Using ILP to Improve Planning in Hierarchical Reinforcement Learning

Mark Reid and Malcolm Ryan

School of Computer Science and Engineering, University of New South Wales
Sydney 2052, Australia
{mreid,malcolmr}@cse.unsw.edu.au

Abstract. Hierarchical reinforcement learning has been proposed as a solution to the problem of scaling up reinforcement learning. The RL-TOPs Hierarchical Reinforcement Learning System is an implementation of this proposal which structures an agent's sensors and actions into various levels of representation and control. Disparity between levels of representation means actions can be misused by the planning algorithm in the system. This paper reports on how ILP was used to bridge these representation gaps and shows empirically how this improved the system's performance. Also discussed are some of the problems encountered when using an ILP system in what is inherently a noisy and incremental domain.

1 Introduction

Reinforcement learning [15] has been studied for many years now as approach to learning control models for robot or software agents. It works superbly on small, low-dimensional domains but has trouble scaling up to larger, more complex, problems. These larger problems have correspondingly larger state spaces which means random exploration and reward back-propagation – the key techniques in reinforcement learning – are much less effective.

Hierarchical reinforcement learning has been proposed as a solution to this lack of scalability and comes in several forms ([12], [16], [11], [3]). The overarching idea is to break up a large reinforcement learning task into smaller subtasks, find policies for the subtasks, then finally recombine them into a solution for the original, larger problem. The approach examined in this paper, the RL-TOPs Hierarchical Reinforcement Learning System[13], represents the agent's state symbolically at various levels of abstraction. This allows for a unique synthesis of reinforcement learning and symbolic planning.

Low-level state representation is ideal when an agent requires a fine-grained view of its world (eg, motor control in robot walking and balancing). In order to make larger task tractable, however, a higher level of abstraction is sometimes required (eg, moving the robot through several rooms in an office block). In the RL-TOPs system the high-level representation of a problem is provided by a domain expert who defines coarse-grained actions and states for the agent. The

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 174–190, 2000.
© Springer-Verlag Berlin Heidelberg 2000

low-level implementation of the coarse actions are left to the agent to invent using reinforcement learning.

The high-level states and actions do not always convey every relevant feature of an agent's state space to the planning side of the RL-TOP system. The work presented in this paper shows that by examining the interplay between the agent's low-level and high-level actions, new high-level features can be constructed using a standard ILP algorithm.

The paper is organized as follows. Section 2 gives an overview of the RL-TOP system and the planning problem motivating the use of ILP to learn new state space features. Section 3 details the transformation of the planning problem into an ILP learning task as well as outlining a method for converting a batch learning algorithm into an incremental learner. Finally, Section 4 describes a simple domain used to test the performance of the RL-TOPs system augmented with an ILP system. An experiment comparing the performance of three reinforcement learning approaches on this domain is then analyzed.

2 Reinforcement-Learnt Teleo-Operators

The key concept in the hierarchical reinforcement system presented in this paper is the *Reinforcement-Learnt Teleo-OPerator* or RL-TOP[12]. It is a synthesis of ideas from planning and reinforcement learning. Like Nilsson's teleo-operators (TOPs)[10], RL-TOPs define agent behaviours by symbolically describing their preconditions and effects. The main advantage RL-TOPs have over standard TOPs is that the agent does not need hard-coded instructions on how to carry out each of its behaviours. Instead, the problem of moving from the set of states defined by an RL-TOP's precondition to the states defined by its intended effect is treated as a reinforcement learning task. This allows a large reinforcement learning task to be broken down into several easier ones which are combined together by a symbolic planner.

The use of RL-TOPs in the system presented in this paper is analogous to the use of *options* as described by Sutton et al in [15], the *Q nodes* of Dietterich's MAXQ system [3] as well as *abstract actions* and *macro-actions* described elsewhere in the literature. The RL-TOP approach distinguishes itself from these other systems by emphasizing the symbolic representation of agent behaviours.

2.1 Definitions and Notation

At any point in time an agent is assumed to be in some *state* $s \in \mathcal{S}$. To move from one state to another an agent has a finite number of *actions* $\mathcal{A} = \{a_1, \ldots, a_k\}$ which are functions from $\mathcal{S}$ to $\mathcal{S}$. Actions need not be deterministic.

Given a set of *goal states* $\mathtt{G} \subset \mathcal{S}$ a reinforcement learning task requires an agent to come up with a *policy* $\pi : \mathcal{S} \to \mathcal{A}$. A *good* policy is one that can take an agent from any initial state, $s_0 \in \mathcal{S}$, through a sequence of steps, $s_0, \ldots, s_n$, such that $s_n \in \mathtt{G}$. A *step* is made from s_i to s_{i+1} by applying the action $\pi(s_i)$

to the state s_i yielding s_{i+1}. An *optimal* policy is one that for any initial state generates the shortest possible sequence of steps to a goal state.

An RL-TOP or *behaviour* B consists of three components, a *precondition*, a *policy*, and a *postcondition* (or effect). The precondition, denoted B.pre, is a set of states in which the behaviour is applicable. The postcondition, B.post, is another set of states which define a behaviour's intended effect. An agent's behaviour is executed by using its policy, B.π, to move from state to state until the agent is no longer in that behaviour's precondition. If, when the execution of the behaviour terminates, the agent is in B.post the behaviour was said to be *successful.* Behaviours are given to the agent by defining its pre- and post-conditions. Finding good policies for each behaviour then becomes a standard reinforcement learning task – steps that lead to a successful execution of a behaviour are rewarded while steps that leave B.pre without ending up in B.post are punished. The implementation details of this reinforcement learning is not sufficiently pertinent to warrant discussion in this paper. We point the interested reader to [13] and [12].

A finite set of predicates, $\mathcal{P} = \{\texttt{P}_1, \ldots, \texttt{P}_m\}$, is called *primitive* if every state $s \in \mathcal{S}$ can be uniquely identified with a conjunction of ground instances of these predicates. A primitive set of predicates allows subsets of the state space to be described by new predicates defined in terms of disjunctions of conjunctions built from elements of $\mathcal{P}$. We call these non-primitive predicates *high-level* predicates. In the remainder of this discussion it will be assumed that goals, preconditions, postconditions and other subsets of $\mathcal{S}$ are defined by high-level predicates.

2.2 Planning and the Frame Axiom

The symbolic representation of behaviours' preconditions and postconditions provides the link between planning and reinforcement learning. As sets they define small reinforcement learning tasks and symbolically it becomes possible to build *Teleo-Reactive (TR) plan trees.* We will briefly explain their construction and limitations through a few examples. A more detailed exposition can be found in [2].

Suppose an agent has a goal G and four behaviours: A, B, C, and D. The left-hand side of Figure 1 is an abstract representation of the agent's state space showing the set of goal states as a shaded rectangle and behaviours as labelled arrows connecting their pre- and postcondition sets.

Since, as the figure shows, that A.post $\subset$ G, it must be the case if the agent is in A.pre and successfully executes behaviour A it will achieve the goal G. It is important to note that this subset test is actually performed in the planning algorithm by testing if the high-level predicate for G subsumes that for A.pre. The problem of reaching a state in G has now been reduced to getting to a state in G *or* getting to a state in A.pre and executing behaviour A. This process is repeated with A.pre as the new goal and B.pre $\cup$ C.pre $\cup$ D.pre are then found to be states from which it is possible to achieve the goal G. The TR tree on the right of the figure shows which actions the agent needs to successfully execute to reach the goal. For example, if the agent state is in the set defined by D.pre it

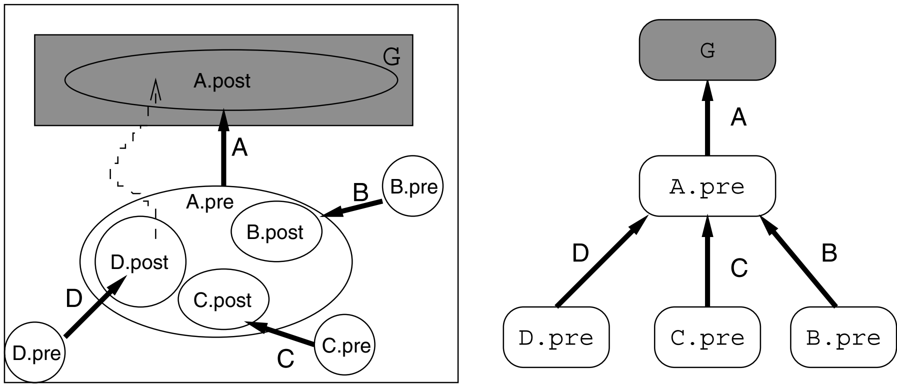

Fig. 1. The diagram on the left shows four behaviours and a goal in a state space. On the right is the corresponding TR plan tree. The dashed line represents a sequence of steps which successfully execute behaviour A

needs to execute behaviour D followed by behaviour A. If an agent moves from a node of a TR tree into a node which is not the one that was expected a *plan failure* is said to have occurred.

The above process for generating TR trees works if the goal state completely subsumes some behaviour's postcondition. The goal in Figure 2, however, is a conjunction of two predicates, $G_1 \wedge G_2$, and only one of them, G_1, subsumes A's postcondition. This means that there are states in A.pre that may not be mapped into goal states when A is executed. As behaviours can be quite complicated it is not at all clear which states in A.pre the agent must start in to ensure it ends up in $G_1 \wedge G_2$ after executing A.

The simplest assumption that can be made in this situation is that executing a behaviour will only make those changes to the agent's state which are needed to get from the preconditions of the behaviour to the postconditions. This assumption is known as the *frame axiom*. Provided this criteria is met we can use it to restrict a behaviour's precondition by conjuncting it with those goal conditions which did not subsume the behaviour's post condition. In Figure 2 A.pre is intersected with G_2. Behaviour D is removed from the TR tree as its postcondition is no longer contained within the states thought to get the agent to the goal. The frame axiom is used again to propagate the G_2 condition through behaviour C but is not needed for B since B.post is contained within G_2.

2.3 Side Effects

The frame axiom does not always hold, especially in complex domains. Primarily, this is because the domain expert who provides the behaviour definitions to the agent cannot always foresee how they will affect the agent's state. If the successful execution of behaviour A from a state $s_0 \in \mathsf{A.pre} \cap G_i$ terminates in

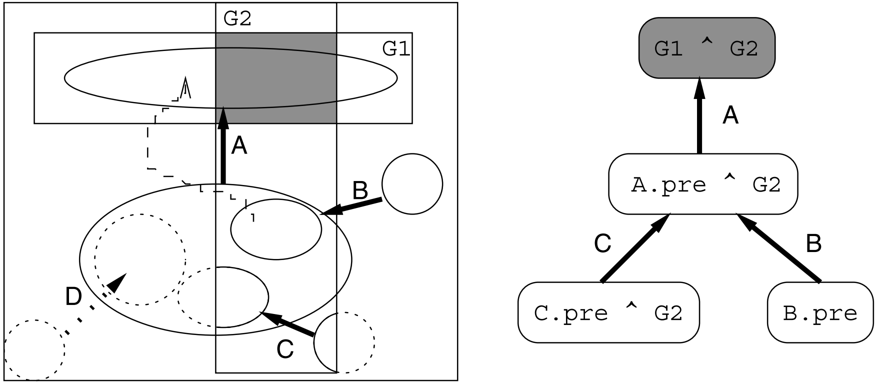

Fig. 2. An example of the frame axiom. Only $\mathtt{G}_1$ subsumes $\mathsf{A}.\mathtt{post}$ so the second goal condition $\mathtt{G}_2$ must be propagated down the TR tree. The dashed line represents a sequence of steps which violate the frame axiom

a state $s_t \notin \mathtt{G}_i$ we say that executing A when in state s_0 *causes a side effect on* $\mathtt{G}_i$. This is denoted $\mathtt{cause}(s_0, \mathsf{A}, \mathtt{G}_i)$. An example of a side effect is shown as the dashed line in Figure 2.

In order to make effective plans the agent needs to know which states in a behaviour's precondition will cause a certain side effect. Precisely, the agent would need some description of the sets

$$\mathtt{cause}(\mathsf{A}, \mathtt{G}_i) = \{s \in \mathcal{S} : \mathtt{cause}(s, \mathsf{A}, \mathtt{G}_i)\}$$

for every behaviour B and every goal condition $\mathtt{G}_i$. In Figure 3 the shaded set labelled $\mathtt{cause}$ is the side effect $\mathtt{cause}(\mathsf{A}, \mathtt{G}_2)$. Since the postcondition of behaviour B falls entirely within this set there is no point considering plans where the agent begins in $\mathsf{B}.\mathtt{pre}$. The agent's only option from these states is to execute behaviour B which will terminate at one of the side effect states. By definition, behaviour A will move the agent outside of the set $\mathtt{G}_2$ and hence outside the goal. The only states which will get the agent to the goal using A are those in $\mathsf{A} \wedge \mathtt{G}_2 \wedge \neg\mathtt{cause}(\mathsf{A}, \mathtt{G}_2)$. This situation is reflected in the new TR tree in the right of the figure.

The simplest approach to constructing the side effect sets is by adding states as they are recognized as causing a side effect. This approach has two problems. Firstly, since the behaviour's policies are being reinforcement learnt as they are used, side effects may be generated due to a bad policy rather than a true side effect of the ideal behaviour. This noise can mean the side effects are over-general which in turn may prevent good planning. The second and more serious difficulty with this method is that only those states already seen to have caused a side effect will be avoided by the agent in the future. If there are regularities across the states causing a side effect we would like the agent to be able exploit them

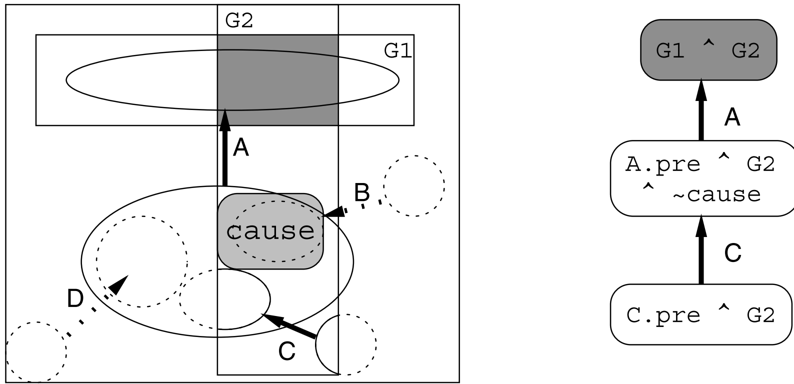

Fig. 3. The side effect set $\texttt{cause}(\mathsf{A}, \texttt{G}_2)$ is shown as the shaded area labelled `cause` on the left. The revised TR tree is shown on the right

and avoid those states before having to test them explicitly. This is the focus of the next section.

3 Learning Side Effects Using ILP

In his system TRAIL, Benson describes the first application of Inductive Logic Programming to the problem of action model learning [2]. In his model an agent is endowed with actions in the form of TOPs. When these are given to an agent the policies and postconditions of the TOPs are fixed and it is up to the agent to determine satisfactory preconditions. The TRAIL system employs a DINUS-like[4] algorithm to induce the preconditions from examples generated from plan failures, successes and a human teacher. Side effects are recognized as a problem in TRAIL but are only treated lightly through simple statistical methods.

In the RL-TOP system the focus is on learning the policy for each behaviour while the pre- and postconditions are assumed to be correct and fixed. Once the behaviours' policies are learnt sufficiently well side effects become the primary source of an agent's poor performance. Our hypothesis for this paper is that ILP can be used to improve an agent's performance by inducing descriptions for side effects in a manner similar to the way TRAIL uses ILP to learn TOP preconditions.

The ILP learning task will be to construct definitions for the predicates $\texttt{cause}(s, \mathsf{A}, \texttt{G}_i)$ for various values of A and $\texttt{G}_i$ as they are required. Once these predicates are learnt the sets $\texttt{cause}(\mathsf{A}, \texttt{G}_i)$ can be incorporated into an agent's TR tree using the $\texttt{cause}/3$ predicate as a test for a state's membership in a side effect set.

3.1 Examples and Background Knowledge

Recall that any state an agent is in can be uniquely described in terms of the primitive predicates $\mathcal{P} = \{\mathtt{P}_1, \ldots, \mathtt{P}_m\}$. In order to record the history of an agent, the first argument of each $\mathtt{P}_i$ will hold a unique state identifier such as the number of steps the agent has taken since the start of some learning task. High-level predicates will also be modified in this way.

Given a sequence of agent steps $s_0, \ldots, s_n$, examples of $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_i)$ can be found by locating a subsequence $s_a, \ldots, s_b$ such that for all $k = a, \ldots, b-1$ each state $s_k \in \mathsf{A}.\mathtt{pre} \wedge \mathtt{G}_i$ and $s_b \notin \mathtt{G}_i$. Since each of the states in the subsequence is one from which A was executed and resulted in $\mathtt{G}_i$ no longer holding, each $\mathtt{cause}(s_k, \mathsf{A}, \mathtt{G}_i)$ is a positive example of the side effect. To get negative examples we need to find states for which the execution of A resulted in $\mathtt{G}_i$ holding true in $\mathsf{A}.\mathtt{post}$. All the states in a subsequence $s_a, \ldots, s_b$ such that $s_k \in \mathsf{A}.\mathtt{pre} \wedge \mathtt{G}_i$ for $k = a, \ldots, b-1$ and $s_b \in \mathsf{A}.\mathtt{post} \wedge \mathtt{G}_i$ are negative examples of $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_i)$.

As an illustration a hypothetical agent history is shown in Table 1. States 2,3 and 4 are positive examples of the side effect $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_2)$ while state 6 is a negative example.

Table 1. An example history trace using the TR tree from Figure 2. Steps marked with '+' or '-' are positive and negative examples of $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_2)$, respectively

Step	Primitive Description	High-level Predicates	Behaviour	Action
0	$\mathtt{P}_1(0,\ldots) \wedge \ldots \wedge \mathtt{P}_m(0,\ldots)$	$\mathsf{B}.\mathtt{pre(0)}$	B	a_3
1	$\mathtt{P}_1(1,\ldots) \wedge \ldots \wedge \mathtt{P}_m(1,\ldots)$	$\mathsf{B}.\mathtt{pre(1)}$	B	a_7
+2	$\mathtt{P}_1(2,\ldots) \wedge \ldots \wedge \mathtt{P}_m(2,\ldots)$	$\mathsf{B}.\mathtt{post(2)} \wedge \mathsf{A}.\mathtt{pre(2)} \wedge \mathtt{G}_2(2)$	A	a_3
+3	$\mathtt{P}_1(3,\ldots) \wedge \ldots \wedge \mathtt{P}_m(3,\ldots)$	$\mathsf{A}.\mathtt{pre(3)} \wedge \mathtt{G}_2(3)$	A	a_2
+4	$\mathtt{P}_1(4,\ldots) \wedge \ldots \wedge \mathtt{P}_m(4,\ldots)$	$\mathsf{A}.\mathtt{pre(4)} \wedge \mathtt{G}_2(4)$	A	a_5
5	$\mathtt{P}_1(5,\ldots) \wedge \ldots \wedge \mathtt{P}_m(5,\ldots)$	$\mathsf{A}.\mathtt{pre(5)}$	A	a_3
−6	$\mathtt{P}_1(6,\ldots) \wedge \ldots \wedge \mathtt{P}_m(6,\ldots)$	$\mathsf{A}.\mathtt{pre(6)} \wedge \mathtt{G}_2(6)$	A	a_6
7	$\mathtt{P}_1(7,\ldots) \wedge \ldots \wedge \mathtt{P}_m(7,\ldots)$	$\mathsf{A}.\mathtt{post}(7) \wedge \mathtt{G}_2(7) \wedge \mathtt{G}_1(7)$	A	—

The background knowledge for this learning task should, at the very least, consist of all those primitive predicates used to describe the agent's state in each of the examples of the side effect to be learnt. The high-level predicates true in the example states may also be useful when learning a side effect. Whether or not any additional relations are required will depend on the agent's domain. In the above example $\mathtt{cause}(3, \mathsf{A}, \mathtt{G}_2)$ is a positive instance so at least each $\mathtt{P}_i(3,\ldots)$ should appear as background knowledge whereas the high-level predicates $\mathsf{A}.\mathtt{pre}(3)$ and $\mathtt{G}_2(3)$ may or may not be useful.

3.2 Implementation

There are three parts to the RL-TOP hierarchical reinforcement learning system: an *actor*, a *planner* and a *reflector*.[1] Using the high-level view afforded by the RL-TOPs the planner builds a TR plan tree which is passed on to the actor. The plan tree is essentially a description of sequences of behaviours the actor should perform to reach the goal from various states in the state space.

The actor is the agent's interface with the primitive level of the domain. After using the TR tree to determine which behaviour is most appropriate, the agent executes and reinforces its policy. Side effects the agent encounters while executing a behaviour are indicative of a problem with the actor's TR tree. When they occur the actor keeps a record of positive and negative instances of them and hands these examples to the reflector.

The reflector's role is to induce descriptions of which states cause side effects from the examples handed to it from the actor along with any background knowledge it may have been given. Once a side effect has been learnt its description is passed to the planner. The planner then builds a new TR tree and passes it to the actor closing the learning cycle.

The induction of side effect descriptions in the reflector is performed by the ILP system LIME [9]. The deciding factors used to make this choice were speed, noise resistance and a familiarity with the system.[2] It is important the reflector be reasonably quick since each of the three components of the system run asynchronously. If the reflector runs too slowly the actor and planner will continue controlling the agent poorly defeating the purpose of learning side effects in the first place. Also, the examples given to the reflector suffer from a unusual form of noise. When the agent begins its learning task its policies are random. This means initially that examples of a behaviours' side effects are most likely due to poor execution. As the agent steadily improves its policies the proportion of noisy examples decreases. LIME proved to be quite robust under these conditions in addition to being fast enough for our requirements.

3.3 Batch Learning from Incrementally Generated Examples

Reinforcement learning relies heavily on repetition. In order to learn a control model for a problem an agent must repeatedly attempt to reach a set goal. Side effect examples are therefore generated by the actor incrementally. Since LIME is a batch learner we will briefly discuss a method of converting an incremental learning problem into a batch learning problem.

Each time a new side effect is encountered by the actor a *example pool* for that side effect is created. As examples of the side effect are generated they are added to the pool. Before any induction is performed the pool must contain at least E^{+}_{min} positive and E^{-}_{min} negative examples. This is to stop the reflector

[1] This is the part of the system which looks back, or *reflects*, on the actions and plans of the other two parts

[2] FOIL and PROGOL were also briefly tried as induction engines for the reflector however LIME showed more promise during our early tests

attempting to induce side effects from insufficient data. Once these lower limits are both met E^+_{min} positive and E^-_{min} negative examples are randomly sampled from those in the pool and passed to LIME.

As the reflector can only learn one side effect at a time those side effects with a sufficient number of examples are queued. Each time a side effect is learnt it is placed directly on the back of the queue to be re-examined at a later date. This process is an attempt to spread the reflector's attention evenly over its various tasks. When a side effect is learnt for the second time a choice must be made to keep the new definition or discard it in favour of the old. In this situation both versions are tested for accuracy on all the examples currently in the side effect's pool and the better one is kept.

The pool also has upper limits to the number of examples it can hold: E^+_{max} and E^-_{max} specify the maximum number of positive and negative examples that can be stored. When one of these maxima is reached any further example of the same sign randomly replaces a like example already in the pool.

When a side effect gets used by the planner the agent will avoid the area it defines. This means there will be a sudden change in the distribution of examples for that side effect. In particular, the number of positive examples generated for a side effect tends to drop drastically after it is learnt for the first time. The random replacement of old examples with new ones is an attempt to smooth out any drastic changes in the example distributions.

The implementation of the reflector used in the experiment described in Section 4 had example pools defined by $E^+_{min} = 100$, $E^-_{min} = 1000$, $E^+_{max} = 1000$, and $E^-_{max} = 10000$. Having ten times as many negative examples as positive was made to help prevent overgeneralisation as well as reflect the actual ratio of examples being generated.

4 Experimental Results

Standard, non-hierarchical reinforcement learning techniques, such as Q-learning, can be shown to converge to optimal policies, given some fairly natural assumptions [15]. This convergence, especially for large state spaces, can be very slow. Hierarchical reinforcement learning trades optimality for speed hoping to find good solutions quickly by breaking a monolithic reinforcement learning task into smaller ones and combining their solutions. Early versions of the RL-TOP hierarchical reinforcement learning system did not have a reflector and so could not identify and avoid side effects. It was believed that this was the cause of some of the early system's sub-optimality. The experiment reported in this section was designed to test if adding the ability to recognize side effects meant the RL-TOP system could match the performance of monolithic reinforcement learning in the long term while retaining the speed gained through the hierarchical approach.

4.1 The Mudworld

The test domain for our experiment, dubbed "The Mudworld", was chosen to best illustrate the impact of side effects on the agent's performance. It is small

enough to be monolithically reinforcement learnt in a reasonable amount of time but large enough for hierarchical techniques to show a speed improvement. The other advantage to using this artificial domain is that the side effects the agent will encounter are quite obvious. The side effect descriptions generated by the reflector can therefore be checked for their pertinence.

As seen in Figure 4 the Mudworld lies on a 60-by-60 grid which is partitioned into seven rooms. The agent's task in this domain is to build a control model which will move it from any starting position in Room 2 to any square Room 6 without being muddy. The agent becomes muddy by stepping into the rectangular patch of mud in Room 4. Once this happens the agent can only become clean again by stepping into the square of water in Room 6. If the agent avoids the mud entirely as it moves from Room 2 to Room 6 (via Rooms 1, 4 and 5, for example) the goal condition of being clean in Room 6 is met as soon as the agent is inside the door to Room 6. If the agent moves through the mud (by choosing to go through Rooms 1, 4 and 7) it does not reach a goal state until it steps in the water near the centre of Room 6. This ensures there is a performance penalty of around 20 steps for a bad choice of route from Room 2 to Room 6.

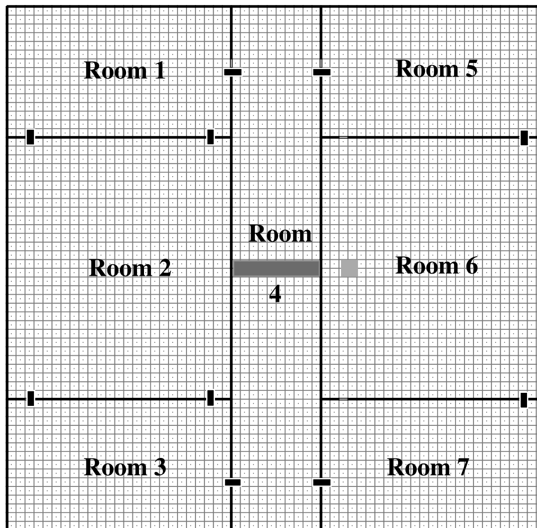

Fig. 4. The Mudworld consists of 7 rooms, connected by doors, on a 60 by 60 grid. The top left of the grid is position (0,0). There is mud in Room 4 (dark grey strip) which can only be cleaned off by the water in Room 6 (light grey square). The agent is started in a random position in Room 2 and its goal is to be clean in Room 6

The agent's primitive state in the Mudworld is described by two primitive predicates: $\texttt{position}(S, X, Y)$ specifying the agent's X and Y coordinate on the grid in state S, and $\texttt{muddy}(S)$ which is true only when the agent is muddy in state S. The agent's primitive actions enable it to move from a grid position to any of its eight adjacent grid positions, walls permitting.

A high-level predicate $\texttt{in_room}(S, R)$ is defined using the primitive predicate $\texttt{position}/3$ and the "less than" relation <. The clause defining all the states in Room 1 is:

$$\texttt{in_room}(S, 1) \quad \texttt{:-} \quad \texttt{position}(S, X, Y), \quad X < 25, \quad Y < 15.$$

The other rooms are defined similarly. Sixteen RL-TOPs are given to the agent specifying the pre- and postconditions for behaviours which move the agent from one room into an adjacent room. All of these behaviours can be summed up with the following template:

$$\texttt{go}(R_1, R_2).\texttt{pre} = \texttt{in_room}(R_1)$$
$$\texttt{go}(R_1, R_2).\texttt{post} = \texttt{in_room}(R_2)$$

The mud in the middle of Room 4 makes planning difficult with the above RL-TOPs. Every plan capable of getting the agent from the left-hand side of the map to the right-hand side must, at some point, pass through Room 4 into either Room 5 or Room 7. Since the only high-level predicate the planner has to describe the agent's state in Room 4 is $\texttt{in_room}(4)$ it has to make a choice between telling the agent to use $\texttt{go}(4, 5)$ or $\texttt{go}(4, 7)$ at this point in the plan. Neither of these behaviours are ideal. If the agent is in the upper half of Room 4 executing the $\texttt{go}(4, 7)$ behaviour will move the agent through the mud causing a side effect. Similarly, if the agent is the the lower half of Room 4 the $\texttt{go}(4, 5)$ behaviour will cause a side effect.

In a sense, the planner's high-level representation of the agent's state is not rich enough to come up with a plan that will always avoid getting muddy. The job of the reflector then is to define new high-level terms which give the planner a better description of the agent's world. The definitions that will do this job in the above situation are:

$$\texttt{cause}(S, \texttt{go}(4, 5), \texttt{muddy}) \quad :- \quad \texttt{position}(S, _, Y), \quad 30 < Y.$$
$$\texttt{cause}(S, \texttt{go}(4, 7), \texttt{muddy}) \quad :- \quad \texttt{position}(S, _, Y), \quad Y < 29.$$

For comparison, Figure 5 shows a set of clauses the reflector had generated for the planner at the end of one of experiments described in the next section. As can be seen the last two clauses in the figure are quite similar to the ideal definitions given above although the last clause is a little over-general. The first two clauses in the figure describe "causes of causes". For example, the first clause states that if an agent is going out of Room 1 it will be in the side effect area for getting muddy when executing go(room(4), room(7)).

```
cause(A, go(room(1),_), cause(go(room(4),room(7)),muddy)) :-
        in_room(A, room(1)).
cause(A, go(room(3),_), cause(go(room(4),room(5)),muddy)) :-
        in_room(A, room(3)).
cause(A, go(B,room(5)), muddy) :-
        in_room(A, B),
        satisfy:not(user:muddy(A)),
        position(A, _, C),
        greater_than(A, C, 30).
cause(A, go(B,room(7)), muddy) :-
        in_room(A, B),
        satisfy:not(user:muddy(A)),
        position(A, _, C),
        less_than(A, C, 38).
```

Fig. 5. Output from the reflector after one run of the experiment. The planner uses these definitions to rebuild the TR tree

4.2 Experimental Setup

A standard measure of agent performance in reinforcement learning is *trial length*. In this experiment a trial begins with the agent starting clean and at some random position in Room 2. The trial comes to an end when the agent is clean and in Room 6. The number of primitive actions, or *steps*, the agent takes to reach the goal from the starting state is the trial's length. After each trial the agent updates its policies, behaviours and plans if necessary and is restarted. In order to assess an agent's long term performance it needs to be allowed to improve over many trials. We therefore define a *run* to be a sequence of 10000 consecutive trials.

Our Mudworld experiment compares three different *approaches*: a monolithic reinforcement learning task in which a single policy is learnt for the entire Mudworld (the MRL approach), an RL-TOPs-based hierarchical reinforcement learning approach that does not use a reflector (the RHRL approach), and the same behaviour-based approach incorporating a reflector to learn side effects (the RHRL+R approach). The testing of each approach consists of twenty independent runs. When we speak of *average trial length* the averaging is done across these twenty runs.

The results of this experiment are summarized in Figure 6. The agent's primitive actions are considered to be atomic and so are used as a unit of time along the horizontal axis. Each curve shows how the average trial length changes over one run of an approach. Each point's horizontal coordinate represents when one (average) trial ends and the next begins. Thus, the right-hand end of each curve gives an indication of how long it takes each approach to perform 10000 trials. The vertical value of each point would ideally show the average trial length but this value is far too erratic, especially in the early stages of a run. To get a clearer, qualitative picture the curve is smoothed by averaging the average trial

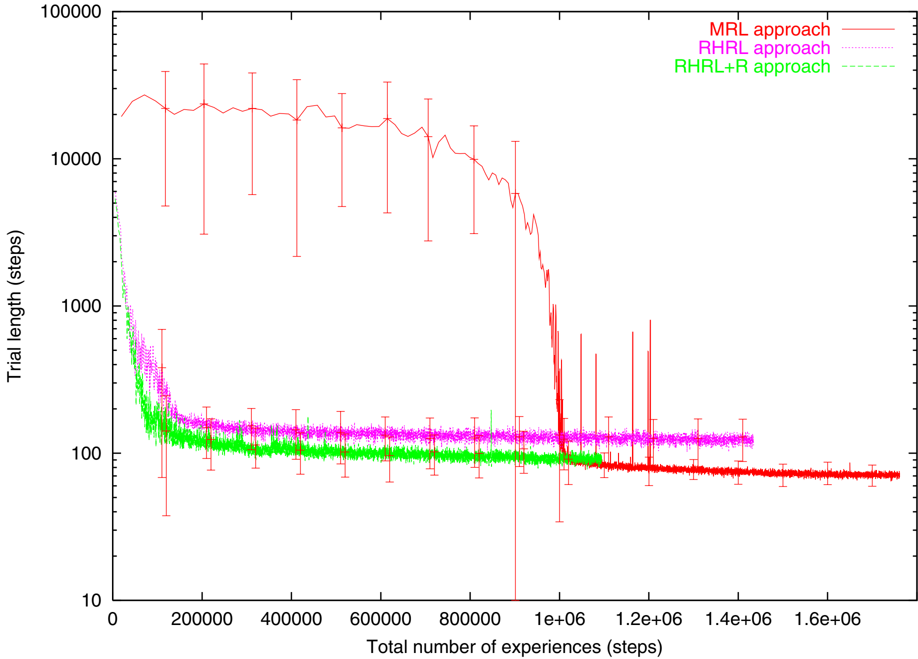

Fig. 6. A comparison of three approaches to learning a control model for a Mudworld agent. Each curve shows how average trial length decreases as the agent becomes more experienced over time

length over a 100-point window about each point. For further clarity the error bars, representing one standard deviation, are only shown every 100000 points.

4.3 Analysis and Conclusions

The most interesting features in Figure 6, in light of our hypothesis, are the relative performances of the three approaches at the end of a run and their convergence rates.

The two hierarchical learning approaches, RHRL and RHRL+R, both show a large improvement in convergence rate compared to the MRL approach which took almost five times as many steps to reach the same level of performance. While the addition of a reflector to the RL-TOPs system was not detrimental to its convergence rate it is unclear from the current data whether or not it offers any improvement.

Figure 7 compares the performance of each of the three approaches after they have converged. Each point on the graph shows the *mean final trial length* for each run in the experiment (20 runs for each of the MRL, RHRL and RHRL+R approaches). It is calculated by averaging the number of steps per trial over the

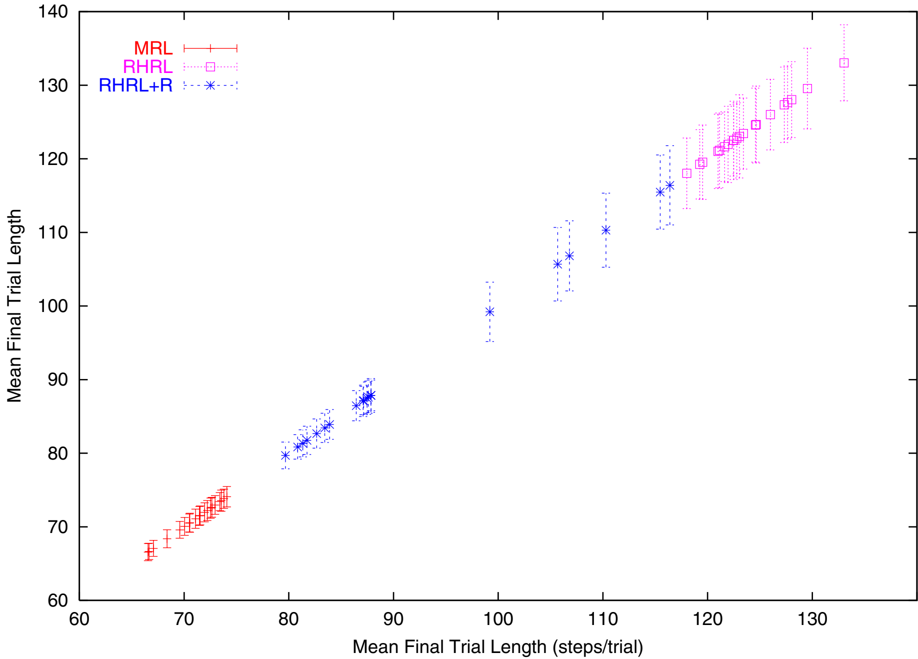

Fig. 7. The mean final trial length (averaged over the final 500 trials) shown for each approach's 20 runs. All of the MRL runs have a final average between 65 and 75 steps per trial. RHRL and RHRL+R runs lie in the ranges 80–117 and 118–135 respectively. The error bars show 99% confidence intervals

last 500 trials in a run. We are assuming that there is no significant improvement taking place between these final trials and therefore the mean is essentially over 500 independent trials of a well-trained agent.

As expected, the performance of monolithic reinforcement learning (MRL) was significantly better than either of the other two approaches. All of its twenty runs had final trial lengths between 65 and 75 steps per trial while the best RHRL+R and RHRL runs were at 80 and 118 steps per trial respectively. A striking feature of the graph is the spread of final trial lengths for the RHRL+R approach. Fourteen of the RHRL+R runs had final trial lengths between 80 and 90 steps per trial while three of its runs were statistically indistinguishable from the best RHRL run at 99% significance. The spread of results for the RHRL+R approach is due to the reflector not always inducing good definitions for side effects. We will be looking into ways of addressing this problem in the future.

As can be seen, there is some sacrifice of optimality for better convergence speed. Even the best RHRL+R run had a higher mean final trial length than the MRL approach. This is because in the reinforcement learning in the RHRL+R approach improves agent policy locally whereas the MRL can optimize over the

entire problem. For example, if the agent starts in the middle left of Room 2 the best policy for the long-term goal (being in Room 6 without being muddy) is to head diagonally towards the door closest to Room 4. However, there is a different best policy for the short-term goal (moving into Room 1). Namely, going through the leftmost door. These inefficiencies are slight compared to the large gain in convergence speed afforded by RHRL+R.

We consider these results to be evidence supporting our hypothesis: that using ILP to predict side effects can improve the long-term performance of the RL-TOP hierarchical reinforcement learning system. This performance is, more often than not, much better than the long-term performance of hierarchical reinforcement learning without a reflector and is comparable to the performance of monolithic reinforcement learning. Furthermore, both heirarchical reinforcement learning methods show a drastic improvement over standard reinforcement in the time it takes to converge to a good policy.

5 Conclusions, Discussion, and Related Work

The work presented in this paper is synthesis of ideas from several areas in artificial intelligence. By representing an agent's state symbolically RL-TOPs are able to forge a link between reinforcement learning and planning. This link is not perfect, however, since the acting and planning sides of an RL-TOP agent represent its world at different levels of abstraction. Watching what happens at these different levels simultaneously is essential to an agent's overall improvement. Inductive logic programming realizes this improvement by allowing the agent to construct new state abstractions which can be used to avoid side effects during the execution of its plans.

This is not the first time ILP has been used in conjunction with reinforcement learning or planning. In [5], Džeroski et al introduce relational reinforcement learning and show that a simple planning task – block's world – can be solved in their framework. The use of ILP in their work differs from that presented here. Rather than inducing descriptions for subsets of the state space as we do here, relational reinforcement learning uses ILP techniques to help learn a Q-tree, a variation of classic reinforcement learning's Q-function.

Inductive and analytic learning algorithms have been used in the past to improve planning and problem-solving. A good overview of work in this area can be found in [8]. However, the focus of much of this is on learning how to improve search heuristics used to generate plans which will take an agent from its current state to a goal state. When generating TR trees in the RL-TOPs system, the planner does not assume the agent is in any particular starting state. Instead, it generates many plans which lead to the goal from various parts of the state space. The aim of this search tree is to be as exhaustive as possible so learning heuristics to prune the search would not be useful in the present system.

Early work in action-model learning in planning, such as Shen's LIVE system [14] and Gil's EXPO system [6], used inductive learning algorithms to refine models of action pre- and postconditions. Their work is restricted to domains

in which the agent's actions must be atomic and deterministic. These assumptions cannot hold in our RL-TOP framework. Firstly, our agent's behaviours are durative since they are executed by following policies involving many atomic actions. Also, these behaviours are inherently non-deterministic since they are being continually improved by reinforcement learning techniques.

As mentioned in Section 3 our system draws inspiration from Benson's TRAIL system for learning action models for autonomous agents in which ILP is used to find preconditions for TOPs. Benson's work addresses some of the short-comings in Shen and Gil's systems as actions which are non-deterministic and durative (instead of atomic) can be modeled in TRAIL.

Unlike all of these systems that combine learning and symbolic planning, the basic planning unit in our system, the RL-TOP, becomes more effective by improving the execution of its behaviour. Since the pre- and postconditions of each behaviour are given in advance and fixed we have adapted some of the ILP ideas found in Benson's work to the problem of inducing side effect definitions. The result is an agent with finely-tuned, reusable, high level actions that enable it to move between fixed, user-defined portions of its state space. Each action's collection of side effects describes how subsets of its pre- and postconditions are related by its policy. With this more detailed view of its actions the agent is able to plan their use more effectively.

The ILP system Lime [9] was chosen to implement the side effect learning in our system due to its noise handling ability and speed. One difficultly with this choice that had to be overcome was that of using a batch learner in an incremental setting. Our fairly straightforward solution proposed in this paper was to pool examples as they were presented incrementally. Once enough examples were collected they could be passed onto Lime as a batch.

The noise resistant, incremental ILP system HILLARY[7] was brought to our attention some time after we were successfully using our example pool approach with Lime. Once we have a running version of HILLARY we hope to compare its performance as a reflector against our present setup.

Other future work will include applying side effect learning to domains more complicated than the Mudworld task presented here. Our long term goal and original motivation for side effect learning is the "Learning to Fly" behavioral cloning task[13]. Although the RL-TOP system has had some success with learning to fly, finding an ILP system capable of handling the complexity of the side effects involved will not be easy. We will also be investigating whether side effect learning is relevant to other hierarchical reinforcement learning systems (eg, [16], [11], [3]).

6 Acknowledgements

The authors would like to thank several people for their help during the writing of this paper. Mike Bain helped clarify some of the ideas during the early stages of this paper and Eric McCreath was always happy to offer assistance with his ILP system Lime. The experiments performed for this paper would have

been much more time-consuming were it not for Waleed Kadous's set up and support of CSE's Beowulf cluster and Virginia Wheway's advice on some of the statistical analysis. The anonymous reviewers also provided us with a number of good references and suggestions.

The first author was supported by an Australian Postgraduate Award while working on this paper.

References

[1] *Proceedings of the 15th International Conference on Machine Learning*. Morgan Kaufmann, 1998.

[2] Scott Benson. *Learning Action Models for Reactive Autonomous Agents*. PhD thesis, Department of Computer Science, Stanford University, 1996.

[3] Thomas G. Dietterich. The maxq method for hierarchical reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning* [1].

[4] S. Džeroski, S. Muggleton, and S. Russel. PAC learnability of determinate logic programs. In *Proceeding of the Fifth ACM Workshop on Computational Learning Theory*, pages 128–135, 1992.

[5] Sašo Džeroski, Luc De Raedt, and Hendrik Blockeel. Relational reinforcement learning. In *Proceedings of the 8th International Workshop on Inductive Logic Programming*, pages 11–22, 1998.

[6] Yolanda Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the 11th International Workshop on Machine Learning*, 1994.

[7] Wayne Iba, James Wogulis, and Pat Langley. Trading off simplicity and coverage in incremental concept learning. In *Proceedings of the 5th International Conference on Machine Learning*, pages 73–79, 1988.

[8] Pat Langley. *Elements of Machine Learning*. Morgan Kaufmann, 1996.

[9] E. McCreath and A. Sharma. Lime: A system for learning relations. In *The 9th International Workshop on Algorithmic Learning Theory*. Springer-Verlag, October 1998.

[10] N. J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[11] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10: Proceedings of the 1997 Conference*, 1998.

[12] Malcolm R. K. Ryan and Mark D. Pendrith. RL-TOPS: An architecture for modularity and re-use in reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning* [1].

[13] Malcolm R. K. Ryan and Mark Reid. Learning to fly: An application of hierarchical reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*. Morgan Kaufmann, (to appear).

[14] Wei-Min Shen. Discovery as autonomous learning from the environment. *Machine Learning*, 12:143–156, 1993.

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[16] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.