 show the learning curves of the above experiments for aileron and elevation respectively. In particular, how many times the aircraft crosses successfully the eight goals with maximum turbulence (every 500 flights) for aileron and elevation control. We continued the experiments for 20,000 episodes without any clear improvements in any of the experiments after the first 3,000 episodes.

As can be seen from the figures, without focusing the search with behavioral cloning, reinforcement learning is unable to learn an adequate strategy in a reasonable time. In complex environments, spurious actions can very easily lead an agent to miss the goal. In this particular domain, going away from the current goal at some intermediate state can lead the agent into a situation where it is impossible to recover and reach the goal without first going away from the goal. The exploratory phase, where new *r-actions* were learned using random exploration, also proved to be useful as the initial traces substantially biased the learning process and limited its applicability.

The policy learned in experiment 1 after 1,500 flights was able to fly the whole mission successfully. Its robustness was tested under different turbulence conditions. Figure 3 shows a human trace of the mission and the trace followed by the learned policy. Table 1 shows the results, averaged over 100 trials, of flying the learned policy on the mission with different levels of turbulence. Two columns are shown per turbulence level, one with percentages passing the way point within 100 ft. (which was used in the reward scheme) and one

Figure 5. Learning curve for elevation for the different experimental set-ups while training



with 200 ft. The important point to note is that the aircraft can recover even if it misses one or more goals, and although it occasionally misses some of the goals, it gets quite close to them, as can be seen from Figure 3.

The learned policies were then tested on a completely different mission, consisting of four way points. The intention was to try maneuvers not previously seen before. The new mission included: a right turn[4], a sharper left climb turn of what it has previously seen before, another quick right turn, and a sharp descending right turn.

Figure 6 shows a human trace and the trace using the previously learned policy (experiment 1) on the new mission with the maximum level of turbulence. The learned policy of the previous mission is clearly able to fly the aircraft on a completely new mission. Table 2 shows the performance of the policy on this new mission with different turbulence levels averaged over 100 trials.

An Application in Mobile Robots

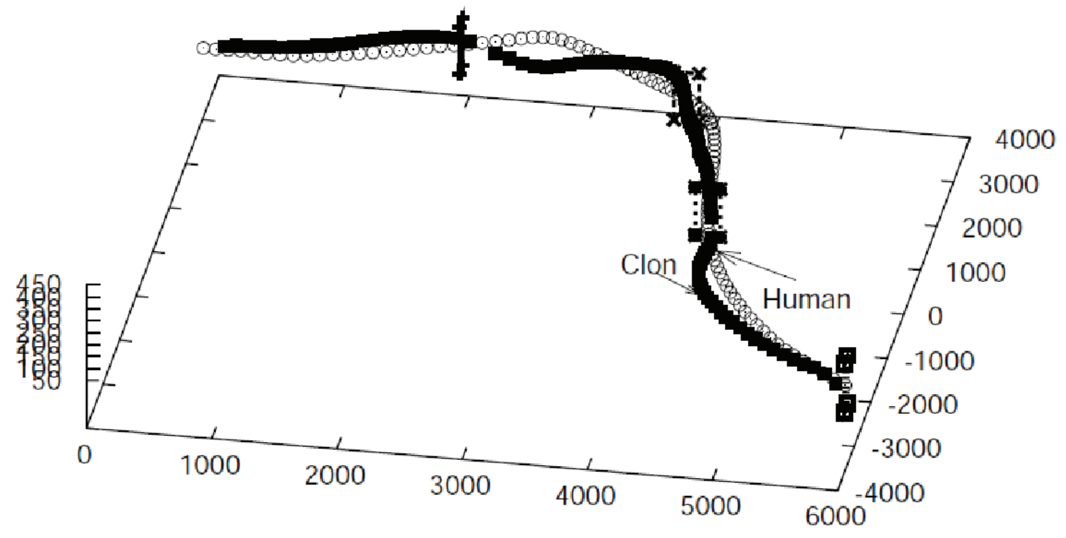
Our second experiment consists of teaching a mobile robot how to perform navigation tasks in office–like environments with continuous actions.

To define a representation in terms of common objects found in office-like environments, like rooms, corridors, walls, doors, and obstacles, we first transform the low-level information from sensors into high-level predicates representing such objects. This transformation process is based on the work developed in (Hernández & Morales, 2006), where they defined natural landmarks, such as: (1) discontinuities, defined as an abrupt variation in the measured distance of two consecutive laser readings, (2) corners, defined as the location where two walls intersect and form and angle, and (3) walls, identified from laser sensor readings using the Hough transform. We also add obstacles identified through sonars and defined as any detected object between certain range. We also used the work described in (Herrera-Vega,

Table 1. Performance of the learned policy of experiment 1 after 1,500 episodes with different levels of turbulence on the eight goals mission

	Turbulence (m/s)/Tolerance					
Stage	0/	0/	5/	5/	10/	10/
	100	200	100	200	100	200
Goal1	0	100	31	75	49	89
Goal2	100	100	16	41	26	46
Goal3	0	100	53	62	51	70
Goal4	0	0	23	35	27	46
Goal5	0	100	57	91	59	95
Goal6	0	0	16	33	24	47
Goal7	100	100	47	74	33	66
Goal8	0	100	35	58	45	61
Aver.	25	75.0	34.75	58.625	39.25	65.0

Figure 6. Flight path trace for a human and for the learned policy on a new mission with 4 goals



2009) to identify the robot's current location such as room, corridor and/or intersection. These natural landmarks, along with the robot's actual location, are used to automatically characterize the relational states that describe the environment in real–time.

The predicates used for developing the robot's r-states are:

- *place*: This predicate returns the robot's location, which can be *in-room*, *in-door*, *in-corridor* and *in-intersection*.

Table 2. Performance of the learned policy on a different mission with different levels of turbulence

	Turbulence (m/s)/Tolerance					
Stage	0/	0/	5/	5/	10/	10/
	100	200	100	200	100	200
Goal1	0	100	66	99	74	100
Goal2	100	100	17	39	29	44
Goal3	100	100	38	70	46	70
Goal4	0	100	51	77	39	58
Aver.	50	100	43	71.25	47	68

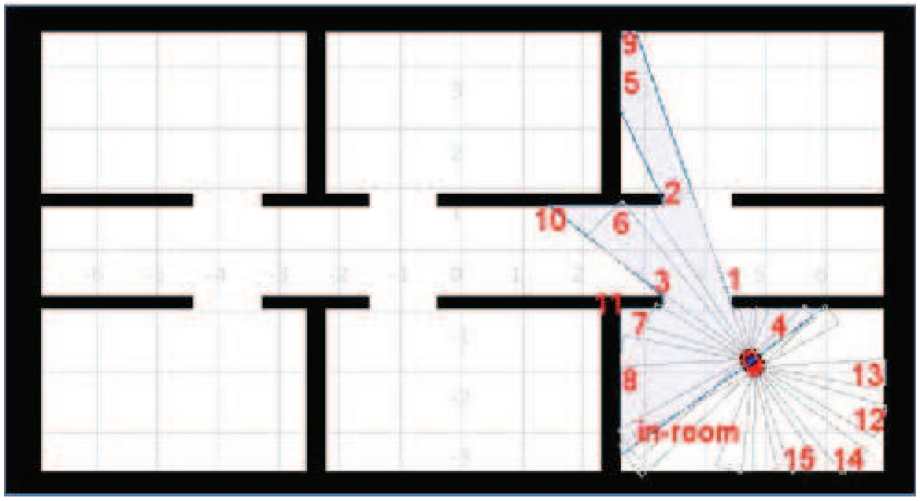
- *doors_detected*: This predicate returns the orientation and distance to doors. A door is characterized by identifying a right discontinuity (*r*) followed by a left discontinuity (*l*) from the natural landmarks. The door's orientation angle and distance values are calculated by averaging the values of the right and left discontinuities angles and distances. The discretized values used for door orientation are: *right* (door's angle between -67.5° and -112.5°), *left* (67.5 to 112.5°), *front* (22.5° to -22.5°), *back* (157.5° to -157.5°), *right-back* (-112.5° to -157.5°), *right-front* (-22.5° to -67.5°), *left-back* (112.5° to 157.5°) and *left-front* (22.5° to 67.5°). The discretized values used fro distance are: *hit* (door's distance between 0*m*. and 0.3*m*.), *close* (0.3*m.* to 1.5*m*.), *near* (1.5*m*. to 4.0*m*.) and *far* (> 4.0*m*). For example, if a right (number 1 in Figure 7) and left (number 3 in Figure 7) discontinuities are obtained from the robot's sensors, then the following predicate is produced *doors_detected*([*front*, *close*, -12.57, 1.27]), which corresponds to the orientation (first and third argument in symbolic and numeric representation) and distance (second and fourth argument) descriptions of a detected door (see Figure 7). For every pair of right and left discontinuities a list with these orientation and distance descriptions is generated.
- *walls_detected*: This predicate returns the length, orientation and distance to walls (type *w* landmarks).[5]

The possible values for the wall's size are: *small* (length between 0.15*m*. and 1.5*m*.), *medium* (1.5*m*. to 4.0*m*.) and *large* (> 4.0*m*.).

- *corners detected*: This predicate returns the orientation and distance to corners (type *c* landmarks).[6]
- *obstacles_detected*: This predicate returns the orientation and distance to obstacles (type *o* landmarks).[7]
- *goal_position*: This predicate returns the relative orientation and distance between the robot and the current goal. It receives the parameter the robot's current position and the goal's current position, though as a trigonometry process, the orientation and distance values are calculated and then discretized.[8]
- *goal_reached*: This predicate indicates if the robot is in its goal position. Possible values are *true* or *false*.

The previous predicates tell the robot if it is in a room, a corridor or an intersection, detect walls, corners, doors, obstacles and corridors and give a rough estimate of the direction and distance to the goal. The action predicates receive as parameters

Figure 7. Robot sensing its environment through laser and sonar sensors and corresponding natural landmarks



the odometer's speed and angle readings, and are defined as follows:

- *go*: This predicate returns the robot's actual moving action. Its possible values are *forward* and *back*.
- *turn*: This predicate returns the robot's actual turning action. Its possible values are *right* and *left*.

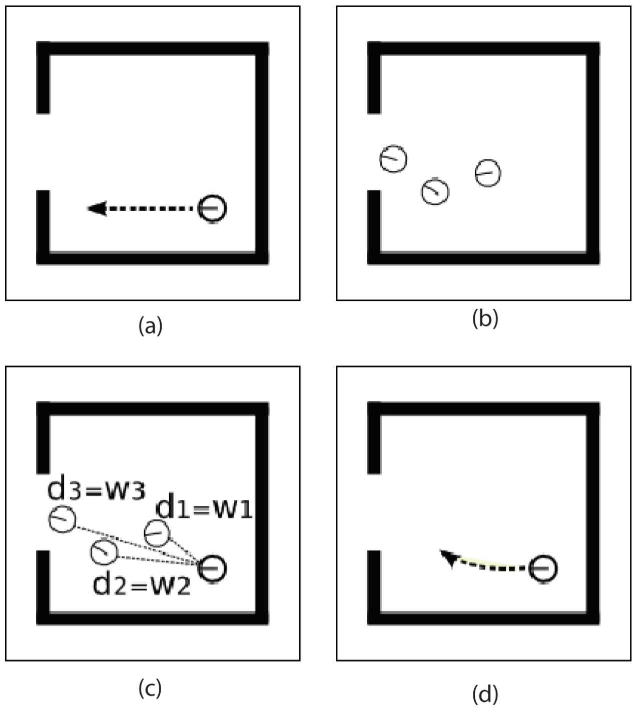
Similarly to the flight simulator domain, initial traces are given to the system which uses the previously defined predicates to induce a set of *r-actions* and used them to learn a discrete actions policy. In the following section we describe a real-time post-processing stage that can be applied to the previous approach to produce policies with continuous actions.

Relational Policies with Continuous Actions

This stage refines the coarse actions from the discrete actions policy previously generated. This is achieved using Locally Weighted Regression *(*LWR). The idea is to combine discrete actions' values give by that policy with the action's values previously observed in traces and stored in a database *DB*. This way the robot follows the policy learned with rQ-learning, but the actions are tuned through the *LWR* process.

What we do is to detect the robot's actual *r*-state. For this *r*-state the discrete actions policy determines the action to be executed (Figure 8(a)). Before performing the action, the robot searches in the *DB* for all the registers that share this same *r-state* description (Figure 8(b)). Once found, the robot gets all of the numeric orientation and distance values from these registers. These orientation and distance values are used to perform a triangulation process. This process allows us to estimate the relative position of the robot from previous traces with respect to the robot's actual position. Once this position has been estimated, a weight is assigned to the previous traces action's values. This weight depends on the distance of the robot from the traces with respect to the actual robot's position (Figure 8(c)). These weights are used to perform the *LWR* that produces continuous *r-actions* (Figure 8(d)).

Figure 8. Continuous actions developing process. (a) r-state and corresponding r-action. (b) A trace segment. (c) Distances and weights. (d) Resulting continuous action.



Once all the distance values (d_i) are calculated for all the registers in the *DB* with the same *r-state*, we apply a Gaussian kernel $w_i(d_i) = exp(-d_i^2))$ to obtain a new weight w_i for each relevant register.

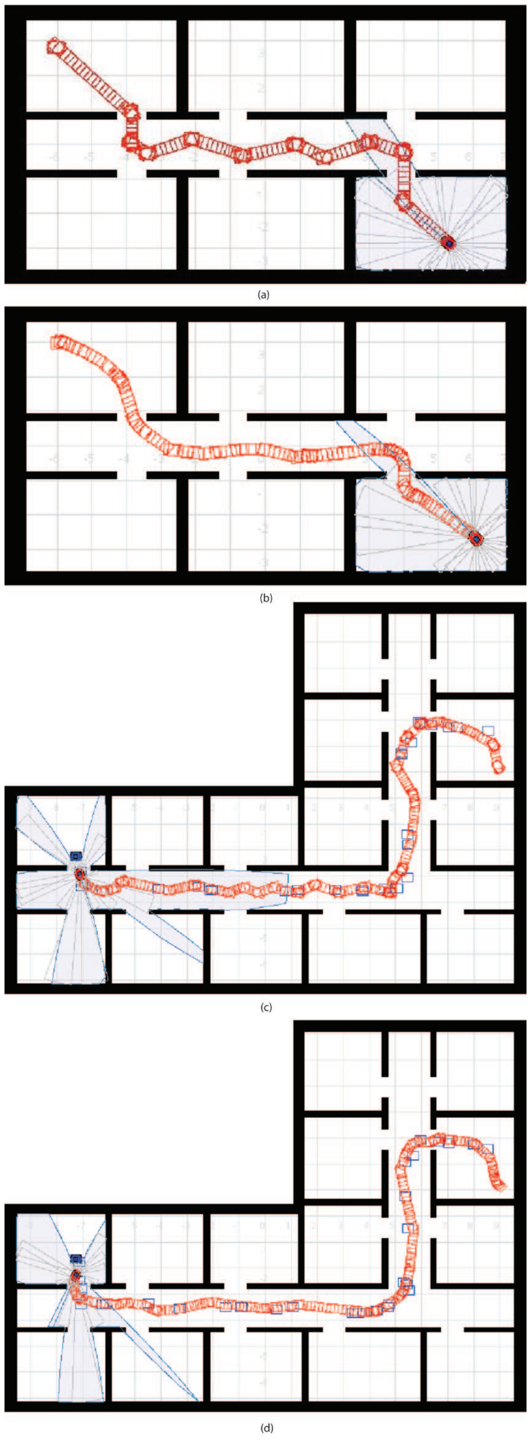
Then, every weight w_i is multiplied by the corresponding speed and angle values ($w_i \times speed_{DBi}$ and $w_i \times angle_{DBi}$) of the *r-state-r-action* pairs retrieved from the *DB*. The resulting values are added to the *r-action* (rA_t = *{disc_speed, disc_angle}*) values of the policy obtained by rQ-learning in order to transform this discrete *r-action* into a continuous action that is executed by the robot. This process is applied to every register read from the *DB* with the same *r-state* description and is repeated every time the robot reaches a new *r-state*.

The main advantage of our approach is the simple and fast strategy to produce continuous actions policies that are able to produce smoother and shorter paths in different environments.

Results. Experiments were carried out in simulation (*Player/Stage* (Vaughan, Gerkey & Howard, 2003)) and with a real robot[9]. Both robots (simulated and real) are equipped with a 180° front *SICK* laser sensor and an array of 4 back sonars at -170°, -150°, 150° and 170°. The laser range is 8.0*m* and the sonar range is 6.0*m*. The tasks to perform in these experiments were: (i) *navigating* through the environment and (ii) *following* a moving object.

The policy learning process was carried out in the Map 1 shown in Figure 9. For each of the two tasks a set of 15 traces was generated in this map. For the *navigation* tasks, the robot and the goal's global position (for the *goal_position* predicate) were calculated using the work developed in (Hernández & Morales, 2006). For the *following* tasks we used a second robot which orientation and angle was calculated through laser sensor.

Figure 9. Tasks examples from Maps 1 (size 15.0m.×8.0m.) and 2 (size 20.0m.×14.0m.). (a) Navigation task with discrete actions. (b) Navigation task with continuous actions. (c) Following task with discrete actions. (d) Following task with continuous actions.



We applied our Behavioral Cloning approach to all the traces, to define *r-states* and induce relevant *r-actions*. Then, *rQ-learning* was applied to learn the policies. For generating the policies, *Q-values* were initialized to -1, $\in = 0.1$, $\gamma = 0.9$ and $\alpha = 0.1$. Positive reinforcement, *r*, (*+100*) was given when reaching a goal (within 0.5 m.), negative reinforcement (*-20*) was given when the robot hit an element and no reward value was given otherwise (*0*). To generate the continuous actions policy, Locally Weighted Regression was applied on-line using a Gaussian kernel. Once the policies were learned, experiments were executed in the training map with different goal positions and in two new and unknown environments for the robot (Map 2 shown in Figure 9(c) and Map 3 shown in Figure 10). A total of 120 experiments were performed: 10 different navigation and 10 following tasks in each map, executed first with the discrete actions policy from the first stage and then with the continuous actions policy from the second stage. Each experiment has a different distance to cover and requires the robot to traverse through different places. The minimum distance to cover was 2*m*. (Manhattan distance), and it was gradually increased up to 18*m.*

Figure 9 shows a *navigation* (Map 1) and a *following* task (Map 2) performed with discrete (left figures) and continuous (right figures) actions, respectively.

Figure 10 shows a *navigation* and a *following* task performed with the real robot, with the discrete and with the continuous actions policy.

We cannot guarantee that the user visited all the possible *r-states*. If the robot reaches an unseen *r-state*, it asks for guidance to the user. Through a joystick, the user indicates the robot which action to execute and the robot stores this new *r-state-r-action* pair in *DB*. The number of unseen *r-states* decreases with the number of experiments.

We evaluated the performance of the discrete and continuous actions policies in three aspects:

1. How close the paths of the tasks are to the paths performed by a user
2. How close the paths o the tasks are from obstacles in the environment
3. What are the execution times of the policies

Figure 11(a)[10] shows results in terms of quality of the performed tasks with the real robot. This comparison is made against tasks performed by humans. All of the tasks performed in the experiments with the real robot, were also performed by a human using a joystick (Figures 10(c), 10(f)), and logs of the paths were saved. The graph shows the normalized squared error between these logs and the trajectories followed by the robot.

Figure 11(b) shows results in terms of how much the robot gets closer to obstacles. This comparison is made using the work developed in (Romero, Morales, & Sucar, 2001). In that work, values were given to the robot according to its proximity to objects or walls. The closer the robot is to an object or wall the higher the cost it is given. Values were given as follows: if the robot is very close to an object (0*m*. to 0.3*m*).a value of -100 was given, if the robot is close to an object (0.3*m*. to 1.0*m*) a value of -3 was given, if the robot is near an object (1.0*m*. to 2.0*m*) a value of -1, otherwise a value of 0 is given. As can be seen in the figure, quadratic error and penalty values for continuous actions policies are lower than those with discrete actions.

Execution times, shown in Figure 12, with the real robot were also registered. Continuous actions policies execute faster paths than the discrete actions policy despite our triangulation and Locally Weighted Regression processes.

In summary, the continuous actions policies are more similar to human traces, are smoother, safer and faster than the discrete actions policies.

Figure 10. Navigation and following tasks examples from Map 3 (size 8.0m.×8.0m.). (a) Navigation task with discrete actions. (b) Navigation task with continuous actions. (c) Navigation task performed by user. (d) Following task with discrete actions. (e) Following task with continuous actions. (f) Following task performed by user.

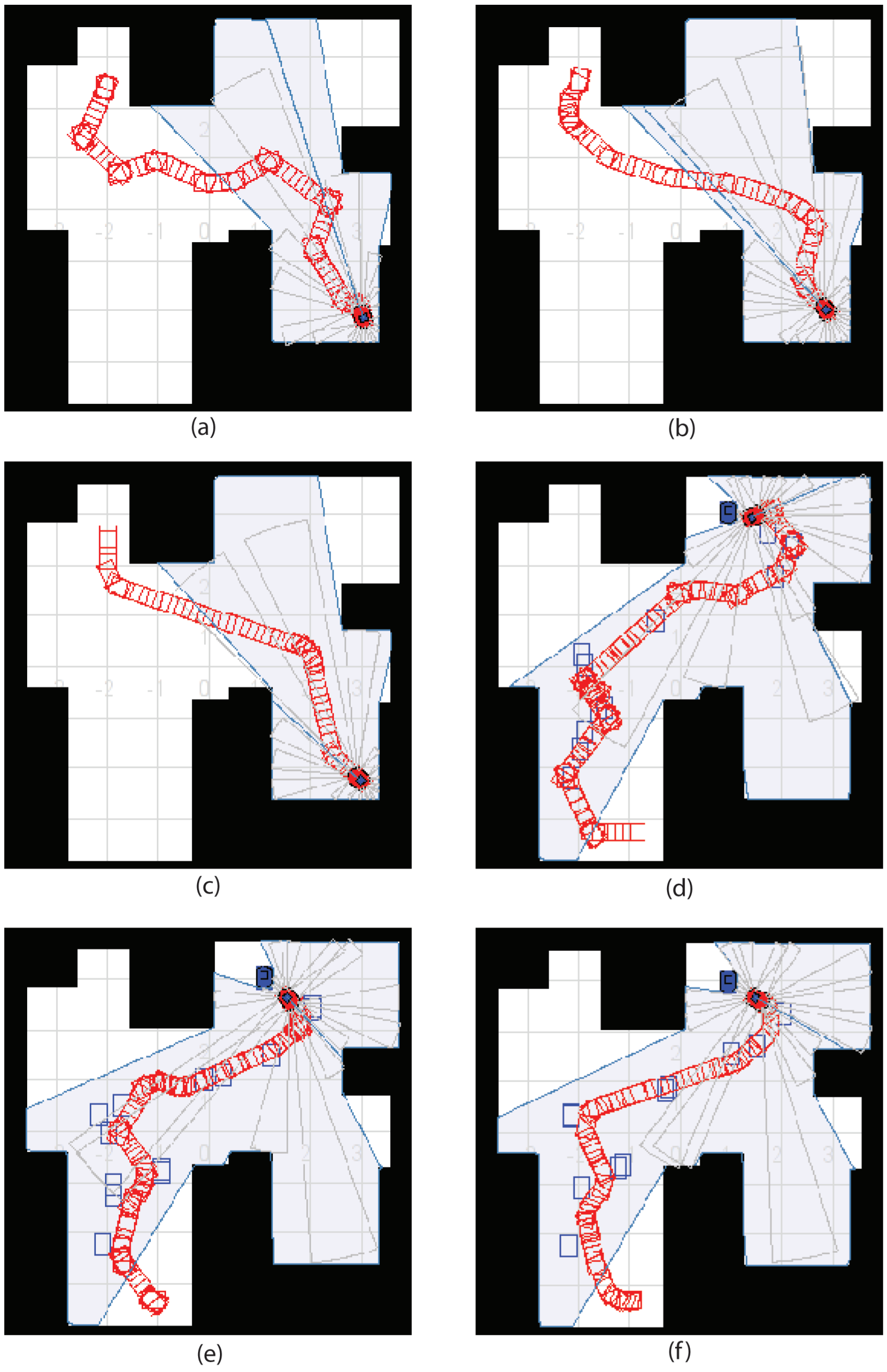
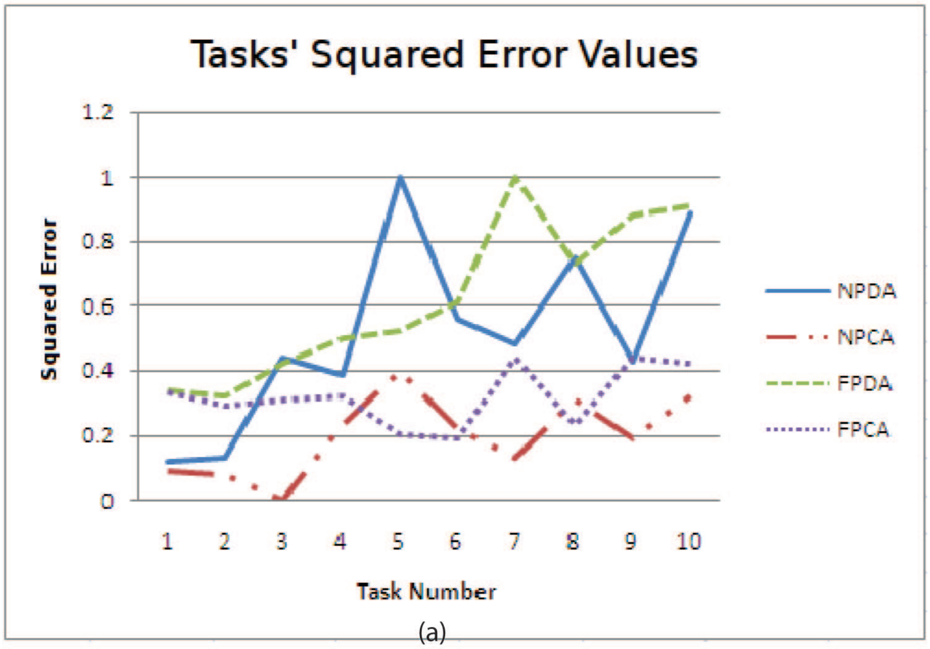


Figure 11. Navigation and following results of the tasks performed by the real robot. (a) Squared error value. (b) Penalty values



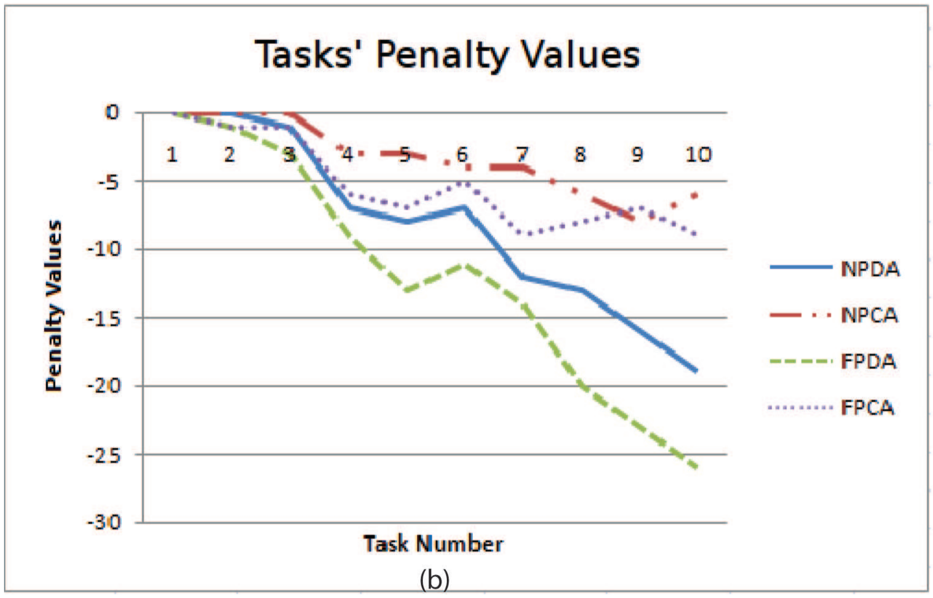
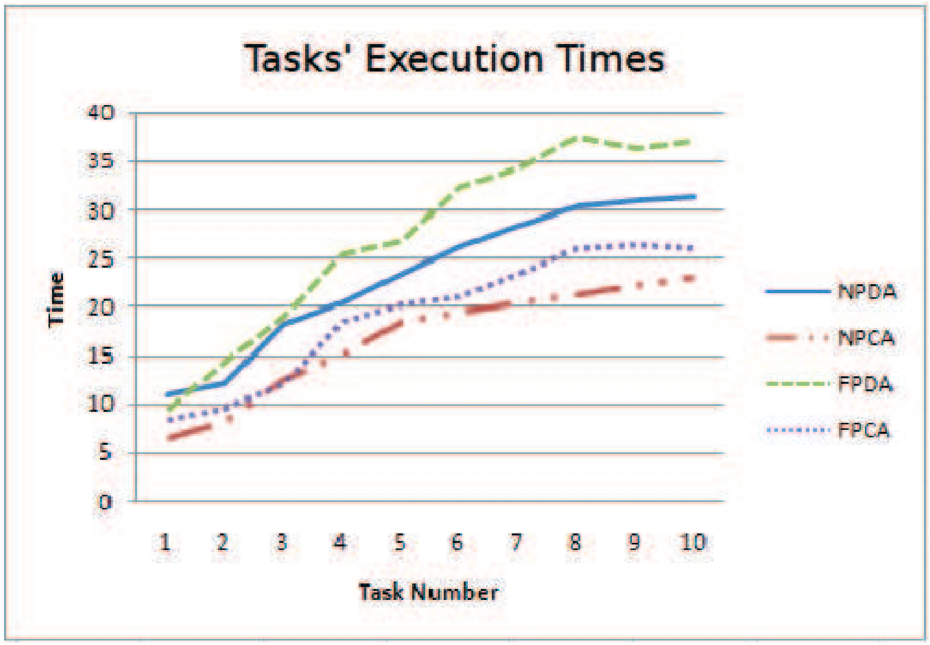


Figure 12. Execution times results



CONCLUSION AND FUTURE WORK

This work introduces an abstraction based on a relational representation for reinforcement learning. The user defines a set of properties to characterize a domain and to abstract the state space. The idea is to capture some relevant properties of the domain that can be used in different instances of the domain and sometimes to solve different, although closely related, problems. It is also shown how to incorporate traces from the user to automatically learn a reduced set of relational actions to achieve faster convergence times.

Under the proposed framework it is easy to incorporate domain knowledge, to use it in large application domains, and to re-use the learned policies on other instances of the problem or on related problems that can be described by the same set of relations.

We also showed two improvements in two challenging domains: (i) an exploration strategy to learn how to behave in unexplored states and (ii) an on-line strategy to produce a continuous actions policy.

There are several future research directions that we are considering. In particular, we would like to improve our current exploration strategy to identify in advance non-visited states to complete the traces provided by the user. We are also considering how to include partial observability in our current framework. Finally, we are planning to incorporate the user in a more active way during the learning process.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous referees for their helpful comments on a previous version of this chapter. The first author thanks Claude Sammut for his support and comments in the development of the flight simulator application and the support from the ARC Centre of Excellence for Autonomous Systems (CAS).

The second author was supported by a grant from CONACyT number 212418 for his Master studies. Part of this work was also supported by grant number 84162 from CONACyT.

REFERENCES

Blockeel, H., & Raedt, L. D. (1998, June). Top-down induction of logical decision trees. *Artificial Intelligence*, *101*, 285–297. doi:10.1016/S0004-3702(98)00034-4

Bratko, I., Urbančič, T., & Sammut, C. (1998). Behavioural cloning: Phenomena, results and problems. automated systems based on human skill. In *IFAC Symposium.* Berlin.

Chapman, D., & Kaelbling, L. (1991). Input generalization in delayed reinforcement learning: an algorithm and performance comparison. In *Proc. of the International Joint Conference on Artificial Intelligence* (pp. 726-731). San Francisco, CA: Morgan Kaufmann.

Charness, N. (1977). Human chess skill . In Frey, P. (Ed.), *Chess skill in man and machine* (pp. 35–53). Springer-Verlag.

Croonenborghs, T., Driessens, K., & Bruynooghe, M. (2007, June). Learning relational options for inductive transfer in relational reinforcement learning. In *Proceedings of the 17th Conference on Inductive Logic Programming* (pp. 88–97). Springer.

de Groot, A. (1965). *Thought and choice in chess*. The Hague, The Netherlands: Mouton.

Dietterich, T. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Driessens, K., & Dzeroski, S. (2002). Integrating experimentation and guidance in relational reinforcement learning. In *Proc. of the Nineteenth International Conference on Machine Learning* (p. 115-122). Morgan Kaufmann.

Driessens, K., & Ramon, J. (2003). Relational instance based regression for relational reinforcement learning. In *Proc. of the International Conference on Machine Learning (icml'03).*

Driessens, K., Ramon, J., & Blockeel, H. (2001). Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In *Proc. of the 13th. European Conference on Machine Learning (ecml-01)* (p. 97-108). Springer.

Dzeroski, S., Raedt, L. D., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, *43*(2), 5–52. doi:10.1023/A:1007694015589

Fern, A., Yoon, S., & Givan, R. (2003). Approximate policy iteration with a policy language bias. In *Proc. of the Neural and Information Processing Conference (NIPS'03).*

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, *29*(5), 1189–1232. doi:10.1214/aos/1013203451

Getoor, L., Koller, D., Taskar, B., & Friedman, N. (2000). Learning probabilistic relational models with structural uncertainty. In *Proceedings of the ICML-2000 Workshop on Attribute-Value and Relational Learning: Crossing the Boundaries* (pp. 13–20).

Getoor, L., & Taskar, B. (2007). *Introduction to statistical relational learning: Adaptive computation and machine learning*. Boston, MA: MIT Press.

Hernández, S. F., & Morales, E. F. (2006, Septemer). Global localization of mobile robots for indoor environments using natural landmarks. *Proc. of the IEEE 2006 ICRAM*, (pp. 29-30).

Herrera-Vega, J. (2009). *Mobile robot localization in topological maps using visual information*. Master's thesis (to be published).

Hinton, G. (1984). *Distributed representations* (Tech. Rep. No. CMU-CS-84-157). Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Kaelbling, L., Littman, M., & Cassandra, A. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, *101*(1-2). doi:10.1016/S0004-3702(98)00023-X

Kanerva, P. (1993). Sparse distributed memory and related models . In Hassoun, M. (Ed.), *Associate neural memories: Theory and implementation* (pp. 50–76). New York, NY: Oxford University Press.

Kersting, K., & Driessens, K. (2008, July). Non-parametric policy gradients: A unified treatment of propositional and relational domains. In *Proceedings of the 25th International Conference on Machine Learning* (vol. 307, pp. 456–463). ACM.

Kersting, K., & Raedt, L. D. (2003). Logical Markov decision programs. In *Proc. of the IJCAI-03 Workshop on Learning Statistical Models of Relational Data.*

Kersting, K., & Raedt, L. D. (2004). Logical markov decision programs and the convergence of td(λ). In *Proc. of the Conference on Inductive Logic Programming.*

Kersting, K., Raedt, L. D., & Kramer, S. (2000). Interpreting Bayesian logic programs. In *Proceedings of the Work-In-Progress Track at the 10th International Conference on Inductive Logic Programming* (pp. 138–155).

Lim, C., & Kim, H. (1991). Cmac-based adaptive critic self-lerning control. *IEEE Transactions on Neural Networks*, *2*, 530–533. doi:10.1109/72.134290

Lloyd, J. (1987). *Foundations of logic programming* (2nd ed.). Berlin, Germany: Springer-Verlag.

McDermott, D. V. (1998). *The planning domain definition language manual.* Technical Report 98-003, Department of Computer Science, Yale University.

Michie, D., Bain, M., & Hayes-Michie, J. (1990). Cognitive models from subcognitive skills . In Grimble, M., McGhee, J., & Mowforth, P. (Eds.), *Knowledge-based systems in industrial control* (pp. 71–90). Peter Peregrinus. doi:10.1049/PBCE044E_ch5

Morales, E. (1997). On learning how to play . In van den Herik, H., & Uiterwijk, J. (Eds.), *Advances in computer chess 8* (pp. 235–250). The Netherlands: Universiteit Maastricht.

Morales, E. (2003). Scaling up reinforcement learning with a relational representation. In *Proc. of the Workshop on Adaptability in Multi-Agent Systems (aorc-20 03)* (p. 15-26).

Natarajan, S., Tadepalli, P., Altendorf, E., Dietterich, T. G., Fern, A., & Restificar, A. (2005). Learning first-order probabilistic models with combining rules. In *ICML '05: Proceedings of the 22nd International Conference on Machine Learning* (pp. 609–616). New York, NY: ACM.

Poggio, T., & Girosi, F. (1990). Regularizationn algorithms for learning that are equivalent to multilayer networks. *Science*, *247*, 978–982. doi:10.1126/science.247.4945.978

Puterman, M. (1994). *Markov decision processes: Discrete stochastic dynamic programming*. New York, NY: Wiley.

Raedt, L. D., Kimmig, A., & Toivonen, H. (2007, January). Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of 20th International Joint Conference on Artificial Intelligence* (pp. 2468–2473). AAAI Press.

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine Learning*, *62*(1-2), 107–136. doi:10.1007/s10994-006-5833-1

Romero, L., Morales, E., & Sucar, L. E. (2001). An exploration and navigation approach for indoor mobile robots considering sensor's perceptual limitations. *Proc. of the IEEE ICRA*, (pp. 3092-3097).

Sammut, C., Hurst, S., Kedzier, D., & Michie, D. (1992). Learning to fly. In *Proc. of the Ninth International Conference on Machine Learning* (p. 385-393). Morgan Kaufmann.

Singh, S., Jaakkola, T., & Jordan, M. (1996). Reinforcement learning with soft state aggregation . In *Neural information processing systems 7*. Cambridge, MA: MIT Press.

Sutton, R., & Barto, A. (1989). *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press.

Sutton, R., Precup, D., & Singh, S. (1999a). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211. doi:10.1016/S0004-3702(99)00052-1

Sutton, R., Precup, D., & Singh, S. (1999b). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211. doi:10.1016/S0004-3702(99)00052-1

van Otterlo, M. (2003). Efficient reinforcement learning using relational aggregation. In *Proc of the Sixth European Workshop on Reinforcement Learning.* Nancy, France.

van Otterlo, M. (2004). Reinforcement learning for relational mdps. In A. Nowé, T. Lenaerts, & K. Steenhaut (Eds.), *Proc. of the Machine Learning Conference of Belgium and The Netherlands* (pp. 138-145).

van Otterlo, M. (2009). *The logic of adaptive behavior: Knowledge representation and algorithms for adaptive sequential decision making under uncertainty in first-order and relational domains*. The Netherlands: IOS Press.

Vaughan, R., Gerkey, B., & Howard, A. (2003). On device abstractions for portable, reusable robot code. *Proc. of the 2003 IEEE/RSJ IROS*, (pp. 11-15).

Watkins, C. (1989). *Learning from delayed rewards*. Unpublished doctoral dissertation, Cambridge University, Cambridge, MA.

Younes, H., & Littman, M. (2004). *Ppddl1.0: An extension to pddl for expressing planning domains with probabilistic effect*s. Technical Report CMU-CS-04-167, Department of Computer Science, Carnegie-Mellon University.

ENDNOTES

1 www.ida.liu.se/~TDDA13/labbar/planning/2003/writing.html

2 The flight simulator application was done while the first author was on sabbatical leave at the University of New South Wales, Australia.

3 Although this is not a strictly accurate model of turbulence, it is a reasonable approximation for these experiments.

4 The training mission involved only left turns.

5 The values used for orientation and distance are the same as with doors.

6 The values used for orientation and distance are the same as with doors.

7 The values used for orientation and distance are the same as with doors.

8 The values used for orientation and distance are the same as with doors.

9 An ActivMedia GuiaBot, www.activrobots.com

10 *NPDA*: Navigation Policy with Discrete Actions, *NPCA*: Navigation Policy with Continuous Actions, *FPDA*: Following Policy with Discrete Actions, *FPCA*: Following Policy with Continuous Actions.